

 Using App Router
Features available in /app

 Latest Version
15.5.4

Getting Started

Welcome to the Next.js documentation!

This **Getting Started** section will help you create your first Next.js app and learn the core features you'll use in every project.

Pre-requisite knowledge

Our documentation assumes some familiarity with web development. Before getting started, it'll help if you're comfortable with:

- HTML
- CSS
- JavaScript
- React

If you're new to React or need a refresher, we recommend starting with our [React Foundations course](#), and the [Next.js Foundations course](#) that has you building an application as you learn.

Next Steps

Installation

Learn how to create a new Next.js application...

Project Struct...

Learn the folder and file conventions in...

Layouts and ...

Learn how to create your first pages and layout...

Linking and ...

Learn how the built-in navigation optimizations...

Server and C...

Learn how you can use React Server and Client...

Partial Prere...

Learn how to use Partial Prerendering and...

Fetching Data

Learn how to fetch data and stream content that...

Updating Data

Learn how to mutate data using Server Functions.

Caching and ...

Learn how to cache and revalidate data in...

Error Handling

Learn how to display expected errors and handle...

CSS

Image Optim...

Learn about the different ways to add CSS to your...

Learn how to optimize images in Next.js

Font Optimiz...

Learn how to optimize fonts in Next.js

Metadata an...

Learn how to add metadata to your pages and create...

Route Handl...

Learn how to use Route Handlers and Middleware

Deploying

Learn how to deploy your Next.js application.

Upgrading

Learn how to upgrade your Next.js applicatio...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4



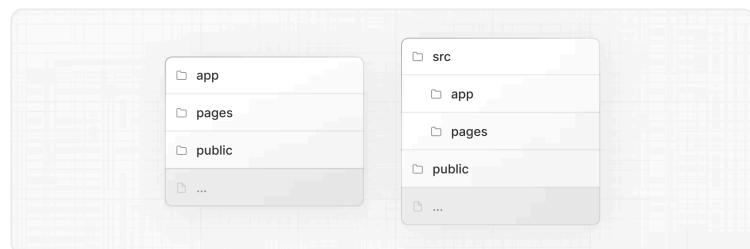
Project structure and organization

This page provides an overview of **all** the folder and file conventions in Next.js, and recommendations for organizing your project.

Folder and file conventions

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

<code>next.config.js</code>	Configuration file for Next.js
<code>package.json</code>	Project dependencies and scripts
<code>instrumentation.ts</code>	OpenTelemetry and Instrumentation file
<code>middleware.ts</code>	Next.js request middleware
<code>.env</code>	Environment variables
<code>.env.local</code>	Local environment variables
<code>.env.production</code>	Production environment variables
<code>.env.development</code>	Development environment variables
<code>.eslintrc.json</code>	Configuration file for ESLint
<code>.gitignore</code>	Git files and folders to ignore
<code>next-env.d.ts</code>	TypeScript declaration file for Next.js
<code>tsconfig.json</code>	Configuration file for TypeScript
<code>jsconfig.json</code>	Configuration file for JavaScript

Routing Files

<code>layout</code>	<code>.js</code>	<code>.jsx</code>	Layout
		<code>.tsx</code>	
<code>page</code>	<code>.js</code>	<code>.jsx</code>	Page
		<code>.tsx</code>	

`loading` .js .jsx Loading UI

.tsx

`not-found` .js .jsx Not found UI

.tsx

`error` .js .jsx Error UI

.tsx

`global-error` .js .jsx Global error UI

.tsx

`route` .js .ts API endpoint

`template` .js .jsx Re-rendered layout

.tsx

`default` .js .jsx 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

Organizing your project

Next.js is **unopinionated** about how you organize and colocate your project files. But it does provide several features to help you organize your project.

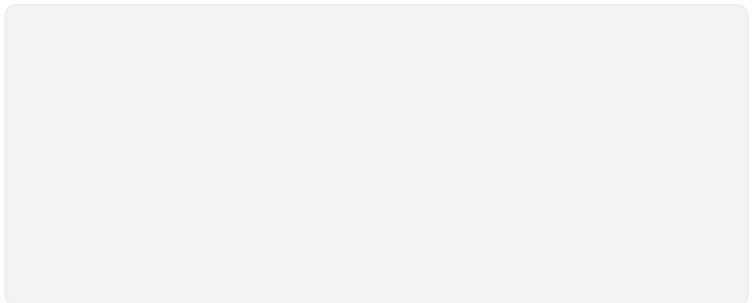
Component hierarchy

The components defined in special files are rendered in a specific hierarchy:

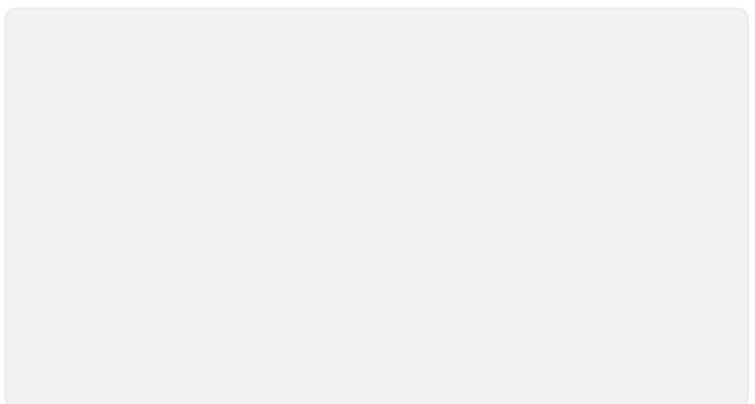
- `layout.js`

- `template.js`

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



The components are rendered recursively in nested routes, meaning the components of a route segment will be nested **inside** the components of its parent segment.



Colocation

In the `app` directory, nested folders define 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.

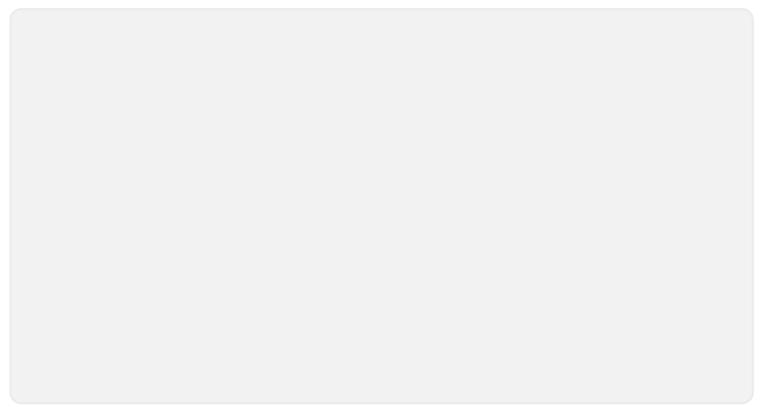
This means that **project files** can be **safely colocated** inside route segments in the `app` directory without accidentally being routable.

Good to know: 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](#).

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.

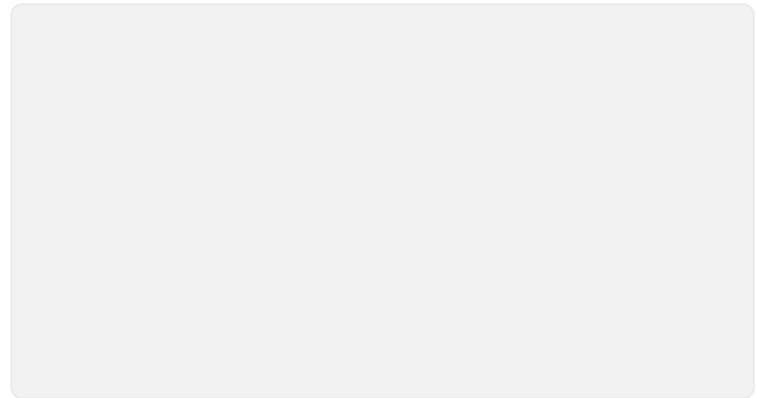
Good to know:

- 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 by site section, intent, or team. e.g. marketing pages, admin pages, etc.
- 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` folder

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

Examples

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.

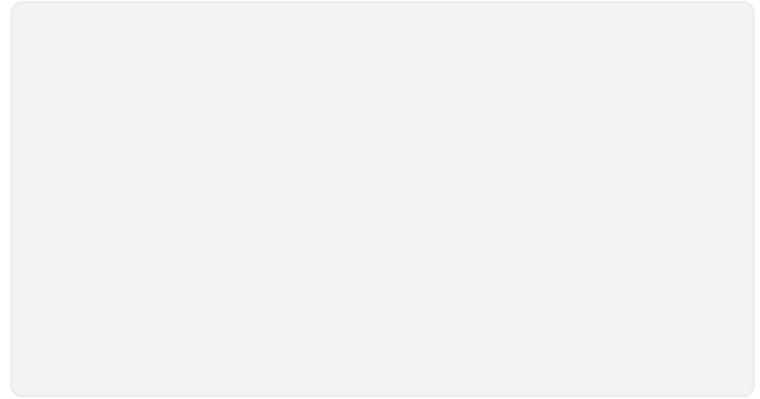
Good to know: In our examples below, we're using `components` and `lib` folders as generalized placeholders, their naming has no special framework significance and your projects might use other folders like `ui`, `utils`, `hooks`, `styles`, etc.

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.

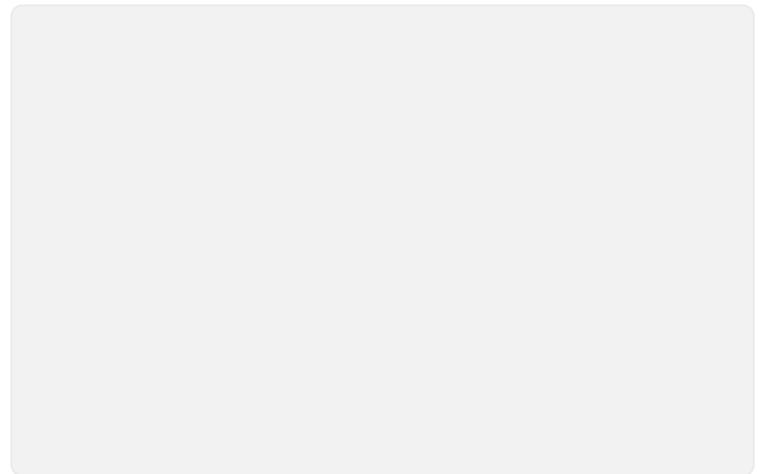
Store project files in top-level folders inside of `app`

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.



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)`).

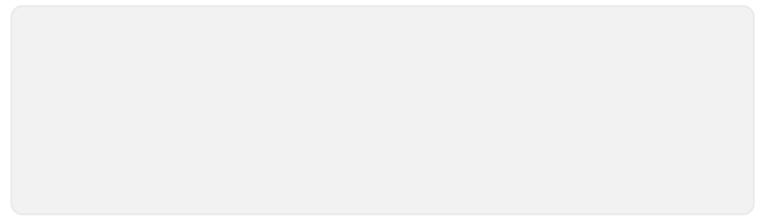
Even though routes inside `(marketing)` and `(shop)` share the same URL hierarchy, you can create a different layout for each group by adding a `layout.js` file inside their folders.

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

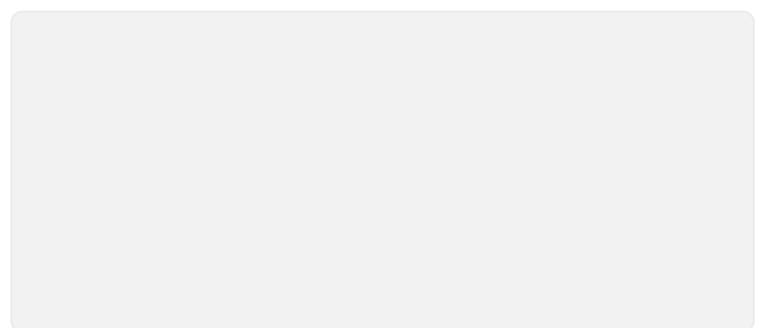
To apply a [loading skeleton](#) via a `loading.js` file to a specific route, create a new route group (e.g., `/overview`) and then move your `loading.tsx` inside that route group.



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.

 Using App Router
Features available in /app

 Latest Version
15.5.4

Layouts and Pages

Next.js uses **file-system based routing**, meaning you can use folders and files to define routes. This page will guide you through how to create layouts and pages, and link between them.

Creating a page

A **page** is UI that is rendered on a specific route. To create a page, add a `page` file inside the `app` directory and default export a React component. For example, to create an index page (`/`):



TS app/page.tsx TypeScript ▾ 

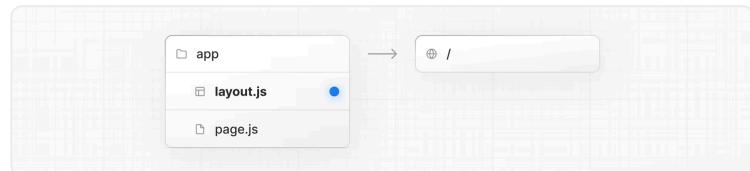
```
export default function Page() {
  return <h1>Hello Next.js!</h1>
}
```

Creating a layout

A layout is UI that is **shared** between multiple pages. On navigation, layouts preserve state, remain interactive, and do not rerender.

You can define a layout by default exporting a React component from a `layout` file. The component should accept a `children` prop which can be a page or another `layout`.

For example, to create a layout that accepts your index page as child, add a `layout` file inside the `app` directory:



The code editor shows the content of `app/layout.tsx`. The file is written in TypeScript and defines a `DashboardLayout` component. The component takes a `children` prop and returns an `html` element with a `lang="en"` attribute. Inside the `html` element, there is an `body` element containing a `main` element with the `children` prop. The code is as follows:

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

The layout above is called a `root layout` because it's defined at the root of the `app` directory. The root layout is **required** and must contain `html` and `body` tags.

Creating a nested route

A nested route is a route composed of multiple URL segments. For example, the `/blog/[slug]`

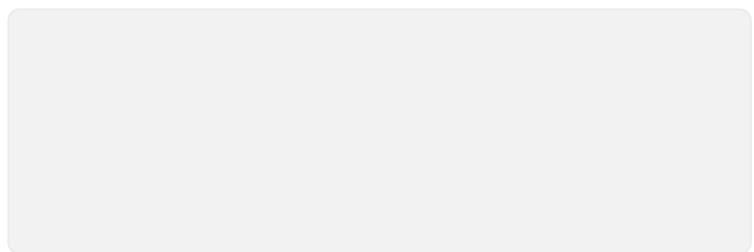
route is composed of three segments:

- / (Root Segment)
- blog (Segment)
- [slug] (Leaf Segment)

In Next.js:

- **Folders** are used to define the route segments that map to URL segments.
- **Files** (like `page` and `layout`) are used to create UI that is shown for a segment.

To create nested routes, you can nest folders inside each other. For example, to add a route for `/blog`, create a folder called `blog` in the `app` directory. Then, to make `/blog` publicly accessible, add a `page.tsx` file:

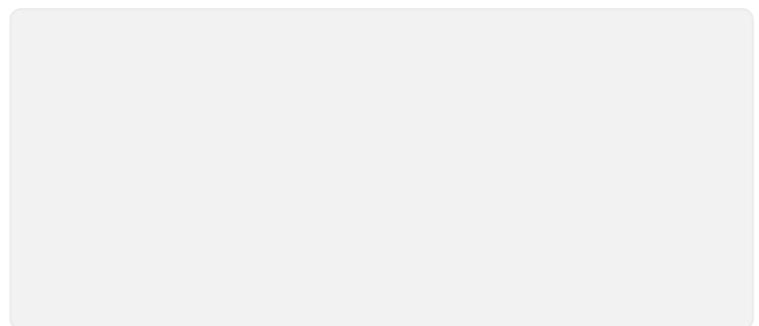


```
TS app/blog/page.tsx TypeScript ▾ ⌂
// Dummy imports
import { getPosts } from '@/lib/posts'
import { Post } from '@/ui/post'

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

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

You can continue nesting folders to create nested routes. For example, to create a route for a specific blog post, create a new `[slug]` folder inside `blog` and add a `page` file:



```
function generateStaticParams() {}

export default function Page() {
  return <h1>Hello, Blog Post Page!</h1>
}
```

Wrapping a folder name in square brackets (e.g. `[slug]`) creates a **dynamic route segment** which is used to generate multiple pages from data. e.g. blog posts, product pages, etc.

Nesting layouts

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

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

```
TS app/blog/layout.tsx
```

TypeScript ▾

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

If you were to combine the two layouts above, the root layout (`app/layout.js`) would wrap the blog layout (`app/blog/layout.js`), which would wrap the blog (`app/blog/page.js`) and blog post page (`app/blog/[slug]/page.js`).

Creating a dynamic segment

[Dynamic segments](#) allow you to create routes that are generated from data. For example, instead of manually creating a route for each individual blog post, you can create a dynamic segment to generate the routes based on blog post data.

To create a dynamic segment, wrap the segment (folder) name in square brackets: `[segmentName]`. For example, in the `app/blog/[slug]/page.tsx` route, the `[slug]` is the dynamic segment.

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

TypeScript ▾

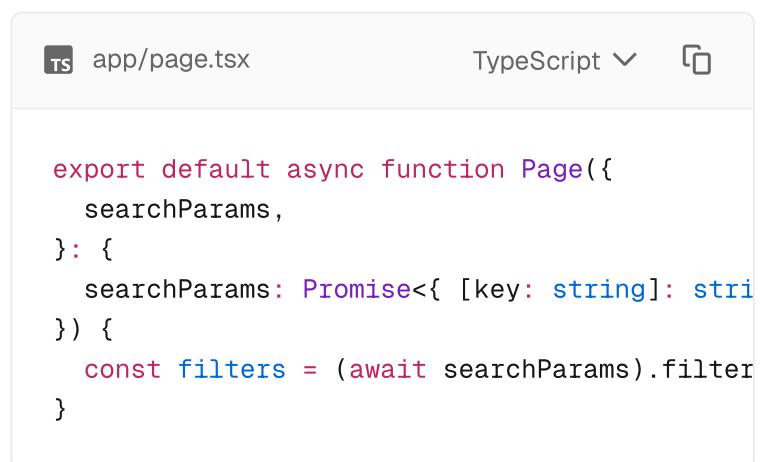
```
export default async function BlogPostPage({  
  params,  
}: {  
  params: Promise<{ slug: string }>  
) {  
  const { slug } = await params  
  const post = await getPost(slug)  
  
  return (  
    <div>  
      <h1>{post.title}</h1>  
      <p>{post.content}</p>  
    </div>  
  )  
}
```

Learn more about [Dynamic Segments](#) and the `params` props.

Nested [layouts within Dynamic Segments](#), can also access the `params` props.

Rendering with search params

In a Server Component **page**, you can access search parameters using the `searchParams` prop:



```
app/page.tsx TypeScript ▾
```

```
export default async function Page({  
  searchParams,  
}: {  
  searchParams: Promise<{ [key: string]: string }>  
) {  
  const filters = (await searchParams).filter  
}
```

Using `searchParams` opts your page into [dynamic rendering](#) because it requires an incoming request to read the search parameters from.

Client Components can read search params using the `useSearchParams` hook.

Learn more about `useSearchParams` in [statically rendered](#) and [dynamically rendered](#) routes.

What to use and when

- Use the `searchParams` prop when you need search parameters to **load data for the page** (e.g. pagination, filtering from a database).
- Use `useSearchParams` when search parameters are used **only on the client** (e.g. filtering a list already loaded via props).
- As a small optimization, you can use `new URLSearchParams(window.location.search)` in **callbacks or event handlers** to read search params without triggering re-renders.

Linking between pages

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

For example, to generate a list of blog posts, import `<Link>` from `next/link` and pass a `href` prop to the component:



```
TS app/ui/post.tsx TypeScript ▾ ⌂
import Link from 'next/link'

export default async function Post({ post })
  const posts = await getPosts()

  return (
    <ul>
```

```
{posts.map((post) => (
  <li key={post.slug}>
    <Link href={`/blog/${post.slug}`}>
      </li>
    ))
</ul>
)}
```

Good to know: `<Link>` is the primary way to navigate between routes in Next.js. You can also use the `useRouter` hook for more advanced navigation.

Route Props Helpers

Next.js exposes utility types that infer `params` and named slots from your route structure:

- **PageProps**: Props for `page` components, including `params` and `searchParams`.
- **LayoutProps**: Props for `layout` components, including `children` and any named slots (e.g. folders like `@analytics`).

These are globally available helpers, generated when running either `next dev`, `next build` or `next typegen`.

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

```
export default async function Page(props: PageProps) {
  const { slug } = await props.params
  return <h1>Blog post: {slug}</h1>
}
```

TS app/dashboard/layout.tsx

```
export default function Layout(props: LayoutProps) {
  return (
    <section>
```

```
{props.children}  
/* If you have app/dashboard/@analytic  
/* {props.analytics} */  
</section>  
)  
}
```

Good to know

- Static routes resolve `params` to `{}`.
- `PageProps`, `LayoutProps` are global helpers — no imports required.
- Types are generated during `next dev`, `next build` or `next typegen`.

API Reference

Learn more about the features mentioned in this page by reading the API Reference.

Linking and ...

Learn how the built-in navigation optimizations...

layout.js

API reference for the layout.js file.

page.js

API reference for the page.js file.

Link Compon...

Enable fast client-side navigation with the built-in...

Dynamic Seg...

Dynamic Route Segments can be used to...

Was this helpful?  



Using App Router
Features available in /app Latest Version
15.5.4

Linking and Navigating



In Next.js, routes are rendered on the server by default. This often means the client has to wait for a server response before a new route can be shown. Next.js comes with built-in [prefetching](#), [streaming](#), and [client-side transitions](#) ensuring navigation stays fast and responsive.

This guide explains how navigation works in Next.js and how you can optimize it for [dynamic routes](#) and [slow networks](#).

How navigation works

To understand how navigation works in Next.js, it helps to be familiar with the following concepts:

- [Server Rendering](#)
- [Prefetching](#)
- [Streaming](#)
- [Client-side transitions](#)

Server Rendering

In Next.js, [Layouts and Pages](#) are [React Server Components ↗](#) by default. On initial and subsequent navigations, the [Server Component](#)

[Payload](#) is generated on the server before being sent to the client.

There are two types of server rendering, based on *when* it happens:

- **Static Rendering (or Prerendering)** happens at build time or during [revalidation](#) and the result is cached.
- **Dynamic Rendering** happens at request time in response to a client request.

The trade-off of server rendering is that the client must wait for the server to respond before the new route can be shown. Next.js addresses this delay by [prefetching](#) routes the user is likely to visit and performing [client-side transitions](#).

Good to know: HTML is also generated for the initial visit.

Prefetching

Prefetching is the process of loading a route in the background before the user navigates to it. This makes navigation between routes in your application feel instant, because by the time a user clicks on a link, the data to render the next route is already available client side.

Next.js automatically prefetches routes linked with the [`<Link>` component](#) when they enter the user's viewport.



```
TS app/layout.tsx TypeScript ▾ ⌂
import Link from 'next/link'

export default function Layout({ children }: {
  return (
    <html>
```

```
<body>
  <nav>
    /* Prefetched when the link is hov
    <Link href="/blog">Blog</Link>
    /* No prefetching */
    <a href="/contact">Contact</a>
  </nav>
  {children}
  </body>
</html>
)
}
```

How much of the route is prefetched depends on whether it's static or dynamic:

- **Static Route:** the full route is prefetched.
- **Dynamic Route:** prefetching is skipped, or the route is partially prefetched if `loading.tsx` is present.

By skipping or partially prefetching dynamic routes, Next.js avoids unnecessary work on the server for routes the users may never visit. However, waiting for a server response before navigation can give the users the impression that the app is not responding.

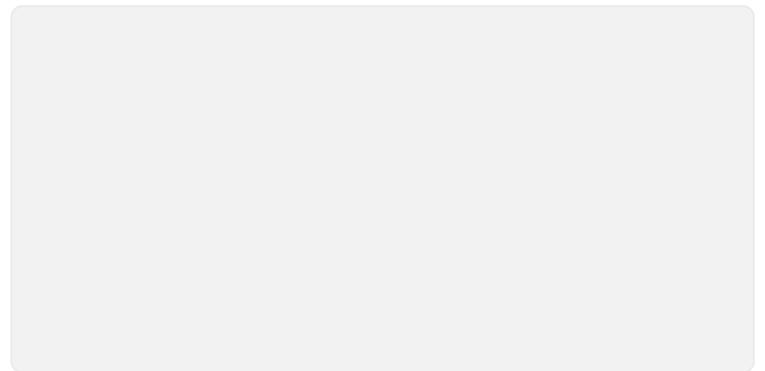
To improve the navigation experience to dynamic routes, you can use [streaming](#).

Streaming

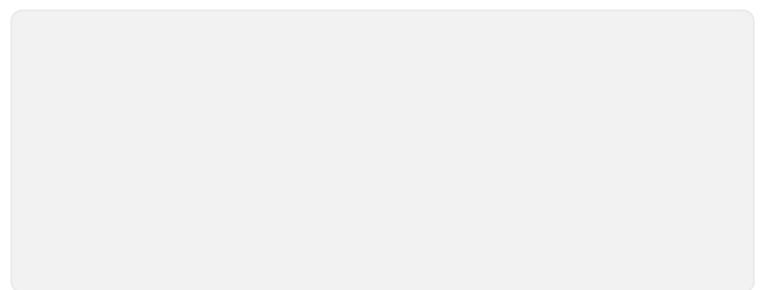
Streaming allows the server to send parts of a dynamic route to the client as soon as they're

ready, rather than waiting for the entire route to be rendered. This means users see something sooner, even if parts of the page are still loading.

For dynamic routes, it means they can be **partially prefetched**. That is, shared layouts and loading skeletons can be requested ahead of time.



To use streaming, create a `loading.tsx` in your route folder:



```
TS app/dashboard/loading.tsx TypeScript ✓ ⌂
export default function Loading() {
  // Add fallback UI that will be shown while
  return <LoadingSkeleton />
}
```

Behind the scenes, Next.js will automatically wrap the `page.tsx` contents in a `<Suspense>` boundary. The prefetched fallback UI will be shown while the route is loading, and swapped for the actual content once ready.

Good to know: You can also use [`<Suspense>`](#) to create loading UI for nested components.

Benefits of `loading.tsx`:

- Immediate navigation and visual feedback for the user.
- Shared layouts remain interactive and navigation is interruptible.
- Improved Core Web Vitals: [TTFB ↗](#), [FCP ↗](#), and [TTI ↗](#).

To further improve the navigation experience, Next.js performs a [client-side transition](#) with the `<Link>` component.

Client-side transitions

Traditionally, navigation to a server-rendered page triggers a full page load. This clears state, resets scroll position, and blocks interactivity.

Next.js avoids this with client-side transitions using the `<Link>` component. Instead of reloading the page, it updates the content dynamically by:

- Keeping any shared layouts and UI.
- Replacing the current page with the prefetched loading state or a new page if available.

Client-side transitions are what makes a server-rendered apps *feel* like client-rendered apps. And when paired with [prefetching](#) and [streaming](#), it enables fast transitions, even for dynamic routes.

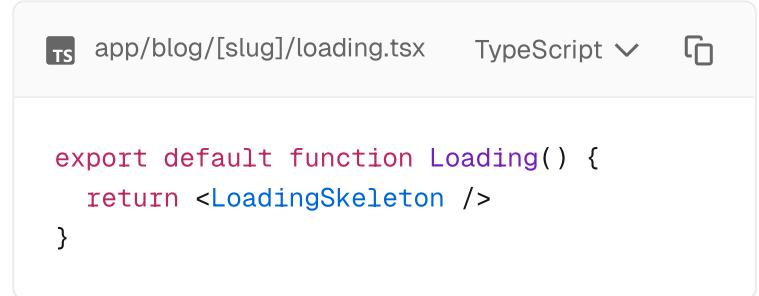
What can make transitions slow?

These Next.js optimizations make navigation fast and responsive. However, under certain conditions, transitions can still *feel* slow. Here are some common causes and how to improve the user experience:

Dynamic routes without `loading.tsx`

When navigating to a dynamic route, the client must wait for the server response before showing the result. This can give the users the impression that the app is not responding.

We recommend adding `loading.tsx` to dynamic routes to enable partial prefetching, trigger immediate navigation, and display a loading UI while the route renders.



```
TS app/blog/[slug]/loading.tsx TypeScript ▾ ⌂
export default function Loading() {
  return <LoadingSkeleton />
}
```

Good to know: In development mode, you can use the Next.js Devtools to identify if the route is static or dynamic. See [devIndicators](#) for more information.

Dynamic segments without `generateStaticParams`

If a [dynamic segment](#) could be prerendered but isn't because it's missing `generateStaticParams`, the route will fallback to dynamic rendering at request time.

Ensure the route is statically generated at build time by adding `generateStaticParams`:



```
export async function generateStaticParams()
  const posts = await fetch('https://.../post')

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

export default async function Page({
  params,
}: {
  params: Promise<{ slug: string }>
}) {
  const { slug } = await params
  // ...
}
```

Slow networks

On slow or unstable networks, prefetching may not finish before the user clicks a link. This can affect both static and dynamic routes. In these cases, the `loading.js` fallback may not appear immediately because it hasn't been prefetched yet.

To improve perceived performance, you can use the `useLinkStatus` hook to show inline visual feedback to the user (like spinners or text glimmers on the link) while a transition is in progress.



```
'use client'

import { useLinkStatus } from 'next/link'

export default function LoadingIndicator() {
  const { pending } = useLinkStatus()
  return pending ? (
    <div role="status" aria-label="Loading" c
  ) : null
}
```

You can "debounce" the loading indicator by adding an initial animation delay (e.g. 100ms) and starting the animation as invisible (e.g. `opacity: 0`). This means the loading indicator will only be shown if the navigation takes longer than the specified delay.

```
.spinner {  
  /* ... */  
  opacity: 0;  
  animation:  
    fadeIn 500ms 100ms forwards,  
    rotate 1s linear infinite;  
}  
  
@keyframes fadeIn {  
  from {  
    opacity: 0;  
  }  
  to {  
    opacity: 1;  
  }  
}  
  
@keyframes rotate {  
  to {  
    transform: rotate(360deg);  
  }  
}
```

Good to know: You can use other visual feedback patterns like a progress bar. View an example [here ↗](#).

Disabling prefetching

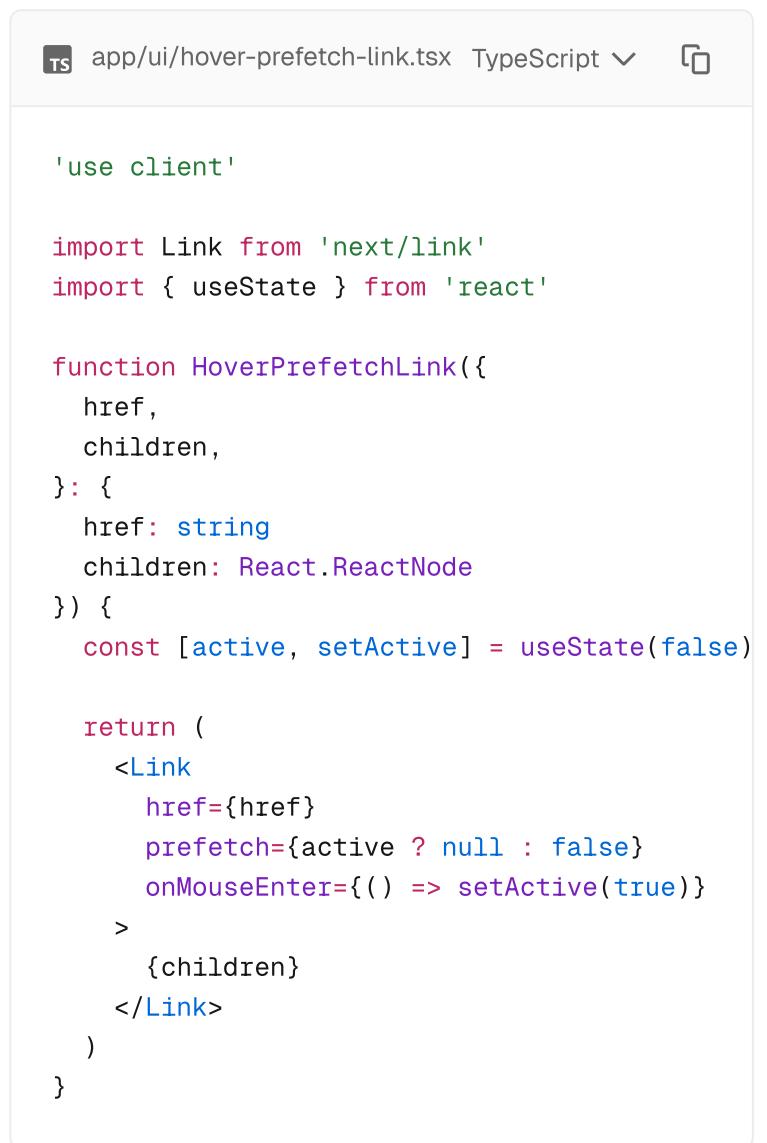
You can opt out of prefetching by setting the `prefetch` prop to `false` on the `<Link>` component. This is useful to avoid unnecessary usage of resources when rendering large lists of links (e.g. an infinite scroll table).

```
<Link prefetch={false} href="/blog">  
  Blog  
</Link>
```

However, disabling prefetching comes with trade-offs:

- **Static routes** will only be fetched when the user clicks the link.
- **Dynamic routes** will need to be rendered on the server first before the client can navigate to it.

To reduce resource usage without fully disabling prefetch, you can prefetch only on hover. This limits prefetching to routes the user is more *likely* to visit, rather than all links in the viewport.



```
TS app/ui-hover-prefetch-link.tsx TypeScript ▾ ⌂

'use client'

import Link from 'next/link'
import { useState } from 'react'

function HoverPrefetchLink({
  href,
  children,
}: {
  href: string
  children: React.ReactNode
}) {
  const [active, setActive] = useState(false)

  return (
    <Link
      href={href}
      prefetch={active ? null : false}
      onMouseEnter={() => setActive(true)}
    >
      {children}
    </Link>
  )
}
```

Hydration not completed

`<Link>` is a Client Component and must be hydrated before it can prefetch routes. On the

initial visit, large JavaScript bundles can delay hydration, preventing prefetching from starting right away.

React mitigates this with Selective Hydration and you can further improve this by:

- Using the [@next/bundle-analyzer](#) plugin to identify and reduce bundle size by removing large dependencies.
 - Moving logic from the client to the server where possible. See the [Server and Client Components](#) docs for guidance.
-

Examples

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(search
  params.set('sort', sortOrder)
  window.history.pushState(null, '', `?${pa
}

return (
  <>
    <button onClick={() => updateSorting('a
    <button onClick={() => updateSorting('d
  </>
)
}
}
```

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
      <button onClick={() => switchLocale('fr
    </>
)
}
}
```

Next Steps

Link Compon...

Enable fast client-side navigation with the built-in...

loading.js

API reference for the loading.js file.

Prefetching

Learn how to configure prefetching in...

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4



Server and Client Components

By default, layouts and pages are [Server Components ↗](#), which lets you fetch data and render parts of your UI on the server, optionally cache the result, and stream it to the client. When you need interactivity or browser APIs, you can use [Client Components ↗](#) to layer in functionality.

This page explains how Server and Client Components work in Next.js and when to use them, with examples of how to compose them together in your application.

When to use Server and Client Components?

The client and server environments have different capabilities. Server and Client components allow you to run logic in each environment depending on your use case.

Use [Client Components](#) when you need:

- [State ↗](#) and [event handlers ↗](#). E.g. `onClick`,
`onChange`.
- [Lifecycle logic ↗](#). E.g. `useEffect`.
- Browser-only APIs. E.g. `localStorage`,
`window`, `Navigator.geolocation`, etc.

- Custom hooks ↗.

Use **Server Components** when you need:

- Fetch data from databases or APIs close to the source.
- Use API keys, tokens, and other secrets without exposing them to the client.
- Reduce the amount of JavaScript sent to the browser.
- Improve the [First Contentful Paint \(FCP\)](#) ↗, and stream content progressively to the client.

For example, the `<Page>` component is a Server Component that fetches data about a post, and passes it as props to the `<LikeButton>` which handles client-side interactivity.



```
TS app/[id]/page.tsx TypeScript ▾
```

```
import LikeButton from '@/app/ui/like-button'
import {getPost} from '@/lib/data'

export default async function Page({
  params,
}: {
  params: Promise<{ id: string }>
}) {
  const { id } = await params
  const post = await getPost(id)

  return (
    <div>
      <main>
        <h1>{post.title}</h1>
        {/* ... */}
        <LikeButton likes={post.likes} />
      </main>
    </div>
  )
}
```

```
TS app/ui/like-button.tsx TypeScript ▾
```

```
'use client'

import { useState } from 'react'

export default function LikeButton({ likes }:
```

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

How do Server and Client Components work in Next.js?

On the server

On the server, Next.js uses React's APIs to orchestrate rendering. The rendering work is split into chunks, by individual route segments ([layouts and pages](#)):

- **Server Components** are rendered into a special data format called the React Server Component Payload (RSC Payload).
- **Client Components** and the RSC Payload are used to [prerender](#) HTML.

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

On the client (first load)

Then, on the client:

1. **HTML** is used to immediately show a fast non-interactive preview of the route to the user.
2. **RSC Payload** is used to reconcile the Client and Server Component trees.
3. **JavaScript** is used to hydrate Client Components and make the application interactive.

What is hydration?

Hydration is React's process for attaching [event handlers](#) to the DOM, to make the static HTML interactive.

Subsequent Navigations

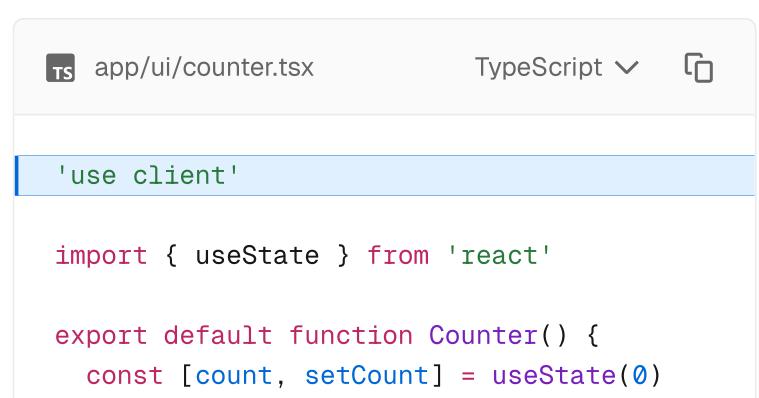
On subsequent navigations:

- The **RSC Payload** is prefetched and cached for instant navigation.
- **Client Components** are rendered entirely on the client, without the server-rendered HTML.

Examples

Using Client Components

You can create a Client Component by adding the ["use client"](#) directive at the top of the file, above your imports.



```
TS app/ui/counter.tsx TypeScript ▾ ⌂
'use client'

import { useState } from 'react'

export default function Counter() {
  const [count, setCount] = useState(0)
```

```
        return (
          <div>
            <p>{count} likes</p>
            <button onClick={() => setCount(count +
              1)}>
          )
    }
```

"use client" is used to declare a **boundary** between the Server and Client module graphs (trees).

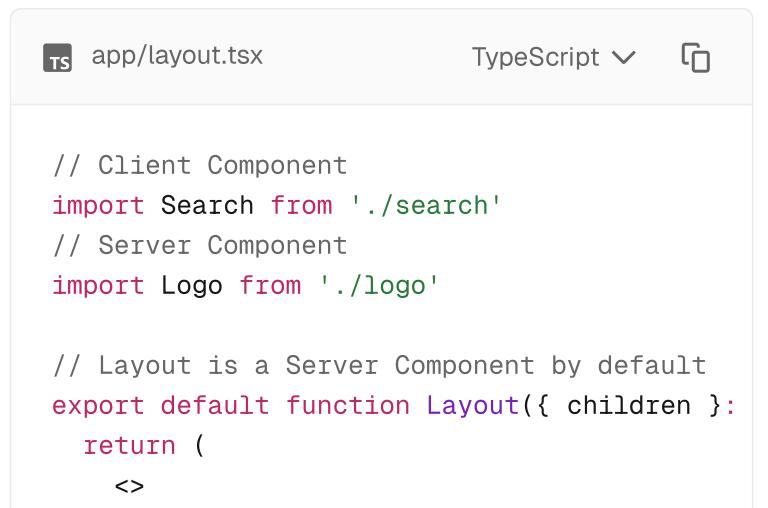
Once a file is marked with "use client", **all its imports and child components are considered part of the client bundle**. This means you don't need to add the directive to every component that is intended for the client.

Reducing JS bundle size

To reduce the size of your client JavaScript bundles, add 'use client' to specific interactive components instead of marking large parts of your UI as Client Components.

For example, the <Layout> component contains mostly static elements like a logo and navigation links, but includes an interactive search bar.

<Search /> is interactive and needs to be a Client Component, however, the rest of the layout can remain a Server Component.



The screenshot shows a code editor interface with a tab labeled "app/layout.tsx". The code is written in TypeScript and defines a "Layout" component. The "Search" component is marked with the "use client" directive, while the rest of the layout (Logo and children) is marked as a Server Component by default.

```
// Client Component
import Search from './search'
// Server Component
import Logo from './logo'

// Layout is a Server Component by default
export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <>
      <Logo />
      {children}
    </>
  )
}
```

```
<nav>
  <Logo />
  <Search />
</nav>
<main>{children}</main>
</>
)
}
```

TS app/ui/search.tsx TypeScript ▾

```
'use client'
```

```
export default function Search() {
  // ...
}
```

Passing data from Server to Client Components

You can pass data from Server Components to Client Components using props.

TS app/[id]/page.tsx TypeScript ▾

```
import LikeButton from '@/app/ui/like-button'
import { getPost } from '@/lib/data'
```

```
export default async function Page({
  params,
}: {
  params: Promise<{ id: string }>
}) {
  const { id } = await params
  const post = await getPost(id)

  return <LikeButton likes={post.likes} />
}
```

TS app/ui/like-button.tsx TypeScript ▾

```
'use client'
```

```
export default function LikeButton({ likes }: {
  // ...
})
```

```
}
```

Alternatively, you can stream data from a Server Component to a Client Component with the [use Hook ↗](#). See an [example](#).

Good to know: Props passed to Client Components need to be [serializable ↗](#) by React.

Interleaving Server and Client Components

You can pass Server Components as a prop to a Client Component. This allows you to visually nest server-rendered UI within Client components.

A common pattern is to use `children` to create a *slot* in a `<ClientComponent>`. For example, a `<Cart>` component that fetches data on the server, inside a `<Modal>` component that uses client state to toggle visibility.

```
TS app/ui/modal.tsx TypeScript ▾ ⌂
'use client'

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

Then, in a parent Server Component (e.g. `<Page>`), you can pass a `<Cart>` as the child of the `<Modal>`:

```
TS app/page.tsx TypeScript ▾ ⌂
import Modal from './ui/modal'
import Cart from './ui/cart'

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

```
<Modal>
  <Cart />
</Modal>
)
}
```

In this pattern, all Server Components will be rendered on the server ahead of time, including those as props. The resulting RSC payload will contain references of where Client Components should be rendered within the component tree.

Context providers

[React context ↗](#) is commonly used to share global state like the current theme. However, React context is not supported in Server Components.

To use context, create a Client Component that accepts `children`:

```
TS app/theme-provider.tsx TypeScript ▾ ⌂
'use client'

import { createContext } from 'react'

export const ThemeContext = createContext({})

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

Then, import it into a Server Component (e.g.

`layout`):

```
TS app/layout.tsx TypeScript ▾ ⌂
import ThemeProvider from './theme-provider'
```

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

Your Server Component will now be able to directly render your provider, and all other Client Components throughout your app will be able to consume this context.

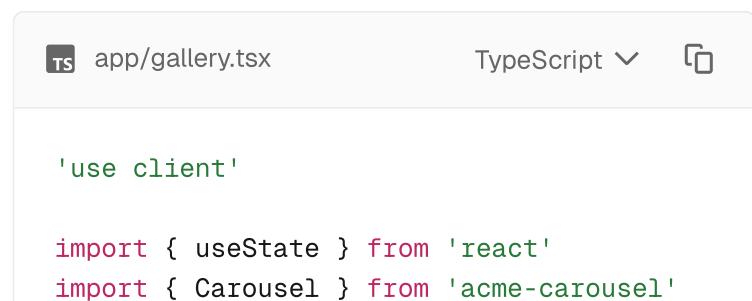
Good to know: You should render providers as deep as possible in the tree – notice how `ThemeProvider` only wraps `{children}` instead of the entire `<html>` document. This makes it easier for Next.js to optimize the static parts of your Server Components.

Third-party components

When using a third-party component that relies on client-only features, you can wrap it in a Client Component to ensure it works as expected.

For example, the `<Carousel />` can be imported from the `acme-carousel` package. 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)}>
        /* Works, since Carousel is used within
        {isOpen && <Carousel />} */
      </div>
    )
}
```

However, if you try to use it directly within a Server Component, you'll see an error. 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:

```
TS app/carousel.tsx TypeScript ▾ ⌂
'use client'

import { Carousel } from 'acme-carousel'

export default Carousel
```

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

```
TS app/page.tsx TypeScript ▾ ⌂
```

```
import Carousel from './carousel'

export default function Page() {
  return (
    <div>
      <p>View pictures</p>
      {/* Works, since Carousel is a Client
       <Carousel />
      </div>
    )
  }
}
```

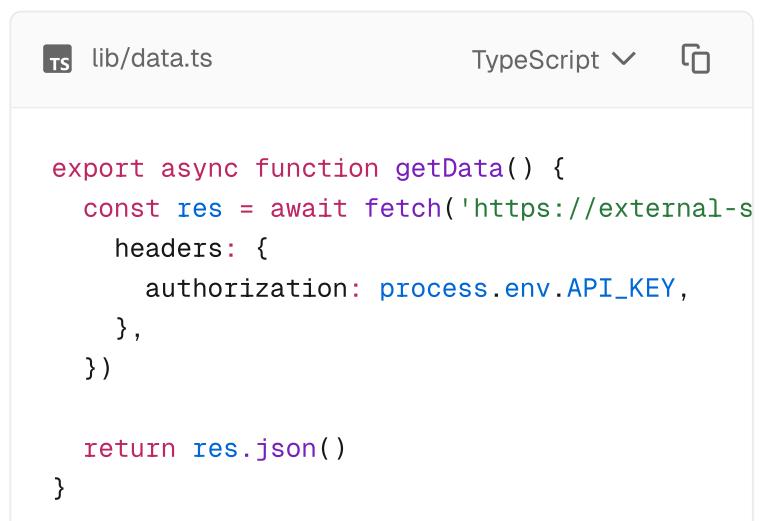
Advice for Library Authors

If you're building a component library, add the `"use client"` directive to entry points that rely on client-only features. This lets your users import components into Server Components without needing to create wrappers.

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.

Preventing environment poisoning

JavaScript modules can be shared between both Server and Client Components modules. This means it's possible to accidentally import server-only code into the client. For example, consider the following function:



The screenshot shows a code editor interface with a tab labeled "lib/data.ts". The file contains the following TypeScript code:

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

  return res.json()
}
```

This function contains an `API_KEY` that should never be exposed to the client.

In Next.js, only environment variables prefixed with `NEXT_PUBLIC_` are included in the client bundle. If variables are not prefixed, Next.js replaces them with an empty string.

As a result, even though `getData()` can be imported and executed on the client, it won't work as expected.

To prevent accidental usage in Client Components, you can use the [server-only package ↗](#).

Then, import the package into a file that contains server-only code:



```
JS lib/data.js

import 'server-only'

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

  return res.json()
}
```

Now, if you try to import the module into a Client Component, there will be a build-time error.

The corresponding [client-only package ↗](#) can be used to mark modules that contain client-only logic like code that accesses the `window` object.

In Next.js, installing `server-only` or `client-only` is **optional**. However, if your linting rules flag extraneous dependencies, you may install them to avoid issues.

pnpm

npm

yarn

bun

>_ Terminal



```
pnpm add server-only
```

Next.js handles `server-only` and `client-only` imports internally to provide clearer error messages when a module is used in the wrong environment. The contents of these packages from NPM are not used by Next.js.

Next.js also provides its own type declarations for `server-only` and `client-only`, for TypeScript configurations where

`noUncheckedSideEffectImports` ↗ is active.

Next Steps

Learn more about the APIs mentioned in this page.

use client

Learn how to use the `use client` directive to rende...

Was this helpful?



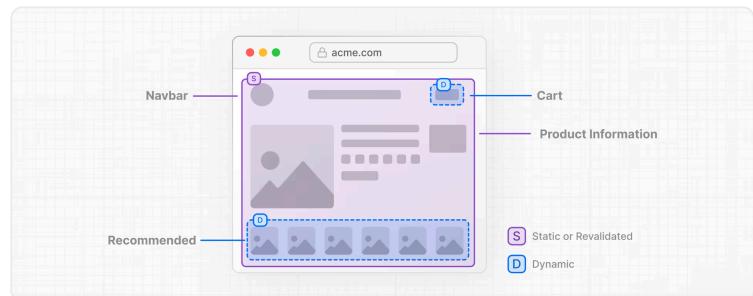
 Using App Router
Features available in /app

 Latest Version
15.5.4

Partial Prerendering

 This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

Partial Prerendering (PPR) is a rendering strategy that allows you to combine static and dynamic content in the same route. This improves the initial page performance while still supporting personalized, dynamic data.



When a user visits a route:

- The server sends a **shell** containing the static content, ensuring a fast initial load.
- The shell leaves **holes** for the dynamic content that will load in asynchronously.
- The dynamic holes are **streamed in parallel**, reducing the overall load time of the page.

 **Watch:** Why PPR and how it works → [YouTube \(10 minutes\)](#) ↗.

How does Partial Prerendering work?

To understand Partial Prerendering, it helps to be familiar with the rendering strategies available in Next.js.

Static Rendering

With Static Rendering, HTML is generated ahead of time—either at build time or through [revalidation](#). The result is cached and shared across users and requests.

In Partial Prerendering, Next.js prerenders a **static shell** for a route. This can include the layout and any other components that don't depend on request-time data.

Dynamic Rendering

With Dynamic Rendering, HTML is generated at **request time**. This allows you to serve personalized content based on request-time data.

A component becomes dynamic if it uses the following APIs:

- `cookies`
- `headers`
- `connection`
- `draftMode`
- `searchParams` prop
- `unstable_noStore`
- `fetch` with `{ cache: 'no-store' }`

In Partial Prerendering, using these APIs throws a special React error that informs Next.js the component cannot be statically rendered, causing a build error. You can use a [Suspense](#) boundary to wrap your component to defer rendering until runtime.

Suspense

React [Suspense](#) ↗ is used to defer rendering parts of your application until some condition is met.

In Partial Prerendering, Suspense is used to mark **dynamic boundaries** in your component tree.

At build time, Next.js prerenders the static content and the `fallback` UI. The dynamic content is **postponed** until the user requests the route.

Wrapping a component in Suspense doesn't make the component itself dynamic (your API usage does), but rather Suspense is used as a boundary that encapsulates dynamic content and enable [streaming](#)

```
JS app/page.js

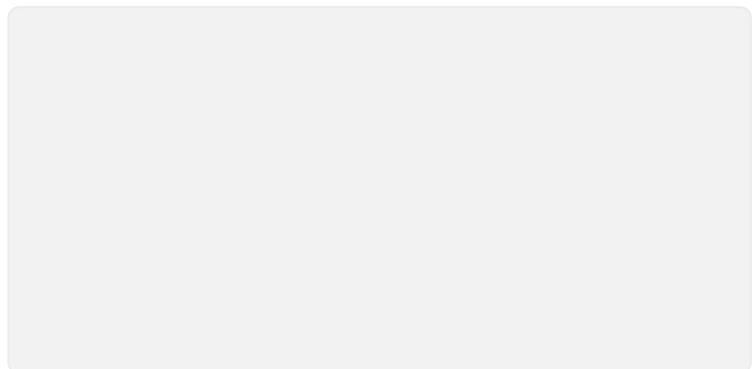
import { Suspense } from 'react'
import StaticComponent from './StaticComponent'
import DynamicComponent from './DynamicComponent'
import Fallback from './Fallback'

export const experimental_ppr = true

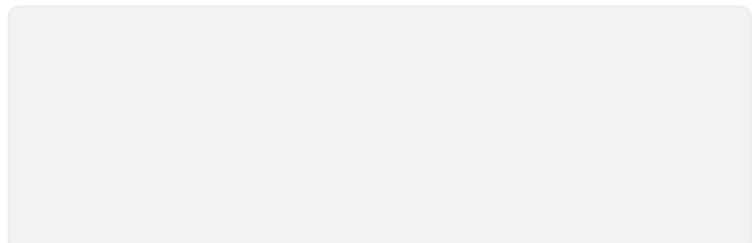
export default function Page() {
  return (
    <>
      <StaticComponent />
      <Suspense fallback={<Fallback />}>
        <DynamicComponent />
      </Suspense>
    </>
  )
}
```

Streaming

Streaming splits the route into chunks and progressively streams them to the client as they become ready. This allows the user to see parts of the page immediately, before the entire content has finished rendering.



In Partial Prerendering, dynamic components wrapped in Suspense start streaming from the server in parallel.



To reduce network overhead, the full response—including static HTML and streamed dynamic parts—is sent in a **single HTTP request**. This avoids extra roundtrips and improves both initial load and overall performance.

Enabling Partial Prerendering

You can enable PPR by adding the `ppr` option to your `next.config.ts` file:

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

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

export default nextConfig
```

The `'incremental'` value allows you to adopt PPR for specific routes:

```
TS /app/dashboard/layout.tsx TypeScript ▾ ⌂
export const experimental_ppr = true

export default function Layout({ children }: ...)
```

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.

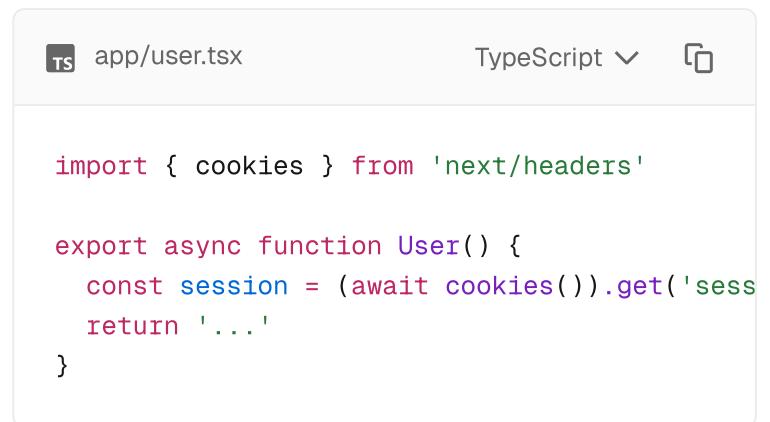
Good to know:

- `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.

Examples

Dynamic APIs

When using Dynamic APIs that require looking at the incoming request, Next.js will opt into dynamic rendering for the route. To continue using PPR, wrap the component with Suspense. For example, the `<User />` component is dynamic because it uses the `cookies` API:

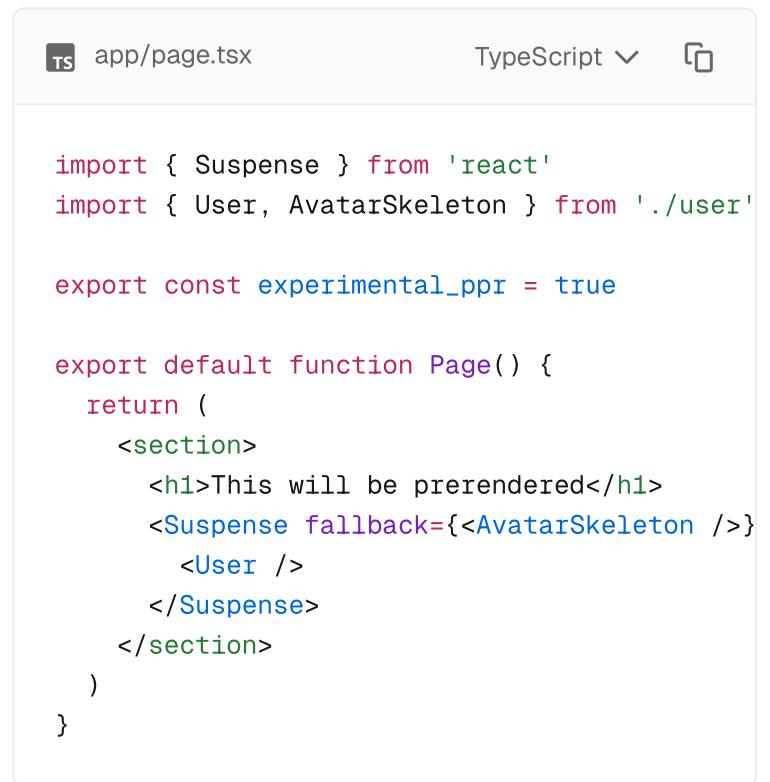


The screenshot shows a code editor window with the file name "app/user.tsx" and a TypeScript dropdown menu. The code is as follows:

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

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

The `<User />` component will be streamed while any other content inside `<Page />` will be prerendered and become part of the static shell.



The screenshot shows a code editor window with the file name "app/page.tsx" and a TypeScript dropdown menu. The code is as follows:

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

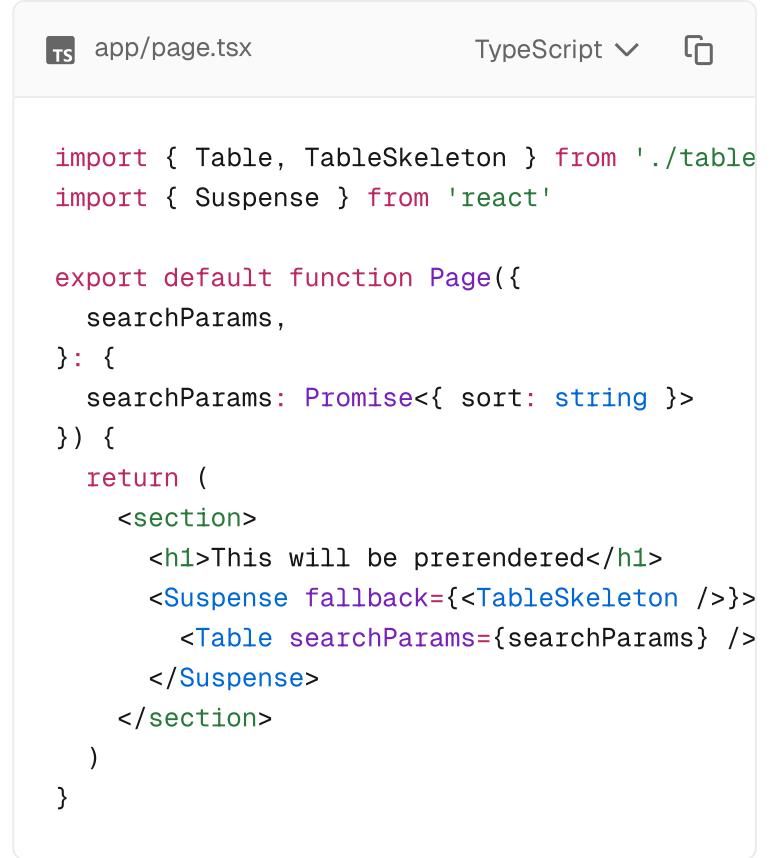
export const experimental_ppr = true

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

Passing dynamic props

Components only opt into dynamic rendering when the value is accessed. For example, if you are reading `searchParams` from a `<Page />`

component, you can forward this value to another component as a prop:



```
import { Table, TableSkeleton } from './table'
import { Suspense } from 'react'

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

Inside of the table component, accessing the value from `searchParams` will make the component dynamic while the rest of the page will be prerendered.



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

Next Steps

Learn more about the config option for Partial
Prerendering.

ppr

Learn how to
enable Partial
Prerendering in...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

Fetching Data

This page will walk you through how you can fetch data in [Server and Client Components](#), and how to [stream](#) components that depend on data.

Fetching data

Server Components

You can fetch data in Server Components using:

1. The [fetch API](#)
2. An [ORM or database](#)

With the [fetch](#) API

To fetch data with the [fetch](#) API, turn your component into an asynchronous function, and await the [fetch](#) call. For example:

TS app/blog/page.tsx TypeScript ▾ ⌂

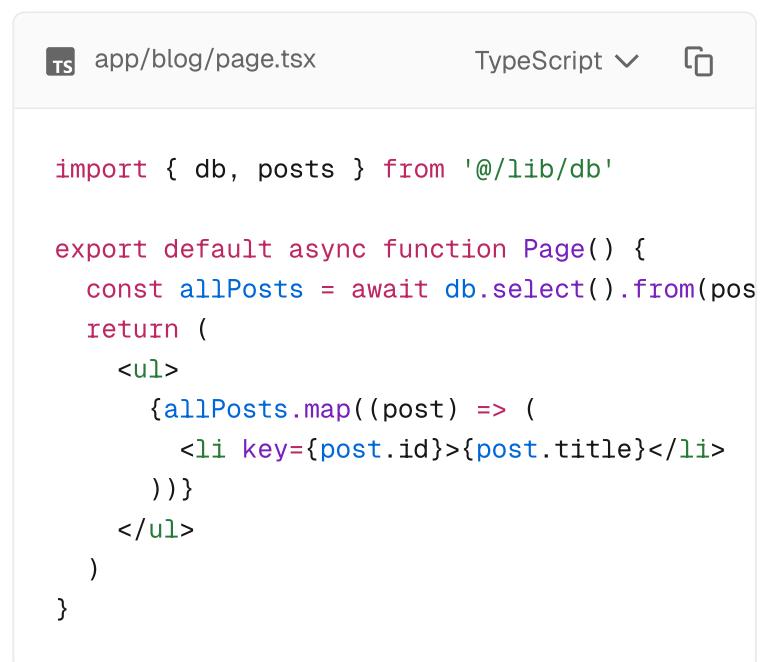
```
export default async function Page() {
  const data = await fetch('https://api.veralabs.com/posts')
  const posts = await data.json()
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      )))
    </ul>
  )
}
```

Good to know:

- `fetch` responses are not cached by default. However, Next.js will [prerender](#) the route and the output will be cached for improved performance. If you'd like to opt into [dynamic rendering](#), use the `{ cache: 'no-store' }` option. See the [fetch API Reference](#).
- During development, you can log `fetch` calls for better visibility and debugging. See the [logging API reference](#).

With an ORM or database

Since Server Components are rendered on the server, you can safely make database queries using an ORM or database client. Turn your component into an asynchronous function, and await the call:



The screenshot shows a code editor interface with a TypeScript file named `app/blog/page.tsx`. The code uses an ORM library (`@/lib/db`) to query a database and render the results as an `ul` list.

```
TS app/blog/page.tsx TypeScript ▾ ⌂

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

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

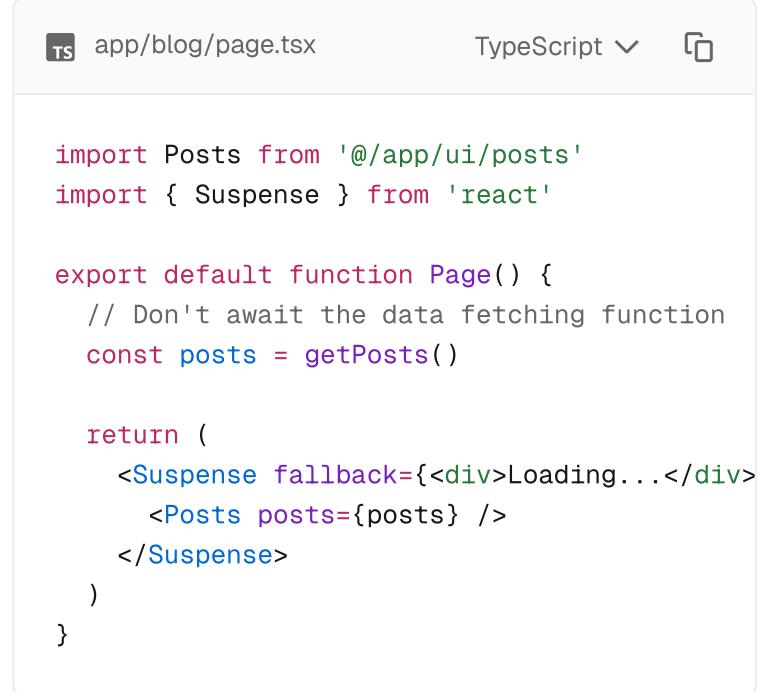
Client Components

There are two ways to fetch data in Client Components, using:

1. React's [use hook ↗](#)
2. A community library like [SWR ↗](#) or [React Query ↗](#)

[Streaming data with the use hook](#)

You can use React's `use` hook ↗ to stream data from the server to client. Start by fetching data in your Server component, and pass the promise to your Client Component as prop:



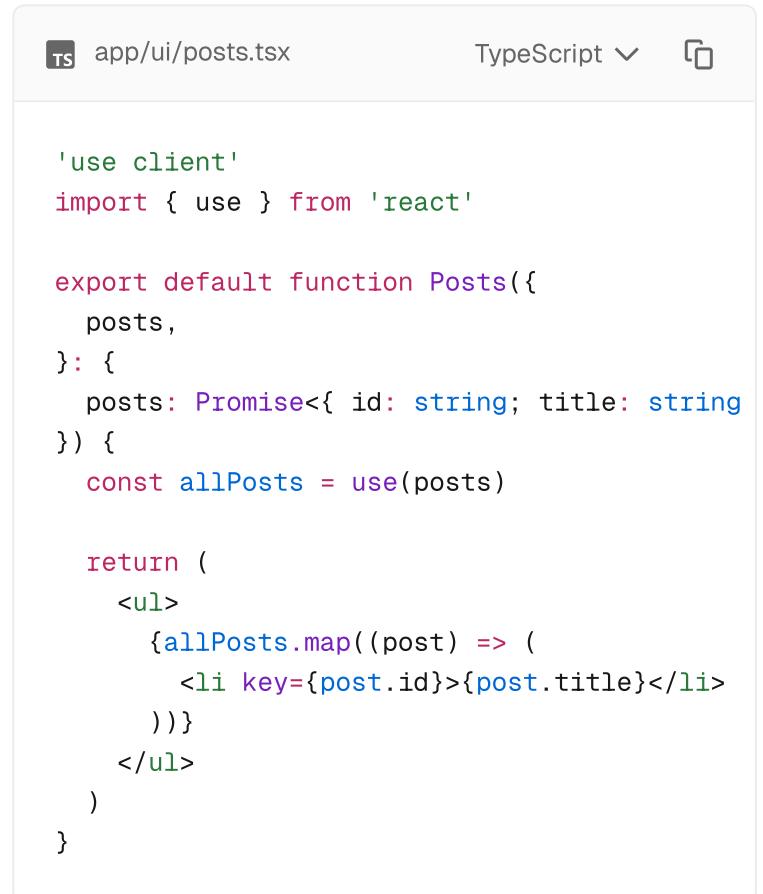
```
TS app/blog/page.tsx TypeScript ⓘ

import Posts from '@/app/ui/posts'
import { Suspense } from 'react'

export default function Page() {
  // Don't await the data fetching function
  const posts = getPosts()

  return (
    <Suspense fallback={<div>Loading...</div>}
      <Posts posts={posts} />
    </Suspense>
  )
}
```

Then, in your Client Component, use the `use` hook to read the promise:



```
'use client'
import { use } from 'react'

export default function Posts({
  posts,
}: {
  posts: Promise<{ id: string; title: string }[]
}) {
  const allPosts = use(posts)

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

In the example above, the `<Posts>` component is wrapped in a `<Suspense>` boundary ↗. This means

the fallback will be shown while the promise is being resolved. Learn more about [streaming](#).

Community libraries

You can use a community library like [SWR](#) or [React Query](#) to fetch data in Client Components. These libraries have their own semantics for caching, streaming, and other features. For example, with SWR:



```
'use client'
import useSWR from 'swr'

const fetcher = (url) => fetch(url).then((r)

export default function BlogPage() {
  const { data, error, isLoading } = useSWR(
    'https://api.vercel.app/blog',
    fetcher
  )

  if (isLoading) return <div>Loading...</div>
  if (error) return <div>Error: {error.message}</div>

  return (
    <ul>
      {data.map((post: { id: string; title: string }) =>
        <li key={post.id}>{post.title}</li>
      )}
    </ul>
  )
}
```

Deduplicate requests and cache data

One way to deduplicate `fetch` requests is with [request memoization](#). With this mechanism, `fetch` calls using `GET` or `HEAD` with the same URL and options in a single render pass are

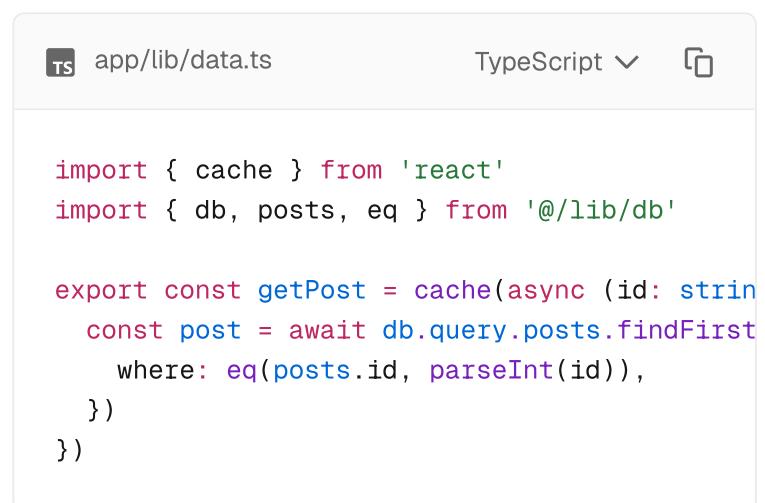
combined into one request. This happens automatically, and you can [opt out](#) by passing an Abort signal to `fetch`.

Request memoization is scoped to the lifetime of a request.

You can also deduplicate `fetch` requests by using Next.js' [Data Cache](#), for example by setting `cache: 'force-cache'` in your `fetch` options.

Data Cache allows sharing data across the current render pass and incoming requests.

If you are *not* using `fetch`, and instead using an ORM or database directly, you can wrap your data access with the [React cache ↗](#) function.



```
TS app/lib/data.ts TypeScript ▾ ⌂

import { cache } from 'react'
import { db, posts, eq } from '@/lib/db'

export const getPost = cache(async (id: string) => {
  const post = await db.query.posts.findFirst({
    where: eq(posts.id, parseInt(id)),
  })
})
```

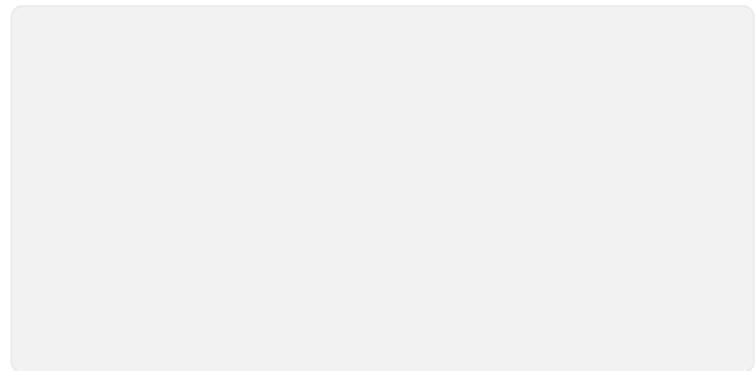
Streaming

Warning: The content below assumes the `cacheComponents` config option is enabled in your application. The flag was introduced in Next.js 15 canary.

When using `async/await` in Server Components, Next.js will opt into [dynamic rendering](#). This means the data will be fetched and rendered on the server for every user request. If there are any slow

data requests, the whole route will be blocked from rendering.

To improve the initial load time and user experience, you can use streaming to break up the page's HTML into smaller chunks and progressively send those chunks from the server to the client.

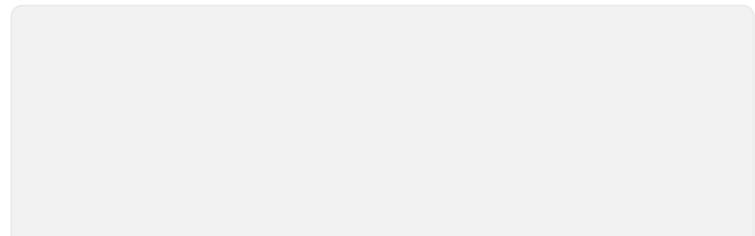


There are two ways you can implement streaming in your application:

1. Wrapping a page with a `loading.js` file
2. Wrapping a component with `<Suspense>`

With `loading.js`

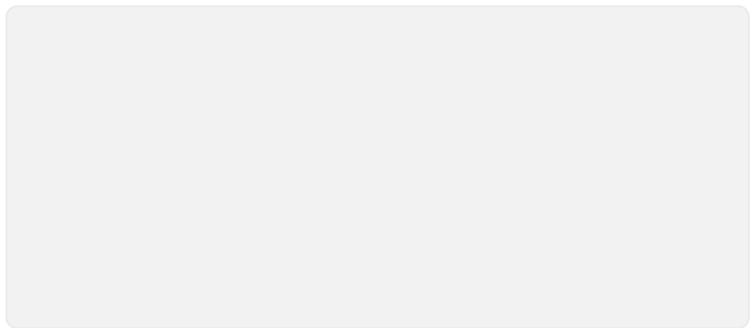
You can create a `loading.js` file in the same folder as your page to stream the **entire page** while the data is being fetched. For example, to stream `app/blog/page.js`, add the file inside the `app/blog` folder.



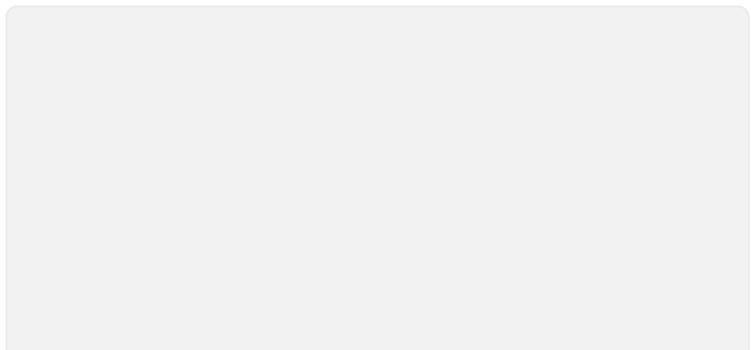
```
TS app/blog/loading.tsx TypeScript ▾ ⌂
export default function Loading() {
```

```
// Define the Loading UI here  
return <div>Loading...</div>  
}
```

On navigation, the user will immediately see the layout and a `loading state` while the page is being rendered. The new content will then be automatically swapped in once rendering is complete.



Behind-the-scenes, `loading.js` will be nested inside `layout.js`, and will automatically wrap the `page.js` file and any children below in a `<Suspense>` boundary.

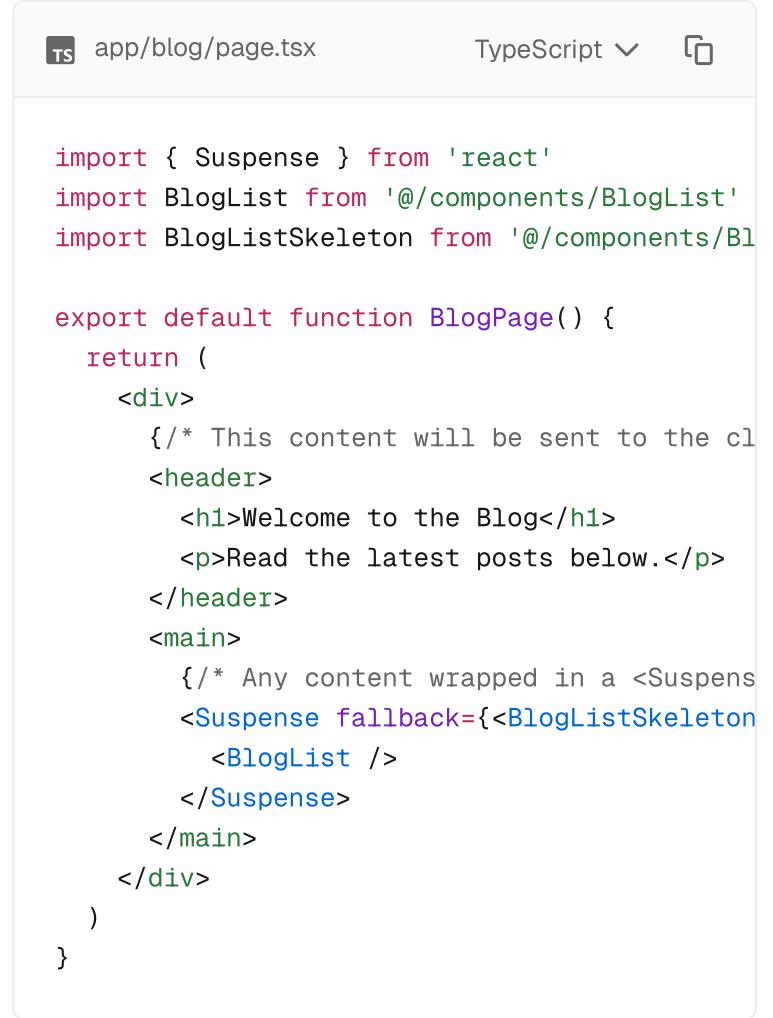


This approach works well for route segments (layouts and pages), but for more granular streaming, you can use `<Suspense>`.

With `<Suspense>`

`<Suspense>` allows you to be more granular about what parts of the page to stream. For example, you can immediately show any page content that falls

outside of the `<Suspense>` boundary, and stream in the list of blog posts inside the boundary.



```
TS app/blog/page.tsx TypeScript ▾
```

```
import { Suspense } from 'react'
import BlogList from '@/components/BlogList'
import BlogListSkeleton from '@/components/BlogListSkeleton'

export default function BlogPage() {
  return (
    <div>
      {/* This content will be sent to the client immediately */}
      <header>
        <h1>Welcome to the Blog</h1>
        <p>Read the latest posts below.</p>
      </header>
      <main>
        {/* Any content wrapped in a <Suspense> block will be
           delayed until the <Suspense fallback=> component is
           rendered. This allows you to show a meaningful loading state
           while the actual content is being fetched. */}
        <Suspense fallback={<BlogListSkeleton />}>
          <BlogList />
        </Suspense>
      </main>
    </div>
  )
}
```

Creating meaningful loading states

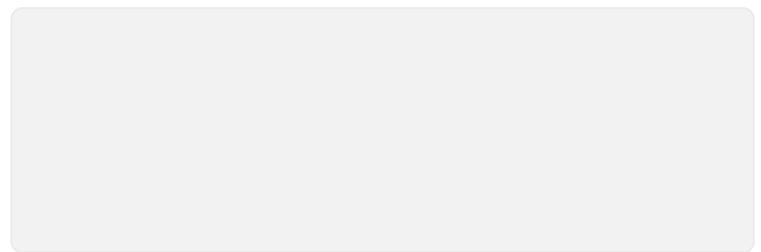
An instant loading state is fallback UI that is shown immediately to the user after navigation. For the best user experience, we recommend designing loading states that are meaningful and help users understand the app is responding. For example, you can use skeletons and spinners, or a small but meaningful part of future screens such as a cover photo, title, etc.

In development, you can preview and inspect the loading state of your components using the [React Devtools ↗](#).

Examples

Sequential data fetching

Sequential data fetching happens when nested components in a tree each fetch their own data and the requests are not [deduplicated](#), leading to longer response times.



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:

A screenshot of a code editor window. The title bar says "TS app/artist/[username]/page.... TypeScript". The code editor displays the following TypeScript code:

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

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

The code uses the `Suspense` component to handle the loading of nested `Playlists` data based on the `artistID` prop.

```
</>
)
}

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

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

To improve the user experience, you should use `React <Suspense>` to show a `fallback` while data is being fetch. This will enable `streaming` and prevent the whole route from being blocked by the sequential data requests.

Parallel data fetching

Parallel data fetching happens when data requests in a route are eagerly initiated and start at the same time.

By default, `layouts and pages` are rendered in parallel. So each segment starts fetching data as soon as possible.

However, within *any* component, multiple `async / await` requests can still be sequential if placed after the other. For example, `getAlbums` will be blocked until `getArtist` is resolved:

```
TS app/artist/[username]/page.... TypeScript ▾ ⌂

import { getArtist, getAlbums } from '@/app/1

export default async function Page({ params } // These requests will be sequential
const { username } = await params
```

```
const artist = await getArtist(username)
const albums = await getAlbums(username)
return <div>{artist.name}</div>
}
```

You can initiate requests in parallel by defining them outside the components that use the data, and resolving them together, for example, with [Promise.all](#) ↗:

TS app/artist/[username]/page.... TypeScript ▾

```
import Albums from './albums'

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

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

export default async function Page({
  params,
}: {
  params: Promise<{ username: string }>
}) {
  const { username } = await params
  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} />
    </>
  )
}
```

Good to know: If one request fails when using [Promise.all](#), the entire operation will fail. To handle this, you can use the [Promise.allSettled](#) ↗ method instead.

Preloading data

You can preload data by creating an utility function that you eagerly call above blocking requests.

`<Item>` conditionally renders based on the `checkIsAvailable()` function.

You can call `preload()` before `checkIsAvailable()` to eagerly initiate `<Item/>` data dependencies. By the time `<Item/>` is rendered, its data has already been fetched.

```
TS app/item/[id]/page.tsx TypeScript ▾
```

```
import { getItem, checkIsAvailable } from '@/'

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

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

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

Additionally, you can use React's `cache` function ↗ and the `server-only package` ↗ to create a reusable utility function. This approach allows you to cache the data fetching function and ensure that it's only executed on the server.

```
import { cache } from 'react'
import 'server-only'
import { getItem } from '@/lib/data'

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

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

API Reference

Learn more about the features mentioned in this page by reading the API Reference.

Data Security

Learn the built-in data security features in Next.j...

fetch

API reference for the extended fetch function.

loading.js

API reference for the loading.js file.

logging

Configure how data fetches are logged to the...

taint

Enable tainting Objects and Values.

Was this helpful?



Using App Router
Features available in /app Latest Version
15.5.4

Updating Data

You can update data in Next.js using React's [Server Functions](#). This page will go through how you can [create](#) and [invoke](#) Server Functions.

What are Server Functions?

A **Server Function** is an asynchronous function that runs on the server. They can be called from client through a network request, which is why they must be asynchronous.

In an `action` or mutation context, they are also called **Server Actions**.

By convention, a Server Action is an async function used with [startTransition](#). This happens automatically when the function is:

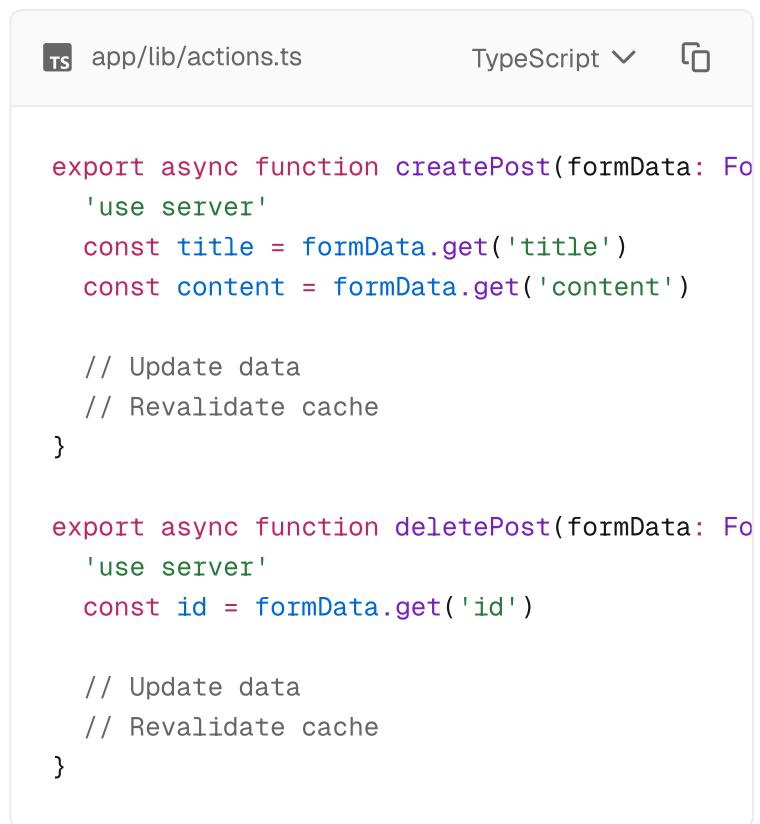
- Passed to a `<form>` using the `action` prop.
- Passed to a `<button>` using the `formAction` prop.

In Next.js, Server Actions integrate with the framework's [caching](#) 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.

Creating Server Functions

A Server Function can be defined by using the `use server` directive. You can place the directive at the top of an **asynchronous** function to mark the function as a Server Function, or at the top of a separate file to mark all exports of that file.



The screenshot shows a code editor window titled "app/lib/actions.ts" with the TypeScript logo. The code is as follows:

```
export async function createPost(formData: FormData) {
  'use server'
  const title = formData.get('title')
  const content = formData.get('content')

  // Update data
  // Revalidate cache
}

export async function deletePost(formData: FormData) {
  'use server'
  const id = formData.get('id')

  // Update data
  // Revalidate cache
}
```

Server Components

Server Functions can be inlined in Server Components by adding the `"use server"` directive to the top of the function body:



The screenshot shows a code editor window titled "app/page.tsx" with the TypeScript logo. The code is as follows:

```
export default function Page() {
  // Server Action
  async function createPost(formData: FormData) {
    'use server'
    // ...
  }

  return <></>
}
```

```
}
```

Good to know: Server Components support progressive enhancement by default, meaning forms that call Server Actions will be submitted even if JavaScript hasn't loaded yet or is disabled.

Client Components

It's not possible to define Server Functions in Client Components. However, you can invoke them in Client Components by importing them from a file that has the `"use server"` directive at the top of it:

```
TS app/actions.ts TypeScript ▾ ⌂
'use server'

export async function createPost() {}
```

```
TS app/ui/button.tsx TypeScript ▾ ⌂
'use client'

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

export function Button() {
  return <button formAction={createPost}>Create post</button>
}
```

Good to know: In Client Components, forms invoking Server Actions will queue submissions if JavaScript isn't loaded yet, and will be prioritized for hydration. After hydration, the browser does not refresh on form submission.

Passing actions as props

You can also pass an action to a Client Component as a prop:

```
<ClientComponent updateItemAction={updateItem
```

TS app/client-component.tsx TypeScript ▾

```
'use client'

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

Invoking Server Functions

There are two main ways you can invoke a Server Function:

1. [Forms](#) in Server and Client Components
2. [Event Handlers](#) and [useEffect](#) in Client Components

Good to know: Server Functions are designed for server-side mutations. The client currently dispatches and awaits them one at a time. This is an implementation detail and may change. If you need parallel data fetching, use [data fetching](#) in Server Components, or perform parallel work inside a single Server Function or [Route Handler](#).

Forms

React extends the HTML [`<form>`](#) element to allow Server Function to be invoked with the HTML `action` prop.

When invoked in a form, the function automatically receives the [`FormData`](#) object. You can extract

the data using the native [FormData](#) methods ↗:

TS

app/ui/form.tsx

TypeScript ▾

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

export function Form() {
  return (
    <form action={createPost}>
      <input type="text" name="title" />
      <input type="text" name="content" />
      <button type="submit">Create</button>
    </form>
  )
}
```

TS

app/actions.ts

TypeScript ▾

```
'use server'

export async function createPost(formData: FormData) {
  const title = formData.get('title')
  const content = formData.get('content')

  // Update data
  // Revalidate cache
}
```

Event Handlers

You can invoke a Server Function in a Client Component by using event handlers such as `onClick`.

TS

app/like-button.tsx

TypeScript ▾

```
'use client'

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

export default function LikeButton({ initialL
  const [likes, setLikes] = useState(initialL

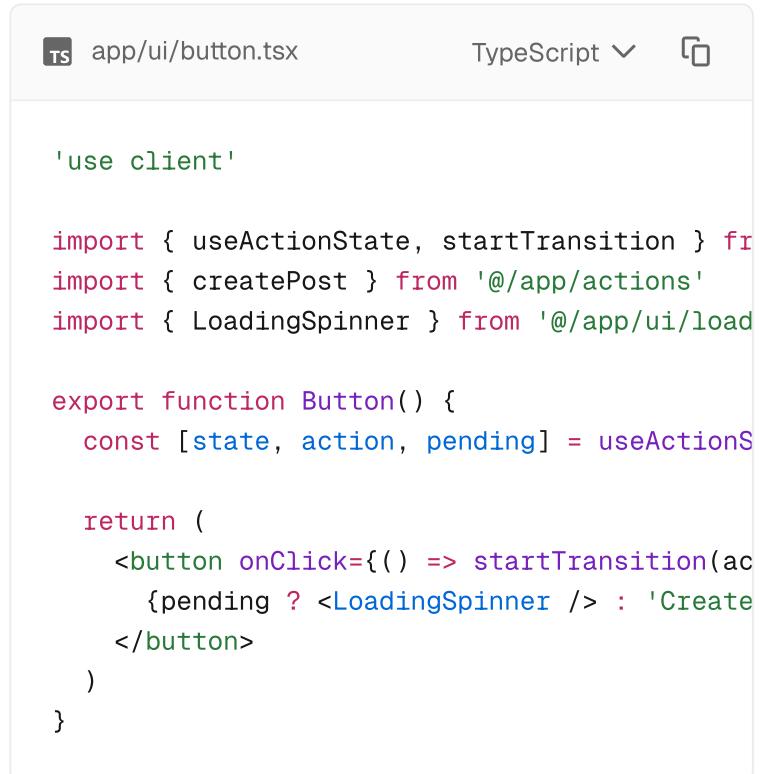
  return (
    <>
```

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

Examples

Showing a pending state

While executing a Server Function, you can show a loading indicator with React's `useActionState` ↗ hook. This hook returns a `pending` boolean:



```
TS app/ui/button.tsx TypeScript ▾
```

```
'use client'

import { useActionState, startTransition } from 'react'
import { createPost } from '@/app/actions'
import { LoadingSpinner } from '@/app/ui/load

export function Button() {
  const [state, action, pending] = useActionS

  return (
    <button onClick={() => startTransition(ac
      {pending ? <LoadingSpinner /> : 'Create
    </button>
  )
}
```

Revalidating

After performing an update, you can revalidate the Next.js cache and show the updated data by

calling `revalidatePath` or `revalidateTag` within the Server Function:



The screenshot shows a code editor window with the file name "app/lib/actions.ts" at the top left. At the top right, there are "TypeScript" dropdown settings and a refresh icon. The code itself is as follows:

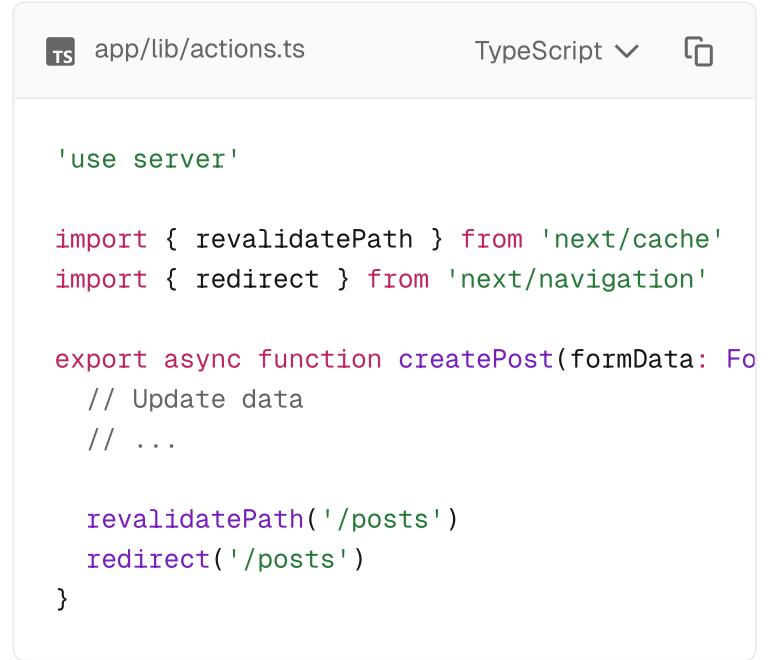
```
import { revalidatePath } from 'next/cache'

export async function createPost(formData: Fo
  'use server'
  // Update data
  // ...

  revalidatePath('/posts')
}
```

Redirecting

You may want to redirect the user to a different page after performing an update. You can do this by calling `redirect` within the Server Function.



The screenshot shows a code editor window with the file name "app/lib/actions.ts" at the top left. At the top right, there are "TypeScript" dropdown settings and a refresh icon. The code is as follows:

```
'use server'

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

export async function createPost(formData: Fo
  // Update data
  // ...

  revalidatePath('/posts')
  redirect('/posts')
}
```

Calling `redirect` throws a framework handled control-flow exception. Any code after it won't execute. If you need fresh data, call `revalidatePath` or `revalidateTag` beforehand.

Cookies

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

When you [set or delete](#) a cookie in a Server Action, Next.js re-renders the current page and its layouts on the server so the **UI reflects the new cookie value**.

Good to know: The server update applies to the current React tree, re-rendering, mounting, or unmounting components, as needed. Client state is preserved for re-rendered components, and effects re-run if their dependencies changed.

TS

app/actions.ts

TypeScript ▾

```
'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')
}
```

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, useTransition }

export default function ViewCount({ initialVi
  const [views, setViews] = useState(initialV
  const [isPending, startTransition] = useTra

  useEffect(() => {
    startTransition(async () => {
      const updatedViews = await incrementVie
      setViews(updatedViews)
    })
  }, [])

// You can use `isPending` to give users fe
return <p>Total Views: {views}</p>
}
```

API Reference

Learn more about the features mentioned in this page by reading the API Reference.

revalidatePath

API Reference for the revalidatePath function.

revalidateTag

API Reference for the revalidateTag function.

redirect

API Reference for the redirect function.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4



Caching and Revalidating

Caching is a technique for storing the result of data fetching and other computations so that future requests for the same data can be served faster, without doing the work again. While revalidation allows you to update cache entries without having to rebuild your entire application.

Next.js provides a few APIs to handle caching and revalidation. This guide will walk you through when and how to use them.

- `fetch`
- `unstable_cache`
- `revalidatePath`
- `revalidateTag`

fetch

By default, `fetch` requests are not cached. You can cache individual requests by setting the `cache` option to `'force-cache'`.

 app/page.tsx

TypeScript ▾



```
export default async function Page() {  
  const data = await fetch('https://...', {  
    }  
  )
```

Good to know: Although `fetch` requests are not cached by default, Next.js will [prerender](#) routes that have `fetch` requests and cache the HTML. If you want to guarantee a route is [dynamic](#), use the [connection API](#).

To revalidate the data returned by a `fetch` request, you can use the `next.revalidate` option.

app/page.tsx

TypeScript

```
export default async function Page() {
  const data = await fetch('https://...', { n
}
```

This will revalidate the data after a specified amount of seconds.

See the [fetch API reference](#) to learn more.

unstable_cache

`unstable_cache` allows you to cache the result of database queries and other async functions. To use it, wrap `unstable_cache` around the function. For example:

```
import { db } from '@/lib/db'
export async function getUserById(id: string)
  return db
    .select()
    .from(users)
    .where(eq(users.id, id))
    .then((res) => res[0])
}
```

app/page.tsx

TypeScript

```
import { unstable_cache } from 'next/cache'
```

```
import { getUserById } from '@/app/lib/data'
```

```
export default async function Page({  
    params,  
}: {  
    params: Promise<{ userId: string }>  
) {  
    const { userId } = await params  
  
    const getCachedUser = unstable_cache(  
        async () => {  
            return getUserById(userId)  
        },  
        [userId] // add the user ID to the cache  
    )  
}
```

The function accepts a third optional object to define how the cache should be revalidated. It accepts:

- `tags` : an array of tags used by Next.js to revalidate the cache.
- `revalidate` : the number of seconds after cache should be revalidated.

app/page.tsx TypeScript ▾

```
const getCachedUser = unstable_cache(  
    async () => {  
        return getUserById(userId)  
    },  
    [userId],  
    {  
        tags: ['user'],  
        revalidate: 3600,  
    }  
)
```

See the [unstable_cache API reference](#) to learn more.

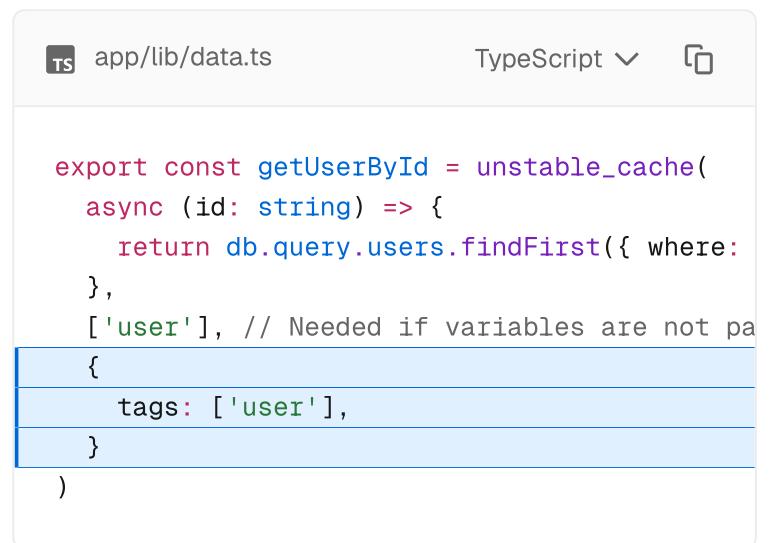
revalidateTag

`revalidateTag` is used to revalidate cache entries based on a tag and following an event. To use it with `fetch`, start by tagging the function with the `next.tags` option:



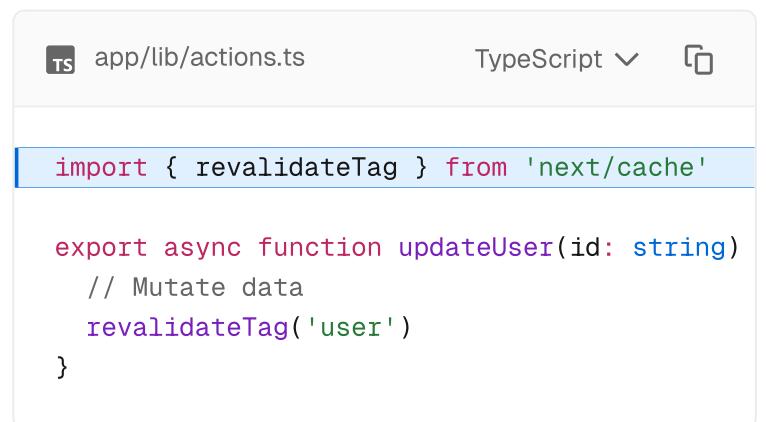
```
TS app/lib/data.ts TypeScript ⓘ
export async function getUserById(id: string)
  const data = await fetch(`https://...`, {
    next: {
      tags: ['user'],
    },
  })
}
```

Alternatively, you can mark an `unstable_cache` function with the `tags` option:



```
TS app/lib/data.ts TypeScript ⓘ
export const getUserById = unstable_cache(
  async (id: string) => {
    return db.query.users.findFirst({ where:
      {},
      ['user'], // Needed if variables are not passed
    },
    {
      tags: ['user'],
    },
  )
)
```

Then, call `revalidateTag` in a [Route Handler](#) or [Server Action](#):



```
TS app/lib/actions.ts TypeScript ⓘ
import { revalidateTag } from 'next/cache'

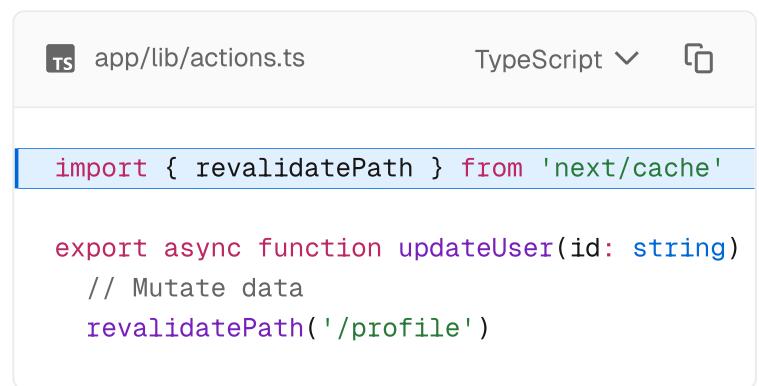
export async function updateUser(id: string)
  // Mutate data
  revalidateTag('user')
}
```

You can reuse the same tag in multiple functions to revalidate them all at once.

See the [revalidateTag API reference](#) to learn more.

revalidatePath

`revalidatePath` is used to revalidate a route and following an event. To use it, call it in a [Route Handler](#) or Server Action:



```
TS app/lib/actions.ts TypeScript ▾ ⌂
import { revalidatePath } from 'next/cache'

export async function updateUser(id: string)
  // Mutate data
  revalidatePath('/profile')
```

See the [revalidatePath API reference](#) to learn more.

API Reference

Learn more about the features mentioned in this page by reading the API Reference.

fetch

API reference for the extended fetch function.

unstable_ca...

API Reference for the unstable_cache...

revalidatePath

API Reference for the revalidatePath function.

revalidateTag

API Reference for the revalidateTag function.

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

Error Handling

Errors can be divided into two categories:

[expected errors](#) and [uncaught exceptions](#). This page will walk you through how you can handle these errors in your Next.js application.

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.

Server Functions

You can use the [useActionState](#) hook to handle expected errors in [Server Functions](#).

For these errors, avoid using `try / catch` blocks and throw errors. Instead, model expected errors as return values.

TS app/actions.ts TypeScript

```
'use server'

export async function createPost(prevState: a
  const title = formData.get('title')
  const content = formData.get('content')

  const res = await fetch('https://api.vercel
```

```
        method: 'POST',
        body: { title, content },
    })
const json = await res.json()

if (!res.ok) {
    return { message: 'Failed to create post' }
}
}
```

You can pass your action to the `useActionState` hook and use the returned `state` to display an error message.

```
TS app/ui/form.tsx TypeScript ▾
```

```
'use client'

import { useActionState } from 'react'
import { createPost } from '@/app/actions'

const initialState = {
    message: '',
}

export function Form() {
    const [state, formAction, pending] = useAct

    return (
        <form action={formAction}>
            <label htmlFor="title">Title</label>
            <input type="text" id="title" name="tit
            <label htmlFor="content">Content</label>
            <textarea id="content" name="content" r
                {state?.message && <p aria-live="polite
                    <button disabled={pending}>Create Post<
                </form>
            )
        }
    }
```

Server Components

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

```
TS app/page.tsx TypeScript ▾
```

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

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

  return '...'
}
```

Not found

You can call the `notFound` function within a route segment and use the `not-found.js` file to show a 404 UI.

```
TS app/blog/[slug]/page.tsx TypeScript ▾ ⌂

import {getPostBySlug} from '@/lib/posts'

export default async function Page({ params }) {
  const { slug } = await params
  const post = getPostBySlug(slug)

  if (!post) {
    notFound()
  }

  return <div>{post.title}</div>
}
```

```
TS app/blog/[slug]/not-found.tsx TypeScript ▾ ⌂
```

```
export default function NotFound() {
  return <div>404 - Page Not Found</div>
}
```

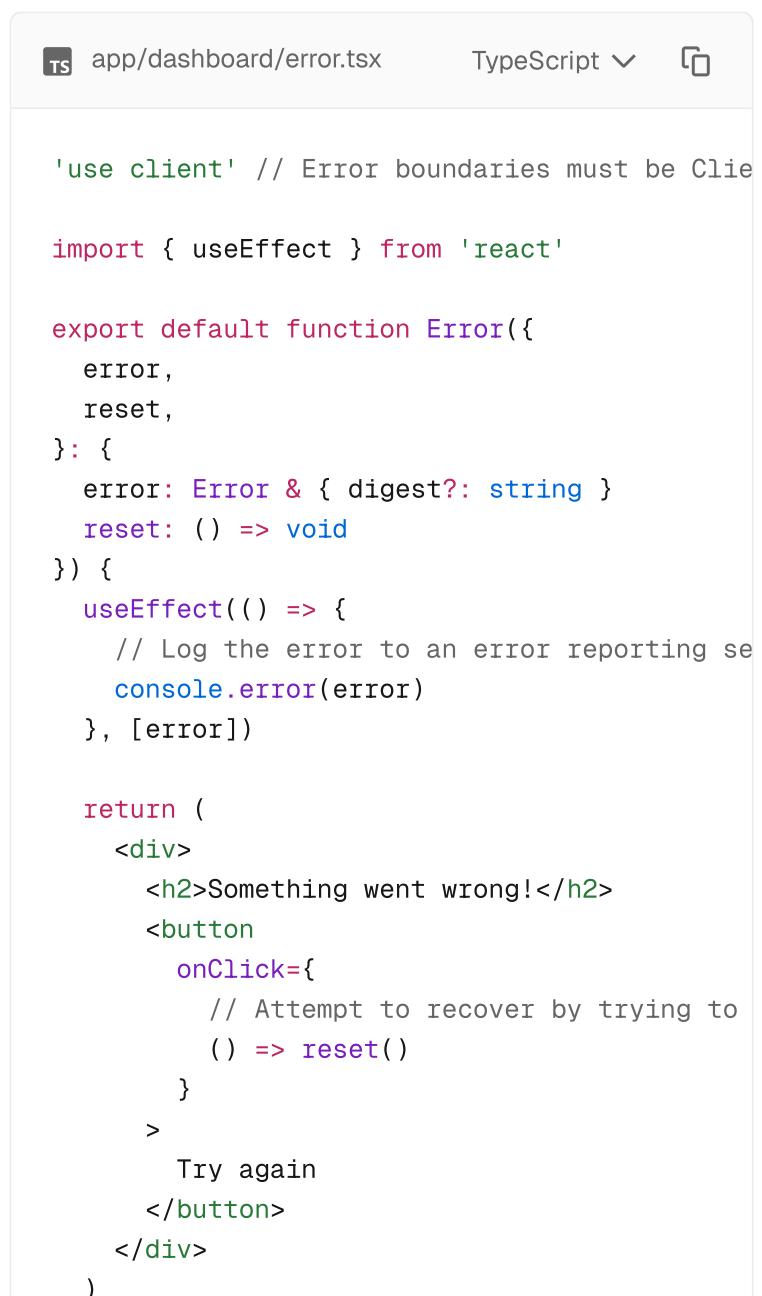
Handling uncaught exceptions

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.

Nested 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.js` file inside a route segment and exporting a React component:



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

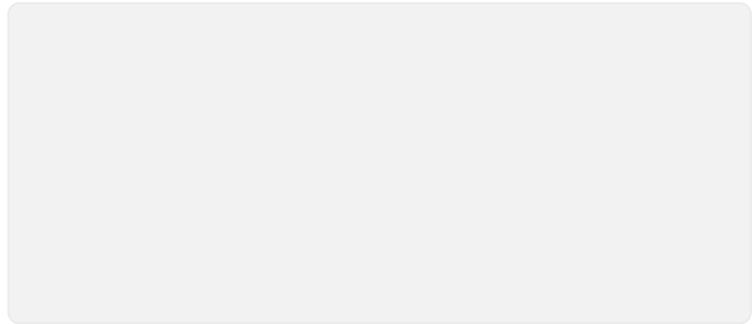
import { useEffect } from 'react'

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
        () => reset()
      }
    >
      Try again
    </button>
  </div>
)
```

```
}
```

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](#).



Error boundaries don't catch errors inside event handlers. They're designed to catch errors [during rendering](#) ↗ to show a **fallback UI** instead of crashing the whole app.

In general, errors in event handlers or async code aren't handled by error boundaries because they run after rendering.

To handle these cases, catch the error manually and store it using `useState` or `useReducer`, then update the UI to inform the user.

```
'use client'

import { useState } from 'react'

export function Button() {
  const [error, setError] = useState(null)

  const handleClick = () => {
    try {
      // do some work that might fail
      throw new Error('Exception')
    } catch (reason) {
      setError(reason)
    }
  }

  if (error) {
```

```
    /* render fallback UI */
}

return (
  <button type="button" onClick={handleClick}
    Click me
  </button>
)
}
```

Note that unhandled errors inside `startTransition` from `useTransition`, will bubble up to the nearest error boundary.

```
'use client'

import { useTransition } from 'react'

export function Button() {
  const [pending, startTransition] = useTrans

  const handleClick = () =>
    startTransition(() => {
      throw new Error('Exception')
    })

  return (
    <button type="button" onClick={handleClick}
      Click me
    </button>
  )
}
```

Global errors

While less common, you can handle errors in the root layout using the `global-error.js` file, 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
    <html>
      <body>
        <h2>Something went wrong!</h2>
        <button onClick={() => reset()}>Try again</button>
      </body>
    </html>
  )
}
```

API Reference

Learn more about the features mentioned in this page by reading the API Reference.

redirect

API Reference for the redirect function.

error.js

API reference for the error.js special file.

notFound

API Reference for the notFound function.

not-found.js

API reference for the not-found.js file.

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

CSS

Next.js provides several ways to style your application using CSS, including:

- [Tailwind CSS](#)
- [CSS Modules](#)
- [Global CSS](#)
- [External Stylesheets](#)
- [Sass](#)
- [CSS-in-JS](#)

Tailwind CSS

[Tailwind CSS ↗](#) is a utility-first CSS framework that provides low-level utility classes to build custom designs.

Install Tailwind CSS:

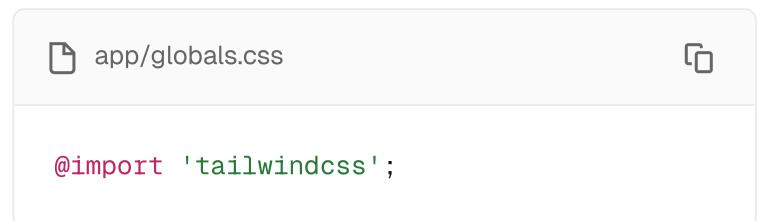
```
pnpm npm yarn bun  
>_ Terminal   
pnpm add -D tailwindcss @tailwindcss/postcss
```

Add the PostCSS plugin to your `postcss.config.mjs` file:

```
js postcss.config.mjs 
```

```
export default {
  plugins: [
    '@tailwindcss/postcss': {},
  ],
}
```

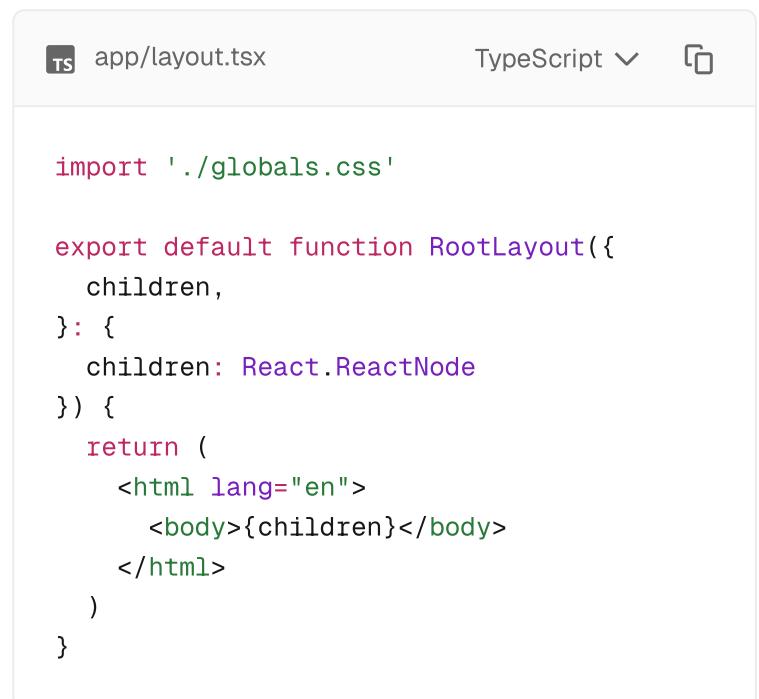
Import Tailwind in your global CSS file:



```
app/globals.css
```

```
@import 'tailwindcss';
```

Import the CSS file in your root layout:



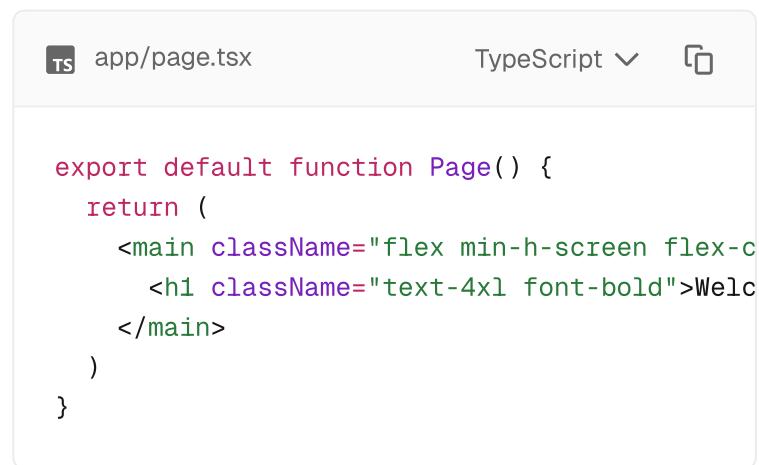
```
app/layout.tsx
```

```
TypeScript
```

```
import './globals.css'

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

Now you can start using Tailwind's utility classes in your application:



```
app/page.tsx
```

```
TypeScript
```

```
export default function Page() {
  return (
    <main className="flex min-h-screen flex-c
      <h1 className="text-4xl font-bold">Welc
    </main>
  )
}
```

Good to know: If you need broader browser support for very old browsers, see the [Tailwind CSS v3 setup instructions](#).

CSS Modules

CSS Modules locally scope CSS by generating unique class names. This allows you to use the same class in different files without worrying about naming collisions.

To start using CSS Modules, create a new file with the extension `.module.css` and import it into any component inside the `app` directory:

app/blog/blog.module.css

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

app/blog/page.tsx

TypeScript ▾

```
import styles from './blog.module.css'  
  
export default function Page() {  
  return <main className={styles.blog}></main>  
}
```

Global CSS

You can use global CSS to apply styles across your application.

Create a `app/global.css` file and import it in the root layout to apply the styles to **every route** in

your application:

app/global.css



```
body {  
  padding: 20px 20px 60px;  
  max-width: 680px;  
  margin: 0 auto;  
}
```

app/layout.tsx

TypeScript ▾



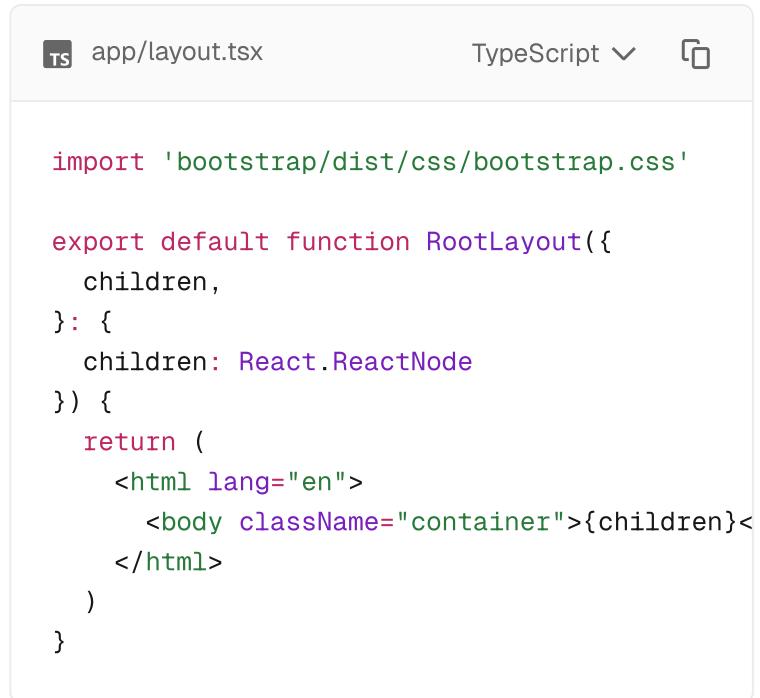
```
// These styles apply to every route in the app  
import './global.css'  
  
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
}) {  
  return (  
    <html lang="en">  
      <body>{children}</body>  
    </html>  
  )  
}
```

Good to know: Global styles can be imported into any layout, page, or component inside the `app` directory. However, since Next.js uses React's built-in support for stylesheets to integrate with Suspense, this currently does not remove stylesheets as you navigate between routes which can lead to conflicts. We recommend using global styles for *truly* global CSS (like Tailwind's base styles), [Tailwind CSS](#) for component styling, and [CSS Modules](#) for custom scoped CSS when needed.

External stylesheets

Stylesheets published by external packages can be imported anywhere in the `app` directory,

including colocated components:



The screenshot shows a code editor window with the following details:

- File name: app/layout.tsx
- TypeScript icon: TS
- TypeScript dropdown: TypeScript ▾
- Copy icon: ⌂

The code itself is a simple React component named RootLayout that wraps its children in an HTML container:import 'bootstrap/dist/css/bootstrap.css'

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

Good to know: In React 19,

`<link rel="stylesheet" href="..." />` can also be used. See the [React link documentation ↗](#) for more information.

Ordering and Merging

Next.js optimizes CSS during production builds by automatically chunking (merging) stylesheets. The **order of your CSS** depends on the **order you import styles in your code**.

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



The screenshot shows a code editor window with the following details:

- File name: page.tsx
- TypeScript icon: TS
- TypeScript dropdown: TypeScript ▾
- Copy icon: ⌂

The code imports `BaseButton` from `./base-button` and `styles` from `./page.module.css`, and defines a `Page` component:import { BaseButton } from './base-button'
import styles from './page.module.css'

export default function Page() {

```
    return <BaseButton className={styles.primary} />
```

base-button.tsx

TypeScript

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

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

Recommendations

To keep CSS ordering predictable:

- Try to contain CSS imports to a single JavaScript or TypeScript entry file
- Import global styles and Tailwind stylesheets in the root of your application.
- **Use Tailwind CSS** for most styling needs as it covers common design patterns with utility classes.
- Use CSS Modules for component-specific styles when Tailwind utilities aren't sufficient.
- Use a consistent naming convention for your CSS modules. For example, using `<name>.module.css` over `<name>.tsx`.
- Extract shared styles into shared components to avoid duplicate imports.
- Turn off linters or formatters that auto-sort imports like ESLint's `sort-imports` ↗.
- You can use the `cssChunking` option in `next.config.js` to control how CSS is chunked.

- In development (`next dev`), CSS updates apply instantly with [Fast Refresh](#).
 - In production (`next build`), all CSS files are automatically concatenated into **many minified and code-split** `.css` files, ensuring the minimal amount of CSS is loaded for a route.
 - CSS still loads with JavaScript disabled in production, but JavaScript is required in development for Fast Refresh.
 - CSS ordering can behave differently in development, always ensure to check the build (`next build`) to verify the final CSS order.
-

Next Steps

Learn more about the alternatives ways you can use CSS in your application.

Tailwind CSS...

Style your Next.js Application using Tailwind CSS v3...

Sass

Style your Next.js application using Sass.

CSS-in-JS

Use CSS-in-JS libraries with Next.js

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

Image Optimization

The Next.js `<Image>` component extends the HTML `` element to provide:

- **Size optimization:** Automatically serving correctly sized images for each device, using modern image formats like WebP.
- **Visual stability:** Preventing [layout shift ↗](#) automatically when images are loading.
- **Faster page loads:** Only loading images when they enter the viewport using native browser lazy loading, with optional blur-up placeholders.
- **Asset flexibility:** Resizing images on-demand, even images stored on remote servers.

To start using `<Image>`, import it from `next/image` and render it within your component.

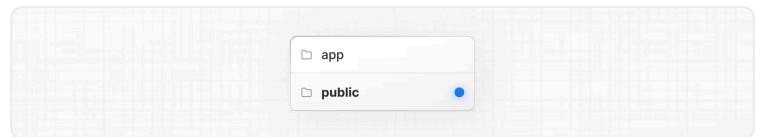
```
TS app/page.tsx TypeScript ▾ ⌂  
  
import Image from 'next/image'  
  
export default function Page() {  
  return <Image src="" alt="" />  
}
```

The `src` property can be a [local](#) or [remote](#) image.

 **Watch:** Learn more about how to use `next/image` → [YouTube \(9 minutes\) ↗](#).

Local images

You can store static files, like images and fonts, under a folder called `public` in the root directory. Files inside `public` can then be referenced by your code starting from the base URL (`/`).

A screenshot of a code editor showing a TypeScript file named 'app/page.tsx'. The code defines a component named 'Page' that returns an `<Image>` element. The `src` prop is set to the string '/profile.png'. The `alt` prop is set to the string 'Picture of the author'. The `width` and `height` props are both set to the number 500.

```
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="/profile.png"
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

If the image is statically imported, Next.js will automatically determine the intrinsic `width` and `height`. These values are used to determine the image ratio and prevent [Cumulative Layout Shift ↗](#) while your image is loading.

A screenshot of a code editor showing a TypeScript file named 'app/page.tsx'. The code imports the `Image` component from 'next/image' and a file named 'ProfileImage' from './profile.png'. It then defines a component named 'Page' that returns an `<Image>` element. The `src` prop is set to the variable `ProfileImage`. The `alt` prop is set to the string 'Picture of the author'. A note at the bottom indicates that the `width={500}` prop is automatically provided by Next.js.

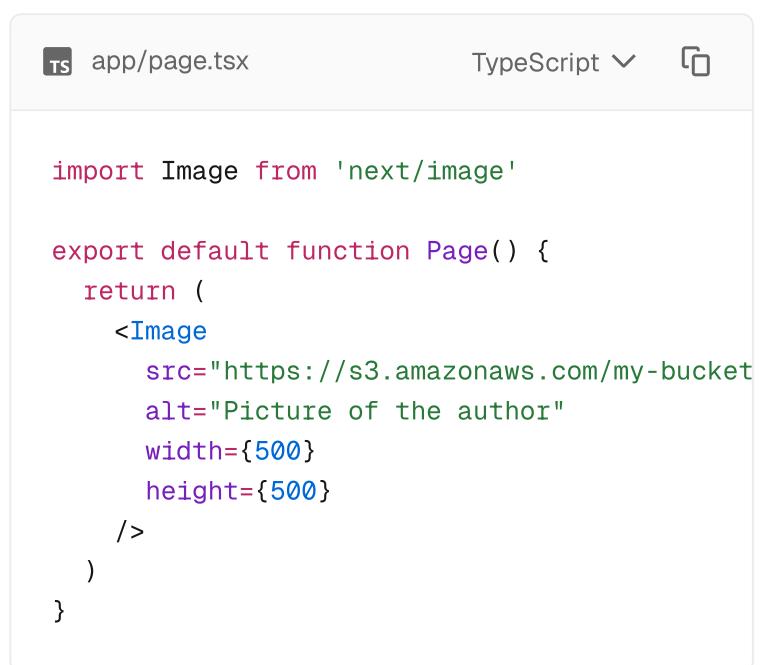
```
import Image from 'next/image'
import ProfileImage from './profile.png'

export default function Page() {
  return (
    <Image
      src={ProfileImage}
      alt="Picture of the author"
      // width={500} automatically provided
    />
  )
}
```

```
// height={500} automatically provided  
// blurDataURL="data:..." automatically  
// placeholder="blur" // Optional blur  
/>  
)  
}
```

Remote images

To use a remote image, you can provide a URL string for the `src` property.



```
TS app/page.tsx TypeScript ▾
```

```
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="https://s3.amazonaws.com/my-bucket
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

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` are used to infer the correct aspect ratio of image and avoid layout shift from the image loading in. Alternatively, you can use the `fill` property to make the image fill the size of the parent element.

To safely allow images from remote servers, you need to 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:

```
next.config.ts  TypeScript ▾
```

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

const config: NextConfig = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 's3.amazonaws.com',
        port: '',
        pathname: '/my-bucket/**',
        search: ''
      },
    ],
  },
}

export default config
```

API Reference

See the API Reference for the full feature set of Next.js Image.

Image Comp...

Optimize Images
in your Next.js
Application using...

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

Font Optimization

The `next/font` module automatically optimizes your fonts and removes external network requests for improved privacy and performance.

It includes **built-in self-hosting** for any font file. This means you can optimally load web fonts with no layout shift.

To start using `next/font`, import it from `next/font/local` or `next/font/google`, call it as a function with the appropriate options, and set the `className` of the element you want to apply the font to. For example:

```
TS app/layout.tsx TypeScript ▾ ⌂

import { Geist } from 'next/font/google'

const geist = Geist({
  subsets: ['latin'],
})

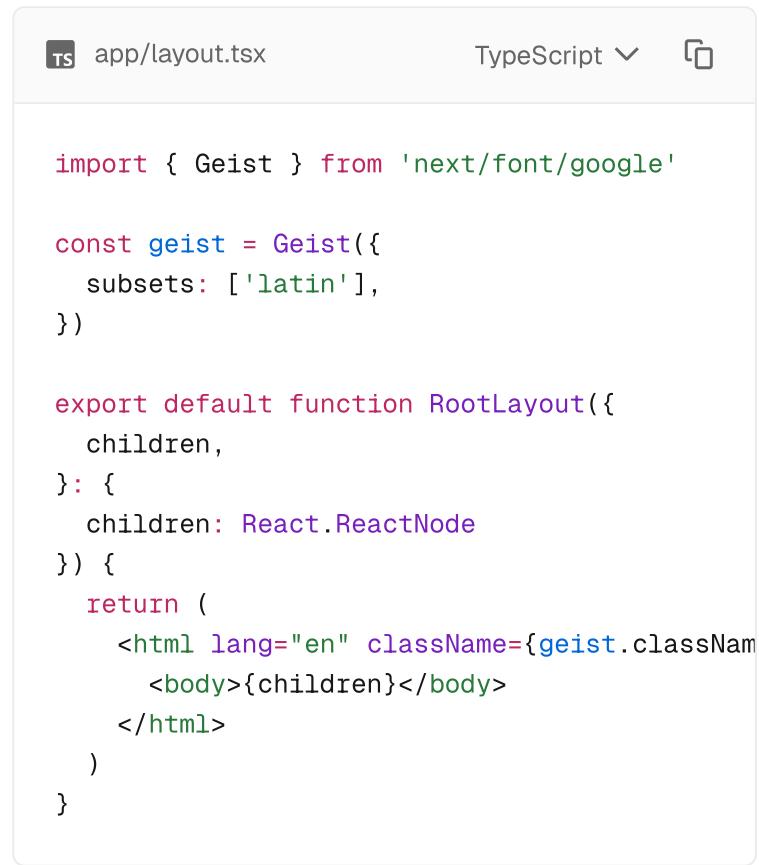
export default function Layout({ children }: {
  return (
    <html lang="en" className={geist.className}>
      <body>{children}</body>
    </html>
  )
})
```

Fonts are scoped to the component they're used in. To apply a font to your entire application, add it to the [Root Layout](#).

Google fonts

You can automatically self-host any Google Font. Fonts are included stored as static assets and served from the same domain as your deployment, meaning no requests are sent to Google by the browser when the user visits your site.

To start using a Google Font, import your chosen font from `next/font/google`:



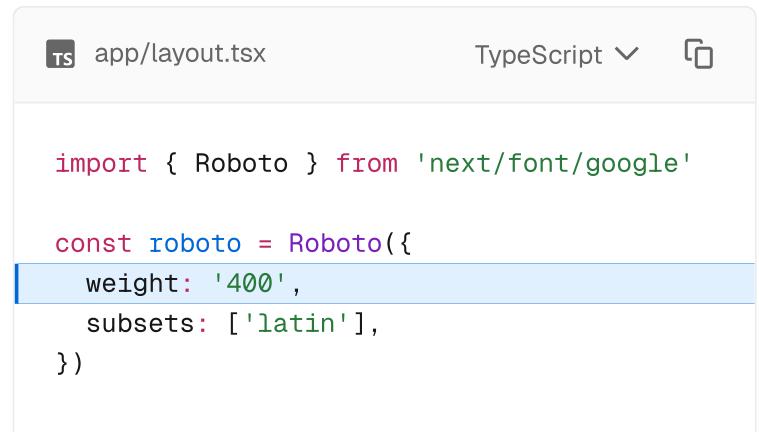
```
TS app/layout.tsx TypeScript ▾
```

```
import { Geist } from 'next/font/google'

const geist = Geist({
  subsets: ['latin'],
})

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

We recommend using [variable fonts ↗](#) for the best performance and flexibility. But if you can't use a variable font, you will need to specify a weight:



```
TS app/layout.tsx TypeScript ▾
```

```
import { Roboto } from 'next/font/google'

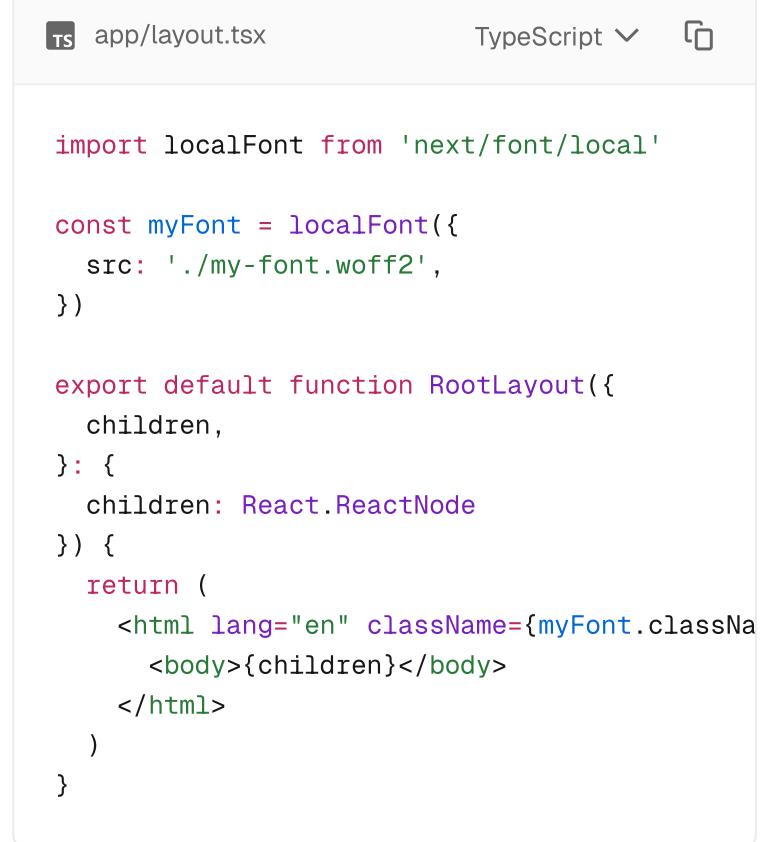
const roboto = Roboto({
  weight: '400',
  subsets: ['latin'],
})
```

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

Local fonts

To use a local font, import your font from

`next/font/local` and specify the `src` of your local font file. Fonts can be stored in the `public` folder or co-located inside the `app` folder. For example:



```
TS app/layout.tsx TypeScript ▾ ⌂  
  
import localFont from 'next/font/local'  
  
const myFont = localFont({  
  src: './my-font.woff2',  
})  
  
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
) {  
  return (  
    <html lang="en" className={myFont.className}  
      <body>{children}</body>  
    </html>  
  )  
}
```

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',
    },
  ],
})
```

API Reference

See the API Reference for the full feature set of Next.js Font

Font

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

Was this helpful?     

 Using App Router
Features available in /app Latest Version
15.5.4

Metadata and OG images

The Metadata APIs can be used to define your application metadata for improved SEO and web shareability and include:

1. The static `metadata` object
2. The dynamic `generateMetadata` function
3. Special `file conventions` that can be used to add static or dynamically generated `favicons` and `OG images`.

With all the options above, Next.js will automatically generate the relevant `<head>` tags for your page, which can be inspected in the browser's developer tools.

The `metadata` object and `generateMetadata` function exports are only supported in Server Components.

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-w
```

The other metadata fields can be defined with the `Metadata` object (for [static metadata](#)) or the `generateMetadata` function (for [generated metadata](#)).

Static metadata

To define static metadata, export a `Metadata object` from a static `layout.js` or `page.js` file. For example, to add a title and description to the blog route:

A screenshot of a code editor window. The status bar at the top shows 'TS' and 'app/blog/layout.tsx'. To the right of the status bar are 'TypeScript' and a dropdown arrow. Below the status bar is a toolbar with a copy icon. The main area contains the following TypeScript code:

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

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

export default function Layout() {}
```

You can view a full list of available options, in the `generateMetadata` documentation.

Generated metadata

You can use `generateMetadata` function to `fetch` metadata that depends on data. For

example, to fetch the title and description for a specific blog post:



The screenshot shows a code editor window with the following details:

- File path: app/blog/[slug]/page.tsx
- Language: TypeScript
- Code content (highlighted in pink):

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

type Props = {
  params: Promise<{ slug: string }>
  searchParams: Promise<{ [key: string]: string }>
}

export async function generateMetadata({
  params,
  searchParams,
  parent: ResolvingMetadata
}: Promise<Metadata>) {
  const slug = (await params).slug

  // fetch post information
  const post = await fetch(`https://api.veralabs.com/posts/${slug}`)
  const res = await post.json()
  const { title, description } = res

  return {
    title,
    description,
  }
}

export default function Page({ params, searchParams }) {
  return (
    <div>
      <h1>{params.slug}</h1>
      <p>{params.description}</p>
    </div>
  )
}
```

Streaming metadata

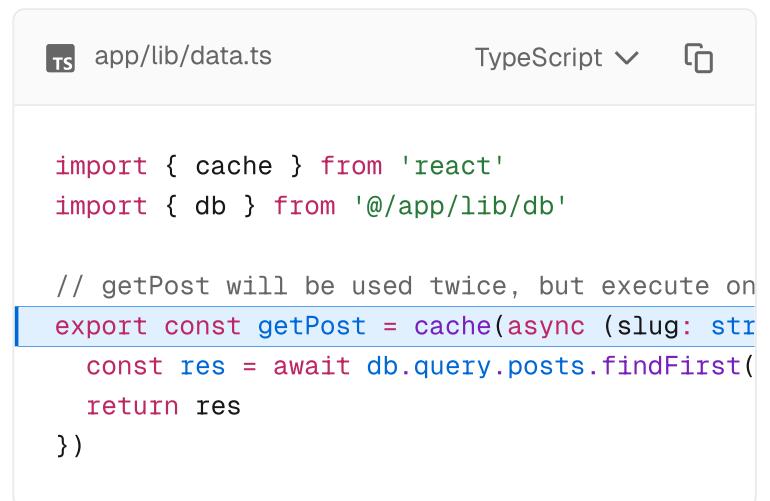
For dynamically rendered pages, if resolving `generateMetadata` might block rendering, Next.js streams the resolved metadata separately and injects it into the HTML as soon as it's ready.

Statically rendered pages don't use this behavior since metadata is resolved at build time.

Learn more about [streaming metadata](#).

Memoizing data requests

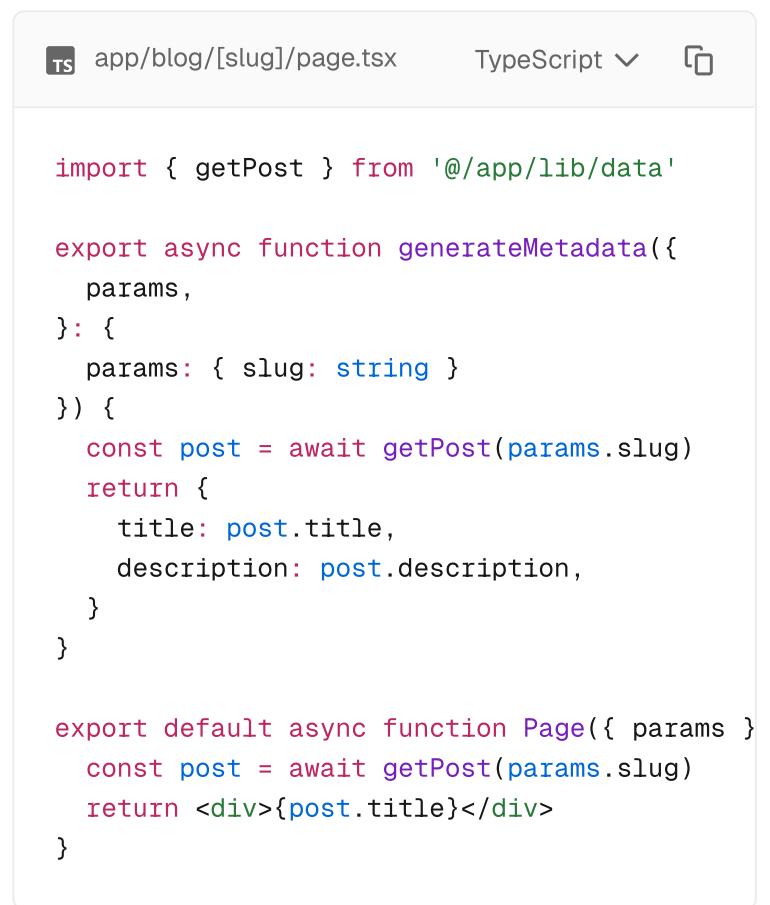
There may be cases where you need to fetch the **same** data for metadata and the page itself. To avoid duplicate requests, you can use React's [cache function ↗](#) to memoize the return value and only fetch the data once. For example, to fetch the blog post information for both the metadata and the page:



app/lib/data.ts TypeScript ▾

```
import { cache } from 'react'
import { db } from '@/app/lib/db'

// getPost will be used twice, but execute once
export const getPost = cache(async (slug: string) => {
  const res = await db.query.posts.findFirst({ where: { slug } })
  return res
})
```



app/blog/[slug]/page.tsx TypeScript ▾

```
import { getPost } from '@/app/lib/data'

export async function generateMetadata({ params, }: { params: { slug: string } }) {
  const post = await getPost(params.slug)
  return {
    title: post.title,
    description: post.description,
  }
}

export default async function Page({ params }) {
  const post = await getPost(params.slug)
  return <div>{post.title}</div>
}
```

File-based metadata

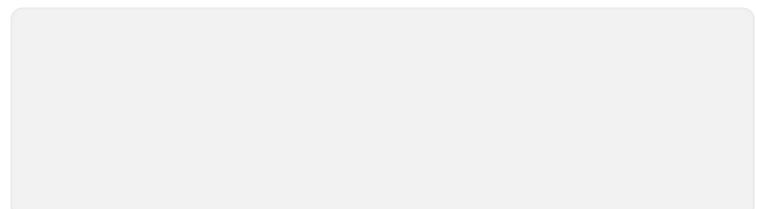
The following special files are available for metadata:

- [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.

Favicons

Favicons are small icons that represent your site in bookmarks and search results. To add a favicon to your application, create a `favicon.ico` and add to the root of the app folder.



You can also programmatically generate favicons using code. See the [favicon docs](#) for more information.

Static Open Graph images

Open Graph (OG) images are images that represent your site in social media. To add a static OG image to your application, create a `opengraph-image.png` file in the root of the app folder.

You can also add OG images for specific routes by creating a `opengraph-image.png` deeper down the folder structure. For example, to create an OG image specific to the `/blog` route, add a `opengraph-image.jpg` file inside the `blog` folder.

The more specific image will take precedence over any OG images above it in the folder structure.

Other image formats such as `jpeg`, `png`, and `gif` are also supported. See the [Open Graph Image docs](#) for more information.

Generated Open Graph images

The [ImageResponse constructor](#) allows you to generate dynamic images using JSX and CSS. This is useful for OG images that depend on data.

For example, to generate a unique OG image for each blog post, add a `opengraph-image.ts` file inside the `blog` folder, and import the `ImageResponse` constructor from `next/og`:

```
import { ImageResponse } from 'next/og'
import {getPost} from '@/app/lib/data'

// Image metadata
export const size = {
  width: 1200,
  height: 630,
}

export const contentType = 'image/png'

// Image generation
export default async function Image({ params }) {
  const post = await getPost(params.slug)

  return new ImageResponse(
    (
      // ImageResponse JSX element
      <div
        style={{
          fontSize: 128,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        {post.title}
      </div>
    )
  )
}
```

`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 flexbox and a subset of CSS properties are supported. Advanced layouts (e.g. `display: grid`) will not work.

API Reference

Learn more about the Metadata APIs mentioned in this page.

generateMet...

Learn how to add Metadata to your Next.js applicatio...

generateVie...

API Reference for the generateViewport...

ImageRespo...

API Reference for the ImageResponse...

Metadata Fil...

API documentation for the metadata file conventions.

favicon, icon,...

API Reference for the Favicon, Icon and Apple Icon fil...

opengraph-i...

API Reference for the Open Graph Image and Twitte...

robots.txt

API Reference for robots.txt file.

sitemap.xml

API Reference for the sitemap.xml file.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

Route Handlers and Middleware

Route Handlers

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



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

Convention

Route Handlers are defined in a [`route.js|ts`](#) file inside the `app` directory:



```
TS app/api/route.ts TypeScript ▾ ⌂
export async function GET(request: 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:

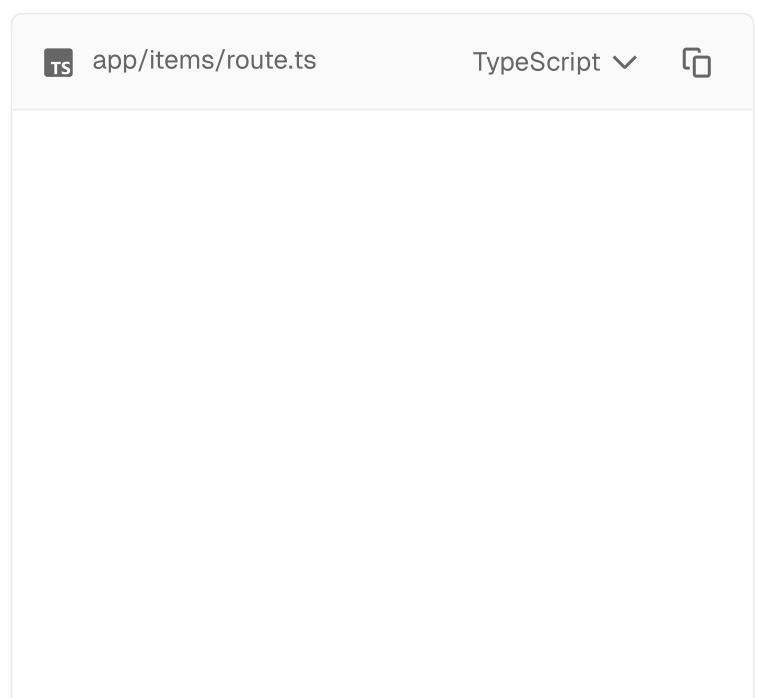
`GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD`, and `OPTIONS`. If an unsupported method is called, Next.js will return a `405 Method Not Allowed` response.

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.

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.



```
TS app/items/route.ts TypeScript ▾
```

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

```

        export const dynamic = 'force-static'

        export async function GET() {
            const res = await fetch('https://data.mongo
            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
<code>app/page.js</code>	<code>app/route.js</code>	✗ Conflict
<code>app/page.js</code>	<code>app/api/route.js</code>	✓ Valid
<code>app/[user]/page.js</code>	<code>app/api/route.js</code>	✓ Valid

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

app/page.ts

TypeScript ▾

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

// Conflict
// `app/route.ts`
export async function POST(request: Request)
```

Read more about how Route Handlers [complement your frontend application](#), or explore the Route Handlers [API Reference](#).

Route Context Helper

In TypeScript, you can type the `context` parameter for Route Handlers with the globally available `RouteContext` helper:

app/users/[id]/route.ts

TypeScript ▾

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

export async function GET(_req: NextRequest,
  const { id } = await ctx.params
  return Response.json({ id })
}
```

Good to know

- Types are generated during `next dev`, `next build` or `next typegen`.

Middleware

Middleware allows you to run code before a request is completed. Then, based on the incoming request, you can modify the response by rewriting, redirecting, modifying the request or response headers, or responding directly.

Use cases

Some common scenarios where Middleware is effective include:

- Quick redirects after reading parts of the incoming request
- Rewriting to different pages based on A/B tests or experiments
- Modifying headers for all pages or a subset of pages

Middleware is *not* a good fit for:

- Slow data fetching
- Session management

Using `fetch` with `options.cache`,
`options.next.revalidate`, or
`options.next.tags`, has no effect in Middleware.

Convention

Create a `middleware.ts` (or `.js`) file in the project root, or inside `src` if applicable, so that it is located at the same level as `pages` or `app`.

Note: While only one `middleware.ts` file is supported per project, you can still organize your middleware logic into modules. 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



```
TS middleware.ts TypeScript ▾ ⌂
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

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

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

Read more about [using middleware](#), or refer to the [middleware API reference](#).

API Reference

Learn more about Route Handlers and Middleware

route.js

API reference for the route.js special file.

middleware.js

API reference for the middleware.js file.

Backend for ...

Learn how to use Next.js as a backend...

Was this helpful?  



 Using App Router
Features available in /app Latest Version
15.5.4

Deploying

Next.js can be deployed as a Node.js server, Docker container, static export, or adapted to run on different platforms.

Deployment Option	Feature Support
Node.js server	All
Docker container	All
Static export	Limited
Adapters	Platform-specific

Node.js server

Next.js can be deployed to any provider that supports Node.js. Ensure your `package.json` has the `"build"` and `"start"` scripts:



```
JSON package.json
```

```
{  
  "scripts": {  
    "dev": "next dev",  
    "build": "next build",  
    "start": "next start"  
  }  
}
```

Then, run `npm run build` to build your application and `npm run start` to start the Node.js server. This server supports all Next.js features. If needed, you can also eject to a [custom server](#).

Node.js deployments support all Next.js features. Learn how to [configure them](#) for your infrastructure.

Templates

- [Flightcontrol ↗](#)
 - [Railway ↗](#)
 - [Replit ↗](#)
-

Docker

Next.js can be deployed to any provider that supports [Docker ↗](#) containers. This includes container orchestrators like Kubernetes or a cloud provider that runs Docker.

Docker deployments support all Next.js features. Learn how to [configure them](#) for your infrastructure.

Note for development: While Docker is excellent for production deployments, consider using local development (`npm run dev`) instead of Docker during development on Mac and Windows for better performance. [Learn more about optimizing local development](#).

Templates

- [Docker ↗](#)
- [Docker Multi-Environment ↗](#)
- [DigitalOcean ↗](#)

- [Fly.io ↗](#)
 - [Google Cloud Run ↗](#)
 - [Render ↗](#)
 - [SST ↗](#)
-

Static 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 [static exports](#), it can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets. This includes tools like AWS S3, Nginx, or Apache.

Running as a [static export does not support](#) Next.js features that require a server. [Learn more](#).

Templates

- [GitHub Pages ↗](#)
-

Adapters

Next.js can be adapted to run on different platforms to support their infrastructure capabilities.

Refer to each provider's documentation for information on supported Next.js features:

- [AWS Amplify Hosting ↗](#)
- [Cloudflare ↗](#)
- [Deno Deploy ↗](#)

- [Netlify ↗](#)
- [Vercel ↗](#)

Note: We are working on a [Deployment Adapters API ↗](#) for all platforms to adopt. After completion, we will add documentation on how to write your own adapters.

Was this helpful?    



Using App Router

Features available in /app



Upgrading



Latest Version

15.5.4

Latest version

To update to the latest version of Next.js, you can use the `upgrade` codemod:

>_ Terminal



```
npx @next/codemod@latest upgrade latest
```

If you prefer to upgrade manually, install the latest Next.js and React versions:

pnpm

npm

yarn

bun

>_ Terminal



```
pnpm i next@latest react@latest react-dom@lat
```

Canary version

To update to the latest canary, make sure you're on the latest version of Next.js and everything is working as expected. Then, run the following command:

>_ Terminal



```
npm i next@canary
```

Features available in canary

The following features are currently available in canary:

Caching:

- `"use cache"`
- `cacheLife`
- `cacheTag`
- `cacheComponents`

Authentication:

- `forbidden`
- `unauthorized`
- `forbidden.js`
- `unauthorized.js`
- `authInterrups`

Version guides

See the version guides for in-depth upgrade instructions.

Version 15

Upgrade your Next.js Application from Version 14 t...

Version 14

Upgrade your Next.js Application from Version 13 t...

Was this helpful?    



Using App Router

Features available in /app



Guides



Latest Version

15.5.4



Analytics

Measure and track page performance using Next.js...

Authenticati...

Learn how to implement authentication in...

Backend for ...

Learn how to use Next.js as a backend...

Caching

An overview of caching mechanisms in...

CI Build Cac...

Learn how to configure CI to cache Next.js...

Content Sec...

Learn how to set a Content Security Policy (CSP) for...

CSS-in-JS

Use CSS-in-JS libraries with Next.js

Custom Serv...

Start a Next.js app programmatically using a custom...

Data Security

Learn the built-in data security features in Next.j...

Debugging

Learn how to debug your Next.js application with...

Draft Mode

Next.js has draft mode to toggle between static...

Environment ...

Learn to add and access environment...

Forms

Learn how to create forms in Next.js with Reac...

ISR

Learn how to create or update static pages at...

Instrumentat...

Learn how to use instrumentation to run code at serve...

International...

Add support for multiple languages with...

JSON-LD

Learn how to add JSON-LD to your Next.js applicatio...

Lazy Loading

Lazy load imported libraries and React...

Developmen...

MDX

Learn how to optimize your local development...

Learn how to configure MDX and use it in your...

Memory Usa...

Optimize memory used by your application in...

Migrating

Learn how to migrate from popular...

Multi-tenant

Learn how to build multi-tenant apps with the App...

Multi-zones

Learn how to build micro-frontends using Next.js...

OpenTeleme...

Learn how to instrument your Next.js app with...

Package Bun...

Learn how to optimize your application's...

Prefetching

Learn how to configure prefetching in...

Production

Recommendations to ensure the best performance and...

PWAs

Learn how to build a Progressive Web Application (PWA)...

Redirecting

Learn the different ways to handle redirects in...

Sass

Style your Next.js application using Sass.

Scripts

Optimize 3rd party scripts with the built-in Script...

Self-Hosting

Learn how to self-host your Next.js application on a...

SPAs

Next.js fully supports building Single-Page...

Static Exports

Next.js enables starting as a static site or Single-Pag...

Tailwind CSS...

Style your Next.js Application using Tailwind CSS v3...

Testing

Learn how to set up Next.js with four commonly...

Third Party Li...

Optimize the performance of third-party...

Upgrading

Learn how to upgrade to the latest versions of...

Videos

Recommendations and best practices for optimizing...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4



How to add analytics to your Next.js application

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.

Client Instrumentation

For more advanced analytics and monitoring needs, Next.js provides a

`instrumentation-client.js|ts` file that runs before your application's frontend code starts executing. This is ideal for setting up global analytics, error tracking, or performance monitoring tools.

To use it, create an `instrumentation-client.js` or `instrumentation-client.ts` file in your application's root directory:

```
JS instrumentation-client.js   
  
// Initialize analytics before the app starts  
console.log('Analytics initialized')  
  
// Set up global error tracking  
window.addEventListener('error', (event) => {
```

```
// Send to your error tracking service  
reportError(event.error)  
})
```

Build Your Own

JS app/_components/web-vitals.js

```
'use client'

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

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

JS app/layout.js

```
import { WebVitals } from './_components/web-vitals'

export default function Layout({ children }) {
  return (
    <html>
      <body>
        <WebVitals />
        {children}
      </body>
    </html>
  )
}
```

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

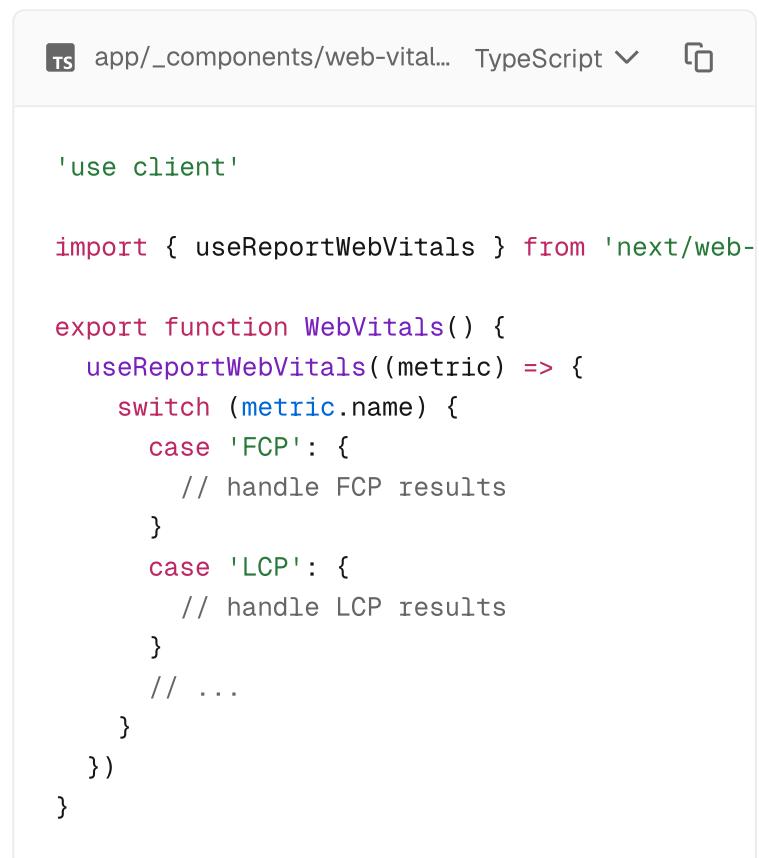
View the [API Reference](#) for more information.

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.



The screenshot shows a code editor window with a TypeScript file open. The file contains code for handling different Web Vitals based on their name. It uses the `useReportWebVitals` hook from the `'next/web-vitals'` package. The code includes cases for FCP, LCP, and other metrics, each handling their respective 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
      }
      // ...
    }
  })
}
```

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
  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
  // https://github.com/vercel/next.js/blob/canary/packages/next/integration/gtag/gtag.js#L10-L12
  window.gtag('event', metric.name, {
    value: Math.round(
      metric.name === 'CLS' ? metric.value : metric.value * 100
    ), // values must be integers
    event_label: metric.id, // id unique to this metric
    non_interaction: true, // avoids affecting other metrics
  })
})
```

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

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4



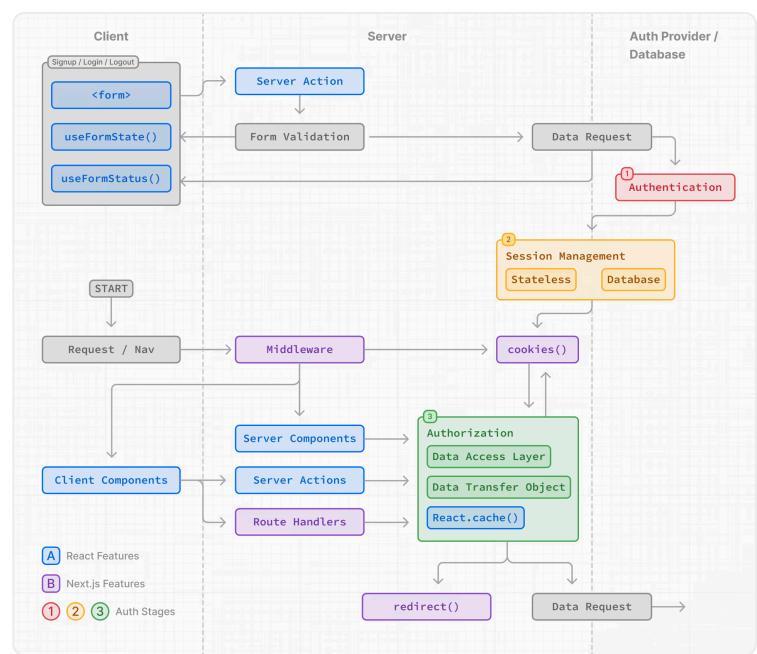
How to implement authentication in Next.js

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 `useActionState` 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:

```
TS app/ui/signup-form.tsx TypeScript ▾ ⌂

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" type="text" />
      </div>
      <div>
        <label htmlFor="email">Email</label>
        <input id="email" name="email" type="text" placeholder="Email" />
      </div>
      <div>
        <label htmlFor="password">Password</label>
        <input id="password" name="password" type="password" />
      </div>
      <button type="submit">Sign Up</button>
    </form>
  )
}
```

```
TS app/actions/auth.ts TypeScript ▾ ⌂

export async function signup(formData: FormData) {
  const response = await fetch('/api/auth/signup', {
    method: 'POST',
    body: formData,
  })
  const data = await response.json()
  if (data.error) {
    throw new Error(data.error)
  }
  return data
}
```

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:

```
TS app/lib/definitions.ts TypeScript ▾
```

```
import { z } from 'zod'

export const SignupFormSchema = z.object({
  name: z
    .string()
    .min(2, { message: 'Name must be at least 2 characters' })
    .trim(),
  email: z.string().email({ message: 'Please enter a valid email address' }),
  password: z
    .string()
    .min(8, { message: 'Be at least 8 characters' })
    .regex(/^[a-zA-Z]+$/i, { message: 'Contain at least one uppercase letter' })
    .regex(/[0-9]+/, { message: 'Contain at least one digit' })
    .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
```

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.

```
TS app/actions/auth.ts TypeScript ▾
```

```
import { SignupFormSchema, FormState } from 'lib/definitions'

export async function signup(state: FormState) {
  // Validate form fields
  const validatedFields = SignupFormSchema.safeParse(state)
  const { name, email, password } = validatedFields.data
```

```

        email: formData.get('email'),
        password: formData.get('password'),
    })

    // If any form fields are invalid, return errors
    if (!validatedFields.success) {
        return {
            errors: validatedFields.error.flatten()
        }
    }

    // Call the provider or db to create a user
}

```

Back in your `<SignupForm />`, you can use React's `useActionState` hook to display validation errors while the form is submitting:



```

TS app/ui/signup-form.tsx TypeScript ▾ ⌂
'use client'

import { signup } from '@/app/actions/auth'
import { useActionState } from 'react'

export default function SignupForm() {
    const [state, action, pending] = useActions(signup)

    return (
        <form action={action}>
            <div>
                <label htmlFor="name">Name</label>
                <input id="name" name="name" placeholder="Name" type="text" />
            </div>
            {state?.errors?.name && <p>{state.errors.name}</p>}
        <div>
            <label htmlFor="email">Email</label>
            <input id="email" name="email" placeholder="Email" type="text" />
        </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>{error}</li>
                    ))}
                </ul>
            </div>
        )}
    )
}

```

```
        <li key={error}>- {error}</li>
      ))
    </ul>
  </div>
)
<button disabled={pending} type="submit">
  Sign Up
</button>
</form>
)
}
```

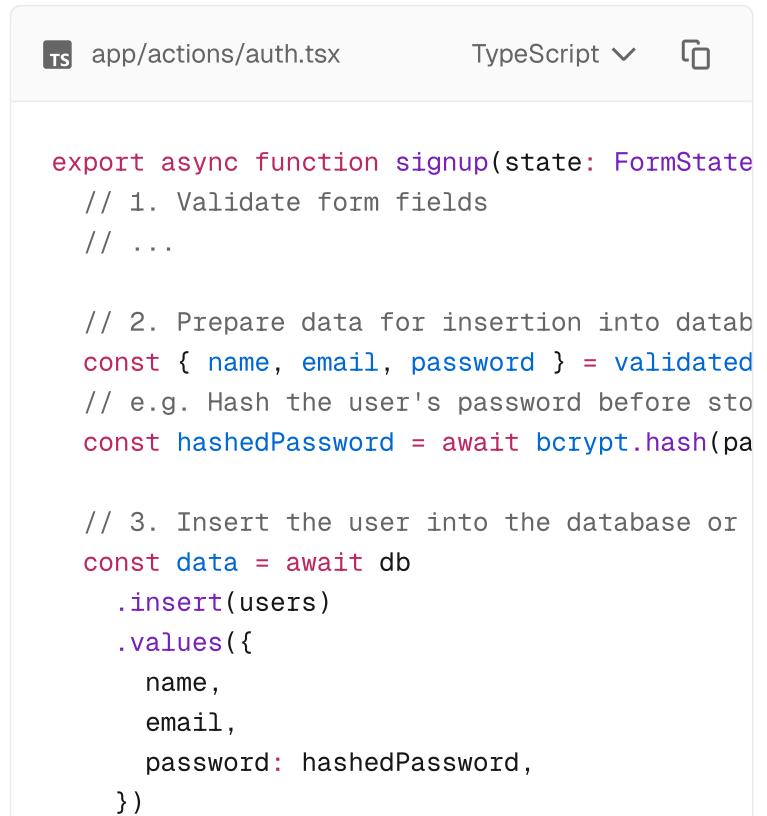
Good to know:

- 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.
- 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:



The screenshot shows a code editor window with a TypeScript file named `app/actions/auth.tsx`. The code defines a `signup` function that performs three main steps: validating form fields, preparing data for insertion into a database, and inserting the user into the database.

```
TS app/actions/auth.tsx TypeScript ▾
```

```
export async function signup(state: FormState
  // 1. Validate form fields
  // ...
  // 2. Prepare data for insertion into database
  const { name, email, password } = validated
  // e.g. Hash the user's password before storage
  const hashedPassword = await bcrypt.hash(password)
  // 3. Insert the user into the database or
  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 creat  
  }  
}  
  
// 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.

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 a 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:

```
>_ 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](#):

```
.env
SESSION_SECRET=your_secret_key
```

You can then reference this key in your session management logic:

```
app/lib/session.js
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.

```
app/lib/session.ts
TypeScript ▾
```

```

import 'server-only'
import { SignJWT, jwtVerify } from 'jose'
import { SessionPayload } from '@/app/lib/def

const secretKey = process.env.SESSION_SECRET
const encodedKey = new TextEncoder().encode(s

export async function encrypt(payload: SessionPayload) {
  return new SignJWT(payload)
    .setProtectedHeader({ alg: 'HS256' })
    .setIssuedAt()
    .setExpirationTime('7d')
    .sign(encodedKey)
}

export async function decrypt(session: string) {
  try {
    const { payload } = await jwtVerify(session, {
      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.

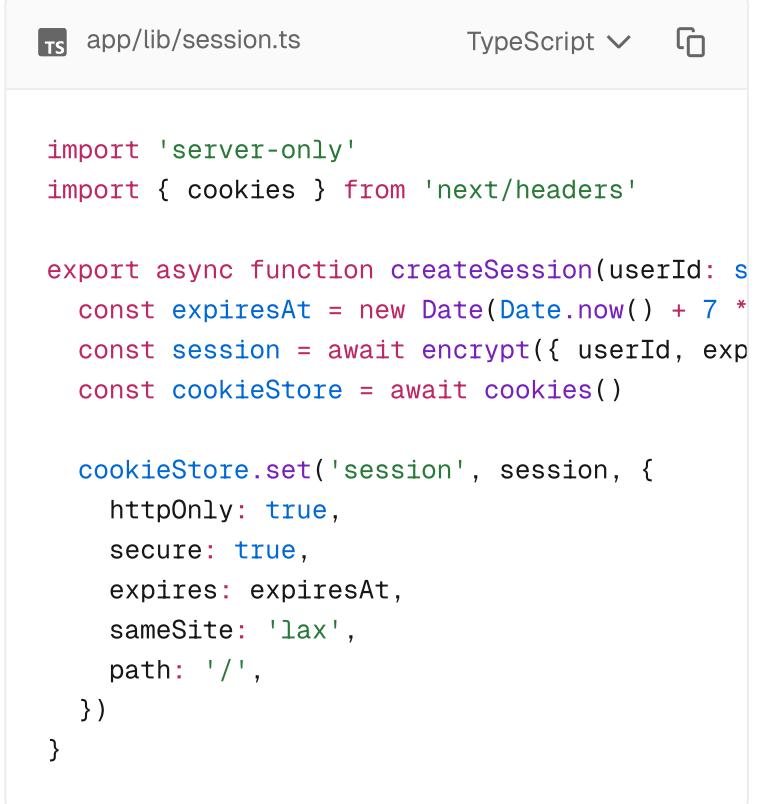
3. Setting cookies (recommended options)

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.



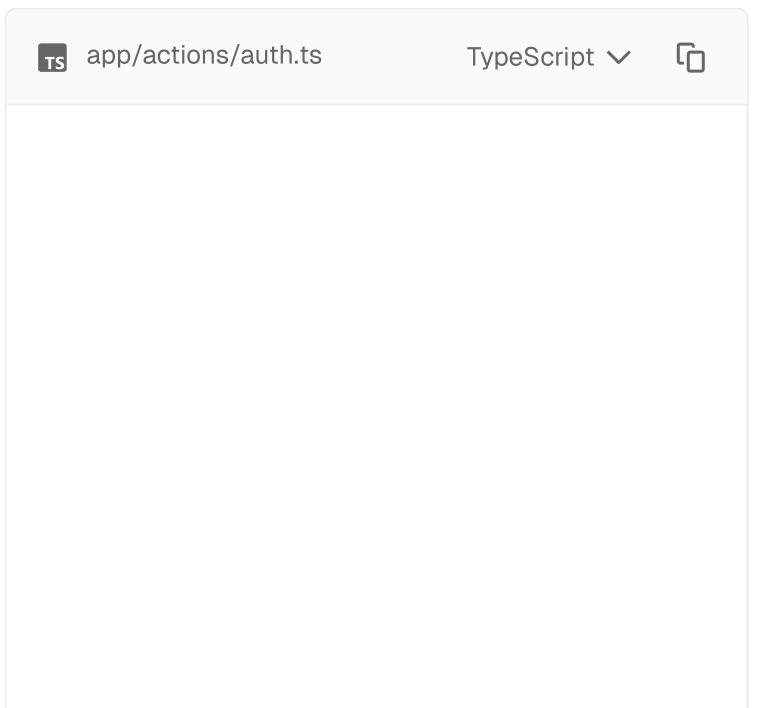
The screenshot shows a code editor window with the file name "app/lib/session.ts" at the top. The code is written in TypeScript and uses the 'server-only' import from 'next/headers'. It defines an asynchronous function "createSession" that takes a userId parameter. Inside the function, it creates a new Date object for expiration, encrypts the session data, and then uses the "cookies" module to set a session cookie with various options: httpOnly: true, secure: true, expires: expiresAt, sameSite: 'lax', path: '/', and a custom key 'session'. The code ends with a closing brace for the function and another closing brace for the module export.

```
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 })
  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:



The screenshot shows a code editor window with the file name "app/actions/auth.ts" at the top. The code is currently empty, consisting of a single blank line.

```
import { createSession } from '@/app/lib/sess

export async function signup(state: FormState
    // Previous steps:
    // 1. Validate form fields
    // 2. Prepare data for insertion into database
    // 3. Insert the user into the database or

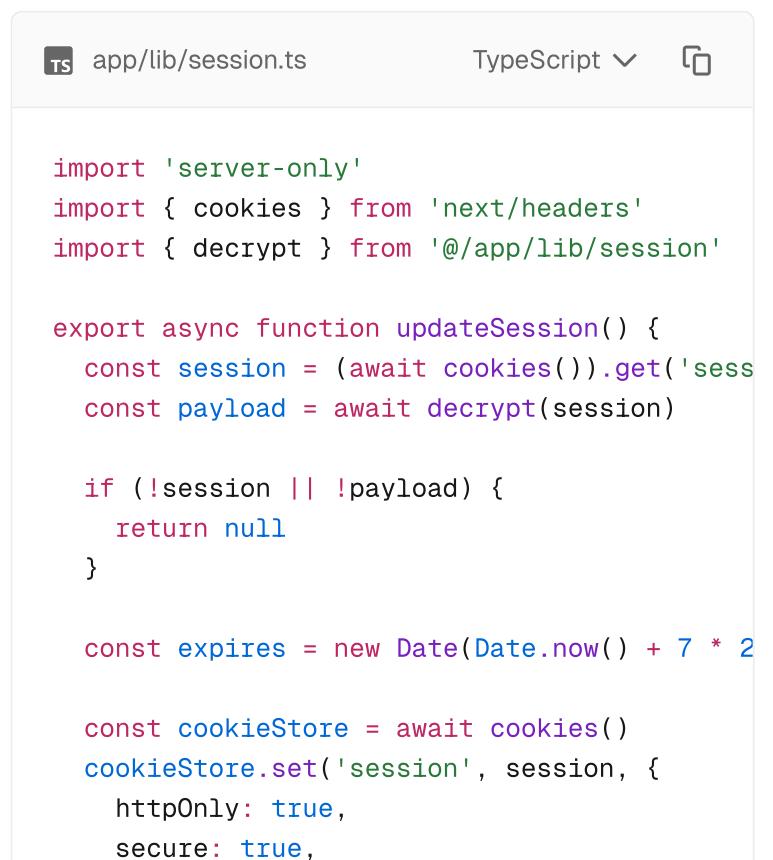
    // Current steps:
    // 4. Create user session
    await createSession(user.id)
    // 5. Redirect user
    redirect('/profile')
}
```

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:



The screenshot shows a code editor interface with a tab labeled "app/lib/session.ts". The code is written in TypeScript and uses the "server-only" module. It defines an asynchronous function "updateSession" that retrieves a session from cookies, decrypts its payload, and then updates it with a new expiration date (7 days from now). The updated session is then stored back into the cookie store.

```
import 'server-only'
import { cookies } from 'next/headers'
import { decrypt } from '@/app/lib/session'

export async function updateSession() {
    const session = (await cookies()).get('session')
    const payload = await decrypt(session)

    if (!session || !payload) {
        return null
    }

    const expires = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)

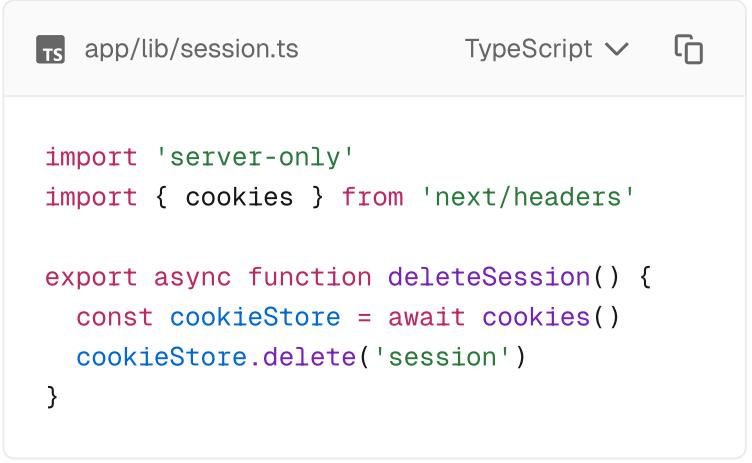
    const cookieStore = await cookies()
    cookieStore.set('session', session, {
        httpOnly: true,
        secure: true,
    })
}
```

```
        expires: expires,  
        sameSite: 'lax',  
        path: '/',  
    })  
}
```

Tip: Check if your auth library supports refresh tokens, which can be used to extend the user's session.

Deleting the session

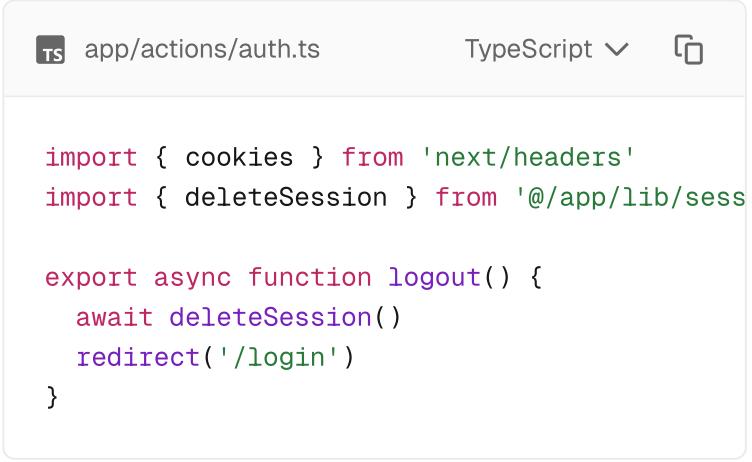
To delete the session, you can delete the cookie:



```
app/lib/session.ts
```

```
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:



```
app/actions/auth.ts
```

```
import { cookies } from 'next/headers'  
import { deleteSession } from '@/app/lib/sess  
  
export async function logout() {  
    await deleteSession()  
    redirect('/login')  
}
```

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:



The screenshot shows a code editor window with the following details:

- File Path:** app/lib/session.ts
- Language:** TypeScript
- Code Content:**

```

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,

    // 3. Store the session in cookies for opti
    const cookieStore = await cookies()
    cookieStore.set('session', session, {
      httpOnly: true,
      secure: true,
      expires: expiresAt,
      sameSite: 'lax',
      path: '/',
    })
  }
}

```

Tips:

- For faster access, you may consider adding server caching for the lifetime of the session. 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.

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:



```
TS middleware.ts TypeScript ▾
```

```
import { NextRequest, NextResponse } from 'next'
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
    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')
    const session = await decrypt(cookie)

    // 4. Redirect to /login if the user is not logged in
    if (isProtectedRoute && !session?.userId) {
      return NextResponse.redirect(new URL('/login'))
    }

    // 5. Redirect to /dashboard if the user is logged in but not on the dashboard
    if (
      isPublicRoute &&
      session?.userId &&
      !req.nextUrl.pathname.startsWith('/dashboard')
    ) {
      return NextResponse.redirect(new URL('/dashboard'))
    }

    return NextResponse.next()
  }

  // Routes Middleware should not run on static files
  export const config = {
    matcher: ['/((?!api|_next/static|_next/image).*)']
  }

```

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:

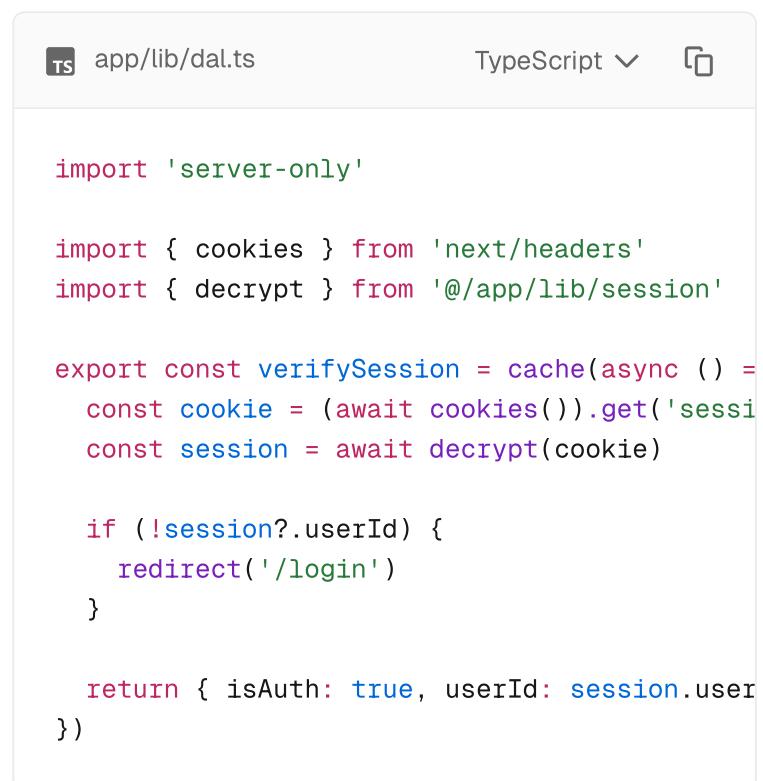
- In Middleware, you can also read cookies using `req.cookies.get('session').value`.
- Middleware uses the [Edge Runtime](#), check if your Auth library and session management library are compatible.
- You can use the `matcher` property in the Middleware to specify which routes Middleware should run 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:



```
TS app/lib/dal.ts TypeScript ▾
```

```
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')
  const session = await decrypt(cookie)

  if (!session?.userId) {
    redirect('/login')
  }

  return { isAuthenticated: true, userId: session.userId }
})
```

You can then invoke the `verifySession()` function in your data requests, Server Actions, Route Handlers:



```
TS app/lib/dal.ts TypeScript ▾
```

```
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
    columns: {
      id: true,
      name: true,
      email: true,
    },
  )
}

const user = data[0]

return user
} catch (error) {
  console.log('Failed to fetch user')
  return null
}
})

```

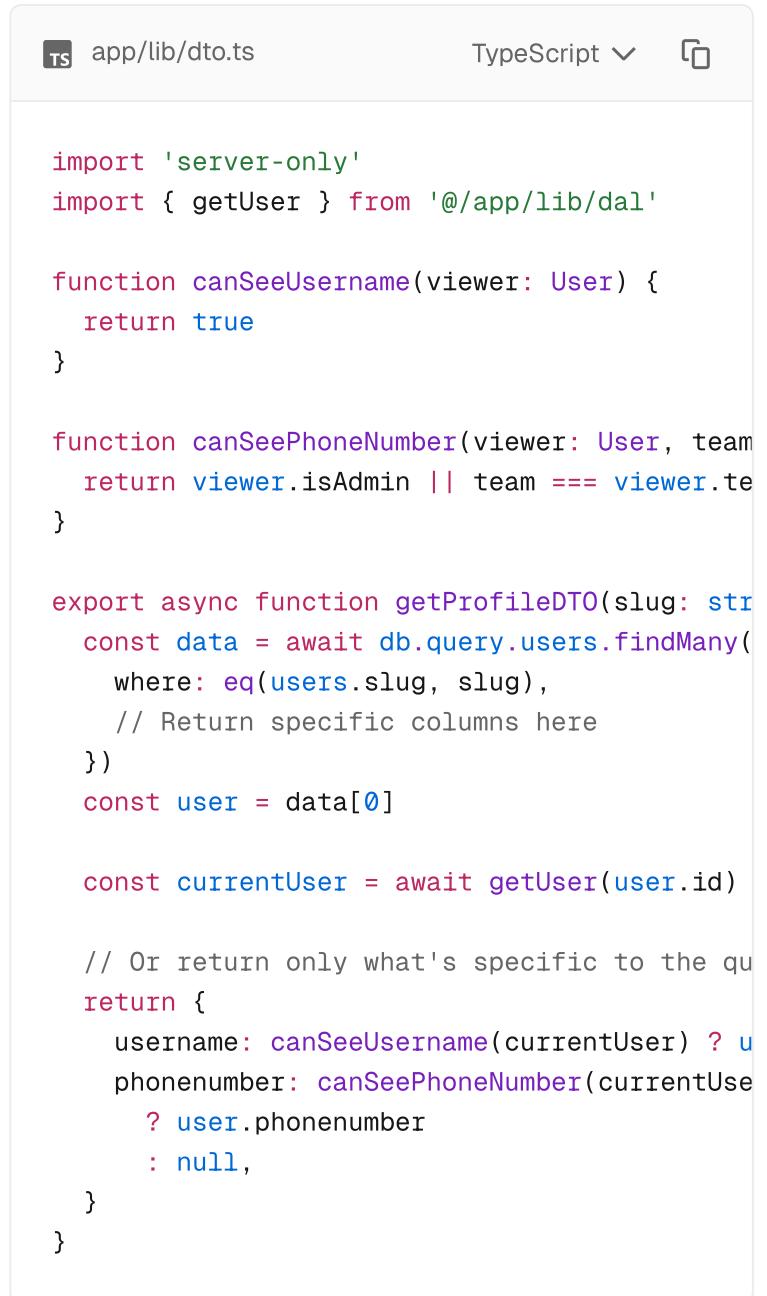
Tip:

- 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.



The screenshot shows a code editor window with the following details:

- File path: app/lib/dto.ts
- Language: TypeScript
- Code content:

```
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
  return {
    username: canSeeUsername(currentUser) ? user.username : null,
    phononenumber: canSeePhoneNumber(currentUser) ? user.phononenumber : 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:

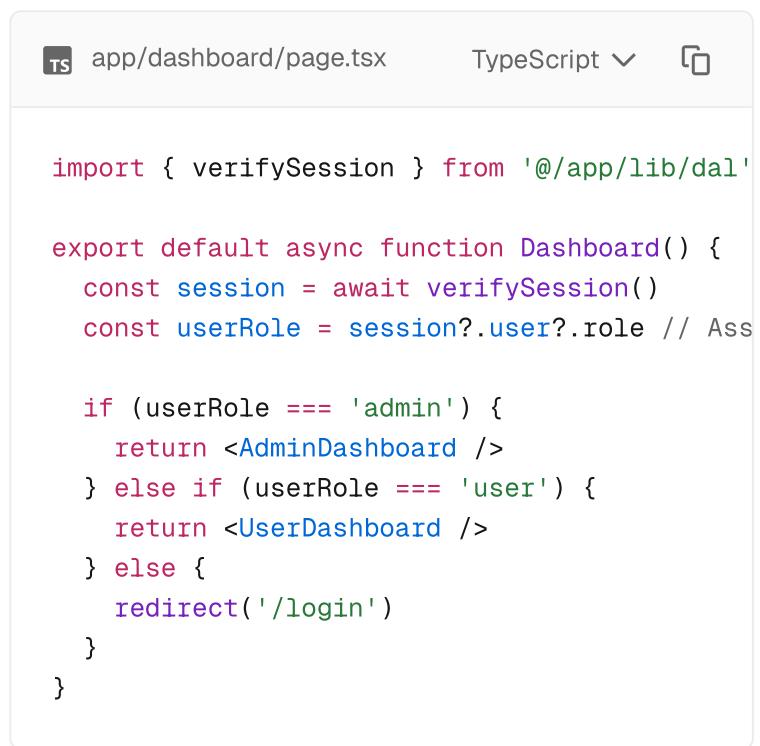
- There are a couple of different ways you can define a DTO, from using `toJSON()`, to individual functions like the example above, or JS classes.

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:



```
TS app/dashboard/page.tsx TypeScript ▾ ⌂

import { verifySession } from '@/app/lib/dal'

export default async function Dashboard() {
  const session = await verifySession()
  const userRole = session?.user?.role // Ass

  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.

```
TS app/layout.tsx TypeScript ▾
```

```
export default async function Layout({  
    children,  
}: {  
    children: React.ReactNode;  
}) {  
    const user = await getUser();  
  
    return (  
        // ...  
    )  
}
```

```
TS app/lib/dal.ts TypeScript ▾
```

```
export const getUser = cache(async () => {  
    const session = await verifySession()  
    if (!session) return null  
  
    // Get user ID from session and fetch data  
})
```

Good to know:

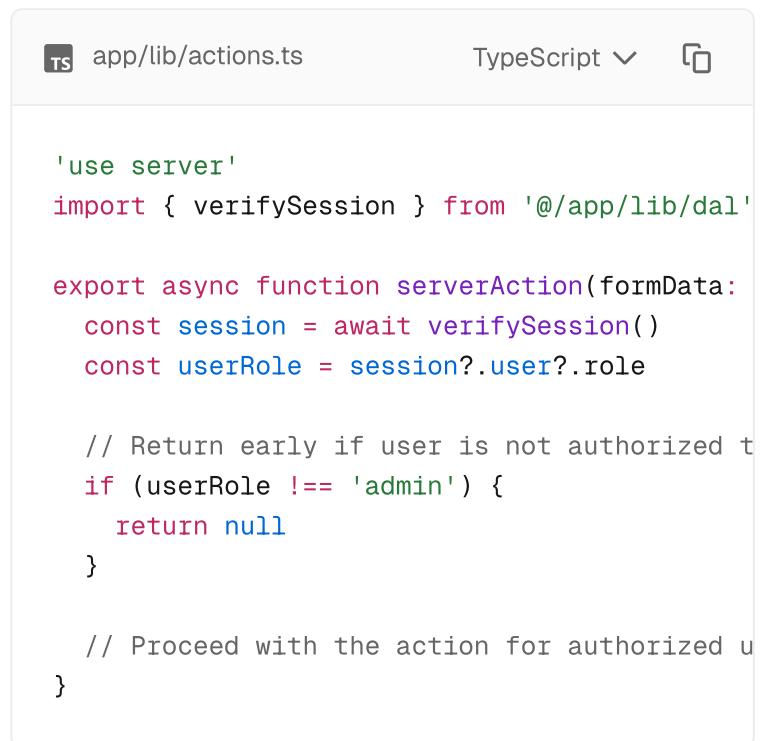
- A common pattern in SPAs is to `return null` in a layout or a top-level component if a user is not authorized. This pattern is **not recommended** since Next.js applications have multiple entry points, which will not prevent nested route

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:



The screenshot shows a code editor window with the following details:

- File path: app/lib/actions.ts
- Language: TypeScript (indicated by the 'TS' icon)
- Code content:

```
'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
  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:



The screenshot shows a code editor window with the following details:

- File path: app/api/route.ts
- Language: TypeScript (indicated by the 'TS' icon)
- Code content:

```
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 'admin' role
  return new Response(null, { status: 403 })
}

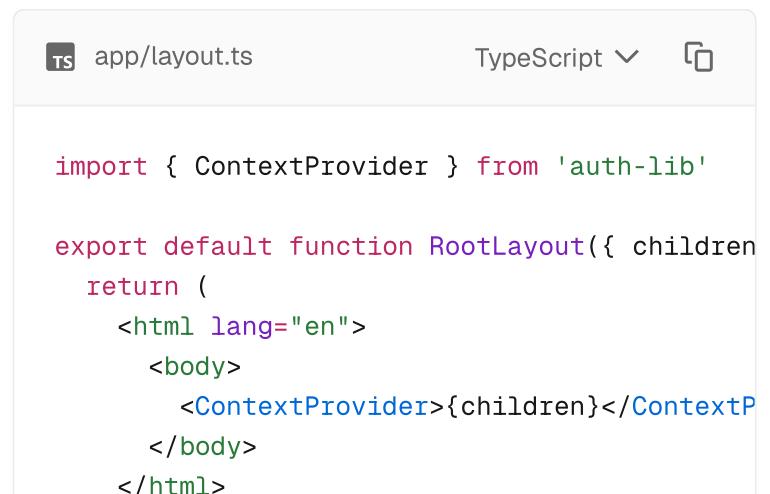
// Continue for authorized users
}
```

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:



```
TS app/layout.ts TypeScript ▾ ⌂

import { ContextProvider } from 'auth-lib'

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

```
)  
}
```

```
'use client';

import { useSession } from "auth-lib";

export default function Profile() {
  const { userId } = useSession();
  const { data } = useSWR(`/api/user/${userId}`)

  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.

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 ↗](#)
- [Better Auth ↗](#)
- [Clerk ↗](#)
- [Kinde ↗](#)
- [Logto ↗](#)
- [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 ↗](#)

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



How to use Next.js as a backend for your frontend

Next.js supports the "Backend for Frontend" pattern. This lets you create public endpoints to handle HTTP requests and return any content type—not just HTML. You can also access data sources and perform side effects like updating remote data.

If you are starting a new project, using

```
create-next-app with the --api flag automatically includes an example route.ts in your new project's app/ folder, demonstrating how to create an API endpoint.
```

```
>_ Terminal   
  
npx create-next-app@latest --api
```

Good to know: Next.js backend capabilities are not a full backend replacement. They serve as an API layer that:

- is publicly reachable
- handles any HTTP request
- can return any content type

To implement this pattern, use:

- [Route Handlers](#)
- [middleware](#)
- In Pages Router, [API Routes](#)

Public Endpoints

Route Handlers are public HTTP endpoints. Any client can access them.

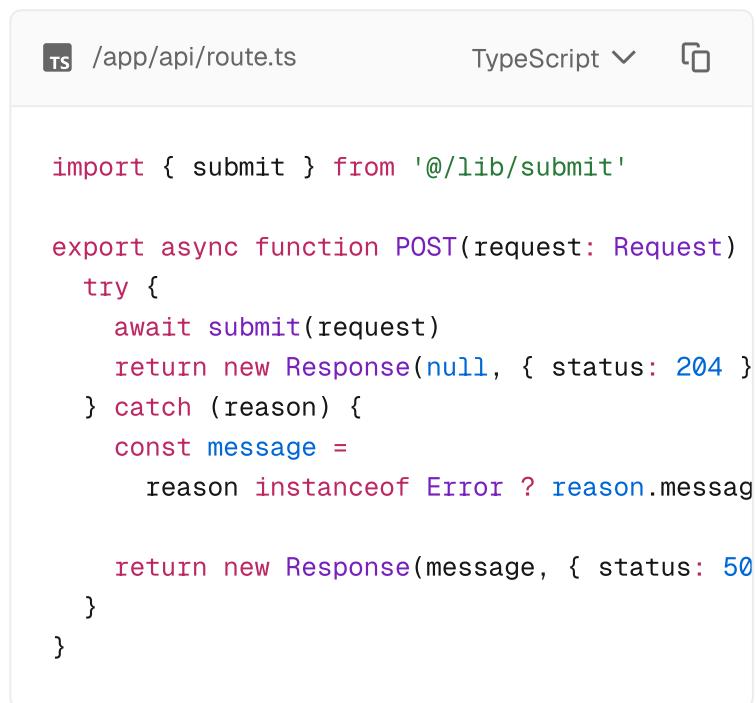
Create a Route Handler using the `route.ts` or `route.js` file convention:



```
TS /app/api/route.ts TypeScript ▾ ⌂
export function GET(request: Request) {}
```

This handles `GET` requests sent to `/api`.

Use `try/catch` blocks for operations that may throw an exception:



```
TS /app/api/route.ts TypeScript ▾ ⌂
import { submit } from '@/lib/submit'

export async function POST(request: Request)
  try {
    await submit(request)
    return new Response(null, { status: 204 })
  } catch (reason) {
    const message =
      reason instanceof Error ? reason.message : 'Unknown error'
    return new Response(message, { status: 500 })
  }
}
```

Avoid exposing sensitive information in error messages sent to the client.

To restrict access, implement authentication and authorization. See [Authentication](#).

Content types

Route Handlers let you serve non-UI responses, including JSON, XML, images, files, and plain text.

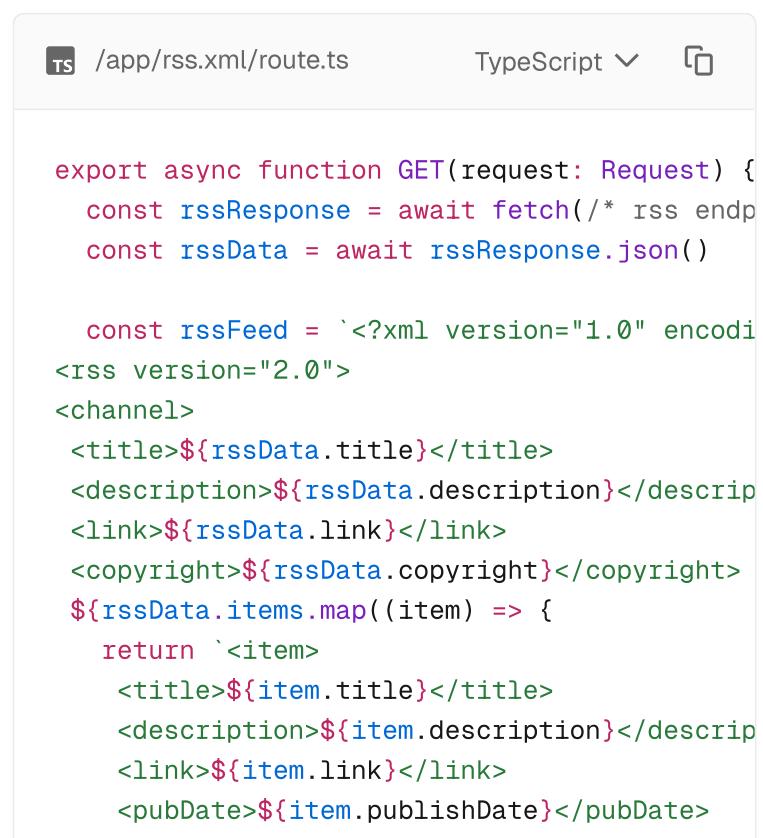
Next.js uses file conventions for common endpoints:

- `sitemap.xml`
- `opengraph-image.jpg`, `twitter-image`
- favicon, app icon, and apple-icon
- `manifest.json`
- `robots.txt`

You can also define custom ones, such as:

- `llms.txt`
- `rss.xml`
- `.well-known`

For example, `app/rss.xml/route.ts` creates a Route Handler for `rss.xml`.



The screenshot shows a code editor interface with a tab bar at the top labeled "TS /app/rss.xml/route.ts" and "TypeScript". The main area contains the following TypeScript code:

```
export async function GET(request: Request) {
  const rssResponse = await fetch(/* rss endpoint */)
  const rssData = await rssResponse.json()

  const rssFeed = `<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
<channel>
  <title>${rssData.title}</title>
  <description>${rssData.description}</description>
  <link>${rssData.link}</link>
  <copyright>${rssData.copyright}</copyright>
  ${rssData.items.map((item) => {
    return `<item>
      <title>${item.title}</title>
      <description>${item.description}</description>
      <link>${item.link}</link>
      <pubDate>${item.publishDate}</pubDate>
    </item>
  `)}
</channel>
</rss>
```

```
        <guid isPermaLink="false">${item.guid}</g
    </item>
  )})
</channel>
</rss>

const headers = new Headers({ 'content-type': 'application/rss+xml' })

return new Response(rssFeed, { headers })
}
```

Sanitize any input used to generate markup.

Consuming request payloads

Use Request [instance methods ↗](#) like `.json()`, `.formData()`, or `.text()` to access the request body.

`GET` and `HEAD` requests don't carry a body.

```
TS /app/api/echo-body/route.ts TypeScript ▾
```

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

Good to know: Validate data before passing it to other systems

```
TS /app/api/send-email/route.ts TypeScript ▾
```

```
import { sendMail, validateInputs } from '@/utils'

export async function POST(request: Request) {
  const formData = await request.formData()
  const email = formData.get('email')
  const contents = formData.get('contents')

  try {
    await validateInputs({ email, contents })
    const info = await sendMail({ email, contents })
    return Response.json({ messageId: info.messageId })
  } catch (error) {
    return Response.json({ error: 'An error occurred while sending the email.' })
  }
}
```

```
    } catch (reason) {
      const message =
        reason instanceof Error ? reason.message : reason

      return new Response(message, { status: 500 })
    }
  }
}
```

You can only read the request body once. Clone the request if you need to read it again:



The screenshot shows a code editor window with a TypeScript file named `/app/api/clone/route.ts`. The code defines a `POST` route handler that attempts to read the request body twice. It clones the request and reads its body twice, which results in an error on the second attempt. A `catch` block handles this error by returning a `500` status response.

```
TS /app/api/clone/route.ts TypeScript ▾
```

```
export async function POST(request: Request) {
  try {
    const clonedRequest = request.clone()

    await request.body()
    await clonedRequest.body()
    await request.body() // Throws error

    return new Response(null, { status: 204 })
  } catch {
    return new Response(null, { status: 500 })
  }
}
```

Manipulating data

Route Handlers can transform, filter, and aggregate data from one or more sources. This keeps logic out of the frontend and avoids exposing internal systems.

You can also offload heavy computations to the server and reduce client battery and data usage.

```
import { parseWeatherData } from '@/lib/weath

export async function POST(request: Request)
  const body = await request.json()
  const searchParams = new URLSearchParams({
```

```
try {
  const weatherResponse = await fetch(`\${
    weatherURL
  }`)

  if (!weatherResponse.ok) {
    /* handle error */
  }

  const weatherData = await weatherResponse.json()
  const payload = parseWeatherData(weatherData)

  return new Response(payload, { status: 200 })
} catch (reason) {
  const message =
    reason instanceof Error ? reason.message : reason

  return new Response(message, { status: 500 })
}
```

Good to know: This example uses `POST` to avoid putting geo-location data in the URL. `GET` requests may be cached or logged, which could expose sensitive info.

Proxying to a backend

You can use a Route Handler as a proxy to another backend. Add validation logic before forwarding the request.

```
TS /app/api/[...slug]/route.ts TypeScript ▾ ⌂

import { isValidRequest } from '@/lib/utils'

export async function POST(request: Request) {
  const clonedRequest = request.clone()
  const isValid = await isValidRequest(clonedRequest)

  if (!isValid) {
    return new Response(null, { status: 400 })
  }

  const { slug } = await params.parse()
  const pathname = slug.join('/')
  const proxyURL = new URL(pathname, 'https://api.example.com')

  return new Response(null, { status: 200 })
}
```

```
const proxyRequest = new Request(proxyURL, {  
  headers: {  
    'User-Agent': 'node-fetch/1.7.10 node/v14.15.4'  
  },  
  referrer: 'https://nextjs.org/docs/api-reference/react-router-dom'  
});  
  
try {  
  return fetch(proxyRequest)  
} catch (reason) {  
  const message =  
    reason instanceof Error ? reason.message : String(reason);  
  
  return new Response(message, { status: 500 })  
}  
}
```

Or use:

- `middleware rewrites`
 - `rewrites` in `next.config.js`.
-

NextRequest and NextResponse

Next.js extends the `Request` ↗ and `Response` ↗ Web APIs with methods that simplify common operations. These extensions are available in both Route Handlers and Middleware.

Both provide methods for reading and manipulating cookies.

`NextRequest` includes the `nextUrl` property, which exposes parsed values from the incoming request, for example, it makes it easier to access request pathname and search params.

`NextResponse` provides helpers like `next()`, `json()`, `redirect()`, and `rewrite()`.

You can pass `NextRequest` to any function expecting `Request`. Likewise, you can return `NextResponse` where a `Response` is expected.

```
TS /app/echo-pathname/route.ts TypeScript
```

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

export async function GET(request: NextRequest) {
  const nextUrl = request.nextUrl

  if (nextUrl.searchParams.get('redirect')) {
    return NextResponse.redirect(new URL('/', nextUrl))
  }

  if (nextUrl.searchParams.get('rewrite')) {
    return NextResponse.rewrite(new URL('/', nextUrl))
  }

  return NextResponse.json({ pathname: nextUrl })
}
```

Learn more about [NextRequest](#) and [NextResponse](#).

Webhooks and callback URLs

Use Route Handlers to receive event notifications from third-party applications.

For example, revalidate a route when content changes in a CMS. Configure the CMS to call a specific endpoint on changes.

```
TS /app/webhook/route.ts TypeScript
```

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

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

  if (token !== process.env.REVALIDATE_SECRET) {
    return NextResponse.json({ success: false })
  }

  const tag = request.nextUrl.searchParams.get('tag')

  if (!tag) {
    return NextResponse.json({ success: false })
  }

  // Revalidate the route with the provided tag
  // ...
}
```

```
revalidateTag(tag)
```

```
    return NextResponse.json({ success: true })
}
```

Callback URLs are another use case. When a user completes a third-party flow, the third party sends them to a callback URL. Use a Route Handler to verify the response and decide where to redirect the user.

TS /app/auth/callback/route.ts TypeScript ▾

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

export async function GET(request: NextRequest) {
  const token = request.nextUrl.searchParams.get('token')
  const redirectUrl = request.nextUrl.searchParams.get('redirectUrl')

  const response = NextResponse.redirect(new URL(redirectUrl))

  response.cookies.set({
    value: token,
    name: '_token',
    path: '/',
    secure: true,
    httpOnly: true,
    expires: undefined, // session cookie
  })

  return response
}
```

Redirects

TS app/api/route.ts TypeScript ▾

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

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

Learn more about redirects in [redirect](#) and [permanentRedirect](#)

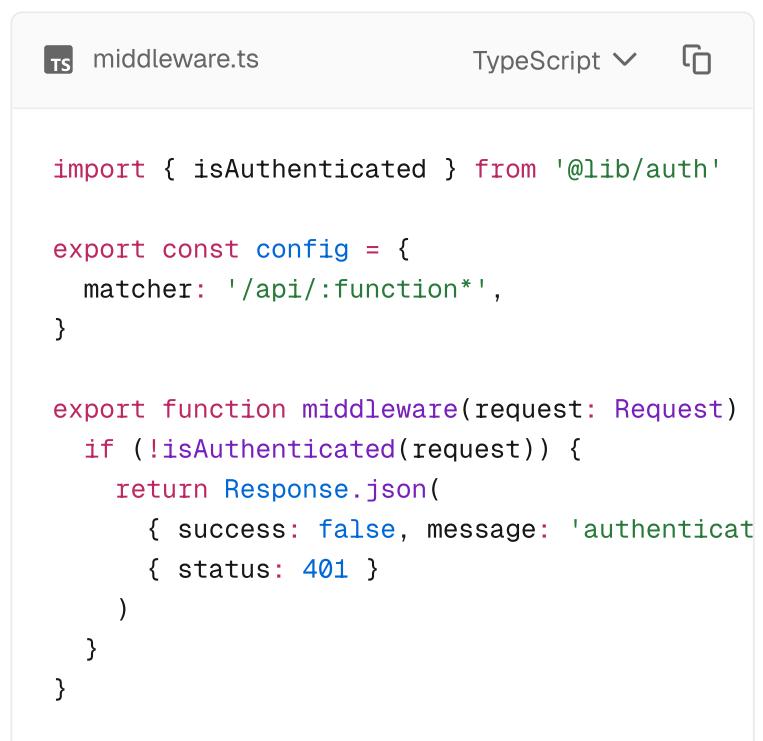
Middleware

Only one middleware file is allowed per project.

Use `config.matcher` to target specific paths.

Learn more about [middleware](#).

Use `middleware` to generate a response before the request reaches a route path.



```
TS middleware.ts TypeScript ▾
```

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

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

export function middleware(request: Request) {
  if (!isAuthenticated(request)) {
    return Response.json(
      { success: false, message: 'authentication required' },
      { status: 401 }
    )
  }
}
```

You can also proxy requests using `middleware`:



```
TS middleware.ts TypeScript ▾
```

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

```
export function middleware(request: Request)
  if (request.nextUrl.pathname === '/proxy-th
    const rewriteUrl = new URL('https://nextj
    return NextResponse.rewrite(rewriteUrl)
  }
}
```

Another type of response `middleware` can produce are redirects:



middleware.ts TypeScript ▾

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

export function middleware(request: Request)
  if (request.nextUrl.pathname === '/v1/docs'
    request.nextUrl.pathname = '/v2/docs'
    return NextResponse.redirect(request.next
  }
}
```

Security

Working with headers

Be deliberate about where headers go, and avoid directly passing incoming request headers to the outgoing response.

- **Upstream request headers:** In Middleware, `NextResponse.next({ request: { headers } })` modifies the headers your server receives and does not expose them to the client.
- **Response headers:** `new Response(... , { headers })`, `NextResponse.json(... , { headers })`, `NextResponse.next({ headers })`, or `response.headers.set(...)` send headers back to the client. If sensitive values were appended to these headers, they will be visible to clients.

Learn more in [NextResponse headers in Middleware](#).

Rate limiting

You can implement rate limiting in your Next.js backend. In addition to code-based checks, enable any rate limiting features provided by your host.



The screenshot shows a code editor window with a TypeScript file named `/app/resource/route.ts`. The code implements rate limiting for a POST endpoint:

```
import { NextResponse } from 'next/server'
import { checkRateLimit } from '@/lib/rate-limit'

export async function POST(request: Request) {
  const { rateLimited } = await checkRateLimit()

  if (rateLimited) {
    return NextResponse.json({ error: 'Rate limit exceeded' })
  }

  return new Response(null, { status: 204 })
}
```

Verify payloads

Never trust incoming request data. Validate content type and size, and sanitize against XSS before use.

Use timeouts to prevent abuse and protect server resources.

Store user-generated static assets in dedicated services. When possible, upload them from the browser and store the returned URI in your database to reduce request size.

Access to protected resources

Always verify credentials before granting access. Do not rely on middleware alone for authentication and authorization.

Remove sensitive or unnecessary data from responses and backend logs.

Rotate credentials and API keys regularly.

Preflight Requests

Preflight requests use the `OPTIONS` method to ask the server if a request is allowed based on origin, method, and headers.

If `OPTIONS` is not defined, Next.js adds it automatically and sets the `Allow` header based on the other defined methods.

- CORS
-

Library patterns

Community libraries often use the factory pattern for Route Handlers.

```
TS /app/api/[...path]/route.ts Copy

import { createHandler } from 'third-party-lib'

const handler = createHandler({
  /* library-specific options */
})

export const GET = handler
// or
export { handler as POST }
```

This creates a shared handler for `GET` and `POST` requests. The library customizes behavior based on the `method` and `pathname` in the request.

Libraries can also provide a `middleware` factory.

middleware.ts



```
import { createMiddleware } from 'third-party'

export default createMiddleware()
```

More examples

See more examples on using [Router Handlers](#) and the [middleware](#) API references.

These examples include, working with [Cookies](#), [Headers](#), [Streaming](#), Middleware [negative matching](#), and other useful code snippets.

Caveats

Server Components

Fetch data in Server Components directly from its source, not via Route Handlers.

For Server Components pre-rendered at build time, using Route Handlers will fail the build step. This is because, while building there is no server listening for these requests.

For Server Components rendered on demand, fetching from Route Handlers is slower due to the extra HTTP round trip between the handler and the render process.

A server side `fetch` request uses absolute URLs. This implies an HTTP round trip, to an external server.

During development, your own development server acts as the external server. At build time there is no server, and at runtime, the server is available through your public facing domain.

Server Components cover most data-fetching needs. However, fetching data client side might be necessary for:

- Data that depends on client-only Web APIs:
 - Geo-location API
 - Storage API
 - Audio API
 - File API
- Frequently polled data

For these, use community libraries like [swr](#) or [react-query](#).

Server Actions

Server Actions let you run server-side code from the client. Their primary purpose is to mutate data from your frontend client.

Server Actions are queued. Using them for data fetching introduces sequential execution.

export mode

`export` mode outputs a static site without a runtime server. Features that require the Next.js runtime are [not supported](#), because this mode produces a static site, and no runtime server.

In `export mode`, only `GET` Route Handlers are supported, in combination with the `dynamic` route segment config, set to `'force-static'`.

This can be used to generate static HTML, JSON, TXT, or other files.

app/hello-world/route.ts



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

export function GET() {
  return new Response('Hello World', { status
}
```

Deployment environment

Some hosts deploy Route Handlers as lambda functions. This means:

- Route Handlers cannot share data between requests.
- The environment may not support writing to File System.
- Long-running handlers may be terminated due to timeouts.
- WebSockets won't work because the connection closes on timeout, or after the response is generated.

API Reference

Learn more about Route Handlers and Middleware

route.js

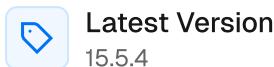
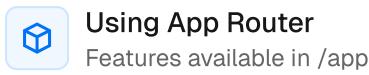
API reference for the route.js special file.

middleware.js

API reference for the middleware.js file.

Was this helpful?  





Caching 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](#).

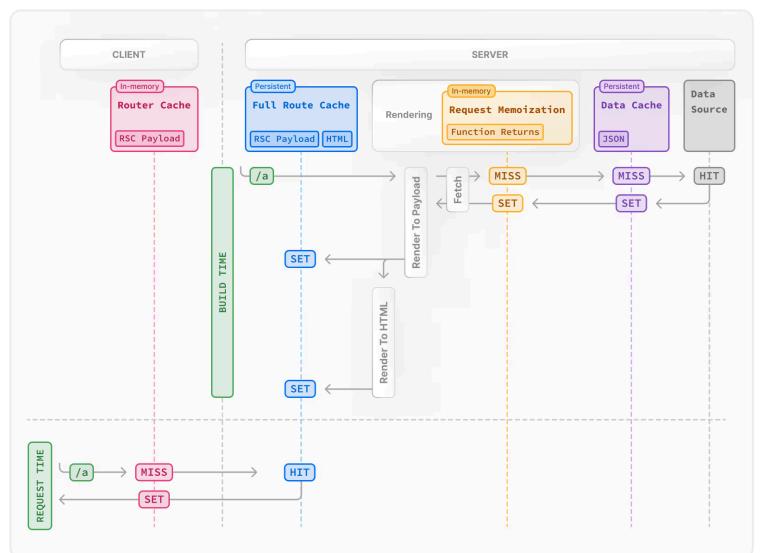
Overview

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 revalic)

Mechanism	What	Where	Purpose	Duration
Full Route Cache	HTML and RSC payload	Server	Reduce rendering cost and improve performance	Persistent (can be invalidated)
Router Cache	RSC Payload	Client	Reduce server requests on navigation	User session time-frame

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.



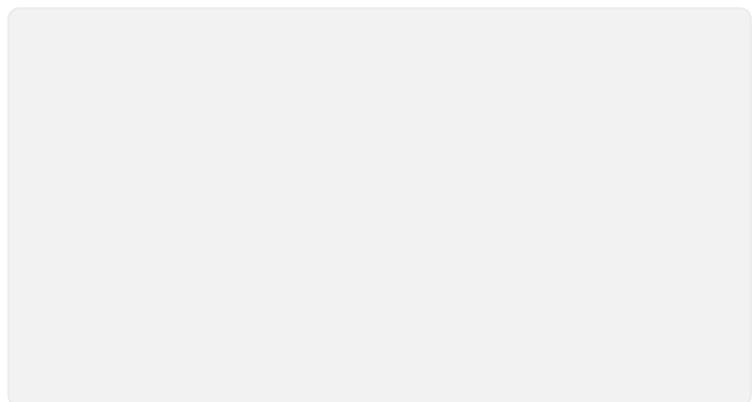
Caching behavior changes depending on whether the route is statically or dynamically rendered, data is cached or uncached, and whether a request is part of an initial visit or a subsequent navigation. Depending on your use case, you can configure the caching behavior for individual routes and data requests.

Fetch caching is **not** supported in `middleware`.

Any fetches done inside of your `middleware` will be uncached.

Request Memoization

Next.js 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.

```
TS app/example.tsx TypeScript ▾ ⌂
async function getItem() {
  // The `fetch` function is automatically me
  // is cached
  const res = await fetch('https://.../item/1')
  return res.json()
}
```

```
// This function is called twice, but only ex  
const item = await getItem() // cache MISS
```

```
// The second call could be anywhere in your  
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`](#) ↗.

JS app/example.js

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

Data Cache

Next.js has a built-in Data Cache that **persists** the result of data fetches across incoming **server requests** and **deployments**. This is possible because Next.js extends the native `fetch` API to allow each request on the server to set its own persistent caching semantics.

Good to know: In the browser, the `cache` option of `fetch` indicates how a request will interact with the browser's HTTP cache, in Next.js, the `cache` option indicates how a server-side request will interact with the server's Data Cache.

You can use the `cache` and `next.revalidate` options of `fetch` to configure the caching behavior.

In development mode, `fetch` data is **reused for Hot Module Replacement (HMR)**, and caching options are ignored for **hard refreshes**.

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.

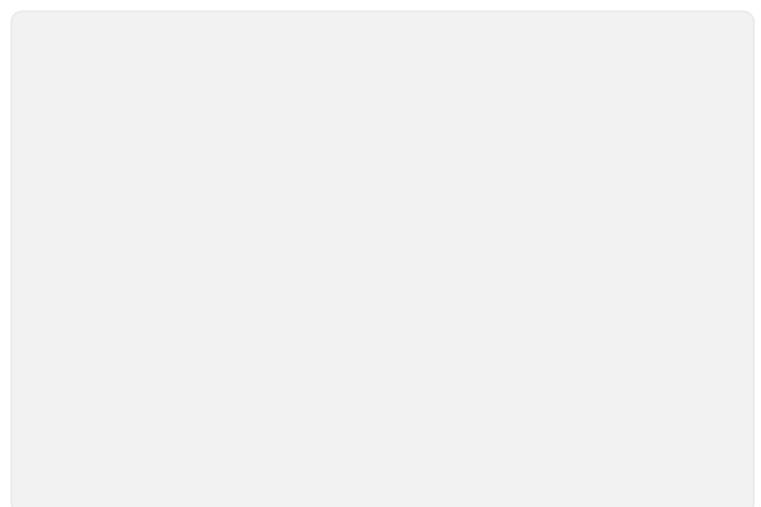
- Once the data is fetched successfully, Next.js will update the Data Cache with the fresh data.
- If the background revalidation fails, the previous data will be kept unaltered.

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.ap
```

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.

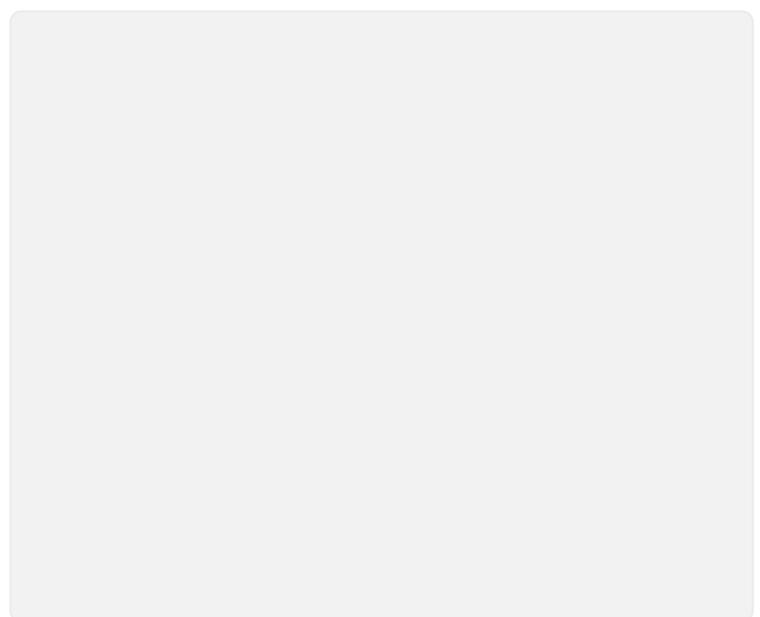
If the route segments are not in the cache, Next.js will fetch the React Server Components Payload

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 explicitly enable 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 browser state and React state in shared layouts.

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.

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	Route Cache	Dynamic Cache
<code><Link prefetch></code>	Cache		
<code>router.prefetch</code>	Cache		
<code>router.refresh</code>		Revalidate	
<code>fetch</code>		Cache	
<code>fetch options.cache</code>		Cache	
		Options	
<code>fetch</code>		Revalidate	Revalidate
<code>options.next.revalidate</code>		Cache	
<code>fetch</code>		Cache	
<code>options.next.tags</code>			
<code>revalidateTag</code>		Revalidate (Server Action)	Revalidate (Client Action)
<code>revalidatePath</code>		Revalidate (Server Action)	Revalidate (Client Action)
<code>const revalidate</code>		Revalidate or Opt out	Revalidate or Opt out

	Router API	Route Cache	Full Cache
<code>const dynamic</code>		Cache or Opt out	Cache or Opt out
<code>cookies</code>	Revalidate (Server Action)	Opt out	Opt out
<code>headers , searchParams</code>		Opt out	
<code>generateStaticParams</code>		Cache	
<code>React.cache</code>			
<code>unstable_cache</code>		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 *not* automatically cached in the Data Cache.

The default caching behavior of `fetch` (e.g., when the `cache` option is not specified) is equal to setting the `cache` option to `no-store`:

```
let data = await fetch('https://api.vercel.ap
```

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: 36 }}
```

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'] }}
```

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 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`:

```
JS app/blog/[slug]/page.js ⚙️

export async function generateStaticParams()
  const posts = await fetch('https://.../post')

  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:

```
JS app/blog/[slug]/page.js ⚙️

export async function generateStaticParams()
  const posts = await fetch('https://.../post')

  // 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' :
```

```
JS app/blog/[slug]/page.js ⚙️

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.

JS app/changelog/[slug]/page.js

```
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.

`fetch` requests using the `GET` or `HEAD` methods are automatically memoized, so you do not need to wrap it in React `cache`. However, for other `fetch` methods, or when using data fetching libraries (such as some database, CMS, or GraphQL clients) that don't inherently memoize requests, you can use `cache` to manually memoize data requests.

TS utils/get-item.ts

TypeScript ▾

```
import { cache } from 'react'
import db from '@/lib/db'
```

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

Was this helpful?  



 Using App Router
Features available in /app

 Latest Version
15.5.4



How to configure Continuous Integration (CI) build caching

To improve build performance, Next.js saves a cache to `.next/cache` that is shared between builds.

To take advantage of this cache in Continuous Integration (CI) environments, your CI workflow will need to be configured to correctly persist the cache between builds.

If your CI is not configured to persist `.next/cache` between builds, you may see a [No Cache Detected](#) error.

Here are some example cache configurations for common CI providers:

Vercel

Next.js caching is automatically configured for you. There's no action required on your part. If you are using Turborepo on Vercel, [learn more here ↗](#).

CircleCI

Edit your `save_cache` step in

`.circleci/config.yml` to include `.next/cache`:

```
steps:  
  - save_cache:  
    key: dependency-cache-{{ checksum "yarn  
    paths:  
      - ./node_modules  
      - ./next/cache
```

If you do not have a `save_cache` key, please follow CircleCI's [documentation on setting up build caching ↗](#).

Travis CI

Add or merge the following into your

`.travis.yml`:

```
cache:  
  directories:  
    - $HOME/.cache/yarn  
    - node_modules  
    - .next/cache
```

GitLab CI

Add or merge the following into your

`.gitlab-ci.yml`:

```
cache:  
  key: ${CI_COMMIT_REF_SLUG}  
  paths:  
    - node_modules/  
    - .next/cache/
```

Netlify CI

Use [Netlify Plugins ↗](#) with
[@netlify/plugin-nextjs ↗](#).

AWS CodeBuild

Add (or merge in) the following to your
`buildspec.yml`:

```
cache:  
  paths:  
    - 'node_modules/**/*' # Cache `node_modul  
    - '.next/cache/**/*' # Cache Next.js for
```

GitHub Actions

Using GitHub's [actions/cache ↗](#), add the following step in your workflow file:

```
uses: actions/cache@v4  
with:  
  # See here for caching with `yarn`, `bun` or  
  path: |  
    ~/.npm  
    ${{ github.workspace }}/.next/cache  
  # Generate a new cache whenever packages or  
  key: ${{ runner.os }}-nextjs-${{ hashFiles }}  
  # If source files changed but packages didn't  
  restore-keys: |  
    ${{ runner.os }}-nextjs-${{ hashFiles }}('**
```

Bitbucket Pipelines

Add or merge the following into your

`bitbucket-pipelines.yml` at the top level (same level as `pipelines`):

```
definitions:  
  caches:  
    nextcache: .next/cache
```

Then reference it in the `caches` section of your pipeline's `step`:

```
- step:  
  name: your_step_name  
  caches:  
    - node  
    - nextcache
```

Heroku

Using Heroku's [custom cache](#), add a `cacheDirectories` array in your top-level `package.json`:

```
"cacheDirectories": [".next/cache"]
```

Azure Pipelines

Using Azure Pipelines' [Cache task](#), add the following task to your pipeline yaml file somewhere prior to the task that executes `next build`:

```
- task: Cache@2  
  displayName: 'Cache .next/cache'
```

```
inputs:  
key: next | $(Agent.OS) | yarn.lock  
path: '$(System.DefaultWorkingDirectory)/'
```

Jenkins (Pipeline)

Using Jenkins' [Job Cacher](#) plugin, add the following build step to your `Jenkinsfile` where you would normally run `next build` or `npm install`:

```
stage("Restore npm packages") {  
    steps {  
        // Writes lock-file to cache based on  
        writeFile file: "next-lock.cache", te  
  
        cache(caches: [  
            arbitraryFileCache(  
                path: "node_modules",  
                includes: "**/*",  
                cacheValidityDecidingFile: "p  
            )  
        ]) {  
            sh "npm install"  
        }  
    }  
}  
stage("Build") {  
    steps {  
        // Writes lock-file to cache based on  
        writeFile file: "next-lock.cache", te  
  
        cache(caches: [  
            arbitraryFileCache(  
                path: ".next/cache",  
                includes: "**/*",  
                cacheValidityDecidingFile: "n  
            )  
        ]) {  
            // aka `next build`  
            sh "npm run build"  
        }  
    }  
}
```

Was this helpful?  





Using App Router

Features available in /app



Latest Version

15.5.4



How to set a Content Security Policy (CSP) for your Next.js application

[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?

CSP can block both inline and external scripts to prevent attacks. A nonce lets you safely allow specific scripts to run—only if they include the matching nonce value.

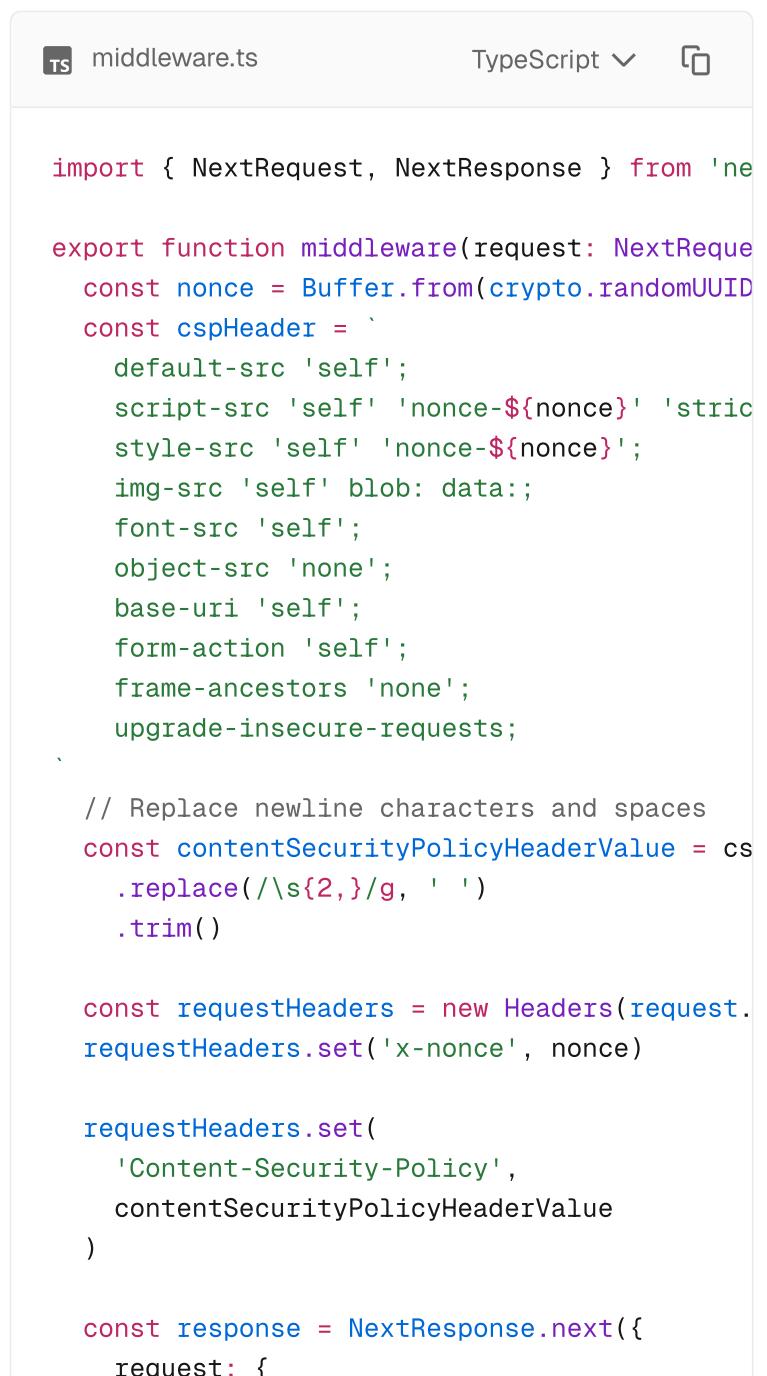
If an attacker wanted to load a script into your page, they'd need to guess the nonce value. That's why the nonce must be unpredictable and unique for every request.

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:



```
TS middleware.ts TypeScript ▾
```

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

export function middleware(request: NextRequest) {
  const nonce = Buffer.from(crypto.randomUUID())
  const cspHeader = `

    default-src 'self';
    script-src 'self' 'nonce-${nonce}' 'strictDynamic';
    style-src 'self' 'nonce-${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 = cs
    .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: {
        'Content-Security-Policy': contentSecurityPolicyHeaderValue,
        'X-Nonce': nonce.toString('hex'),
      },
    },
  })
}
```

```

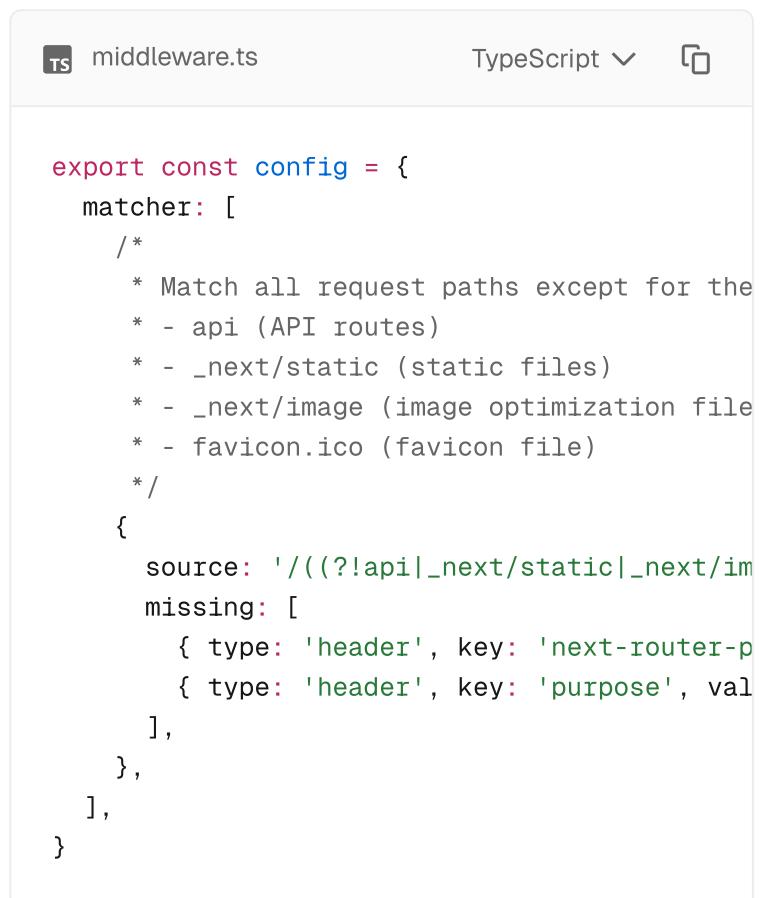
        headers: requestHeaders,
    },
})
response.headers.set(
    'Content-Security-Policy',
    contentSecurityPolicyHeaderValue
)

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.



```

TS middleware.ts TypeScript ▾
export const config = {
  matcher: [
    /*
     * Match all request paths except for the
     * - api (API routes)
     * - _next/static (static files)
     * - _next/image (image optimization file)
     * - favicon.ico (favicon file)
    */
    {
      source: '/((?!api|_next/static|_next/im|missing: [
        { type: 'header', key: 'next-router-p
        { type: 'header', key: 'purpose', val
      ],
    },
    ],
  ],
}

```

How nonces work in Next.js

To use a nonce, your page must be **dynamically rendered**. This is because Next.js applies nonces during **server-side rendering**, based on the CSP header present in the request. Static pages are

generated at build time, when no request or response headers exist—so no nonce can be injected.

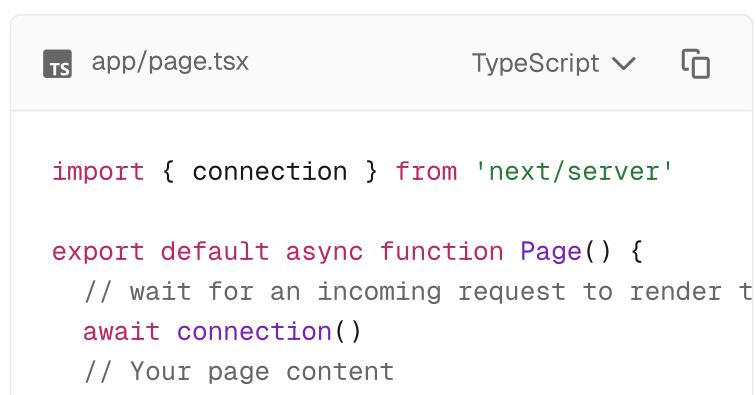
Here's how nonce support works in a dynamically rendered page:

1. **Middleware generates a nonce:** Your middleware creates a unique nonce for the request, adds it to your `Content-Security-Policy` header, and also sets it in a custom `x-nonce` header.
2. **Next.js extracts the nonce:** During rendering, Next.js parses the `Content-Security-Policy` header and extracts the nonce using the `'nonce-{value}'` pattern.
3. **Nonce is applied automatically:** Next.js attaches the nonce to:
 - 4. Framework scripts (React, Next.js runtime)
 - 5. Page-specific JavaScript bundles
 - 6. Inline styles and scripts generated by Next.js
 - 7. Any `<Script>` components using the `nonce` prop

Because of this automatic behavior, you don't need to manually add a nonce to each tag.

Forcing dynamic rendering

If you're using nonces, you may need to explicitly opt pages into dynamic rendering:



```
TS app/page.tsx TypeScript ▾ ⌂

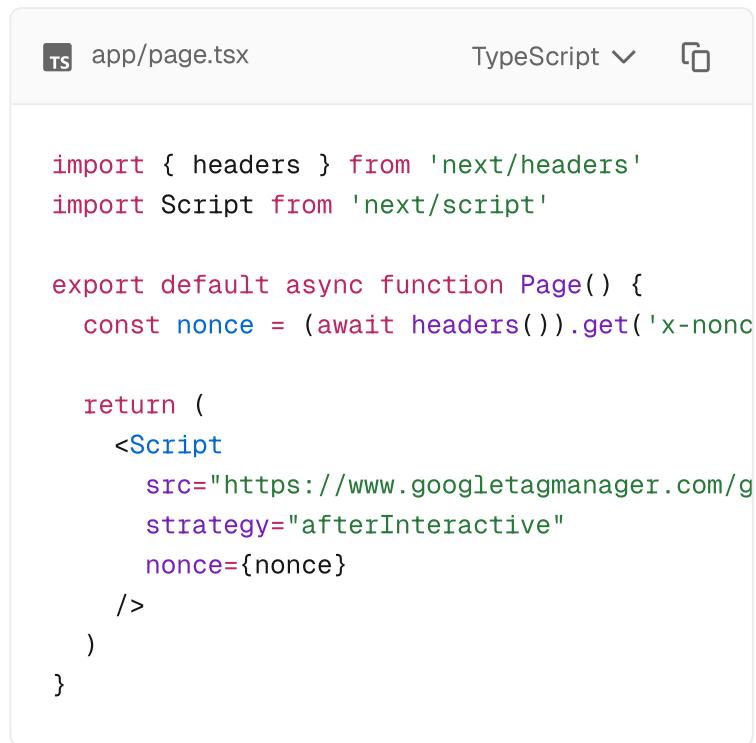
import { connection } from 'next/server'

export default async function Page() {
  // wait for an incoming request to render t
  await connection()
  // Your page content
```

```
}
```

Reading the nonce

You can read the nonce from a [Server Component](#) using `headers`:



```
TS app/page.tsx TypeScript ▾
```

```
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/g
      strategy="afterInteractive"
      nonce={nonce}
    />
  )
}
```

Static vs Dynamic Rendering with CSP

Using nonces has important implications for how your Next.js application renders:

Dynamic Rendering Requirement

When you use nonces in your CSP, **all pages must be dynamically rendered**. This means:

- Pages will build successfully but may encounter runtime errors if not properly configured for dynamic rendering
- Each request generates a fresh page with a new nonce

- Static optimization and Incremental Static Regeneration (ISR) are disabled
- Pages cannot be cached by CDNs without additional configuration
- **Partial Prerendering (PPR) is incompatible** with nonce-based CSP since static shell scripts won't have access to the nonce

Performance Implications

The shift from static to dynamic rendering affects performance:

- **Slower initial page loads:** Pages must be generated on each request
- **Increased server load:** Every request requires server-side rendering
- **No CDN caching:** Dynamic pages cannot be cached at the edge by default
- **Higher hosting costs:** More server resources needed for dynamic rendering

When to use nonces

Consider nonces when:

- You have strict security requirements that prohibit '`'unsafe-inline'`'
- Your application handles sensitive data
- You need to allow specific inline scripts while blocking others
- Compliance requirements mandate strict CSP

Without Nonces

For applications that do not require nonces, you can set the CSP header directly in your [next.config.js](#) file:



```
JS next.config.js

const cspHeader = `

default-src 'self';
script-src 'self' 'unsafe-eval' 'unsafe-i
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,
              ),
          },
        ],
      },
    ],
  },
}
```

Subresource Integrity (Experimental)

As an alternative to nonces, Next.js offers experimental support for hash-based CSP using Subresource Integrity (SRI). This approach allows you to maintain static generation while still having a strict CSP.

Good to know: This feature is experimental and only available with webpack bundler in App Router applications.

How SRI works

Instead of using nonces, SRI generates cryptographic hashes of your JavaScript files at build time. These hashes are added as `integrity` attributes to script tags, allowing browsers to verify that files haven't been modified during transit.

Enabling SRI

Add the experimental SRI configuration to your `next.config.js`:

```
JS next.config.js Copy

/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    sri: {
      algorithm: 'sha256', // or 'sha384' or
    },
  },
}

module.exports = nextConfig
```

CSP configuration with SRI

When SRI is enabled, you can continue using your existing CSP policies. SRI works independently by adding `integrity` attributes to your assets:

Good to know: For dynamic rendering scenarios, you can still generate nonces with middleware if needed, combining both SRI integrity attributes and nonce-based CSP approaches.

```
const cspHeader = `
  default-src 'self';
  script-src 'self';
  style-src 'self';
  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 = {
  experimental: {
    sri: {
      algorithm: 'sha256',
    },
  },
  async headers() {
    return [
      {
        source: '/(.*)',
        headers: [
          {
            key: 'Content-Security-Policy',
            value: cspHeader.replace(/\n/g, ''),
          },
        ],
      },
    ],
  },
}
```

Benefits of SRI over nonces

- **Static generation:** Pages can be statically generated and cached
- **CDN compatibility:** Static pages work with CDN caching
- **Better performance:** No server-side rendering required for each request
- **Build-time security:** Hashes are generated at build time, ensuring integrity

Limitations of SRI

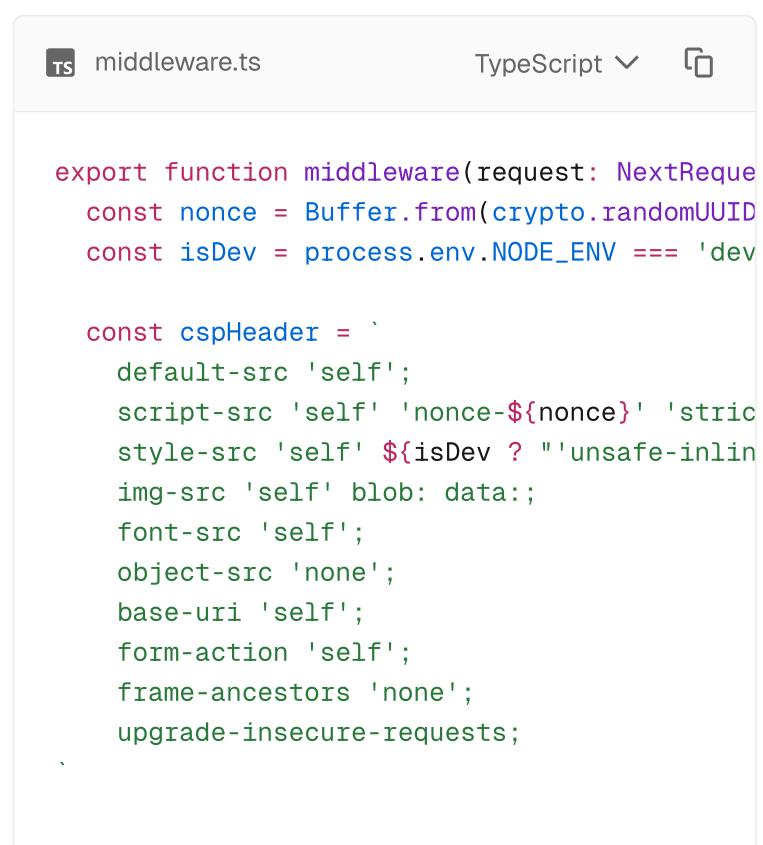
- **Experimental:** Feature may change or be removed
 - **Webpack only:** Not available with Turbopack
 - **App Router only:** Not supported in Pages Router
 - **Build-time only:** Cannot handle dynamically generated scripts
-

Development vs Production Considerations

CSP implementation differs between development and production environments:

Development Environment

In development, you will need to enable `'unsafe-eval'` to support APIs that provide additional debugging information:



The screenshot shows a code editor window with a TypeScript file named `middleware.ts`. The file contains the following code:

```
export function middleware(request: NextRequest) {
  const nonce = Buffer.from(crypto.randomUUID());
  const isDev = process.env.NODE_ENV === 'development';

  const cspHeader = `;
    default-src 'self';
    script-src 'self' 'nonce-${nonce}' 'strictDynamic';
    style-src 'self' ${isDev ? "'unsafe-inline'" : "'self'"} 'self';
    img-src 'self' blob: data:;
    font-src 'self';
    object-src 'none';
    base-uri 'self';
    form-action 'self';
    frame-ancestors 'none';
    upgrade-insecure-requests;
  `;
}
```

```
// Rest of middleware implementation  
}
```

Production Deployment

Common issues in production:

- **Nonce not applied:** Ensure your middleware runs on all necessary routes
- **Static assets blocked:** Verify your CSP allows Next.js static assets
- **Third-party scripts:** Add necessary domains to your CSP policy

Troubleshooting

Third-party Scripts

When using third-party scripts with CSP:

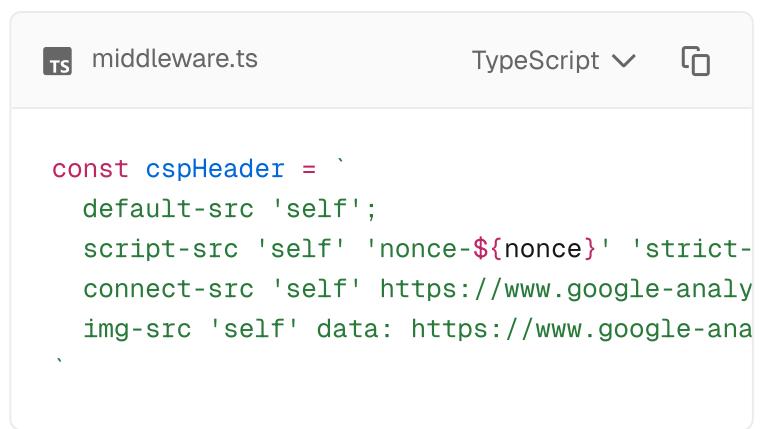


The screenshot shows a code editor window with the following details:

- File name: app/layout.tsx
- TypeScript version: v4.5
- Content:

```
TS  app/layout.tsx  TypeScript  □  
  
import { GoogleTagManager } from '@next/third  
import { headers } from 'next/headers'  
  
export default async function RootLayout({  
    children,  
}: {  
    children: React.ReactNode  
}) {  
    const nonce = (await headers()).get('x-nonce')  
  
    return (  
        <html lang="en">  
            <body>  
                {children}  
                <GoogleTagManager gtmId="GTM-XYZ" nonce={nonce}>  
            </body>  
        </html>  
    )  
}
```

Update your CSP to allow third-party domains:



A screenshot of a code editor window titled "middleware.ts". The file contains TypeScript code for generating a Content Security Policy (CSP) header. The code defines a constant `cspHeader` with a multi-line string value containing directives like `default-src`, `script-src`, `connect-src`, and `img-src`. The code editor interface includes tabs for "TypeScript" and a save icon.

```
const cspHeader = `default-src 'self';script-src 'self' 'nonce-${nonce}' 'strict-connect-src 'self' https://www.google-analy if using WebAssembly- 4. Service workers: Add appropriate policies for service worker scripts

```

Version History

Version	Changes
v14.0.0	Experimental SRI support added for hash-based CSP
v13.4.20	Recommended for proper nonce handling and CSP header parsing.

Next Steps

middleware.js

API reference for the middleware.js file.

headers

API reference for the headers function.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

How to use CSS-in-JS libraries

Warning: 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 ↗.

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.

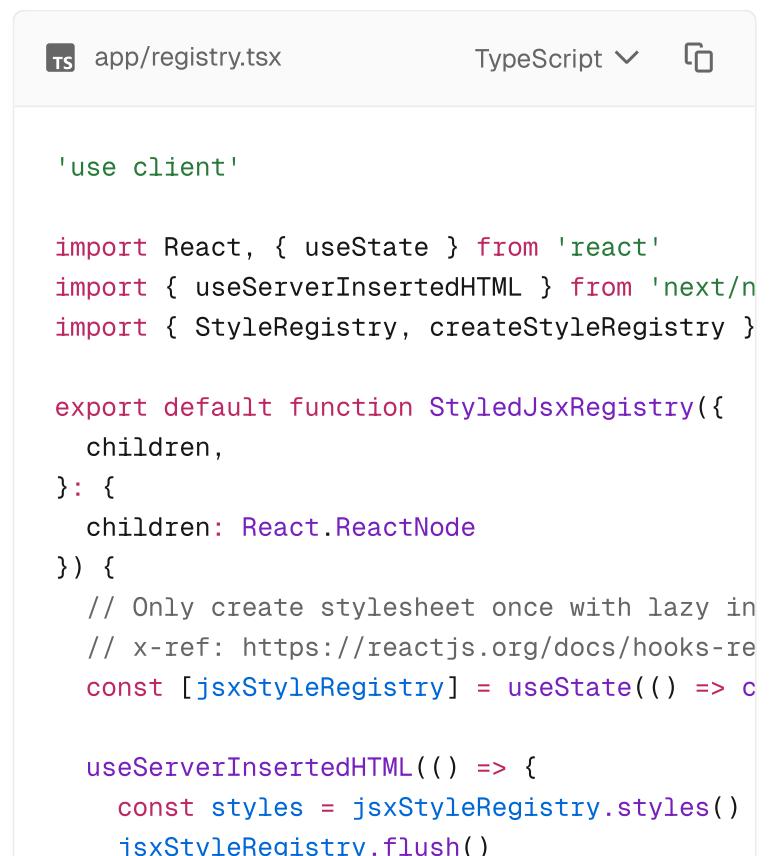
Configuring CSS-in-JS in `app`

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

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

`styled-jsx`

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



```
TS app/registry.tsx TypeScript ▾ ⌂

'use client'

import React, { useState } from 'react'
import { useServerInsertedHTML } from 'next/n
import { StyleRegistry, createStyleRegistry } }

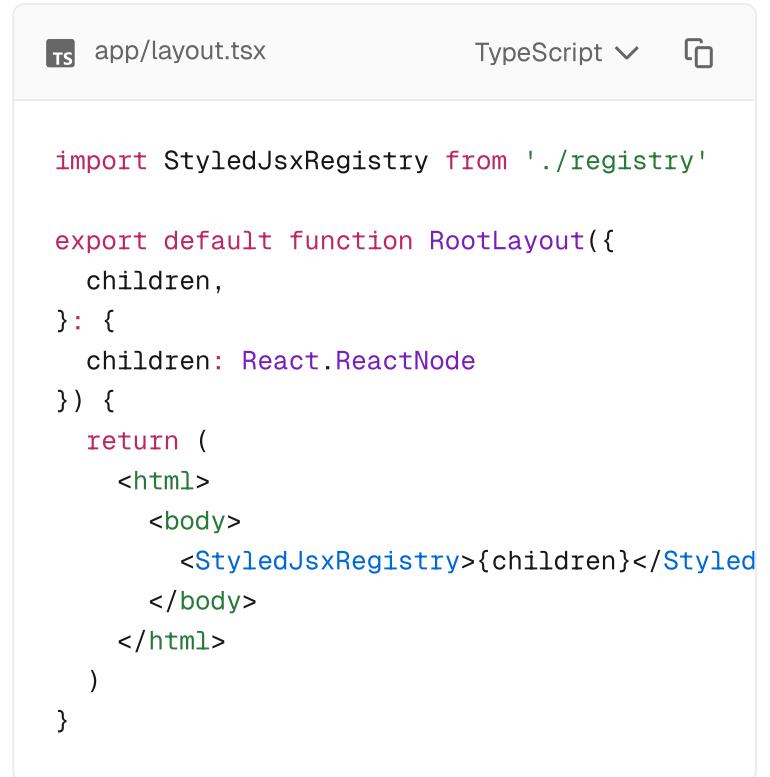
export default function StyledJsxRegistry({ children, }: { children: React.ReactNode }) {
  // Only create stylesheet once with lazy in
  // x-ref: https://reactjs.org/docs/hooks-re
  const [jsxStyleRegistry] = useState(() => c

    useServerInsertedHTML(() => {
      const styles = jsxStyleRegistry.styles()
      jsxStyleRegistry.flush()
    })
  )
}
```

```
        return <>{styles}</>
    })

    return <StyleRegistry registry={jsxStyleReg
}
```

Then, wrap your `root layout` with the registry:



```
TS app/layout.tsx TypeScript ▾ ⌂

import StyledJsxRegistry from './registry'

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



```
JS 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.



The screenshot shows a code editor window with the file `lib/registry.tsx` open. The code is written in TypeScript and uses the `styled-components` library. It defines a function `StyledComponentsRegistry` that takes `children` as a prop. Inside the function, it uses the `useServerInsertedHTML` hook to inject styles into the head of the document. It also creates a `styledComponentsStyleSheet` and uses it to clear styles before injecting them. The code then returns a `StyleSheetManager` component with the `children` prop. The code editor has a TypeScript dropdown menu and a copy icon.

```
'use client'

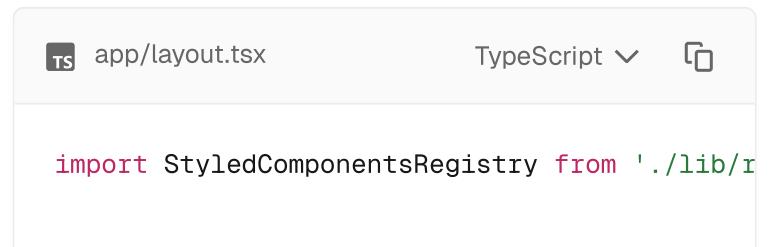
import React, { useState } from 'react'
import { useServerInsertedHTML } from 'next/n
import { ServerStyleSheet, StyleSheetManager

export default function StyledComponentsRegis
    children,
}: {
    children: React.ReactNode
}) {
    // Only create stylesheet once with lazy in
    // x-ref: https://reactjs.org/docs/hooks-re
    const [styledComponentsStyleSheet] = useState(
        useServerInsertedHTML(() => {
            const styles = styledComponentsStyleSheet
            styledComponentsStyleSheet.instance.clear
            return <>{styles}</>
        })
    )

    if (typeof window !== 'undefined') return <

    return (
        <StyleSheetManager sheet={styledComponent
            {children}
        </StyleSheetManager>
    )
}
}
```

Wrap the `children` of the root layout with the style registry component:



The screenshot shows a code editor window with the file `app/layout.tsx` open. The code imports `StyledComponentsRegistry` from the `lib/registry` module and wraps the `children` prop of the layout component with it. The code editor has a TypeScript dropdown menu and a copy icon.

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

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

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

How to set up a custom server in Next.js

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](#).
- 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:

```
ts server.ts TypeScript ▾   
  
import { createServer } from 'http'  
import { parse } from 'url'  
import next from 'next'  
  
const port = parseInt(process.env.PORT || '30  
const dev = process.env.NODE_ENV !== 'product  
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:${
      dev ? 'development' : process.env.NODE_
    }`
  )
})
```

`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:



A screenshot of a code editor showing a file named "package.json". The file contains the following JSON code:

```
{
  "scripts": {
    "dev": "node server.js",
    "build": "next build",
    "start": "NODE_ENV=production node server"
  }
}
```

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 <code>next.config.js</code> . Defaults to <code>{}</code>
dev	Boolean	(Optional) Whether or not to launch Next.js in dev mode. Defaults to <code>false</code>
dir	String	(Optional) Location of the Next.js project. Defaults to <code>'.'</code>
quiet	Boolean	(Optional) Hide error messages containing server information. Defaults to <code>false</code>
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.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



How to think about data security in Next.js

[React Server Components ↗](#) improve performance and simplify data fetching, but also shift where and how data is accessed, changing some of the traditional security assumptions for handling data in frontend apps.

This guide will help you understand how to think about data security in Next.js and how to implement best practices.

Data fetching approaches

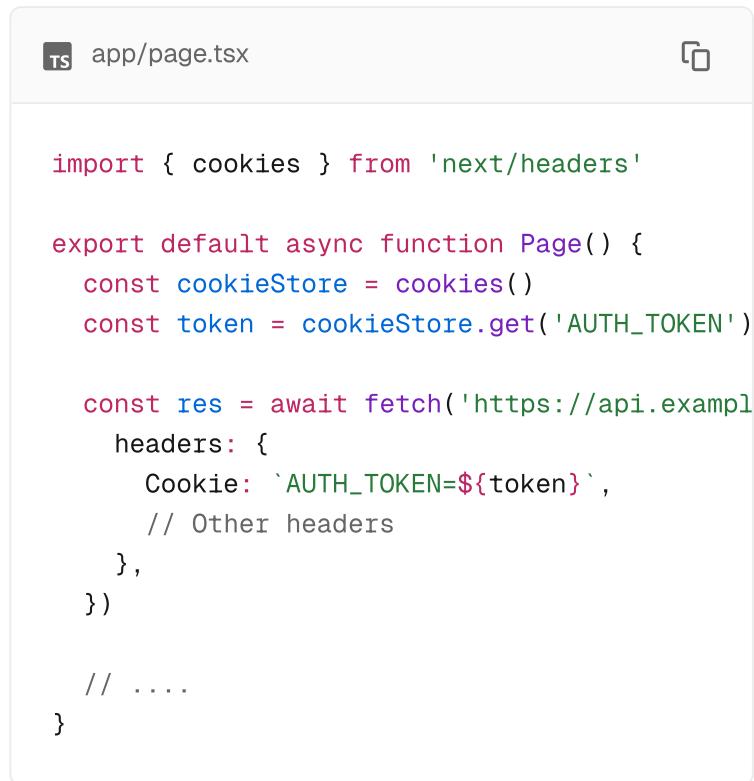
There are three main approaches we recommend for fetching data in Next.js, depending on the size and age of your project:

- [HTTP APIs](#): for existing large applications and organizations.
- [Data Access Layer](#): for new projects.
- [Component-Level Data Access](#): for prototypes and learning.

We recommend choosing one data fetching approach and avoiding mixing them. This makes it clear for both developers working in your code base and security auditors what to expect.

External HTTP APIs

You should follow a **Zero Trust** model when adopting Server Components in an existing project. You can continue calling your existing API endpoints such as REST or GraphQL from Server Components using `fetch`, just as you would in Client Components.



```
TS app/page.tsx

import { cookies } from 'next/headers'

export default async function Page() {
  const cookieStore = cookies()
  const token = cookieStore.get('AUTH_TOKEN')

  const res = await fetch('https://api.example.com')
    .headers({
      'Cookie': `AUTH_TOKEN=${token}`,
      // Other headers
    })
    .then((r) => r.json())
    .catch((e) => console.error(e))

  // ...
}
```

This approach works well when:

- You already have security practices in place.
- Separate backend teams use other languages or manage APIs independently.

Data Access Layer

For new projects, we recommend creating a dedicated **Data Access Layer (DAL)**. This is a internal library that controls how and when data is fetched, and what gets passed to your render context.

A Data Access Layer should:

- Only run on the server.

- Perform authorization checks.
- Return safe, minimal **Data Transfer Objects** (**DTOs**).

This approach centralizes all data access logic, making it easier to enforce consistent data access and reduces the risk of authorization bugs. You also get the benefit of sharing an in-memory cache across different parts of a request.

TS data/auth.ts

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

// Cached helper methods makes it easy to get
// without manually passing it around. This d
// Component to Server Component which minimi
// Component.
export const getCurrentUser = cache(async () {
  const token = cookies().get('AUTH_TOKEN')
  const decodedToken = await decryptAndValida
  // Don't include secret tokens or private i
  // Use classes to avoid accidentally passin
  return new User(decodedToken.id)
})
```

TS data/user-dto.tsx

```
import 'server-only'
import { getCurrentUser } from './auth'

function canSeeUsername(viewer: User) {
  // Public info for now, but can change
  return true
}

function canSeePhoneNumber(viewer: User, team
  // Privacy rules
  return viewer.isAdmin || team === viewer.te
}

export async function getProfileDTO(slug: str
  // Don't pass values, read back cached valu
  // use a database API that supports safe te
  const [rows] = await sql`SELECT * FROM user
```

```
const userData = rows[0]

const currentUser = await getCurrentUser()

// only return the data relevant for this q
// <https://www.w3.org/2001/tag/doc/APIMinis
return {
  username: canSeeUsername(currentUser) ? u
  phononenumber: canSeePhoneNumber(currentUser)
    ? userData.phonenumber
    : null,
}
}
```

TS app/page.tsx



```
import { getProfile } from '../data/user'

export async function Page({ params: { slug } })
  // This page can now safely pass around thi
  // that it shouldn't contain anything sensi
  const profile = await getProfile(slug);
  ...
}
```

Good to know: Secret keys should be stored in environment variables, but only the Data Access Layer should access `process.env`. This keeps secrets from being exposed to other parts of the application.

Component-level data access

For quick prototypes and iteration, database queries can be placed directly in Server Components.

This approach, however, makes it easier to accidentally expose private data to the client, for example:

TS app/page.tsx



```
import Profile from './components/profile.tsx'
```

```
export async function Page({ params: { slug } })
  const [rows] = await sql`SELECT * FROM user
  const userData = rows[0]
  // EXPOSED: This exposes all the fields in
  // we are passing the data from the Server
  return <Profile user={userData} />
}
```

TS app/ui/profile.tsx

```
'use client'
```

```
// BAD: This is a bad props interface because
// Client Component needs and it encourages s
// data down. A better solution would be to a
// the fields necessary for rendering the pro
export default async function Profile({ user
  return (
    <div>
      <h1>{user.name}</h1>
      ...
    </div>
  )
}
```

You should sanitize the data before passing it to
the Client Component:

TS data/user.ts

```
import { sql } from './db'

export async function getUser(slug: string) {
  const [rows] = await sql`SELECT * FROM user
  const user = rows[0]

  // Return only the public fields
  return {
    name: user.name,
  }
}
```

TS app/page.tsx

```
import { getUser } from '../data/user'
import Profile from './ui/profile'
```

```
export default async function Page({  
  params: { slug },  
}: {  
  params: { slug: string }  
) {  
  const publicProfile = await getUser(slug)  
  return <Profile user={publicProfile} />  
}
```

Reading data

Passing data from server to client

On the initial load, both Server and Client Components run on the server to generate HTML. However, they execute in isolated module systems. This ensures that Server Components can access private data and APIs, while Client Components cannot.

Server Components:

- Run only on the server.
- Can safely access environment variables, secrets, databases, and internal APIs.

Client Components:

- Run on the server during pre-rendering, but must follow the same security assumptions as code running in the browser.
- Must not access privileged data or server-only modules.

This ensures the app is secure by default, but it's possible to accidentally expose private data through how data is fetched or passed to components.

Tainting

To prevent accidental exposure of private data to the client, you can use React Taint APIs:

- [experimental_taintObjectReference](#) ↗ for data objects.
- [experimental_taintUniqueValue](#) ↗ for specific values.

You can enable usage in your Next.js app with the [experimental.taint](#) option in `next.config.js`:



```
next.config.js
```

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

This prevents the tainted objects or values from being passed to the client. However, it's an additional layer of protection, you should still filter and sanitize the data in your [DAL](#) before passing it to React's render context.

Good to know:

- By default, environment variables are only available on the Server. Next.js exposes any environment variable prefixed with `NEXT_PUBLIC_` to the client. [Learn more](#).
- Functions and classes are already blocked from being passed to Client Components by default.

Preventing client-side execution of server-only code

To prevent server-only code from being executed on the client, you can mark a module with the [server-only](#) ↗ package:

pnpm npm yarn bun

>_ Terminal



```
pnpm add server-only
```

TS lib/data.ts



```
import 'server-only'
```

```
// ...
```

This ensures that proprietary code or internal business logic stays on the server by causing a build error if the module is imported in the client environment.

Mutating Data

Next.js handles mutations with [Server Actions ↗](#).

Built-in Server Actions Security features

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 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.

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'

// If this action **is** used in our application
// will create a secure ID to allow the client
// and call the Server Action.
export async function updateUserAction(formData)

// If this action **is not** used in our application
// will automatically remove this code during
// and will not create a public endpoint.
export async function deleteUserAction(formData)
```

Validating client input

You should always validate input from client, as they can be easily modified. For example, form data, URL parameters, headers, and searchParams:

TS app/page.tsx

```
// BAD: Trusting searchParams directly
export default async function Page({ searchParams }) {
  const isAdmin = searchParams.get('isAdmin')
  if (isAdmin === 'true') {
    // Vulnerable: relies on untrusted client
    return <AdminPanel />
  }
}
```

```
}
```

```
// GOOD: Re-verify every time
import { cookies } from 'next/headers'
import { verifyAdmin } from './auth'

export default async function Page() {
  const token = cookies().get('AUTH_TOKEN')
  const isAdmin = await verifyAdmin(token)

  if (isAdmin) {
    return <AdminPanel />
  }
}
```

Authentication and authorization

You should always ensure that a user is authorized to perform an action. For example:



```
TS app/actions.ts

'use server'

import { auth } from './lib'

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

  // ...
}
```

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

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:

```
TS app/page.tsx
```

```
export default async function Page() {
  const publishVersion = await getLatestVersi

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

  return (
    <form>
      <button formAction={publish}>Publish</b
    </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.

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 variable **must** be AES-GCM encrypted.

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.

```
JS next.config.js

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

Learn more about [Security and Server Actions](#).

Avoiding side-effects during rendering

Mutations (e.g. logging out users, updating databases, invalidating caches) should never be a side-effect, either in Server or Client Components. Next.js explicitly prevents setting cookies or triggering cache revalidation within render methods to avoid unintended side effects.

```
TS app/page.tsx

// BAD: Triggering a mutation during rendering
export default async function Page({ searchParams }) {
  if (searchParams.get('logout')) {
    cookies().delete('AUTH_TOKEN')
  }

  return <UserProfile />
}
```

Instead, you should use Server Actions to handle mutations.

```
TS app/page.tsx
```

```
// GOOD: Using Server Actions to handle mutations
import { logout } from './actions'

export default function Page() {
  return (
    <>
      <UserProfile />
      <form action={logout}>
        <button type="submit">Logout</button>
      </form>
    </>
  )
}
```

Good to know: Next.js uses `POST` requests to handle mutations. This prevents accidental side-effects from GET requests, reducing Cross-Site Request Forgery (CSRF) risks.

Auditing

If you're doing an audit of a Next.js project, here are a few things we recommend looking extra at:

- **Data Access Layer:** Is there an established practice for an isolated Data Access Layer? Verify that database packages and environment variables are not imported outside the Data Access Layer.
- **"use client" files:** Are the Component props expecting private data? Are the type signatures overly broad?
- **"use server" files:** Are the Action arguments validated in the action or inside the Data Access Layer? Is the user re-authorized inside the action?
- **/[param]/.** Folders with brackets are user input. Are params validated?

- `middleware.ts` and `route.ts`: Have a lot of power. Spend extra time auditing these using traditional techniques. Perform Penetration Testing or Vulnerability Scanning regularly or in alignment with your team's software development lifecycle.
-

Next Steps

Learn more about the topics mentioned in this guide.

Authenticati...

Learn how to implement authentication in...

Content Sec...

Learn how to set a Content Security Policy (CSP) for...

Forms

Learn how to create forms in Next.js with Reac...

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4



How to use debugging tools with Next.js

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 (Fi
      "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_mod
      "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.

In the "Next.js: debug full stack" configuration, `serverReadyAction.action` specifies which browser to open when the server is ready. `debugWithEdge` means to launch the Edge browser. If you are using Chrome, change this value to `debugWithChrome`.

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/_/`.

React Developer Tools

For React-specific debugging, install the [React Developer Tools](#) [↗] browser extension. This essential tool helps you:

- Inspect React components

- Edit props and state
- Identify performance problems

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:

```
>_ Terminal
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`:

```
JSON package.json
{
  "scripts": {
    "dev": "NODE_OPTIONS='--inspect' next dev"
  }
}
```

Launching the Next.js dev server with the `--inspect` flag will look something like this:

```
>_ Terminal
Debugger listening on ws://127.0.0.1:9229/0cf
For help, see: https://nodejs.org/en/docs/ins
ready - started server on 0.0.0.0:3000, url:
```

For Chrome:

1. Open a new tab and visit `chrome://inspect`
2. Click **Configure...** to ensure both debugging ports are listed
3. Add both `localhost:9229` and `localhost:9230` if they're not already present
4. Look for your Next.js application in the **Remote Target** section
5. Click **inspect** to open a separate DevTools window
6. 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 icon underneath the Next.js version indicator on the error overlay. By clicking that icon, 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.



```
package.json
```

```
{  
  "scripts": {  
    "dev": "cross-env NODE_OPTIONS='--inspect'  
  }  
}
```

`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 ↗](#)

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



How to preview content with Draft Mode in Next.js

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.

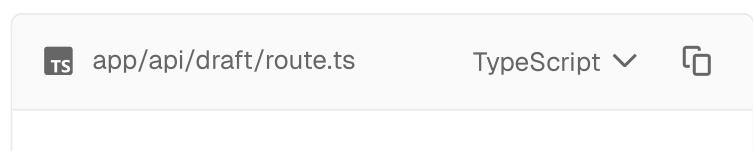
Step 1: Create a Route Handler

Create a [Route Handler](#). It can have any name, for example, `app/api/draft/route.ts`.



```
TS app/api/draft/route.ts TypeScript ▾ ⌂
export async function GET(request: Request) {
  return new Response('')
}
```

Then, import the `draftMode` function and call the `enable()` method.



```
TS app/api/draft/route.ts TypeScript ▾ ⌂
import { draftMode } from 'next/server';
draftMode.enable();
```

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

export async function GET(request: 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:

```
>_ Terminal ✖  
  
https://<your-site>/api/draft?secret=<token>&
```

- <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}

1. 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 :

```
TS app/api/draft/route.ts TypeScript ✖  
  
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  
    if (secret !== 'MY_SECRET_TOKEN' || !slug)  
        return new Response('Invalid token', { status: 401 })  
  
    // Fetch the headless CMS to check if the post exists  
    // getPostBySlug would implement the required logic  
    // ...  
}
```

```
const post = awaitgetPostBySlug(slug)

// If the slug doesn't exist prevent draft
if (!post) {
  return new Response('Invalid slug', { status: 404 })
}

// 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 it's handled by the cookie
redirect(post.slug)
}
```

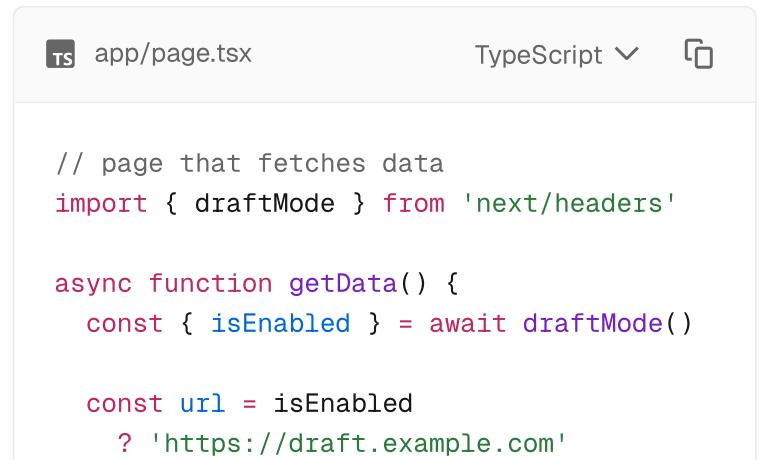
If it succeeds, then the browser will be redirected to the path you want to view with the draft mode cookie.

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`.



```
TS app/page.tsx TypeScript ▾
```

```
// 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.

Next Steps

See the API reference for more information on how to use Draft Mode.

draftMode

API Reference for the `draftMode` function.

Was this helpful?



 Using App Router
Features available in /app Latest Version
15.5.4

How to use environment variables in Next.js

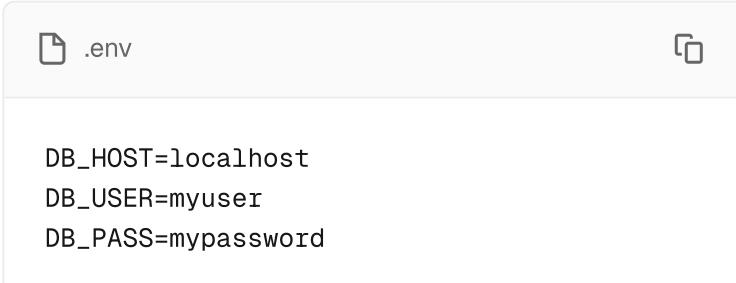
Next.js comes with built-in support for environment variables, which allows you to do the following:

- Use `.env` to load environment variables
- Bundle environment variables for the browser by prefixing with `NEXT_PUBLIC_`

Warning: The default `create-next-app` template ensures all `.env` files are added to your `.gitignore`. You almost never want to commit these files to your repository.

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
```

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-----"
```

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:

JS app/api/route.js

```
export async function GET() {
  const db = await myDB.connect({
    host: process.env.DB_HOST,
    username: process.env.DB_USER,
    password: process.env.DB_PASS,
  })
  // ...
}
```

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
```

envConfig.ts

TypeScript

```
import { loadEnvConfig } from '@next/env'

const projectDir = process.cwd()
loadEnvConfig(projectDir)
```

Then, you can import the configuration where needed. For example:

orm.config.ts

TypeScript

```
import './envConfig.ts'
```

```
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:

.env

```
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:

> Terminal



```
NEXT_PUBLIC_ANALYTICS_ID=abcdefghijklm
```

This will tell Next.js to replace all references to `process.env.NEXT_PUBLIC_ANALYTICS_ID` in the Node.js environment with the value from the environment in which you run `next build`, allowing you to use it anywhere in your code. It will be inlined into any JavaScript sent to the browser.

Note: After being built, your app will no longer respond to changes to these environment variables. For instance, if you use a Heroku pipeline to promote

slugs built in one environment to another environment, or if you build and deploy a single Docker image to multiple environments, all

`NEXT_PUBLIC_` variables will be frozen with the value evaluated at build time, so these values need to be set appropriately when the project is built. If you need access to runtime environment values, you'll have to setup your own API to provide them to the client (either on demand or during initialization).

JS pages/index.js



```
import setupAnalyticsService from '../lib/my-  
  
// 'NEXT_PUBLIC_ANALYTICS_ID' can be used here  
// It will be transformed at build time to `s  
setupAnalyticsService(process.env.NEXT_PUBLIC  
  
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  
const varName = 'NEXT_PUBLIC_ANALYTICS_ID'  
setupAnalyticsService(process.env[varName])  
  
// This will NOT be inlined, because it uses  
const env = process.env  
setupAnalyticsService(env.NEXT_PUBLIC_ANALYTIC
```

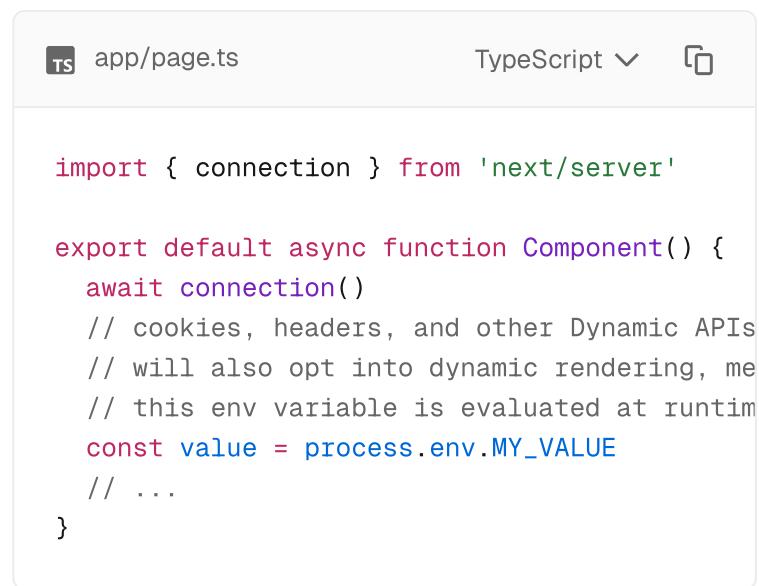
Runtime Environment Variables

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`.

You can safely read environment variables on the server during dynamic rendering:



```
app/page.ts  TypeScript ▾
```

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

export default async function Component() {
  await connection()
  // cookies, headers, and other Dynamic APIs
  // will also opt into dynamic rendering, me
  // 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 `runtimesConfig` option, as this does not work with the standalone output mode. Instead, we recommend [incrementally adopting](#) the App Router if you need this feature.

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
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.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

How to create forms with Server Actions

React Server Actions are [Server Functions](#) that execute on the server. They can be called in Server and Client Components to handle form submissions. This guide will walk you through how to create forms in Next.js with Server Actions.

How it works

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

When used in a form, the function automatically receives the `FormData` object. You can then extract the data using the native `FormData` methods :

```
TS app/invoices/page.tsx TypeScript ▾ ⌂  
  
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 the cache  
  }  
}
```

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

Good to know: When working with forms that have multiple fields, you can use the `entries()` method with JavaScript's `Object.fromEntries()`. For example:

```
const rawFormData =
Object.fromEntries(formData)
```

Passing additional arguments

Outside of form fields, you can pass additional arguments to a Server Function using the JavaScript `bind` method. For example, to pass the `userId` argument to the `updateUser` Server Function:

```
TS app/client-component.tsx TypeScript ▾
```

```
'use client'

import { updateUser } from './actions'

export function UserProfile({ userId }: { use
  const updateUserWithId = updateUser.bind(nu

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

The Server Function will receive the `userId` as an additional argument:

```
TS app/actions.ts TypeScript ▾
```

```
'use server'

export async function updateUser(userId: str
```

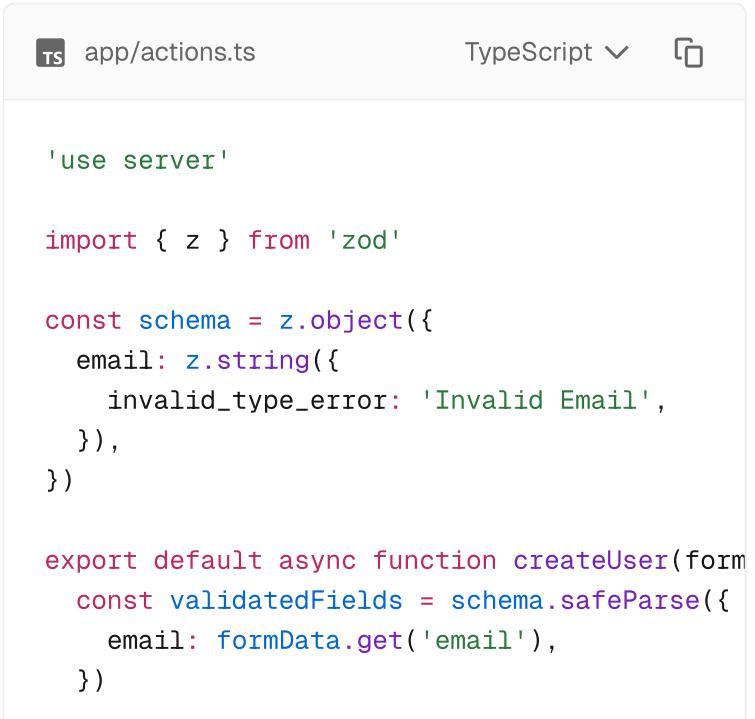
Good to know:

- 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 and supports progressive enhancement.

Form validation

Forms can be validated on the client or server.

- For **client-side validation**, you can use the HTML attributes like `required` and `type="email"` for basic validation.
- For **server-side validation**, you can use a library like [zod](#) ↗ to validate the form fields. For example:



The screenshot shows a code editor window with a TypeScript file named `app/actions.ts`. The code uses Zod to define a schema for an email field and then performs validation on form data.

```
TS app/actions.ts TypeScript ▾ ⌂

'use server'

import { z } from 'zod'

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

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

```

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

// Mutate data
}
```

Validation errors

To display validation errors or messages, turn the component that defines the `<form>` into a Client Component and use React [useActionState](#) ↗.

When using `useActionState`, the Server function signature will change to receive a new `prevState` or `initialState` parameter as its first argument.



```
'use server'

import { z } from 'zod'

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

You can then conditionally render the error message based on the `state` object.



```
'use client'

import { useActionState } from 'react'
```

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

const initialState = {
  message: '',
}

export function Signup() {
  const [state, formAction, pending] = useActionState(initialState)

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

Pending states

The `useActionState` hook exposes a `pending` boolean that can be used to show a loading indicator or disable the submit button while the action is being executed.

```
TS app/ui/signup.tsx TypeScript ▾ ⌂

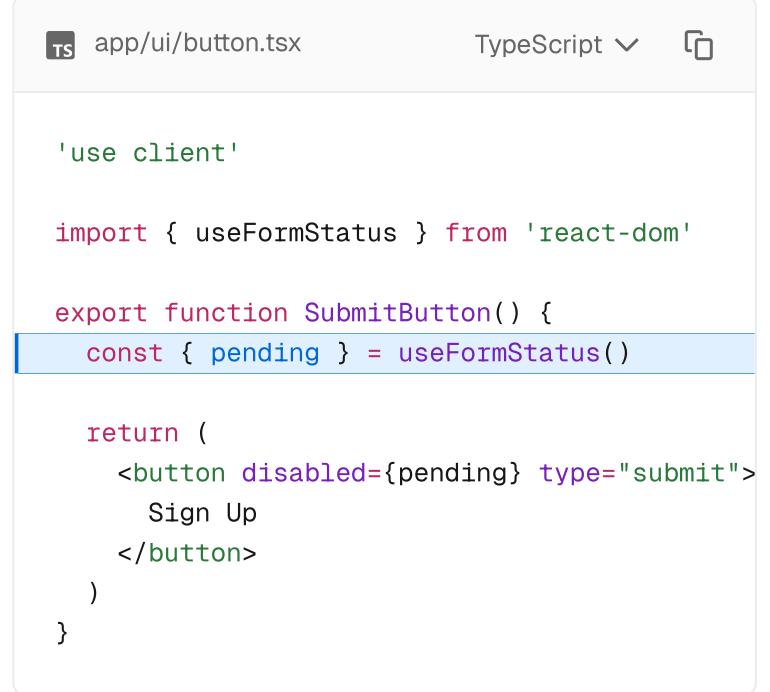
'use client'

import { useActionState } from 'react'
import { createUser } from '@/app/actions'

export function Signup() {
  const [state, formAction, pending] = useActionState(initialState)

  return (
    <form action={formAction}>
      /* Other form elements */
      <button disabled={pending}>Sign up</button>
    </form>
  )
}
```

Alternatively, you can use the `useFormStatus` hook to show a loading indicator while the action is being executed. When using this hook, you'll need to create a separate component to render the loading indicator. For example, to disable the button when the action is pending:



The screenshot shows a code editor window with the file name `app/ui/button.tsx` and the language set to `TypeScript`. The code defines a `SubmitButton` component that uses the `useFormStatus` hook to manage a `pending` state. The component returns a disabled `button` element with the value "Sign Up".

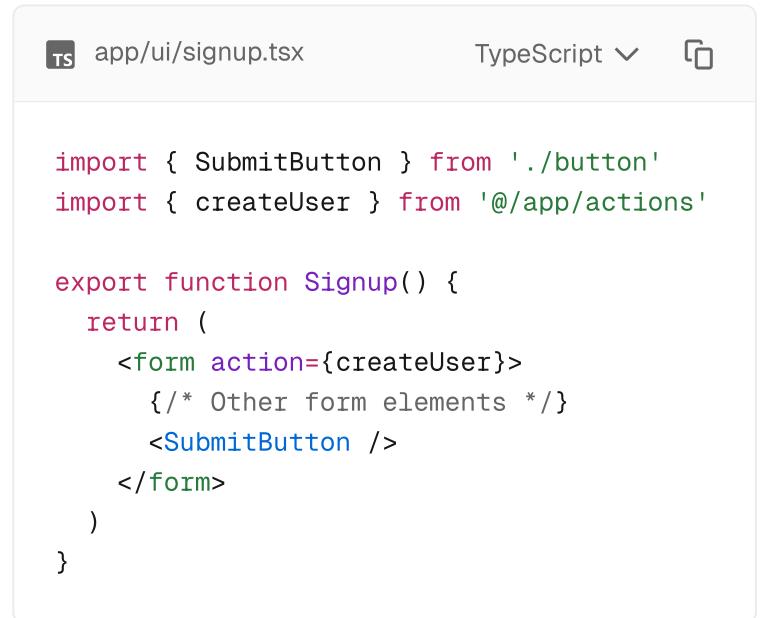
```
'use client'

import { useFormStatus } from 'react-dom'

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

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

You can then nest the `SubmitButton` component inside the form:



The screenshot shows a code editor window with the file name `app/ui/signup.tsx` and the language set to `TypeScript`. The code defines a `Signup` component that imports `SubmitButton` and `createUser`. It returns a `form` element with an `action` attribute set to `createUser`, containing other form elements and a `SubmitButton`.

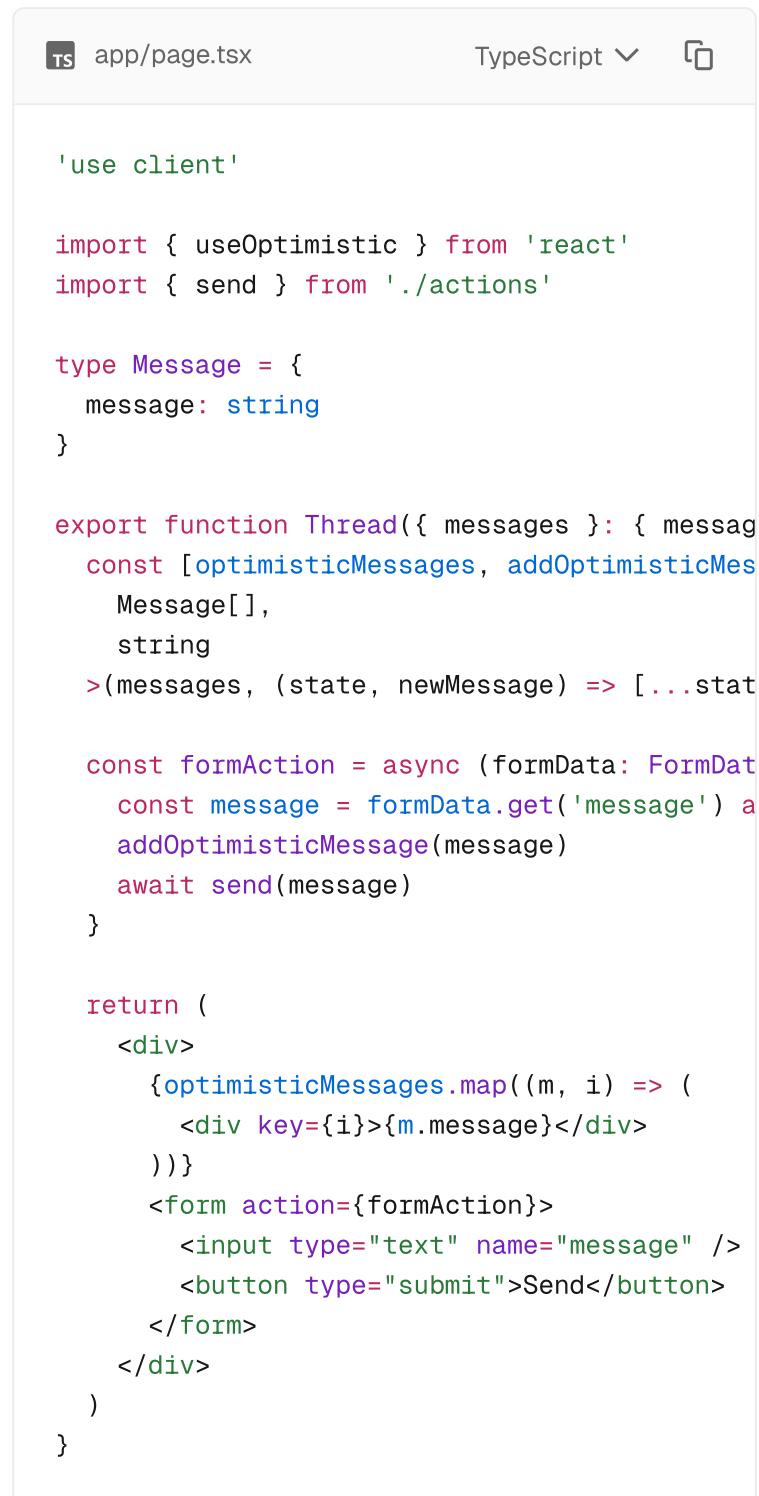
```
import { SubmitButton } from './button'
import { createUser } from '@/app/actions'

export function Signup() {
  return (
    <form action={createUser}>
      /* Other form elements */
      <SubmitButton />
    </form>
  )
}
```

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.

Optimistic updates

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



The screenshot shows a code editor window with the following details:

- File: app/page.tsx
- Language: TypeScript
- Icon: A small TS icon is visible on the left.
- Code Content:

```
'use client'

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

type Message = {
  message: string
}

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

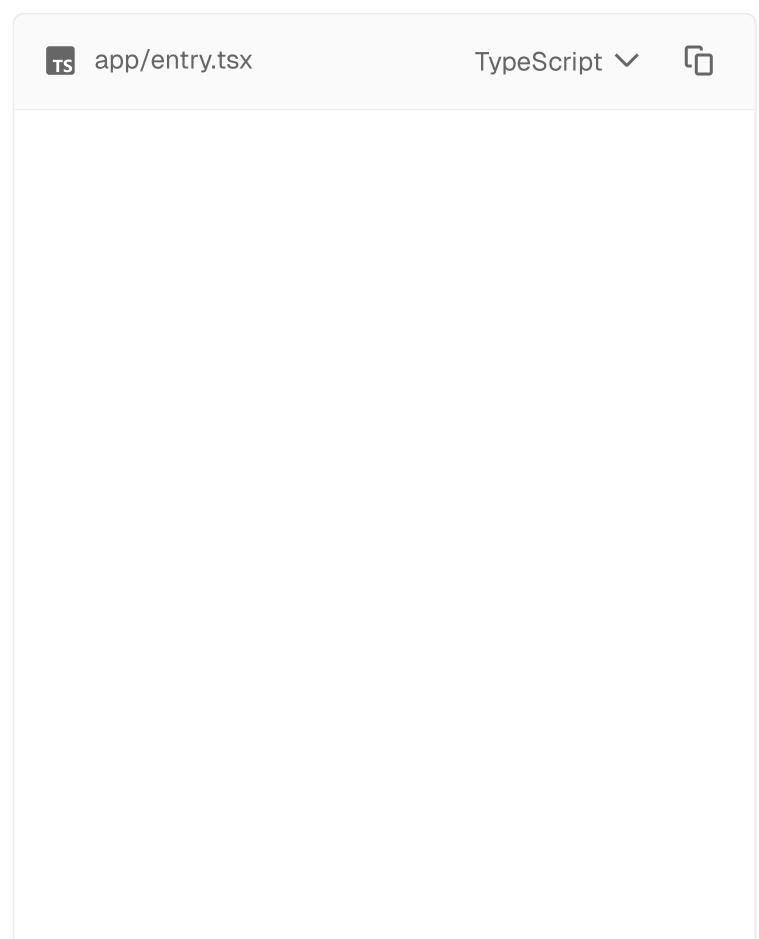
Nested form elements

You can call Server Actions 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 [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:



The screenshot shows a code editor window with a light gray background. At the top, there are two tabs: one labeled "app/entry.tsx" with a small "TS" icon, and another labeled "TypeScript" with a dropdown arrow. In the bottom right corner of the editor area, there is a small square icon with a circular arrow, likely a refresh or save button.

```
'use client'

export function Entry() {
  const handleKeyDown = (e: React.KeyboardEvent) =>
    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=>
  </div>
)
}
```

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

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



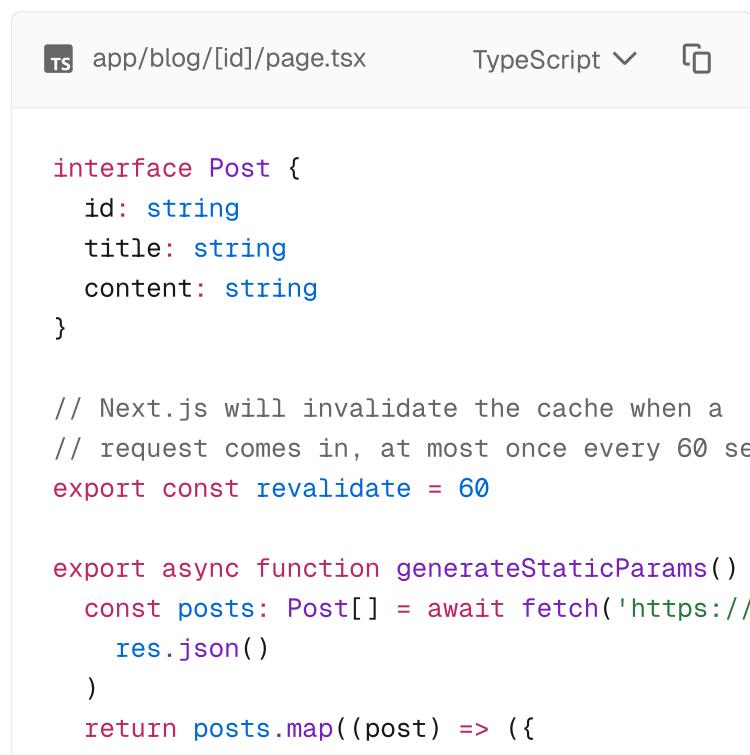
How to implement Incremental Static Regeneration (ISR)

► 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:



The screenshot shows a code editor window with a TypeScript file named `app/blog/[id]/page.tsx`. The code defines an interface `Post` and an `export const revalidate = 60` declaration. It also includes a function `generateStaticParams` that fetches posts from a URL and maps them into static parameters.

```
TS app/blog/[id]/page.tsx TypeScript ▾ ⌂

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

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

export async function generateStaticParams()
  const posts: Post[] = await fetch('https://
    res.json()
  )
  return posts.map((post) => ({
```

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

    export default async function Page({
      params,
    }: {
      params: Promise<{ id: string }>
    }) {
      const { id } = await params
      const post: Post = await fetch(`https://api
        (res) => res.json()
      )
      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
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 return the cached (now stale) page
4. The cache is invalidated and a new version of the page begins generating in the background
5. Once generated successfully, the next request will return the updated page and cache it for subsequent requests
6. If `/blog/26` is requested, and it exists, the page will be generated on-demand. This behavior can be changed by using a different `dynamicParams` value. However, if the post does not exist, then 404 is returned.

Reference

Route segment config

- `revalidate`
- `dynamicParams`

Functions

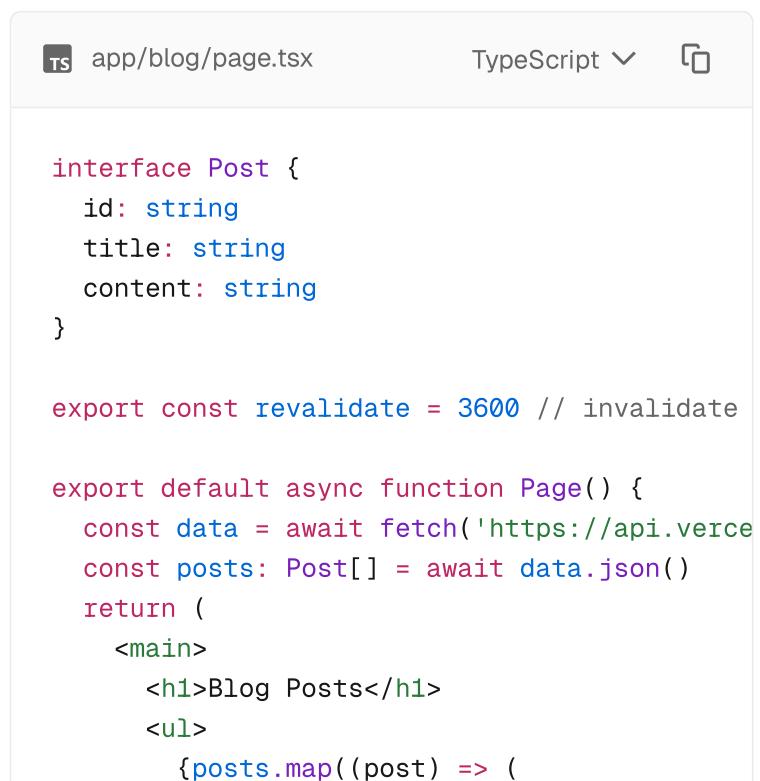
- `revalidatePath`
- `revalidateTag`

Examples

Time-based revalidation

This fetches and displays a list of blog posts on /blog. After an hour has passed, the next visitor will still receive the cached (stale) version of the page immediately for a fast response.

Simultaneously, Next.js triggers regeneration of a fresh version in the background. Once the new version is successfully generated, it replaces the cached version, and subsequent visitors will receive the updated content.



The screenshot shows a code editor window with a TypeScript file named `app/blog/page.tsx`. The file contains the following code:

```
TS app/blog/page.tsx TypeScript ▾ ⌂

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

export const revalidate = 3600 // invalidate

export default async function Page() {
  const data = await fetch('https://api.veralabs.com/blogs')
  const posts: Post[] = await data.json()
  return (
    <main>
      <h1>Blog Posts</h1>
      <ul>
        {posts.map((post) => (
          <li>
            <h2>{post.title}</h2>
            <p>{post.content}</p>
          </li>
        ))}
      </ul>
    </main>
  )
}
```

```
<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 cached 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 invalidate the cache for the entire route. The next request to that route will trigger regeneration and serve fresh data, which will then be cached for subsequent requests.

Note: `revalidatePath` invalidates the cache entries but regeneration happens on the next request. If you want to eagerly regenerate the cache entry immediately instead of waiting for the next request, you can use the Pages router `res.revalidate` method. We're working on adding new methods to provide eager regeneration capabilities for the App Router.

TS app/actions.ts

TypeScript ▾



```
'use server'
```

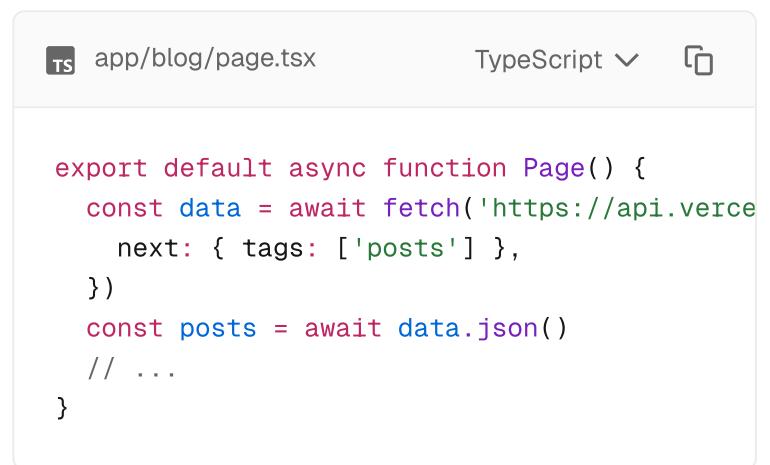
```
import { revalidatePath } from 'next/cache'

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

[View a demo ↗](#) and [explore the source code ↗](#).

On-demand revalidation with `revalidateTag`

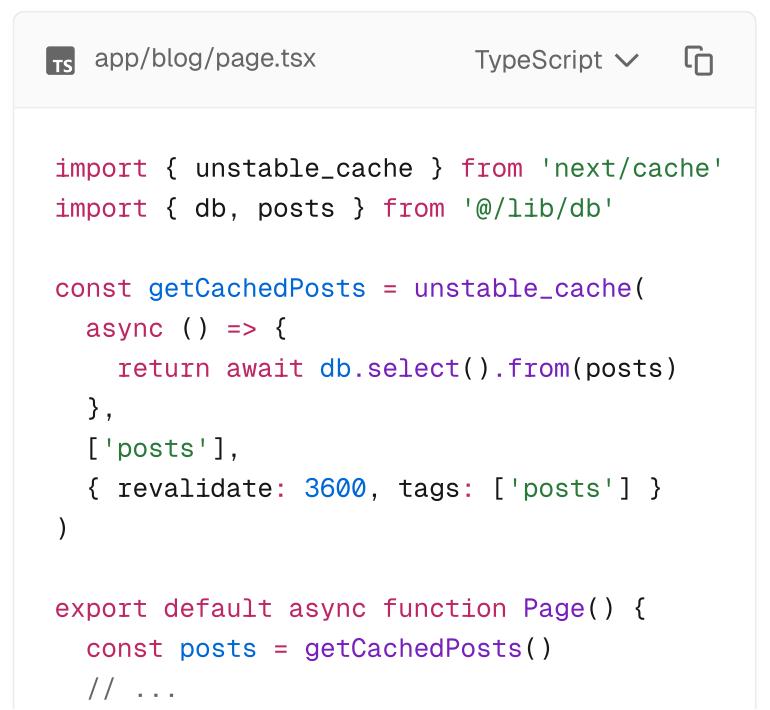
For most use cases, prefer revalidating entire paths. If you need more granular control, you can use the `revalidateTag` function. For example, you can tag individual `fetch` calls:



```
TS app/blog/page.tsx TypeScript ▾
```

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

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



```
TS app/blog/page.tsx TypeScript ▾
```

```
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](#):



```
TS app/actions.ts TypeScript ▾ ⌂

'use server'

import { revalidateTag } from 'next/cache'

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

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](#).

Customizing the cache location

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.](#)



```
JS next.config.js

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:



```
File .env

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`.
-

Platform Support

Deployment Option	Supported
Node.js server	Yes
Docker container	Yes
Static export	No
Adapters	Platform-specific

Learn how to [configure ISR](#) when self-hosting Next.js.

Version history

Version	Changes
v14.1.0	Custom <code>cacheHandler</code> 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 .

Was this helpful?





Using App Router
Features available in /app



Latest Version
15.5.4



How to set up instrumentation

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.

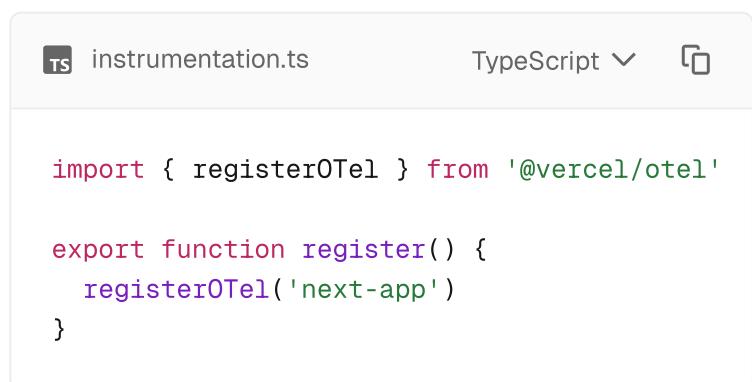
Convention

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](#) ↗ :



TS instrumentation.ts TypeScript ▾ ⌂

```
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')
}
```

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:



The screenshot shows a code editor window with the file name `instrumentation.ts`. The code is written in TypeScript and uses conditional imports based on the `NEXT_RUNTIME` environment variable:

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

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

Learn more about Instrumentation

[instrumentat...](#)

API reference for
the
`instrumentation.j...`

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

How to implement JSON-LD in your Next.js application

[JSON-LD ↗](#) is a format for structured data that can be used by search engines and AI to help them understand the structure of the page beyond pure 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.

The following snippet uses `JSON.stringify`, which does not sanitize malicious strings used in XSS injection. To prevent this type of vulnerability, you can scrub `HTML` tags from the `JSON-LD` payload, for example, by replacing the character, `<`, with its unicode equivalent, `\u003c`.

Review your organization's recommended approach to sanitize potentially dangerous strings, or use community maintained alternatives for `JSON.stringify` such as, [serialize-javascript ↗](#).

```
TS app/products/[id]/page.tsx TypeScript ▾   
  
export default async function Page({ params }  
  const { id } = await params  
  const product = await getProduct(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).replace(/\n/g, '')
}}}
/>
{/* ... */}
</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'

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'
}
```

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



How to lazy load Client Components and libraries

[Lazy loading ↗](#) in Next.js helps improve the initial loading performance of an application by decreasing the amount of JavaScript needed to render a route.

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

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

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

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

`next/dynamic`

`next/dynamic` is a composite of [React.lazy\(\)](#) ↗ and [Suspense](#) ↗. It behaves the same way in the

app and pages directories to allow for incremental migration.

Examples

Importing Client Components

```
JS app/page.js

'use client'

import { useState } from 'react'
import dynamic from 'next/dynamic'

// Client Components:
const ComponentA = dynamic(() => import('../components/ComponentA'))
const ComponentB = dynamic(() => import('../components/ComponentB'))
const ComponentC = dynamic(() => import('../components/ComponentC'))

export default function ClientComponentExample() {
  const [showMore, setShowMore] = useState(false)

  return (
    <div>
      {/* Load immediately, but in a separate component */}
      <ComponentA />

      {/* Load on demand, only when/if the condition is met */}
      {showMore && <ComponentB />}
      <button onClick={() => setShowMore(!showMore)}>Show More</button>

      {/* Load only on the client side */}
      <ComponentC />
    </div>
  )
}
```

Note: When a Server Component dynamically imports a Client Component, automatic [code splitting ↗](#) is currently **not** supported.

Skiping SSR

When using `React.lazy()` and Suspense, Client Components will be [prerendered ↗](#) (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('../c
```

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.

```
JS app/page.js
```

```
import dynamic from 'next/dynamic'

// Server Component:
const ServerComponent = dynamic(() => import('

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.

```
JS app/page.js

'use client'

import { useState } from 'react'

const names = ['Tim', 'Joe', 'Bel', 'Lee']

export default function Page() {
  const [results, setResults] = useState()

  return (
    <div>
      <input
        type="text"
        placeholder="Search"
        onChange={async (e) => {
          const { value } = e.currentTarget
          // Dynamically load fuse.js
          const Fuse = (await import('fuse.js'))
          const fuse = new Fuse(names)

          setResults(fuse.search(value))
        }}
      />
      <pre>Results: {JSON.stringify(results,
      </div>
    )
  }
}
```

Adding a custom loading component

```
JS app/page.js

'use client'
```

```
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
       <WithCustomLoading />
      </div>
    )
  }
}
```

Importing Named Exports

To dynamically import a named export, you can return it from the Promise returned by `import()` function:

JS components/hello.js

```
'use client'

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

JS app/page.js

```
import dynamic from 'next/dynamic'

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

Was this helpful? 😢 😕 😊 😍

 Using App Router
Features available in /app

 Latest Version
15.5.4



How to optimize your local development environment

Next.js is designed to provide a great developer experience. As your application grows, you might notice slower compilation times during local development. This guide will help you identify and fix common compile-time performance issues.

Local dev vs. production

The development process with `next dev` is different than `next build` and `next start`.

`next dev` compiles routes in your application as you open or navigate to them. This enables you to start the dev server without waiting for every route in your application to compile, which is both faster and uses less memory. Running a production build applies other optimizations, like minifying files and creating content hashes, which are not needed for local development.

Improving local dev performance

1. Check your computer's antivirus

Antivirus software can slow down file access.

Try adding your project folder to the antivirus exclusion list. While this is more common on Windows machines, we recommend this for any system with an antivirus tool installed.

2. Update Next.js and enable Turbopack

Make sure you're using the latest version of Next.js. Each new version often includes performance improvements.

Turbopack is a new bundler integrated into Next.js that can improve local performance.

```
npm install next@latest  
npm run dev --turbopack
```

Learn more about [Turbopack](#). See our [upgrade guides](#) and codemods for more information.

3. Check your imports

The way you import code can greatly affect compilation and bundling time. Learn more about [optimizing package bundling](#) and explore tools like [Dependency Cruiser ↗](#) or [Madge ↗](#).

Icon libraries

Libraries like `@material-ui/icons`, `@phosphor-icons/react`, or `react-icons` can import thousands of icons, even if you only use a few. Try to import only the icons you need:

```
// Instead of this:  
import { TriangleIcon } from '@phosphor-icons'  
  
// Do this:  
import { TriangleIcon } from '@phosphor-icons'
```

You can often find what import pattern to use in the documentation for the icon library you're using. This example follows [@phosphor-icons/react](#) ↗ recommendation.

Libraries like `react-icons` includes many different icon sets. Choose one set and stick with that set.

For example, if your application uses `react-icons` and imports all of these:

- `pi` (Phosphor Icons)
- `md` (Material Design Icons)
- `tb` (tabler-icons)
- `cg` (cssgg)

Combined they will be tens of thousands of modules that the compiler has to handle, even if you only use a single import from each.

Barrel files

"Barrel files" are files that export many items from other files. They can slow down builds because the compiler has to parse them to find if there are side-effects in the module scope by using the import.

Try to import directly from specific files when possible. [Learn more about barrel files](#) ↗ and the built-in optimizations in Next.js.

Optimize package imports

Next.js can automatically optimize imports for certain packages. If you are using packages that

utilize barrel files, add them to your `next.config.js`:

```
module.exports = {
  experimental: {
    optimizePackageImports: ['package-name'],
  },
}
```

Turbopack automatically analyzes imports and optimizes them. It does not require this configuration.

4. Check your Tailwind CSS setup

If you're using Tailwind CSS, make sure it's set up correctly.

A common mistake is configuring your `content` array in a way which includes `node_modules` or other large directories of files that should not be scanned.

Tailwind CSS version 3.4.8 or newer will warn you about settings that might slow down your build.

1. In your `tailwind.config.js`, be specific about which files to scan:

```
module.exports = {
  content: [
    './src/**/*.{js,ts,jsx,tsx}', // Good
    // This might be too broad
    // It will match `packages/**/node_module
    // '.../.../packages/**/*.{js,ts,jsx,tsx}`,
  ],
}
```

2. Avoid scanning unnecessary files:

```
module.exports = {
  content: [
    // Better - only scans the 'src' folder
```

```
'.../.../packages/ui/src/**/*.{js,ts,jsx,ts
```

```
],
```

```
}
```

5. Check custom webpack settings

If you've added custom webpack settings, they might be slowing down compilation.

Consider if you really need them for local development. You can optionally only include certain tools for production builds, or explore moving to Turbopack and using [loaders](#).

6. Optimize memory usage

If your app is very large, it might need more memory.

[Learn more about optimizing memory usage.](#)

7. Server Components and data fetching

Changes to Server Components cause the entire page to re-render locally in order to show the new changes, which includes fetching new data for the component.

The experimental `serverComponentsHmrCache` option allows you to cache `fetch` responses in Server Components across Hot Module Replacement (HMR) refreshes in local development. This results in faster responses and reduced costs for billed API calls.

[Learn more about the experimental option.](#)

8. Consider local development over Docker

If you're using Docker for development on Mac or Windows, you may experience significantly slower performance compared to running Next.js locally.

Docker's filesystem access on Mac and Windows can cause Hot Module Replacement (HMR) to take seconds or even minutes, while the same application runs with fast HMR when developed locally.

This performance difference is due to how Docker handles filesystem operations outside of Linux environments. For the best development experience:

- Use local development (`npm run dev` or `pnpm dev`) instead of Docker during development
- Reserve Docker for production deployments and testing production builds
- If you must use Docker for development, consider using Docker on a Linux machine or VM

[Learn more about Docker deployment](#) for production use.

Tools for finding problems

Detailed fetch logging

Use the `logging.fetches` option in your `next.config.js` file, to see more detailed information about what's happening during development:

```
module.exports = {
  logging: {
    fetches: {
```

```
        fullUrl: true,  
    },  
},  
}  
}
```

[Learn more about fetch logging.](#)

Turbopack tracing

Turbopack tracing is a tool that helps you understand the performance of your application during local development. It provides detailed information about the time taken for each module to compile and how they are related.

1. Make sure you have the latest version of Next.js installed.
2. Generate a Turbopack trace file:

```
NEXT_TURBOPACK_TRACING=1 npm run dev
```

3. Navigate around your application or make edits to files to reproduce the problem.
4. Stop the Next.js development server.
5. A file called `trace-turbopack` will be available in the `.next` folder.
6. You can interpret the file using

```
npx next internal trace [path-to-file]:
```

```
npx next internal trace .next/trace-turbopack
```

On versions where `trace` is not available, the command was named `turbo-trace-server`:

```
npx next internal turbo-trace-server .next/tr
```

7. Once the trace server is running you can view the trace at <https://trace.nextjs.org/>.
 8. By default the trace viewer will aggregate timings, in order to see each individual time you can switch from "Aggregated in order" to "Spans in order" at the top right of the viewer.
-

Still having problems?

Share the trace file generated in the [Turbopack Tracing](#) section and share it on [GitHub Discussions](#) or [Discord](#).

Was this helpful?    

 Using App Router
Features available in /app Latest Version
15.5.4

How to use mark-down and MDX in Next.js

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="htt
```

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:

```
>_ Terminal   
npm install @next/mdx @mdx-js/loader @mdx-js/
```

Configure `next.config.mjs`

Update the `next.config.mjs` file at your project's root to configure it to use MDX:

```
JS next.config.mjs   
  
import createMDX from '@next/mdx'  
  
/** @type {import('next').NextConfig} */  
const nextConfig = {  
  // Configure `pageExtensions` to include md  
  pageExtensions: ['js', 'jsx', 'md', 'mdx'],  
  // Optionally, add any other Next.js config  
}  
  
const withMDX = createMDX({  
  // Add markdown plugins here, as desired  
})
```

```
// Merge MDX config with Next.js config  
export default withMDX(nextConfig)
```

This allows `.mdx` files to act as pages, routes, or imports in your application.

Handling `.md` files

By default, `next/mdl` only compiles files with the `.mdx` extension. To handle `.md` files with webpack, update the `extension` option:

`JS` next.config.mjs



```
const withMDX = createMDX({  
  extension: /\.(\.md|\.mdx)$/,  
})
```

Add an `mdl-components.tsx` file

Create an `mdl-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.

`TS` mdl-components.tsx

TypeScript ▾



```
import type { MDXComponents } from 'mdl/types'  
  
const components: MDXComponents = {}  
  
export function useMDXComponents(): MDXComponents  
  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
```

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:

```
.
├── app/
│   └── mdx-page/
│       └── 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 the MDX file inside the page to display the content:

```
TS  app/mdx-page/page.tsx  TypeScript ▾  ⌂
import Welcome from '@/markdown/welcome.mdx'

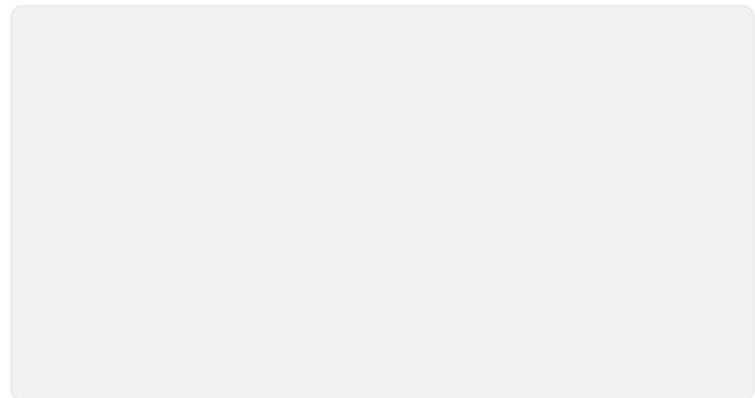
export default function Page() {
  return <Welcome />
}
```

Navigating to the `/mdx-page` route should display your rendered MDX page.

Using dynamic imports

You can import dynamic MDX components instead of using filesystem routing for MDX files.

For example, you can have a dynamic route segment which loads MDX components from a separate directory:



`generateStaticParams` can be used to prerender the provided routes. By marking `dynamicParams` as `false`, accessing a route not defined in `generateStaticParams` will 404.

```
TS app/blog/[slug]/page.tsx TypeScript ▾ ⌂
export default async function Page({
  params,
}: {
  params: Promise<{ slug: string }>
}) {
  const { slug } = await params
  const { default: Post } = await import(`@/c

  return <Post />
}

export function generateStaticParams() {
  return [{ slug: 'welcome' }, { slug: 'about' }]
}

export const dynamicParams = false
```

Good to know: Ensure you specify the `.mdx` file extension in your import. While it is not required to use `module path aliases` (e.g., `@/content`), it does simplify your import path.

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 an
// React component you want, including inline
// components from other libraries, and more.

const components = {
  // Allows customizing built-in components,
  h1: ({ children }) => (
    <h1 style={{ color: 'red', fontSize: '48px' }}>
      {children}
    </h1>
  ),
  img: (props) => (
    <Image
      sizes="100vw"
      style={{ width: '100%', height: 'auto' }}
      {...(props as ImageProps)}
    />
  ),
} satisfies MDXComponents

export function useMDXComponents(): MDXComponents {
  return 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](#).

```

TS app/mdx-page/page.tsx TypeScript ▾ ⌂
import Welcome from '@/markdown/welcome.mdx'

function CustomH1({ children }) {
  return <h1 style={{ color: 'blue', fontSize: '24px' }}>
    {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.



```
TS app/mdx-page/layout.tsx TypeScript ▾
```

```
export default function MdxLayout({ children
  // Create any shared layout or styles here
  return <div style={{ color: 'blue' }}>{children}
}
```

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.



```
TS app/mdx-page/layout.tsx TypeScript ▾
```

```
export default function MdxLayout({ children
  // Create any shared layout or styles here
  return (
    <div className="prose prose-headings:mt-8">
      {children}
    </div>
  )
}
```

Frontmatter

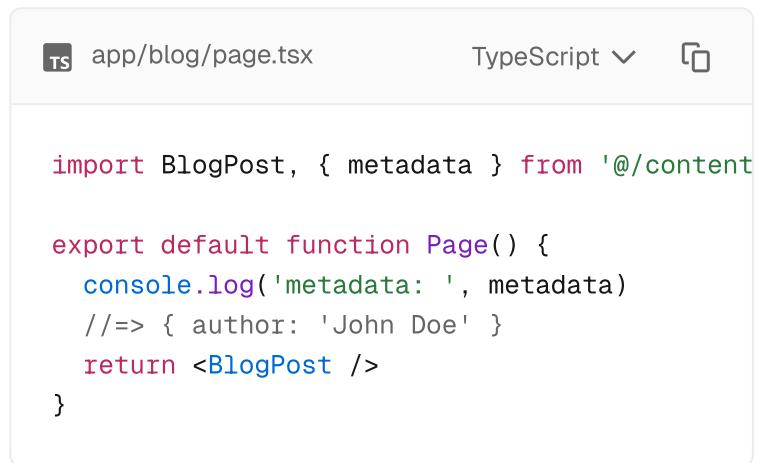
Frontmatter is a YAML like key/value pairing that can be used to store data about a page.

`@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:

Metadata can now be referenced outside of the MDX file:



```
TS app/blog/page.tsx TypeScript ▾ ⌂

import BlogPost, { metadata } from '@/content

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` or `next.config.ts` as the configuration file.

```
JS next.config.mjs

import remarkGfm from 'remark-gfm'
import createMDX from '@next/mdx'

/** @type {import('next').NextConfig} */
const nextConfig = {
    // Allow .mdx extensions for files
    pageExtensions: ['js', 'jsx', 'md', 'mdx'],
    // Optionally, add any other Next.js config
}

const withMDX = createMDX({
    // Add markdown plugins here, as desired
    options: {
        remarkPlugins: [remarkGfm],
        rehypePlugins: [],
    },
})

// Combine MDX and Next.js config
export default withMDX(nextConfig)
```

Using Plugins with Turbopack

To use plugins with [Turbopack](#), upgrade to the latest `@next/mdx` and specify plugin names using a string:

```
JS next.config.mjs
```

```
import createMDX from '@next/mdx'

/** @type {import('next').NextConfig} */
const nextConfig = {
  pageExtensions: ['js', 'jsx', 'md', 'mdx'],
}

const withMDX = createMDX({
  options: {
    remarkPlugins: [
      // Without options
      'remark-gfm',
      // With options
      ['remark-toc', { heading: 'The Table' }],
    ],
    rehypePlugins: [
      // Without options
      'rehype-slug',
      // With options
      ['rehype-katex', { strict: true, throwOnWarning: false }],
    ],
  },
})

export default withMDX(nextConfig)
```

Good to know:

remark and rehype plugins without serializable options cannot be used yet with [TurboPack](#), because JavaScript functions can't be passed to Rust.

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 CMS, database, or anywhere else. A community package for this use is [next-mdx-remote-client](#).

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-client`:

```
TS app/mdx-page-remote/page.... TypeScript ▾ ⌂  
  
import { MDXRemote } from 'next-mdx-remote-cl  
  
export default async function RemoteMdxPage()  
  // MDX text - can be from a database, CMS,  
  const res = await fetch('https://...')  
  const markdown = await res.text()  
  return <MDXRemote source={markdown} />  
}
```

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
    .use(remarkRehype) // Transform to HTML AST
    .use(rehypeSanitize) // Sanitize HTML input
    .use(rehypeStringify) // Convert AST into string
    .process('Hello, Next.js!')

  console.log(String(file)) // <p>Hello, Next.js!
}
```

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`:

```
JS next.config.js
```

```
module.exports = withMDX({
  experimental: {
    mdxRs: true,
  },
})
```

`mdxRs` also accepts an object to configure how to transform mdx files.

JS next.config.js



```
module.exports = withMDX({
  experimental: {
    mdxRs: {
      jsxRuntime?: string // Custo
      jsxImportSource?: string // Custo
      mdxType?: 'gfm' | 'commonmark' // Config
    },
  },
})
```

Helpful Links

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

Was this helpful?

 Using App Router
Features available in /app Latest Version
15.5.4

How to optimize memory usage

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 `v14.2.0`, 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:

```
JS next.config.mjs

/** @type {import('next').NextConfig} */
const nextConfig = {
  webpack: (
    config,
    { buildId, dev, isServer, defaultLoaders, nextRuntime }
  ) => {
    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:

```
JS next.config.mjs

/** @type {import('next').NextConfig} */
const nextConfig = {
  eslint: {
    // Warning: This allows production builds to success
```

```
// your project has ESLint errors.  
ignoreDuringBuilds: true,  
,  
typescript: {  
    // !! WARN !!  
    // Dangerously allow production builds to successfully  
    // 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.

Preloading Entries

When the Next.js server starts, it preloads each page's JavaScript modules into memory, rather than at request time.

This optimization allows for faster response times, in exchange for a larger initial memory footprint.

To disable this optimization, set the

`experimental.preloadEntriesOnStart` flag to `false`.



```
TS next.config.ts TypeScript ▾ ⌂

import type { NextConfig } from 'next'

const config: NextConfig = {
  experimental: {
    preloadEntriesOnStart: false,
  },
}

export default config
```

Next.js doesn't unload these JavaScript modules, meaning that even with this optimization disabled, the memory footprint of your Next.js server will eventually be the same if all pages are eventually requested.

Was this helpful?    



Using App Router
Features available in /app



Migrating



Latest Version
15.5.4



App Router

Learn how to
upgrade your
existing Next.js...

Create React...

Learn how to
migrate your
existing React...

Vite

Learn how to
migrate your
existing React...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4



How to migrate from Pages 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:

```
>_ Terminal   
  
npm install next@latest react@latest react-do
```

ESLint Version

If you're using ESLint, you need to upgrade your ESLint version:

> Terminal



```
npm install -D eslint-config-next@latest
```

Good to know: You may need to restart the ESLint server in VS Code for the ESLint changes to take effect. Open the Command Palette (cmd+shift+p on Mac; ctrl+shift+p on Windows) and search for ESLint: Restart ESLint Server.

Next Steps

After you've updated, see the following sections for next steps:

- [Upgrade new features](#): 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 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
<Link href="/about">
  <a>About</a>
</Link>

// Next.js 13: `<Link>` always renders `<a>`
<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

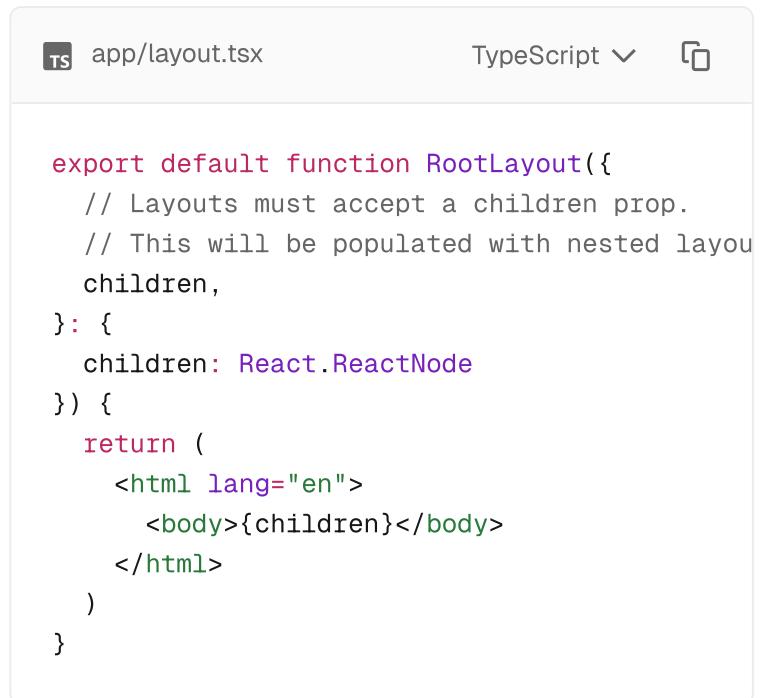
Update to the latest Next.js version (requires 13.4 or greater):

```
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`.

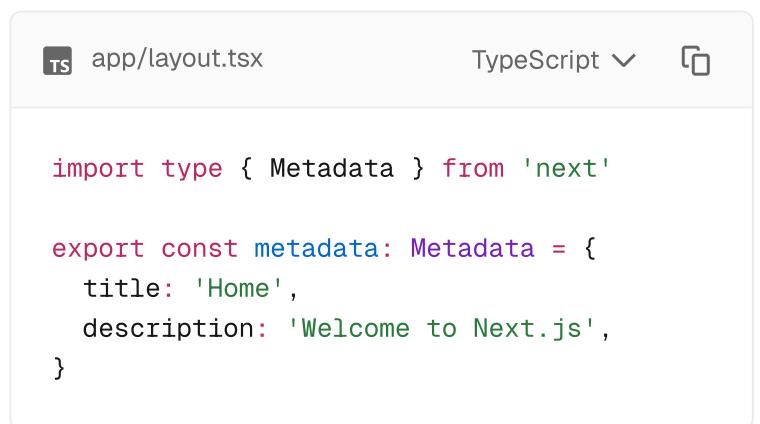


```
TS app/layout.tsx TypeScript ▾ ⌂

export default function RootLayout({
  // Layouts must accept a children prop.
  // This will be populated with nested layout
  // children,
}: {
  children: React.ReactNode
}) {
  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](#):



```
TS app/layout.tsx TypeScript ▾ ⌂

import type { Metadata } from 'next'

export const metadata: 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)

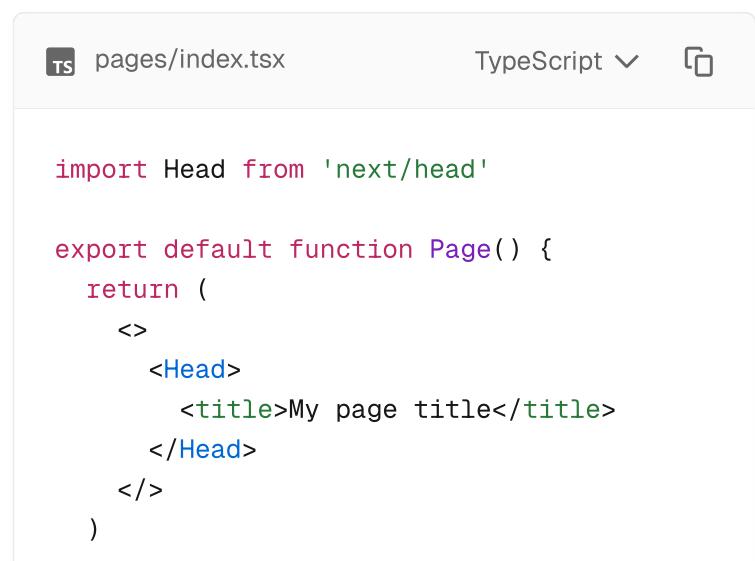
Next.js recommended adding a [property to Page components](#) to achieve per-page layouts in the `pages` directory. This pattern can be replaced with native support for [nested layouts](#) in the `app` directory.

► 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:



The screenshot shows a code editor window with the following details:

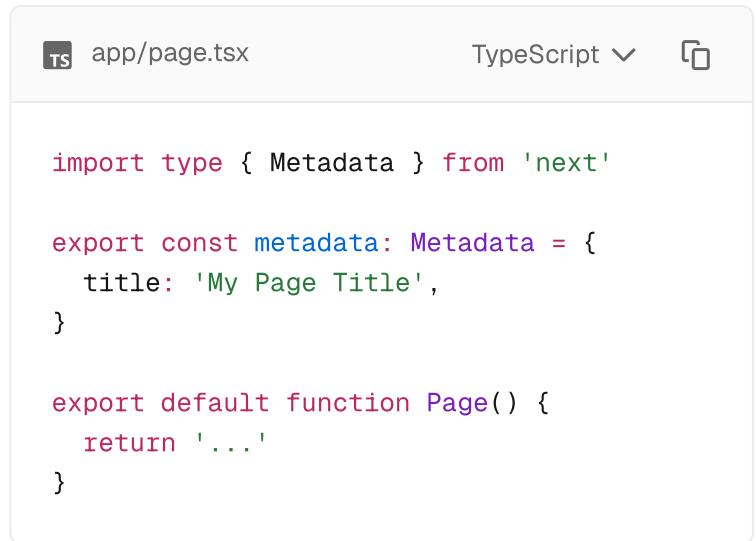
- File name: `pages/index.tsx`
- TypeScript version: `TypeScript v`
- Copy icon: `C`

```
import Head from 'next/head'

export default function Page() {
  return (
    <>
    <Head>
      <title>My page title</title>
    </Head>
    </>
  )
}
```

}

After:



```
TS app/page.tsx TypeScript ▾
```

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

export const metadata: 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
<code>index.js</code>	<code>page.js</code>	<code>/</code>
<code>about.js</code>	<code>about/page.js</code>	<code>/about</code>
<code>blog/[slug].js</code>	<code>blog/[slug]/page.js</code>	<code>/blog/p[1]</code>

We recommend breaking down the migration of a page into two main steps:

- Step 1: Move the default exported Page Component into a new Client Component.
- Step 2: Import the new Client Component into a new `page.js` file inside the `app` directory.

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`.

The screenshot shows a code editor interface with the following details:

- File name: `app/home-page.tsx`
- Language: TypeScript (indicated by the `TS` icon)
- File type: `JavaScript` (indicated by the `.js` extension)
- Editor interface: Includes tabs for `TypeScript` and `Copy` (represented by a clipboard icon).

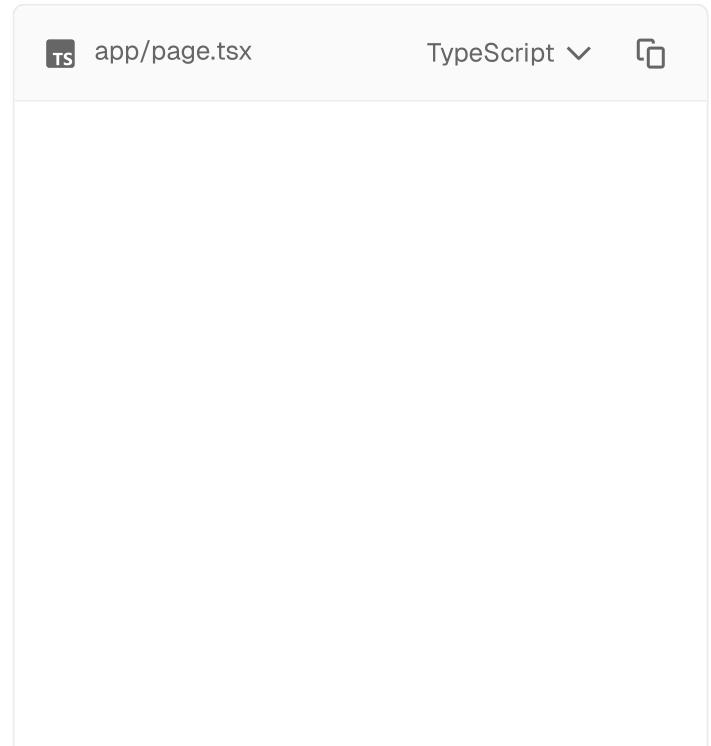
The code area is currently empty, showing a blank white space.

```
'use client'

// This is a Client Component (same as component)
// It receives data as props, has access to state
// prerendered on the server during the initial render
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} />
}

```

- 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.

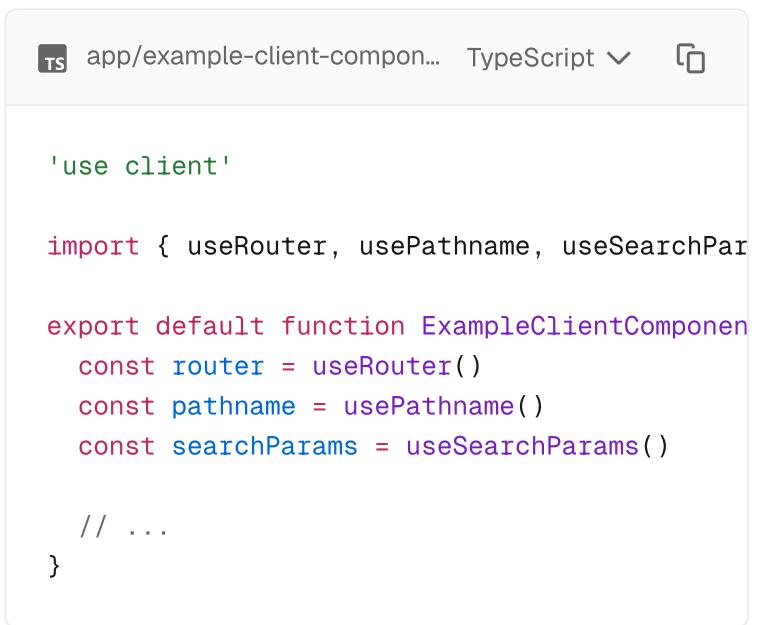
Step 5: Migrating Routing Hooks

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.



The screenshot shows a code editor window with a TypeScript file open. The file contains the following code:

```

'ts' app/example-client-compon... TypeScript ▾ ⌂

'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 manual
  // Similar to `getStaticProps`.
  // `force-cache` is the default and can be
  const staticData = await fetch(`https://...`)

  // This request should be refetched on every
  // Similar to `getServerSideProps`.
  const dynamicData = await fetch(`https://...`)

  // This request should be cached with a lifetime
  // Similar to `getStaticProps` with the `revalidate` prop.
  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).

```
js pages/dashboard.js
```

```
// `pages` directory

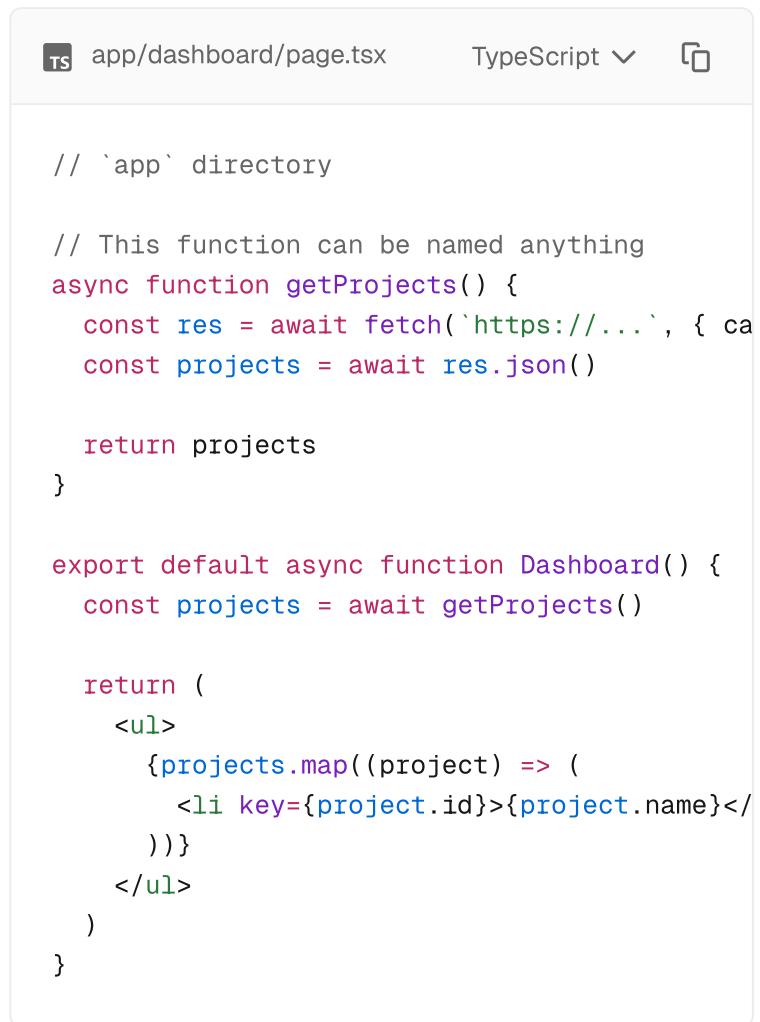
export async function getServerSideProps() {
  const res = await fetch(`https://...`)
  const projects = await res.json()

  return { props: { projects } }
}

export default function Dashboard({ projects }) {
  return (
    <ul>
      {projects.map((project) => (
        <li key={project.id}>{project.name}</li>
      ))}
    </ul>
  )
}
```

In the App 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://...`, { ca
  const projects = await res.json()

  return projects
}

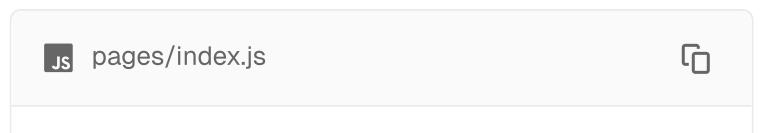
export default async function Dashboard() {
  const projects = await getProjects()

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



```
js pages/index.js
```

```
// `pages` directory

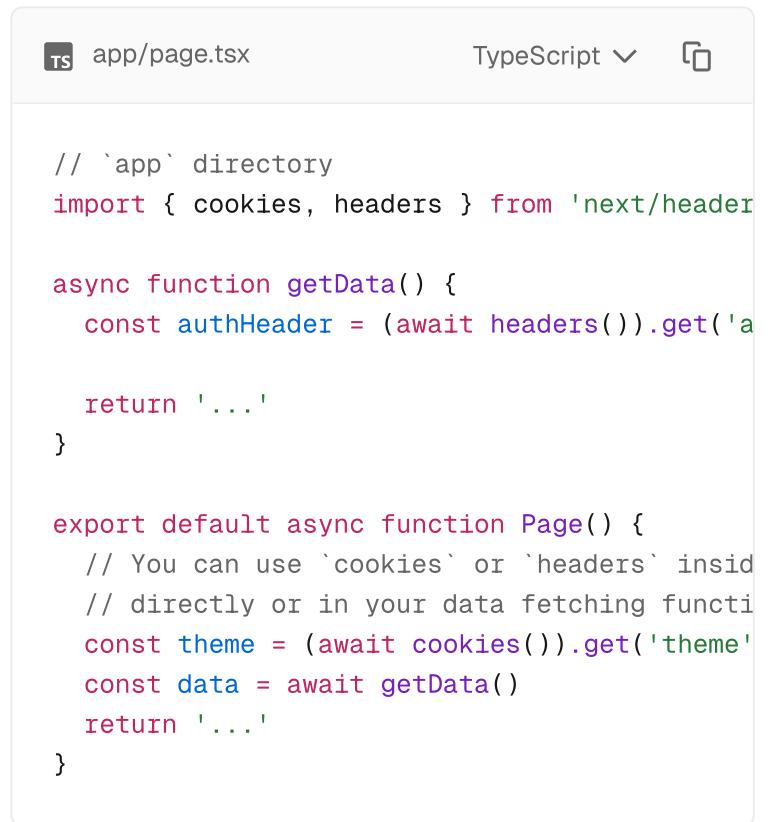
export async function getServerSideProps({ req }) {
  const authHeader = req.getHeaders()['authorization'];
  const theme = req.cookies['theme'];

  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/header'

async function getData() {
  const authHeader = (await headers()).get('authorization');
  return '...'
}

export default async function Page() {
  // You can use `cookies` or `headers` inside
  // 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.

JS pages/index.js

```
// `pages` directory

export async function getStaticProps() {
  const res = await fetch(`https://...`)
  const projects = await res.json()

  return { props: { projects } }
}

export default function Index({ projects }) {
  return projects.map((project) => <div>{proj
}
```

In the `app` directory, data fetching with `fetch()` will default to `cache: 'force-cache'`, which will cache the request data until manually invalidated. This is similar to `getStaticProps` in the `pages` directory.

JS app/page.js

```
// `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>{proj
}
```

Dynamic paths (`getStaticPaths`)

In the `pages` directory, the `getStaticPaths` function is used to define the dynamic paths that should be pre-rendered at build time.

```
JS pages/posts/[id].js Copy  
  
// `pages` directory  
import PostLayout from '@/components/post-lay  
  
export async function getStaticPaths() {  
  return {  
    paths: [{ params: { id: '1' } }, { params  
    }  
  }  
  
  export async function getStaticProps({ params  
    const res = await fetch(`https://.../posts/  
    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.

```
JS app/posts/[id]/page.js Copy  
  
// `app` directory  
import PostLayout from '@/components/post-lay  
  
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.

JS pages/posts/[id].js

```
// `pages` directory

export async function getStaticPaths() {
  return {
    paths: [],
    fallback: 'blocking'
  };
}
```

```
}
```

```
export async function getStaticProps({ params
  ...
}

export default function Post({ post }) {
  return ...
}
```

In the `app` directory the `config.dynamicParams` property controls how params outside of `generateStaticParams` are handled:

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

This replaces the

`fallback: true | false | 'blocking'` option of `getStaticPaths` in the `pages` directory. The `fallback: 'blocking'` option is not included in `dynamicParams` because the difference between `'blocking'` and `true` is negligible with streaming.

`JS app/posts/[id]/page.js`



```
// `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.

```
JS pages/index.js

// `pages` directory

export async function getStaticProps() {
  const res = await fetch(`https://.../posts`)
  const posts = await res.json()

  return {
    props: { posts },
    revalidate: 60,
  }
}

export default function Index({ posts }) {
  return (
    <Layout>
      <PostList posts={posts} />
    </Layout>
  )
}
```

In the `app` directory, data fetching with `fetch()` can use `revalidate`, which will cache the request for the specified amount of seconds.

```
JS app/page.js

// `app` directory
```

```
async function getPosts() {
  const res = await fetch(`https://.../posts`)
  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.



```
TS app/api/route.ts TypeScript ▾ ⌂
export async function GET(request: 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](#).

Single-Page Applications

If you are also migrating to Next.js from a Single-Page Application (SPA) at the same time, see our [documentation](#) to learn more.

Step 7: Styling

In the `pages` directory, global stylesheets are restricted to only `pages/_app.js`. With the `app` directory, this restriction has been lifted. Global

styles can be added to any layout, page, or component.

- [CSS Modules](#)
- [Tailwind CSS](#)
- [Global Styles](#)
- [CSS-in-JS](#)
- [External Stylesheets](#)
- [Sass](#)

Tailwind CSS

If you're using Tailwind CSS, you'll need to add the `app` directory to your `tailwind.config.js` file:

```
tailwind.config.js
```

```
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:

```
app/layout.js
```

```
import '../styles/globals.css'

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

Learn more about [styling with Tailwind CSS](#)

Using App Router together with Pages Router

When navigating between routes served by the different Next.js routers, there will be a hard navigation. Automatic link prefetching with `next/link` will not prefetch across routers.

Instead, you can [optimize navigations ↗](#) between App Router and Pages Router to retain the prefetched and fast page transitions. [Learn more ↗](#)

Codemods

Next.js provides Codemod transformations to help upgrade your codebase when a feature is deprecated. See [Codemods](#) for more information.

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4



How to migrate from Create React App to Next.js

This guide will help you migrate an existing Create React App (CRA) 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 rendering. 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 mitigated with code splitting. However,

if you try to do code splitting manually, you can inadvertently introduce network waterfalls. Next.js provides automatic code splitting and tree-shaking built into its router and build pipeline.

Network waterfalls

A common cause of poor performance occurs when applications make sequential client-server requests to fetch data. One pattern for data fetching in a [SPA](#) is to render a placeholder, and then fetch data after the component has mounted. Unfortunately, a child component can only begin fetching data after its parent has finished loading its own data, resulting in a “waterfall” of requests.

While client-side data fetching is supported in Next.js, Next.js also lets you move data fetching to the server. This often eliminates client-server waterfalls altogether.

Fast and intentional loading states

With built-in support for [streaming through React Suspense](#), you can define which parts of your UI load first and in what order, without creating 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 or component-level basis. For example, you could fetch data from your CMS and render blog posts at build time (SSG) for quick load speeds, or fetch data at request time (SSR) when necessary.

Middleware

Next.js Middleware allows you to run code on the server before a request is completed. For instance, you can avoid a flash of unauthenticated content by redirecting a user to a login page in the middleware for authenticated-only pages. You can also use it for features like A/B testing, experimentation, and [internationalization](#).

Built-in Optimizations

Images, fonts, and third-party scripts often have a large impact on an application's performance. Next.js includes specialized components and APIs that automatically optimize them for you.

Migration Steps

Our goal 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 treat your application as a purely client-side application ([SPA](#)) without immediately replacing your existing router. This reduces complexity and merge conflicts.

Note: If you are using advanced CRA configurations such as a custom `homepage` field in your `package.json`, a custom service worker, or specific Babel/webpack tweaks, please see the **Additional Considerations** section at the end of this guide for tips on replicating or adapting these features in Next.js.

Step 1: Install the Next.js Dependency

Install Next.js in your existing project:



```
npm install next@latest
```

Step 2: Create the Next.js Configuration File

Create a `next.config.ts` at the root of your project (same level as your `package.json`). This file holds your [Next.js configuration options](#).

next.config.ts



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

const nextConfig: NextConfig = {
  output: 'export', // Outputs a Single-Page
  distDir: 'build', // Changes the build output
}

export default nextConfig
```

Note: Using `output: 'export'` means you're doing a static export. You will **not** have access to server-side features like SSR or APIs. You can remove this line to leverage Next.js server features.

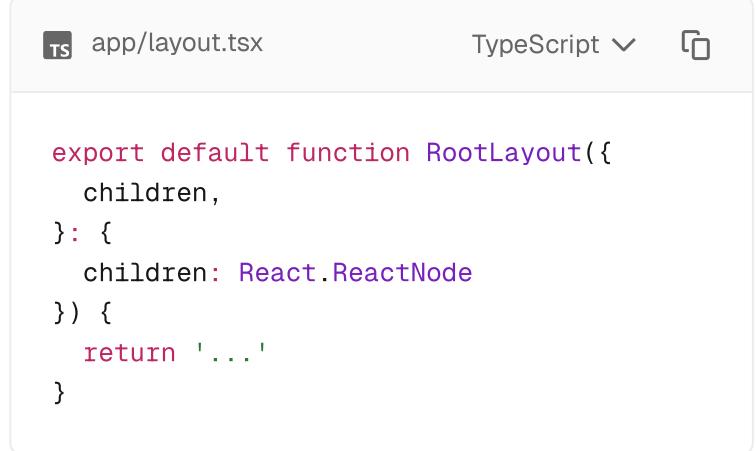
Step 3: Create the Root Layout

A Next.js [App Router](#) application must include a [root layout](#) file, which is a [React Server Component](#) that will wrap all your pages.

The closest equivalent of the root layout file in a CRA application is `public/index.html`, which includes your `<html>`, `<head>`, and `<body>` tags.

1. Create a new `app` directory inside your `src` folder (or at your project root if you prefer `app` at the root).

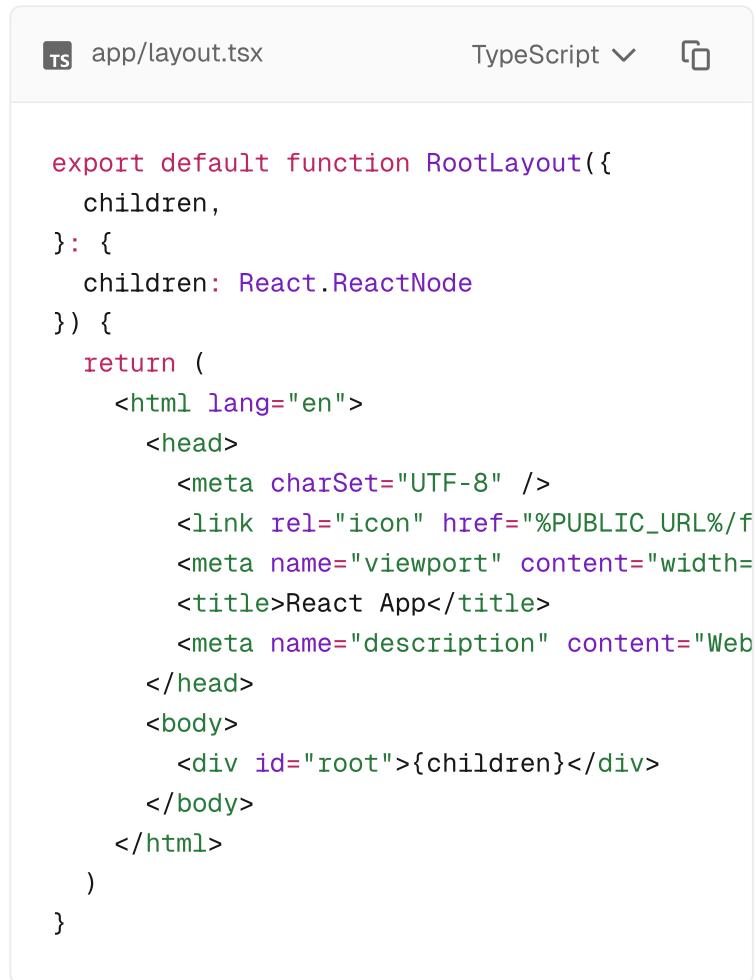
2. Inside the `app` directory, create a `layout.tsx` (or `layout.js`) file:



```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
) {  
  return '<...>'  
}
```

Now copy the content of your old `index.html` into this `<RootLayout>` component. Replace `body div#root` (and `body noscript`) with `<div id="root">{children}</div>`.



```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
) {  
  return (  
    <html lang="en">  
      <head>  
        <meta charSet="UTF-8" />  
        <link rel="icon" href="%PUBLIC_URL%/" />  
        <meta name="viewport" content="width=device-width, initial-scale=1" />  
        <title>React App</title>  
        <meta name="description" content="Webpack + React + TypeScript + Vite" />  
      </head>  
      <body>  
        <div id="root">{children}</div>  
      </body>  
    </html>  
  )  
}
```

Good to know: Next.js ignores CRA's `public/manifest.json`, additional iconography, and [testing configuration](#) by default. If you need these, Next.js has support with its [Metadata API](#) and [Testing setup](#).

Step 4: Metadata

Next.js automatically includes the

<meta charset="UTF-8" /> and

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

tags, so you can remove them from <head>:

The screenshot shows a code editor interface with the following details:

- File Path:** app/layout.tsx
- TypeScript Version:** The status bar indicates "TypeScript 4.5".
- Code Content:** The code defines a React component named `RootLayout`. It takes an array of children as a prop and returns an `html` element with `lang="en"`. The `html` element contains a `head` section with a `link` to an icon, a `title` of "React App", and a `meta` description. The `head` is enclosed in a `body`, which is enclosed in an `html` element.

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
}) {  
  return (  
    <html lang="en">  
      <head>  
        <link rel="icon" href="%PUBLIC_URL%/" />  
        <title>React App</title>  
        <meta name="description" content="Web  
      </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:



```
app/layout.tsx TypeScript ▾
```

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

```
<head>
  <title>React App</title>
  <meta name="description" content="Web
  </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:



```
TS app/layout.tsx TypeScript ▾
```

```
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>
  )
}
```

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 CRA, Next.js supports [CSS Modules](#) out of the box. It also supports [global CSS imports](#).

If you have a global CSS file, import it into your `app/layout.tsx`:



```
TS app/layout.tsx TypeScript ▾ ⌂

import './index.css'

export const 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>
  )
}
```

If you're using Tailwind CSS, see our [installation docs](#).

Step 6: Create the Entrypoint Page

Create React App uses `src/index.tsx` (or `index.js`) as the entry point. In Next.js (App Router), each folder inside the `app` directory corresponds to a route, and each folder should have a `page.tsx`.

Since we want to keep the app as an SPA for now and intercept **all** routes, we'll use an [optional catch-all route](#).

1. Create a `[[...slug]]` directory inside `app`.

```
app
└ [ ...slug ]
  └ page.tsx
  └ layout.tsx
```

1. Add the following to `page.tsx`:

```
TS app/[...slug]/page.tsx TypeScript ▾ ⌂

export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return '...' // We'll update this
}
```

This tells Next.js to generate a single route for the empty slug (`/`), effectively mapping **all** routes to the same page. This page is a [Server Component](#), prerendered into static HTML.

Step 7: Add a Client-Only Entrypoint

Next, we'll embed your CRA's root App component inside a [Client Component](#) so that all logic remains client-side. If this is your first time using Next.js, it's worth knowing that clients components (by default) are still prerendered on the server. You can think about them as having the additional capability of running client-side JavaScript.

Create a `client.tsx` (or `client.js`) in `app/[...slug]/`:

```
TS app/[...slug]/client.tsx TypeScript ▾ ⌂

'use client'

import dynamic from 'next/dynamic'

const App = dynamic(() => import('../App'))
```

```
export function ClientOnly() {
  return <App />
}
```

- The `'use client'` directive makes this file a **Client Component**.
- The `dynamic` import with `ssr: false` disables server-side rendering for the `<App />` component, making it truly client-only (SPA).

Now update your `page.tsx` (or `page.js`) to use your new component:

```
TS app/[...slug]/page.tsx TypeScript ▾ ⌂

import { ClientOnly } from './client'

export function generateStaticParams() {
  return [{ slug: [''] }]
}

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

Step 8: Update Static Image Imports

In CRA, importing an image file returns 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 `` 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 `` 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 `` dimension attribute's value, which can cause the image to appear distorted.

Keeping the `` 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 `` 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.

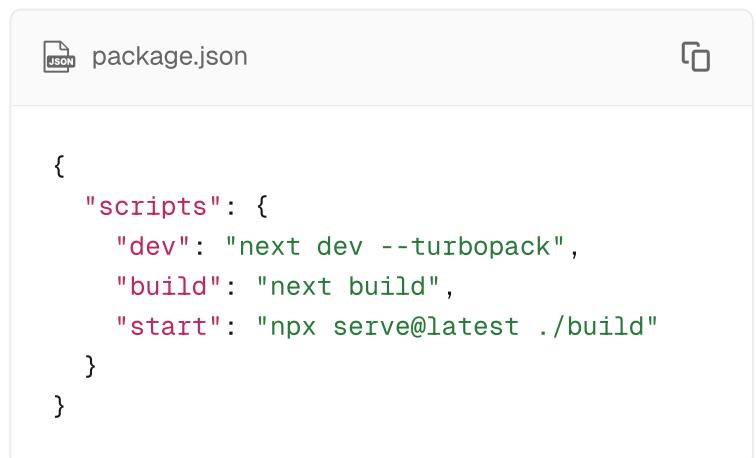
Step 9: Migrate Environment Variables

Next.js supports [environment variables](#) similarly to CRA but **requires** a `NEXT_PUBLIC_` prefix for any variable you want to expose in the browser.

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 10: Update Scripts in `package.json`

Update your `package.json` scripts to use Next.js commands. Also, add `.next` and `next-env.d.ts` to your `.gitignore`:



```
JSON package.json
```

```
{  
  "scripts": {  
    "dev": "next dev --turbopack",  
    "build": "next build",  
    "start": "npx serve@latest ./build"  
  }  
}
```

```
gitignore
```

```
next  
next-env.d.ts  
.next
```

```
# ...
.next
next-env.d.ts
```

Now you can run:

```
npm run dev
```

Open <http://localhost:3000>. You should see your application now running on Next.js (in SPA mode).

Step 11: Clean Up

You can now remove artifacts that are specific to Create React App:

- `public/index.html`
- `src/index.tsx`
- `src/react-app-env.d.ts`
- The `reportWebVitals` setup
- The `react-scripts` dependency (uninstall it from `package.json`)

Additional Considerations

Using a Custom `homepage` in CRA

If you used the `homepage` field in your CRA `package.json` to serve the app under a specific subpath, you can replicate that in Next.js using the `basePath` configuration in `next.config.ts`:

```
TS next.config.ts
```

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

```
const nextConfig: NextConfig = {
  basePath: '/my-subpath',
  // ...
}

export default nextConfig
```

Handling a Custom Service Worker

If you used CRA's service worker (e.g.,

`serviceWorker.js` from `create-react-app`), you can learn how to create [Progressive Web Applications \(PWAs\)](#) with Next.js.

Proxying API Requests

If your CRA app used the `proxy` field in

`package.json` to forward requests to a backend server, you can replicate this with [Next.js rewrites](#) in `next.config.ts`:

```
TS next.config.ts

import { NextConfig } from 'next'

const nextConfig: NextConfig = {
  async rewrites() {
    return [
      {
        source: '/api/:path*',
        destination: 'https://your-backend.co
      },
    ]
  },
}
```

Custom Webpack / Babel Config

If you had a custom webpack or Babel configuration in CRA, you can extend Next.js's config in `next.config.ts`:

```
TS next.config.ts
```

```
import { NextConfig } from 'next'

const nextConfig: NextConfig = {
  webpack: (config, { isServer }) => {
    // Modify the webpack config here
    return config
  },
}

export default nextConfig
```

Note: This will require disabling Turbopack by removing `--turbopack` from your `dev` script.

TypeScript Setup

Next.js automatically sets up TypeScript if you have a `tsconfig.json`. Make sure

`next-env.d.ts` is listed in your `tsconfig.json` `include` array:

```
{
  "include": [
    "next-env.d.ts",
    "app/**/*",
    "src/**/*"
  ]
}
```

Bundler Compatibility

Both Create React App and Next.js default to webpack for bundling. Next.js also offers [Turbopack](#) for faster local development with:

```
next dev --turbopack
```

You can still provide a [custom webpack configuration](#) if you need to migrate advanced webpack settings from CRA.

Next Steps

If everything worked, you now have a functioning Next.js application running as a single-page application. You aren't yet leveraging Next.js features like server-side rendering or file-based routing, but you can now do so incrementally:

- **Migrate from React Router** to the [Next.js App Router](#) for:
 - 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](#)
- **Enable ESLint** with Next.js recommended rules by running `npx next lint` and configuring it to match your project's needs

Note: Using a static export (`output: 'export'`) [does not currently support ↗](#) the `useParams` hook or other server features. To use all Next.js features, remove `output: 'export'` from your `next.config.ts`.

Was this helpful?    



Using App Router
Features available in /app



Latest Version
15.5.4



How to migrate 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:

>_ Terminal

```
npm install next@latest
```

Step 2: Create the Next.js Configuration File

Create a `next.config.mjs` at the root of your project. This file will hold your [Next.js configuration options](#).

`next.config.mjs`

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export', // Outputs a Single-Page
  distDir: './dist', // Changes the build out
}

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"]
  "exclude": [".node_modules"]
}
```

You can find more information about configuring TypeScript on the [Next.js docs](#).

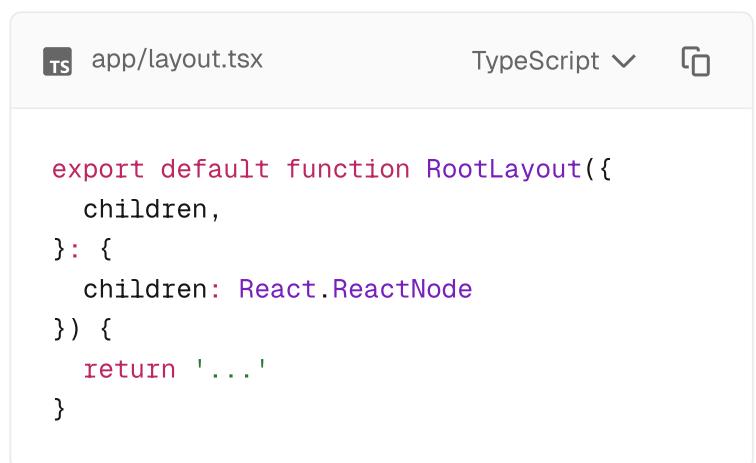
Step 4: Create the Root Layout

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` folder.
2. Create a new `layout.tsx` file inside that `app` directory:



```
TS app/layout.tsx TypeScript ▾ ⌂

export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
}) {  
  return '...'  
}
```

Good to know: `.js`, `.jsx`, or `.tsx` extensions can be used for Layout files.

1. 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>`:



```
TS app/layout.tsx TypeScript ▾ ⌂

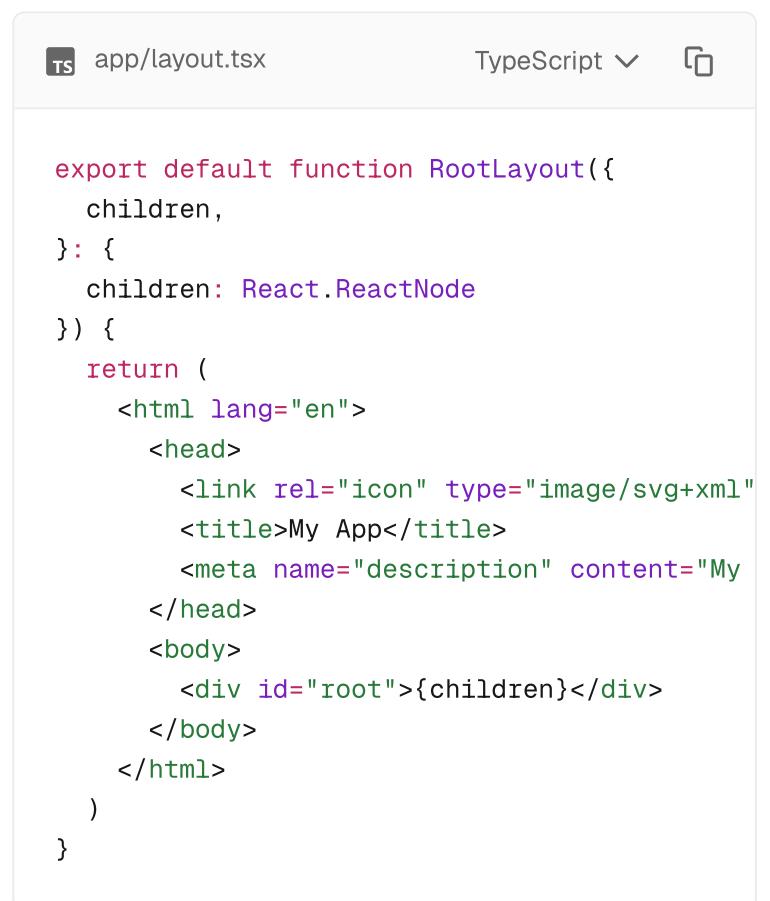
export default function RootLayout({
```

```

        children,
    }: {
        children: React.ReactNode
    ) {
        return (
            <html lang="en">
                <head>
                    <meta charset="UTF-8" />
                    <link rel="icon" type="image/svg+xml"
                    <meta name="viewport" content="width=
                    <title>My App</title>
                    <meta name="description" content="My
                </head>
                <body>
                    <div id="root">{children}</div>
                </body>
            </html>
        )
    }
}

```

1. Next.js already includes by default the `meta charset` ↗ and `meta viewport` ↗ tags, so you can safely remove those from your `<head>`:



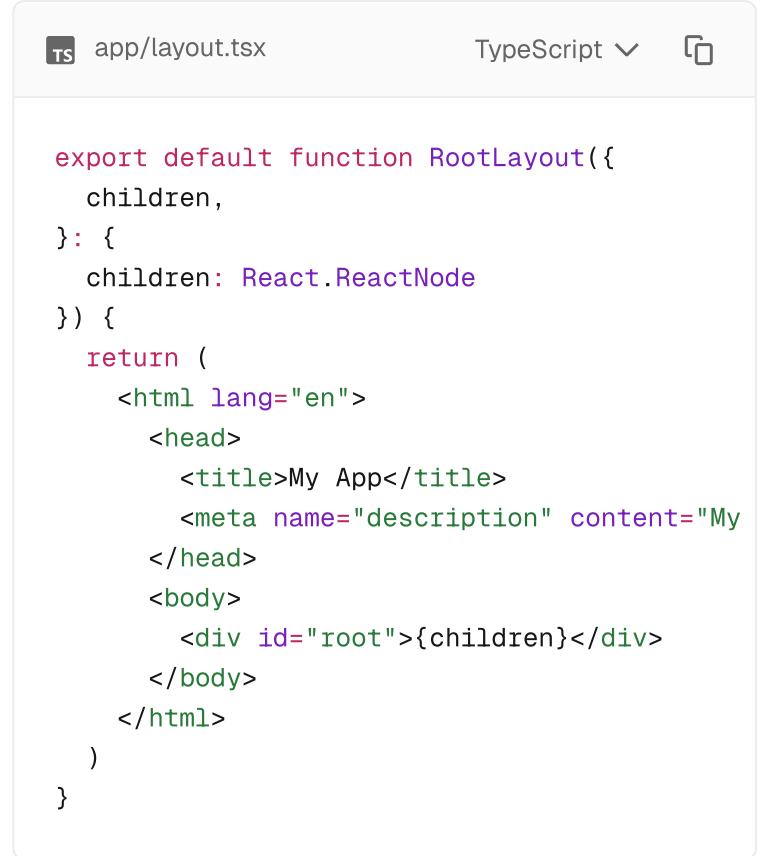
```

TS app/layout.tsx TypeScript ⓘ
export default function RootLayout({
    children,
}: {
    children: React.ReactNode
}) {
    return (
        <html lang="en">
            <head>
                <link rel="icon" type="image/svg+xml"
                <title>My App</title>
                <meta name="description" content="My
            </head>
            <body>
                <div id="root">{children}</div>
            </body>
        </html>
    )
}

```

1. Any `metadata files` such as `favicon.ico`, `icon.png`, `robots.txt` are automatically added to the application `<head>` tag as long as you have

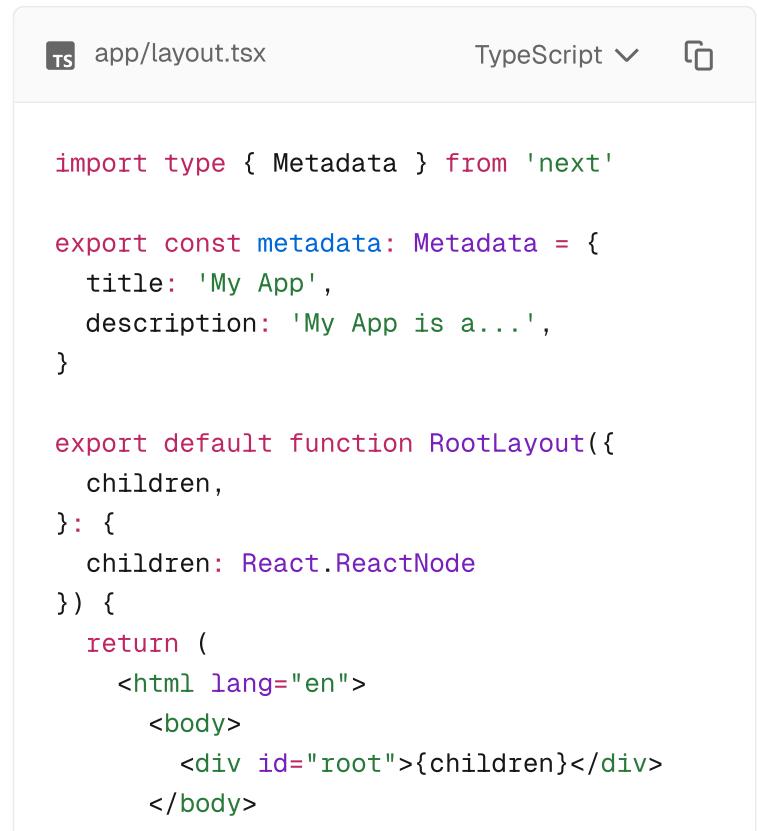
them placed into the top level of the `app` directory. After moving **all supported files** into the `app` directory you can safely delete their `<link>` tags:



```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
}) {  
  return (  
    <html lang="en">  
      <head>  
        <title>My App</title>  
        <meta name="description" content="My  
      </head>  
      <body>  
        <div id="root">{children}</div>  
      </body>  
    </html>  
  )  
}
```

1. Finally, Next.js can manage your last `<head>` tags with the **Metadata API**. Move your final metadata info into an exported `metadata object`:



```
TS app/layout.tsx TypeScript ▾
```

```
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>  
  )  
}
```

```
</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 folders 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.

1. Create a new `page.tsx` file inside the `app/[[... slug]]` directory with the following content:

```
TS app/[[...slug]]/page.tsx TypeScript ▾ ⌂
```

```
import '../index.css'

export function generateStaticParams() {
  return [{ slug: [''] }]
}

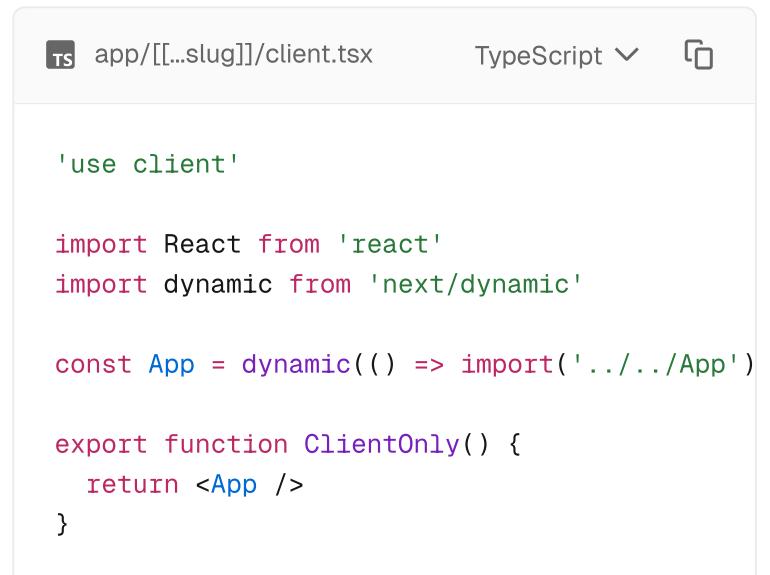
export default function Page() {
  return '...' // We'll update this
}
```

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.



```
TS app/[...slug]/client.tsx TypeScript ▾ ⌂

'use client'

import React from 'react'
import dynamic from 'next/dynamic'

const App = dynamic(() => import('../App'))

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'))
```

Now, update your entrypoint page to use the new component:



```
TS app/[...slug]/page.tsx TypeScript ▾
```

```
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:



```
TS App.tsx
```

```
import image from './img.png' // `image` will
```

```
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 `` 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 `` 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 `` dimension attribute's value, which can cause the image to appear distorted.

Keeping the `` 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'
```

1. Pass the image `src` property instead of the whole image object to your `` 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"
```

1. 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
  distDir: './dist', // Changes the build out
  basePath: process.env.NEXT_PUBLIC_BASE_PATH
}

export default nextConfig
```

1. 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](#)

Was this helpful?    



Using App Router

Features available in /app



Latest Version

15.5.4

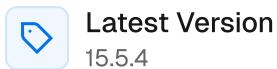
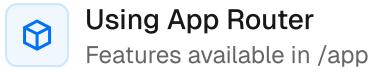


How to build multi-tenant apps in Next.js

If you are looking to build a single Next.js application that serves multiple tenants, we have [built an example ↗](#) showing our recommended architecture.

Was this helpful?





How to build micro-frontends using multi-zones and Next.js

▼ Examples

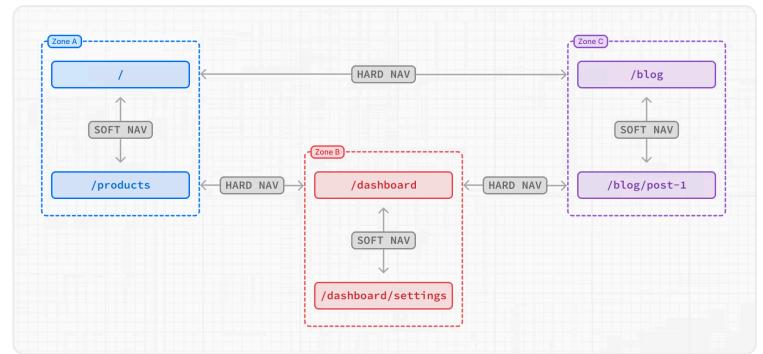
- [With Zones ↗](#)

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.

```
JS next.config.js Copy
/* @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.



A screenshot of a code editor showing the file `next.config.js`. The code defines a `nextConfig` object with an `assetPrefix` of `'/blog-static'` and an `async rewrites()` function. The `rewrites` function returns an array of `beforeFiles` objects, each with a `source` of `'/blog-static/_next/:path+'` and a `destination` of `'/_next/:path+'`.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  assetPrefix: '/blog-static',
  async rewrites() {
    return [
      {
        beforeFiles: [
          {
            source: '/blog-static/_next/:path+',
            destination: '/_next/:path+',
          },
        ],
      },
    ],
  },
}

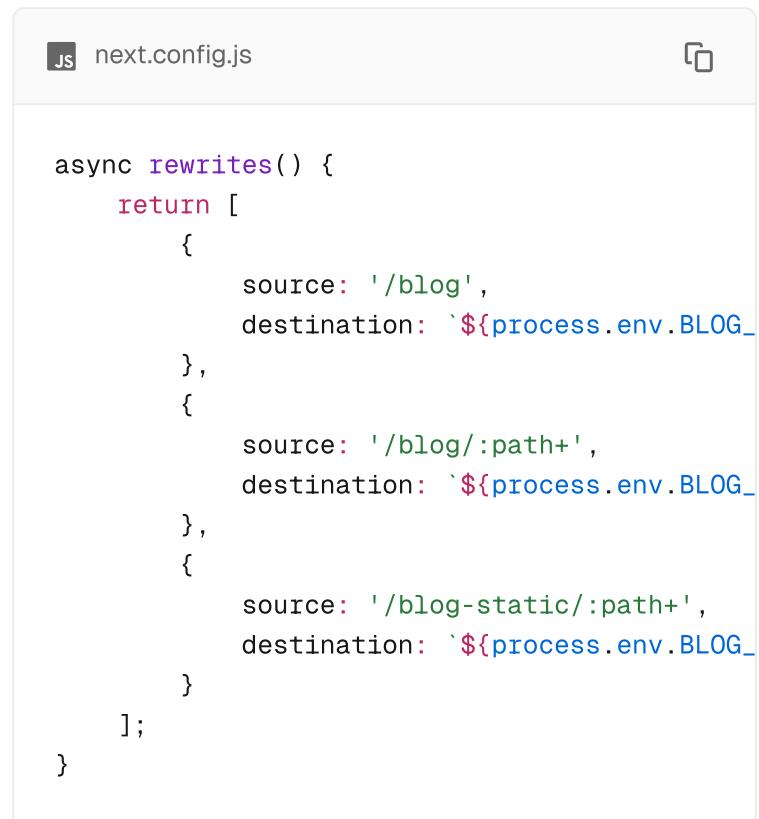
export default nextConfig
```

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, and you also need to rewrite the requests for the static assets. For example:



```
next.config.js

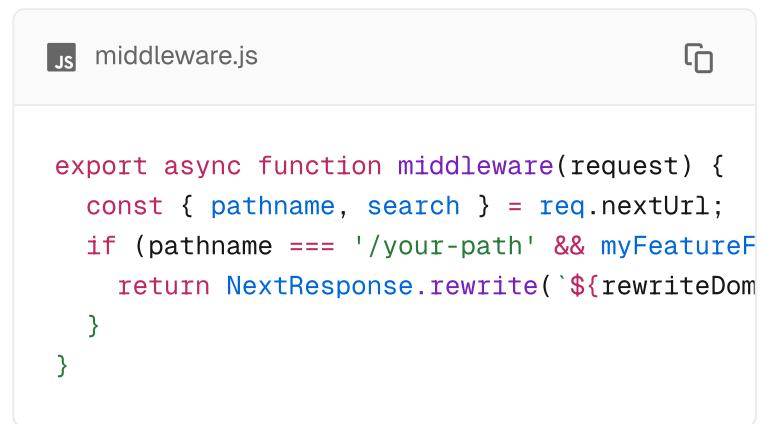
async rewrites() {
  return [
    {
      source: '/blog',
      destination: `${process.env.BLOG_}`,
    },
    {
      source: '/blog/:path+',
      destination: `${process.env.BLOG_}`,
    },
    {
      source: '/blog-static/:path+',
      destination: `${process.env.BLOG_}`,
    }
  ];
}
```

`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.



```
JS middleware.js

export async function middleware(request) {
  const { pathname, search } = req.nextUrl;
  if (pathname === '/your-path' && myFeatureF
    return NextResponse.rewrite(`${
      rewriteDom
    })
}
```

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.

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:

```
JS next.config.js ✖  
  
const nextConfig = {  
  experimental: {  
    serverActions: {  
      allowedOrigins: ['your-production-domai  
    },  
  },  
}
```

See [`serverActions.allowedOrigins`](#) for more information.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



How to set up instrumentation with OpenTelemetry

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.

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-1
```

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



```
your-project/instrumentation... TypeScript ▾
```

```
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.

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.

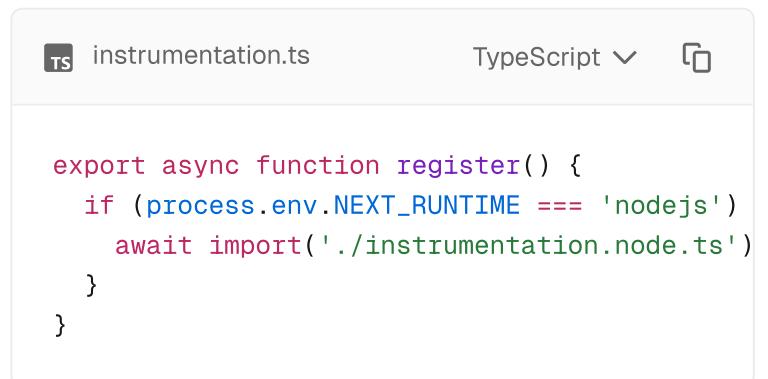
Firstly you need to install OpenTelemetry packages:



```
npm install @opentelemetry/sdk-node @opentelemetry/instrumentation
```

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')
}
```

```
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace'
import { Resource } from '@opentelemetry/resource'
import { NodeSDK } from '@opentelemetry/sdk-node'
import { SimpleSpanProcessor } from '@opentelemetry/processor-simple-span'
import { ATTR_SERVICE_NAME } from '@opentelemetry/const'

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 ↗](#).

A terminal window icon with a magnifying glass symbol inside a square frame.

```
❯ Terminal
npm install @opentelemetry/api
```

The following example demonstrates a function that fetches GitHub stars and adds a custom `fetchGithubStars` span to track the fetch request's result:

```
import { trace } from '@opentelemetry/api'

export async function fetchGithubStars() {
  return await trace
    .getTracer('nextjs-example')
    .startActiveSpan('fetchGithubStars', async
      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.

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4



How to optimize package bundling

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.

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`.

JS next.config.js



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

const withBundleAnalyzer = require('@next/bun
  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.

JS next.config.js

```
/** @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](#).

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`.

JS 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](#).

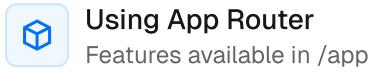
Next Steps

Learn more about optimizing your application for production.

Production

Recommendations to ensure the best performance and...

Was this helpful?    



Using App Router

Features available in /app



Prefetching



Latest Version

15.5.4



Prefetching makes navigating between different routes in your application feel instant. Next.js tries to intelligently prefetch by default, based on the links used in your application code.

This guide will explain how prefetching works and show common implementation patterns:

- [Automatic prefetch](#)
- [Manual prefetch](#)
- [Hover-triggered prefetch](#)
- [Extending or ejecting link](#)
- [Disabled prefetch](#)

How does prefetching work?

When navigating between routes, the browser requests assets for the page like HTML and JavaScript files. Prefetching is the process of fetching these resources *ahead* of time, before you navigate to a new route.

Next.js automatically splits your application into smaller JavaScript chunks based on routes. Instead of loading all the code upfront like traditional SPAs, only the code needed for the current route is loaded. This reduces the initial load time while other parts of the app are loaded in the background. By the time you click the link, the

resources for the new route have already been loaded into the browser cache.

When navigating to the new page, there's no full page reload or browser loading spinner. Instead, Next.js performs a [client-side transition](#), making the page navigation feel instant.

Prefetching static vs. dynamic routes

	Static page	Dynamic page
Prefetched	Yes, full route	No, unless loading.js
Client Cache TTL	5 min (default)	Off, unless enabled
Server roundtrip on click	No	Yes, streamed after shell

Good to know: During the initial navigation, the browser fetches the HTML, JavaScript, and React Server Components (RSC) Payload. For subsequent navigations, the browser will fetch the RSC Payload for Server Components and JS bundle for Client Components.

Automatic prefetch



```
TS app/ui/nav-link.tsx TypeScript ▾ ⌂
import Link from 'next/link'

export default function NavLink() {
```

```
    return <Link href="/about">About</Link>
}
```

Context	Prefetched payload	Client Cache TTL
No <code>loading.js</code>	Entire page	Until app reload
With <code>loading.js</code>	Layout to first loading boundary	30s (configurable)

Automatic prefetching runs only in production.

Disable with `prefetch={false}` or use the wrapper in [Disabled Prefetch](#).

Manual prefetch

```
'use client'

import { useRouter } from 'next/navigation'

const router = useRouter()
router.prefetch('/pricing')
```

Call `router.prefetch()` to warm routes outside the viewport or in response to analytics, hover, scroll, etc.

Hover-triggered prefetch

Proceed with caution: Extending `Link` opts you into maintaining prefetching, cache invalidation, and accessibility concerns. Proceed only if defaults are insufficient.

Next.js tries to do the right prefetching by default, but power users can eject and modify based on their needs. You have the control between performance and resource consumption.

For example, you might have to only trigger prefetches on hover, instead of when entering the viewport (the default behavior):

```
'use client'

import Link from 'next/link'
import { useState } from 'react'

export function HoverPrefetchLink({
  href,
  children,
}: {
  href: string
  children: React.ReactNode
}) {
  const [active, setActive] = useState(false)

  return (
    <Link
      href={href}
      prefetch={active ? null : false}
      onMouseEnter={() => setActive(true)}
    >
      {children}
    </Link>
  )
}
```

`prefetch=null` restores default (static) prefetching once the user shows intent.

Extending or ejecting link

You can extend the `<Link>` component to create your own custom prefetching strategy. For example, using the [ForesightJS ↗](#) library which

prefetches links by predicting the direction of the user's cursor.

Alternatively, you can use `useRouter` to recreate some of the native `<Link>` behavior. However, be aware this opts you into maintaining prefetching and cache invalidation.

```
'use client'

import { useRouter } from 'next/navigation'
import { useEffect } from 'react'

function ManualPrefetchLink({
  href,
  children,
}: {
  href: string
  children: React.ReactNode
}) {
  const router = useRouter()

  useEffect(() => {
    let cancelled = false
    const poll = () => {
      if (!cancelled) router.prefetch(href, {
        })
      poll()
    }
    return () => {
      cancelled = true
    }
  }, [href, router])

  return (
    <a
      href={href}
      onClick={(event) => {
        event.preventDefault()
        router.push(href)
      }}
    >
      {children}
    </a>
  )
}
```

`onInvalidate` is invoked when Next.js suspects cached data is stale, allowing you to refresh the prefetch.

Good to know: Using an `a` tag will cause a full page navigation to the destination route, you can use `onClick` to prevent the full page navigation, and then invoke `router.push` to navigate to the destination.

Disabled prefetch

You can fully disable prefetching for certain routes for more fine-grained control over resource consumption.

```
'use client'

import Link, { LinkProps } from 'next/link'

function NoPrefetchLink({
  prefetch,
  ...rest
}: LinkProps & { children: React.ReactNode }) {
  return <Link {...rest} prefetch={false} />
}
```

For example, you may still want to have consistent usage of `<Link>` in your application, but links in your footer might not need to be prefetched when entering the viewport.

Prefetching optimizations

Good to know: Layout deduplication and prefetch scheduling are part of upcoming optimizations. Currently available in Next.js canary via the `experimental.clientSegmentCache` flag.

Client cache

Next.js stores prefetched React Server Component payloads in memory, keyed by route segments.

When navigating between sibling routes (e.g.

/dashboard/settings → /dashboard/analytics

), it reuses the parent layout and only fetches the updated leaf page. This reduces network traffic and improves navigation speed.

Prefetch scheduling

Next.js maintains a small task queue, which prefetches in the following order:

1. Links in the viewport
2. Links showing user intent (hover or touch)
3. Newer links replace older ones
4. Links scrolled off-screen are discarded

The scheduler prioritizes likely navigations while minimizing unused downloads.

Partial Prerendering (PPR)

When PPR is enabled, a page is divided into a static shell and a streamed dynamic section:

- The shell, which can be prefetched, streams immediately
- Dynamic data streams when ready
- Data invalidations (`revalidateTag`, `revalidatePath`) silently refresh associated prefetches

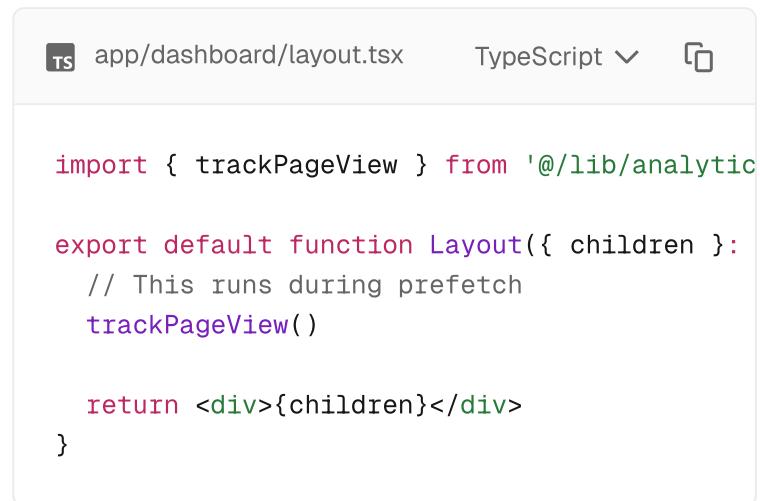
Troubleshooting

**Triggering unwanted side-effects
during prefetching**

If your layouts or pages are not [pure ↗](#) and have side-effects (e.g. tracking analytics), these might be triggered when the route is prefetched, not when the user visits the page.

To avoid this, you should move side-effects to a `useEffect` hook or a Server Action triggered from a Client Component.

Before:



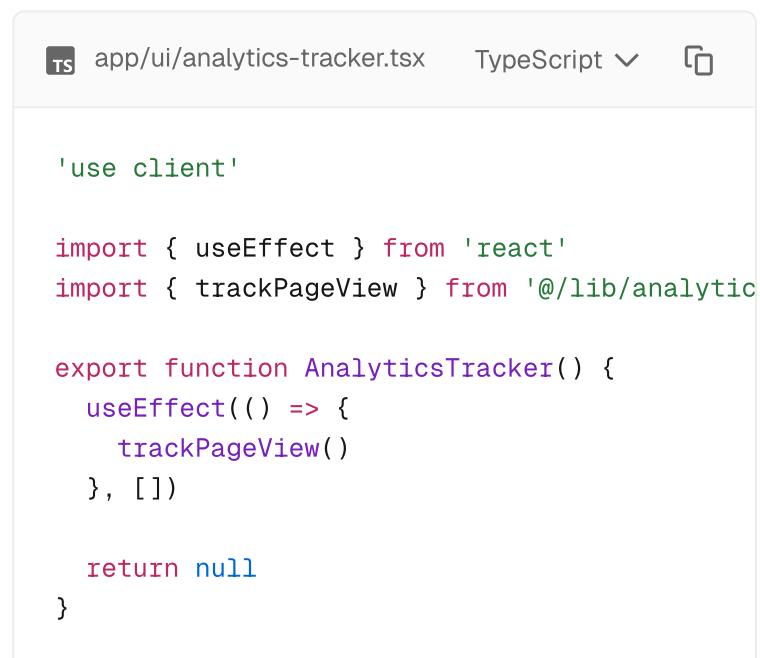
The screenshot shows a code editor window for a file named `app/dashboard/layout.tsx`. The file contains the following code:

```
import { trackPageView } from '@/lib/analytics'

export default function Layout({ children }: { children: React.ReactNode }) {
  // This runs during prefetch
  trackPageView()

  return <div>{children}</div>
}
```

After:



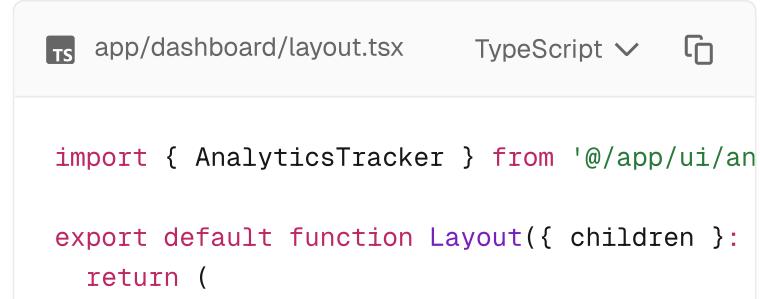
The screenshot shows a code editor window for a file named `app/ui/analytics-tracker.tsx`. The file contains the following code:

```
'use client'

import { useEffect } from 'react'
import { trackPageView } from '@/lib/analytics'

export function AnalyticsTracker() {
  useEffect(() => {
    trackPageView()
  }, [])

  return null
}
```



The screenshot shows a code editor window for a file named `app/dashboard/layout.tsx`. The file contains the following code:

```
import { AnalyticsTracker } from '@/app/ui/analytics'

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

```
<div>
  <AnalyticsTracker />
  {children}
</div>
)
}
```

Preventing too many prefetches

Next.js automatically prefetches links in the viewport when using the `<Link>` component.

There may be cases where you want to prevent this to avoid unnecessary usage of resources, such as when rendering a large list of links (e.g. an infinite scroll table).

You can disable prefetching by setting the `prefetch` prop of the `<Link>` component to `false`.

```
TS app/ui/no-prefetch-link.tsx TypeScript ▾ ⌂
<Link prefetch={false} href={`/blog/${post.id}
  {post.title}
</Link>
```

However, this means static routes will only be fetched on click, and dynamic routes will wait for the server to render before navigating.

To reduce resource usage without disabling prefetch entirely, you can defer prefetching until the user hovers over a link. This targets only links the user is likely to visit.

```
TS app/ui/hover-prefetch-link.tsx TypeScript ▾ ⌂
'use client'

import Link from 'next/link'
import { useState } from 'react'
```

```
export function HoverPrefetchLink({  
  href,  
  children,  
}: {  
  href: string  
  children: React.ReactNode  
) {  
  const [active, setActive] = useState(false)  
  
  return (  
    <Link  
      href={href}  
      prefetch={active ? null : false}  
      onMouseEnter={() => setActive(true)}  
    >  
    {children}  
    </Link>  
  )  
}
```

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



How to optimize your Next.js application for 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](#) and [before going to production](#), 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.
- **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.

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.
- **Client and Server Components:** 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.

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.

UI and accessibility

- **Forms and Validation:** Use Server Actions to handle form submissions, server-side validation, and handle errors.
- **Global Error UI:** Add `app/global-error.tsx` to provide consistent, accessible fallback UI and recovery for uncaught errors across your app.
- **Global 404:** Add `app/global-not-found.tsx` to serve an accessible 404 for unmatched routes across your app.
- **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.
- **<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-javascript-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.

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 ↗](#)

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4



How to build a Progressive Web Application (PWA) with Next.js

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:



The screenshot shows a code editor window with the title bar "app/manifest.ts" and "TypeScript". The code itself is a TypeScript file defining a manifest object. It includes fields for name, short_name, description, start_url, display, background_color, theme_color, and icons (with both 192x192 and 512x512 pixel sizes).

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

export default function manifest(): MetadataRoute {
  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, send

function urlBase64ToUint8Array(base64String:
  const padding = '='.repeat((4 - (base64Stri
  const base64 = (base64String + padding).rep

  const rawData = window.atob(base64)
  const outputArray = new Uint8Array(rawData.

    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(null)
}

const [message, setMessage] = useState('')

useEffect(() => {
  if ('serviceWorker' in navigator && 'PushManager' in navigator)
    setIsSupported(true)
    registerServiceWorker()
}, [])

async function registerServiceWorker() {
  const registration = await navigator.serviceWorker.register(
    '/',
    { updateViaCache: 'none' }
  )
  const sub = await registration.pushManager.getSubscription()
  setSubscription(sub)
}

async function subscribeToPush() {
  const registration = await navigator.serviceWorker.register(
    '/',
    { userVisibleOnly: true,
      applicationServerKey: urlBase64ToInt8Array(
        process.env.NEXT_PUBLIC_VAPID_PUBLIC_KEY
      ),
      vapidPublicKey: process.env.NEXT_PUBLIC_VAPID_PUBLIC_KEY
    }
  )
  setSubscription(sub)
  const serializedSub = JSON.parse(JSON.stringify(sub))
  await subscribeUser(serializedSub)
}

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</p>
}

return (

```

```

<div>
  <h3>Push Notifications</h3>
  {subscription ? (
    <>
      <p>You are subscribed to push notif
      <button onClick={unsubscribeFromPus
      <input
        type="text"
        placeholder="Enter notification m
        value={message}
        onChange={(e) => setMessage(e.tar
      />
      <button onClick={sendTestNotificati
    </>
  ) : (
    <>
      <p>You are not subscribed to push n
      <button onClick={subscribeToPush}>S
    </>
  )}
  </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(true)

  useEffect(() => {
    setIsIOS(
      /iPad|iPhone|iPod/.test(navigator.userAgent)
    )

    setIsStandalone(window.matchMedia('(display
  }, [])

  if (isStandalone) {
    return null // Don't show install button
  }

  return (
    <div>
      <h3>Install App</h3>
      <button>Add to Home Screen</button>
      {isIOS && (
        <p>

```

```

To install this app on your iOS dev
<span role="img" aria-label="share
    {' '}
    &{' '}
</span>
and then "Add to Home Screen"
<span role="img" aria-label="plus i
    {' '}
    +{' '}
</span>.

</p>
)
</div>
)
}
}

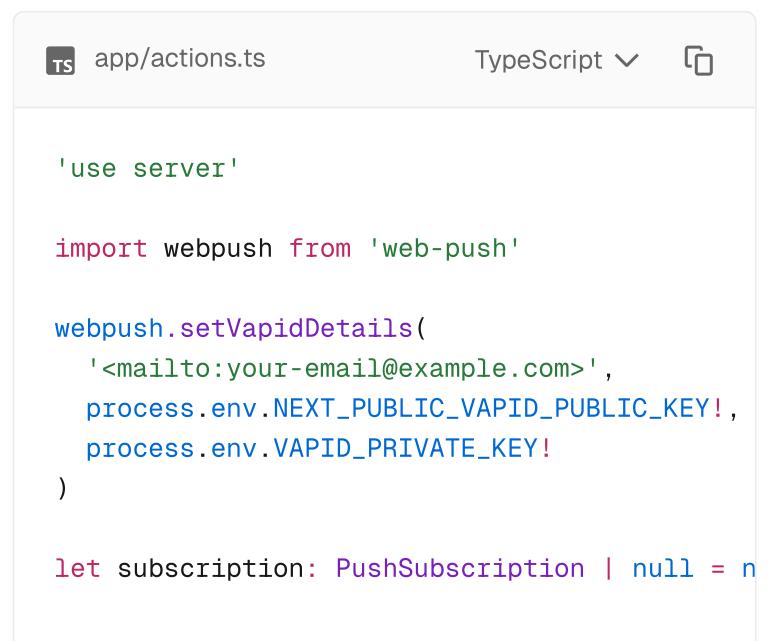
export default function Page() {
  return (
    <div>
      <PushNotificationManager />
      <InstallPrompt />
    </div>
  )
}

```

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.



```

TS app/actions.ts TypeScript ▾ ⌂
'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 = n

```

```

        export async function subscribeUser(sub: PushSubscription) {
            subscription = sub
            // In a production environment, you would want to store this in a database
            // For example: await db.subscriptions.create(sub)
            return { success: true }
        }

        export async function unsubscribeUser() {
            subscription = null
            // In a production environment, you would want to delete this from a database
            // For example: await db.subscriptions.delete(sub)
            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')
                return { success: false, error: 'Failed to send push 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. The simplest way is to use the web-push CLI directly:

First, install web-push globally:

>_ Terminal



```
npm install -g web-push
```

Generate the VAPID keys by running:

>_ Terminal



```
web-push generate-vapid-keys
```

Copy the output and paste the keys into your `.env` file:

```
NEXT_PUBLIC_VAPID_PUBLIC_KEY=your_public_key_here  
VAPID_PRIVATE_KEY=your_private_key_here
```

5. Creating a Service Worker

Create a `public/sw.js` file for your service worker:

JS public/sw.js



```
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.showNot  
  }  
})  
  
self.addEventListener('notificationclick', fu
```

```
console.log('Notification click received.')
event.notification.close()
event.waitUntil(clients.openWindow('<https:
})
```

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).

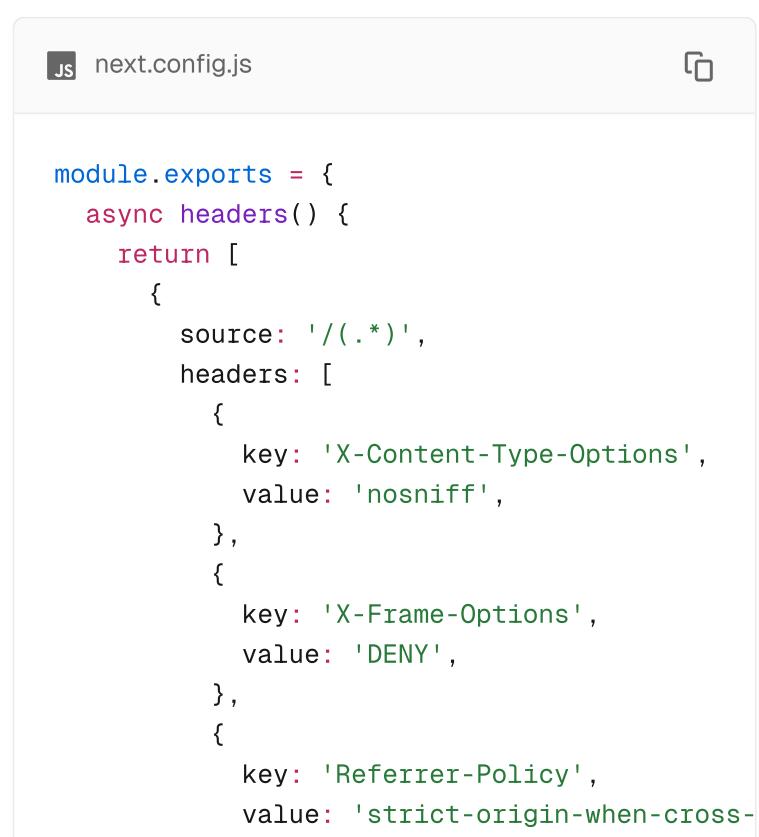
7. Testing Locally

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:



```
JS next.config.js
```

```
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-
```

```
        },
        ],
        {
            source: '/sw.js',
            headers: [
                {
                    key: 'Content-Type',
                    value: 'application/javascript; c
                },
                {
                    key: 'Cache-Control',
                    value: 'no-cache, no-store, must-
                },
                {
                    key: 'Content-Security-Policy',
                    value: "default-src 'self'; scrip
                },
            ],
        },
    ],
},
}
```

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.

Extending your PWA

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.

Next Steps

manifest.json

API Reference for manifest.json file.

Was this helpful?    



Using App Router
Features available in /app



Latest Version
15.5.4



How to handle redirects in Next.js

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.

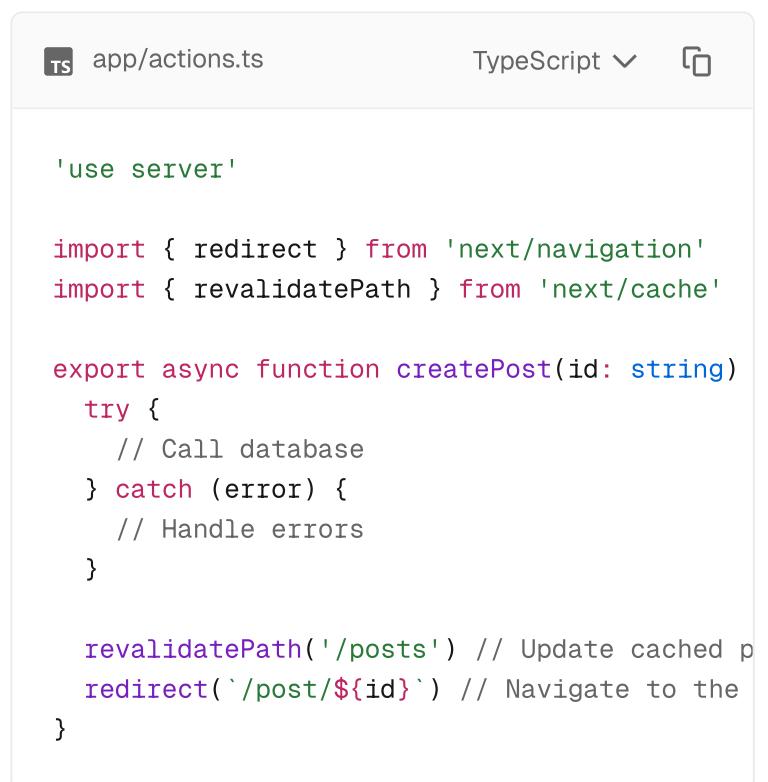
API	Purpose	Where
<code>redirect</code>	Redirect user after a mutation or event	Server Components, Server Actions, Route Handlers
<code>permanentRedirect</code>	Redirect user after a mutation or event	Server Components, Server Actions, Route Handlers
<code>useRouter</code>	Perform a client-side navigation	Event Handlers in Client Components
<code>redirects in next.config.js</code>	Redirect an incoming request based on a path	<code>next.config.js</code> file
<code>NextResponse.redirect</code>	Redirect an incoming	Middleware

API	Purpose	Where
	request based on a condition	

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:



```
TS app/actions.ts TypeScript ▾ ⌂
'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 p
  redirect(`/post/${id}`) // Navigate to the
}
```

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.

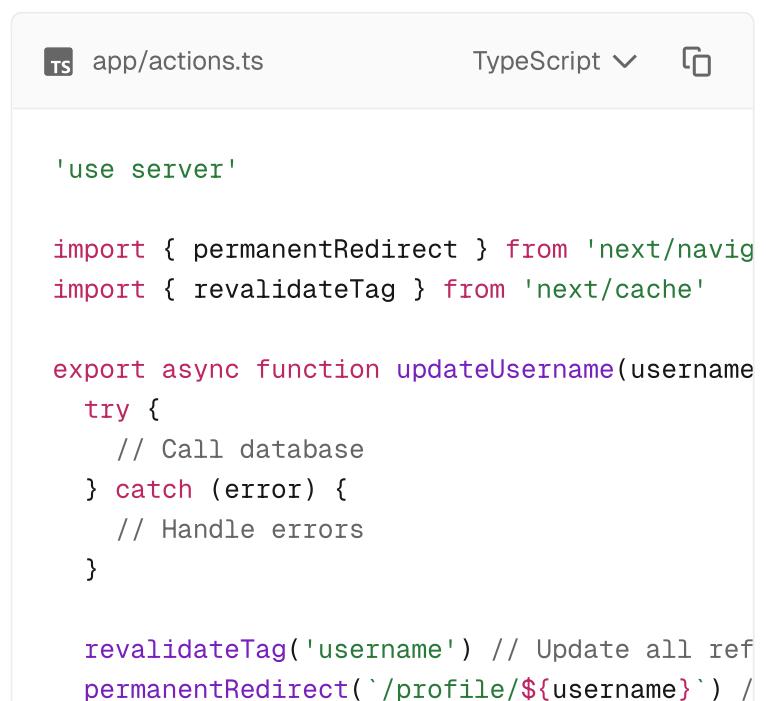
- `redirect` throws an error so it should be called **outside** the `try` block when using `try/catch` statements.
- `redirect` can be called in Client Components during the rendering process but not in event handlers. You can use the `useRouter` hook instead.
- `redirect` also accepts absolute URLs 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:



```
TS app/actions.ts TypeScript ⓘ

'use server'

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

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

  revalidateTag('username') // Update all refs
  permanentRedirect(`profile/${username}`) /
}
```

}

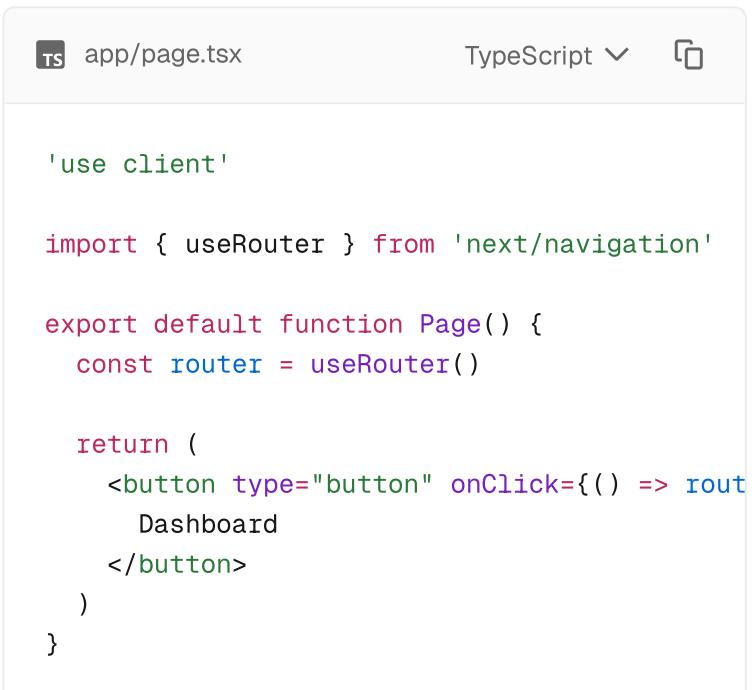
Good to know:

- `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:



```
TS app/page.tsx TypeScript ▾ ⌂

'use client'

import { useRouter } from 'next/navigation'

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.

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:

```
TS next.config.ts TypeScript ▾
```

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

const nextConfig: NextConfig = {
  async redirects() {
    return [
      // Basic redirect
      {
        source: '/about',
        destination: '/',
        permanent: true,
      },
      // Wildcard path matching
      {
        source: '/blog/:slug',
        destination: '/news/:slug',
        permanent: true,
      },
    ]
  },
}

export default nextConfig
```

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

Good to know:

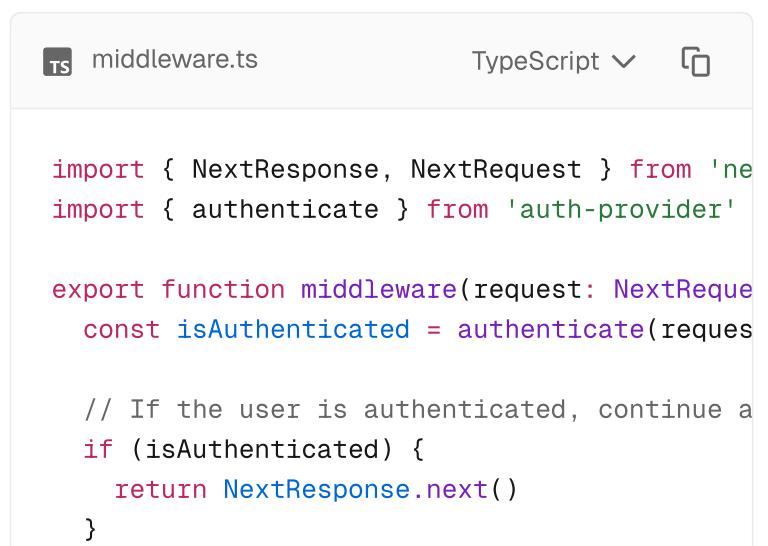
- `redirects` can return a 307 (Temporary Redirect) or 308 (Permanent Redirect) status code with the `permanent` option.
- `redirects` may have a limit on platforms. For example, on Vercel, there's a limit of 1,024 redirects. To manage a large number of redirects (1000+), consider creating a custom solution 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:



```
TS middleware.ts TypeScript ▾ ⌂

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 a
  if (isAuthenticated) {
    return NextResponse.next()
  }
}
```

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

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.

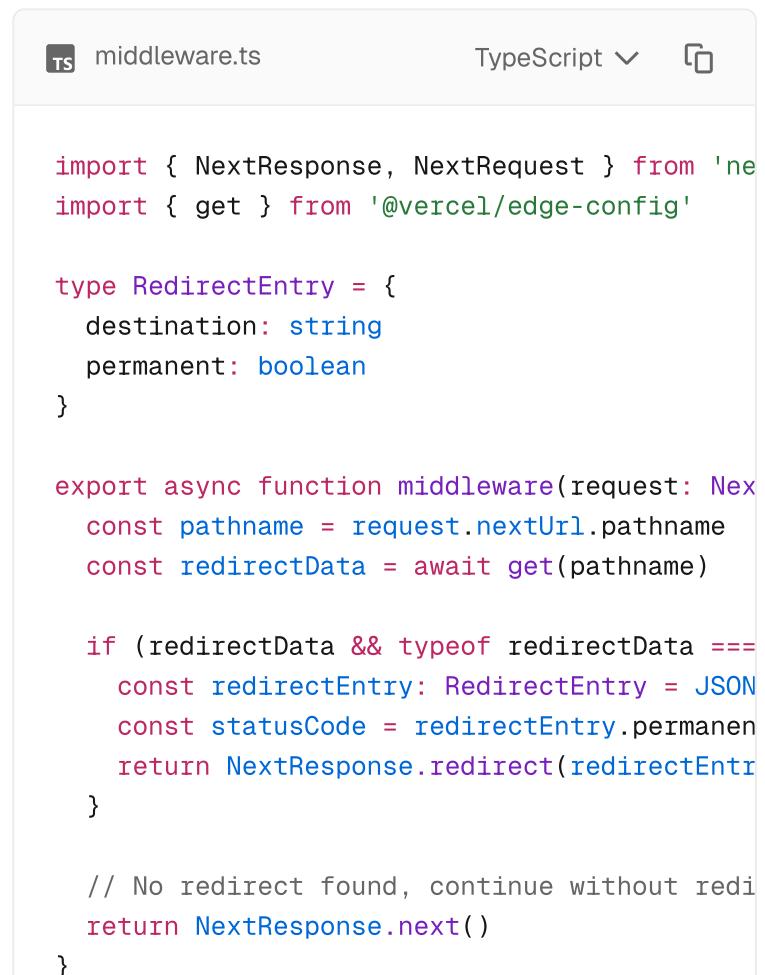
1. Creating and storing a redirect map

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:



```
TS middleware.ts TypeScript ▾ ⌂

import { NextResponse, NextRequest } from 'next'
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 === 'object') {
    const redirectEntry: RedirectEntry = JSON.parse(redirectData)
    const statusCode = redirectEntry.permanent ? 301 : 302
    return NextResponse.redirect(redirectEntry.destination, { status: statusCode })
  }

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

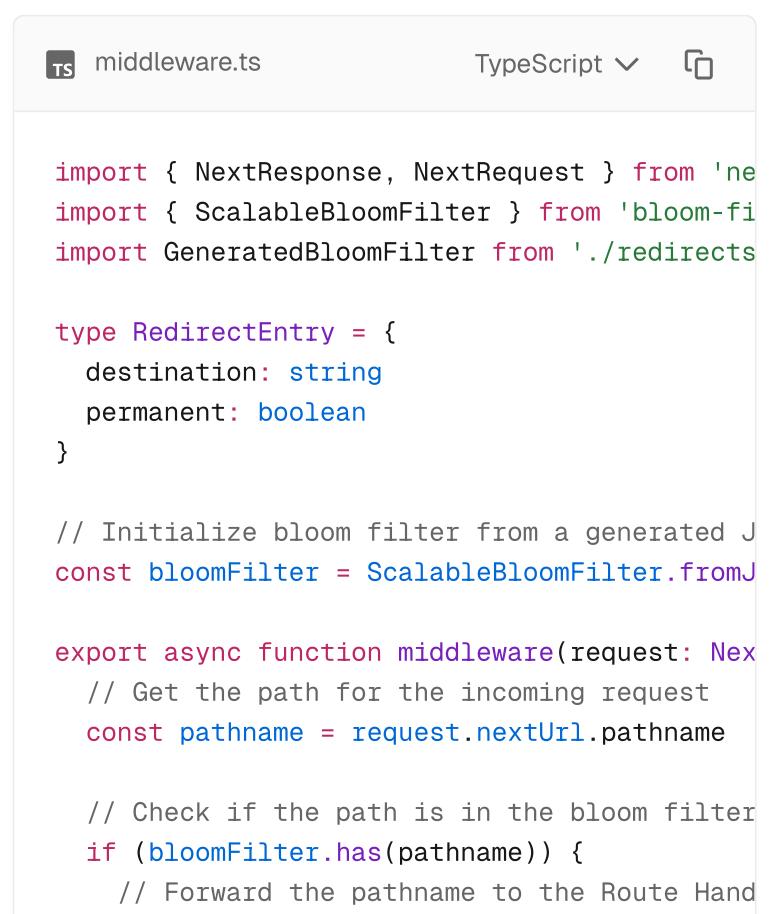
2. Optimizing data lookup performance

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
- 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](#) 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.



The screenshot shows a code editor window with the file name "middleware.ts" at the top left. To the right of the file name are "TypeScript" and a dropdown menu. On the far right is a close button. The code editor displays the following TypeScript code:

```
TS middleware.ts TypeScript ▾
```

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

type RedirectEntry = {
  destination: string
  permanent: boolean
}

// Initialize bloom filter from a generated JSON file
const bloomFilter = ScalableBloomFilter.fromJSONFile('bloom.json')

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
    return NextResponse.redirect(`https://${request.nextUrl.host}${pathname}`)
  }
}
```

```

    const api = new URL(
      `/api/redirects? pathname=${encodeURIComponent(
        request.nextUrl.origin
      )}

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

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

        if (redirectEntry) {
          // Determine the status code
          const statusCode = redirectEntry.statusCode

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

  // No redirect found, continue the request
  return NextResponse.next()
}

```

Then, in the Route Handler:



```

TS app/api/redirects/route.ts TypeScript ▾ ⌂

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

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 redirect object
  const redirect = (redirects as Record<string, RedirectEntry>)[
    pathname
  ]

  // Account for bloom filter false positives
  if (redirect) {
    return NextResponse.redirect(redirect.destination)
  }
}

// No redirect found, continue the request
return NextResponse.next()
}

```

```
if (!redirect) {  
  return new Response('No redirect', { stat  
})  
  
// Return the redirect entry  
return NextResponse.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.

Next Steps

redirect

API Reference for the redirect function.

permanentR...

API Reference for the permanentRedire...

middleware.js

API reference for the middleware.js file.

redirects

Add redirects to your Next.js app.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

How to use Sass

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` ↗:

Terminal

```
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`.

next.config.ts

TypeScript



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

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

```
    },  
}  
  
export default nextConfig
```

Implementation

You can use the `implementation` property to specify the Sass implementation to use. By default, Next.js uses the [sass](#) package.



The screenshot shows a code editor window with the file name `next.config.ts` at the top. The `TypeScript` dropdown is set to `TypeScript`. The code in the editor is:

```
import type { NextConfig } from 'next'  
  
const nextConfig: NextConfig = {  
  sassOptions: {  
    implementation: 'sass-embedded',  
  },  
}  
  
export default nextConfig
```

Sass Variables

Next.js supports Sass variables exported from CSS Module files.

For example, using the exported `primaryColor` Sass variable:



The screenshot shows a code editor window with the file name `app/variables.module.scss` at the top. The code in the editor is:

```
$primary-color: #64ff00;  
  
:export {  
  primaryColor: $primary-color;  
}
```



The screenshot shows a code editor window with the file name `app/page.js` at the top. The code in the editor is:

```
// maps to root `/` URL  
  
import variables from './variables.module.scss'
```

```
export default function Page() {  
  return <h1 style={{ color: variables.primary }}>  
}
```

Was this helpful?    



Using App Router
Features available in /app



Latest Version
15.5.4



How to load and optimize scripts



Layout Scripts

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

```
TS app/dashboard/layout.tsx TypeScript ▾
```

```
import Script from 'next/script'

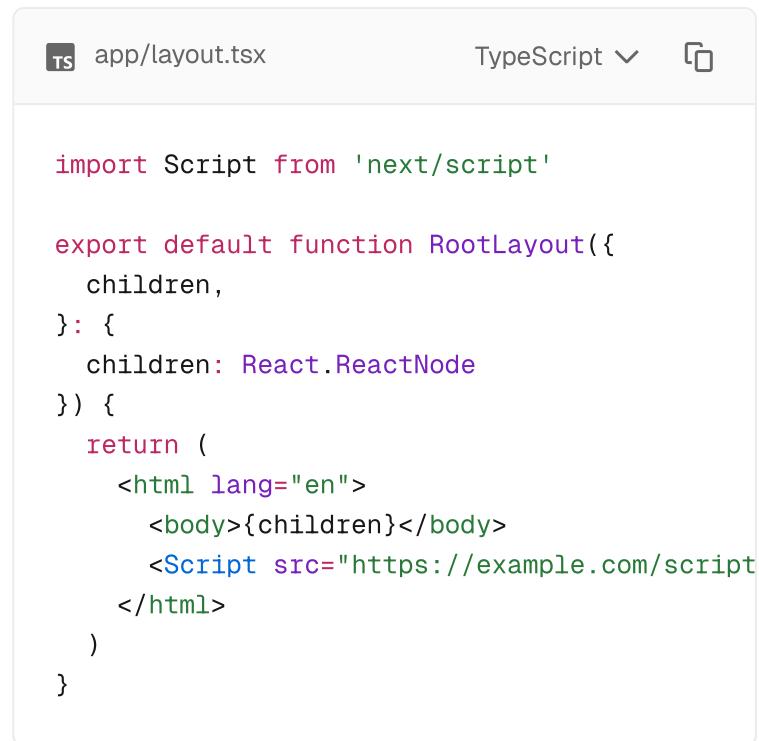
export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <>
      <section>{children}</section>
      <Script src="https://example.com/script" />
    </>
  )
}
```

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:



```
TS app/layout.tsx TypeScript ▾
```

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

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 Router. 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`:

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:

Terminal

```
npm run dev
```

You'll see instructions like these: Please install Partytown by running

```
npm install @builder.io/partytown
```

Once setup is complete, defining

`strategy="worker"` will automatically instantiate Partytown in your application and offload the script to a web worker.



The screenshot shows a code editor interface with a TypeScript file named `pages/home.tsx`. The code defines a `Home` component that returns a `<Script>` component with a `src` attribute pointing to a URL. The code editor has a `TS` icon, a `TypeScript` dropdown, and a copy icon.

```
import Script from 'next/script'

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

There are a number of trade-offs that need to be considered when loading a third-party script in a web worker. Please see Partytown's [tradeoffs ↗](#) documentation for more information.

Inline Scripts

Inline scripts, or scripts not loaded from an external file, are also supported by the `Script` component. They can be written by placing the JavaScript within curly braces:

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

Or by using the `dangerouslySetInnerHTML` property:

```
<Script  
  id="show-banner"  
  dangerouslySetInnerHTML={{  
    __html: `document.getElementById('banner'  
  }}  
/>
```

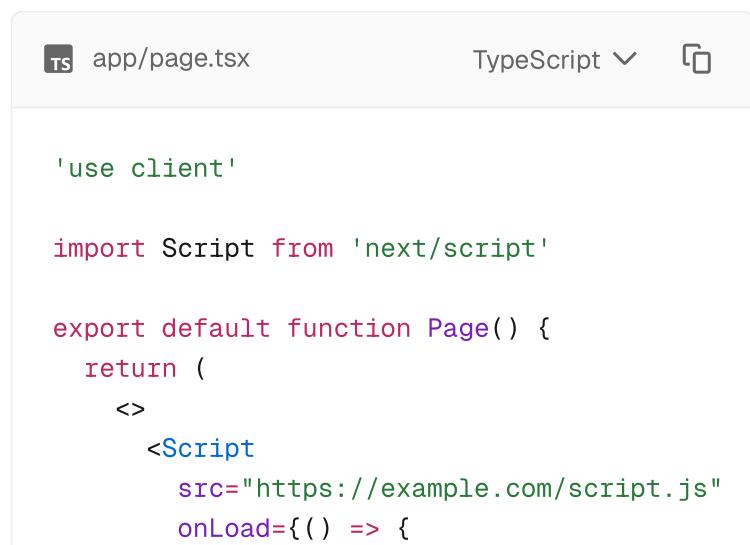
Warning: An `id` property must be assigned for inline scripts in order for Next.js to track and optimize the script.

Executing Additional Code

Event handlers can be used with the `Script` component to execute additional code after a certain event occurs:

- `onLoad` : Execute code after the script has finished loading.
- `onReady` : Execute code after the script has finished loading and every time the component is mounted.
- `onError` : Execute code if the script fails to load.

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



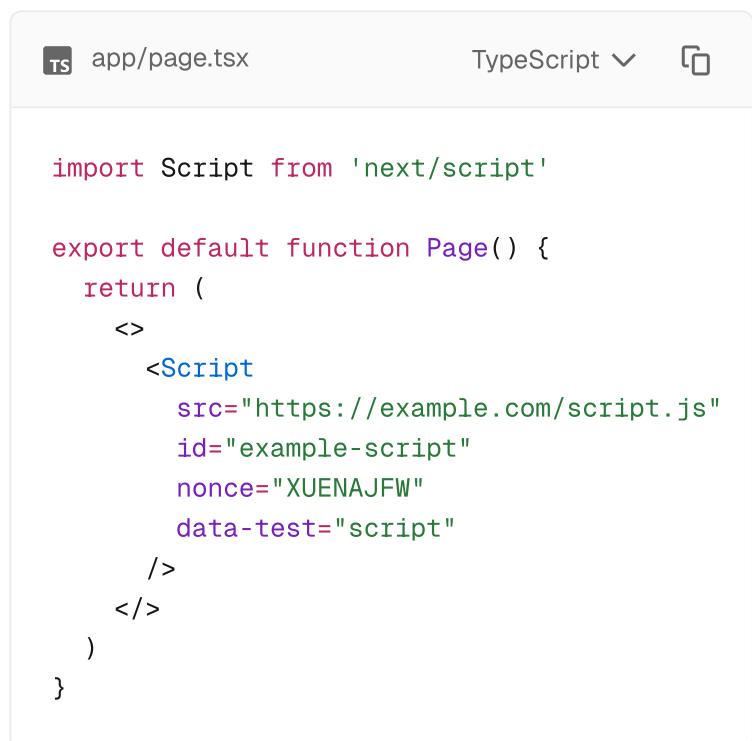
```
TS app/page.tsx TypeScript ▾ ⌂  
  
'use client'  
  
import Script from 'next/script'  
  
export default function Page() {  
  return (  
    <>  
    <Script  
      src="https://example.com/script.js"  
      onLoad={() => {
```

```
        console.log('Script has loaded')
    }
    />
    </>
)
}
```

Refer to the [next/script](#) API reference to learn more about each event handler and view examples.

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.



The screenshot shows a code editor interface with a TypeScript file named `app/page.tsx`. The code defines a `Page` component that returns a `<Script>` component with various attributes: `src`, `id`, `nonce`, and `data-test`.

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

API Reference

Script Comp...

Optimize third-party scripts in your Next.js...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

How to self-host your Next.js application

When [deploying](#) your Next.js app, you may want to configure how different features are handled based on your infrastructure.

 **Watch:** Learn more about self-hosting Next.js → [YouTube \(45 minutes\)](#).

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 the [Edge runtime](#), 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. If you do not want this, you can use the [full Node.js runtime](#) to run Middleware.

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](#).

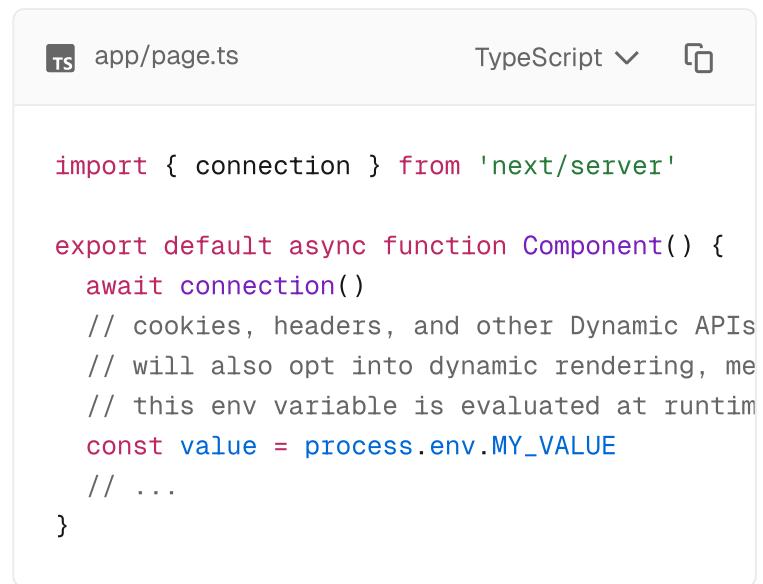
Environment Variables

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`.

You safely read environment variables on the server during dynamic rendering.



```
TS app/page.ts TypeScript ▾ ⌂

import { connection } from 'next/server'

export default async function Component() {
  await connection()
  // cookies, headers, and other Dynamic APIs
  // will also opt into dynamic rendering, me
  // 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:

```
JS next.config.js

module.exports = {
  cacheHandler: require.resolve('./cache-hand
  cacheMaxMemorySize: 0, // disable default i
}
```

Then, create `cache-handler.js` in the root of your project, for example:

```
JS cache-handler.js

const cache = new Map()

module.exports = class CacheHandler {
  constructor(options) {
    this.options = options
  }

  async get(key) {
    // This could be stored anywhere, like du
    return cache.get(key)
}
```

```
}
```

```
async set(key, data, ctx) {
  // This could be stored anywhere, like du
  cache.set(key, {
    value: data,
    lastModified: Date.now(),
    tags: ctx.tags,
  })
}

async revalidateTag(tags) {
  // tags is either a string or an array of
  tags = [tags].flat()
  // Iterate over all entries in the cache
  for (let [key, value] of cache) {
    // If the value's tags include the spec
    if (value.tags.some((tag) => tags.inclu
      cache.delete(key)
    )
  }
}

// If you want to have temporary in memory
// before the next request you can leverage
resetRequestCache() {}

}
```

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`:



```
JS next.config.js

module.exports = {
  generateBuildId: async () => {
    // This could be anything, using the late
    return process.env.GIT_HASH
  },
}
```

Version Skew

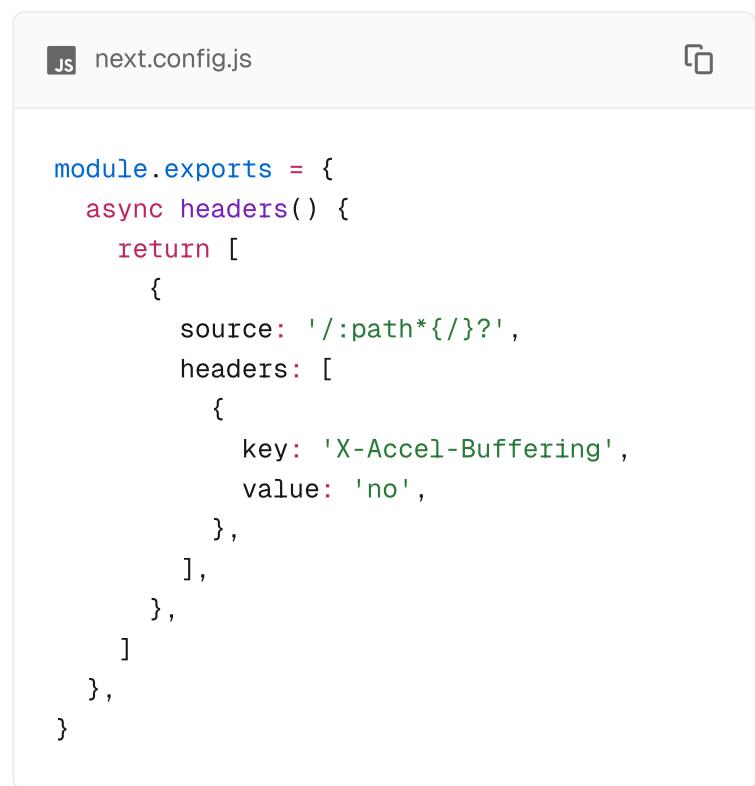
Next.js will automatically mitigate most instances of [version skew](#) ↗ and automatically reload the application to retrieve new assets when detected. For example, if there is a mismatch in the `deploymentId`, transitions between pages will perform a hard navigation versus using a prefetched value.

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.

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`:



```
JS next.config.js

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-only 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-cacheable.

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.

As Partial Prerendering moves to stable, we will provide support through the Deployment Adapters API.

after

`after` is fully supported when self-hosting with `next start`.

When stopping the server, ensure a graceful shutdown by sending `SIGINT` or `SIGTERM` signals and waiting. This allows the Next.js server to wait until after pending callback functions or promises used inside `after` have finished.

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4



How to build single-page applications with Next.js

Next.js fully supports building Single-Page Applications (SPAs).

This includes fast route transitions with prefetching, client-side data fetching, using browser APIs, integrating with third-party client libraries, creating static routes, and more.

If you have an existing SPA, you can migrate to Next.js without large changes to your code. Next.js then allows you to progressively add server features as needed.

What is a Single-Page Application?

The definition of a SPA varies. We'll define a "strict SPA" as:

- **Client-side rendering (CSR):** The app is served by one HTML file (e.g. `index.html`). Every route, page transition, and data fetch is handled by JavaScript in the browser.
- **No full-page reloads:** Rather than requesting a new document for each route, client-side JavaScript manipulates the current page's DOM and fetches data as needed.

Strict SPAs often require large amounts of JavaScript to load before the page can be interactive. Further, client data waterfalls can be challenging to manage. Building SPAs with Next.js can address these issues.

Why use Next.js for SPAs?

Next.js can automatically code split your JavaScript bundles, and generate multiple HTML entry points into different routes. This avoids loading unnecessary JavaScript code on the client-side, reducing the bundle size and enabling faster page loads.

The `next/link` component automatically [prefetches](#) routes, giving you the fast page transitions of a strict SPA, but with the advantage of persisting application routing state to the URL for linking and sharing.

Next.js can start as a static site or even a strict SPA where everything is rendered client-side. If your project grows, Next.js allows you to progressively add more server features (e.g. [React Server Components](#), [Server Actions](#), and more) as needed.

Examples

Let's explore common patterns used to build SPAs and how Next.js solves them.

Using React's `useContext` within a Context Provider

We recommend fetching data in a parent component (or layout), returning the Promise, and then unwrapping the value in a Client Component with React's [use hook ↗](#).

Next.js can start data fetching early on the server. In this example, that's the root layout — the entry point to your application. The server can immediately begin streaming a response to the client.

By “hoisting” your data fetching to the root layout, Next.js starts the specified requests on the server early before any other components in your application. This eliminates client waterfalls and prevents having multiple roundtrips between client and server. It can also significantly improve performance, as your server is closer (and ideally colocated) to where your database is located.

For example, update your root layout to call the Promise, but do *not* await it.



The screenshot shows a code editor window with the following details:

- File name: app/layout.tsx
- TypeScript version: v4.5
- Code content:

```
TS app/layout.tsx TypeScript ⓘ

import { UserProvider } from './user-provider'
import { getUser } from './user' // some serv

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  let userPromise = getUser() // do NOT await

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

While you can [defer and pass a single Promise](#) as a prop to a Client Component, we generally see this pattern paired with a React context provider. This enables easier access from Client Components with a custom React Hook.

You can forward a Promise to the React context provider:

```
TS app/user-provider.ts TypeScript ▾ ⌂

'use client';

import { createContext, useContext, ReactNode }

type User = any;
type UserContextType = {
  userPromise: Promise<User | null>;
};

const UserContext = createContext<UserContextType>();

export function useUser(): UserContextType {
  let context = useContext(UserContext);
  if (context === null) {
    throw new Error('useUser must be used with a provider');
  }
  return context;
}

export function UserProvider({
  children,
  userPromise
}: {
  children: ReactNode;
  userPromise: Promise<User | null>;
}) {
  return (
    <UserContext.Provider value={{ userPromise }}
      {children}
    </UserContext.Provider>
  );
}
```

Finally, you can call the `useUser()` custom hook in any Client Component and unwrap the Promise:

```
TS app/profile.tsx TypeScript ▾ ⌂
```

```
'use client'

import { use } from 'react'
import { useUser } from './user-provider'

export function Profile() {
  const { userPromise } = useUser()
  const user = use(userPromise)

  return '...'
}
```

The component that consumes the Promise (e.g. `Profile` above) will be suspended. This enables partial hydration. You can see the streamed and prerendered HTML before JavaScript has finished loading.

SPAs with SWR

[SWR ↗](#) is a popular React library for data fetching.

With SWR 2.3.0 (and React 19+), you can gradually adopt server features alongside your existing SWR-based client data fetching code. This is an abstraction of the above `use()` pattern. This means you can move data fetching between the client and server-side, or use both:

- **Client-only:** `useSWR(key, fetcher)`
- **Server-only:** `useSWR(key)` + RSC-provided data
- **Mixed:** `useSWR(key, fetcher)` + RSC-provided data

For example, wrap your application with

`<SWRConfig>` and a `fallback`:

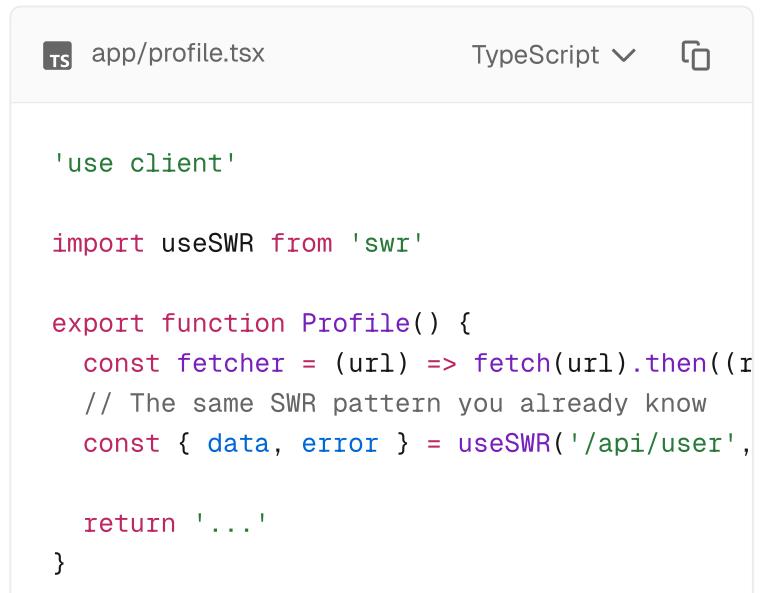


```
TS app/layout.tsx TypeScript ▾ ⌂

import { SWRConfig } from 'swr'
import { getUser } from './user' // some serv
```

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
) {  
  return (  
    <SWRConfig  
      value={ {  
        fallback: {  
          // We do NOT await getUser() here  
          // Only components that read this do  
          '/api/user': getUser(),  
        },  
      }  
    }  
    >  
    {children}  
  </SWRConfig>  
)  
}
```

Because this is a Server Component, `getUser()` can securely read cookies, headers, or talk to your database. No separate API route is needed. Client components below the `<SWRConfig>` can call `useSWR()` with the same key to retrieve the user data. The component code with `useSWR` **does not require any changes** from your existing client-fetching solution.



```
'use client'  
  
import useSWR from 'swr'  
  
export function Profile() {  
  const fetcher = (url) => fetch(url).then((r)  
    // The same SWR pattern you already know  
    const { data, error } = useSWR('/api/user',  
  
    return '...'  
}
```

The `fallback` data can be prerendered and included in the initial HTML response, then immediately read in the child components using

`useSWR`. SWR's polling, revalidation, and caching still run **client-side only**, so it preserves all the interactivity you rely on for an SPA.

Since the initial `fallback` data is automatically handled by Next.js, you can now delete any conditional logic previously needed to check if `data` was `undefined`. When the data is loading, the closest `<Suspense>` boundary will be suspended.

	SWR	RSC	RSC + SWR
SSR data	✗	✓	✓
Streaming while SSR	✗	✓	✓
Deduplicate requests	✓	✓	✓
Client-side features	✓	✗	✓

SPAs with React Query

You can use React Query with Next.js on both the client and server. This enables you to build both strict SPAs, as well as take advantage of server features in Next.js paired with React Query.

Learn more in the [React Query documentation ↗](#).

Rendering components only in the browser

Client components are [prerendered ↗](#) during `next build`. If you want to disable prerendering for a Client Component and only load it in the browser environment, you can use `next/dynamic`:

```
import dynamic from 'next/dynamic'  
  
const ClientOnlyComponent = dynamic(() => imp
```

```
    ssr: false,  
  })
```

This can be useful for third-party libraries that rely on browser APIs like `window` or `document`. You can also add a `useEffect` that checks for the existence of these APIs, and if they do not exist, return `null` or a loading state which would be prerendered.

Shallow routing on the client

If you are migrating from a strict SPA like [Create React App](#) or [Vite](#), you might have existing code which shallow routes to update the URL state. This can be useful for manual transitions between views in your application *without* using the default Next.js file-system routing.

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`.

```
'use client'

import { useSearchParams } from 'next/navigation'

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

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

  return (
    <>
```

```
<button onClick={() => updateSorting('a')}>a</button>
<button onClick={() => updateSorting('d')}>d</button>
)
}
```

Learn more about how [routing and navigation](#) work in Next.js.

Using Server Actions in Client Components

You can progressively adopt Server Actions while still using Client Components. This allows you to remove boilerplate code to call an API route, and instead use React features like `useActionState` to handle loading and error states.

For example, create your first Server Action:



```
TS app/actions.ts TypeScript ▾ ⌂
'use server'

export async function create() {}
```

You can import and use a Server Action from the client, similar to calling a JavaScript function. You do not need to create an API endpoint manually:



```
TS app/button.tsx TypeScript ▾ ⌂
'use client'

import { create } from './actions'

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

Learn more about [mutating data with Server Actions](#).

Static export (optional)

Next.js also supports generating a fully [static site](#).

This has some advantages over strict SPAs:

- **Automatic code-splitting:** Instead of shipping a single `index.html`, Next.js will generate an HTML file per route, so your visitors get the content faster without waiting for the client JavaScript bundle.
- **Improved user experience:** Instead of a minimal skeleton for all routes, you get fully rendered pages for each route. When users navigate client side, transitions remain instant and SPA-like.

To enable a static export, update your configuration:

```
TS next.config.ts

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  output: 'export',
}

export default nextConfig
```

After running `next build`, Next.js will create an `out` folder with the HTML/CSS/JS assets for your application.

Note: Next.js server features are not supported with static exports. [Learn more](#).

Migrating existing projects to Next.js

You can incrementally migrate to Next.js by following our guides:

- [Migrating from Create React App](#)
- [Migrating from Vite](#)

If you are already using a SPA with the Pages Router, you can learn how to [incrementally adopt the App Router](#).

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



How to create a static export of your Next.js application

Next.js enables starting as a static site or Single-Page Application (SPA), then later optionally upgrading to use features that require a server.

When running `next build`, Next.js generates an HTML file per route. By breaking a strict SPA into individual HTML files, Next.js can avoid loading unnecessary JavaScript code on the client-side, reducing the bundle size and enabling faster page loads.

Since Next.js supports this static export, it can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets.

Configuration

To enable a static export, change the output mode inside `next.config.js`:

```
JS next.config.js   
  
/**  
 * @type {import('next').NextConfig}  
 */  
const nextConfig = {  
  output: 'export',  
  
  // Optional: Change links `/me` -> `/me/` a
```

```
// trailingSlash: true,  
  
// Optional: Prevent automatic `/me` -> `/m  
// skipTrailingSlashRedirect: true,  
  
// Optional: Change the output directory `o  
// distDir: 'dist',  
}  
  
module.exports = nextConfig
```

After running `next build`, Next.js will create an `out` folder with the HTML/CSS/JS assets for your application.

Supported Features

The core of Next.js has been designed to support static exports.

Server Components

When you run `next build` to generate a static export, Server Components consumed inside the `app` directory will run during the build, similar to traditional static-site generation.

The resulting component will be rendered into static HTML for the initial page load and a static payload for client navigation between routes. No changes are required for your Server Components when using the static export, unless they consume [dynamic server functions](#).

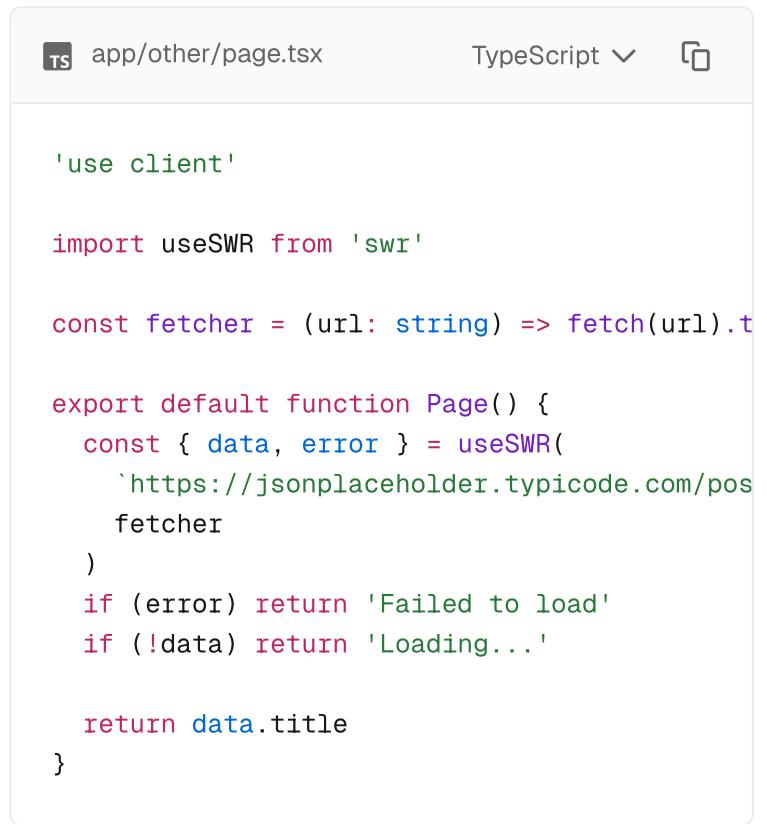


```
TS app/page.tsx TypeScript ▾ ⌂  
  
export default async function Page() {  
  // This fetch will run on the server during  
  const res = await fetch('https://api.example.com/data')  
  const data = await res.json()  
  return data  
}
```

```
    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.



```
TS app/other/page.tsx TypeScript ▾
```

```
'use client'

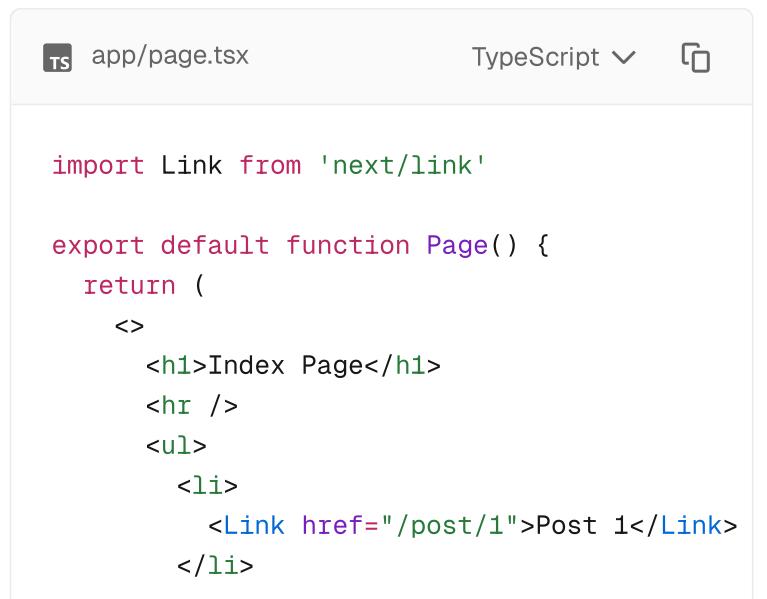
import useSWR from 'swr'

const fetcher = (url: string) => fetch(url).t

export default function Page() {
  const { data, error } = useSWR(
    'https://jsonplaceholder.typicode.com/pos
    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:



```
TS app/page.tsx TypeScript ▾
```

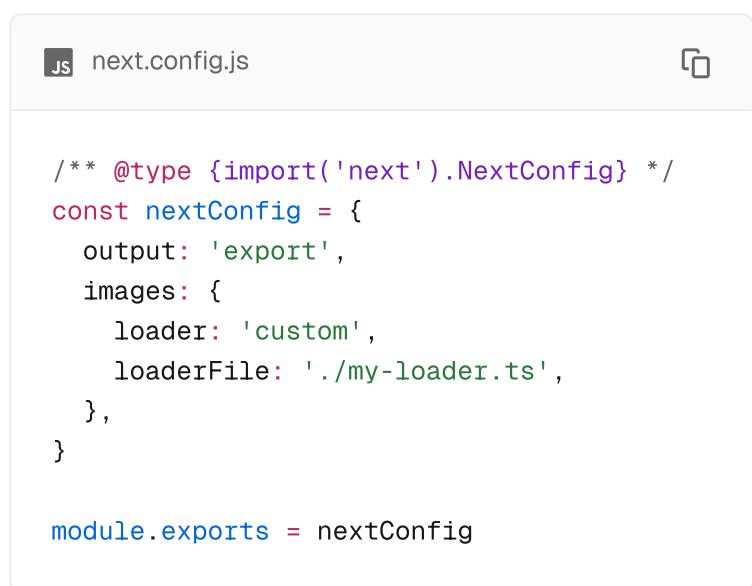
```
import Link from 'next/link'

export default function Page() {
  return (
    <>
      <h1>Index Page</h1>
      <hr />
      <ul>
        <li>
          <Link href="/post/1">Post 1</Link>
        </li>
      </ul>
    </>
  )
}
```

```
<li>
  <Link href="/post/2">Post 2</Link>
</li>
</ul>
</>
)
}
```

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:

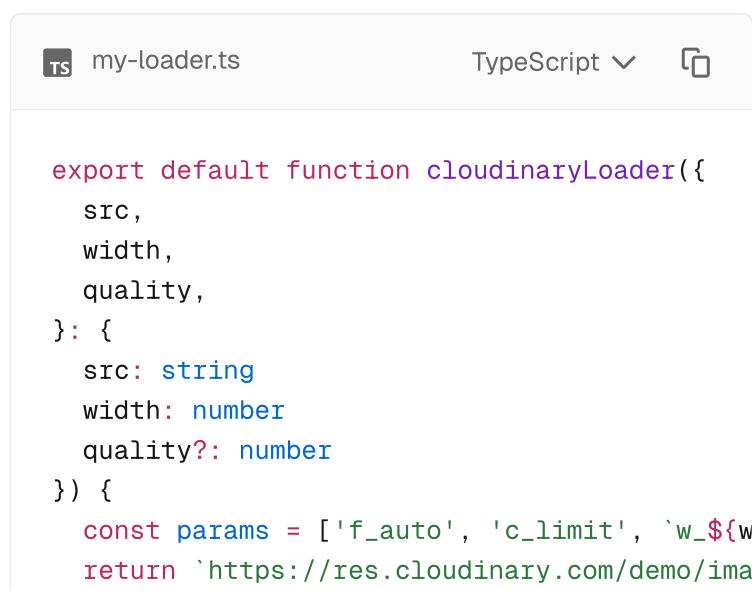


A screenshot of a code editor showing the file `next.config.js`. The code defines a custom image loader using the Cloudinary service. It sets the output to 'export' and uses a custom loader file named `my-loader.ts`.

```
/* @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:



A screenshot of a code editor showing the file `my-loader.ts` written in TypeScript. It defines a function `cloudinaryLoader` that takes `src`, `width`, and `quality` as parameters. It then constructs a URL using these parameters along with other Cloudinary-specific parameters like `f_auto` and `c_limit`.

```
export default function cloudinaryLoader({
  src,
  width,
  quality,
}: {
  src: string
  width: number
  quality?: number
}) {
  const params = ['f_auto', 'c_limit', `w_${width}`]
  return `https://res.cloudinary.com/demo/ima...
```

```
' ,  
) } ${src} `  
}
```

You can then use `next/image` in your application, defining relative paths to the image in Cloudinary:

```
TS app/page.tsx TypeScript ▾ 
```

```
import Image from 'next/image'  
  
export default function Page() {  
  return <Image alt="turtles" src="/turtles.j  
}
```

Route Handlers

Route Handlers will render a static response when running `next build`. Only the `GET` HTTP verb is supported. This can be used to generate static HTML, JSON, TXT, or other files from cached or uncached data. For example:

```
TS app/data.json/route.ts TypeScript ▾ 
```

```
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
- Intercepting Routes

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'
```

Deploying

With a static export, Next.js can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets.

When running `next build`, Next.js generates the static export into the `out` folder. For example, let's say you have the following routes:

- `/`
- `/blog/[id]`

After running `next build`, Next.js will generate the following files:

- `/out/index.html`
- `/out/404.html`
- `/out/blog/post-1.html`
- `/out/blog/post-2.html`

If you are using a static host like Nginx, you can configure rewrites from incoming requests to the correct files:

 nginx.conf



```
server {
  listen 80;
  server_name acme.com;

  root /var/www/out;

  location / {
    try_files $uri $uri.html $uri/ =404;
  }

  # This is necessary when `trailingSlash: fa
  # You can omit this when `trailingSlash: tr
  location /blog/ {
    rewrite ^/blog/(.*)$ /blog/$1.html break;
  }

  error_page 404 /404.html;
  location = /404.html {
    internal;
  }
}
```

Version History

Version	Changes
v14.0.0	<code>next export</code> has been removed in favor of <code>"output": "export"</code>
v13.4.0	App Router (Stable) adds enhanced static export support, including using React Server Components and Route Handlers.
v13.3.0	<code>next export</code> is deprecated and replaced with <code>"output": "export"</code>

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4



How to install Tailwind CSS v3 in your Next.js application

This guide will walk you through how to install [Tailwind CSS v3](#) in your Next.js application.

Good to know: For the latest Tailwind 4 setup, see the [Tailwind CSS setup instructions](#).

Installing Tailwind v3

Install Tailwind CSS and its peer dependencies, then run the `init` command to generate both `tailwind.config.js` and `postcss.config.js` files:

```
pnpm npm yarn bun
>_ Terminal
pnpm add -D tailwindcss@^3 postcss autoprefixer
npx tailwindcss init -p
```

Configuring Tailwind v3

Configure your template paths in your
tailwind.config.js file:

tailwind.config.js



```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    './app/**/*.{js,ts,jsx,tsx,mdx}',
    './pages/**/*.{js,ts,jsx,tsx,mdx}',
    './components/**/*.{js,ts,jsx,tsx,mdx}',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Add the Tailwind directives to your global CSS file:

app/globals.css



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

Import the CSS file in your root layout:

app/layout.tsx

TypeScript ▾



```
import './globals.css'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  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.



app/page.tsx TypeScript ▾

```
export default function Page() {
  return <h1 className="text-3xl font-bold un
```

Usage with Turbopack

As of Next.js 13.1, Tailwind CSS and PostCSS are supported with [Turbopack ↗](#).

Was this helpful?    



Using App Router
Features available in /app



Latest Version
15.5.4



Testing



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:

Cypress

Learn how to set up Cypress with Next.js for End-to...

Jest

Learn how to set up Jest with Next.js for Unit...

Playwright

Vitest

Learn how to set
up Playwright with
Next.js for End-to...

Learn how to set
up Vitest with
Next.js for Unit...

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

How to set up Cypress with Next.js

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:

- 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.

>_ Terminal



```
npx create-next-app@latest --example with-cyp
```

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": "eslint",
    "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` file has the following configuration:

TS

cypress.config.ts

TypeScript



```
import { defineConfig } from 'cypress'

export default defineConfig({
  e2e: {
    setupNodeEvents(on, config) {},
  },
})
```

Then, create two new Next.js files:

JS

app/page.js



```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

JS

app/about/page.js



```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>About</h1>
      <Link href="/">Home</Link>
    </div>
  )
}
```

Add a test to check your navigation is working correctly:

JS

cypress/e2e/app.cy.js

```
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"
    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` file has the following configuration:

```
TS cypress.config.ts TypeScript ▾ ⌂

import { defineConfig } from 'cypress'

export default 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:

```
TS cypress/component/about.cy.tsx ⌂

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  
cy.get('h1').contains('Home')
```

```
// Validate that a link with the expected  
// Following the link is better suited to
```

```
cy.get('a[href="/about"]').should('be.vis')
```

```
)  
})
```

Good to know:

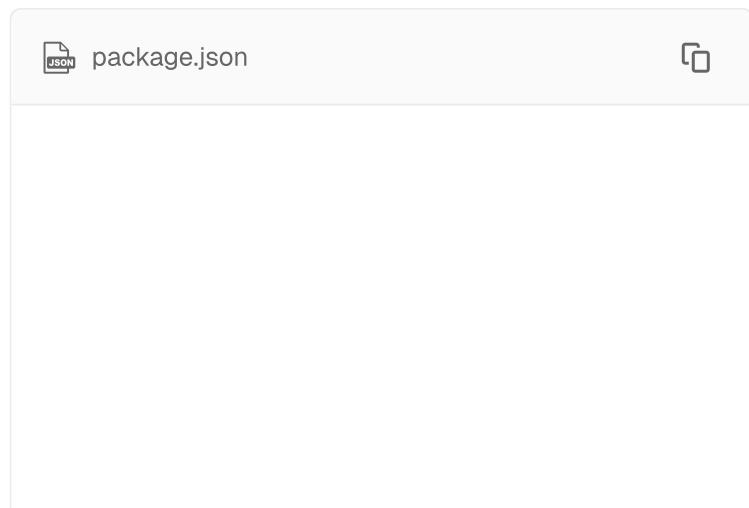
- Cypress currently doesn't support Component Testing for `async` Server Components. We recommend using E2E testing.
- Since component tests do not require a Next.js server, features like `<Image />` that rely on a server being available may not function out-of-the-box.

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://  
    "e2e:headless": "start-server-and-test de  
    "component": "cypress open --component",  
    "component:headless": "cypress run --comp  
  }  
}
```

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 ↗](#)

Was this helpful?



 Using App Router
Features available in /app Latest Version
15.5.4

How to set up Jest with Next.js

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:

>_ Terminal



```
npx create-next-app@latest --example with-jes
```

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:

> Terminal

```
npm install -D jest jest-environment-jsdom @t
# or
yarn add -D jest jest-environment-jsdom @test
# or
pnpm install -D jest jest-environment-jsdom @
```

Generate a basic Jest configuration file by running the following command:

> Terminal

```
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 jest.config.ts

TypeScript ▾

```
import type { Config } from 'jest'
import nextJest from 'next/jest'

const createJestConfig = nextJest({
  // Provide the path to your Next.js app to
  dir: './',
})

// Add any custom config to be passed to Jest
const config: Config = {
  coverageProvider: 'v8',
  testEnvironment: 'jsdom',
```

```
// Add more setup options before each test
// setupFilesAfterEnv: ['<rootDir>/jest.set
}

// createJestConfig is exported this way to e
export default 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](#).

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>/component'
}
```

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'
```

Then, inside `jest.setup`, add the following import:



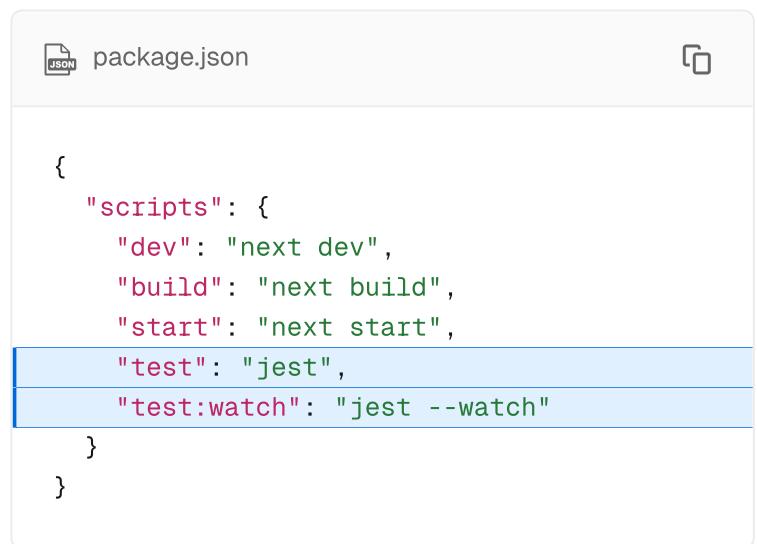
```
import '@testing-library/jest-dom'
```

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:



```
JSON package.json
```

```
{
  "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 `<Page />` component successfully renders a heading:

JS app/page.js

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

JS __tests__/page.test.jsx

```
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')

    expect(heading).toBeInTheDocument()
  })
})
```

Optionally, add a [snapshot test ↗](#) to keep track of any unexpected changes in your component:

JS __tests__/snapshot.js

```
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:

> Terminal



```
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.

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4



How to set up Playwright with Next.js

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.

```
>_ Terminal   
  
npx create-next-app@latest --example with-pla
```

Manual setup

To install Playwright, run the following command:

```
>_ Terminal   
  
npx create-next-app@latest --example with-pla
```

```
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:

app/page.tsx

```
import Link from 'next/link'

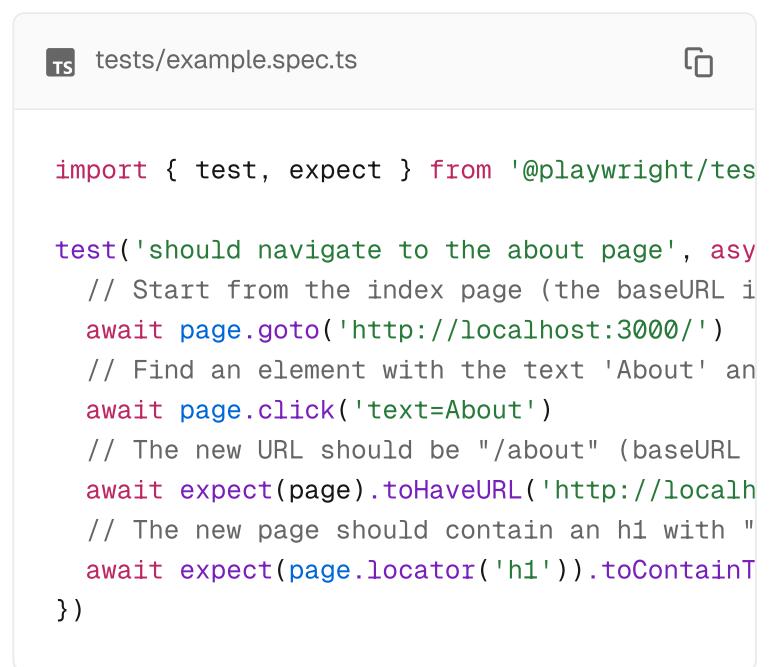
export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

app/about/page.tsx

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>About</h1>
      <Link href="/">Home</Link>
    </div>
  )
}
```

Then, add a test to verify that your navigation is working correctly:



```
TS tests/example.spec.ts
```

```
import { test, expect } from '@playwright/test'

test('should navigate to the about page', async () => {
  // Start from the index page (the baseURL is optional)
  await page.goto('http://localhost:3000/')
  // Find an element with the text 'About' and click it
  await page.click('text=About')
  // The new URL should be "/about" (baseURL is optional)
  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 ↗](#)

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

How to set up Vitest with Next.js

Vitest 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 **E2E tests** for `async` components.

Quickstart

You can use `create-next-app` with the Next.js [with-vitest ↗](#) example to quickly get started:

```
>_ Terminal   
  
npx create-next-app@latest --example with-vit
```

Manual Setup

To manually set up Vitest, install `vitest` and the following packages as dev dependencies:

```
>_ Terminal
```

```
# Using TypeScript  
npm install -D vitest @vitejs/plugin-react js  
# Using JavaScript  
npm install -D vitest @vitejs/plugin-react js
```

Create a `vitest.config.mts|js` file in the root of your project, and add the following options:

```
vitest.config.mts TypeScript ▾
```

```
import { defineConfig } from 'vitest/config'  
import react from '@vitejs/plugin-react'  
import tsconfigPaths from 'vite-tsconfig-path'  
  
export default defineConfig({  
  plugins: [tsconfigPaths(), 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`:

```
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:

```
TS app/page.tsx TypeScript ▾
```

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
TS __tests__/page.test.tsx TypeScript ▾
```

```
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 })).toBeInTheDocument()
})
```

Good to know: The example above uses the common `__tests__` convention, but test files can also be colocated inside the `app` router.

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 ↗](#)

Was this helpful?

Using App Router
Features available in /app

Latest Version
15.5.4



How to optimize third-party libraries

`@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:

```
>_ Terminal   
npm install @next/third-parties@latest next@1
```

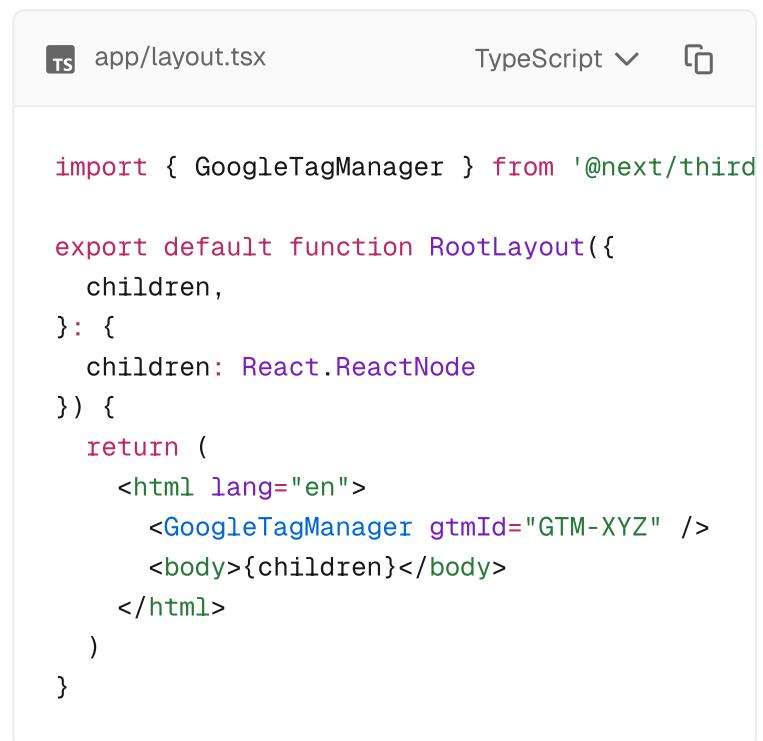
`@next/third-parties` is currently an **experimental** library under active development. We recommend installing it with the **latest** or **canary** flags while we work on adding more third-party integrations.

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:



The screenshot shows a code editor window with the following details:

- File name: `app/layout.tsx`
- Language: TypeScript (indicated by the `TS` icon)
- Version: TypeScript (dropdown menu)
- Copy button: A standard copy icon in the top right corner.

The code itself is a React component named `RootLayout` that wraps its children in an `html` element with `lang="en"`, an `GoogleTagManager` component with `gtmId="GTM-XYZ"`, and a `body` element containing the children.

```
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>
  )
}
```

To load Google Tag Manager for a single route, include the component in your page file:



The screenshot shows a code editor window with the following details:

- File name: `app/page.js`
- Language: JavaScript (indicated by the `JS` icon)
- Copy button: A standard copy icon in the top right corner.

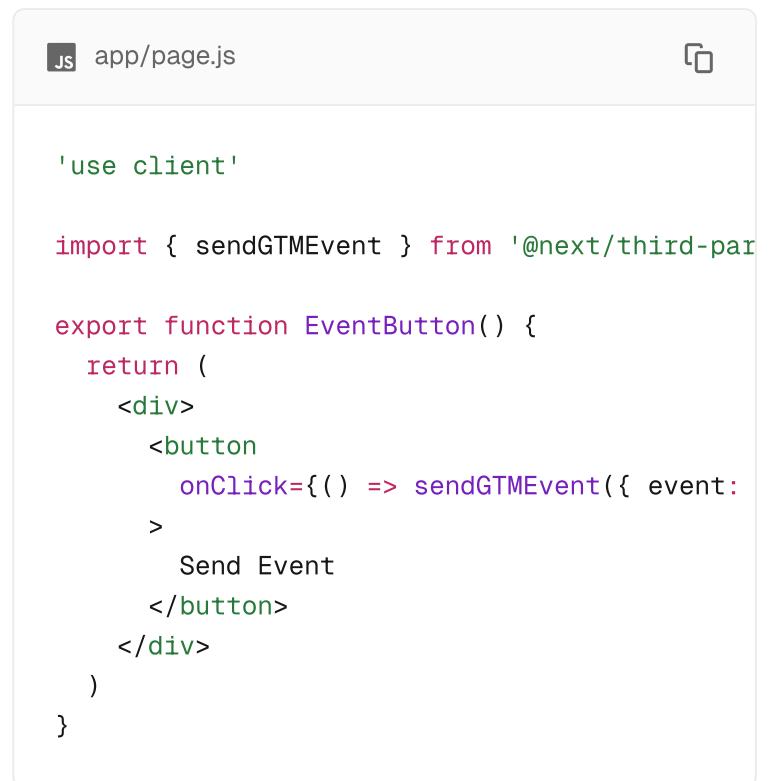
The code defines a `Page` component that returns a `GoogleTagManager` component with `gtmId="GTM-XYZ"`.

```
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.



```
JS app/page.js

'use client'

import { sendGTMEvent } from '@next/third-par

export function EventButton() {
  return (
    <div>
      <button
        onClick={() => sendGTMEvent({ event:
      >
        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 in the format GTM- .
gtmScriptUrl	Optional	GTM script URL. Defaults to https://www.googletagmanager.com
dataLayer	Optional	Data layer object to instantiate.
dataLayerName	Optional	Name of the data layer. Defaults to <code>dataLayer</code> .
auth	Optional	Value of authentication parameter for environment snippets.
preview	Optional	Value of preview parameter 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:

```
TS app/layout.tsx TypeScript ▾ ⌂
import { GoogleAnalytics } from '@next/third-
```

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

To load Google Analytics for a single route, include the component in your page file:

JS app/page.js

```
import { GoogleAnalytics } from '@next/third-party'

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.

JS app/page.js

```
'use client'

import { sendGAEvent } from '@next/third-party'

export function EventButton() {
  return (
    <div>
      <button
        onClick={() => sendGAEvent('event', 'button-clicked')}
      >
```

```
        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.

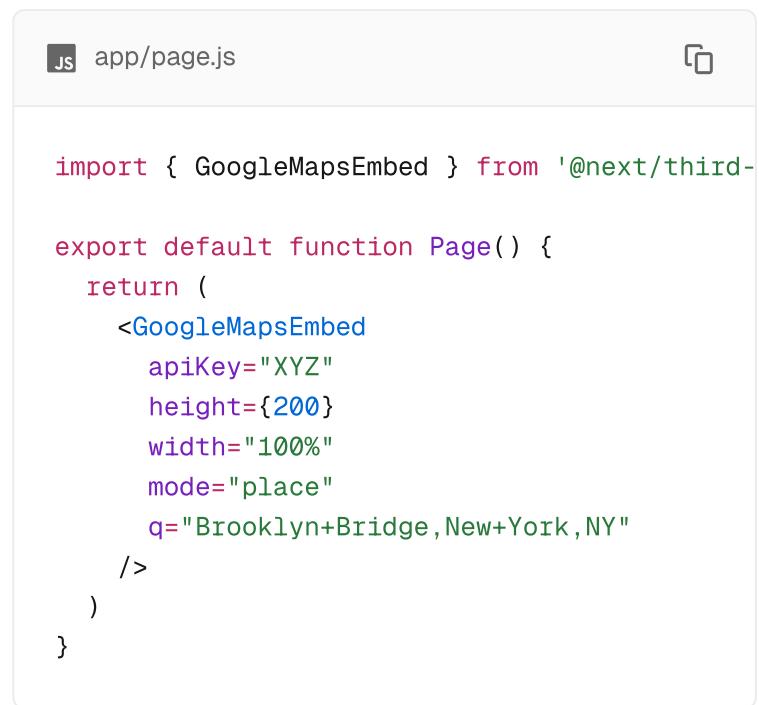
Options

Options to pass to the `<GoogleAnalytics>` component.

Name	Type	Description
<code>gaId</code>	Required	Your measurement ID ↗. Usually starts with G-.
<code>dataLayerName</code>	Optional	Name of the data layer. Defaults to <code>dataLayer</code> .
<code>nonce</code>	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.



```
JS app/page.js

import { GoogleMapsEmbed } from '@next/third-



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
<code>apiKey</code>	Required	Your api key.
<code>mode</code>	Required	Map mode ↗
<code>height</code>	Optional	Height of the embed. Defaults to <code>auto</code> .
<code>width</code>	Optional	Width of the embed. Defaults to <code>auto</code> .
<code>style</code>	Optional	Pass styles to the iframe.
<code>allowfullscreen</code>	Optional	Property to allow certain map parts to go full screen.

Name	Type	Description
loading	Optional	Defaults to lazy. Consider changing if you know your embed will be above the fold.
q	Optional	Defines map marker location. <i>This may be required depending on the map mode.</i>
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 geopolitical 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.

```
JS app/page.js
```

```
import { YouTubeEmbed } from '@next/third-par

export default function Page() {
  return <YouTubeEmbed videoid="ogfYd705cRs"
}
```

Options

Name	Type	Description
videoid	Required	YouTube video id.
width	Optional	Width of the video container. Def auto
height	Optional	Height of the video container. De auto
playlabel	Optional	A visually hidden label for the pla for accessibility.
params	Optional	The video player params defined Params are passed as a query pa string. Eg: params="controls=0&start=108
style	Optional	Used to apply styles to the video container.

Was this helpful?    



Using App Router
Features available in /app



Upgrade Guides



Latest Version
15.5.4



Learn how to upgrade to the latest versions of Next.js following the versions-specific guides:

Codemods

Use codemods to upgrade your Next.js codebase...

Version 14

Upgrade your Next.js Application from Version 13 t...

Version 15

Upgrade your Next.js Application from Version 14 t...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4



Codemods



Codemods are transformations that run on your codebase programmatically. This allows a large number of changes to be programmatically applied without having to manually go through every file.

Next.js provides Codemod transformations to help upgrade your Next.js codebase when an API is updated or deprecated.

Usage

In your terminal, navigate (`cd`) into your project's folder, then run:

```
>_ Terminal   
  
npx @next/codemod <transform> <path>
```

Replacing `<transform>` and `<path>` with appropriate values.

- `transform` - name of transform
- `path` - files or directory to transform
- `--dry` Do a dry-run, no code will be edited
- `--print` Prints the changed output for comparison

Codemods

16.0

Migrate from `next lint` to ESLint CLI

`next-lint-to-eslint-cli`

```
>_ Terminal ┌─────────┐  
          └─────────┘  
  
npx @next/codemod@canary next-lint-to-eslint-
```

This codemod migrates projects from using `next lint` to using the ESLint CLI with your local ESLint config. It:

- Creates an `eslint.config.mjs` file with Next.js recommended configurations
- Updates `package.json` scripts to use `eslint .` instead of `next lint`
- Adds necessary ESLint dependencies to `package.json`
- Preserves existing ESLint configurations when found

For example:

```
JSON package.json ┌─────────┐  
          └─────────┘  
  
{  
  "scripts": {  
    "lint": "next lint"  
  }  
}
```

Becomes:

```
JSON package.json ┌─────────┐  
          └─────────┘  
  
{  
  "scripts": {  
    "lint": "eslint ."  
  }  
}
```

```
{  
  "scripts": {  
    "lint": "eslint ."  
  }  
}
```

And creates:

```
eslint.config.mjs
```

```
import { dirname } from 'path'  
import { fileURLToPath } from 'url'  
import { FlatCompat } from '@eslint/eslintrc'  
  
const __filename = fileURLToPath(import.meta.url)  
const __dirname = dirname(__filename)  
  
const compat = new FlatCompat({  
  baseDirectory: __dirname,  
})  
  
const eslintConfig = [  
  ...compat.extends('next/core-web-vitals', {  
    ignores: [  
      'node_modules/**',  
      '.next/**',  
      'out/**',  
      'build/**',  
      'next-env.d.ts',  
    ],  
  }),  
]  
  
export default eslintConfig
```

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.

>_ Terminal



```
npx @next/codemod@latest app-dir-runtime-conf
```

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`

>_ Terminal



```
npx @next/codemod@latest next-async-request-a
```

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/header'
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,
  type UnsafeUnwrappedHeaders,
} from 'next/headers'
const token = (cookies() as unknown as UnsafeUnwrappedCookies).get('token')

function useToken() {
  const token = use(cookies()).get('token')
  return token
}

export default async function Page() {
  const name = (await cookies()).get('name')
}

function getHeader() {
  return (headers() as unknown as UnsafeUnwrappedHeaders).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 | str
}) {
  const { value } = searchParams
  if (value === 'foo') {
    // ...
  }
}

export function generateMetadata({ params }: {
  const { slug } = params
  return {
    title: `My Page - ${slug}`,
  }
})
```

Transforms into:

```
// page.tsx
export default async function Page(props: {
  params: Promise<{ slug: string }>
  searchParams: Promise<{ [key: string]: str
}) {
  const searchParams = await props.searchPara
  const { value } = searchParams
  if (value === 'foo') {
    // ...
  }
}

export async function generateMetadata(props: {
  params: Promise<{ slug: string }>
}) {
  const params = await props.params
  const { slug } = 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

>_ Terminal

2

```
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 '@verc

export function GET(req: NextRequest) {
  const geo = geolocation(req)
  const ip = ipAddress(req)
}

}
```

14.0

Migrate `ImageResponse` imports

`next-og-import`

>_ Terminal



```
npx @next/codemod@latest next-og-import .
```

This codemod moves transforms imports from `next/server` to `next/og` for usage of [Dynamic OG Image Generation](#).

For example:

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

Transforms into:

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

Use `viewport` export

`metadata-to-viewport-export`

>_ Terminal



```
npx @next/codemod@latest metadata-to-viewport
```

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

>_ Terminal



```
npx @next/codemod@latest built-in-next-font .
```

This codemod uninstalls the `@next/font` package and transforms `@next/font` imports into the built-in `next/font`.

For example:

```
import { Inter } from '@next/font/google'
```

Transforms into:

```
import { Inter } from 'next/font/google'
```

13.0

Rename Next Image Imports

next-image-to-legacy-image



```
npx @next/codemod@latest next-image-to-legacy
```

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" hei
      <Image2 src="/test.png" width="500" hei
    </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" hei
      <Image2 src="/test.png" width="500" hei
    </div>
  )
}
```

[Migrate to the New Image Component](#)

next-image-experimental

>_ Terminal



```
npx @next/codemod@latest next-image-experimen
```

Dangerously migrates from `next/legacy/image` to the new `next/image` by adding inline styles and removing unused props.

- Removes `layout` prop and adds `style`.
- Removes `objectFit` prop and adds `style`.
- Removes `objectPosition` prop and adds `style`.
- Removes `lazyBoundary` prop.
- Removes `lazyRoot` prop.

Remove `<a>` Tags From Link Components

new-link

>_ Terminal



```
npx @next/codemod@latest new-link .
```

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')}>A
</Link>
```

```
// transforms into
<Link href="/about" onClick={() => console.lo
    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`

>_ Terminal



```
npx @next/codemod cra-to-next
```

Migrates a Create React App project to Next.js; creating a Pages Router and necessary config to match behavior. Client-side only rendering is leveraged initially to prevent breaking compatibility due to `window` usage during SSR and can be enabled seamlessly to allow the gradual adoption of Next.js specific features.

Please share any feedback related to this transform [in this discussion ↗](#).

10

Add React imports

add-missing-react-import

>_ Terminal



```
npx @next/codemod add-missing-react-import
```

Transforms files that do not import `React` to include the import in order for the new [React JSX transform ↗](#) to work.

For example:

js my-component.js



```
export default class Home extends React.Component {
  render() {
    return <div>Hello World</div>
  }
}
```

Transforms into:

js my-component.js



```
import React from 'react'
export default class Home extends React.Component {
  render() {
    return <div>Hello World</div>
  }
}
```

9

Transform Anonymous Components into Named Components

name-default-component

>_ Terminal



```
npx @next/codemod name-default-component
```

Versions 9 and above.

Transforms anonymous components into named components to make sure they work with [Fast Refresh ↗](#).

For example:

```
JS my-component.js ✖  
  
export default function () {  
  return <div>Hello World</div>  
}
```

Transforms into:

```
JS my-component.js ✖  
  
export default function MyComponent() {  
  return <div>Hello World</div>  
}
```

The component will have a camel-cased name based on the name of the file, and it also works with arrow functions.

8

Transform AMP HOC into page config

[withamp-to-config](#)

```
>_ Terminal ✖  
  
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`

>_ Terminal



```
npx @next/codemod url-to-withrouter
```

Transforms the deprecated automatically injected `url` property on top level pages to using `withRouter` and the `router` property it injects.
Read more here:

<https://nextjs.org/docs/messages/url-deprecated>

For example:

From



```
import React from 'react'
export default class extends React.Component {
  render() {
```

```
const { pathname } = this.props.url
return <div>Current pathname: {pathname}<
      >
    </div>
```

To



```
import React from 'react'
import { withRouter } from 'next/router'
export default withRouter(
  class extends React.Component {
    render() {
      const { pathname } = this.props.router
      return <div>Current pathname: {pathname}
    }
  }
)
```

This is one case. All the cases that are transformed (and tested) can be found in the

[__testfixtures__ directory ↗](#).

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4



How to upgrade to version 14

Upgrading from 13 to 14

To update to Next.js version 14, run the following command using your preferred package manager:

>_ Terminal



```
npm i next@next-14 react@18 react-dom@18 && n
```

>_ Terminal



```
yarn add next@next-14 react@18 react-dom@18 &
```

>_ Terminal



```
pnpm i next@next-14 react@18 react-dom@18 &&
```

>_ Terminal



```
bun add next@next-14 react@18 react-dom@18 &&
```

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.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



How to upgrade to version 15

Upgrading from 14 to 15

To update to Next.js version 15, you can use the `upgrade codemod`:

>_ Terminal



```
npx @next/codemod@canary upgrade latest
```

If you prefer to do it manually, ensure that you're installing the latest Next & React versions:

>_ Terminal



```
npm i next@latest react@latest react-dom@late
```

Good to know:

- If you see a peer dependencies warning, you may need to update `react` and `react-dom` to the suggested versions, or 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.

- 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 ↗](#).

Good to know: If you are using TypeScript, ensure you also upgrade `@types/react` and `@types/react-dom` to their latest versions.

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

app/page.tsx TypeScript ▾

```
import { cookies, type UnsafeUnwrappedCookies }

// Before
const cookieStore = cookies()
const token = cookieStore.get('token')

// After
const cookieStore = cookies() as unknown as U
// 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

app/page.tsx

TypeScript ▾

```
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')
```

draftMode

Recommended Async Usage

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

// Before
const { isEnabled } = draftMode()

// After
const { isEnabled } = await draftMode()
```

Temporary Synchronous Usage

app/page.tsx

TypeScript ▾

```
import { draftMode, type UnsafeUnwrappedDraft } from 'next/headers'

// Before
const { isEnabled } = draftMode()

// After
// will log a warning in dev
const { isEnabled } = draftMode() as unknown
```

params & searchParams

Asynchronous Layout

app/layout.tsx

TypeScript ▾

```

// Before
type Params = { slug: string }

export function generateMetadata({ params }: {
  const { slug } = params
})

export default async function Layout({
  children,
  params,
}: {
  children: React.ReactNode
  params: Params
}) {
  const { slug } = params
}

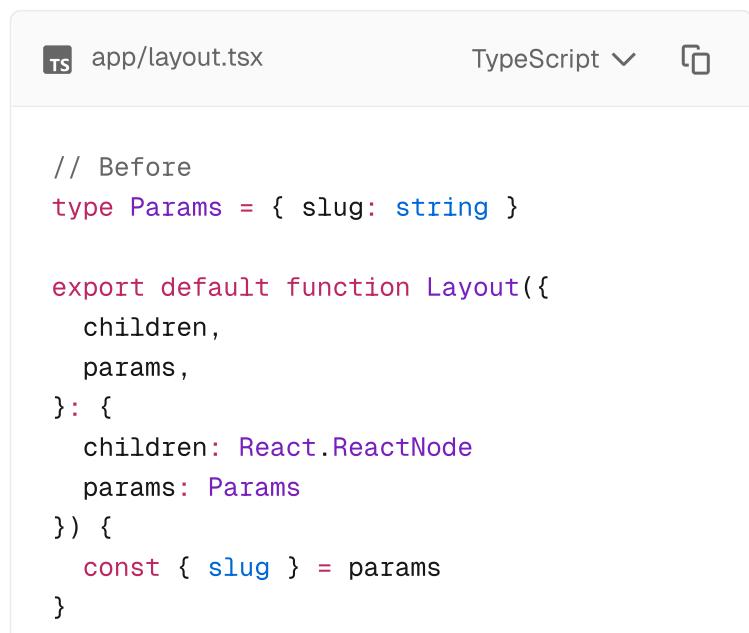
// After
type Params = Promise<{ slug: string }>

export async function generateMetadata({ para
  const { slug } = await params
}

export default async function Layout({
  children,
  params,
}: {
  children: React.ReactNode
  params: Params
}) {
  const { slug } = await params
}

```

Synchronous Layout



The screenshot shows a code editor interface with the following details:

- File:** app/layout.tsx
- TypeScript Version:** TypeScript 4.5 (indicated by the dropdown menu)
- Code Preview:** A preview window on the right shows the rendered layout component.
- Code Content:**

```

// Before
type Params = { slug: string }

export default function Layout({
  children,
  params,
}: {
  children: React.ReactNode
  params: Params
}) {
  const { slug } = params
}

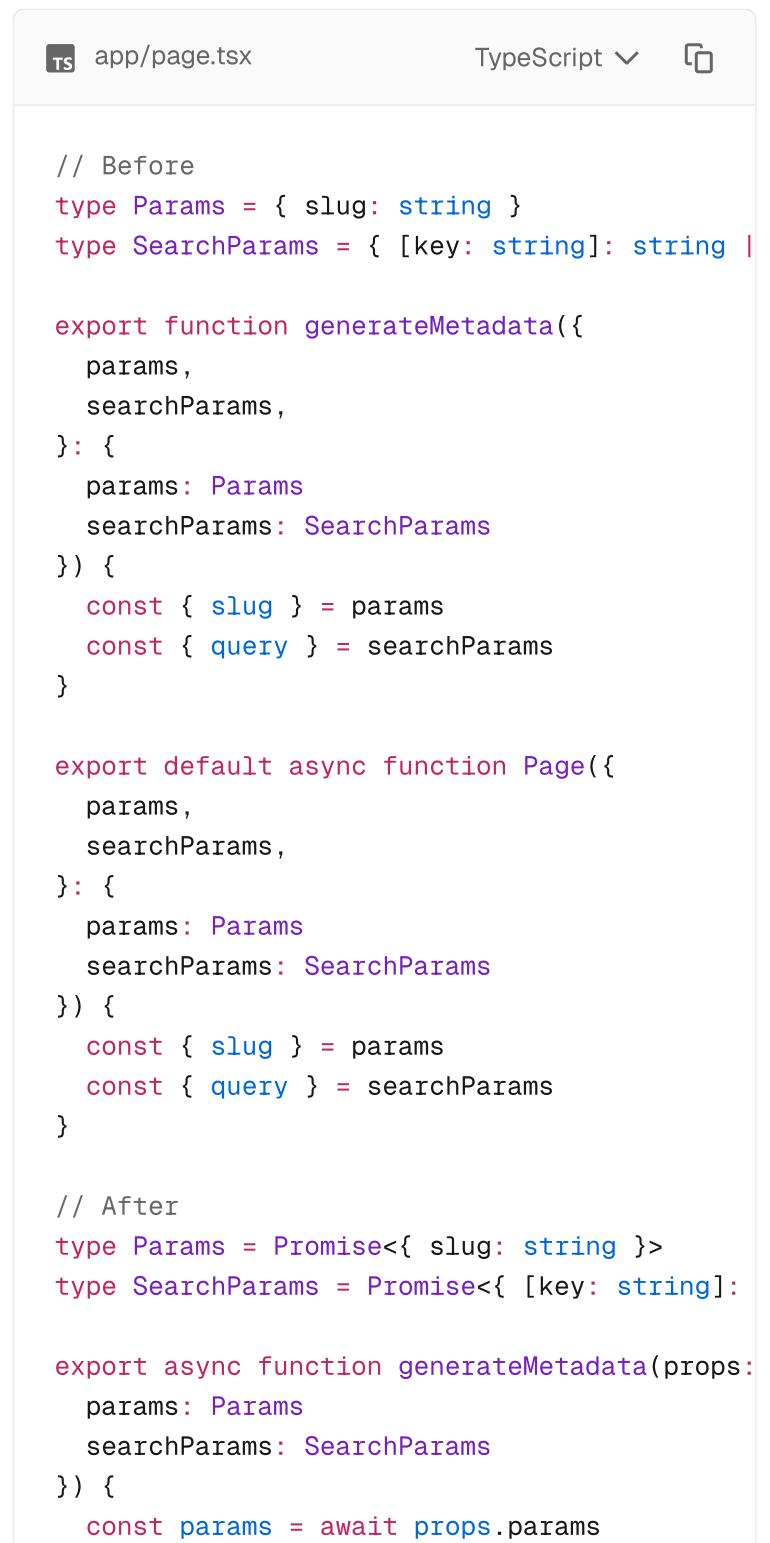
```

```
// 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
}
```

Asynchronous Page



The screenshot shows a code editor interface with a tab labeled "app/page.tsx" and a "TypeScript" dropdown set to "TypeScript". The code is divided into two sections: "Before" and "After".

Before:

```
// Before
type Params = { slug: string }
typeSearchParams = { [key: string]: string }

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
```

After:

```
type Params = Promise<{ slug: string }>
typeSearchParams = Promise<{ [key: string]: string }>

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
}

```

Synchronous Page

```

'use client'

// Before
type Params = { slug: string }
type SearchParams = { [key: string]: string | string[] }

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[] }>

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, search }) {
  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

```
TS app/api/route.ts TypeScript ▾ ⌂

// Before
type Params = { slug: string }

export async function GET(request: Request, s
  const params = segmentData.params
  const slug = params.slug
}

// After
type Params = Promise<{ slug: string }>

export async function GET(request: Request, s
  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.

```
JS app/layout.js

export default async function RootLayout() {
  const a = await fetch('https://...') // Not
  const b = await fetch('https://...', { cach

  // ...
}
```

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.

```
JS app/layout.js

// Since this is the root layout, all fetch r
// that don't set their own cache option will
export const fetchCache = 'default-cache'

export default async function RootLayout() {
  const a = await fetch('https://...') // Cac
  const b = await fetch('https://...', { cach

  // ...
}
```

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.

JS app/api/route.js



```
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:

JS next.config.js



```
/** @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.

JS app/layout.js



```
// Before
import { Inter } from '@next/font/google'

// After
import { Inter } from 'next/font/google'
```

bundlePagesRouterDependencies

`experimental.bundlePagesExternals` is now stable and renamed to `bundlePagesRouterDependencies`.

JS next.config.js



```
/** @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`.

```
JS next.config.js ✖  
  
/** @type {import('next').NextConfig} */  
const nextConfig = {  
  // Before  
  experimental: {  
    serverComponentsExternalPackages: ['packa  
  },  
  
  // 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](#) ↗ instead:

TS middleware.ts



```
import { geolocation } from '@vercel/function
import type { NextRequest } from 'next/server

export function middleware(request: NextRequest) {
  const { city } = geolocation(request)

  // ...
}
```

TS middleware.ts



```
import { ipAddress } from '@vercel/functions'
import type { NextRequest } from 'next/server

export function middleware(request: NextRequest) {
  const ip = ipAddress(request)

  // ...
}
```

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

How to use and optimize videos

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

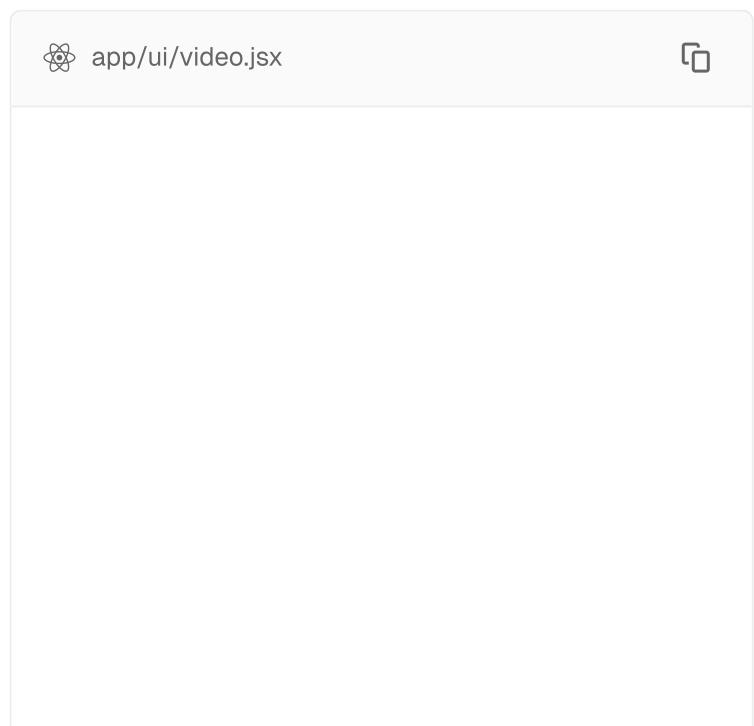
Using `<video>` and `<iframe>`

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.

```
⚙️ app/ui/video.jsx
```



```

export function Video() {
  return (
    <video width="320" height="240" controls
      <source src="/path/to/video.mp4" type=""
      <track
        src="/path/to/captions.vtt"
        kind="subtitles"
        srcLang="en"
        label="English"
      />
      Your browser does not support the video
    </video>
  )
}

```

Common `<video>` tag attributes

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

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

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.



```
app/page.jsx
export default function Page() {
  return (
    <iframe src="https://www.youtube.com/embed/">
  )
}
```

Common `<iframe>` tag attributes

Attribute	Description	Example Value
<code>src</code>	The URL of the page to embed.	<code><iframe src="https://example.com/"></code>

Attribute	Description	Example Value
width	Sets the width of the iframe.	<code><iframe width="500"</code>
height	Sets the height of the iframe.	<code><iframe height="300"</code>
allowFullScreen	Allows the iframe content to be displayed in full-screen mode.	<code><iframe allowFullScreen /></code>
sandbox	Enables an extra set of restrictions on the content within the iframe.	<code><iframe sandbox /></code>
loading	Optimize loading behavior (e.g., lazy loading).	<code><iframe loading="lazy"</code>
title	Provides a title for the iframe to support accessibility.	<code><iframe title="Description"</code>

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.

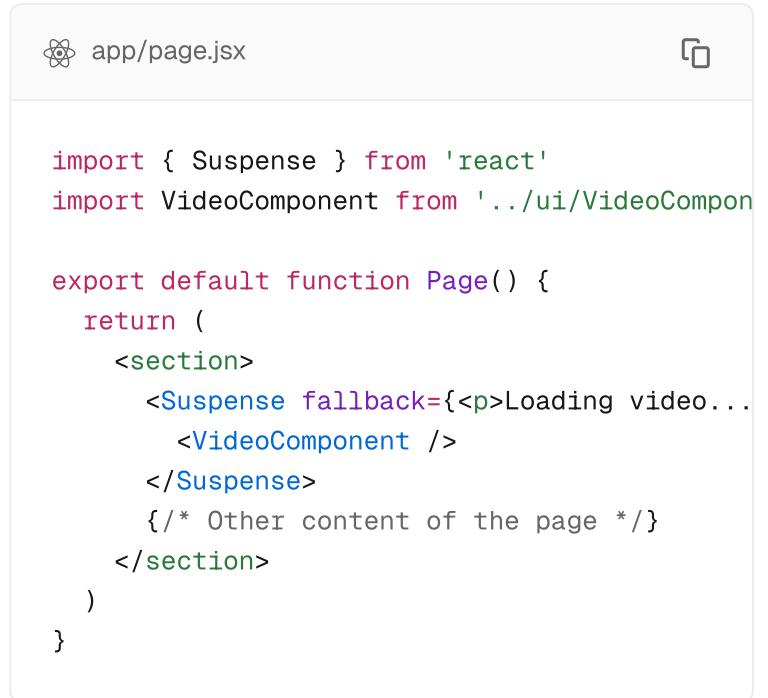
```
app/ui/video-component.jsx
```

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

```
        return <iframe src={src} 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 ↗](#).



The screenshot shows a code editor window with the file name "app/page.jsx" at the top. The code inside the file uses the `Suspense` component from the `'react'` library to handle video loading. It includes a fallback message and a placeholder component (`VideoComponent`) while the video loads.

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

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

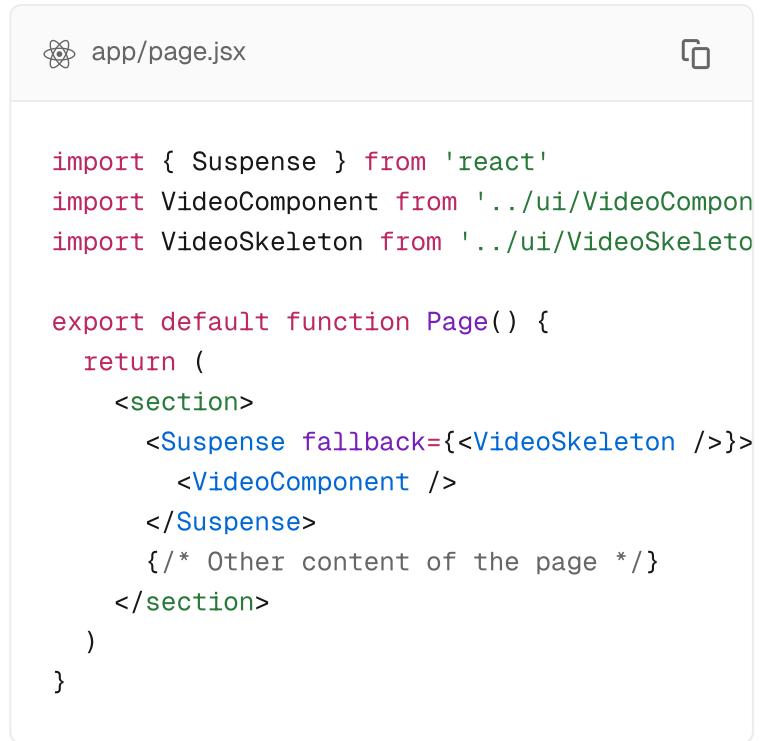
Good to know: When embedding videos from external platforms, consider the following best practices:

- 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:



```
app/page.jsx

import { Suspense } from 'react'
import VideoComponent from '../ui/VideoCompon
import VideoSkeleton from '../ui/VideoSkeleto

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:

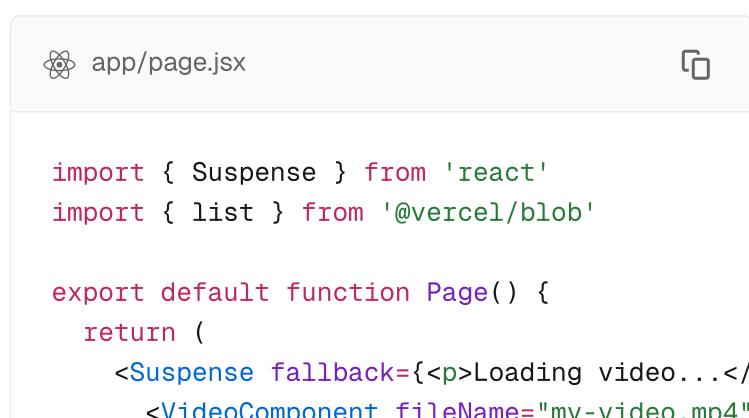
1. Uploading a video to Vercel Blob

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:



```
app/page.jsx

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>
  )
}

function VideoComponent({ fileName }) {
  const blob = await list(fileName)
  const url = blob.url
  return <video src={url} />
}
```

```
</Suspense>
)
}

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

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

app/page.jsx

```
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="Your browser does not support the video">
      <source src={url} type="video/mp4" />
      <track src={captionsUrl} kind="subtitles" />
    </video>
  )
}
```

```
<track src={captionsUrl} kind="subtitle"
      Your browser does not support the video
      </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 `<ClVideoPlayer>` 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.

Was this helpful?    



 **Using App Router**
Features available in /app

 **Latest Version**
15.5.4

API Reference

Directives

Directives are used to modify the behavior of your...

Components

API Reference for Next.js built-in components.

File-system ...

API Reference for Next.js file-system conventions.

Functions

API Reference for Next.js Functions and Hooks.

Configuration

Learn how to configure Next.js applications.

CLI

API Reference for the Next.js Command Line...

Edge Runtime

API Reference for the Edge Runtime.

Turbopack

Turbopack is an incremental bundler optimize...

Was this helpful?    



Using App Router
Features available in /app



Directives



Latest Version
15.5.4



The following directives are available:

use cache

Learn how to use
the use cache
directive to cache...

use client

Learn how to use
the use client
directive to rende...

use server

Learn how to use
the use server
directive to...

Was this helpful?





Using App Router
Features available in /app



use cache



Latest Version
15.5.4



💡 This feature is currently available in the canary channel and subject to change. Try it out by upgrading Next.js, and share your feedback on GitHub.

The `use cache` directive allows you to mark a route, React component, or a function as cacheable. It can be used at the top of a file to indicate that all exports in the file should be cached, or inline at the top of function or component to cache the return value.

Usage

`use cache` is currently an experimental feature. To enable it, add the `useCache` option to your `next.config.ts` file:

```
TS next.config.ts TypeScript ▾ ⌂

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    useCache: true,
  },
}

export default nextConfig
```

Good to know: `use cache` can also be enabled with

the `cacheComponents` option.

Then, add `use cache` 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
}
```

How `use cache` works

Cache keys

A cache entry's key is generated using a serialized version of its inputs, which includes:

- Build ID (generated for each build)
- Function ID (a secure identifier unique to the function)
- The `serializable` ↗ function arguments (or props).

The arguments passed to the cached function, as well as any values it reads from the parent scope automatically become a part of the key. This

means, the same cache entry will be reused as long as its inputs are the same.

Non-serializable arguments

Any non-serializable arguments, props, or closed-over values will turn into references inside the cached function, and can be only passed through and not inspected nor modified. These non-serializable values will be filled in at the request time and won't become a part of the cache key.

For example, a cached function can take in JSX as a `children` prop and return `<div>{children}</div>`, but it won't be able to introspect the actual `children` object. This allows you to nest uncached content inside a cached component.



```
TS app/ui/cached-component.tsx TypeScript ▾ ⌂
function CachedComponent({ children }: { chil
  'use cache'
  return <div>{children}</div>
}
```

Return values

The return value of the cacheable function must be serializable. This ensures that the cached data can be stored and retrieved correctly.

`use cache at build time`

When used at the top of a `layout` or `page`, the route segment will be prerendered, allowing it to later be **revalidated**.

This means `use cache` cannot be used with **request-time APIs** like `cookies` or `headers`.

`use cache at runtime`

On the **server**, the cache entries of individual components or functions will be cached in-memory.

Then, on the **client**, any content returned from the server cache will be stored in the browser's memory for the duration of the session or until **revalidated**.

During revalidation

By default, `use cache` has server-side revalidation period of **15 minutes**. While this period may be useful for content that doesn't require frequent updates, you can use the `cacheLife` and `cacheTag` APIs to configure when the individual cache entries should be revalidated.

- `cacheLife` : Configure the cache entry lifetime.
- `cacheTag` : Create tags 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` API docs for more information.

Examples

Caching an entire route with `use cache`

To prerender an entire route, add `use cache` to the top of **both** the `layout` and `page` files. Each of these segments are treated as separate entry points in your application, and will be cached independently.

```
TS app/layout.tsx TypeScript ▾ ⌂

'use cache'

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

Any components imported and nested in `page` file will inherit the cache behavior of `page`.

```
TS app/page.tsx TypeScript ▾ ⌂

'use cache'

async function Users() {
  const users = await fetch('/api/users')
  // loop through users
}

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

Good to know:

- If `use cache` is added only to the `layout` or the `page`, only that route segment and any components imported into it will be cached.
- If any of the nested children in the route use [Dynamic APIs](#), then the route will opt out of prerendering.

Caching a component's output with `use cache`

You can use `use cache` at the component level to cache any fetches or computations performed within that component. The cache entry will be reused as long as the serialized props produce the same value in each instance.

```
TS app/components/bookings.ts... TypeScript ▾
```

```
export async function Bookings({ type = 'hair' } : BookingsProps) {
  'use cache'
  const data = await getBookingsData()
  return data
}

interface BookingsProps {
  type: string
}
```

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, a database query, or a slow computation.

```
TS app/actions.ts... TypeScript ▾
```

```
const getBookingsData = 'use cache'
```

```
export async function getData() {
  'use cache'

  const data = await fetch('/api/data')
  return data
}
```

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.



The screenshot shows a code editor interface with a TypeScript file named `app/page.tsx`. The code defines a `Page` component that uses the `getData` function to fetch uncached data. It then wraps this data in a `CacheComponent`, which in turn wraps a `DynamicComponent` that receives the uncached data as a prop. The `CacheComponent` also has a `children` prop, which is passed through to a `PrerenderedComponent` that displays the cached data. The code editor includes syntax highlighting for TypeScript and JSX, and a status bar at the bottom indicates the file type and language.

```
export default 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')
  return (
    <div>
      <PrerenderedComponent data={cachedData}>
        {children}
      </PrerenderedComponent>
    </div>
  )
}
```

You can also pass Server Actions through cached components to Client Components without invoking them inside the cacheable function.



The screenshot shows a code editor interface with a TypeScript file named `app/page.tsx`. The code imports a `ClientComponent` from a local module. It then defines a `Page` component that uses the `ClientComponent` to render its content. The code editor includes syntax highlighting for TypeScript and JSX, and a status bar at the bottom indicates the file type and language.

```
import ClientComponent from './ClientComponent'

export default async function Page() {
```

```

const performUpdate = async () => {
  'use server'
  // Perform some server-side update
  await db.update(...)
}

return <CacheComponent performUpdate={performUpdate}>

```

```

async function CachedComponent({
  performUpdate,
}: {
  performUpdate: () => Promise<void>
}) {
  'use cache'
  // Do not call performUpdate here
  return <ClientComponent action={performUpdate}>
}

```

app/ClientComponent.tsx TypeScript ▾

```

'type': 'client'

export default function ClientComponent({
  action,
}: {
  action: () => Promise<void>
}) {
  return <button onClick={action}>Update</button>
}

```

Platform Support

Deployment Option	Supported
Node.js server	Yes
Docker container	Yes
Static export	No
Adapters	Platform-specific

Learn how to [configure caching](#) when self-hosting Next.js.

Version History

Version	Changes
---------	---------

v15.0.0	"use cache" is introduced as an experimental feature.
---------	---

Related

View related API references.

useCache

Learn how to enable the useCache flag in...

cacheCompo...

Learn how to enable the cacheComponent...

cacheLife

Learn how to set up cacheLife configurations in...

cacheTag

Learn how to use the cacheTag function to...

cacheLife

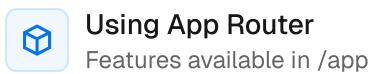
Learn how to use the cacheLife function to set th...

revalidateTag

API Reference for the revalidateTag function.

Was this helpful?  





use client

The `'use client'` directive declares an entry point for the components 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.

Good to know:

You do not need to add the `'use client'` directive to every file that contains Client Components. You only need to add it to the files whose components you want to render directly within Server Components. The `'use client'` directive defines the client-server boundary ↗, and the components exported from such a file serve as entry points to the client.

Usage

To declare an entry point for the Client Components, add the `'use client'` directive **at the top of the file**, before any imports:

```
TS app/components/counter.tsx TypeScript ▾   
'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>
        )
      }
```

When using the `'use client'` directive, the props of the Client Components must be [serializable](#). This means the props need to be in a format that React can serialize when sending data from the server to the client.



```
TS app/components/counter.tsx TypeScript ▾
```

```
'use client'

export default function Counter({
  onClick /* ✖ Function is not serializable */
}) {
  return (
    <div>
      <button onClick={onClick}>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:

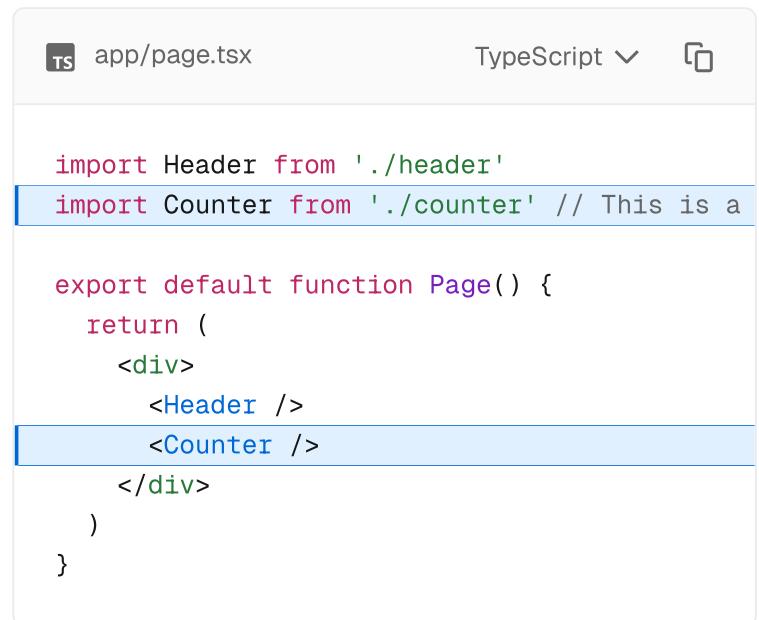
1. **Server Components:** Use for static content, data fetching, and SEO-friendly elements.
2. **Client Components:** Use for interactive elements that require state, effects, or browser APIs.

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.



```
TS app/page.tsx TypeScript ⓘ  
  
import Header from './header'  
import Counter from './counter' // This is a  
  
export default function Page() {  
  return (  
    <div>  
      <Header />  
      <Counter />  
    </div>  
  )  
}
```

Reference

See the [React documentation ↗](#) for more information on `'use client'`.

Was this helpful?    



Using App Router
Features available in /app



Latest Version
15.5.4



use 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.

```
TS app/actions.ts TypeScript ▾ ⌂
'use server'
import { db } from '@/lib/db' // Your database

export async function createUser(data: { name: string }) {
  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.

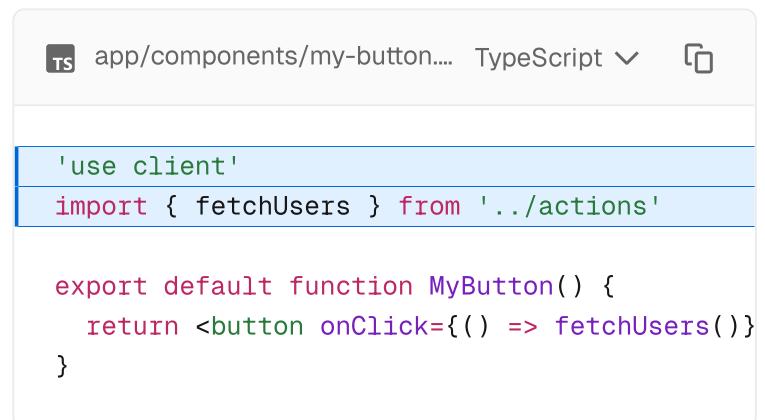
Assuming you have a `fetchUsers` Server Function in `actions.ts`:



```
'use server'
import { db } from '@/lib/db' // Your database

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()}>
}
```

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 { EditPost } from './edit-post'
import { revalidatePath } from 'next/cache'
```

```
export default async function PostPage({ para
  const post = await getPost(params.id)

  async function updatePost(formData: FormData)
    'use server'
    await savePost(params.id, formData)
    revalidatePath(`/posts/${params.id}`)
  }

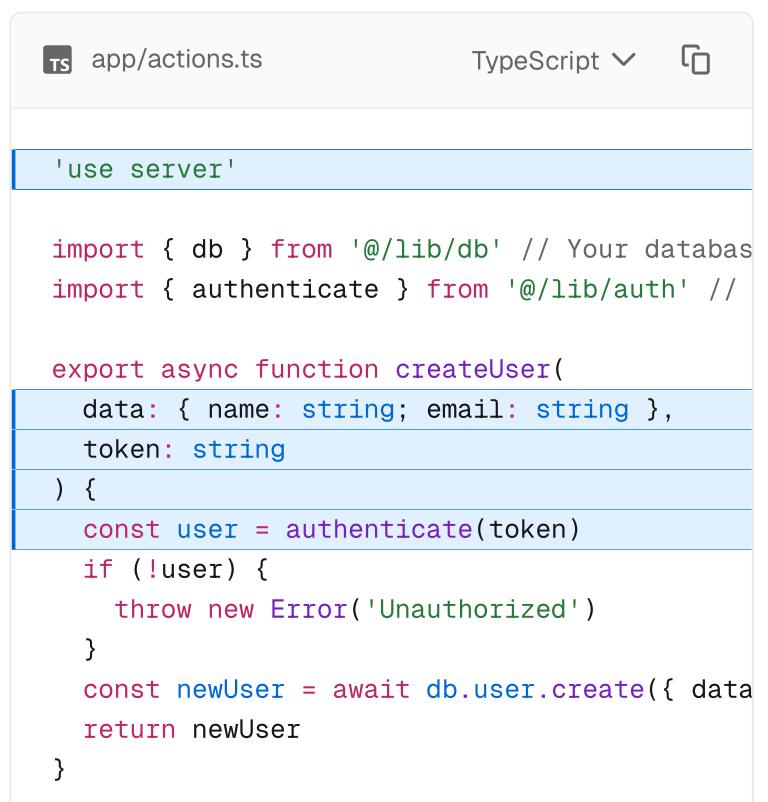
  return <EditPost updatePostAction={updatePo
}>
```

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.



The screenshot shows a code editor interface with the following details:

- File name: `app/actions.ts`
- TypeScript version: `TypeScript` (dropdown menu)
- Code content:

```
'use server'

import { db } from '@/lib/db' // Your database
import { authenticate } from '@/lib/auth' //

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
}
```

Reference

See the [React documentation](#) ↗ for more information on `useServer`.

Was this helpful?    



Using App Router
Features available in /app



Components



Latest Version
15.5.4



Font

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

Form Compo...

Learn how to use
the ``<Form>``
component to...

Image Compo...

Optimize Images
in your Next.js
Application using...

Link Compon...

Enable fast client-
side navigation
with the built-in...

Script Compo...

Optimize third-
party scripts in
your Next.js...

Was this helpful?

Using App Router
Features available in /app

Latest Version
15.5.4

Font Module

`next/font` automatically optimizes your fonts (including custom fonts) and removes external network requests for improved privacy and performance.

It includes **built-in automatic self-hosting** for any font file. This means you can optimally load web fonts with no [layout shift ↗](#).

You can also conveniently use all [Google Fonts ↗](#). 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.**

```
TS app/layout.tsx TypeScript   
  
import { Inter } from 'next/font/google'  
  
// If loading a variable font, you don't need  
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>  
  )  
}
```

 Watch: Learn more about using `next/font` →
[YouTube \(6 minutes\)](#).

Reference

Key	font/google	font/local	Ty
<code>src</code>	X	✓	St or of Ol
<code>weight</code>	✓	✓	St or
<code>style</code>	✓	✓	St or
<code>subsets</code>	✓	X	Ar St
<code>axes</code>	✓	X	Ar St
<code>display</code>	✓	✓	St
<code>preload</code>	✓	✓	Bo
<code>fallback</code>	✓	✓	Ar St
<code>adjustFontFallback</code>	✓	✓	Bo or St
<code>variable</code>	✓	✓	St
<code>declarations</code>	X	✓	Ar Ol

`src`

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

```
Array<{path: string, weight?: string, style?: string}>
```

) relative to the directory where the font loader function is called.

Used in `next/font/local`

- Required

Examples:

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

weight

The font `weight` ↗ with the following possibilities:

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

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

- Required if the font being used is **not** variable ↗

Examples:

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

style

The font `style` ↗ with the following possibilities:

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

Used in `next/font/google` and

`next/font/local`

- Optional

Examples:

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

subsets

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

Used in `next/font/google`

- Optional

Examples:

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

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

axes

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

Used in `next/font/google`

- Optional

Examples:

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

display

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

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

- Optional

Examples:

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

preload

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

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

- Optional

Examples:

- `preload: false`

fallback

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

- Optional

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

Examples:

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

arial

adjustFontFallback

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

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

- Optional

Examples:

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

variable

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

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

- Optional

Examples:

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

declarations

An array of font face descriptor ↗ key-value pairs that define the generated @font-face further.

Used in next/font/local

- Optional

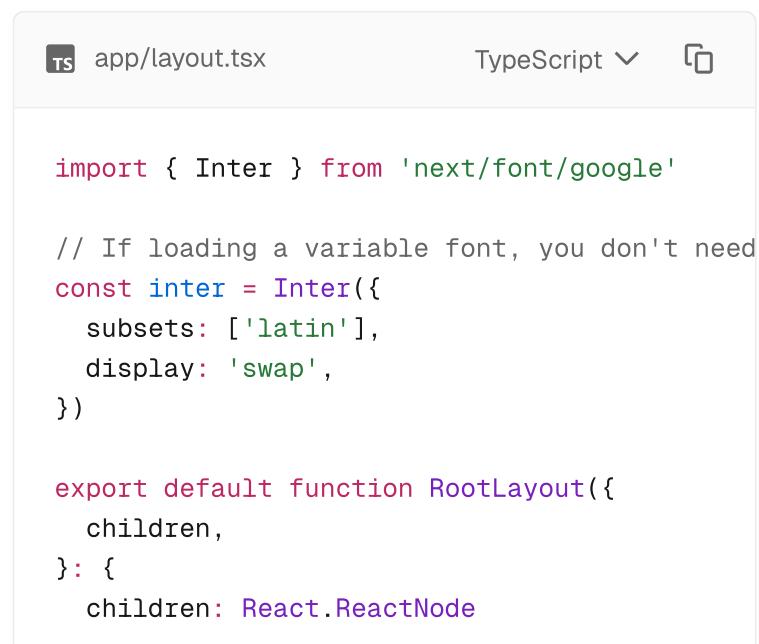
Examples:

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

Examples

Google Fonts

To use a Google font, import it from next/font/google as a function. We recommend using variable fonts ↗ for the best performance and flexibility.



The screenshot shows a code editor window with a TypeScript file named 'app/layout.tsx'. The code imports the 'Inter' font from 'next/font/google' and defines a 'RootLayout' component that uses the font.

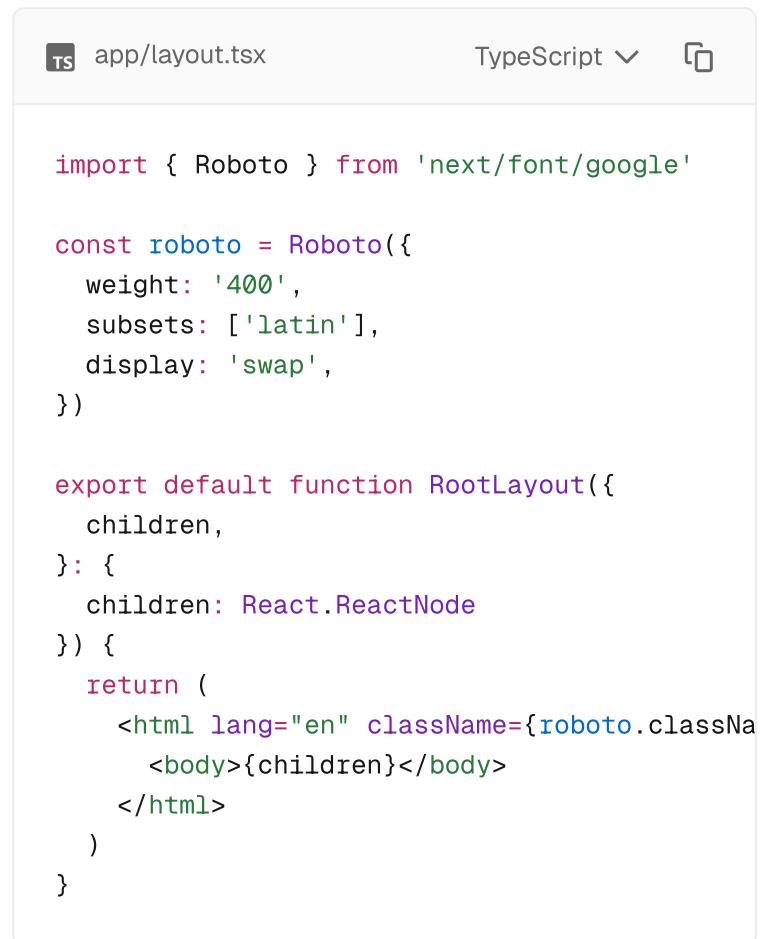
```
TS app/layout.tsx TypeScript ▾ ⌂
import { Inter } from 'next/font/google'

// If loading a variable font, you don't need
const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <head>
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <link href={inter.href} rel="stylesheet" />
      </head>
      <body>
        {children}
      </body>
    </html>
  )
}
```

```
) {  
    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**:

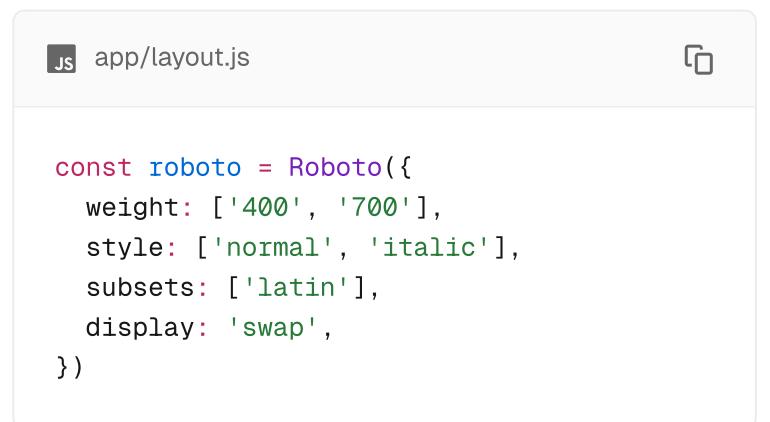


The screenshot shows a code editor window with a TypeScript file named "app/layout.tsx". The file contains the following code:

```
TS app/layout.tsx TypeScript ▾
```

```
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>  
    )  
}
```

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



The screenshot shows a code editor window with a JavaScript file named "app/layout.js". The file contains the following code:

```
JS app/layout.js
```

```
const roboto = Roboto({  
    weight: ['400', '700'],  
    style: ['normal', 'italic'],  
    subsets: ['latin'],  
    display: 'swap',  
})
```

Good to know: Use an underscore (_) for font names with multiple words. E.g. `Roboto Mono` should be imported as `Roboto_Mono`.

Specifying a subset

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

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



```
TS app/layout.tsx TypeScript ▾ ⌂
const inter = Inter({ subsets: ['latin'] })
```

View the [Font API Reference](#) for more information.

Using Multiple Fonts

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

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



```
TS app/fonts.ts TypeScript ▾ ⌂
import { Inter, Roboto_Mono } from 'next/font'

export const inter = Inter({
  subsets: ['latin'],
```

```
        display: 'swap',
    })

export const roboto_mono = Roboto_Mono({
    subsets: ['latin'],
    display: 'swap',
})
```

app/layout.tsx TypeScript

```
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>
    )
}
```

app/page.tsx TypeScript

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

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

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

Alternatively, you can create a [CSS variable](#) and use it with your preferred CSS solution:

app/layout.tsx TypeScript

```
import { Inter, Roboto_Mono } from 'next/font';
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.varia
      <body>
        <h1>My App</h1>
        <div>{children}</div>
      </body>
    </html>
  )
}
```

app/global.css

```
html {
  font-family: var(--font-inter);
}

h1 {
  font-family: var(--font-roboto-mono);
}
```

In the example above, `Inter` will be applied globally, and any `<h1>` tags will be styled with `Roboto Mono`.

Recommendation: Use multiple fonts conservatively since each new font is an additional resource the client has to download.

Local Fonts

Import `next/font/local` and specify the `src` of your local font file. We recommend using [variable fonts ↗](#) for the best performance and flexibility.



The screenshot shows a code editor window with the following details:

- File name: `app/layout.tsx`
- TypeScript version: `TypeScript v`
- Code content:

```
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>
  )
}
```

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',
    },
  ],
})
```

```
        style: 'italic',
    },
],
})
```

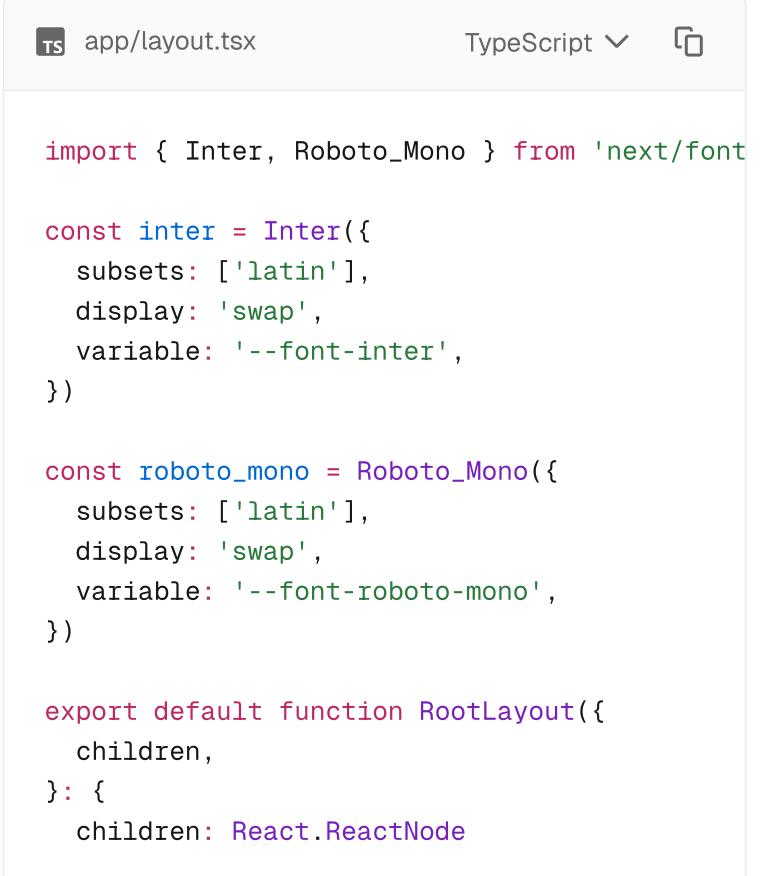
View the [Font API Reference](#) for more information.

With Tailwind CSS

`next/font` integrates seamlessly with [Tailwind CSS](#) ↗ using [CSS variables](#).

In the example below, we use the `Inter` and `Roboto_Mono` fonts from `next/font/google` (you can use any Google Font or Local Font). Use the `variable` option to define a CSS variable name, such as `inter` and `roboto_mono` for these fonts, respectively. Then, apply `inter.variable` and `roboto_mono.variable` to include the CSS variables in your HTML document.

Good to know: You can add these variables to the `<html>` or `<body>` tag, depending on your preference, styling needs or project requirements.



The screenshot shows a code editor interface with a TypeScript file named `app/layout.tsx`. The code defines two font configurations: `inter` and `roboto_mono`, both using the `Robo-Mono` font from the `next/font` package. Each font configuration includes a `variable` option to define a CSS variable name. The code is then used in a `RootLayout` component to apply these variables to the `<html>` tag.

```
TS app/layout.tsx TypeScript ▾ ⌂
import { Inter, Roboto_Mono } from 'next/font'

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>
      <head>
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <link href={inter.variable} rel="stylesheet" />
        <link href={roboto_mono.variable} rel="stylesheet" />
      </head>
      <body>
        {children}
      </body>
    </html>
  )
}
```

```
) {
    return (
        <html
            lang="en"
            className={`${inter.variable} ${roboto_}
        >
            <body>{children}</body>
        </html>
    )
}
```

Finally, add the CSS variable to your [Tailwind CSS config](#):

```
global.css
```

```
@import 'tailwindcss';

@theme inline {
    --font-sans: var(--font-inter);
    --font-mono: var(--font-roboto-mono);
}
```

Tailwind CSS v3

```
tailwind.config.js
```

```
/* @type {import('tailwindcss').Config} */
module.exports = {
    content: [
        './pages/**/*.{js,ts,jsx,tsx}',
        './components/**/*.{js,ts,jsx,tsx}',
        './app/**/*.{js,ts,jsx,tsx}',
    ],
    theme: {
        extend: {
            fontFamily: {
                sans: ['var(--font-inter)'],
                mono: ['var(--font-roboto-mono)'],
            },
        },
    },
    plugins: [],
}
```

You can now use the `font-sans` and `font-mono` utility classes to apply the font to your elements.

```
<p class="font-sans ...">The quick brown fox  
<p class="font-mono ...">The quick brown fox
```

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
```

`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',
})
```

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>
```

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`.

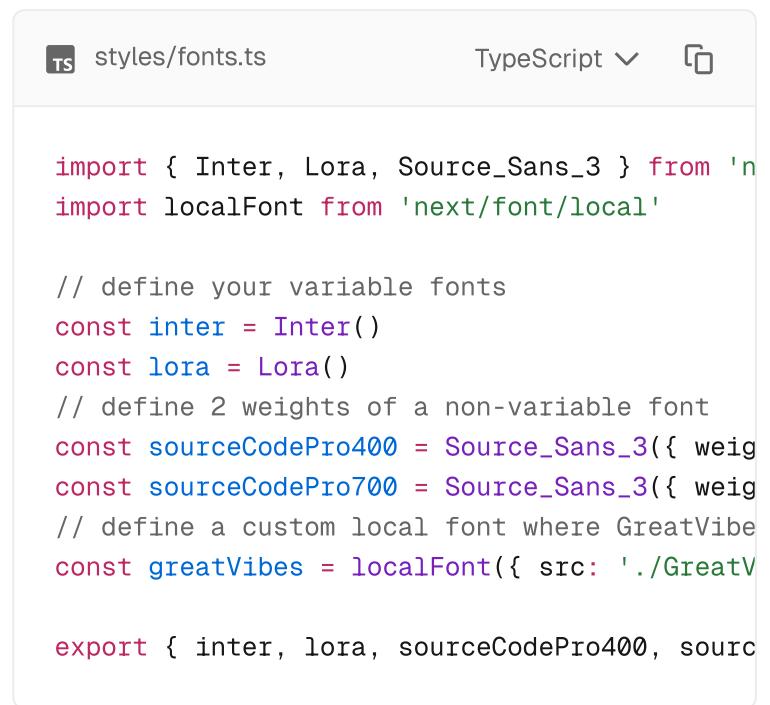
Using a font definitions file

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:



```
TS styles/fonts.ts TypeScript ▾
```

```
import { Inter, Lora, Source_Sans_3 } from 'next/font/local'

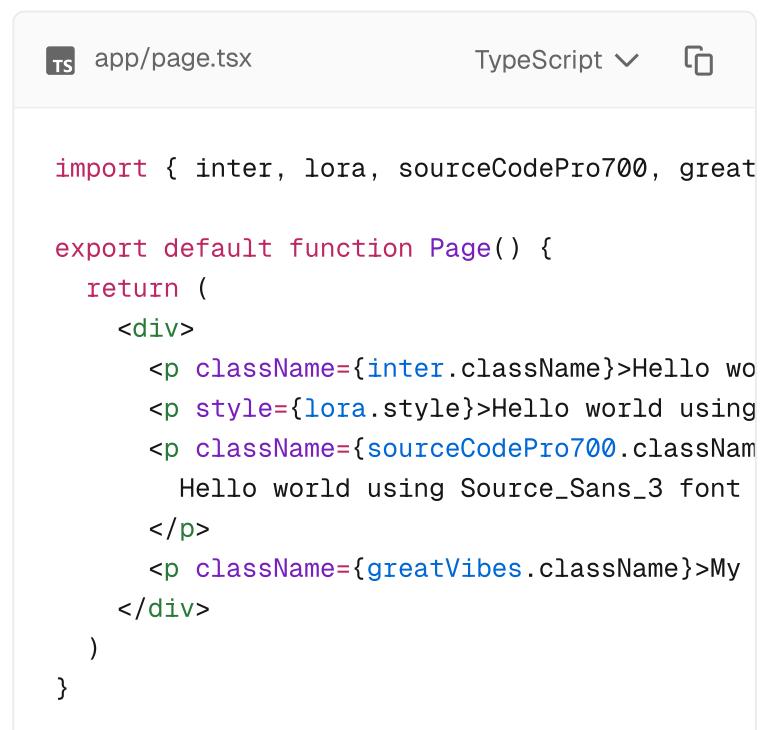
// define your variable fonts
const inter = Inter()
const lora = Lora()

// 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 GreatVibe is the fallback
const greatVibes = localFont({ src: './GreatVibes-Regular.woff2' })

export { inter, lora, sourceCodePro400, sourceCodePro700, greatVibes }
```

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

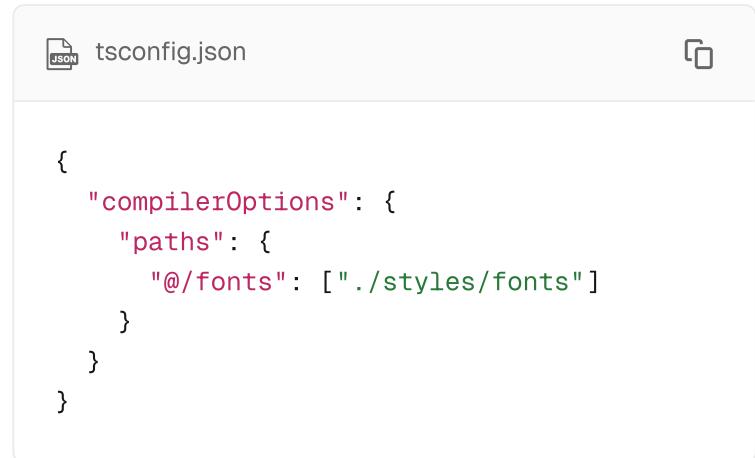


```
TS app/page.tsx TypeScript ▾
```

```
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</p>
      <p className={greatVibes.className}>My favorite 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:



A screenshot of a code editor showing a `tsconfig.json` file. The file contains the following JSON code:

```
{  
  "compilerOptions": {  
    "paths": {  
      "@/fonts": ["./styles/fonts"]  
    }  
  }  
}
```

You can now import any font definition as follows:



A screenshot of a code editor showing an `app/about/page.tsx` file. The file contains the following TypeScript code:

```
import { greatVibes, sourceCodePro400 } from
```

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.

Version Changes

Version Changes

v13.2.0 @next/font renamed to next/font.

Installation no longer required.

v13.0.0 @next/font was added.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

Form Component

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:

```
TS /app/ui/search.tsx TypeScript ▾ ⌂

import Form from 'next/form'

export default function Page() {
  return (
    <Form action="/search">
      /* On submission, the input value will
       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.

`action (string)` Props

When `action` is a string, the `<Form>` component supports the following props:

Prop	Example	Type	Required
<code>action</code>	<code>action="/search"</code>	<code>string</code> (URL or relative path)	Yes
<code>replace</code>	<code>replace={false}</code>	<code>boolean</code>	-
<code>scroll</code>	<code>scroll={true}</code>	<code>boolean</code>	-
<code>prefetch</code>	<code>prefetch={true}</code>	<code>boolean</code>	-

- `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
<code>action</code>	<code>action={myAction}</code>	<code>function</code> (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

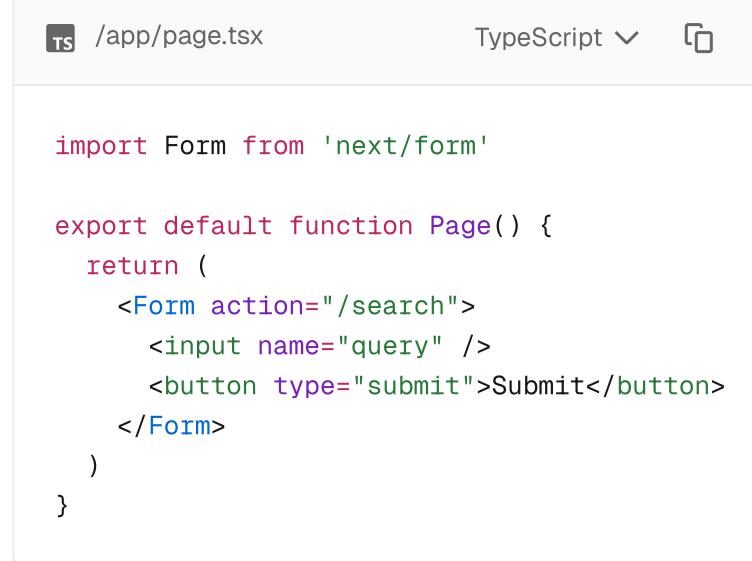
navigation, however, this approach doesn't support prefetching.

- 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'

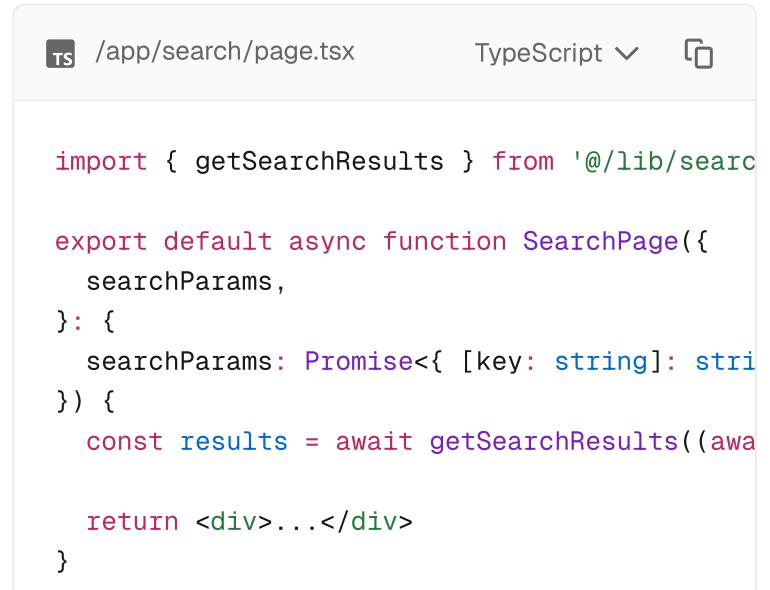
export default function Page() {
  return (
    <Form action="/search">
      <input name="query" />
      <button type="submit">Submit</button>
    </Form>
  )
}
```

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: Promise<{ [key: string]: string }>
}) {
  const results = await getSearchResults((awa

  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`:

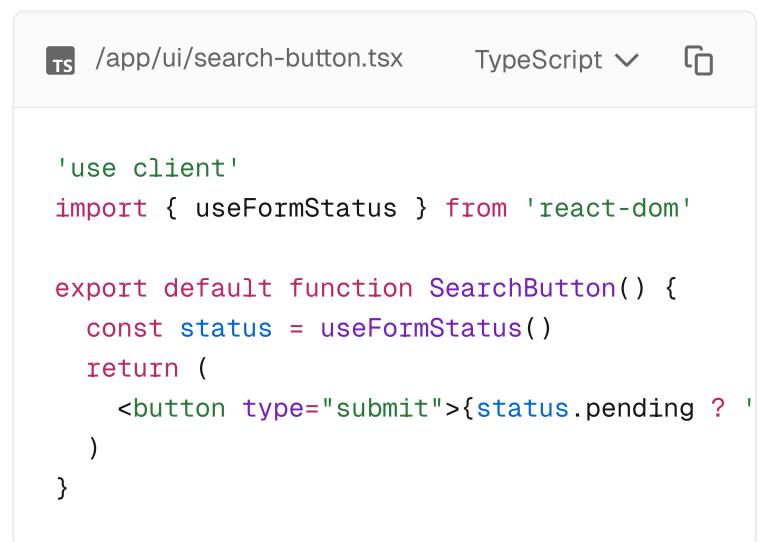


The screenshot shows a code editor window with the file path `/app/search/loading.tsx` at the top. The code is a simple function named `Loading` that returns a `<div>` element containing the text "Loading...". The code editor interface includes tabs for "TypeScript" and a copy icon.

```
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:

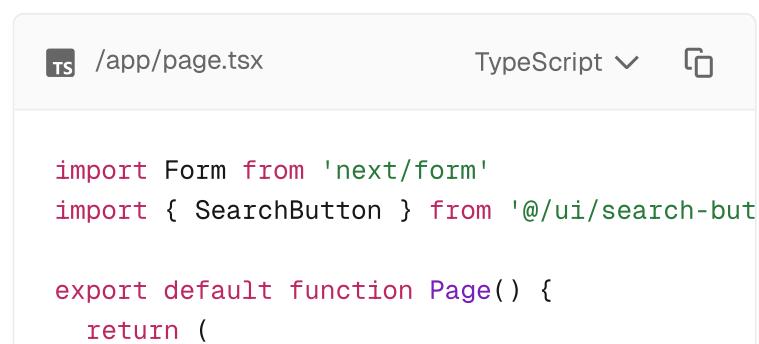


The screenshot shows a code editor window with the file path `/app/ui/search-button.tsx` at the top. The code defines a component `SearchButton` that uses the `useFormStatus` hook from `'react-dom'`. It checks the `status.pending` value to determine if the button should be labeled "Submitting..." or "Submit". The code editor interface includes tabs for "TypeScript" and a copy icon.

```
'use client'
import { useFormStatus } from 'react-dom'

export default function SearchButton() {
  const status = useFormStatus()
  return (
    <button type="submit">{status.pending ? 'Submitting...' : 'Submit'}</button>
  )
}
```

Then, update the search form page to use the `SearchButton` component:



The screenshot shows a code editor window with the file path `/app/page.tsx` at the top. The code imports `Form` from `'next/form'` and the `SearchButton` component from `@/ui/search-button`. It then defines a function `Page` that returns a `<Form>` component with a `SearchButton` as its submit button. The code editor interface includes tabs for "TypeScript" and a copy icon.

```
import Form from 'next/form'
import { SearchButton } from '@/ui/search-button'

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

```
<Form action="/search">
  <input name="query" />
  <SearchButton />
</Form>
)
}
```

Mutations with Server Actions

You can perform mutations by passing a function to the `action` prop.

```
TS /app/posts/create/page.tsx TypeScript ▾ ⌂

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.

```
TS /app/posts/actions.ts TypeScript ▾ ⌂

'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}`)
}
```

Then, in the new page, you can fetch data using the `params` prop:

```
TS /app/posts/[id]/page.tsx TypeScript ▾ ⌂

import {getPost} from '@/posts/data'

export default async function PostPage({
  params,
}: {
  params: Promise<{ id: string }>
}) {
  const { id } = await params
  const data = await getPost(id)

  return (
    <div>
      <h1>{data.title}</h1>
      {/* ... */}
    </div>
  )
}
```

See the [Server Actions](#) docs for more examples.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

Image Component

The Next.js Image component extends the HTML `` element for automatic image optimization.

 app/page.js



```
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="/profile.png"
      width={500}
      height={500}
      alt="Picture of the author"
    />
  )
}
```

Reference

Props

The following props are available:

Prop	Example
<code>src</code>	<code>src="/profile.png"</code>
<code>alt</code>	<code>alt="Picture of the author"</code>
<code>width</code>	<code>width={500}</code>

Prop

`height`

`height={500}`

`fill`

`fill={true}`

`loader`

`loader={imageLoader}`

`sizes`

`sizes="(max-width: 768px) 100v
33vw"`

`quality`

`quality={80}`

`priority`

`priority={true}`

`placeholder`

`placeholder="blur"`

`style`

`style={{objectFit: "contain"}}`

`onLoadingComplete`

`onLoadingComplete={img =>
done()}`

`onLoad`

`onLoad={event => done()}`

`onError`

`onError(event => fail())`

`loading`

`loading="lazy"`

`blurDataURL`

`blurDataURL="data:image/jpeg .."`

`overrideSrc`

`overrideSrc="/seo.png"`

`src`

The source of the image. Can be one of the following:

An internal path string.

```
<Image src="/profile.png" />
```

An absolute external URL (must be configured with [remotePatterns](#)).

```
<Image src="https://example.com/profile.png"
```

A static import.

```
import profile from './profile.png'

export default function Page() {
  return <Image src={profile} />
}
```

Good to know: For security reasons, the Image Optimization API using the default `loader` will *not* forward headers when fetching the `src` image. If the `src` image requires authentication, consider using the `unoptimized` property to disable Image Optimization.

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 about [image accessibility guidelines ↗](#).

width and height

The `width` and `height` properties represent the [intrinsic ↗](#) image size in pixels. This property is used to infer the correct **aspect ratio** used by

browsers to reserve space for the image and avoid layout shift during loading. It does not determine the *rendered size* of the image, which is controlled by CSS.

```
<Image src="/profile.png" width={500} height=
```

You **must** set both `width` and `height` properties unless:

- The image is statically imported.
- The image has the `fill` property

If the height and width are unknown, we recommend using the `fill` property.

`fill`

A boolean that causes the image to expand to the size of the parent element.

```
<Image src="/profile.png" fill={true} />
```

Positioning:

- The parent element **must** assign `position`:
`"relative"`, `"fixed"`, `"absolute"`.
- By default, the `` element uses `position: "absolute"`.

Object Fit:

If no styles are applied to the image, the image will stretch to fit the container. You can use `objectFit` to control cropping and scaling.

- `"contain"` : The image will be scaled down to fit the container and preserve aspect ratio.
- `"cover"` : The image will fill the container and be cropped.

Learn more about [position ↗](#) and [object-fit ↗](#).

loader

A custom function used to generate the image URL. The function receives the following parameters, and returns a URL string for the image:

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

```
'use client'

import Image from 'next/image'

const imageLoader = ({ src, width, quality }) =>
  return `https://example.com/${src}?w=${width}&q=${quality}`

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 `onLoad`, which accept a function, requires using [Client Components ↗](#) to serialize the provided function.

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

sizes

Define the sizes of the image at different breakpoints. Used by the browser to choose the most appropriate size from the generated `srcset`.

```
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) 500px, 600px"
      />
    </div>
  )
}
```

`sizes` should be used when:

- The image is using the `fill` prop
- CSS is used to make the image responsive

If `sizes` is missing, the browser assumes the image will be as wide as the viewport (`100vw`). This can cause unnecessarily large images to be downloaded.

In addition, `sizes` affects how `srcset` is generated:

- Without `sizes`: Next.js generates a limited `srcset` (e.g. `1x`, `2x`), suitable for fixed-size images.
- With `sizes`: Next.js generates a full `srcset` (e.g. `640w`, `750w`, etc.), optimized for responsive layouts.

Learn more about `srcset` and `sizes` on [web.dev](#) ↗ and [mdn](#) ↗.

`quality`

An integer between `1` and `100` that sets the quality of the optimized image. Higher values increase file size and visual fidelity. Lower values reduce file size but may affect sharpness.

```
// Default quality is 75  
<Image quality={75} />
```

If you've configured `qualities` in `next.config.js`, the value must match one of the allowed entries.

Good to know: If the original image is already low quality, setting a high quality value will increase the file size without improving appearance.

style

Allows passing CSS styles to the underlying image element.

```
const imageStyle = {  
  borderRadius: '50%',  
  border: '1px solid #fff',  
  width: '100px',  
  height: 'auto',  
}  
  
export default function ProfileImage() {  
  return <Image src="..." style={imageStyle}>  
}
```

Good to know: If you're using the `style` prop to set a custom width, be sure to also set `height: 'auto'` to preserve the image's aspect ratio.

priority

A boolean that indicates if the image should be preloaded.

```
// Default priority is false  
<Image priority={false} />
```

- `true` : Preloads ↗ the image. Disables lazy loading.
- `false` : Lazy loads the image.

When to use it:

- The image is above the fold.
- The image is the [Largest Contentful Paint \(LCP\)](#) ↗ element.
- You want to improve the initial loading performance of your page.

When not to use it:

- When the `loading` prop is used (will trigger warnings).

`loading`

Controls when the image should start loading.

```
// Defaults to lazy
<Image loading="lazy" />
```

- `lazy` : Defer loading the image until it reaches a calculated distance from the viewport.
- `eager` : Load the image immediately, regardless of its position in the page.

Use `eager` only when you want to ensure the image is loaded immediately.

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

`placeholder`

Specifies a placeholder to use while the image is loading, improving the perceived loading performance.

```
// defaults to empty  
<Image placeholder="empty" />
```

- `empty` : No placeholder while the image is loading.
- `blur` : Use a blurred version of the image as a placeholder. Must be used with the `blurDataURL` property.
- `data:image/...` : Uses the [Data URL ↗](#) as the placeholder.

Examples:

- [blur placeholder ↗](#)
- [Shimmer effect with data URL placeholder prop ↗](#)
- [Color effect with blurDataURL prop ↗](#)

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

blurDataURL

A [Data URL ↗](#) to be used as a placeholder image before the image successfully loads. Can be automatically set or used with the `placeholder="blur"` property.

```
<Image placeholder="blur" blurDataURL="..." /
```

The image is automatically enlarged and blurred, so a very small image (10px or less) is recommended.

Automatic

If `src` is a static import of a `jpg`, `png`, `webp`, or `avif` file, `blurDataURL` is added automatically—unless the image is animated.

Manually set

If the image is dynamic or remote, you must provide `blurDataURL` yourself. To generate one, you can use:

- A online tool like [png-pixel.com ↗](#)
- A library like [Placeholder ↗](#)

A large `blurDataURL` may hurt performance. Keep it small and simple.

Examples:

- Default `blurDataURL` prop [↗](#)
- Color effect with `blurDataURL` prop [↗](#)

onLoad

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

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

The callback function will be called with one argument, the event which has a `target` that references the underlying `` element.

Good to know: Using props like `onLoad`, which accept a function, requires using [Client Components ↗](#) to serialize the provided function.

onError

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

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

Good to know: Using props like `onError`, which accept a function, requires using [Client Components](#) ↗ to serialize the provided function.

unoptimized

A boolean that indicates if the image should be optimized. This is useful for images that do not benefit from optimization such as small images (less than 1KB), vector images (SVG), or animated images (GIF).

```
import Image from 'next/image'

const UnoptimizedImage = (props) => {
  // Default is false
  return <Image {...props} unoptimized />
}
```

- `true` : The source image will be served as-is from the `src` instead of changing quality, size, or format.
- `false` : The source image will be optimized.

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

next.config.js

```
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 ``.

```
JS input.js
```

```
<Image src="/profile.jpg" />
```

```
output.html
```

```

```

decoding

A hint to the browser indicating if it should wait for the image to be decoded before presenting other

content updates or not.

```
// Default is async  
<Image decoding="async" />
```

- `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. The browser chooses the best approach.

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.

Deprecated props

`onLoadingComplete`

Warning: Deprecated in Next.js 14, use `onLoad` instead.

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

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

```
'use client'
```

```
<Image onLoadingComplete={(img) => console.lo
```

Good to know: Using props like `onLoadingComplete`, which accept a function, requires using [Client Components ↗](#) to serialize the provided function.

Configuration options

You can configure the Image Component in `next.config.js`. The following options are available:

`localPatterns`

Use `localPatterns` in your `next.config.js` file to allow images from specific local paths to be optimized and block all others.

```
JS next.config.js

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

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` error.

`remotePatterns`

Use `remotePatterns` in your `next.config.js` file to allow images from specific external paths and

block all others. This ensures that only external images from your account can be served.

```
JS next.config.js Copy  
  
module.exports = {  
  images: {  
    remotePatterns: [new URL('https://example')],  
  },  
}
```

You can also configure `remotePatterns` using the object:

```
JS next.config.js Copy  
  
module.exports = {  
  images: {  
    remotePatterns: [  
      {  
        protocol: 'https',  
        hostname: 'example.com',  
        port: '',  
        pathname: '/account123/**',  
        search: '',  
      },  
    ],  
  },  
}
```

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`.

Wildcard Patterns:

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. This syntax does not work in the middle of the pattern.

`next.config.js`

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
        port: '',
        search: ''
      },
    ],
  },
}
```

This allows subdomains like `image.example.com`.
Query strings and custom ports are still blocked.

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.

Query Strings:

You can also restrict query strings using the `search` property:

`next.config.js`

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'assets.example.com',
        search: '?v=1727111025337'
      },
    ],
  },
}
```

```
},  
}
```

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`.

`loaderFile`

`loaderFiles` allows you to use a custom image optimization service instead of Next.js.

```
JS next.config.js
```

```
module.exports = {  
  images: {  
    loader: 'custom',  
    loaderFile: './my/image/loader.js',  
  },  
}
```

The path must be relative to the project root. The file must export a default function that returns a URL string:

```
JS my/image/loader.js
```

```
'use client'  
  
export default function myImageLoader({ src,  
  return `https://example.com/${src}?w=${width}`  
})
```

Example:

- [Custom Image Loader Configuration](#)

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

path

If you want to change or prefix the default path for the Image Optimization API, you can do so with the `path` property. The default value for `path` is `/_next/image`.

```
JS next.config.js

module.exports = {
  images: {
    path: '/my-prefix/_next/image',
  },
}
```

deviceSizes

`deviceSizes` allows you to specify a list of device width breakpoints. These widths are used when the `next/image` component uses `sizes` prop to ensure the correct image is served for the user's device.

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

```
JS next.config.js

module.exports = {
  images: {
    deviceSizes: [640, 750, 828, 1080, 1200],
  },
}
```

imageSizes

`imageSizes` allows you to specify a list of image widths. These widths are concatenated with the array of `device sizes` to form the full array of sizes used to generate image `srcset` ↗.

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

```
module.exports = {
  images: {
    imageSizes: [16, 32, 48, 64, 96, 128, 256]
  },
}
```

`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`.

qualities

`qualities` allows you to specify a list of image quality values.

```
module.exports = {
  images: {
    qualities: [25, 50, 75],
  },
}
```

In the example above, only three qualities are allowed: 25, 50, and 75. If the `quality` prop does not match a value in this array, the image will fail with a `400` Bad Request.

formats

`formats` allows you to specify a list of image formats to be used.

```
module.exports = {
  images: {
    // Default
    formats: ['image/webp'],
  },
}
```

}

Next.js automatically detects 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), it will use the original image's format.

You can enable AVIF support, which will fallback to the original format of the src image if the browser [does not support AVIF ↗](#):

next.config.js

```
module.exports = {
  images: {
    formats: ['image/avif'],
  },
}
```

Good to know:

- We still recommend using WebP for most use cases.
- 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, but 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.

`minimumCacheTTL`

`minimumCacheTTL` allows you to 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`.

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

```
JS next.config.js

module.exports = {
  images: {
    minimumCacheTTL: 60, // 1 minute
  },
}
```

You can increase the TTL to reduce the number of revalidations and potentially lower cost:

```
JS next.config.js

module.exports = {
  images: {
    minimumCacheTTL: 2678400, // 31 days
  },
}
```

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 the cached file `<distDir>/cache/images`.

disableStaticImages

`disableStaticImages` allows you to disable static image imports.

The default behavior allows you to import static files such as `import icon from './icon.png'` and then pass that to the `src` property. In some cases, you may wish to disable this feature if it conflicts with other plugins that expect the import to behave differently.

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

`JS next.config.js`



```
module.exports = {
  images: {
    disableStaticImages: true,
  },
}
```

dangerouslyAllowSVG

`dangerouslyAllowSVG` allows you to serve SVG images.

`JS next.config.js`



```
module.exports = {
  images: {
    dangerouslyAllowSVG: true,
  },
}
```

By default, Next.js does not optimize SVG images for a few reasons:

- SVG is a vector format meaning it can be resized losslessly.
- SVG has many of the same features as HTML/CSS, which can lead to vulnerabilities

without proper Content Security Policy (CSP) headers.

We recommend using the `unoptimized` prop when the `src` prop is known to be SVG. This happens automatically when `src` ends with `".svg"`.

```
<Image src="/my-image.svg" unoptimized />
```

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.

```
JS next.config.js
```

```
module.exports = {
  images: {
    dangerouslyAllowSVG: true,
    contentDispositionType: 'attachment',
    contentSecurityPolicy: "default-src 'self'
  },
}
```

contentDispositionType

`contentDispositionType` allows you to configure the [Content-Disposition](#) ↗ header.

```
JS next.config.js
```

```
module.exports = {
  images: {
    contentDispositionType: 'inline',
  },
}
```

contentSecurityPolicy

`contentSecurityPolicy` allows you to configure the [Content-Security-Policy](#) ↗ header for images. This is particularly important when using [dangerouslyAllowSVG](#) to prevent scripts embedded in the image from executing.



```
next.config.js
```

```
module.exports = {
  images: {
    contentSecurityPolicy: "default-src 'self"
  },
}
```

By default, the [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.

Deprecated configuration options

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:

```
JS next.config.js

module.exports = {
  images: {
    domains: ['assets.acme.com'],
  },
}
```

Functions

getImageProps

The `getImageProps` function can be used to get the props that would be passed to the underlying `` element, and instead pass them to another component, style, canvas, etc.

```
import { getImageProps } from 'next/image'

const { props } = getImageProps({
  src: 'https://example.com/image.jpg',
  alt: 'A scenic mountain view',
  width: 1200,
  height: 800,
})

function ImageWithCaption() {
  return (
    <figure>
      <img {...props} />
      <figcaption>A scenic mountain view</fig
    </figure>
  )
}
```

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.

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`

Examples

Styling images

Styling the Image component is similar to styling a normal `` element, but there are a few guidelines to keep in mind:

Use `className` or `style`, not `styled-jsx`. In most cases, we recommend using the `className` prop. This can be an imported [CSS Module](#), a [global stylesheet](#), etc.

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

export default function MyImage() {
  return <Image className={styles.image} src=/>
}
```

You can also use the `style` prop to assign inline styles.

```
export default function MyImage() {
  return (
    <Image style={{ borderRadius: '8px' }} src=/>
  )
}
```

When using `fill`, the parent element must have `position: relative` or `display: block`. This is necessary for the proper rendering of the image element in that layout mode.

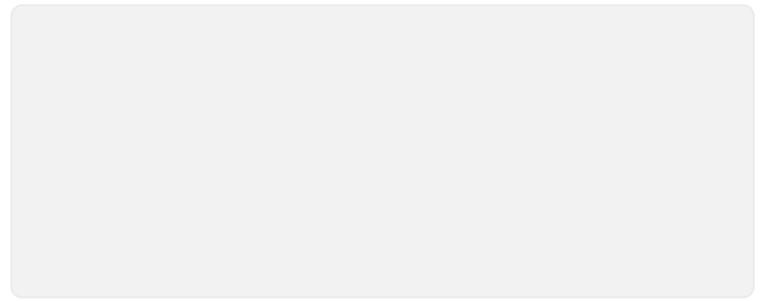
```
<div style={{ position: 'relative' }}>
  <Image fill src="/my-image.png" alt="My Image"
</div>
```

You cannot use `styled-jsx` because it's scoped to the current component (unless you mark the style

as `global`).

Responsive images with a static export

When you import a static image, Next.js automatically sets its width and height based on the file. You can make the image responsive by setting the style:



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

export default function Responsive() {
  return (
    <div style={{ display: 'flex', flexDirect
      <Image
        alt="Mountains"
        // Importing an image will
        // automatically set the width and he
        src={mountains}
        sizes="100vw"
        // Make the image display full width
        // and preserve its aspect ratio
        style={{
          width: '100%',
          height: 'auto',
        }}
      />
    </div>
  )
}
```

Responsive images with a remote URL

If the source image is a dynamic or a remote URL, you must provide the width and height props so Next.js can calculate the aspect ratio:

```
js components/page.js
```

```
import Image from 'next/image'

export default function Page({ photoUrl }) {
  return (
    <Image
      src={photoUrl}
      alt="Picture of the author"
      sizes="100vw"
      style={{
        width: '100%',
        height: 'auto',
      }}
      width={500}
      height={300}
    />
  )
}
```

Try it out:

- [Demo the image responsive to viewport ↗](#)

Responsive image with `fill`

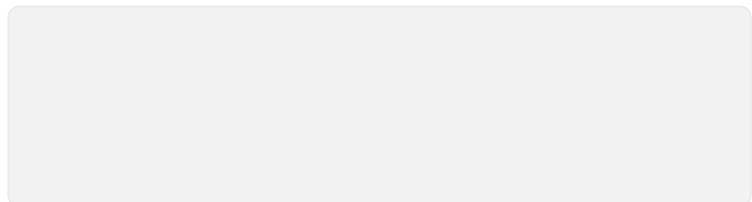
If you don't know the aspect ratio of the image, you can add the `fill` prop with the `objectFit` prop set to `cover`. This will make the image fill the full width of its parent container.

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

```
export default function Fill() {
  return (
    <div
      style={{
        display: 'grid',
        gridGap: '8px',
        gridTemplateColumns: 'repeat(auto-fit
      }}
    >
      <div style={{ position: 'relative', wid
        <Image
          alt="Mountains"
          src={mountains}
          fill
          sizes="(min-width: 808px) 50vw, 100
          style={{
            objectFit: 'cover', // cover, con
          }}
        />
      </div>
      /* And more images in the grid... */
    </div>
  )
}
```

Background Image

Use the `fill` prop to make the image cover the entire screen area:



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

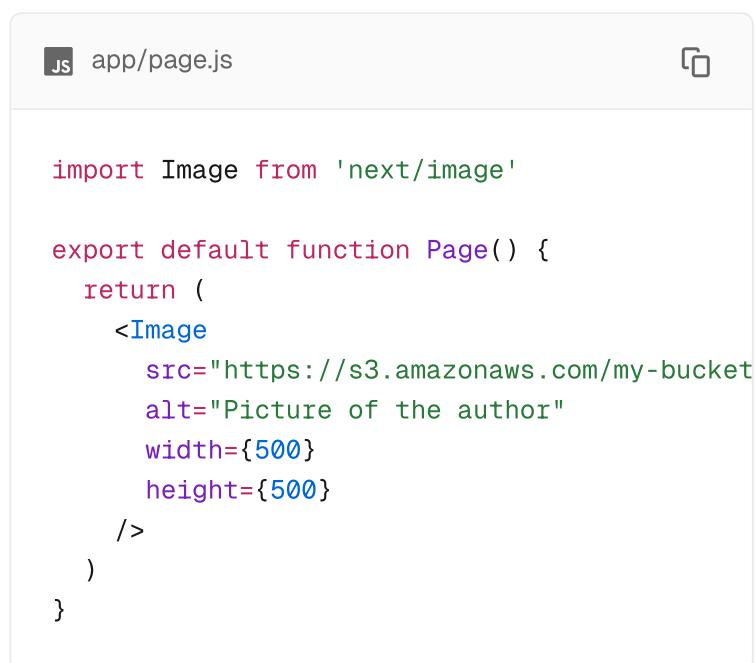
export default function Background() {
  return (
    <Image
      alt="Mountains"
      src={mountains}
      placeholder="blur"
      quality={100}
      fill
      sizes="100vw"
      style={{
        objectFit: 'cover',
      }}
    >
```

```
    }  
  />  
)  
}
```

For examples of the Image component used with the various styles, see the [Image Component Demo ↗](#).

Remote images

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



```
JS app/page.js ✖  
  
import Image from 'next/image'  
  
export default function Page() {  
  return (  
    <Image  
      src="https://s3.amazonaws.com/my-bucket  
      alt="Picture of the author"  
      width={500}  
      height={500}  
    />  
  )  
}
```

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.

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:

next.config.js

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 's3.amazonaws.com',
        port: '',
        pathname: '/my-bucket/**',
        search: ''
      }
    ]
  }
}
```

Theme detection

If you want to display a different image for light and dark mode, you can create a new component that wraps two `<Image>` components and reveals the correct one based on a CSS media query.

components/theme-image.module.css

```
.imgDark {
  display: none;
}

@media (prefers-color-scheme: dark) {
  .imgLight {
    display: none;
  }
  .imgDark {
    display: unset;
  }
}
```

components/theme-image.tsx TypeScript

```
import styles from './theme-image.module.css'
import Image, { ImageProps } from 'next/image'
```

```

type Props = Omit<ImageProps, 'src' | 'priority'>
  srcLight: string
  srcDark: string
}

const ThemeImage = (props: Props) => {
  const { srcLight, srcDark, ...rest } = props

  return (
    <>
      <Image {...rest} src={srcLight} className="light" />
      <Image {...rest} src={srcDark} className="dark" />
    </>
  )
}

```

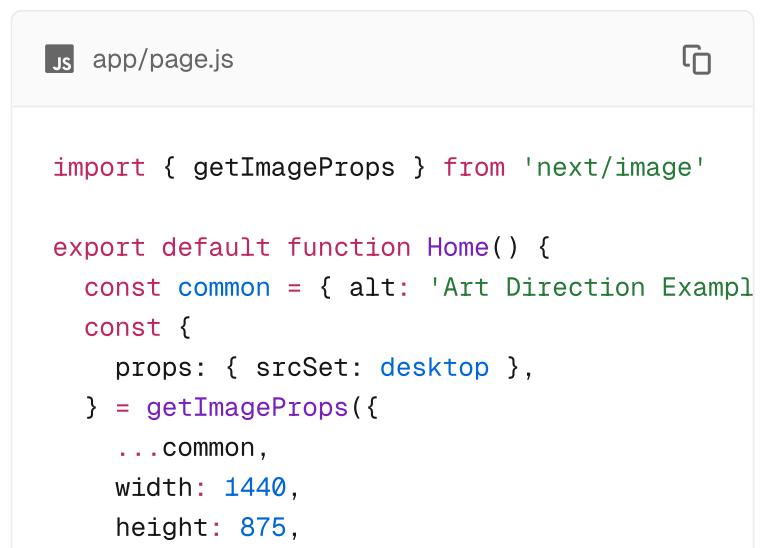
Good to know: The default behavior of `loading="lazy"` ensures that only the correct image is loaded. You cannot use `priority` or `loading="eager"` because that would cause both images to load. Instead, you can use `fetchPriority="high"` ↗.

Try it out:

- [Demo light/dark mode theme detection ↗](#)

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()`.



```

JS app/page.js

import { getImageProps } from 'next/image'

export default function Home() {
  const common = { alt: 'Art Direction Example' }
  const { props: { srcSet: desktop }, ...common, width: 1440, height: 875, } = getImageProps({
    ...
  })
}

```

```

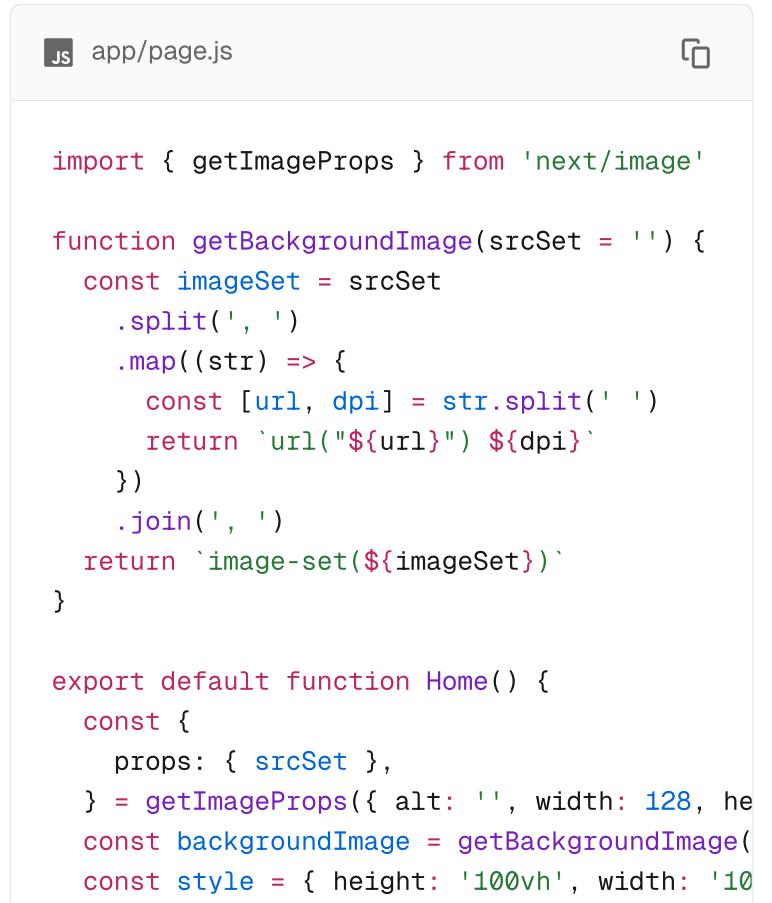
        quality: 80,
        src: '/desktop.jpg',
    })
const {
    props: { srcSet: mobile, ...rest },
} = getImageProps({
    ...common,
    width: 750,
    height: 1334,
    quality: 70,
    src: '/mobile.jpg',
})
}

return (
    <picture>
        <source media="(min-width: 1000px)" src={mobile}>
        <source media="(min-width: 500px)" src={mobile}>
        <img {...rest} style={{ width: '100%' }}>
    </picture>
)
}

```

Background CSS

You can even convert the `srcSet` string to the [image-set\(\)](#) ↗ CSS function to optimize a background image.



```

JS app/page.js

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 })
    const backgroundImage = getBackgroundImage(srcSet)
    const style = { height: '100vh', width: '100%' }
    return (
        <div style={style}>
            <img alt="Background image" style={backgroundImage} />
        </div>
    )
}

```

```
        return (
          <main style={style}>
            <h1>Hello World</h1>
          </main>
        )
      }
```

Version History

Version	Changes
v15.3.0	<code>remotePatterns</code> added support for array of <code>URL</code> objects.
v15.0.0	<code>contentDispositionType</code> configuration default changed to <code>attachment</code> .
v14.2.23	<code>qualities</code> configuration added.
v14.2.15	<code>decoding</code> prop added and <code>localPatterns</code> configuration added.
v14.2.14	<code>remotePatterns.search</code> prop added.
v14.2.0	<code>overrideSrc</code> prop added.
v14.1.0	<code>getImageProps()</code> is stable.
v14.0.0	<code>onLoadingComplete</code> prop and <code>domains</code> config deprecated.
v13.4.14	<code>placeholder</code> prop support for <code>data:/image ...</code>
v13.2.0	<code>contentDispositionType</code> configuration added.
v13.0.6	<code>ref</code> prop added.
v13.0.0	The <code>next/image</code> import was renamed to <code>next/legacy/image</code> . The <code>next/future/image</code> import was renamed to <code>next/image</code> . A codemod is available to safely and automatically rename your imports. <code></code> wrapper removed. <code>layout</code> ,

Version Changes

`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 ``.

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.

Was this helpful?

 Using App Router
Features available in /app

 Latest Version
15.5.4

Script Component

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

```
TS app/dashboard/page.tsx TypeScript ▾ ⌂  
  
import Script from 'next/script'  
  
export default function Dashboard() {  
  return (  
    <>  
      <Script src="https://example.com/script"  
      </>  
    )  
}
```

Props

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

Prop	Example	Type
<code>src</code>	<code>src="http://example.com/script"</code>	String
<code>strategy</code>	<code>strategy="lazyOnload"</code>	String

Prop	Example	Type
onLoad	onLoad={onLoadFunc}	Funct
onReady	onReady={onReadyFunc}	Funct
onError	onError={onErrorFunc}	Funct

Required Props

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

src

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

Optional Props

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

strategy

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

- `beforeInteractive` : Load before any Next.js code and before any page hydration occurs.
- `afterInteractive` : **(default)** Load early but after some hydration on the page occurs.

- `lazyOnload` : Load during browser idle time.
- `worker` : (experimental) Load in a web worker.

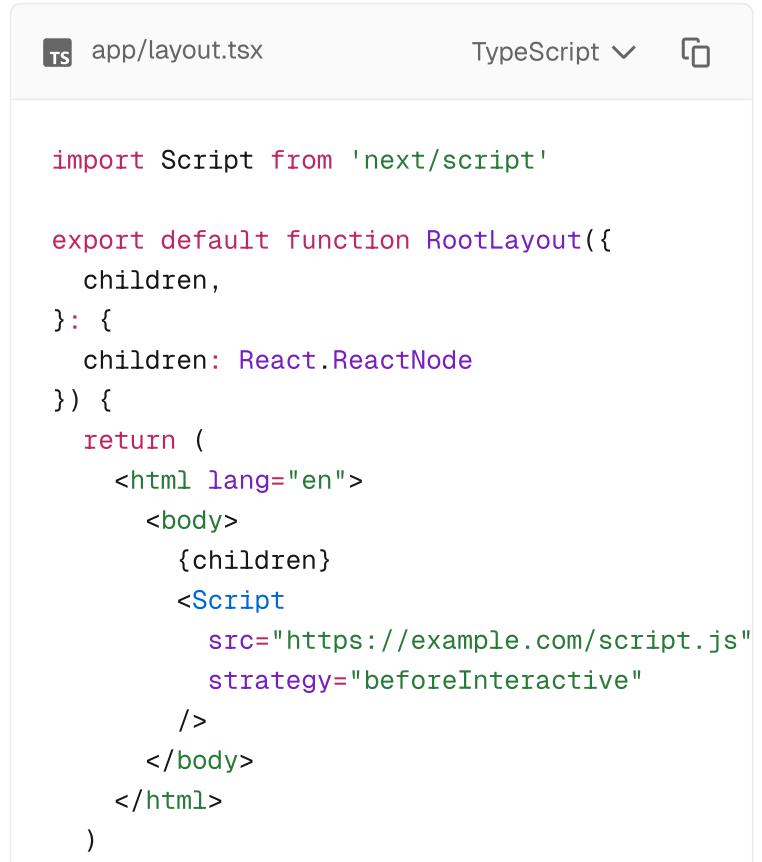
beforeInteractive

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

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

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

This strategy should only be used for critical scripts that need to be fetched as soon as possible.



```
TS app/layout.tsx TypeScript ▾ ⌂

import Script from 'next/script'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>
        {children}
        <Script
          src="https://example.com/script.js"
          strategy="beforeInteractive"
        />
      </body>
    </html>
  )
}
```

```
}
```

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

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

- Bot detectors
- Cookie consent managers

afterInteractive

Scripts that use the `afterInteractive` strategy are injected into the HTML client-side and will load after some (or all) hydration occurs on the page.

This is the default strategy of the Script component and should be used for any script that needs to load as soon as possible but not before any first-party Next.js code.

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

JS app/page.js



```
import Script from 'next/script'

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

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

- Tag managers
- Analytics

lazyOnload

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

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

```
JS app/page.js ✖  
  
import Script from 'next/script'  
  
export default function Page() {  
  return (  
    <>  
      <Script src="https://example.com/script"  
      </>  
    )  
  }  
}
```

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

- Chat support plugins
- Social media widgets

worker

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

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

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

`JS` `next.config.js`

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

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

`TS` `pages/home.tsx`

TypeScript ▾

```
import Script from 'next/script'

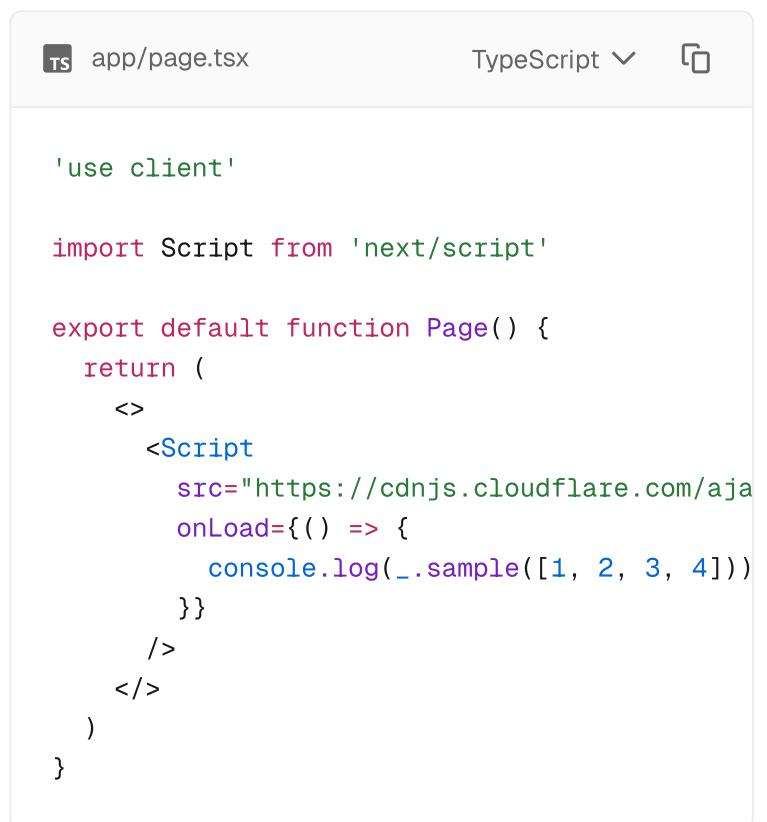
export default function Home() {
  return (
    <>
      <Script src="https://example.com/script" />
    </>
  )
}
```

`onLoad`

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

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

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



```
'use client'

import Script from 'next/script'

export default function Page() {
  return (
    <>
    <Script
      src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.min.js"
      onLoad={() =>
        console.log(_.sample([1, 2, 3, 4]))
      }
    />
    </>
  )
}
```

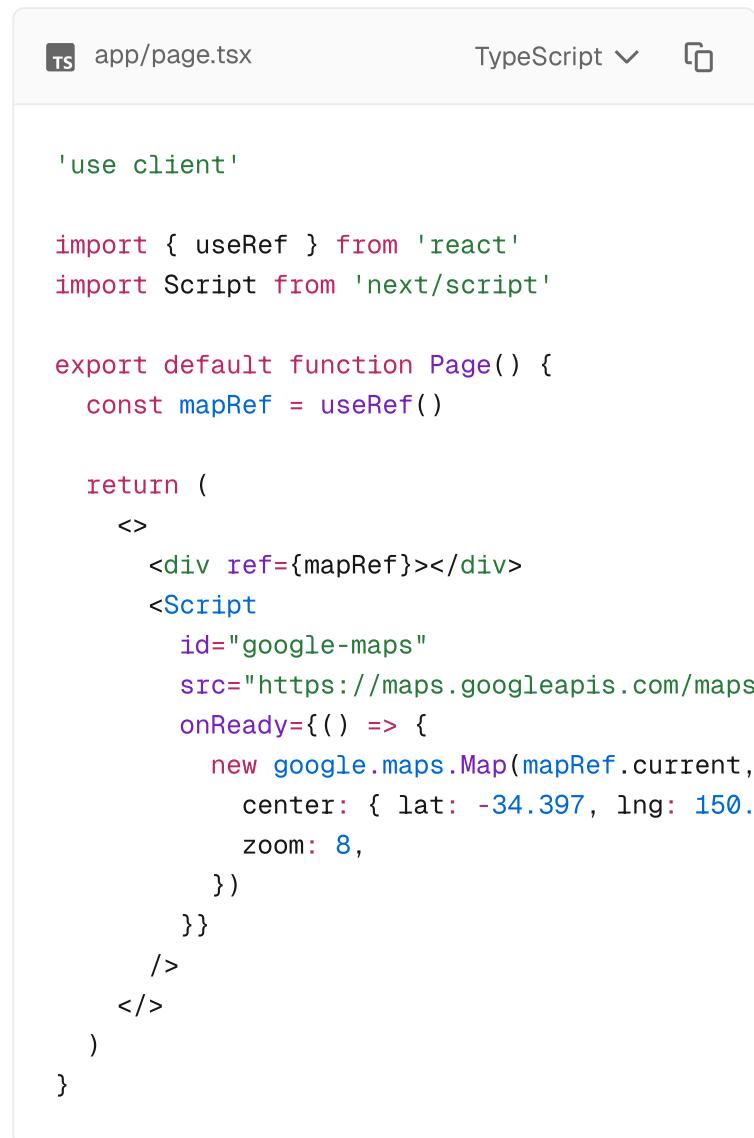
onReady

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

Some third-party scripts require users to run JavaScript code after the script has finished

loading and every time the component is mounted (after a route navigation for example). You can execute code after the script's load event when it first loads and then after every subsequent component re-mount using the `onReady` property.

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



```
'use client'

import { useRef } from 'react'
import Script from 'next/script'

export default function Page() {
  const mapRef = useRef()

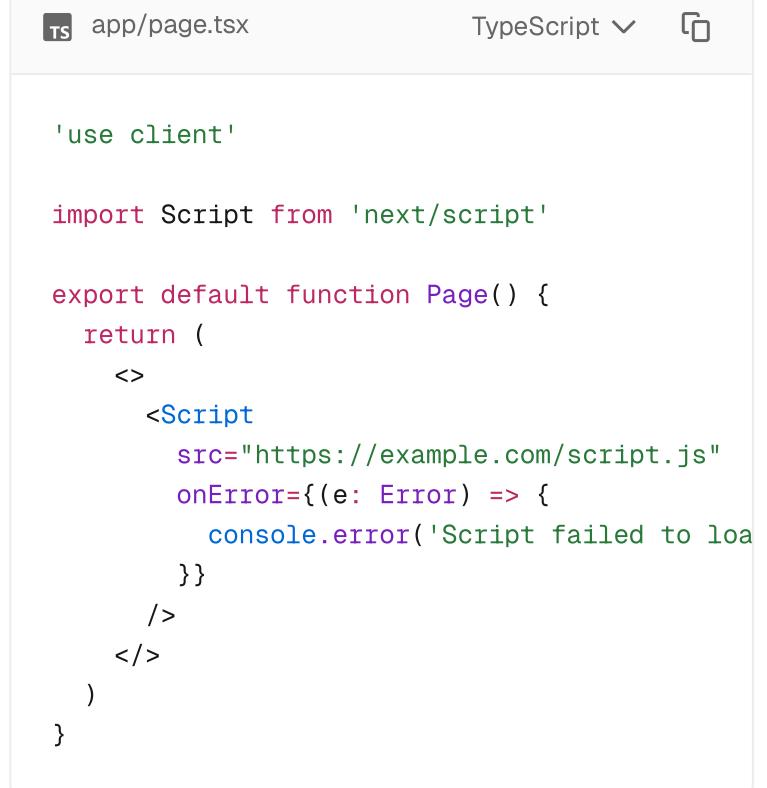
  return (
    <>
      <div ref={mapRef}></div>
      <Script
        id="google-maps"
        src="https://maps.googleapis.com/maps
        onReady={() => {
          new google.maps.Map(mapRef.current,
            center: { lat: -34.397, lng: 150.
              zoom: 8,
            })
        }}>
      />
    </>
  )
}
```

onError

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

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

onError property:



```
app/page.tsx
TypeScript ▾
```

```
'use client'

import Script from 'next/script'

export default function Page() {
  return (
    <>
    <Script
      src="https://example.com/script.js"
      onError={(e: Error) => {
        console.error('Script failed to load')
      }}
    />
    </>
  )
}
```

Version History

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

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

Was this helpful?    



Using App Router
Features available in /app



Latest Version
15.5.4



File-system conventions

default.js

API Reference for the default.js file.

Dynamic Seg...

Dynamic Route Segments can be used to...

error.js

API reference for the error.js special file.

forbidden.js

API reference for the forbidden.js special file.

instrumentat...

API reference for the instrumentation.j...

instrumentat...

Learn how to add client-side instrumentation t...

Intercepting ...

Use intercepting routes to load a new route within...

layout.js

API reference for the layout.js file.

loading.js

API reference for the loading.js file.

mdx-components.js

API reference for the mdx-components.js file.

middleware.js

API reference for the middleware.js file.

not-found.js

API reference for the not-found.js file.

page.js

API reference for the page.js file.

Parallel Routes

Simultaneously render one or more pages in the same...

public

Next.js allows you to serve static files, like images,...

route.js

API reference for the route.js special file.

Route Groups

Route Groups can be used to partition your...

Route Segments

Learn about how to configure options for Next.j...

src

template.js

Save pages under
the `src` folder as
an alternative to...

API Reference for
the template.js
file.

unauthorize...

API reference for
the
unauthorized.js...

Metadata Fil...

API documentation
for the metadata
file conventions.

Was this helpful?



 Using App Router
Features available in /app

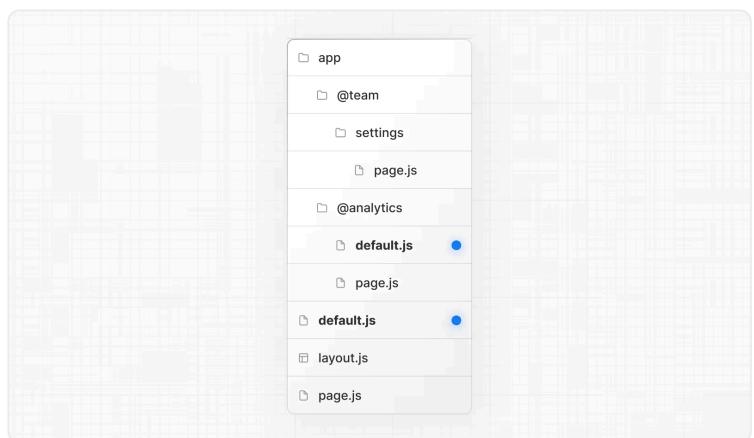
 Latest Version
15.5.4

default.js

The `default.js` file is used to render a fallback within [Parallel Routes](#) when Next.js cannot recover a `slot's` active state after a full-page load.

During [soft navigation](#), Next.js keeps track of the active *state* (subpage) for each slot. However, for hard navigations (full-page load), Next.js cannot recover the active state. In this case, a `default.js` file can be rendered for subpages that don't match the current URL.

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.

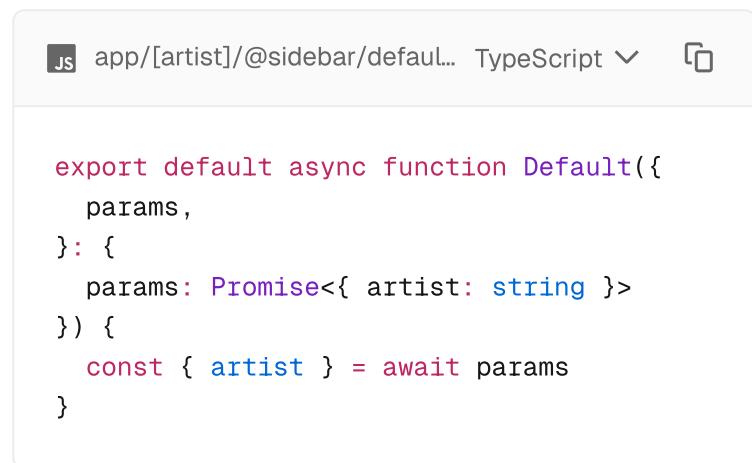
On refresh, Next.js will render a `default.js` for `@analytics`. If `default.js` doesn't exist, a `404` is rendered instead.

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.

Reference

`params` (optional)

A promise that resolves to an object containing the [dynamic route parameters](#) from the root segment down to the slot's subpages. For example:



```
JS app/[artist]/@sidebar/default... TypeScript ▾ ⌂
export default async function Default({
  params,
}: {
  params: Promise<{ artist: string }>
}) {
  const { artist } = await params
}
```

Example	URL
app/[artist]/@sidebar/default.js	/zack
app/[artist]/[album]/@sidebar/default.js	/zack/n

- 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.

Learn more about Parallel Routes

Parallel Rout...

Simultaneously render one or more pages in the sam...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

Dynamic Route Segments

When you don't know the exact route 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, a blog could include the following route `app/blog/[slug]/page.js` where `[slug]` is the Dynamic Segment for blog posts.

```
TS app/blog/[slug]/page.tsx TypeScript ▾   
  
export default async function Page({  
  params,  
}: {  
  params: Promise<{ slug: string }>  
}) {  
  const { slug } = await params  
  return <div>My Post: {slug}</div>  
}
```

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

Example

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

In Client Components

In a Client Component **page**, dynamic segments from props can be accessed using the [use](#) ↗ hook.



```
'use client'
import { use } from 'react'

export default function BlogPostPage({
  params,
}: {
  params: Promise<{ slug: string }>
}) {
  const { slug } = use(params)

  return (
    <div>
      <p>{slug}</p>
    </div>
  )
}
```

Alternatively Client Components can use the [useParams](#) hook to access the `params` anywhere in the Client Component tree.

Catch-all Segments

Dynamic Segments can be extended to **catch-all** subsequent segments by adding an ellipsis inside

the brackets `[... folderName]`.

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

Route	Example URL	params
<code>app/shop/[... slug]/page.js</code>	<code>/shop/a</code>	<code>{ slug: ['a'] }</code>
<code>app/shop/[... slug]/page.js</code>	<code>/shop/a/b</code>	<code>{ slug: ['a', 'b'] }</code>
<code>app/shop/[... slug]/page.js</code>	<code>/shop/a/b/c</code>	<code>{ slug: ['a', 'b', 'c'] }</code>

Optional Catch-all Segments

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

For example, `app/shop/[[... slug]]/page.js` will **also** match `/shop`, in addition to `/shop/clothes`, `/shop/clothes/tops`, `/shop/clothes/tops/t-shirts`.

The difference between **catch-all** and **optional catch-all** segments is that with optional, the route without the parameter is also matched (`/shop` in the example above).

Route	Example URL	params
<code>app/shop/[[... slug]]/page.js</code>	<code>/shop</code>	<code>{ slug: undefined }</code>

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'] }

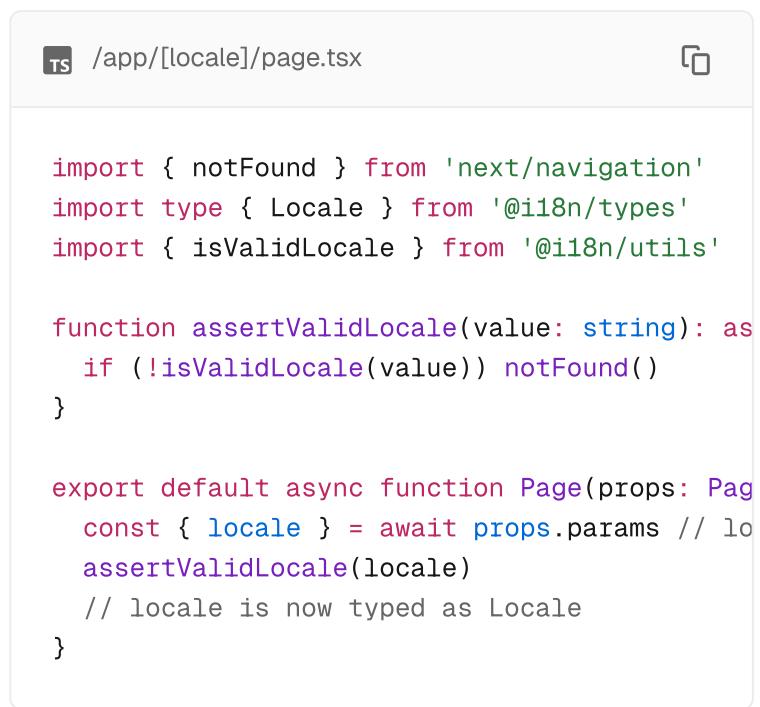
TypeScript

When using TypeScript, you can add types for `params` depending on your configured route segment — use `PageProps<'/route'>`, `LayoutProps<'/route'>`, or `RouteContext<'/route'>` to type `params` in `page`, `layout`, and `route` respectively.

Route `params` values are typed as `string`, `string[]`, or `undefined` (for optional catch-all segments), because their values aren't known until runtime. Users can enter any URL into the address bar, and these broad types help ensure that your application code handles all these possible cases.

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

If you're working on a route where `params` can only have a fixed number of valid values, such as a `[locale]` param with a known set of language codes, you can use runtime validation to handle any invalid params a user may enter, and let the rest of your application work with the narrower type from your known set.



```
TS /app/[locale]/page.tsx

import { notFound } from 'next/navigation'
import type { Locale } from '@i18n/types'
import { isValidLocale } from '@i18n/utils'

function assertValidLocale(value: string): any
  if (!isValidLocale(value)) notFound()
}

export default async function Page(props: PageProps) {
  const { locale } = await props.params // locale is now typed as Locale
  assertValidLocale(locale)
  // locale is now typed as Locale
}
```

Behavior

- 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.

Examples

With `generateStaticParams`

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

```
TS app/blog/[slug]/page.tsx TypeScript ▾ ⌂

export async function generateStaticParams()
  const posts = await fetch('https://.../post')

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

When using `fetch` inside the `generateStaticParams` function, the requests are [automatically deduplicated](#). This avoids multiple network calls for the same data Layouts, Pages, and other `generateStaticParams` functions, speeding up build time.

Next Steps

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

[generateStat...](#)

API reference for
the
`generateStaticPa...`

Was this helpful?

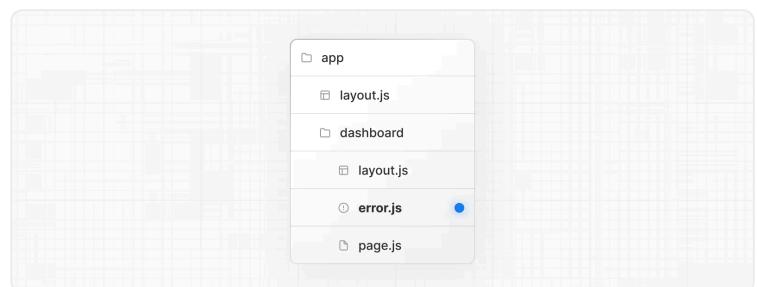


 Using App Router
Features available in /app

 Latest Version
15.5.4

error.js

An **error** file allows you to handle unexpected runtime errors and display fallback UI.



TS app/dashboard/error.tsx TypeScript ▾ ⌂

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

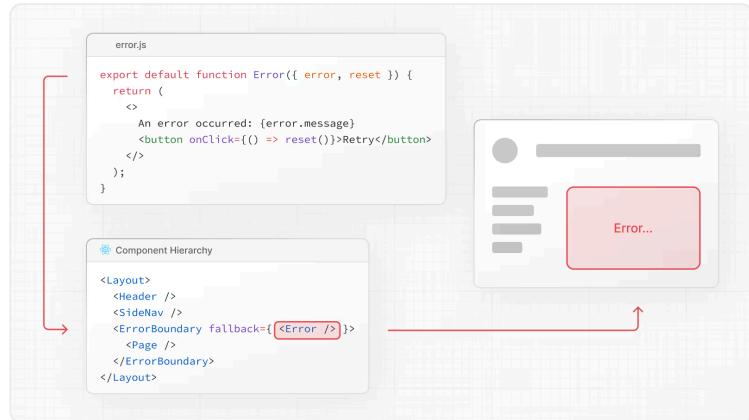
import { useEffect } from 'react'

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
        // reset the error boundary
        () => reset()
      }
    >
      Try again
    </button>
  </div>
)
```

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

`error.js` wraps a route segment and its nested children in a [React Error Boundary](#) . When an error throws within the boundary, the `error` component shows as the fallback UI.



Good to know:

- The [React DevTools](#) allow you to toggle error boundaries to test error states.
- If you want errors to bubble up to the parent error boundary, you can `throw` when rendering the `error` component.

Reference

Props

`error`

An instance of an [`Error`](#) object forwarded to the `error.js` Client Component.

Good to know: During development, the `Error` object forwarded to the client will be serialized and include the `message` of the original error for easier debugging. However, **this behavior is different in**

production to avoid leaking potentially sensitive details included in the error to the client.

error.message

- Errors forwarded from Client Components show the original `Error` message.
- Errors forwarded from Server Components show a generic message with an identifier. This is to prevent leaking sensitive details. You can use the identifier, under `errors.digest`, to match the corresponding server-side logs.

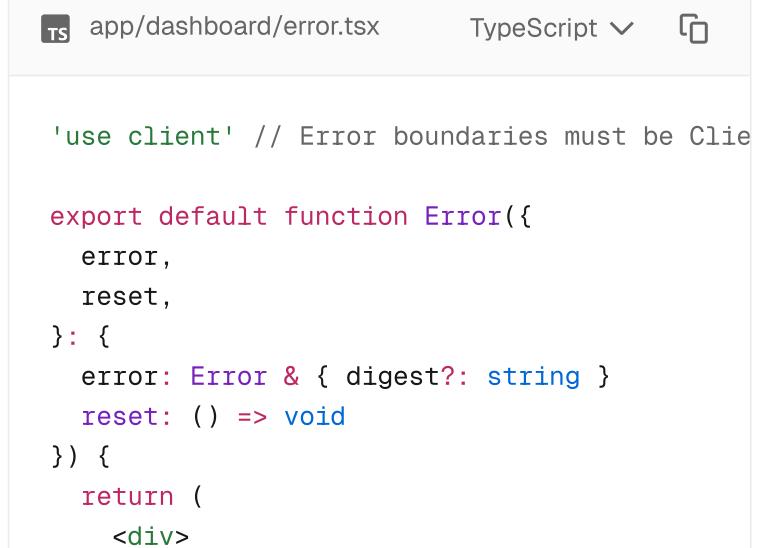
error.digest

An automatically generated hash of the error thrown. It can be used to match the corresponding error in server-side logs.

reset

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

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



The screenshot shows a code editor interface with a tab bar at the top labeled "app/dashboard/error.tsx" and "TypeScript". The code area contains the following TypeScript code:

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

export default function Error({
  error,
  reset,
}: {
  error: Error & { digest?: string }
  reset: () => void
}) {
  return (
    <div>
```

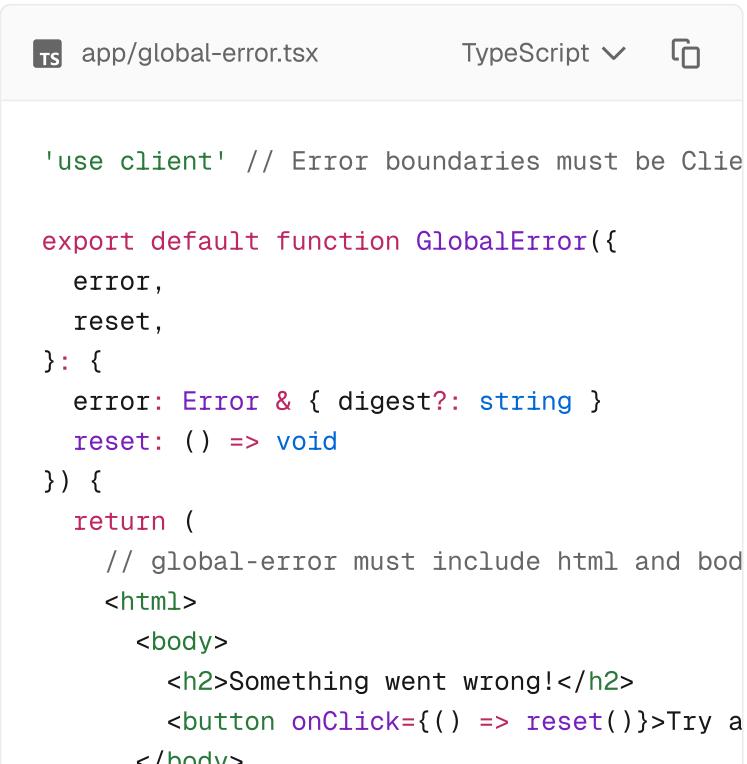
```
        <h2>Something went wrong!</h2>
        <button onClick={() => reset()}>Try again</button>
    )
}
```

Examples

Global Error

While less common, you can handle errors in the root layout or template using `global-error.jsx`, located in the root app directory, even when leveraging [internationalization](#). Global error UI must define its own `<html>` and `<body>` tags, global styles, fonts, or other dependencies that your error page requires. This file replaces the root layout or template when active.

Good to know: Error boundaries must be [Client Components](#), which means that `metadata` and `generateMetadata` exports are not supported in `global-error.jsx`. As an alternative, you can use the React `<title>` ↗ component.



The screenshot shows a code editor interface with a tab labeled "app/global-error.tsx". The code is written in TypeScript and defines a global error boundary component. The code includes a title element with the text "Something went wrong!", a button to reset the state, and a comment indicating that the component must include HTML and body tags.

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

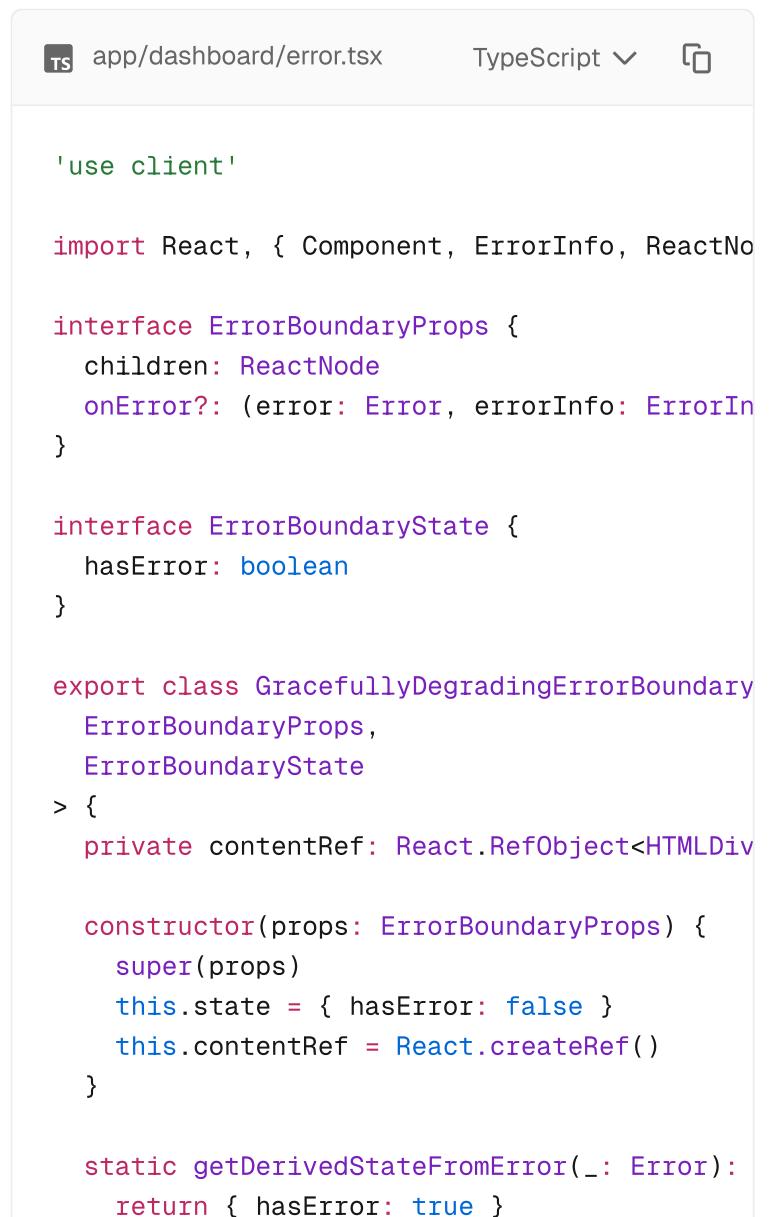
export default function GlobalError({
    error,
    reset,
}: {
    error: Error & { digest?: string }
    reset: () => void
}) {
    return (
        // global-error must include html and body
        <html>
            <body>
                <h2>Something went wrong!</h2>
                <button onClick={()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

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

Graceful error recovery with a custom error boundary

When rendering fails on the client, it can be useful to show the last known server rendered UI for a better user experience.

The `GracefullyDegradingErrorBoundary` is an example of a custom error boundary that captures and preserves the current HTML before an error occurs. If a rendering error happens, it re-renders the captured HTML and displays a persistent notification bar to inform the user.



The screenshot shows a code editor window with the following details:

- File path: app/dashboard/error.tsx
- Language: TypeScript
- Code content:

```
'use client'

import React, { Component, ErrorInfo, ReactNo
  interface ErrorBoundaryProps {
    children: ReactNode
    onError?: (error: Error, errorInfo: ErrorIn
  }

  interface ErrorBoundaryState {
    hasError: boolean
  }

  export class GracefullyDegradingErrorBoundary
    ErrorBoundaryProps,
    ErrorBoundaryState
  > {
    private contentRef: React.RefObject<HTMLDiv

    constructor(props: ErrorBoundaryProps) {
      super(props)
      this.state = { hasError: false }
      this.contentRef = React.createRef()
    }

    static getDerivedStateFromError(_: Error):
      return { hasError: true }
```

```
}
```

```
componentDidCatch(error: Error, errorThrown: string) {
  if (this.props.onError) {
    this.props.onError(error, errorThrown)
  }
}

render() {
  if (this.state.hasError) {
    // Render the current HTML content with
    return (
      <>
      <div
        ref={this.contentRef}
        suppressHydrationWarning
        dangerouslySetInnerHTML={{
          __html: this.contentRef.current
        }}
      />
      <div className="fixed bottom-0 left-0 w-full h-10 flex items-center justify-center text-white font-semibold">
        <p>An error occurred during page rendering</p>
      </div>
    </>
  )
}

return <div ref={this.contentRef}>{this.props.children}</div>
}

export default GracefullyDegradingErrorBoundary
```

Version History

Version	Changes
v15.2.0	Also display <code>global-error</code> in development.
v13.1.0	<code>global-error</code> introduced.
v13.0.0	<code>error</code> introduced.

Learn more about error handling

Error Handling

Learn how to display expected errors and handle...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

forbidden.js

 This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

The **forbidden** file is used to render UI when the **forbidden** function is invoked during authentication. Along with allowing you to customize the UI, Next.js will return a **403** status code.

TS app/forbidden.tsx TypeScript ▾

```
import Link from 'next/link'

export default function Forbidden() {
  return (
    <div>
      <h2>Forbidden</h2>
      <p>You are not authorized to access thi
      <Link href="/">Return Home</Link>
    </div>
  )
}
```

Reference

Props

forbidden.js components do not accept any props.

Version History

Version	Changes
v15.1.0	<code>forbidden.js</code> introduced.

Next Steps

forbidden

API Reference for
the `forbidden`
function.

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

instrumentation.js

The `instrumentation.js|ts` file is used to integrate observability tools into your application, allowing you to track the performance and behavior, and to debug issues in production.

To use it, place the file in the **root** of your application or inside a `src` folder if using one.

Exports

`register` (optional)

The file exports a `register` function that is called **once** when a new Next.js server instance is initiated. `register` can be an async function.

TS instrumentation.ts TypeScript

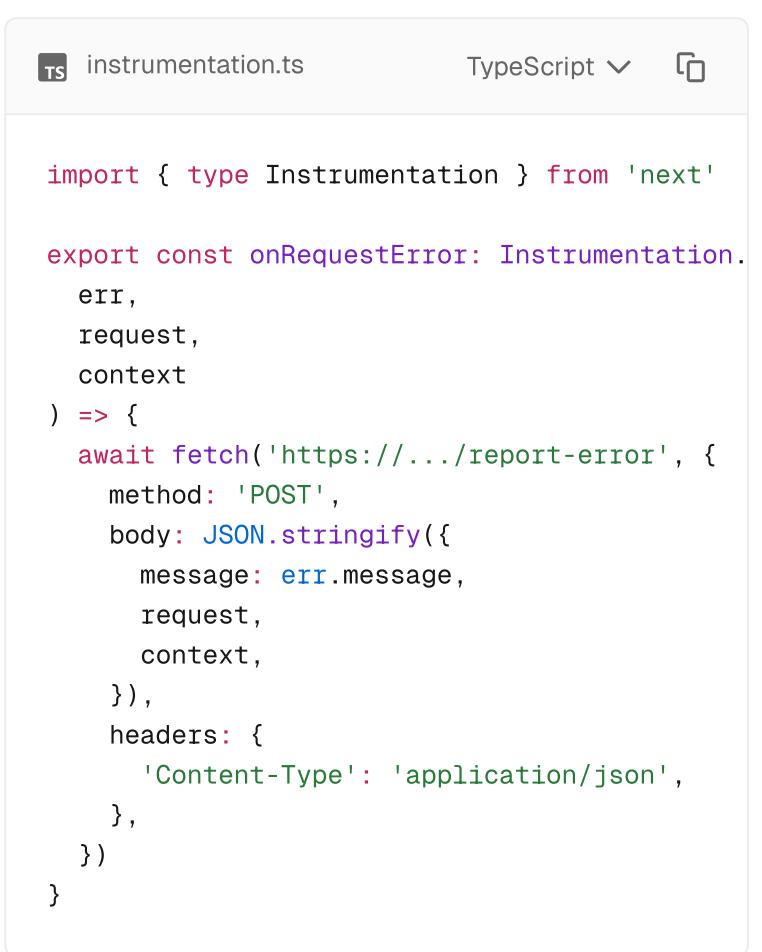
```
import { registerOTel } from '@vercel/otel'

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

`onRequestError` (optional)

You can optionally export an `onRequestError` function to track **server** errors to any custom observability provider.

- If you're running any async tasks in `onRequestError`, make sure they're awaited. `onRequestError` will be triggered when the Next.js server captures the error.
- The `error` instance might not be the original error instance thrown, as it may be processed by React if encountered during Server Components rendering. If this happens, you can use `digest` property on an error to identify the actual error type.



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

export const onRequestError: Instrumentation.  
err,  
request,  
context  
) => {  
  await fetch('https://.../report-error', {  
    method: 'POST',  
    body: JSON.stringify({  
      message: err.message,  
      request,  
      context,  
    }),  
    headers: {  
      'Content-Type': 'application/json',  
    },  
  })  
}
```

Parameters

The function accepts three parameters: `error`, `request`, and `context`.



```
export function onRequestError(  
  error: { digest: string } & Error,  
  request: {  
    path: string // resource path, e.g. /blog  
    method: string // request method. e.g. GET  
    headers: { [key: string]: string }  
  },  
  headers: { [key: string]: string }  
)
```

```
},
context: {
  routerKind: 'Pages Router' | 'App Router'
  routePath: string // the route file path,
  routeType: 'render' | 'route' | 'action'
  renderSource:
    | 'react-server-components'
    | 'react-server-components-payload'
    | 'server-rendering'
  revalidateReason: 'on-demand' | 'stale' |
  renderType: 'dynamic' | 'dynamic-resume'
}
): void | Promise<void>
```

- `error` : The caught error itself (type is always `Error`), and a `digest` property which is the unique ID of the error.
 - `request` : Read-only request information associated with the error.
 - `context` : The context in which the error occurred. This can be the type of router (App or Pages Router), and/or (Server Components (`'render'`)), Route Handlers (`'route'`), Server Actions (`'action'`), or Middleware (`'middleware'`)).

Specifying the runtime

The `instrumentation.js` file works in both the Node.js and Edge runtime, however, you can use `process.env.NEXT_RUNTIME` to target a specific runtime.

```
js instrumentation.js

export function register() {
  if (process.env.NEXT_RUNTIME === 'edge') {
    return require('./register.edge')
  } else {
    return require('./register.node')
  }
}

export function onRequestError() {
  if (process.env.NEXT_RUNTIME === 'edge') {
```

```
        return require('./on-request-error.edge')
    } else {
        return require('./on-request-error.node')
    }
}
```

Version History

Version	Changes
v15.0.0	onRequestError introduced, instrumentation stable
v14.0.4	Turbopack support for instrumentation
v13.2.0	instrumentation introduced as an experimental feature

Learn more about Instrumentation

Instrumentat...

Learn how to use
instrumentation to
run code at serve...

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

instrumentation-client.js

The `instrumentation-client.js|ts` file allows you to add monitoring, analytics code, and other side-effects that run before your application becomes interactive. This is useful for setting up performance tracking, error monitoring, polyfills, or any other client-side observability tools.

To use it, place the file in the **root** of your application or inside a `src` folder.

Usage

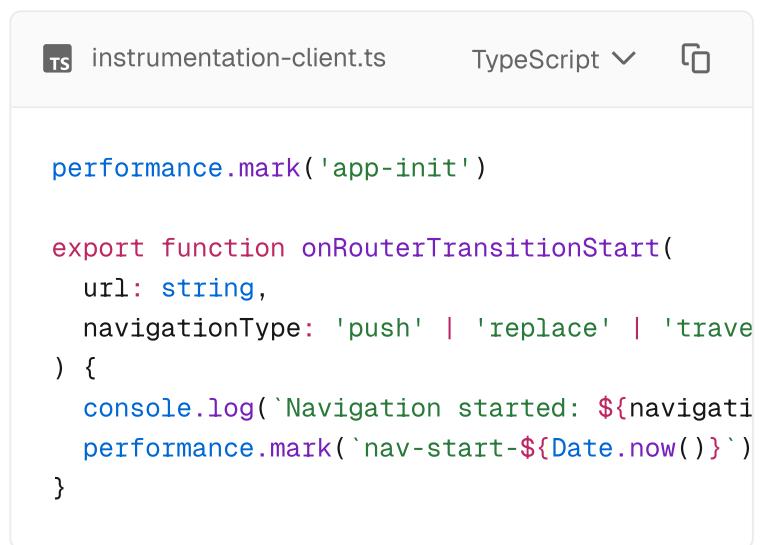
Unlike [server-side instrumentation](#), you do not need to export any specific functions. You can write your monitoring code directly in the file:

```
TS instrumentation-client.ts TypeScript ▾ ⌂  
  
// Set up performance monitoring  
performance.mark('app-init')  
  
// Initialize analytics  
console.log('Analytics initialized')  
  
// Set up error tracking  
window.addEventListener('error', (event) => {  
  // Send to your error tracking service  
  reportError(event.error)  
})
```

Error handling: Implement try-catch blocks around your instrumentation code to ensure robust monitoring. This prevents individual tracking failures from affecting other instrumentation features.

Router navigation tracking

You can export an `onRouterTransitionStart` function to receive notifications when navigation begins:



```
TS instrumentation-client.ts TypeScript ▾ ⌂

performance.mark('app-init')

export function onRouterTransitionStart(
  url: string,
  navigationType: 'push' | 'replace' | 'traverse'
) {
  console.log(`Navigation started: ${navigationType}`)
  performance.mark(`nav-start-${Date.now()}`)
}
```

The `onRouterTransitionStart` function receives two parameters:

- `url: string` - The URL being navigated to
- `navigationType: 'push' | 'replace' | 'traverse'` - The type of navigation

Performance considerations

Keep instrumentation code lightweight.

Next.js monitors initialization time in development and will log warnings if it takes longer than 16ms,

Execution timing

The `instrumentation-client.js` file executes at a specific point in the application lifecycle:

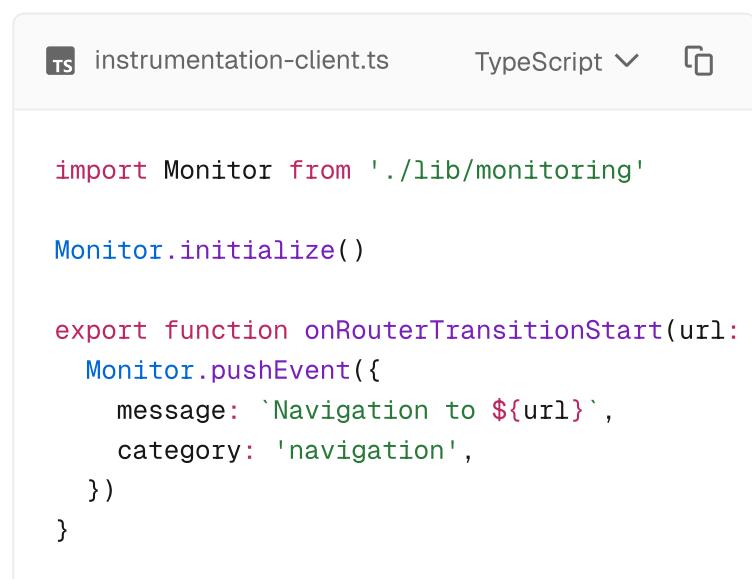
1. **After** the HTML document is loaded
2. **Before** React hydration begins
3. **Before** user interactions are possible

This timing makes it ideal for setting up error tracking, analytics, and performance monitoring that needs to capture early application lifecycle events.

Examples

Error tracking

Initialize error tracking before React starts and add navigation breadcrumbs for better debugging context.



A screenshot of a code editor showing a TypeScript file named `instrumentation-client.ts`. The code imports a `Monitor` class from a local library and initializes it. It then exports a function `onRouterTransitionStart` which pushes an event to the monitor with a message about the navigation URL and a category of 'navigation'.

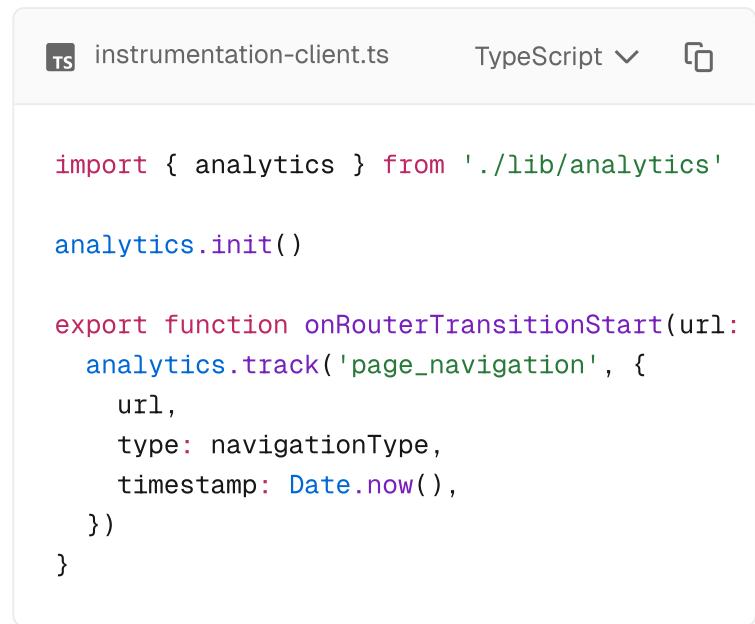
```
TS instrumentation-client.ts TypeScript ▾ ⌂
import Monitor from './lib/monitoring'

Monitor.initialize()

export function onRouterTransitionStart(url: string) {
  Monitor.pushEvent({
    message: `Navigation to ${url}`,
    category: 'navigation',
  })
}
```

Analytics tracking

Initialize analytics and track navigation events with detailed metadata for user behavior analysis.



instrumentation-client.ts TypeScript

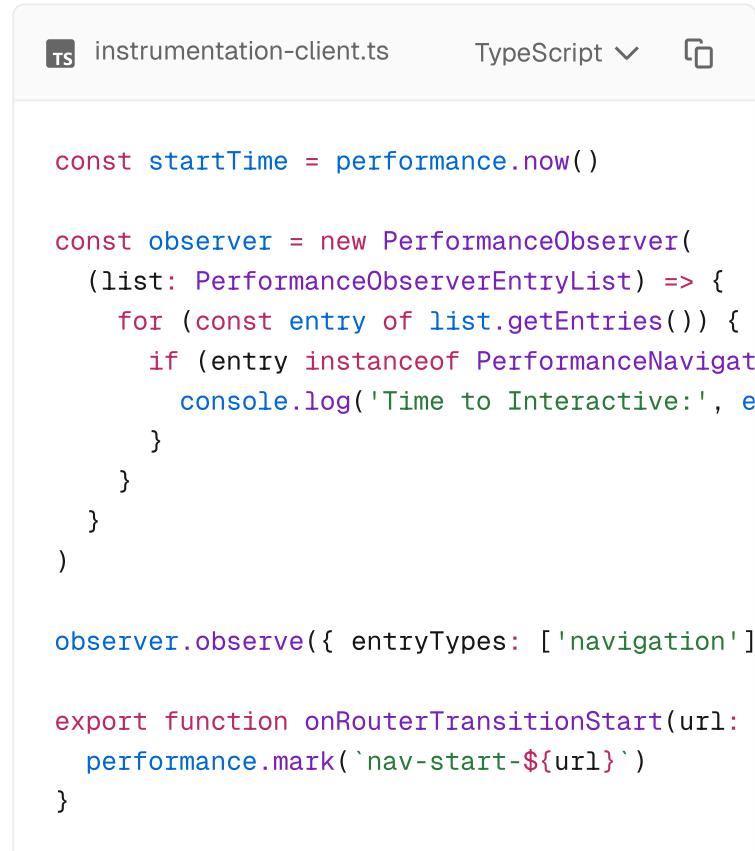
```
import { analytics } from './lib/analytics'

analytics.init()

export function onRouterTransitionStart(url: string) {
  analytics.track('page_navigation', {
    url,
    type: 'navigation',
    timestamp: Date.now(),
  })
}
```

Performance monitoring

Track Time to Interactive and navigation performance using the Performance Observer API and performance marks.



instrumentation-client.ts TypeScript

```
const startTime = performance.now()

const observer = new PerformanceObserver(
  (list: PerformanceObserverEntryList) => {
    for (const entry of list.getEntries()) {
      if (entry instanceof PerformanceNavigationEntry) {
        console.log('Time to Interactive:', entry.duration)
      }
    }
  }
)

observer.observe({ entryTypes: ['navigation'] })

export function onRouterTransitionStart(url: string) {
  performance.mark(`nav-start-${url}`)
}
```

Polyfills

Load polyfills before application code runs. Use static imports for immediate loading and dynamic imports for conditional loading based on feature detection.



```
instrumentation-client.ts  TypeScript ▾
```

```
import './lib/polyfills'

if (!window.ResizeObserver) {
  import('./lib/polyfills/resize-observer').then(mod =>
  window.ResizeObserver = mod.default
)
}
```

Version history

Version	Changes
v15.3	instrumentation-client introduced

Was this helpful?    

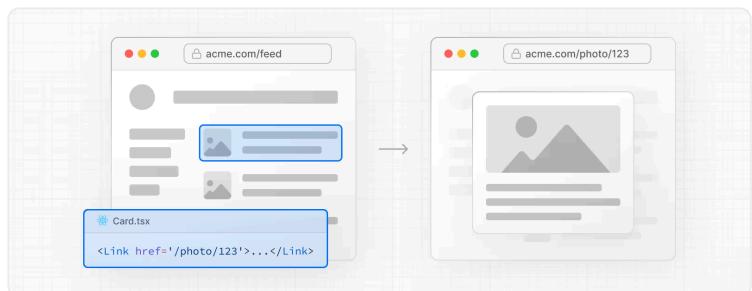
 Using App Router
Features available in /app

 Latest Version
15.5.4

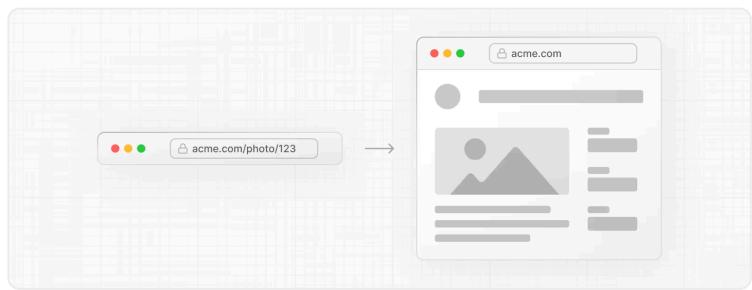
Intercepting Routes

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.



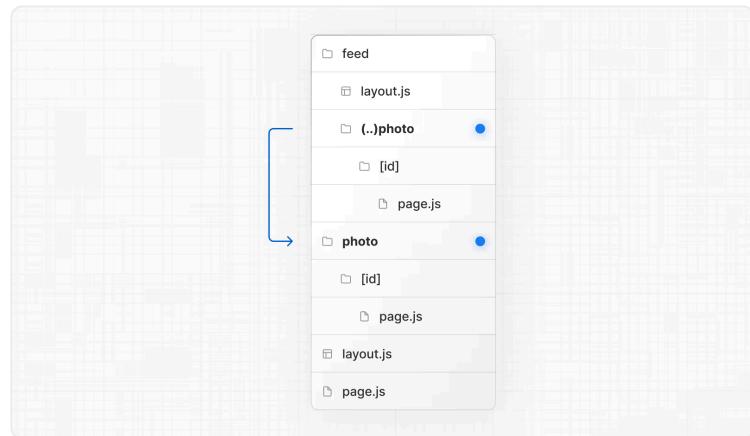
Convention

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



Good to know: The `(...)` convention is based on *route segments*, not the file-system. For example, it does not consider `@slot` folders in [Parallel Routes](#).

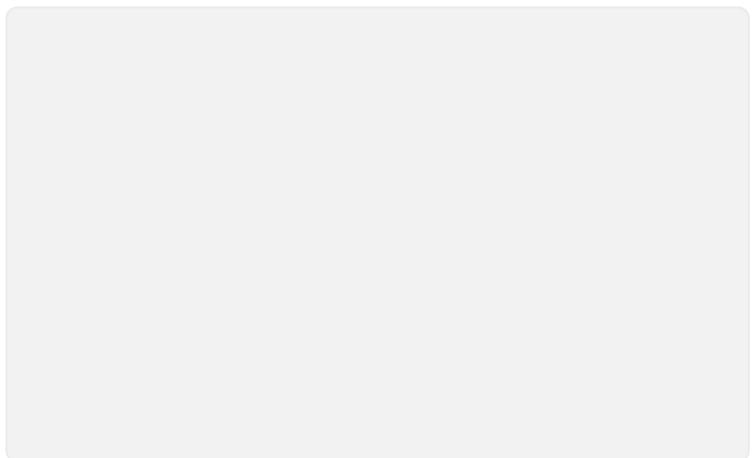
Examples

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.

Next Steps

Learn how to create modals with Intercepted and Parallel Routes.

Parallel Rout...

Simultaneously
render one or more
pages in the sam...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

layout.js

The `layout` file is used to define a layout in your Next.js application.

```
TS app/dashboard/layout.tsx TypeScript ▾ ⌂  
  
export default function DashboardLayout({  
  children,  
}: {  
  children: React.ReactNode  
}) {  
  return <section>{children}</section>  
}
```

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

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

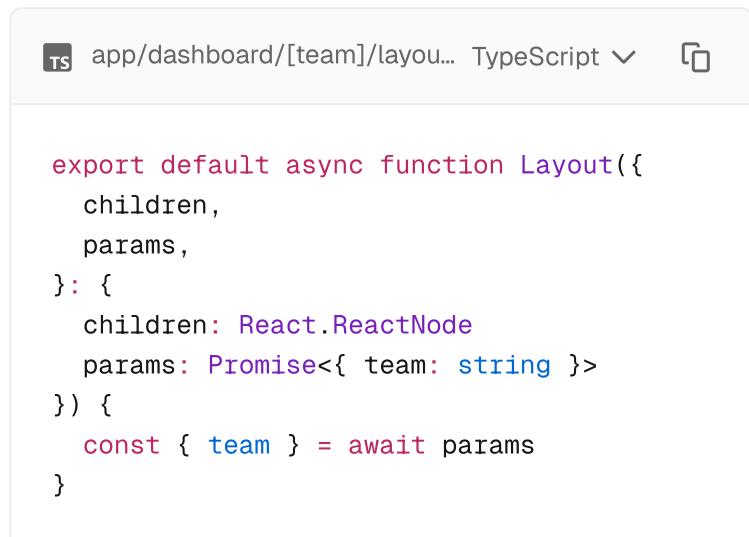
Props

children (required)

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

params (optional)

A promise that resolves to an object containing the [dynamic route parameters](#) object from the root segment down to that layout.



```
TS app/dashboard/[team]/layout.ts TypeScript ▾ ⌂
export default async function Layout({
  children,
  params,
}: {
  children: React.ReactNode
  params: Promise<{ team: string }>
}) {
  const { team } = await params
}
```

Example Route

URL

app/dashboard/[team]/layout.js	/dashboard/1
--------------------------------	--------------

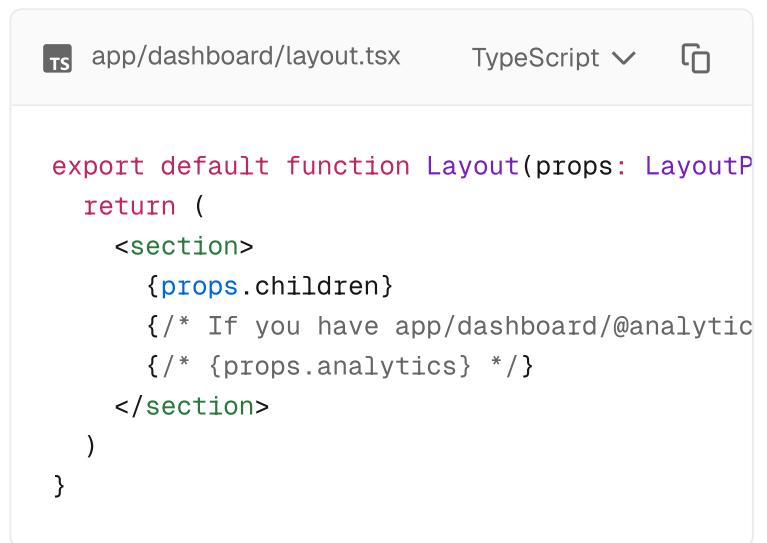
app/shop/[tag]/[item]/layout.js	/shop/1/2
---------------------------------	-----------

app/blog/[...slug]/layout.js	/blog/1/2
------------------------------	-----------

- 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.

Layout Props Helper

You can type layouts with `LayoutProps` to get a strongly typed `params` and named slots inferred from your directory structure. `LayoutProps` is a globally available helper.



```
TS app/dashboard/layout.tsx TypeScript ▾ ⌂
export default function Layout(props: LayoutProps) {
  return (
    <section>
      {props.children}
      {/* If you have app/dashboard/@analytic
       {/* {props.analytics} */}
      </section>
    )
  }
}
```

Good to know:

- Types are generated during `next dev`, `next build` or `next typegen`.

Root Layout

The `app` directory **must** include a **root layout**, which is the top-most layout in the root `app` directory. Typically, the root layout is `app/layout.js`.



```
TS app/layout.tsx TypeScript ▾ ⌂
export default function Layout(props: LayoutProps) {
  return (
    <div>
      {props.children}
      {/* If you have app/@analytic
       {/* {props.analytics} */}
      </div>
    )
  }
}
```

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

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

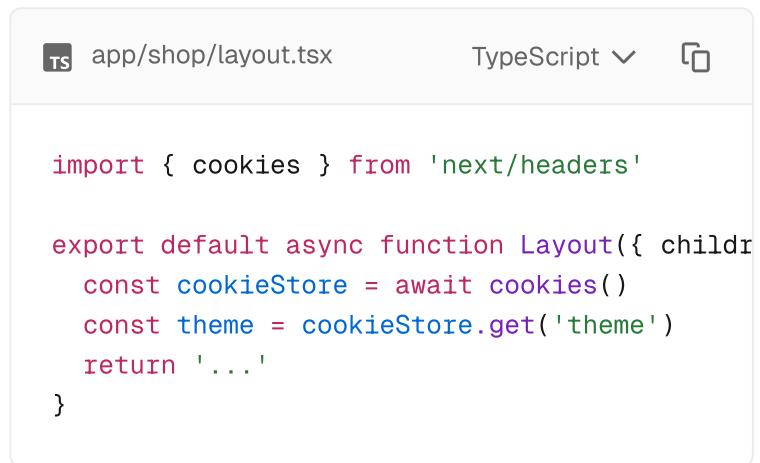
Caveats

Request Object

Layouts are cached in the client during navigation to avoid unnecessary server requests.

Layouts do not rerender. They can be cached and reused to avoid unnecessary computation when navigating between pages. By restricting layouts from accessing the raw request, Next.js can prevent the execution of potentially slow or expensive user code within the layout, which could negatively impact performance.

To access the request object, you can use [headers](#) and [cookies](#) APIs in [Server Components](#) and Functions.



```
TS app/shop/layout.tsx TypeScript ▾ ⌂

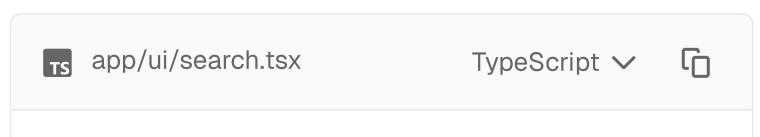
import { cookies } from 'next/headers'

export default async function Layout({ children }) {
  const cookieStore = await cookies()
  const theme = cookieStore.get('theme')
  return `...
}
```

Query params

Layouts do not rerender on navigation, so they cannot access search params which would otherwise become stale.

To access updated query parameters, you can use the Page [searchParams](#) prop, or read them inside a Client Component using the [useSearchParams](#) hook. Since Client Components re-render on navigation, they have access to the latest query parameters.



```
TS app/ui/search.tsx TypeScript ▾ ⌂

import { searchParams } from 'next/router'

function Search() {
  console.log(searchParams)
}
```

```
'use client'

import { useSearchParams } from 'next/navigation'

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

  const search = searchParams.get('search')

  return '...'
}
```

TS app/shop/layout.tsx TypeScript

```
import Search from '@/app/ui/search'

export default function Layout({ children }) {
  return (
    <>
      <Search />
      {children}
    </>
  )
}
```

Pathname

Layouts do not re-render on navigation, so they do not access pathname which would otherwise become stale.

To access the current pathname, you can read it inside a Client Component using the [usePathname](#) hook. Since Client Components re-render during navigation, they have access to the latest pathname.

TS app/ui/breadcrumbs.tsx TypeScript

```
'use client'

import { usePathname } from 'next/navigation'

// Simplified breadcrumbs logic
export default function Breadcrumbs() {
```

```
const pathname = usePathname()
const segments = pathname.split('/')

return (
  <nav>
    {segments.map((segment, index) => (
      <span key={index}>
        {'>'}
        {segment}
      </span>
    ))}
  </nav>
)
}
```

TS app/docs/layout.tsx TypeScript ▾

```
import { Breadcrumbs } from '@/app/ui/Breadcrumbs'

export default function Layout({ children }) {
  return (
    <>
      <Breadcrumbs />
      <main>{children}</main>
    </>
  )
}
```

Fetching Data

Layouts cannot pass data to their `children`.

However, you can fetch the same data in a route more than once, and use React `cache ↗` to dedupe the requests without affecting performance.

Alternatively, when using `fetch` in Next.js, requests are automatically deduped.

TS app/lib/data.ts TypeScript ▾

```
export async function getUser(id: string) {
  const res = await fetch(`https://.../users/${id}`)
  return res.json()
}
```

```

import { getUser } from '@/app/lib/data'
import { UserName } from '@/app/ui/user-name'

export default async function Layout({ children }) {
  const user = await getUser('1')

  return (
    <>
    <nav>
      {/* ... */}
      <UserName user={user.name} />
    </nav>
    {children}
    </>
  )
}

```

```

import { getUser } from '@/app/lib/data'
import { UserName } from '@/app/ui/user-name'

export default async function Page() {
  const user = await getUser('1')

  return (
    <div>
      <h1>Welcome {user.name}</h1>
    </div>
  )
}

```

Accessing child segments

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.

```
'use client'
```

```
import Link from 'next/link'
import { useSelectedLayoutSegment } from 'nex

export default function NavLink({
  slug,
  children,
}: {
  slug: string
  children: React.ReactNode
}) {
  const segment = useSelectedLayoutSegment()
  const isActive = slug === segment

  return (
    <Link
      href={`/blog/${slug}`}
      // Change style depending on whether the
      style={{ fontWeight: isActive ? 'bold' :
      >
        {children}
      </Link>
    )
  }
}
```

TS app/blog/layout.tsx TypeScript ▾

```
import { NavLink } from './nav-link'
import getPosts from './get-posts'

export default async function Layout({
  children,
}: {
  children: React.ReactNode
}) {
  const featuredPosts = await getPosts()
  return (
    <div>
      {featuredPosts.map((post) => (
        <div key={post.id}>
          <NavLink slug={post.slug}>{post.title}</NavLink>
        </div>
      ))}
      <div>{children}</div>
    </div>
  )
}
```

Examples

Metadata

You can modify the `<head>` HTML elements such as `title` and `meta` using the `metadata object` or `generateMetadata function`.

```
TS app/layout.tsx TypeScript ▾ ⌂

import type { Metadata } from 'next'

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

export default function Layout({ children }: {
  return '...'
})
```

Good to know: You should **not** manually add `<head>` tags such as `<title>` and `<meta>` to root layouts. Instead, use the [Metadata APIs](#) which automatically handles advanced requirements such as streaming and de-duplicating `<head>` elements.

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:

```
TS app/ui/nav-links.tsx TypeScript ▾ ⌂

'use client'

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

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

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

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

app/layout.tsx

TypeScript

```
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>
  )
}
```

Displaying content based on `params`

Using [dynamic route segments](#), you can display or fetch specific content based on the `params` prop.

app/dashboard/layout.tsx

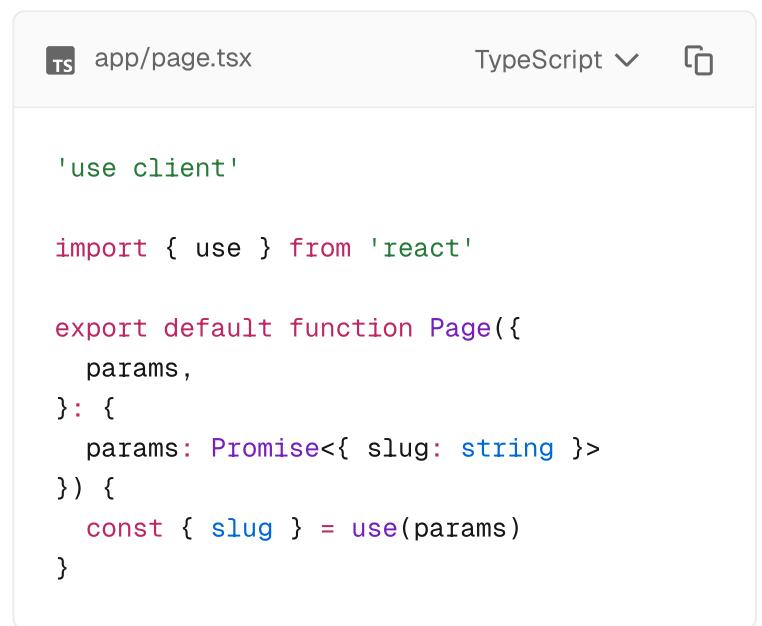
```
export default async function DashboardLayout({ children, params }: { children: React.ReactNode; params: Promise<{ team: string }> }) {
```

```
const { team } = await params

return (
  <section>
    <header>
      <h1>Welcome to {team}'s Dashboard</h1>
    </header>
    <main>{children}</main>
  </section>
)
}
```

Reading `params` in Client Components

To use `params` in a Client Component (which cannot be `async`), you can use React's `use` ↗ function to read the promise:



```
'use client'

import { use } from 'react'

export default function Page({
  params,
}: {
  params: Promise<{ slug: string }>
}) {
  const { slug } = use(params)
}
```

Version History

Version	Changes
v15.0.0-RC	<code>params</code> is now a promise. A codemod is available.
v13.0.0	<code>layout</code> introduced.

Was this helpful?  

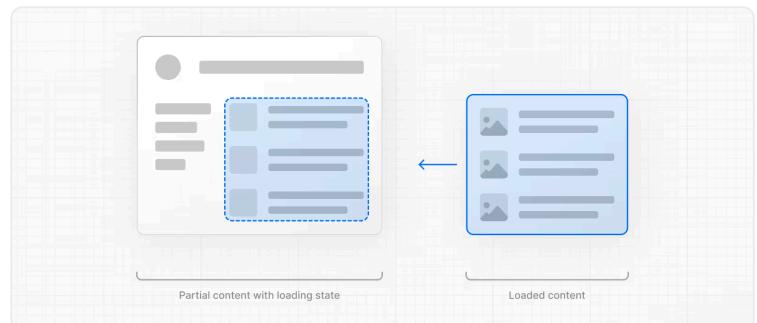


 Using App Router
Features available in /app

 Latest Version
15.5.4

loading.js

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 streams in. The new content is automatically swapped in once complete.



 app/feed/loading.tsx

TypeScript ▾



```
export default function Loading() {  
  // Or a custom loading skeleton component  
  return <p>Loading...</p>  
}
```

Inside the `loading.js` file, you can add any light-weight loading UI. You may find it helpful to use the [React Developer Tools](#) to manually toggle Suspense boundaries.

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

Reference

Parameters

Loading UI components do not accept any parameters.

Behavior

Navigation

- The Fallback UI is [prefetched](#), making navigation is immediate unless prefetching hasn't completed.
- 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.

Instant Loading States

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

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

```
TS app/dashboard/loading.tsx TypeScript ▾
```

```
export default function Loading() {
  // You can add any UI inside Loading, including
  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.

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.

Browser limits

[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.

Platform Support

Deployment Option	Supported
Node.js server	Yes
Docker container	Yes
Static export	No
Adapters	Platform-specific

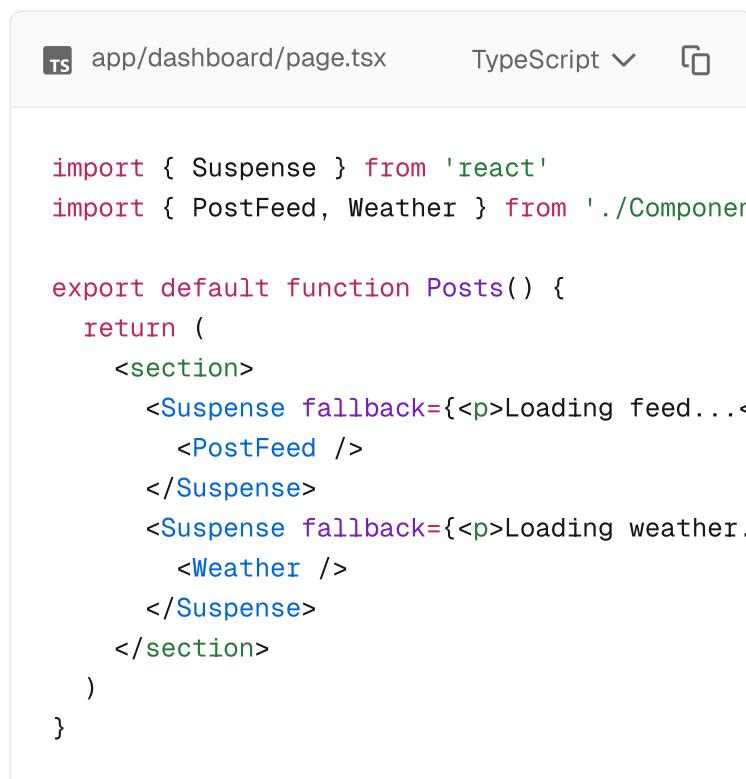
Learn how to [configure streaming](#) when self-hosting Next.js.

Examples

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 ↗](#).

`<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.



```
TS app/dashboard/page.tsx TypeScript ▾
```

```
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 ↗](#).

Version History

Version	Changes
v13.0.0	loading introduced.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

mdx-components.js

The `mdx-components.js|tsx` file is required to use `@next/mdx` with App Router and will not work without it. Additionally, you can use it to customize styles.

Use the file `mdx-components.tsx` (or `.js`) in the root of your project to define MDX Components. For example, at the same level as `pages` or `app`, or inside `src` if applicable.

```
TS  mdx-components.tsx  TypeScript ▾  ⌂  
  
import type { MDXComponents } from 'mdx/types'  
  
const components: MDXComponents = {}  
  
export function useMDXComponents(): MDXComponents  
  return components  
}
```

Exports

useMDXComponents function

The file must export a single function named `useMDXComponents`. This function does not accept any arguments.

```
TS  mdx-components.tsx  TypeScript ▾  ⌂  
  
import type { MDXComponents } from 'mdx/types'
```

```
const components: MDXComponents = {}  
  
export function useMDXComponents(): MDXCompon  
    return components  
}
```

Version History

Version	Changes
v13.1.2	MDX Components added

Learn more about MDX Components

MDX

Learn how to configure MDX and use it in your...

Was this helpful?



Using App Router

Features available in /app

Latest Version

15.5.4

middleware.js

The `middleware.js` file is used to write [Middleware](#) and run code on the server 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 executes before routes are rendered. It's particularly useful for implementing custom server-side logic like authentication, logging, or handling redirects.

Good to know:

Middleware is meant to be invoked separately of your render code and in optimized cases deployed to your CDN for fast redirect/rewrite handling, you should not attempt relying on shared modules or globals.

To pass information from Middleware to your application, use [headers](#), [cookies](#), [rewrites](#), [redirects](#), or the URL.

Create a `middleware.ts` (or `.js`) file in the project root, or inside `src` if applicable, so that it is located at the same level as `pages` or `app`.

A screenshot of a code editor interface. The title bar says "middleware.ts" with a "TS" icon. To the right, there are dropdown menus for "TypeScript" and a copy/paste icon. The main editor area contains some TypeScript code for a Next.js middleware.

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

// This function can be marked `async` if using
export function middleware(request: NextRequest) {
  return NextResponse.redirect(new URL('/home'), {
    headers: {
      'Set-Cookie': 'session=12345'
    }
  })
}

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

}

Exports

Middleware function

The file must export a single function, either as a default export or named `middleware`. Note that multiple middleware from the same file are not supported.

JS middleware.js



```
// Example of default export
export default function middleware(request) {
    // Middleware logic
}
```

Config object (optional)

Optionally, a config object can be exported alongside the Middleware function. This object includes the `matcher` to specify paths where the Middleware applies.

Matcher

The `matcher` option allows you to target specific paths for the Middleware to run on. You can specify these paths in several ways:

- For a single path: Directly use a string to define the path, like `'/about'`.
- For multiple paths: Use an array to list multiple paths, such as `matcher: ['/about', '/contact']`, which applies the Middleware to both `/about` and `/contact`.



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

Additionally, the `matcher` option supports complex path specifications through regular expressions, such as

```
matcher:
['/((?!api|_next/static|_next/image|.*\\.png$).*)']
```

, enabling precise control over which paths to include or exclude.

The `matcher` option accepts an array of objects with the following keys:

- `source`: The path or pattern used to match the request paths. It can be a string for direct path matching or a pattern for more complex matching.
- `regexp` (optional): A regular expression string that fine-tunes the matching based on the source. It provides additional control over which paths are included or excluded.
- `locale` (optional): A boolean that, when set to `false`, ignores locale-based routing in path matching.
- `has` (optional): Specifies conditions based on the presence of specific request elements such as headers, query parameters, or cookies.
- `missing` (optional): Focuses on conditions where certain request elements are absent, like missing headers or cookies.



```
export const config = {
  matcher: [
    {
      source: '/api/*',
      regexp: '^/api/(.*)',
    }
  ]
}
```

```
locale: false,  
has: [  
  { type: 'header', key: 'Authorization',  
  { type: 'query', key: 'userId', value:  
  ],  
  missing: [{ type: 'cookie', key: 'session'  
  },  
  ],  
}
```

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:

- The matcher values need to be constants so they can be statically analyzed at build-time. Dynamic values such as variables will be ignored.
- For backward compatibility, Next.js always considers /public as /public/index. Therefore, a matcher of /public/:path will match.

Params

request

When defining Middleware, the default export function accepts a single parameter, `request`. This parameter is an instance of `NextRequest`, which represents the incoming HTTP request.



```
ts middleware.ts TypeScript ▾ ⌂

import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  // Middleware logic goes here
}
```

Good to know:

- `NextRequest` is a type that represents incoming HTTP requests in Next.js Middleware, whereas `NextResponse` is a class used to manipulate and send back HTTP responses.

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](#)

Good to know: For redirects, you can also use `Response.redirect` instead of `NextResponse.redirect`.

Execution order

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`

Runtime

Middleware defaults to using the Edge runtime. As of v15.5, we have support for using the Node.js runtime. To enable, in your middleware file, set the runtime to `nodejs` in the `config` object:



The screenshot shows a code editor with a TypeScript file named `middleware.ts`. The file contains the following code:

```
export const config = {
  runtime: 'nodejs',
}
```

The code is syntax-highlighted, with `TS` indicating it's a TypeScript file. The `runtime` key is highlighted in blue, and its value `'nodejs'` is highlighted in green. The code editor interface includes tabs for "TypeScript" and a copy icon.

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.

next.config.js

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

middleware.js

```
const legacyPrefixes = ['/docs', '/blog']

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

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

  // apply trailing slash handling
  if (
    !pathname.endsWith('/') &&
    !pathname.match(/((?!\.well-known(?:\.\*)?)|index|api)/))
  ) {
    return NextResponse.redirect(
      new URL(`/${req.nextUrl.pathname}/`, req.url)
    )
  }
}
```

`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.

next.config.js

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

middleware.js

```
export default async function middleware(req) {  
  const { pathname } = req.nextUrl  
  
  // GET /_next/data/build-id/hello.json  
  
  console.log(pathname)  
  // with the flag this now /_next/data/build-i  
  // without the flag this would be normalized  
}
```

Examples

Conditional Statements

middleware.ts

TypeScript

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

Using Cookies

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

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



The screenshot shows a code editor window with a tab labeled "middleware.ts" and a status bar indicating "TypeScript". The code itself is a middleware function for Next.js:

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

export function middleware(request: NextRequest) {
    // Assume a "Cookie:nextjs=fast" header to be
    // Getting cookies from the request using the
    let cookie = request.cookies.get('nextjs')
    console.log(cookie) // => { name: 'nextjs', v
    const allCookies = request.cookies.getAll()
    console.log(allCookies) // => [{ name: 'nextj

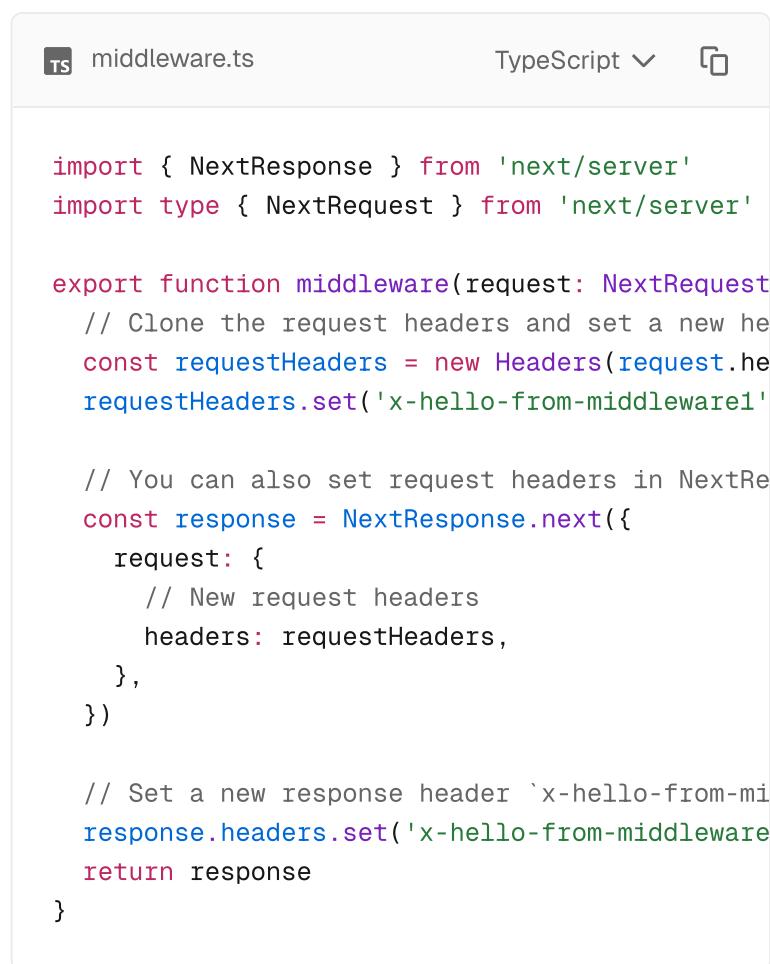
    request.cookies.has('nextjs') // => true
    request.cookies.delete('nextjs')
    request.cookies.has('nextjs') // => false

    // Setting cookies on the response using the
    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', v
    // The outgoing response will have a `Set-Coo

    return response
}
```

Setting Headers

You can set request and response headers using the `NextResponse` API (setting `request` headers is available since Next.js v13.0.0).



```
TS middleware.ts TypeScript ▾ ⌂

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
  const requestHeaders = new Headers(request.headers)
  requestHeaders.set('x-hello-from-middleware1', 'true')

  // You can also set request headers in NextResponse
  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', 'true')
  return response
}
```

Note that the snippet uses:

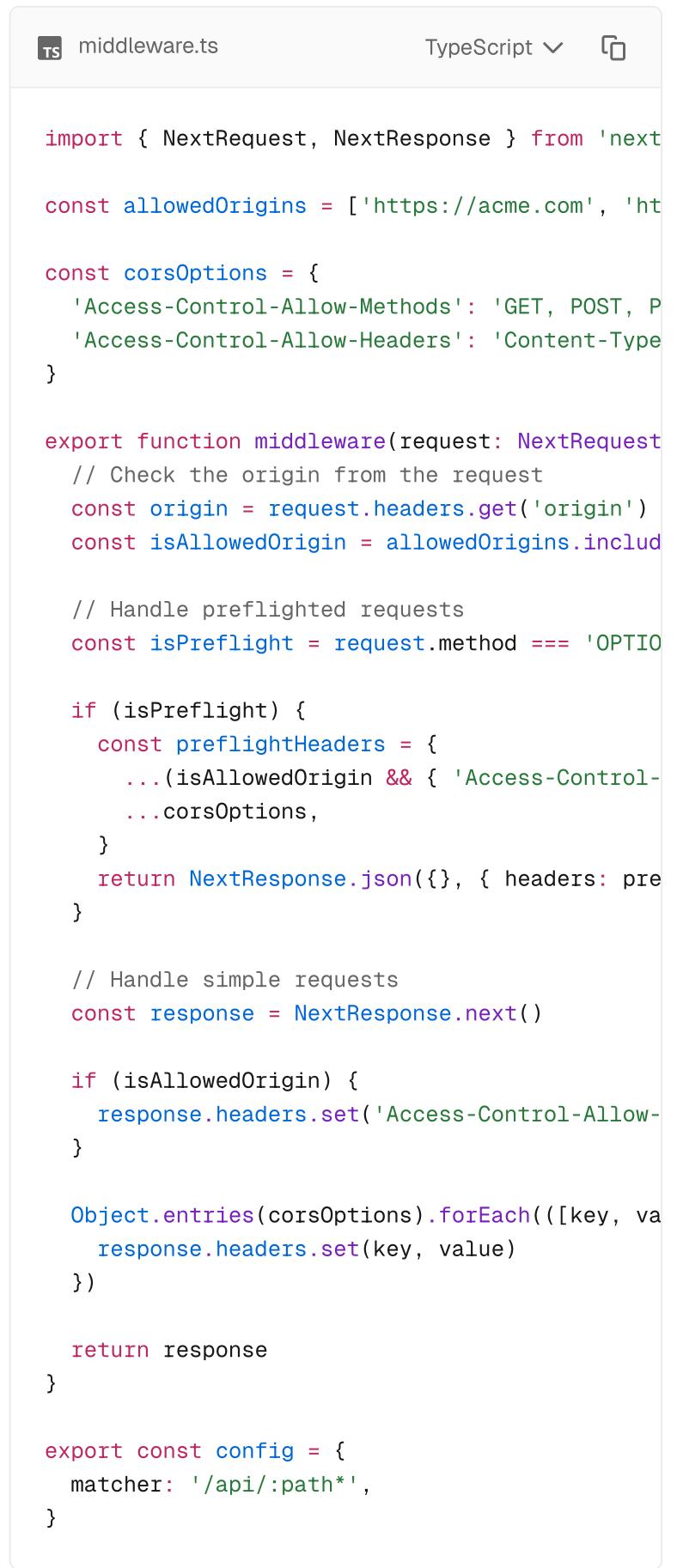
- `NextResponse.next({ request: { headers: requestHeaders } })` to make `requestHeaders` available upstream
- **NOT** `NextResponse.next({ headers: requestHeaders })` which makes `requestHeaders` available to clients

Learn more in [NextResponse headers in Middleware](#).

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.



```
TS middleware.ts TypeScript ▾
```

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

const allowedOrigins = ['https://acme.com', 'ht']

const corsOptions = {
  'Access-Control-Allow-Methods': 'GET, POST, P
  'Access-Control-Allow-Headers': 'Content-Type
}

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

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

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

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

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

  Object.entries(corsOptions).forEach(([key, va
    response.headers.set(key, value)
  })

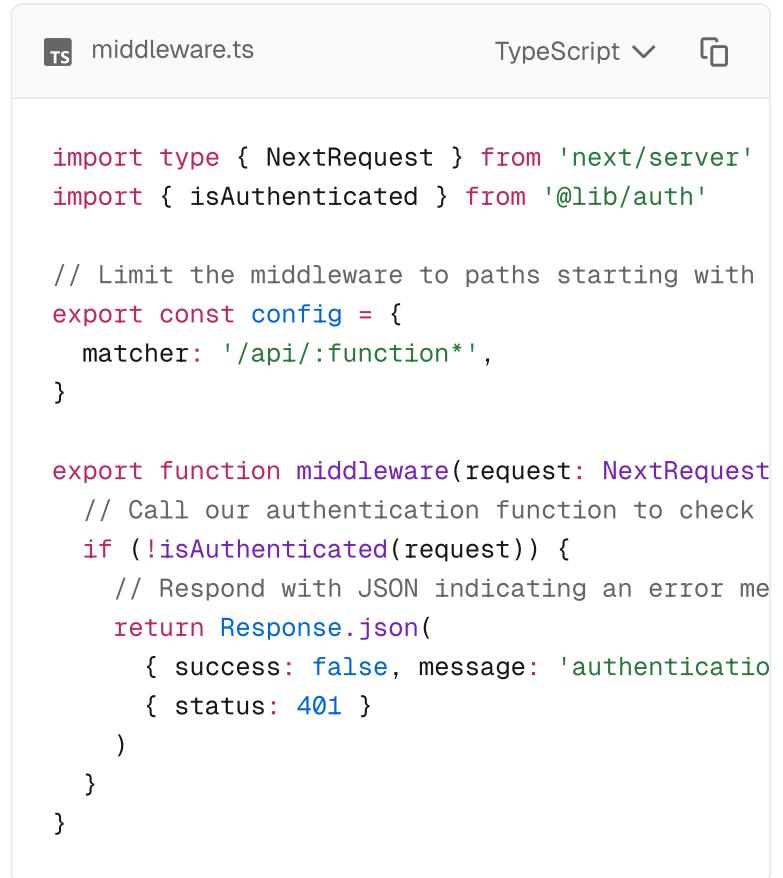
  return response
}

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 ↗](#))



The screenshot shows a code editor window with a TypeScript file named `middleware.ts`. The file contains the following code:

```
import type { NextRequest } from 'next/server'
import { isAuthenticated } from '@lib/auth'

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

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

Negative matching

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:



The screenshot shows a code editor window with a JavaScript file named `middleware.js`. The file contains the following code:

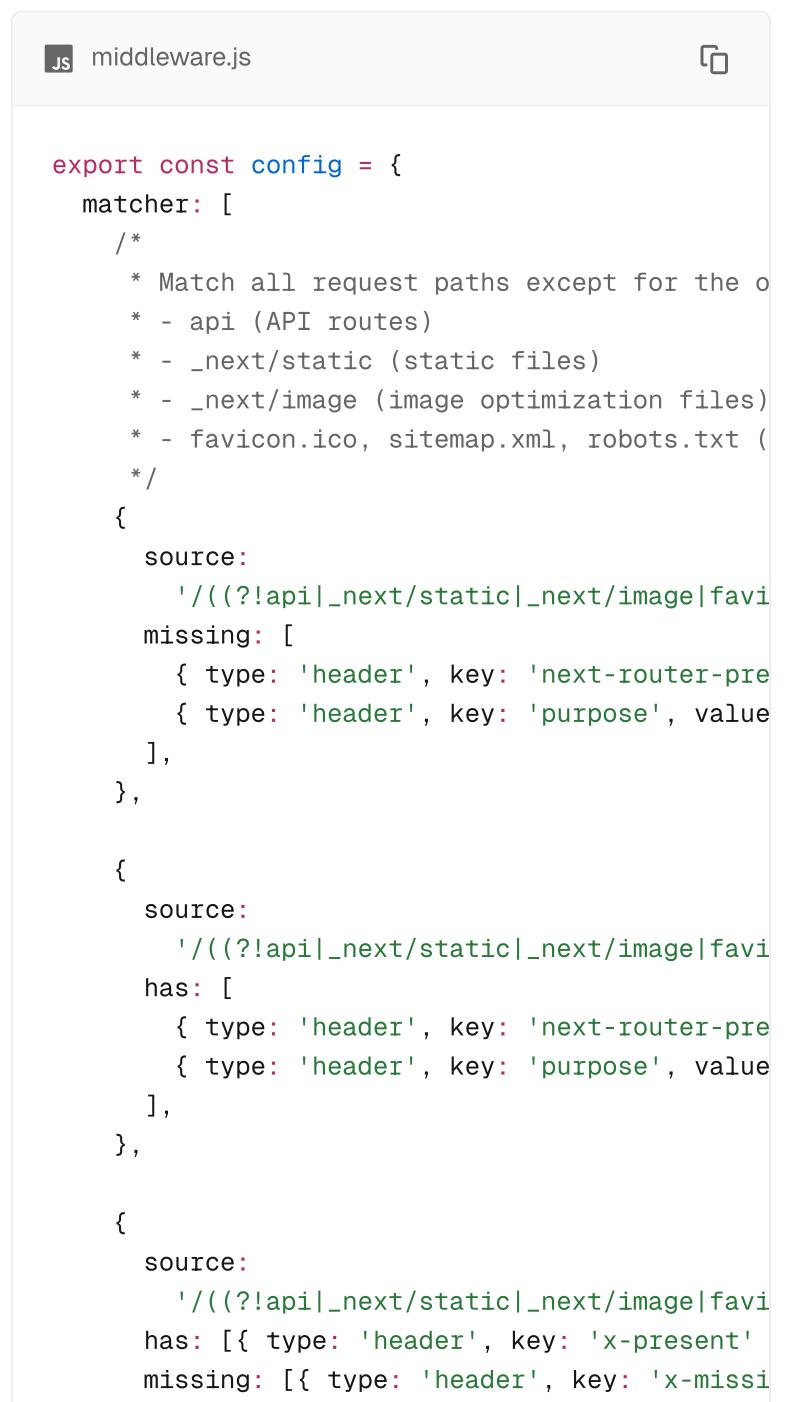
```
import type { NextRequest } from 'next/server'
import { isAuthenticated } from '@lib/auth'

// Limit the middleware to paths starting with
export const config = {
  matcher: '/api/(?!function)/.*',
}

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

```
export const config = {
  matcher: [
    /*
     * Match all request paths except for the ones
     * - api (API routes)
     * - _next/static (static files)
     * - _next/image (image optimization files)
     * - favicon.ico, sitemap.xml, robots.txt (etc)
     */
    '/((?!api|_next/static|_next/image|favicon.ico|robots.txt).*)',
  ],
}
```

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



The screenshot shows a code editor window with a tab labeled "middleware.js". The code within the editor is identical to the one shown above, but it includes additional sections for "missing" and "has" arrays under the "source" key.

```
export const config = {
  matcher: [
    /*
     * Match all request paths except for the ones
     * - api (API routes)
     * - _next/static (static files)
     * - _next/image (image optimization files)
     * - favicon.ico, sitemap.xml, robots.txt (etc)
     */
    '/((?!api|_next/static|_next/image|favicon.ico|robots.txt).*)',
  ],
  source: {
    missing: [
      { type: 'header', key: 'next-router-preload' },
      { type: 'header', key: 'purpose', value: 'navigation' },
    ],
  },
  source: {
    has: [
      { type: 'header', key: 'next-router-preload' },
      { type: 'header', key: 'purpose', value: 'navigation' },
    ],
  },
  source: {
    has: [{ type: 'header', key: 'x-present' }],
    missing: [{ type: 'header', key: 'x-missing' }],
  },
}
```

```
    },  
    ],  
}
```

waitFor and NextFetchEvent

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

The `waitFor()` 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.



A screenshot of a code editor showing a file named `middleware.ts`. The code defines a middleware function that uses `waitFor()` to wait for a fetch request to complete before proceeding.

```
ts middleware.ts □  
  
import { NextResponse } from 'next/server'  
import type { NextFetchEvent, NextRequest } from 'next/headers'  
  
export function middleware(req: NextRequest, event: NextFetchEvent) {  
  event.waitFor()  
  fetch('https://my-analytics-platform.com', {  
    method: 'POST',  
    body: JSON.stringify({ pathname: req.nextUrl.pathname })  
  })  
  
  return NextResponse.next()  
}
```

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'

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

The entire middleware function can also be tested.

```
import { isRewrite, getRewrittenUrl } from 'next'

const request = new NextRequest('https://nextjs.org')
const response = await middleware(request)
expect(isRewrite(response)).toEqual(true)
expect(getRewrittenUrl(response)).toEqual('http://nextjs.org')
// getRedirectUrl could also be used if the res
```

Platform support

Deployment Option	Supported
Node.js server	Yes
Docker container	Yes
Static export	No
Adapters	Platform-specific

Learn how to [configure Middleware](#) when self-hosting Next.js.

Version history

Version Changes

v15.5.0 Middleware can now use the Node.js runtime (stable)

v15.2.0 Middleware can now use the Node.js runtime (experimental)

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

Learn more about Middleware

NextRequest

API Reference for NextRequest.

NextResponse

API Reference for NextResponse.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

not-found.js

Next.js provides two conventions to handle not found cases:

- `not-found.js` : Used when you call the `notFound` function in a route segment.
- `global-not-found.js` : Used to define a global 404 page for unmatched routes across your entire app. This is handled at the routing level and doesn't depend on rendering a layout or page.

not-found.js

The `not-found` file is used to render UI when the `notFound` function is thrown within a route segment. Along with serving a custom UI, Next.js will return a `200` HTTP status code for streamed responses, and `404` for non-streamed responses.

TS app/not-found.tsx TypeScript ▾

```
function notFound() {  
  return {  
    props: {},  
    children: "Not Found",  
  };  
}
```

```
import Link from 'next/link'

export default function NotFound() {
  return (
    <div>
      <h2>Not Found</h2>
      <p>Could not find requested resource</p>
      <Link href="/">Return Home</Link>
    </div>
  )
}
```

global-not-found.js (experimental)

The `global-not-found.js` file lets you define a 404 page for your entire application. Unlike `not-found.js`, which works at the route level, this is used when a requested URL doesn't match any route at all. Next.js **skips rendering** and directly returns this global page.

The `global-not-found.js` file bypasses your app's normal rendering, which means you'll need to import any global styles, fonts, or other dependencies that your 404 page requires.

Good to know: A smaller version of your global styles, and a simpler font family could improve performance of this page.

`global-not-found.js` is useful when you can't build a 404 page using a combination of `layout.js` and `not-found.js`. This can happen in two cases:

- Your app has multiple root layouts (e.g. `app/(admin)/layout.tsx` and

`app/(shop)/layout.tsx`), so there's no single layout to compose a global 404 from.

- Your root layout is defined using top-level dynamic segments (e.g. `app/[country]/layout.tsx`), which makes composing a consistent 404 page harder.

To enable it, add the `globalNotFound` flag in `next.config.ts`:

```
TS next.config.ts

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    globalNotFound: true,
  },
}

export default nextConfig
```

Then, create a file in the root of the `app` directory: `app/global-not-found.js`:

```
TS app/global-not-found.tsx TypeScript ✓

// Import global styles and fonts
import './globals.css'
import { Inter } from 'next/font/google'
import type { Metadata } from 'next'

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

export const metadata: Metadata = {
  title: '404 - Page Not Found',
  description: 'The page you are looking for'
}

export default function GlobalNotFound() {
  return (
    <html lang="en" className={inter.className}>
      <body>
        <h1>404 - Page Not Found</h1>
        <p>This page does not exist.</p>
      </body>
    </html>
  )
}
```

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

Unlike `not-found.js`, this file must return a full HTML document, including `<html>` and `<body>` tags.

Reference

Props

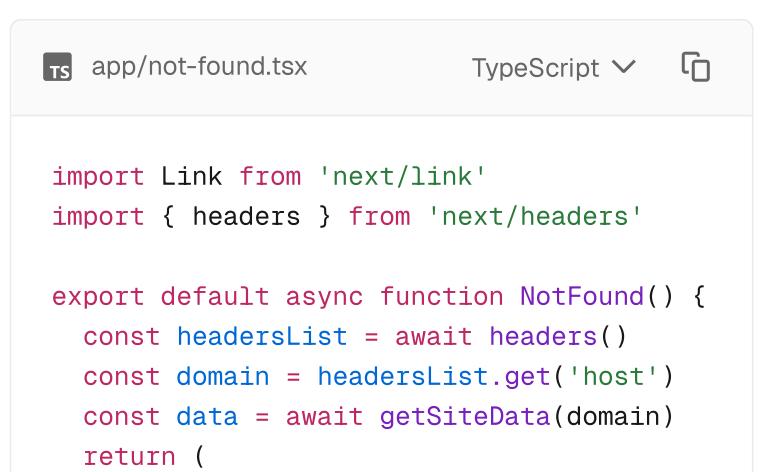
`not-found.js` or `global-not-found.js` components do not accept any props.

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

Examples

Data Fetching

By default, `not-found` is a Server Component. You can mark it as `async` to fetch and display data:



app/not-found.tsx TypeScript ▾

```
import Link from 'next/link'
import { headers } from 'next/headers'

export default async function NotFound() {
  const headersList = await headers()
  const domain = headersList.get('host')
  const data = await getSiteData(domain)
  return (
    <div>
      <h1>404</h1>
      <p>Not found</p>
      <Link href="/">Go back</Link>
    </div>
  )
}
```

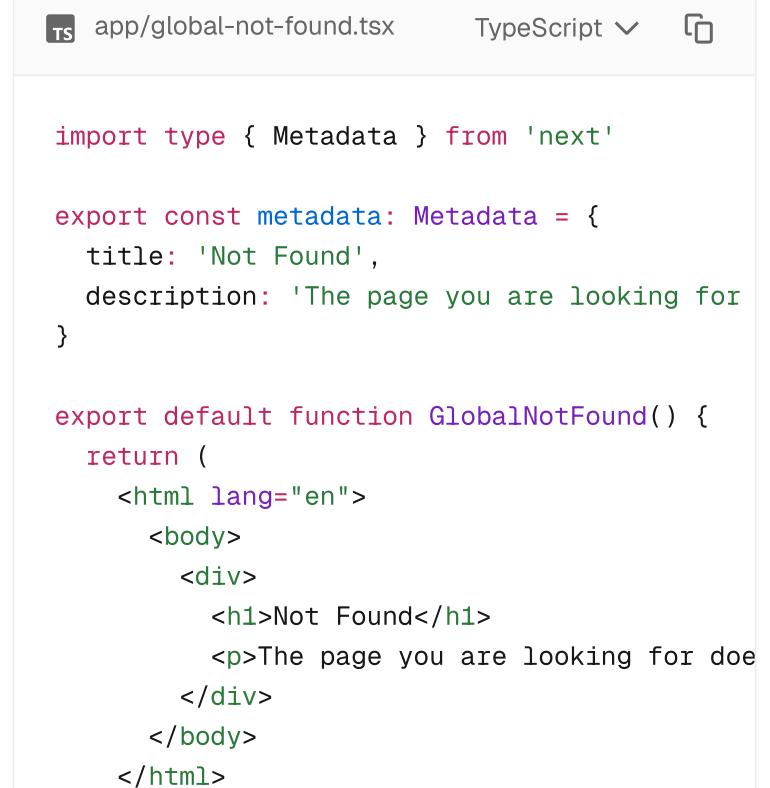
```
<div>
  <h2>Not Found: {data.name}</h2>
  <p>Could not find requested resource</p>
  <p>
    View <Link href="/blog">all posts</Link>
  </p>
</div>
)
}
```

If you need to use Client Component hooks like `usePathname` to display content based on the path, you must fetch data on the client-side instead.

Metadata

For `global-not-found.js`, you can export a `metadata` object or a `generateMetadata` function to customize the `<title>`, `<meta>`, and other head tags for your 404 page:

Good to know: Next.js automatically injects `<meta name="robots" content="noindex" />` for pages that return a 404 status code, including `global-not-found.js` pages.



The screenshot shows a code editor window with the file `app/global-not-found.tsx`. The code defines a `Metadata` object and a `GlobalNotFound` component. The `Metadata` object contains `title` and `description` fields. The `GlobalNotFound` component returns an `html` element with `body` and `div` children, including an `h1` and a `p` element.

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

export const metadata: Metadata = {
  title: 'Not Found',
  description: 'The page you are looking for'
}

export default function GlobalNotFound() {
  return (
    <html lang="en">
      <body>
        <div>
          <h1>Not Found</h1>
          <p>The page you are looking for does not exist.</p>
        </div>
      </body>
    </html>
  )
}
```

```
)  
}
```

Version History

Version	Changes
v15.4.0	global-not-found.js introduced (experimental).
v13.3.0	Root app/not-found handles global unmatched URLs.
v13.0.0	not-found introduced.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

page.js

The `page` file allows you to define UI that is **unique** to a route. You can create a page by default exporting a component from the file:

```
TS app/blog/[slug]/page.tsx TypeScript ▾ ⌂  
  
export default function Page({  
  params,  
  searchParams,  
}: {  
  params: Promise<{ slug: string }>  
  searchParams: Promise<{ [key: string]: string }>  
) {  
  return <h1>My Page</h1>  
}
```

Good to know

- The `.js`, `.jsx`, or `.tsx` file extensions can be used for `page`.
- A `page` is always the **leaf** of the route subtree.
- A `page` file is required to make a route segment **publicly accessible**.
- Pages are [Server Components ↗](#) by default, but can be set to a [Client Component ↗](#).

Reference

Props

params (optional)

A promise that resolves to an object containing the [dynamic route parameters](#) from the root segment down to that page.



```
TS app/shop/[slug]/page.tsx TypeScript ▾
```

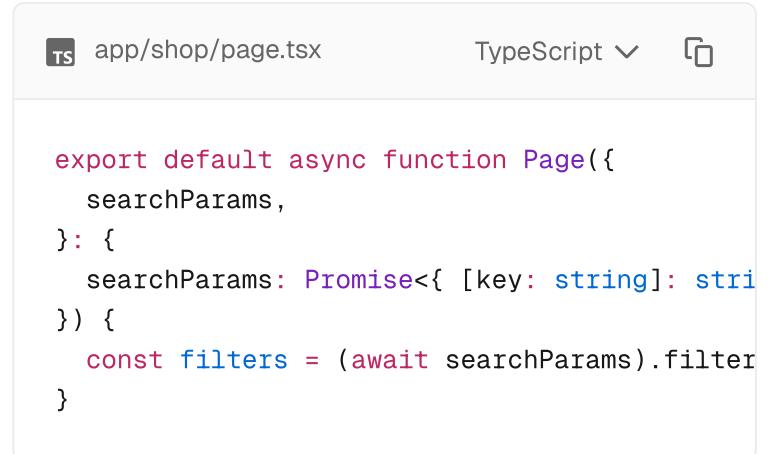
```
export default async function Page({  
  params,  
}: {  
  params: Promise<{ slug: string }>  
}) {  
  const { slug } = await params  
}
```

Example Route	URL
app/shop/[slug]/page.js	/shop/1
app/shop/[category]/[item]/page.js	/shop/1/2
app/shop/[... slug]/page.js	/shop/1/2

- 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.

searchParams (optional)

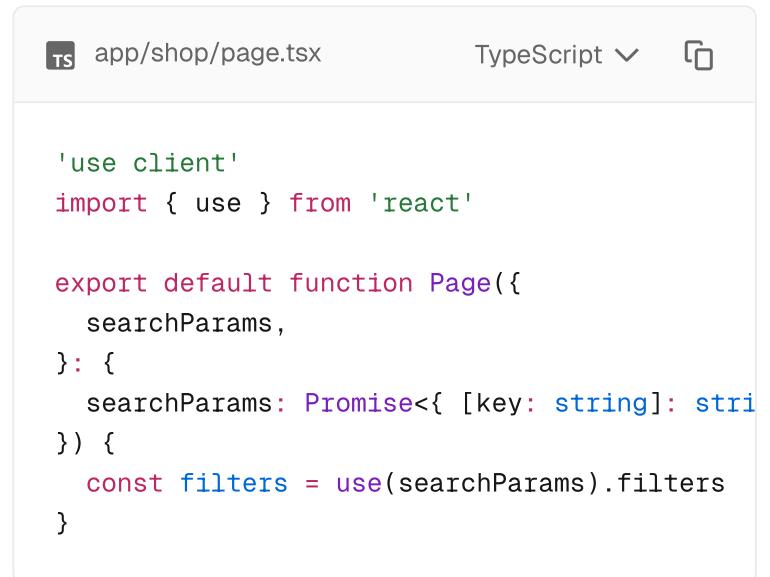
A promise that resolves to an object containing the [search parameters](#) ↗ of the current URL. For example:



app/shop/page.tsx TypeScript ↗

```
export default async function Page({  
  searchParams,  
}: {  
  searchParams: Promise<{ [key: string]: string }>  
) {  
  const filters = (await searchParams).filter  
}
```

Client Component **pages** can also access `searchParams` using React's [use](#) ↗ hook:



app/shop/page.tsx TypeScript ↗

```
'use client'  
import { use } from 'react'  
  
export default function Page({  
  searchParams,  
}: {  
  searchParams: Promise<{ [key: string]: string }>  
) {  
  const filters = use(searchParams).filters  
}
```

Example URL

searchParams

/shop?a=1

Promise<{ a: '1' }>

/shop?a=1&b=2

Promise<{ a: '1', b: '2' }>

/shop?a=1&a=2

Promise<{ a: ['1', '2'] }>

- Since the `searchParams` prop is a promise. You must use `async/await` or React's [use](#) ↗ function to access the values.

- In version 14 and earlier, `searchParams` 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.
- `searchParams` is a **Dynamic API** whose values cannot be known ahead of time. Using it will opt the page into **dynamic rendering** at request time.
- `searchParams` is a plain JavaScript object, not a `URLSearchParams` instance.

Page Props Helper

You can type pages with `PageProps` to get strongly typed `params` and `searchParams` from the route literal. `PageProps` is a globally available helper.



```
TS app/blog/[slug]/page.tsx TypeScript ▾ ⌂
export default async function Page(props: PageProps) {
  const { slug } = await props.params
  const query = await props.searchParams
  return <h1>Blog Post: {slug}</h1>
}
```

Good to know

- Using a literal route (e.g. `'/blog/[slug]'`) enables autocomplete and strict keys for `params`.
- Static routes resolve `params` to `{}`.
- Types are generated during `next dev`, `next build`, or with `next typegen`.

Examples

Displaying content based on `params`

Using [dynamic route segments](#), you can display or fetch specific content for the page based on the `params` prop.



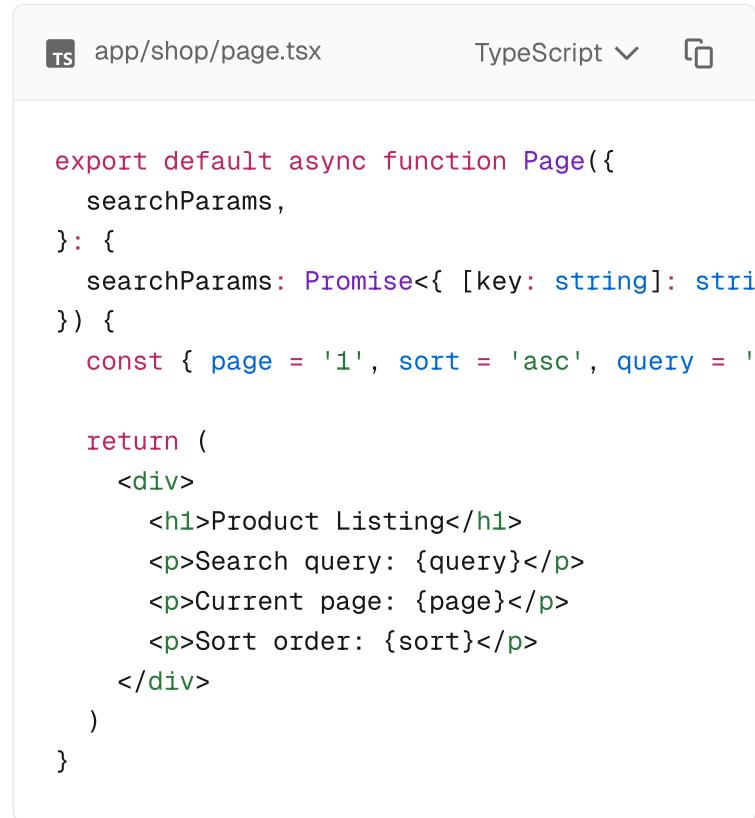
The screenshot shows a code editor window with the following details:

- File: app/blog/[slug]/page.tsx
- Language: TypeScript
- Content:

```
export default async function Page({
  params,
}: {
  params: Promise<{ slug: string }>
}) {
  const { slug } = await params
  return <h1>Blog Post: {slug}</h1>
}
```

Handling filtering with `searchParams`

You can use the `searchParams` prop to handle filtering, pagination, or sorting based on the query string of the URL.



The screenshot shows a code editor window with the following details:

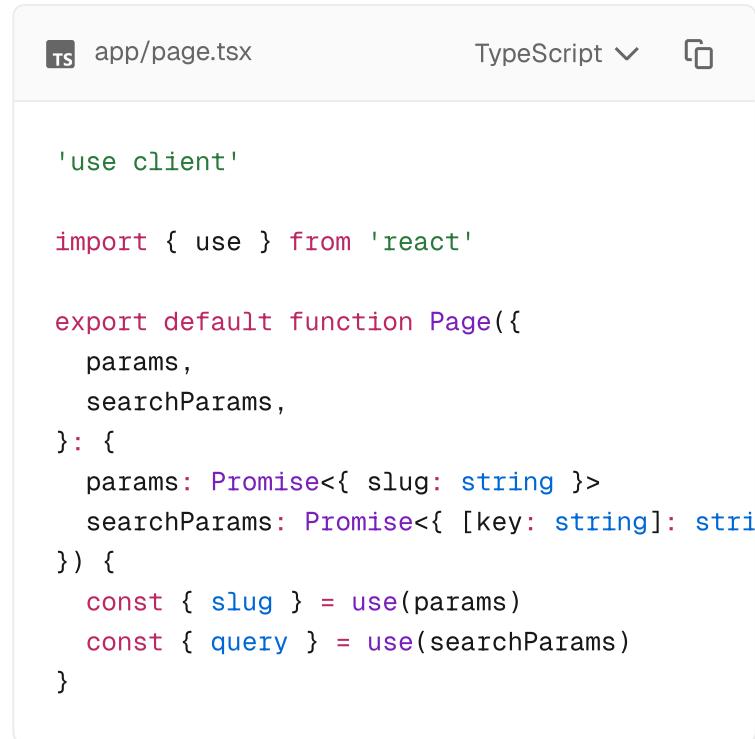
- File: app/shop/page.tsx
- Language: TypeScript
- Content:

```
export default async function Page({
  searchParams,
}: {
  searchParams: Promise<{ [key: string]: string }>
}) {
  const { page = '1', sort = 'asc', query = '' } = searchParams

  return (
    <div>
      <h1>Product Listing</h1>
      <p>Search query: {query}</p>
      <p>Current page: {page}</p>
      <p>Sort order: {sort}</p>
    </div>
  )
}
```

Reading `searchParams` and `params` in Client Components

To use `searchParams` and `params` in a Client Component (which cannot be `async`), you can use React's `use` ↗ function to read the promise:



```
TS app/page.tsx TypeScript ▾
```

```
'use client'

import { use } from 'react'

export default function Page({
  params,
  searchParams,
}: {
  params: Promise<{ slug: string }>
  searchParams: Promise<{ [key: string]: string }>
}) {
  const { slug } = use(params)
  const { query } = use(searchParams)
}
```

Version History

Version	Changes
v15.0.0-RC	<code>params</code> and <code>searchParams</code> are now promises. A codemod is available.
v13.0.0	<code>page</code> introduced.

Was this helpful?    

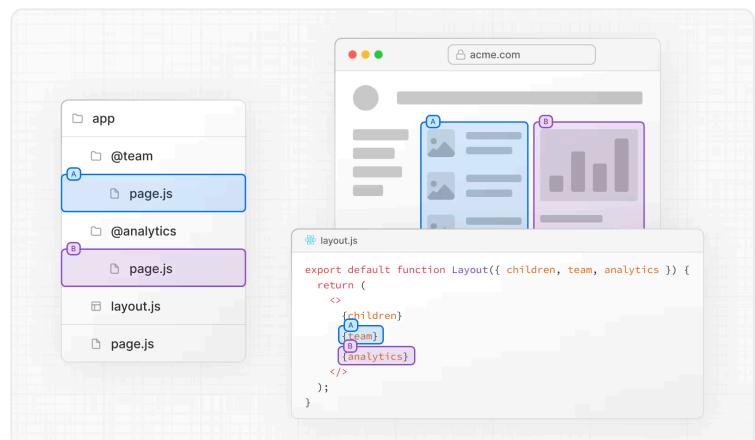
 Using App Router
Features available in /app

 Latest Version
15.5.4

Parallel Routes

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:



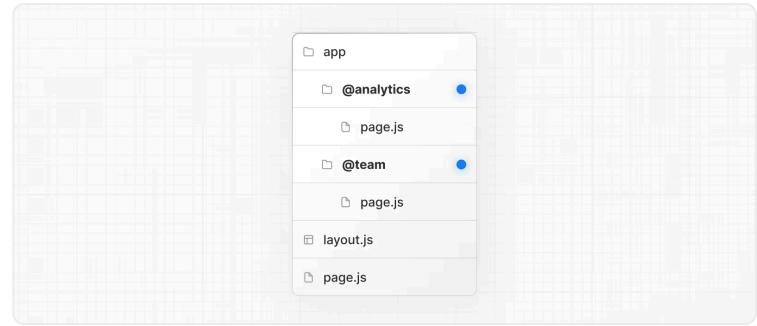
Convention

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:

```
TS app/layout.tsx TypeScript ▾
```

```
export default function Layout({
  children,
  team,
  analytics,
}: {
  children: React.ReactNode
  analytics: React.ReactNode
  team: React.ReactNode
}) {
  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.

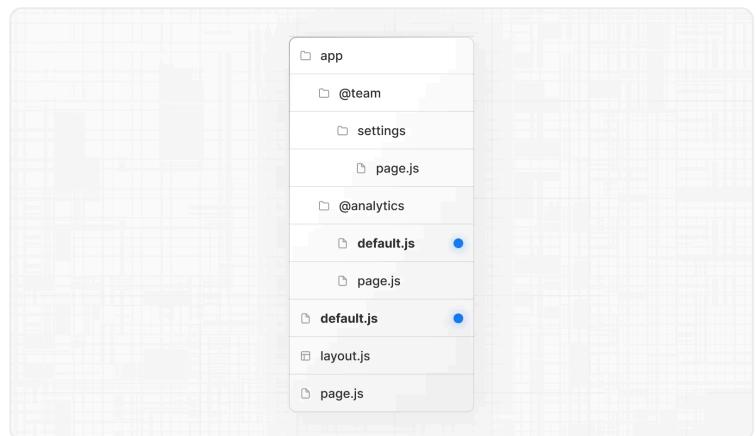
Good to know:

- 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`.

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.

Behavior

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.

Examples

With `useSelectedLayoutSegment(s)`

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

TS app/layout.tsx

TypeScript ▾



```
'use client'
```

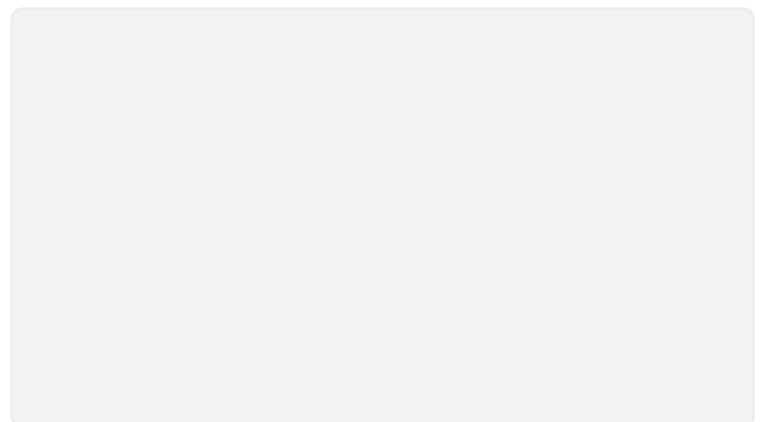
```
import { useSelectedLayoutSegment } from 'nex

export default function Layout({ auth }: { au
  const loginSegment = useSelectedLayoutSegme
  // ...
}
```

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

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:



TS app/dashboard/layout.tsx TypeScript ▾ ⌂

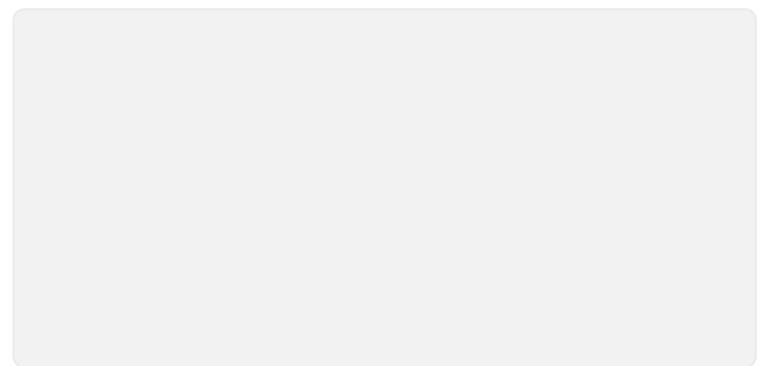
```
import { checkUserRole } from '@/lib/auth'

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

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:

```
TS app/@analytics/layout.tsx TypeScript ▾ ⌂
import Link from 'next/link'

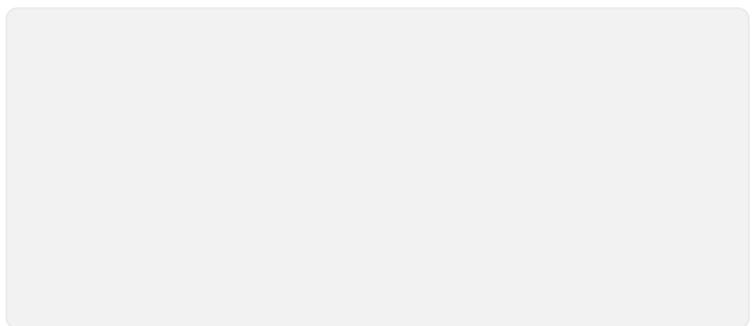
export default function Layout({ children }: {
  return (
    <>
    <nav>
      <Link href="/page-views">Page Views</Link>
      <Link href="/visitors">Visitors</Link>
    </nav>
    <div>{children}</div>
  </>
)
}
```

Modals

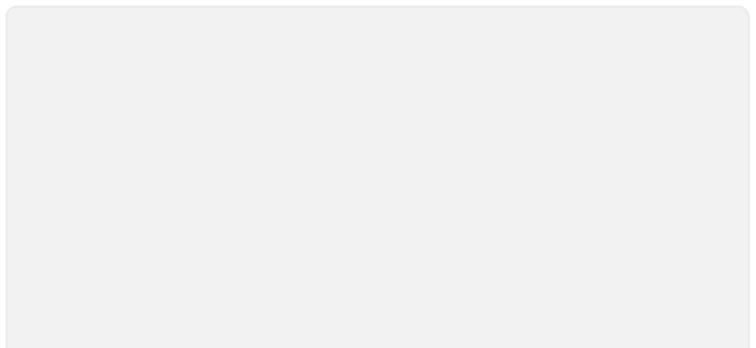
Parallel Routes can be used together with [Intercepting Routes](#) to create modals that support deep linking. This allows you to solve common challenges when building modals, such as:

- 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.



```
TS app/login/page.tsx TypeScript ▾ ⌂
import { Login } from '@/app/ui/login'

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

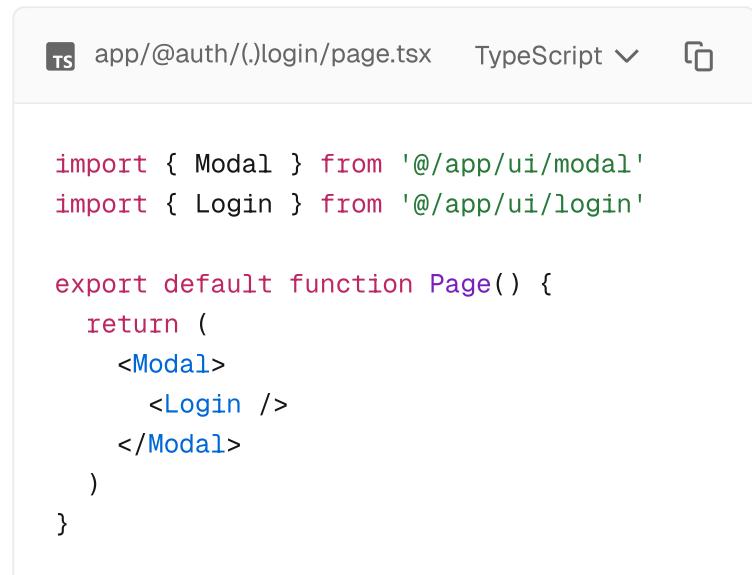
Then, inside the `@auth` slot, add `default.js` file that returns `null`. This ensures that the modal is not rendered when it's not active.



```
app/@auth/default.tsx TypeScript ▾
```

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

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



```
app/@auth/(.)login/page.tsx TypeScript ▾
```

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

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

Good to know:

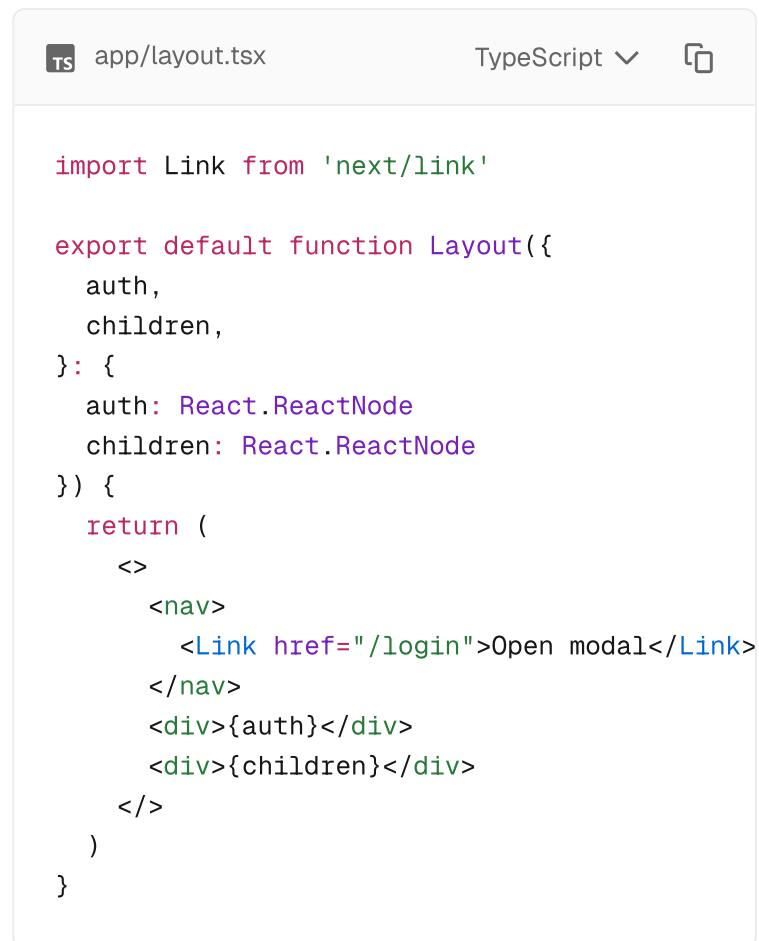
- The convention `(.)` is used for intercepting routes. See [Intercepting Routes](#) docs for more information.
- By separating the `<Modal>` functionality from the modal content (`<Login>`), you can ensure any content inside the modal, e.g. `forms`, are Server Components. See [Interleaving Client and 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.



The screenshot shows a code editor window with a TypeScript file named `app/layout.tsx`. The code defines a `Layout` component that takes `auth` and `children` props. It returns a `<>` element containing a `<nav>` element with a `<Link href="/login">Open modal</Link>` link, followed by two `<div>` elements: one for `auth` and one for `children`.

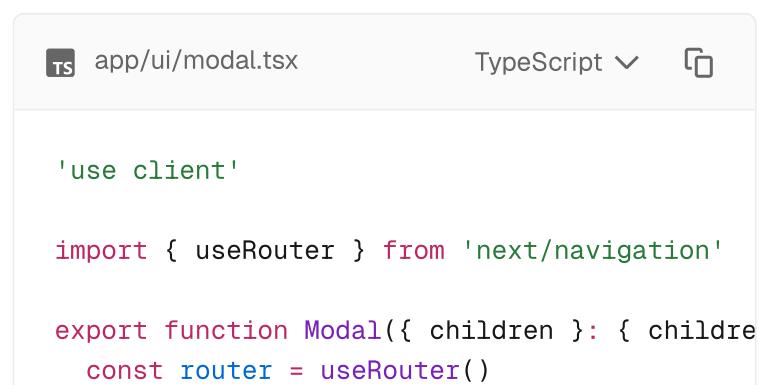
```
import Link from 'next/link'

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>
    </>
  )
}
```

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.



The screenshot shows a code editor window with a TypeScript file named `app/ui/modal.tsx`. The code defines a `Modal` component that takes `children` as a prop. Inside, it uses `useRouter` to get the router and then calls `router.back()`.

```
'use client'

import { useRouter } from 'next/navigation'

export function Modal({ children }: { children: React.ReactNode }) {
  const router = useRouter()
  router.back()
  return (
    <div>{children}</div>
  )
}
```

```
        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:

app/ui/modal.tsx

```
import Link from 'next/link'

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

app/@auth/page.tsx

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

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

app/@auth/[...catchAll]/page...

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

Good to know:

- We use a catch-all route in our `@auth` slot to close the modal because of how parallel routes behave. 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.

Next Steps

default.js

API Reference for
the default.js file.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

public Folder

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:

```
JS avatar.js   
  
import Image from 'next/image'  
  
export function Avatar({ id, alt }) {  
  return <Image src={`/avatars/${id}.png`} alt={alt} />  
}  
  
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

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

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

route.js

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

TS route.ts

TypeScript ▾

```
export async function GET() {  
  return Response.json({ message: 'Hello World' })  
}
```

Reference

HTTP Methods

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

TS route.ts

TypeScript ▾

```
export async function GET(request: Request) {  
  
  export async function HEAD(request: Request)  
  
  export async function POST(request: Request)  
  
  export async function PUT(request: Request)  
  
  export async function DELETE(request: Request)  
  
  export async function PATCH(request: Request)
```

```
// If `OPTIONS` is not defined, Next.js will  
export async function OPTIONS(request: Request
```

Parameters

request (optional)

The `request` object is a [NextRequest](#) object, which is an extension of the Web [Request](#) API. `NextRequest` gives you further control over the incoming request, including easily accessing `cookies` and an extended, parsed, URL object `nextUrl`.

route.ts

TypeScript



```
import type { NextRequest } from 'next/server'  
  
export async function GET(request: NextRequest) {  
  const url = request.nextUrl  
}
```

context (optional)

- `params`: a promise that resolves to an object containing the [dynamic route parameters](#) for the current route.

TS

app/dashboard/[team]/route... TypeScript



```
export async function GET(  
  request: Request,  
  { params }: { params: Promise<{ team: string }> } {  
    const { team } = await params  
  }
```

Example

URL

app/dashboard/[team]/route.js

/dashboard/1

Pa
te
};>

Example

URL

```
app/shop/[tag]/[item]/route.js
```

```
/shop/1/2
```

Pi
tæ
i1
};
]
[
'2

```
app/blog/...slug/route.js
```

```
/blog/1/2
```

Pi
s]
[
'2

Route Context Helper

You can type the Route Handler context using

`RouteContext` to get strongly typed `params` from a route literal. `RouteContext` is a globally available helper.

TS app/users/[id]/route.ts

TypeScript



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

export async function GET(_req: NextRequest,
  const { id } = await ctx.params
  return Response.json({ id })
}
```

Good to know

- Types are generated during `next dev`, `next build` or `next typegen`.

Examples

Cookies

You can read or set cookies with `cookies` from `next/headers`.

TS route.ts

TypeScript

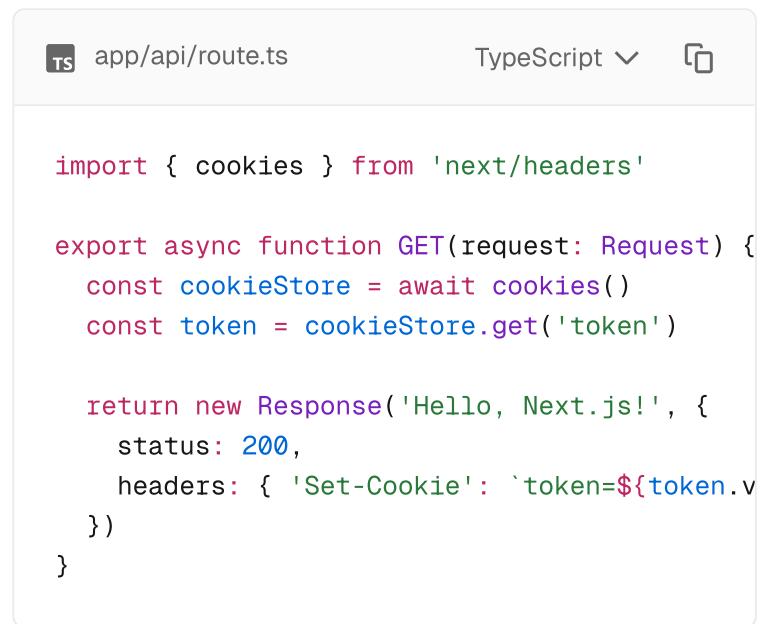


```
import { cookies } from 'next/headers'

export async function GET(request: NextRequest) {
  const cookieStore = await cookies()

  const a = cookieStore.get('a')
  const b = cookieStore.set('b', '1')
  const c = cookieStore.delete('c')
}
```

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



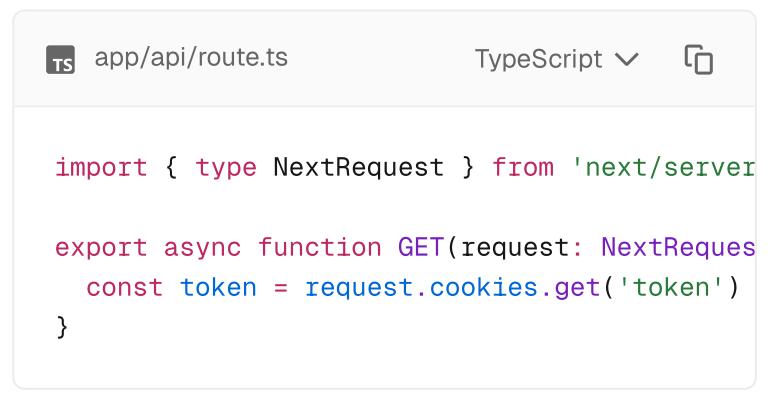
The screenshot shows a code editor window with a TypeScript file named `app/api/route.ts`. The code defines a `GET` handler that retrieves a cookie, sets a new cookie, and returns a `Response` object with a `Set-Cookie` header containing the token value.

```
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}` }
  })
}
```

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



The screenshot shows a code editor window with a TypeScript file named `app/api/route.ts`. The code defines a `GET` handler that reads a cookie from the `request.cookies` object.

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

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

Headers

You can read headers with `headers` from `next/headers`.



The screenshot shows a code editor window with a file named `route.ts`. The file contains a single line of code that imports the `headers` module.

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

TypeScript ▾

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

export async function GET(request: NextRequest) {
  const headersList = await headers()
  const referer = headersList.get('referer')
}
```

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

TS app/api/route.ts TypeScript ▾

```
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 },
  })
}
```

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

TS app/api/route.ts TypeScript ▾

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

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

Revalidating Cached Data

You can [revalidate cached data](#) using the `revalidate` route segment config option.

TS app/posts/route.ts TypeScript ▾

TypeScript ▾

```
export const revalidate = 60

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

  return Response.json(posts)
}
```

Redirects

```
TS app/api/route.ts TypeScript ▾ ⌂

import { redirect } from 'next/navigation'

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

Dynamic Route Segments

Route Handlers can use [Dynamic Segments](#) to create request handlers from dynamic data.

```
TS app/items/[slug]/route.ts TypeScript ▾ ⌂

export async function GET(
  request: Request,
  { params }: { params: Promise<{ slug: string }> }
) {
  const { slug } = await params // 'a', 'b'
}
```

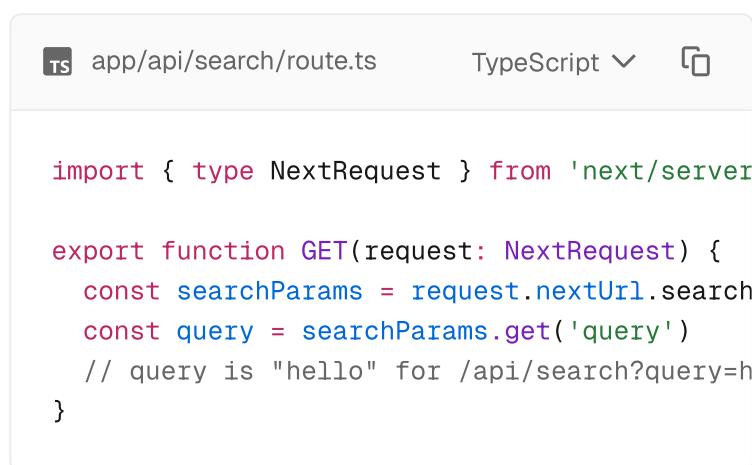
Example

Route	URL	params
app/items/[slug]/route.js	/items/a	Promise<{ slug: 'a' }>
app/items/[slug]/route.js	/items/b	Promise<{ slug: 'b' }>

Route	URL	Example params
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 includes [some additional convenience methods](#), such as those for more easily handling query parameters.



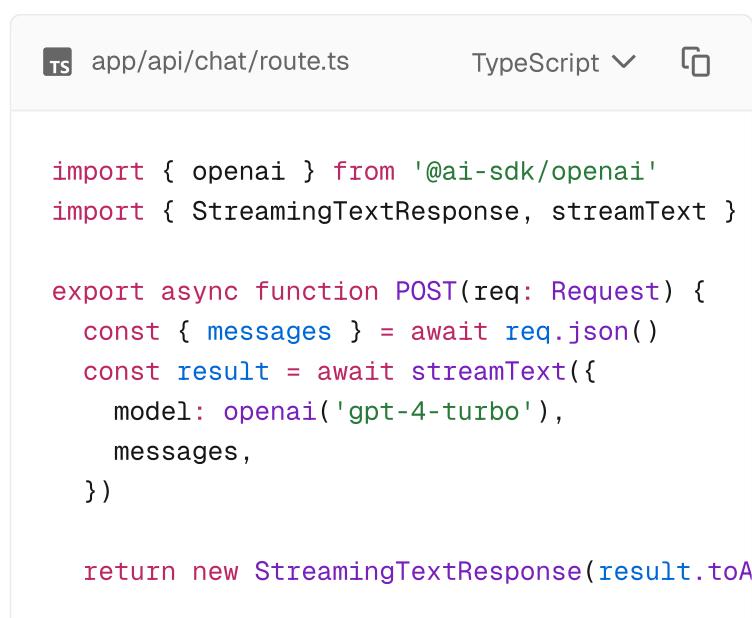
```
TS app/api/search/route.ts TypeScript ▾ ⌂

import { type NextRequest } from 'next/server'

export function GET(request: NextRequest) {
  const searchParams = request.nextUrl.search
  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 ↗](#).



```
TS app/api/chat/route.ts TypeScript ▾ ⌂

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

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.toA
```

```
}
```

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



The screenshot shows a code editor window with the following details:

- File: app/api/route.ts
- Language: TypeScript
- Icon: A small icon representing the file type or status.

```
// https://developer.mozilla.org/docs/Web/API
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)
}
```

Request Body

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

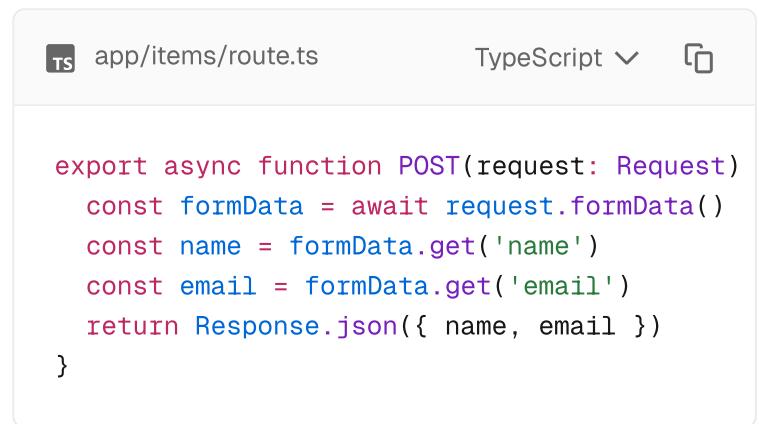


app/items/route.ts TypeScript

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

Request Body FormData

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



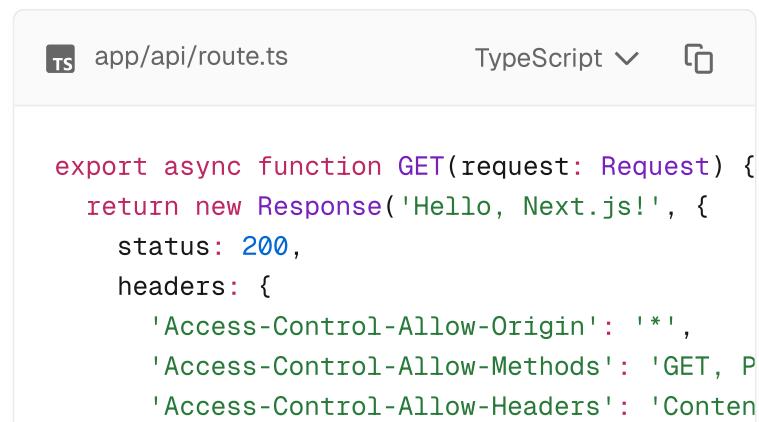
app/items/route.ts TypeScript

```
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 })
}
```

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:



app/api/route.ts TypeScript

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

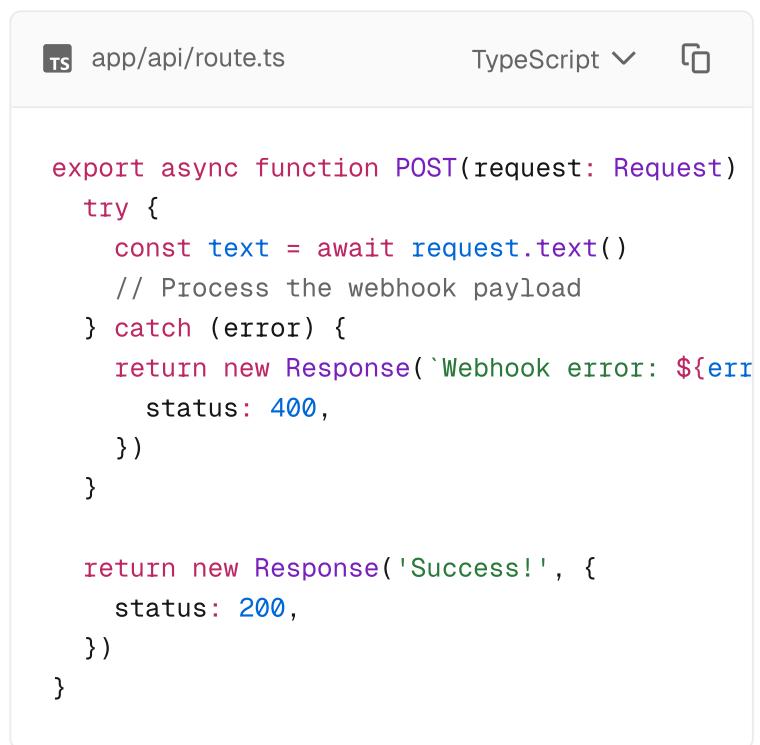
```
    },  
  })  
}
```

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:



```
TS app/api/route.ts TypeScript ▾
```

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

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'
```

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

Version History

Version	Changes
v15.0.0-RC	<code>context.params</code> is now a promise. A codemod is available
v15.0.0-RC	The default caching for <code>GET</code> handlers was changed from static to dynamic
v13.2.0	Route Handlers are introduced.



 Using App Router
Features available in /app

 Latest Version
15.5.4

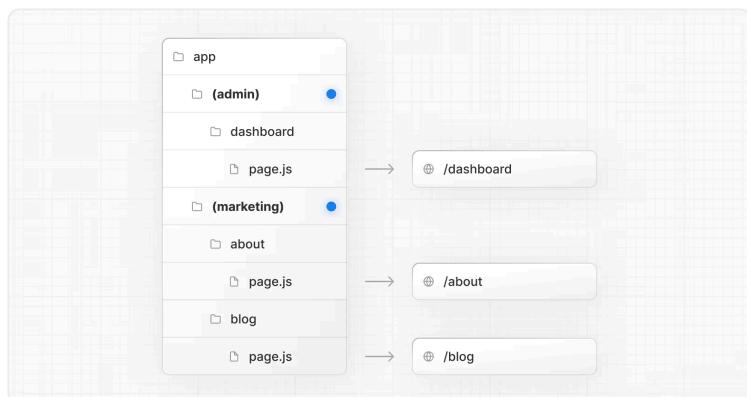
Route Groups

Route Groups are a folder convention that let you organize routes by category or team.

Convention

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

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



Use cases

- Organizing routes by team, concern, or feature.
- Defining multiple [root layouts](#).
- Opting specific route segments into sharing a layout, while keeping others out.

Caveats

- **Full page load:** If you navigate between routes that use different root layouts, it'll trigger a full page reload. For example, navigating from `/cart` that uses `app/(shop)/layout.js` to `/blog` that uses `app/(marketing)/layout.js`. This **only** applies to multiple root layouts.
- **Conflicting paths:** Routes in different groups should not resolve to the same URL path. For example, `(marketing)/about/page.js` and `(shop)/about/page.js` would both resolve to `/about` and cause an error.
- **Top-level root layout:** If you use multiple root layouts without a top-level `layout.js` file, make sure your home route `(/)` is defined within one of the route groups, e.g. `app/(marketing)/page.js`.

Was this helpful?    

 Using App Router

Features available in /app

 Latest Version

15.5.4

Route Segment Config

The options outlined on this page are disabled if the `cacheComponents` flag is on, and will eventually be deprecated in the future.

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

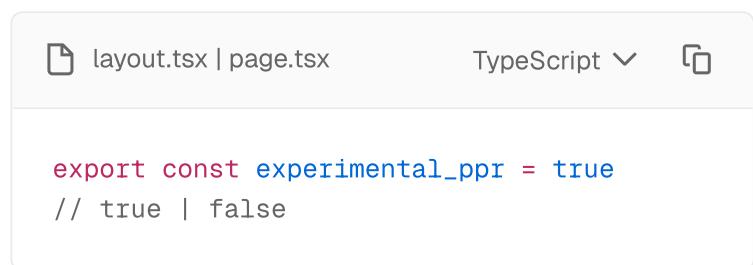
Option	Type	Default
<code>experimental_ppr</code>	boolean	
<code>dynamic</code>	'auto' 'force-dynamic' 'error' 'force-static'	'auto'
<code>dynamicParams</code>	boolean	true
<code>revalidate</code>	false 0 number	false
<code>fetchCache</code>	'auto' 'default-cache' 'only-cache' 'force-cache' 'force-no-store'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

Option	Type	Default
<code>maxDuration</code>	number	Set by deployment platform

Options

`experimental_ppr`

Enable [Partial Prerendering \(PPR\)](#) for a layout or page.



A screenshot of a code editor interface. At the top, there's a toolbar with icons for file operations and a dropdown menu labeled "TypeScript". Below the toolbar, the code editor shows two files: "layout.tsx" and "page.tsx". In the code area, the following line of code is visible:

```
export const experimental_ppr = true
// true | false
```

`dynamic`

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



A screenshot of a code editor interface. At the top, there's a toolbar with icons for file operations and a dropdown menu labeled "TypeScript". Below the toolbar, the code editor shows three files: "layout.tsx", "page.tsx", and "route.ts". In the code area, the following line of code is visible:

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

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

- `'auto'` (default): The default option to cache as much as possible without preventing any components from opting into dynamic behavior.
- `'force-dynamic'` : Force [dynamic rendering](#), which will result in routes being rendered for each user at request time. This option is equivalent to:
 - Setting the option of every `fetch()` request in a layout or page to `{ cache: 'no-store', next: { revalidate: 0 } }`.
 - Setting the segment config to `export const fetchCache = 'force-no-store'`
- `'error'` : Force static rendering and cache the data of a layout or page by causing an error if any components use [Dynamic APIs](#) or uncached data. This option is equivalent to:
 - `getStaticProps()` in the `pages` directory.
 - Setting the option of every `fetch()` request in a layout or page to `{ cache: 'force-cache' }`.
 - Setting the segment config to `fetchCache = 'only-cache'`.
- `'force-static'` : Force static rendering and cache the data of a layout or page by forcing `cookies`, `headers()` and `useSearchParams()` to return empty values. It is possible to `revalidate`, `revalidatePath`, or `revalidateTag`, in pages or layouts rendered with `force-static`.

Good to know:

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

dynamicParams

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

layout.tsx | page.tsx

TypeScript

```
export const dynamicParams = true // true | f
```

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

Good to know:

- This option replaces the `fallback: true | false | blocking` option of `getStaticPaths` in the `pages` directory.
- To statically render all paths the first time they're visited, you'll need to return an empty array in `generateStaticParams` or utilize `export const dynamic = 'force-static'`.
- When `dynamicParams = true`, the segment uses [Streaming Server Rendering](#).

revalidate

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

layout.tsx | page.tsx | route.ts

TypeScript

```
export const revalidate = false  
// false | 0 | number
```

- `false` (default): The default heuristic to cache any `fetch` requests that set their `cache` option to `'force-cache'` or are discovered before a [Dynamic API](#) is used. Semantically equivalent to `revalidate: Infinity` which effectively means the resource should be cached indefinitely. It is still possible for individual `fetch` requests to use `cache: 'no-store'` or `revalidate: 0` to avoid being cached and make the route dynamically rendered. Or set `revalidate` to a positive number lower than the route default to increase the revalidation frequency of a route.
- `0`: Ensure a layout or page is always [dynamically rendered](#) even if no Dynamic APIs or uncached data fetches are discovered. This option changes the default of `fetch` requests that do not set a `cache` option to `'no-store'` but leaves `fetch` requests that opt into `'force-cache'` or use a positive `revalidate` as is.
- `number` : (in seconds) Set the default revalidation frequency of a layout or page to `n` seconds.

Good to know:

- The `revalidate` value needs to be statically analyzable. For example `revalidate = 600` is valid, but `revalidate = 60 * 10` is not.
- The `revalidate` value is not available when using `runtime = 'edge'`.
- In Development, Pages are *always* rendered on-demand and are never cached. This allows you to see changes immediately without waiting for a revalidation period to pass.

Revalidation Frequency

- The lowest `revalidate` across each layout and page of a single route will determine the

revalidation frequency of the *entire* route. This ensures that child pages are revalidated as frequently as their parent layouts.

- Individual `fetch` requests can set a lower `revalidate` than the route's default `revalidate` to increase the revalidation frequency of the entire route. This allows you to dynamically opt-in to more frequent revalidation for certain routes based on some criteria.

fetchCache

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

runtime

We recommend using the Node.js runtime for rendering your application, and the Edge runtime for Middleware.

```
TS layout.tsx | page.tsx | route.ts TypeScript ▾ ⌂
export const runtime = 'nodejs'
// 'nodejs' | 'edge'
```

- `'nodejs'` (default)
- `'edge'`

preferredRegion

```
TS layout.tsx | page.tsx | route.ts TypeScript ▾ ⌂
export const preferredRegion = 'auto'
// 'auto' | 'global' | 'home' | ['iad1', 'sfo']
```

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

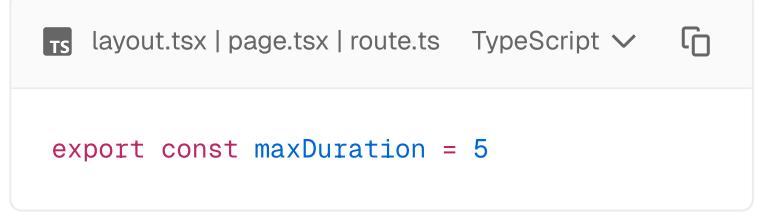
Good to know:

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

maxDuration

By default, Next.js does not limit the execution of server-side logic (rendering a page or handling an API). Deployment platforms can use `maxDuration` from the Next.js build output to add specific execution limits.

Note: This setting requires Next.js `13.4.10` or higher.



TS layout.tsx | page.tsx | route.ts TypeScript ▾ ⌂

```
export const maxDuration = 5
```

Good to know:

- If using [Server Actions](#), set the `maxDuration` at the page level to change the default timeout of all Server Actions used on the page.

generateStaticParams

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

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

Version History

Version

v15.0.0-
RC

```
export const runtime = "experimental-  
edge"
```

deprecated. A [codemod](#) is available.

Was this helpful?     

 Using App Router
Features available in /app

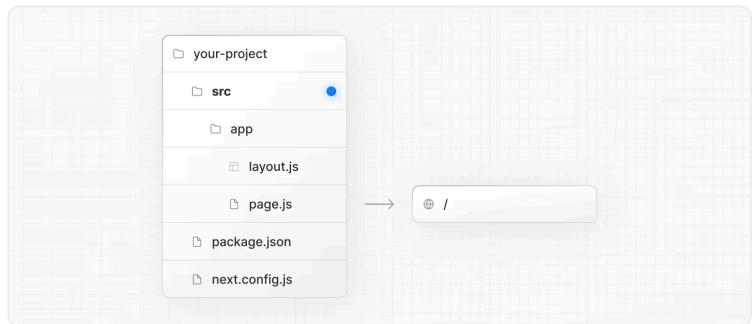
 Latest Version
15.5.4

src Folder

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` folder.

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` folder, 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` folder.
- 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/`.

Next Steps

Project Struc...

Learn the folder and file conventions in...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

template.js

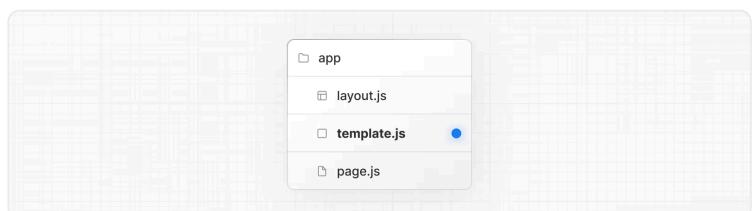
A **template** file is similar to a [layout](#) in that it wraps a layout or page. Unlike layouts that persist across routes and maintain state, templates are given a unique key, meaning children Client Components reset their state on navigation.

They are useful when you need to:

- Resynchronize `useEffect` on navigation.
- Reset the state of a child Client Components on navigation. For example, an input field.
- To change default framework behavior. For example, Suspense boundaries inside layouts only show a fallback on first load, while templates show it on every navigation.

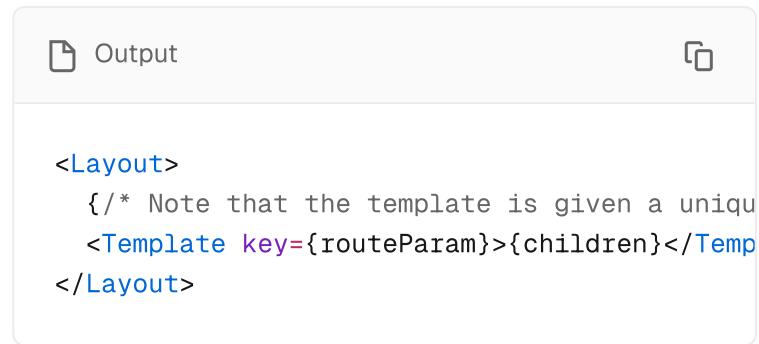
Convention

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 }  
  return <div>{children}</div>  
}
```

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



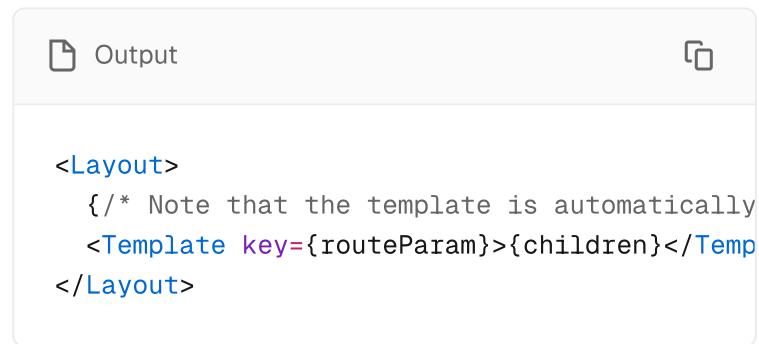
The screenshot shows a code editor window with a title bar labeled "Output". The main area contains the following code:

```
<Layout>  
  /* Note that the template is given a unique key */  
  <Template key={routeParams}>{children}</Template>  
</Layout>
```

Props

`children` (required)

Template accepts a `children` prop.



The screenshot shows a code editor window with a title bar labeled "Output". The main area contains the following code:

```
<Layout>  
  /* Note that the template is automatically remounted on navigation */  
  <Template key={routeParams}>{children}</Template>  
</Layout>
```

Behavior

- **Server Components:** By default, templates are Server Components.
- **Remount on navigation:** Templates receive a unique key automatically. Navigating to a new

route causes the template and its children to remount.

- **State reset:** Any Client Component inside the template will reset its state on navigation.
 - **Effect re-run:** Effects like `useEffect` will synchronize as the component remounts.
 - **DOM reset:** DOM elements inside the template are fully recreated.
-

Version History

Version	Changes
v13.0.0	<code>template</code> introduced.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

unauthorized.js

 This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

The `unauthorized` file is used to render UI when the `unauthorized` function is invoked during authentication. Along with allowing you to customize the UI, Next.js will return a `401` status code.

TS app/unauthorized.tsx TypeScript ▾ 

```
import Login from '@/app/components/Login'

export default function Unauthorized() {
  return (
    <main>
      <h1>401 - Unauthorized</h1>
      <p>Please log in to access this page.</p>
      <Login />
    </main>
  )
}
```

Reference

Props

`unauthorized.js` components do not accept any props.

Examples

Displaying login UI to unauthenticated users

You can use `unauthorized` function to render the `unauthorized.js` file with a login UI.

```
TS app/dashboard/page.tsx TypeScript ▾ ⌂

import { verifySession } from '@/app/lib/dal'
import { unauthorized } from 'next/navigation'

export default async function DashboardPage() {
  const session = await verifySession()

  if (!session) {
    unauthorized()
  }

  return <div>Dashboard</div>
}
```

```
TS app/unauthorized.tsx TypeScript ▾ ⌂

import Login from '@/app/components/Login'

export default function UnauthorizedPage() {
  return (
    <main>
      <h1>401 - Unauthorized</h1>
      <p>Please log in to access this page.</p>
      <Login />
    </main>
  )
}
```

Version History

Version Changes

v15.1.0

unauthorized.js introduced.

Next Steps

unauthorized

API Reference for
the unauthorized
function.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



Metadata Files API Reference



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

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

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

Good to know:

- Special Route Handlers like `sitemap.ts`, `opengraph-image.tsx`, and `icon.tsx`, and other **metadata files** are cached by default.
- If using along with `middleware.ts`, configure the `matcher` to exclude the metadata files.

[favicon, icon,...](#)

API Reference for the Favicon, Icon and Apple Icon fil...

[manifest.json](#)

API Reference for manifest.json file.

[opengraph-i...](#)

API Reference for
the Open Graph
Image and Twitte...

[robots.txt](#)

API Reference for
robots.txt file.

[sitemap.xml](#)

API Reference for
the sitemap.xml
file.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

favicon, icon, and apple-icon

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

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

There are two ways to set app icons:

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

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

Use an image file to set an app icon by placing a `favicon`, `icon`, or `apple-icon` image file within your `/app` directory. The `favicon` image can only be located in the top level of `app/`.

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

File convention	Supported file types	Valid locations
<code>favicon</code>	<code>.ico</code>	<code>app/</code>

File convention	Supported file types	Valid locations
icon	.ico , .jpg , .jpeg , .png , .svg	app/**/*
apple-icon	.jpg , .jpeg , .png	app/**/*

favicon

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

```

 <head> output 
<link rel="icon" href="/favicon.ico" sizes="a

```

icon

Add an `icon.(ico|jpg|jpeg|png|svg)` image file.

```

 <head> output 
<link
  rel="icon"
  href="/icon?<generated>"
  type="image/<generated>"
  sizes="<generated>"
/>

```

apple-icon

Add an `apple-icon.(jpg|jpeg|png)` image file.

```

 <head> output 

```

```
<link  
    rel="apple-touch-icon"  
    href="/apple-icon?<generated>"  
    type="image/<generated>"  
    sizes="<generated>"  
/>
```

Good to know:

- You can set multiple icons by adding a number suffix to the file name. For example, `icon1.png`, `icon2.png`, etc. Numbered files will sort lexically.
- Favicons can only be set in the root `/app` segment. If you need more granularity, you can use `icon`.
- The appropriate `<link>` tags and attributes such as `rel`, `href`, `type`, and `sizes` are determined by the icon type and metadata of the evaluated file.
- For example, a 32 by 32px `.png` file will have `type="image/png"` and `sizes="32x32"` attributes.
- `sizes="any"` is added to icons when the extension is `.svg` or the image size of the file is not determined. More details in this [favicon handbook ↗](#).

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

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

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

File convention	Supported file types
<code>icon</code>	<code>.js</code> , <code>.ts</code> , <code>.tsx</code>
<code>apple-icon</code>	<code>.js</code> , <code>.ts</code> , <code>.tsx</code>

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

```
TS app/icon.tsx TypeScript ▾
```

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

// Image metadata
export const size = {
  width: 32,
  height: 32,
}
export const contentType = 'image/png'

// Image generation
export default function Icon() {
  return new ImageResponse(
    (
      // ImageResponse JSX element
      <div
        style={{
          fontSize: 24,
          background: 'black',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
          color: 'white',
        }}
      >
        A
      </div>
    ),
    // ImageResponse options
    {
      // For convenience, we can re-use the e
      // config to also set the ImageResponse
      ...size,
    }
  )
}
```

```
□ <head> output ▾
```

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

Good to know:

- By default, generated icons are **statically optimized** (generated at build time and cached) unless they use **Dynamic APIs** or uncached data.
- You can generate multiple icons in the same file using `generateImageMetadata`.
- You cannot generate a `favicon` icon. Use `icon` or a `favicon.ico` file instead.
- App icons are special Route Handlers that are cached by default unless they use a **Dynamic API** or **dynamic config** option.

Props

The default export function receives the following props:

`params (optional)`

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



```
TS app/shop/[slug]/icon.tsx TypeScript ▾ ⌂
export default function Icon({ params }: { pa
// ...
})
```

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

Returns

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

ReadableStream | Response .

Good to know: ImageResponse satisfies this return type.

Config exports

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

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

size

```
TS icon.tsx | apple-icon.tsx TypeScript ▾ ⌂  
  
export const size = { width: 32, height: 32 }  
  
export default function Icon() {}
```

<head> output

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

contentType

```
TS icon.tsx | apple-icon.tsx TypeScript ▾ ⌂  
  
export const contentType = 'image/png'  
  
export default function Icon() {}
```

<head> output

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

Route Segment Config

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

Version History

Version	Changes
---------	---------

v13.3.0	<code>favicon</code> <code>icon</code> and <code>apple-icon</code> introduced
---------	---

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

manifest.json

Add or generate a `manifest.(json|webmanifest)` file that matches the [Web Manifest Specification](#) ↗ in the `root` of `app` directory to provide information about your web application for the browser.

Static Manifest file

```
 app/manifest.json | app/manifest.webmanifest   
  
{  
  "name": "My Next.js Application",  
  "short_name": "Next.js App",  
  "description": "An application built with N  
  "start_url": "/"  
  // ...  
}
```

Generate a Manifest file

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

Good to know: `manifest.js` is special Route Handlers that is cached by default unless it uses a [Dynamic API](#) or [dynamic config](#) option.

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

export default function manifest(): MetadataR
return {
  name: 'Next.js App',
  short_name: 'Next.js App',
  description: 'Next.js App',
  start_url: '/',
  display: 'standalone',
  background_color: '#fff',
  theme_color: '#fff',
  icons: [
    {
      src: '/favicon.ico',
      sizes: 'any',
      type: 'image/x-icon',
    },
  ],
}
}
```

Manifest Object

The manifest object contains an extensive list of options that may be updated due to new web standards. For information on all the current options, refer to the [MetadataRoute.Manifest](#) type in your code editor if using [TypeScript](#) or see the [MDN ↗ docs](#).

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

opengraph-image and twitter-image

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

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

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

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

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

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

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

File convention

Supported file types

`opengraph-image`

.jpg , .jpeg , .png , .gif

File convention

Supported file types

twitter-image

.jpg , .jpeg , .png , .gif

opengraph-image.alt

.txt

twitter-image.alt

.txt

Good to know:

The `twitter-image` file size must not exceed [5MB ↗](#), and the `opengraph-image` file size must not exceed [8MB ↗](#). If the image file size exceeds these limits, the build will fail.

opengraph-image

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

 <head> output 

```
<meta property="og:image" content=<generated  
<meta property="og:image:type" content=<gen  
<meta property="og:image:width" content=<gen  
<meta property="og:image:height" content=<ge
```

twitter-image

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

 <head> output 

```
<meta name="twitter:image" content=<generate  
<meta name="twitter:image:type" content=<gen  
<meta name="twitter:image:width" content=<ge  
<meta name="twitter:image:height" content=<g
```

opengraph-image.alt.txt

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

```
📄 opengraph-image.alt.txt 🗑  
  
About Acme
```

```
📄 <head> output 🗑  
  
<meta property="og:image:alt" content="About
```

`twitter-image.alt.txt`

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

```
📄 twitter-image.alt.txt 🗑  
  
About Acme
```

```
📄 <head> output 🗑  
  
<meta property="twitter:image:alt" content="A
```

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

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

Generate a route segment's shared image by creating an `opengraph-image` or `twitter-image` route that default exports a function.

File convention	Supported file types
<code>opengraph-image</code>	<code>.js</code> , <code>.ts</code> , <code>.tsx</code>
<code>twitter-image</code>	<code>.js</code> , <code>.ts</code> , <code>.tsx</code>

Good to know:

- By default, generated images are **statically optimized** (generated at build time and cached) unless they use **Dynamic APIs** or uncached data.
- You can generate multiple Images in the same file using `generateImageMetadata`.
- `opengraph-image.js` and `twitter-image.js` are special Route Handlers that is cached by default unless it uses a **Dynamic API** or **dynamic config** option.

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

```
TS app/about/opengraph-image... TypeScript ▾ ⌂
```

```
import { ImageResponse } from 'next/og'
import { readFile } from 'node:fs/promises'
import { join } from 'node:path'

// Image metadata
export const alt = 'About Acme'
export const size = {
  width: 1200,
  height: 630,
}

export const contentType = 'image/png'

// Image generation
export default async function Image() {
  // Font loading, process.cwd() is Next.js p
  const interSemiBold = await readFile(
    join(process.cwd(), 'assets/Inter-SemiBol
  )
```

```

        return new ImageResponse(
          (
            // ImageResponse JSX element
            <div
              style={{
                fontSize: 128,
                background: 'white',
                width: '100%',
                height: '100%',
                display: 'flex',
                alignItems: 'center',
                justifyContent: 'center',
              }})
            >
              About Acme
            </div>
          ),
          // ImageResponse options
        {
          // For convenience, we can re-use the e
          // size config to also set the ImageRes
          ...size,
          fonts: [
            {
              name: 'Inter',
              data: interSemiBold,
              style: 'normal',
              weight: 400,
            },
          ],
        }
      )
    }
  }
}

```

 <head> output



```

<meta property="og:image" content=<generated
<meta property="og:image:alt" content="About
<meta property="og:image:type" content="image
<meta property="og:image:width" content="1200
<meta property="og:image:height" content="630

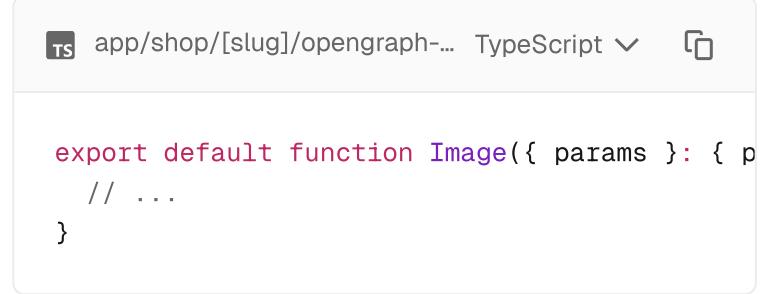
```

Props

The default export function receives the following props:

params (optional)

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



The screenshot shows a code editor window with a TypeScript file named `app/shop/[slug]/opengraph-image.ts`. The code defines a function `Image` that takes a parameter `params` and returns a promise. The code is as follows:

```
export default function Image({ params }: { p  
// ...  
})
```

Route	URL	parameters
<code>app/shop/opengraph-image.js</code>	<code>/shop</code>	<code>{ }</code>
<code>app/shop/[slug]/opengraph-image.js</code>	<code>/shop/1</code>	<code>{ '1' }</code>
<code>app/shop/[tag]/[item]/opengraph-image.js</code>	<code>/shop/1/2</code>	<code>{ '1', '2' }</code>

Returns

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

Good to know: `ImageResponse` satisfies this return type.

Config exports

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

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

alt

TS opengraph-image.tsx | twitte... TypeScript ↗

```
export const alt = 'My images alt text'
```

```
export default function Image() {}
```

size

TS opengraph-image.tsx | twitte... TypeScript ↗

```
export const size = { width: 1200, height: 63
```

```
export default function Image() {}
```

<head> output ↗

```
<meta property="og:image:width" content="1200
<meta property="og:image:height" content="630
```

contentType

TS opengraph-image.tsx | twitte... TypeScript ↗

```
export const contentType = 'image/png'
```

```
export default function Image() {}
```

📄 <head> output



```
<meta property="og:image:type" content="image"
```

Route Segment Config

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

Examples

Using external data

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

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

TS app/posts/[slug]/opengraph... TypeScript ▾



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

```
export const alt = 'About Acme'
```

```
export const size = {
```

```
  width: 1200,
```

```
  height: 630,
```

```
}
```

```
export const contentType = 'image/png'
```

```
export default async function Image({ params })
```

```
  const post = await fetch(`https://.../posts`)
```

```
)
```

```
  res.json()
```

```
)
```

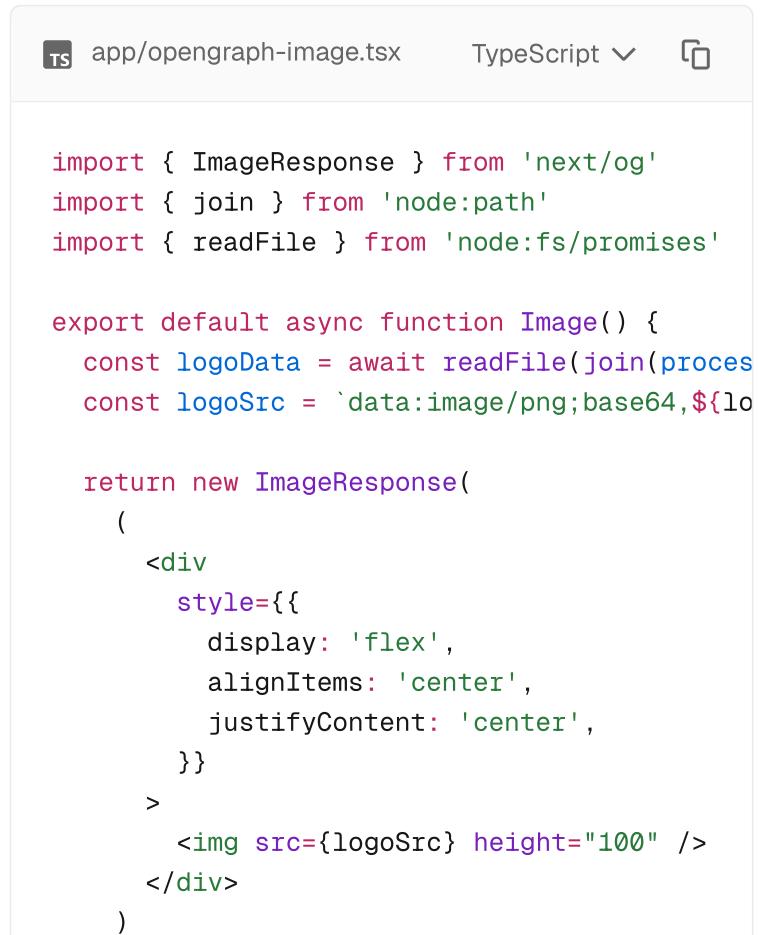
```

        <div
          style={{
            fontSize: 48,
            background: 'white',
            width: '100%',
            height: '100%',
            display: 'flex',
            alignItems: 'center',
            justifyContent: 'center',
          }}
        >
          {post.title}
        </div>
      ),
      {
        ...size,
      }
    )
  }
}

```

Using Node.js runtime with local assets

These examples use the Node.js runtime to fetch a local image from the file system and pass it to the `` `src` attribute, either as a base64 string or an `ArrayBuffer`. Place the local asset relative to the project root, not the example source file.



The screenshot shows a code editor window with the following details:

- File:** app/opengraph-image.tsx
- TypeScript Version:** TypeScript 4.5
- Code Preview:** A preview area shows a large, centered image placeholder with a question mark icon.
- Code Content:**

```

import { ImageResponse } from 'next/og'
import { join } from 'node:path'
import { readFile } from 'node:fs/promises'

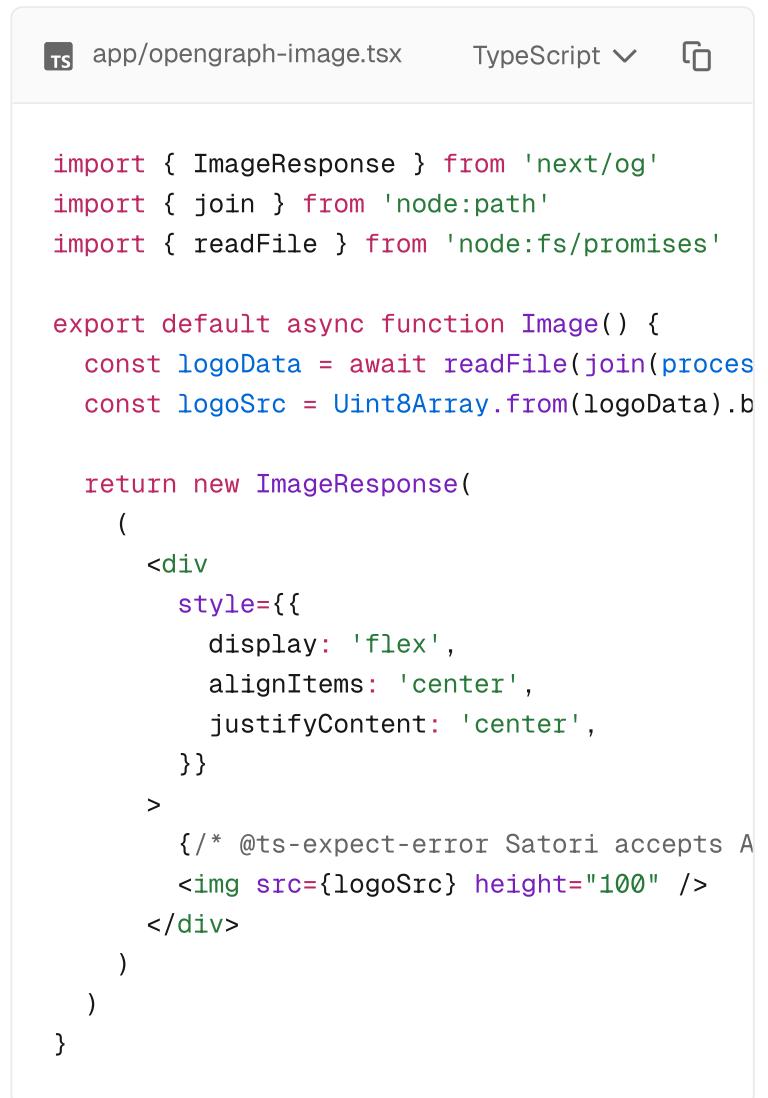
export default async function Image() {
  const logoData = await readFile(join(process.cwd(), 'public/logo.png'))
  const logoSrc = `data:image/png;base64,${logoData.toString('base64')}`

  return new ImageResponse(
    (
      <div
        style={{
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        <img src={logoSrc} height="100" />
      </div>
    )
  )
}

```

```
)  
}
```

Passing an `ArrayBuffer` to the `src` attribute of an `` element is not part of the HTML spec. The rendering engine used by `next/og` supports it, but because TypeScript definitions follow the spec, you need a `@ts-expect-error` directive or similar to use this [feature ↗](#).



```
TS app/opengraph-image.tsx TypeScript ▾
```

```
import { ImageResponse } from 'next/og'
import { join } from 'node:path'
import { readFile } from 'node:fs/promises'

export default async function Image() {
  const logoData = await readFile(join(process.cwd(), 'public/logo.png'))
  const logoSrc = Uint8Array.from(logoData).buffer

  return new ImageResponse(
    (
      <div
        style={{
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        {/* @ts-expect-error Satori accepts A */}
        <img src={logoSrc} height="100" />
      </div>
    )
  )
}
```

Version History

Version	Changes
v13.3.0	opengraph-image and twitter-image introduced.

Was this helpful?  



 **Using App Router**
Features available in /app

 **Latest Version**
15.5.4

robots.txt

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

Static `robots.txt`

 app/robots.txt 

```
User-Agent: *
Allow: /
Disallow: /private/

Sitemap: https://acme.com/sitemap.xml
```

Generate a Robots file

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

Good to know: `robots.js` is a special Route Handlers that is cached by default unless it uses a [Dynamic API](#) or [dynamic config](#) option.

 app/robots.ts  

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

```
export default function robots(): MetadataRoute {
  return {
    rules: [
      {
        userAgent: '*',
        allow: '/',
        disallow: '/private/',
      },
      {
        sitemap: 'https://acme.com/sitemap.xml',
      }
    ]
  }
}
```

Output:

```
User-Agent: *
Allow: /
Disallow: /private/

Sitemap: https://acme.com/sitemap.xml
```

Customizing specific user agents

You can customise how individual search engine bots crawl your site by passing an array of user agents to the `rules` property. For example:



The screenshot shows a code editor window with the following details:

- File name: app/robots.ts
- TypeScript version: 4.5
- Content:

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

export default function robots(): MetadataRoute {
  return {
    rules: [
      {
        userAgent: 'Googlebot',
        allow: ['/'],
        disallow: '/private/',
      },
      {
        userAgent: ['Applebot', 'Bingbot'],
        disallow: ['/'],
      },
    ],
    sitemap: 'https://acme.com/sitemap.xml',
  }
}
```

Output:

```
User-Agent: Googlebot
Allow: /
Disallow: /private/

User-Agent: Applebot
Disallow: /

User-Agent: Bingbot
Disallow: /

Sitemap: https://acme.com/sitemap.xml
```

Robots object

```
type Robots = {
  rules:
  | {
    userAgent?: string | string[]
    allow?: string | string[]
    disallow?: string | string[]
    crawlDelay?: number
  }
  | Array<{
    userAgent: string | string[]
    allow?: string | string[]
    disallow?: string | string[]
    crawlDelay?: number
  }>
  sitemap?: string | string[]
  host?: string
}
```

Version History

Version	Changes
v13.3.0	<code>robots</code> introduced.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

sitemap.xml

`sitemap.(xml|js|ts)` is a special file that matches the [Sitemaps XML format](#) to help search engine crawlers index your site more efficiently.

Sitemap files (.xml)

For smaller applications, you can create a

`sitemap.xml` file and place it in the root of your `app` directory.

```
app/sitemap.xml
```

```
<urlset xmlns="http://www.sitemaps.org/schema">
  <url>
    <loc>https://acme.com</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
    <changefreq>yearly</changefreq>
    <priority>1</priority>
  </url>
  <url>
    <loc>https://acme.com/about</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.8</priority>
  </url>
  <url>
    <loc>https://acme.com/blog</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.5</priority>
  </url>
</urlset>
```

Generating a sitemap using code (.js, .ts)

You can use the `sitemap.(js|ts)` file convention to programmatically **generate** a sitemap by exporting a default function that returns an array of URLs. If using TypeScript, a `Sitemap` type is available.

Good to know: `sitemap.js` is a special Route Handler that is cached by default unless it uses a [Dynamic API](#) or [dynamic config](#) option.



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

export default function sitemap(): MetadataRoute[] {
  return [
    {
      url: 'https://acme.com',
      lastModified: new Date(),
      changeFrequency: 'yearly',
      priority: 1,
    },
    {
      url: 'https://acme.com/about',
      lastModified: new Date(),
      changeFrequency: 'monthly',
      priority: 0.8,
    },
    {
      url: 'https://acme.com/blog',
      lastModified: new Date(),
      changeFrequency: 'weekly',
      priority: 0.5,
    },
  ]
}
```

Output:

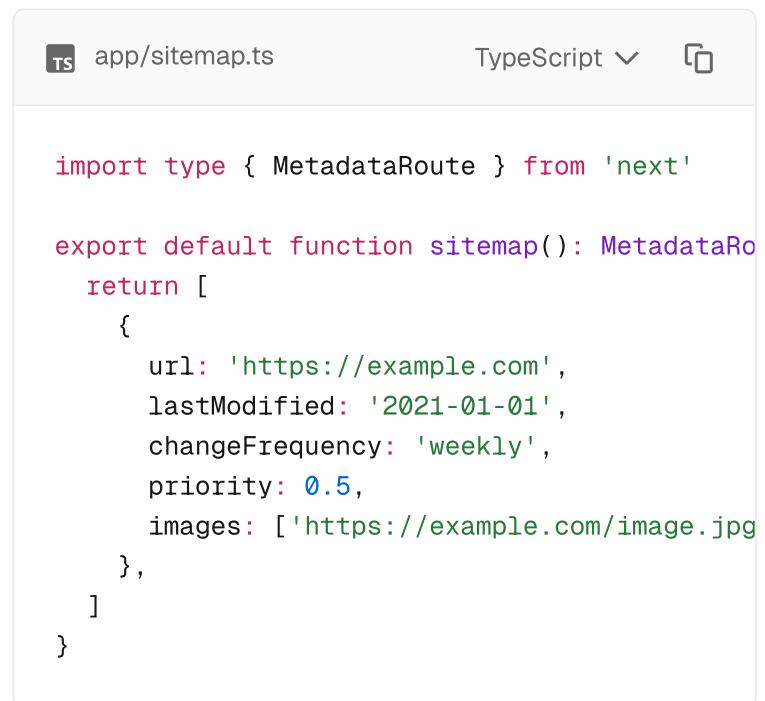


```
<urlset xmlns="http://www.sitemaps.org/schema">
<url>
  <loc>https://acme.com</loc>
  <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  <changefreq>yearly</changefreq>
  <priority>1</priority>
</url>
<url>
  <loc>https://acme.com/about</loc>
  <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  <changefreq>monthly</changefreq>
  <priority>0.8</priority>
</url>
<url>
  <loc>https://acme.com/blog</loc>
  <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  <changefreq>weekly</changefreq>
  <priority>0.5</priority>
</url>
</urlset>
```

```
</url>
<url>
  <loc>https://acme.com/about</loc>
  <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  <changefreq>monthly</changefreq>
  <priority>0.8</priority>
</url>
<url>
  <loc>https://acme.com/blog</loc>
  <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  <changefreq>weekly</changefreq>
  <priority>0.5</priority>
</url>
</urlset>
```

Image Sitemaps

You can use `images` property to create image sitemaps. Learn more details in the [Google Developer Docs ↗](#).



The screenshot shows a code editor window with the file name `app/sitemap.ts`. The code is written in TypeScript and defines a function `sitemap()` that returns an array of `MetadataRoute` objects. Each object has properties for URL, last modified date, change frequency, priority, and a list of images.

```
TS app/sitemap.ts TypeScript ▾ ⌂
import type { MetadataRoute } from 'next'

export default function sitemap(): MetadataRoute[] {
  return [
    {
      url: 'https://example.com',
      lastModified: '2021-01-01',
      changeFrequency: 'weekly',
      priority: 0.5,
      images: ['https://example.com/image.jpg']
    },
  ]
}
```

Output:



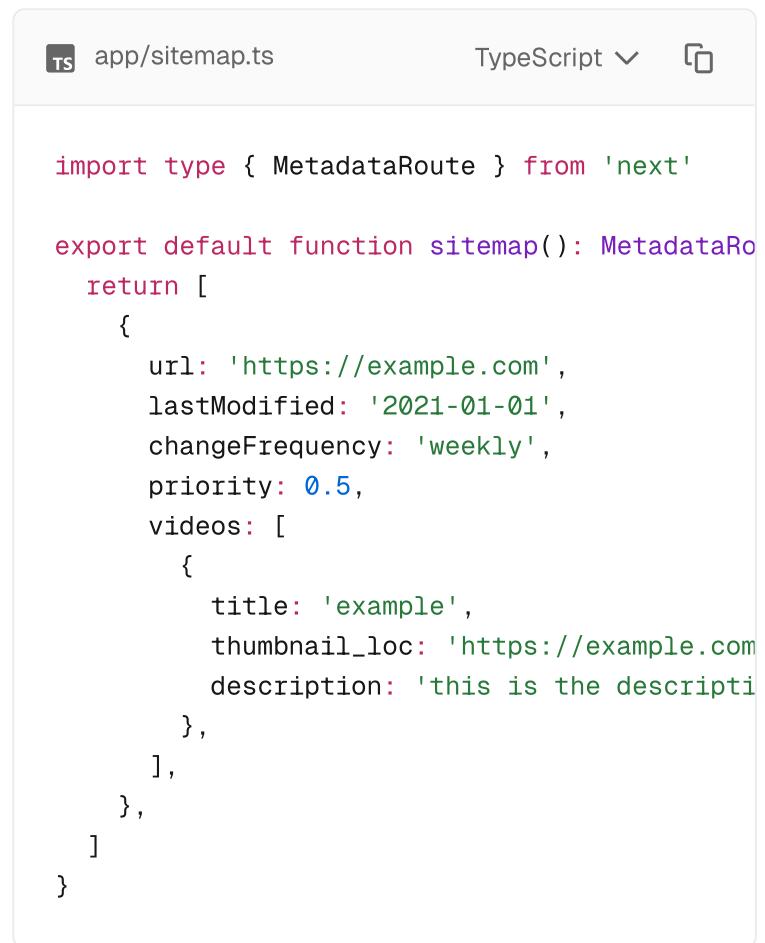
The screenshot shows a file named `acme.com/sitemap.xml`. The content is an XML sitemap with a root element `<urlset>` containing two `<url>` elements. Each `<url>` element contains a `<loc>` tag with the value `https://acme.com/about` and a `<image>` tag with the value `https://acme.com/image.jpg`.

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset
  xmlns="http://www.sitemaps.org/schemas/site
  xmlns:image="http://www.google.com/schemas/
>
<url>
```

```
<loc>https://example.com</loc>
<image:image>
  <image:loc>https://example.com/image.jp
</image:image>
<lastmod>2021-01-01</lastmod>
<changefreq>weekly</changefreq>
<priority>0.5</priority>
</url>
</urlset>
```

Video Sitemaps

You can use `videos` property to create video sitemaps. Learn more details in the [Google Developer Docs ↗](#).



The screenshot shows a code editor window with the following details:

- File name: `app/sitemap.ts`
- Language: TypeScript (indicated by the `.ts` file extension and the "TypeScript" dropdown menu)
- Code content:

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

export default function sitemap(): MetadataRoute {
  return [
    {
      url: 'https://example.com',
      lastModified: '2021-01-01',
      changeFrequency: 'weekly',
      priority: 0.5,
      videos: [
        {
          title: 'example',
          thumbnail_loc: 'https://example.com',
          description: 'this is the descrip
        },
      ],
    },
  ]
}
```

Output:



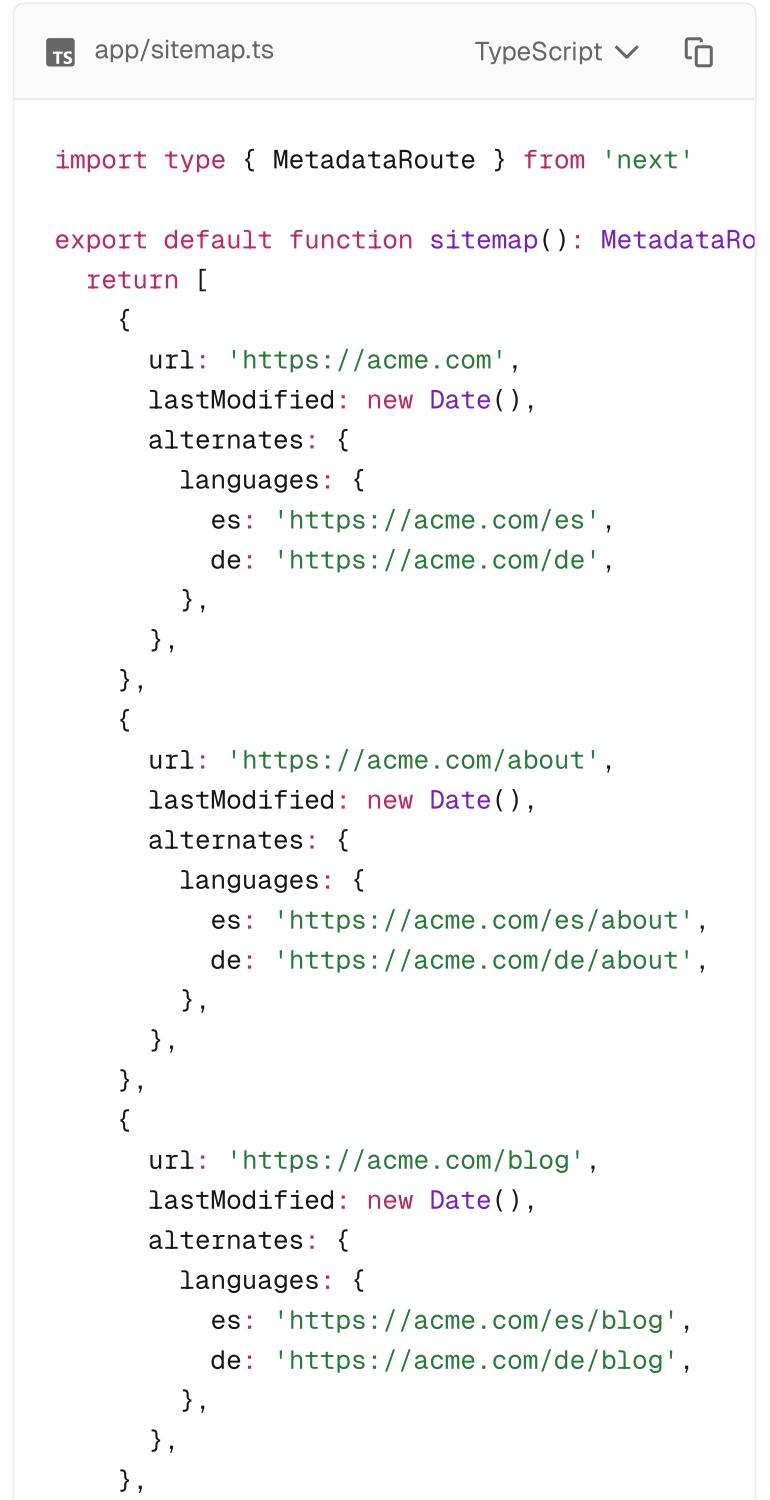
The screenshot shows a code editor window with the following details:

- File name: `acme.com/sitemap.xml`
- Content:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset
  xmlns="http://www.sitemaps.org/schemas/site
  xmlns:video="http://www.google.com/schemas/
>
```

```
<url>
  <loc>https://example.com</loc>
  <video:video>
    <video:title>example</video:title>
    <video:thumbnail_loc>https://example.co
      <video:description>this is the descript
    </video:video>
    <lastmod>2021-01-01</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.5</priority>
  </url>
</urlset>
```

Generate a localized Sitemap



The screenshot shows a code editor interface with the following details:

- File:** app/sitemap.ts
- TypeScript Version:** indicated by a dropdown menu.
- Code Content:** TypeScript code for generating a sitemap with localized routes.

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

export default function sitemap(): MetadataRoute[] {
  return [
    {
      url: 'https://acme.com',
      lastModified: new Date(),
      alternates: {
        languages: {
          es: 'https://acme.com/es',
          de: 'https://acme.com/de',
        },
      },
    },
    {
      url: 'https://acme.com/about',
      lastModified: new Date(),
      alternates: {
        languages: {
          es: 'https://acme.com/es/about',
          de: 'https://acme.com/de/about',
        },
      },
    },
    {
      url: 'https://acme.com/blog',
      lastModified: new Date(),
      alternates: {
        languages: {
          es: 'https://acme.com/es/blog',
          de: 'https://acme.com/de/blog',
        },
      },
    },
  ],
}
```

```
]
}
```

Output:



The screenshot shows a code editor window with a file titled "acme.com/sitemap.xml". The XML code defines a URL set for the website acme.com, including three URLs with their respective hreflangs and lastmod dates.

```
<urlset xmlns="http://www.sitemaps.org/schema">
<url>
  <loc>https://acme.com</loc>
  <xhtml:link
    rel="alternate"
    hreflang="es"
    href="https://acme.com/es"/>
  <xhtml:link
    rel="alternate"
    hreflang="de"
    href="https://acme.com/de"/>
  <lastmod>2023-04-06T15:02:24.021Z</lastmod>
</url>
<url>
  <loc>https://acme.com/about</loc>
  <xhtml:link
    rel="alternate"
    hreflang="es"
    href="https://acme.com/es/about"/>
  <xhtml:link
    rel="alternate"
    hreflang="de"
    href="https://acme.com/de/about"/>
  <lastmod>2023-04-06T15:02:24.021Z</lastmod>
</url>
<url>
  <loc>https://acme.com/blog</loc>
  <xhtml:link
    rel="alternate"
    hreflang="es"
    href="https://acme.com/es/blog"/>
  <xhtml:link
    rel="alternate"
    hreflang="de"
    href="https://acme.com/de/blog"/>
  <lastmod>2023-04-06T15:02:24.021Z</lastmod>
</url>
</urlset>
```

Generating multiple sitemaps

While a single sitemap will work for most applications. For large web applications, you may need to split a sitemap into multiple files.

There are two ways you can create multiple sitemaps:

- By nesting `sitemap.(xml|js|ts)` inside multiple route segments e.g. `app/sitemap.xml` and `app/products/sitemap.xml`.
- By using the `generateSitemaps` function.

For example, to split a sitemap using `generateSitemaps`, return an array of objects with the sitemap `id`. Then, use the `id` to generate the unique sitemaps.



The screenshot shows a code editor window with the following details:

- File name: `app/product/sitemap.ts`
- Language: TypeScript (indicated by the `TS` icon)
- Content:

```
import type { MetadataRoute } from 'next'
import { BASE_URL } from '@/app/lib/constants'

export async function generateSitemaps() {
    // Fetch the total number of products and c
    return [{ id: 0 }, { id: 1 }, { id: 2 }, {
    }

    export default async function sitemap({
        id,
    }: {
        id: number
    }): Promise<MetadataRoute.Sitemap> {
        // Google's limit is 50,000 URLs per sitema
        const start = id * 50000
        const end = start + 50000
        const products = await getProducts(
            `SELECT id, date FROM products WHERE id B
        )
        return products.map((product) => ({
            url: `${BASE_URL}/product/${product.id}`,
            lastModified: product.date,
        }))
    }
}
```

Your generated sitemaps will be available at `/.../sitemap/[id]`. For example,

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

Returns

The default function exported from `sitemap.(xml|ts|js)` should return an array of objects with the following properties:

```
type Sitemap = Array<{
  url: string
  lastModified?: string | Date
  changeFrequency?: 
    | 'always'
    | 'hourly'
    | 'daily'
    | 'weekly'
    | 'monthly'
    | 'yearly'
    | 'never'
  priority?: number
  alternates?: {
    languages?: Languages<string>
  }
}>
```

Version History

Version	Changes
v14.2.0	Add localizations support.
v13.4.14	Add <code>changeFrequency</code> and <code>priority</code> attributes to sitemaps.
v13.3.0	<code>sitemap</code> introduced.

Next Steps

Learn how to use the generateSitemaps function.

generateSite...

Learn how to use
the
generateSiteMap...

Was this helpful?    



Using App Router
Features available in /app



Functions



Latest Version
15.5.4



after

API Reference for
the after function.

cacheLife

Learn how to use
the cacheLife
function to set th...

cacheTag

Learn how to use
the cacheTag
function to...

connection

API Reference for
the connection
function.

cookies

API Reference for
the cookies
function.

draftMode

API Reference for
the draftMode
function.

fetch

API reference for
the extended fetch
function.

forbidden

API Reference for
the forbidden
function.

generateImage

Learn how to generate multiple images in a single file.

generateMetadata

Learn how to add Metadata to your Next.js application.

generateSiteMap

Learn how to use the generateSiteMap function.

generateStaticParams

API reference for the generateStaticParams function.

generateViewport

API Reference for the generateViewport function.

headers

API reference for the headers function.

ImageResponse

API Reference for the ImageResponse function.

NextRequest

API Reference for the NextRequest function.

NextResponse

API Reference for the NextResponse function.

NotFound

API Reference for the NotFound function.

permanentRedirect

redirect

API Reference for
the
permanentRedire...
[View](#)

API Reference for
the redirect
function.
[View](#)

revalidatePath

API Reference for
the revalidatePath
function.
[View](#)

revalidateTag

API Reference for
the revalidateTag
function.
[View](#)

unauthorized

API Reference for
the unauthorized
function.
[View](#)

unstable_ca...

API Reference for
the
unstable_cache...
[View](#)

unstable_no...

API Reference for
the
unstable_noStore...
[View](#)

unstable_reth...

API Reference for
the
unstable_rethrow...
[View](#)

useLinkStatus

API Reference for
the useLinkStatus
hook.
[View](#)

useParams

API Reference for
the useParams
hook.
[View](#)

usePathname

API Reference for
the usePathname
hook.
[View](#)

useReportW...

API Reference for
the
useReportWebVit...
[View](#)

[useRouter](#)

API reference for the useRouter hook.

[useSearchP...](#)

API Reference for the useSearchParams...

[useSelected...](#)

API Reference for the useSelectedLayout...

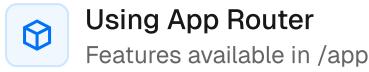
[useSelected...](#)

API Reference for the useSelectedLayout...

[userAgent](#)

The userAgent helper extends the Web Request API...

Was this helpful?    



after

`after` allows you to schedule work to be executed after a response (or prerender) is finished. This is useful for tasks and other side effects that should not block the response, such as logging and analytics.

It can be used in [Server Components](#) (including `generateMetadata`), [Server Actions](#), [Route Handlers](#), and [Middleware](#).

The function accepts a callback that will be executed after the response (or prerender) is finished:

```
TS app/layout.tsx TypeScript ▾ ⌂
import { after } from 'next/server'
// Custom logging function
import { log } from '@/app/utils'

export default function Layout({ children }: {
  after(() => {
    // Execute after the layout is rendered a
    log()
  })
  return <>{children}</>
})
```

Good to know: `after` is not a [Dynamic API](#) and calling it does not cause a route to become dynamic. If it's used within a static page, the callback will execute at build time, or whenever a page is revalidated.

Reference

Parameters

- A callback function which will be executed after the response (or prerender) is finished.

Duration

`after` will run for the platform's default or configured max duration of your route. If your platform supports it, you can configure the timeout limit using the `maxDuration` route segment config.

Good to know

- `after` will be executed even if the response didn't complete successfully. Including when an error is thrown or when `notFound` or `redirect` is called.
- You can use React `cache` to deduplicate functions called inside `after`.
- `after` can be nested inside other `after` calls, for example, you can create utility functions that wrap `after` calls to add additional functionality.

Examples

With request APIs

You can use request APIs such as `cookies` and `headers` inside `after` in Server Actions and

[Route Handlers](#). This is useful for logging activity after a mutation. For example:



```
app/api/route.ts  TypeScript ▾
```

```
import { after } from 'next/server'
import { cookies, headers } from 'next/header'
import { logUserAction } from '@/app/utils'

export async function POST(request: Request)
  // Perform mutation
  // ...

  // Log user activity for analytics
  after(async () => {
    const userAgent = (await headers()).get('u
    const sessionCookie =
      (await cookies().get('session-id'))?.va

    logUserAction({ sessionCookie, userAgent
  })

  return new Response(JSON.stringify({ status
    status: 200,
    headers: { 'Content-Type': 'application/j
  })
}
```

However, you cannot use these request APIs inside `after` in [Server Components](#). This is because Next.js needs to know which part of the tree access the request APIs to support [Partial Prerendering](#), but `after` runs after React's rendering lifecycle.

Platform Support

Deployment Option	Supported
Node.js server	Yes
Docker container	Yes

Deployment Option	Supported
Static export	No
Adapters	Platform-specific

Learn how to [configure `after`](#) when self-hosting Next.js.

- ▶ **Reference: supporting `after` for serverless platforms**

Version History

Version History	Description
v15.1.0	<code>after</code> became stable.
v15.0.0-rc	<code>unstable_after</code> introduced.

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

cacheLife

This feature is currently available in the canary channel and subject to change. Try it out by upgrading Next.js, and share your feedback on GitHub.

The `cacheLife` function is used to set the cache lifetime of a function or component. It should be used alongside the `use cache` directive, and within the scope of the function or component.

Usage

To use `cacheLife`, enable the `cacheComponents` flag in your `next.config.js` file:

```
TS next.config.ts TypeScript ▾   
  
import type { NextConfig } from 'next'  
  
const nextConfig: NextConfig = {  
  experimental: {  
    cacheComponents: true,  
  },  
}  
  
export default nextConfig
```

Then, import and invoke the `cacheLife` function within the scope of the function or component:

```
TS app/page.tsx TypeScript ▾   
  
// code here
```

```
'use cache'  
import { unstable_cacheLife as cacheLife } fr  
  
export default async function Page() {  
  cacheLife('hours')  
  return <div>Page</div>  
}
```

Reference

Default cache profiles

Next.js provides a set of named cache profiles modeled on various timescales. If you don't specify a cache profile in the `cacheLife` function alongside the `use cache` directive, Next.js will automatically apply the `default` cache profile.

However, we recommend always adding a cache profile when using the `use cache` directive to explicitly define caching behavior.

Profile	stale	revalidate	expire	Description
default	5 minutes	15 minutes	1 year	Default profile, suitable for content that doesn't need frequent updates
seconds	0	1 second	1 second	For rapidly changing content requiring near real-time updates

Profile	stale	revalidate	expire	Description
	minutes	5 minutes	1 minute	1 hour
	hours	5 minutes	1 hour	1 day
	days	5 minutes	1 day	1 week
	weeks	5 minutes	1 week	30 days
	max	5 minutes	30 days	1 year

The string values used to reference cache profiles don't carry inherent meaning; instead they serve as semantic labels. This allows you to better understand and manage your cached content within your codebase.

Good to know: Updating the `staleTimes` and `expireTime` config options also updates the `stale` and `expire` properties of the `default` cache profile.

Custom cache profiles

You can configure custom cache profiles by adding them to the `cacheLife` option in your `next.config.ts` file.

Cache profiles are objects that contain the following properties:

Property	Value	Description	Requirement
<code>stale</code>	<code>number</code>	Duration the client should cache a value without checking the server.	Optional
<code>revalidate</code>	<code>number</code>	Frequency at which the cache should refresh on the server; stale values may be served while revalidating.	Optional
<code>expire</code>	<code>number</code>	Maximum duration for which a value can remain stale before switching to dynamic fetching; must be longer than <code>revalidate</code> .	Optional - Must be longer than <code>revalidate</code> .

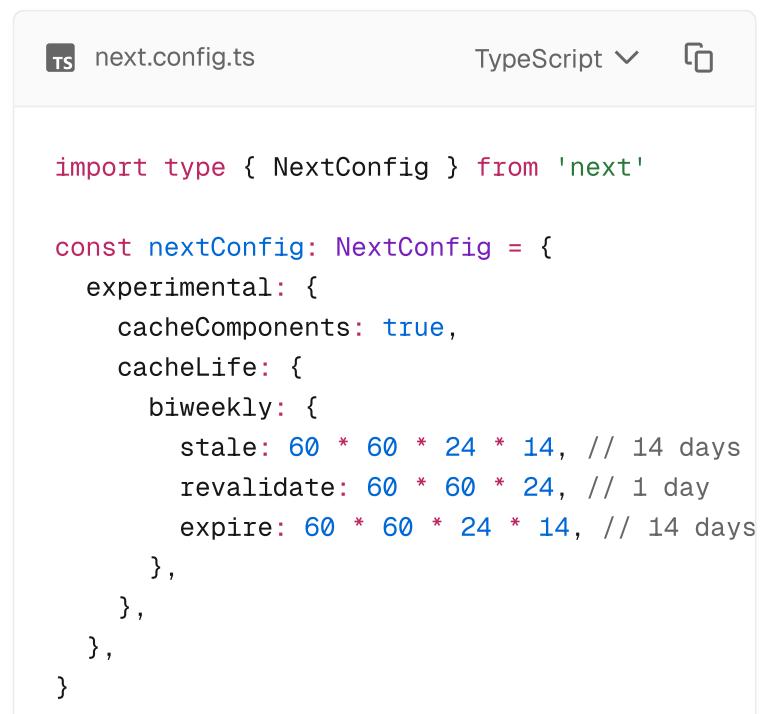
The "stale" property differs from the `staleTimes` setting in that it specifically controls client-side router caching. While `staleTimes` is a global setting that affects all instances of both dynamic and static data, the `cacheLife` configuration allows you to define "stale" times on a per-function or per-route basis.

Good to know: The "stale" property does not set the `Cache-control: max-age` header. It instead controls the client-side router cache.

Examples

Defining reusable cache profiles

You can create a reusable cache profile by defining them in your `next.config.ts` file. Choose a name that suits your use case and set values for the `stale`, `revalidate`, and `expire` properties. You can create as many custom cache profiles as needed. Each profile can be referenced by its name as a string value passed to the `cacheLife` function.



A screenshot of a code editor interface. At the top, there's a toolbar with a TypeScript icon (TS), the file name "next.config.ts", a "TypeScript" dropdown, and a refresh icon. The main area contains the following TypeScript code:

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

const nextConfig: NextConfig = {
  experimental: {
    cacheComponents: true,
    cacheLife: {
      biweekly: {
        stale: 60 * 60 * 24 * 14, // 14 days
        revalidate: 60 * 60 * 24, // 1 day
        expire: 60 * 60 * 24 * 14, // 14 days
      },
    },
  },
}
```

```
module.exports = nextConfig
```

The example above caches for 14 days, checks for updates daily, and expires the cache after 14 days. You can then reference this profile throughout your application by its name:

```
TS app/page.tsx
```

```
'use cache'
import { unstable_cacheLife as cacheLife } from 'next/cache'

export default async function Page() {
  cacheLife('biweekly')
  return <div>Page</div>
}
```

Overriding the default cache profiles

While the default cache profiles provide a useful way to think about how fresh or stale any given part of cacheable output can be, you may prefer different named profiles to better align with your applications caching strategies.

You can override the default named cache profiles by creating a new configuration with the same name as the defaults.

The example below shows how to override the default “days” cache profile:

```
TS next.config.ts
```

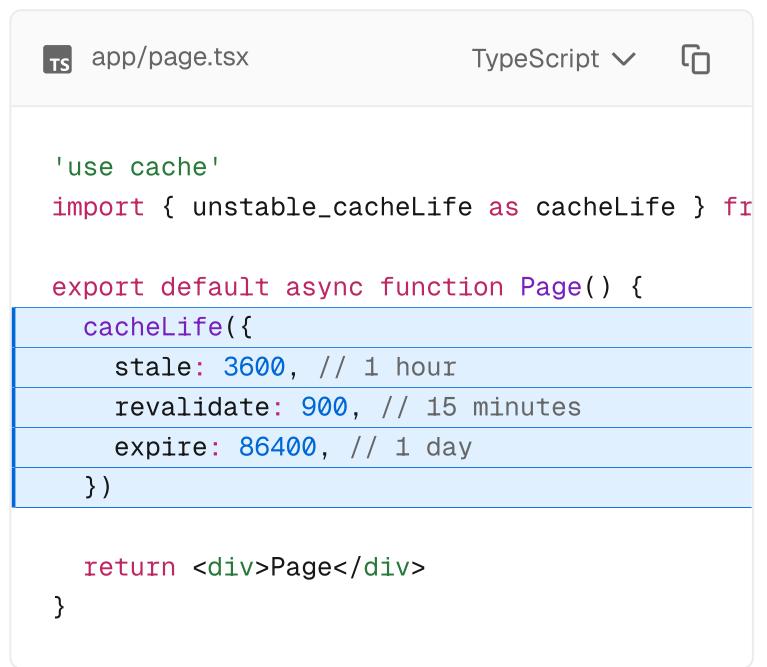
```
const nextConfig = {
  experimental: {
    cacheComponents: true,
    cacheLife: {
      days: {
        stale: 3600, // 1 hour
        revalidate: 900, // 15 minutes
        expire: 86400, // 1 day
      }
    }
  }
}
```

```
        },
      },
    },
  }

module.exports = nextConfig
```

Defining cache profiles inline

For specific use cases, you can set a custom cache profile by passing an object to the `cacheLife` function:



```
TS app/page.tsx TypeScript ▾ ⌂

'use cache'
import { unstable_cacheLife as cacheLife } from 'next/cache'

export default async function Page() {
  cacheLife({
    stale: 3600, // 1 hour
    revalidate: 900, // 15 minutes
    expire: 86400, // 1 day
  })
}

return <div>Page</div>
}
```

This inline cache profile will only be applied to the function or file it was created in. If you want to reuse the same profile throughout your application, you can [add the configuration](#) to the `cacheLife` property of your `next.config.ts` file.

Nested usage of `use cache` and `cacheLife`

When defining multiple caching behaviors in the same route or component tree, if the inner caches specify their own `cacheLife` profile, the outer cache will respect the shortest cache duration among them. **This applies only if the outer cache**

does not have its own explicit `cacheLife` profile defined.

For example, if you add the `use cache` directive to your page, without specifying a cache profile, the default cache profile will be applied implicitly (`cacheLife("default")`). If a component imported into the page also uses the `use cache` directive with its own cache profile, the outer and inner cache profiles are compared, and shortest duration set in the profiles will be applied.

```
TS app/components/parent.tsx

// Parent component
import { unstable_cacheLife as cacheLife } from 'next/cache'
import { ChildComponent } from './child'

export async function ParentComponent() {
  'use cache'
  cacheLife('days')

  return (
    <div>
      <ChildComponent />
    </div>
  )
}
```

And in a separate file, we defined the Child component that was imported:

```
TS app/components/child.tsx

// Child component
import { unstable_cacheLife as cacheLife } from 'next/cache'

export async function ChildComponent() {
  'use cache'
  cacheLife('hours')
  return <div>Child Content</div>

  // This component's cache will respect the
}
```

Related

View related API references.

cacheCompo...

Learn how to enable the cacheComponent...

use cache

Learn how to use the use cache directive to cache...

revalidateTag

API Reference for the revalidateTag function.

cacheTag

Learn how to use the cacheTag function to...

Was this helpful?





Using App Router
Features available in /app



cacheTag



Latest Version
15.5.4



This feature is currently available in the canary channel and subject to change. Try it out by upgrading Next.js, and share your feedback on GitHub.

The `cacheTag` function allows you to tag cached data for on-demand invalidation. By associating tags with cache entries, you can selectively purge or revalidate specific cache entries without affecting other cached data.

Usage

To use `cacheTag`, enable the `cacheComponents` flag in your `next.config.js` file:

next.config.ts TypeScript ▾

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

const nextConfig: NextConfig = {
  experimental: {
    cacheComponents: true,
  },
}

export default nextConfig
```

The `cacheTag` function takes one or more string values.

app/data.ts TypeScript ▾

```
import { unstable_cacheTag as cacheTag } from 'next/cache'

export async function getData() {
  'use cache'
  cacheTag('my-data')
  const data = await fetch('/api/data')
  return data
}
```

You can then purge the cache on-demand using [revalidateTag](#) API in another function, for example, a [route handler](#) or [Server Action](#):

```
TS app/action.ts TypeScript ▾ ⌂

'use server'

import { revalidateTag } from 'next/cache'

export default async function submit() {
  await addPost()
  revalidateTag('my-data')
}
```

Good to know

- **Idempotent Tags:** Applying the same tag multiple times has no additional effect.
- **Multiple Tags:** You can assign multiple tags to a single cache entry by passing multiple string values to `cacheTag`.

```
cacheTag('tag-one', 'tag-two')
```

Examples

Tagging components or functions

Tag your cached data by calling `cacheTag` within a cached function or component:

```
TS app/components/bookings.t... TypeScript ▾ ⌂

import { unstable_cacheTag as cacheTag } from

interface BookingsProps {
  type: string
}

export async function Bookings({ type = 'hair' }) {
  'use cache'
  cacheTag('bookings-data')

  async function getBookingsData() {
    const data = await fetch(`api/bookings?${type}`)
    return data
  }

  return //...
}
```

Creating tags from external data

You can use the data returned from an async function to tag the cache entry.

```
TS app/components/bookings.t... TypeScript ▾ ⌂

import { unstable_cacheTag as cacheTag } from

interface BookingsProps {
  type: string
}

export async function Bookings({ type = 'hair' }) {
  async function getBookingsData() {
    'use cache'
    const data = await fetch(`api/bookings?${type}`)
    cacheTag('bookings-data', data.id)
    return data
  }

  return //...
}
```

Invalidating tagged cache

Using `revalidateTag`, you can invalidate the cache for a specific tag when needed:

```
TS app/actions.ts TypeScript ▾ ⌂

use server

import { revalidateTag } from 'next/cache'

export async function updateBookings() {
  await updateBookingData()
  revalidateTag('bookings-data')
}
```

Related

View related API references.

cacheCompo...

Learn how to enable the cacheComponent...

use cache

Learn how to use the use cache directive to cache...

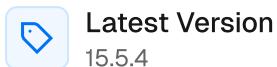
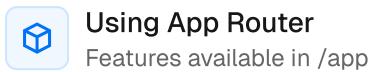
revalidateTag

API Reference for the revalidateTag function.

cacheLife

Learn how to use the cacheLife function to set th...

Was this helpful?    



connection

The `connection()` function allows you to indicate rendering should wait for an incoming user request before continuing.

It's useful when a component doesn't use [Dynamic APIs](#), but you want it to be dynamically rendered at runtime and not statically rendered at build time. This usually occurs when you access external information that you intentionally want to change the result of a render, such as `Math.random()` or `new Date()`.

```
TS app/page.tsx TypeScript ▾
```

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

export default async function Page() {
  await connection()
  // Everything below will be excluded from p
  const rand = Math.random()
  return <span>{rand}</span>
}
```

Reference

Type

```
function connection(): Promise<void>
```

Parameters

- The function does not accept any parameters.

Returns

- The function returns a `void` Promise. It is not meant to be consumed.
-

Good to know

- `connection` replaces `unstable_noStore` to better align with the future of Next.js.
- The function is only necessary when dynamic rendering is required and common Dynamic APIs are not used.

Version History

Version	Changes
v15.0.0	<code>connection</code> stabilized.
v15.0.0-RC	<code>connection</code> introduced.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

cookies

`cookies` is an **async** function that allows you to read the HTTP incoming request cookies in [Server Components](#), and read/write outgoing request cookies in [Server Actions](#) or [Route Handlers](#).

TS app/page.tsx

TypeScript ▾

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

export default async function Page() {
  const cookieStore = await cookies()
  const theme = cookieStore.get('theme')
  return '...'
}
```

Reference

Methods

The following methods are available:

Method	Return Type	Description
<code>get('name')</code>	Object	Accepts a cookie name and returns an object with the name and value.
<code>getAll()</code>	Array of objects	Returns a list of all the cookies with a matching name.

Method	Type	Description
<code>has('name')</code>	Boolean	Accepts a cookie name and returns a boolean based on if the cookie exists.
<code>set(name, value, options)</code>	-	Accepts a cookie name, value, and options and sets the outgoing request cookie.
<code>delete(name)</code>	-	Accepts a cookie name and deletes the cookie.
<code>clear()</code>	-	Deletes all cookies.
<code>toString()</code>	String	Returns a string representation of the cookies.

Options

When setting a cookie, the following properties from the `options` object are supported:

Option	Type	Description
<code>name</code>	String	Specifies the name of the cookie.
<code>value</code>	String	Specifies the value to be stored in the cookie.
<code>expires</code>	Date	Defines the exact date when the cookie will expire.
<code>maxAge</code>	Number	Sets the cookie's lifespan in seconds.
<code>domain</code>	String	Specifies the domain where the cookie is

Option	Type	Description
		available.
path	String, default: '/'	Limits the cookie's scope to a specific path within the domain.
secure	Boolean	Ensures the cookie is sent only over HTTPS connections for added security.
httpOnly	Boolean	Restricts the cookie to HTTP requests, preventing client-side access.
sameSite	Boolean, 'lax', 'strict', 'none'	Controls the cookie's cross-site request behavior.
priority	String ("low", "medium", "high")	Specifies the cookie's priority
partitioned	Boolean	Indicates whether the cookie is partitioned .

The only option with a default value is `path`.

To learn more about these options, see the [MDN docs](#).

Good to know

- `cookies` is an **asynchronous** function that returns a promise. You must use `async/await` or React's [use](#) function to access cookies.
- In version 14 and earlier, `cookies` was a synchronous function. To help with

backwards compatibility, you can still access it synchronously in Next.js 15, but this behavior will be deprecated in the future.

- `cookies` is a [Dynamic API](#) whose returned values cannot be known ahead of time. Using it in a layout or page will opt a route into [dynamic rendering](#).
- The `.delete` method can only be called:
 - In a [Server Action](#) or [Route Handler](#).
 - If it belongs to the same domain from which `.set` is called. For wildcard domains, the specific subdomain must be an exact match. Additionally, the code must be executed on the same protocol (HTTP or HTTPS) as the cookie you want to delete.
- HTTP does not allow setting cookies after streaming starts, so you must use `.set` in a [Server Action](#) or [Route Handler](#).

Understanding Cookie Behavior in Server Components

When working with cookies in Server Components, it's important to understand that cookies are fundamentally a client-side storage mechanism:

- **Reading cookies** works in Server Components because you're accessing the cookie data that the client's browser sends to the server in the HTTP request headers.
- **Setting cookies** cannot be done directly in a Server Component, even when using a Route Handler or Server Action. This is because

cookies are actually stored by the browser, not the server.

The server can only send instructions (via `Set-Cookie` headers) to tell the browser to store cookies - the actual storage happens on the client side. This is why cookie operations that modify state (`.set`, `.delete`, `.clear`) must be performed in a Route Handler or Server Action where the response headers can be properly set.

Understanding Cookie Behavior in Server Actions

After you set or delete a cookie in a Server Action, Next.js re-renders the current page and its layouts on the server so the UI reflects the new cookie value. See the [Caching guide](#).

The UI is not unmounted, but effects that depend on data coming from the server will re-run.

To refresh cached data too, call `revalidatePath` or `revalidateTag` inside the action.

Examples

Getting a cookie

You can use the

`(await cookies()).get('name')` method to get a single cookie:

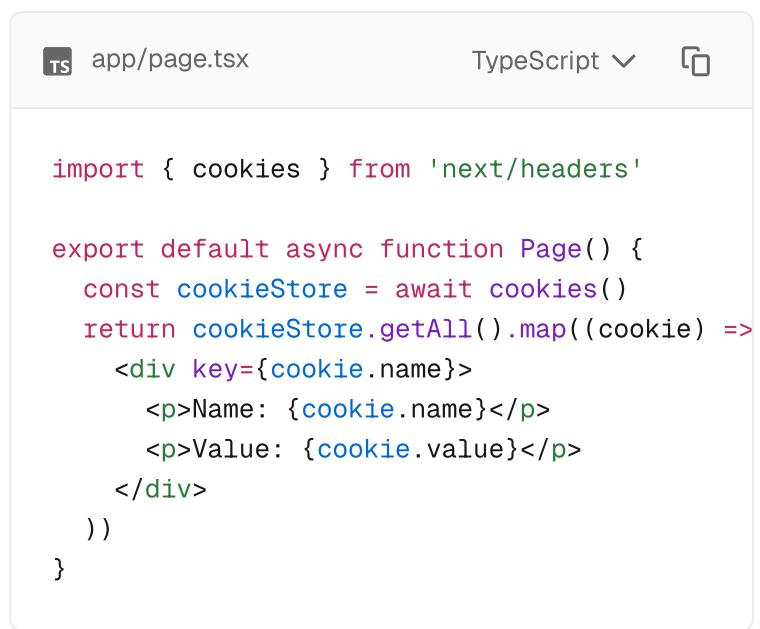


```
TS app/page.tsx TypeScript ▾   
  
import { cookies } from 'next/headers'
```

```
export default async function Page() {
  const cookieStore = await cookies()
  const theme = cookieStore.get('theme')
  return '...'
}
```

Getting all cookies

You can use the `(await cookies()).getAll()` method to get all cookies with a matching name. If `name` is unspecified, it returns all the available cookies.



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

export default async function Page() {
  const cookieStore = await cookies()
  return cookieStore.getAll().map((cookie) =>
    <div key={cookie.name}>
      <p>Name: {cookie.name}</p>
      <p>Value: {cookie.value}</p>
    </div>
  )
}
```

Setting a cookie

You can use the

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

method in a [Server Action](#) or [Route Handler](#) to set a cookie. The `options` object is optional.



```
'use server'

import { cookies } from 'next/headers'

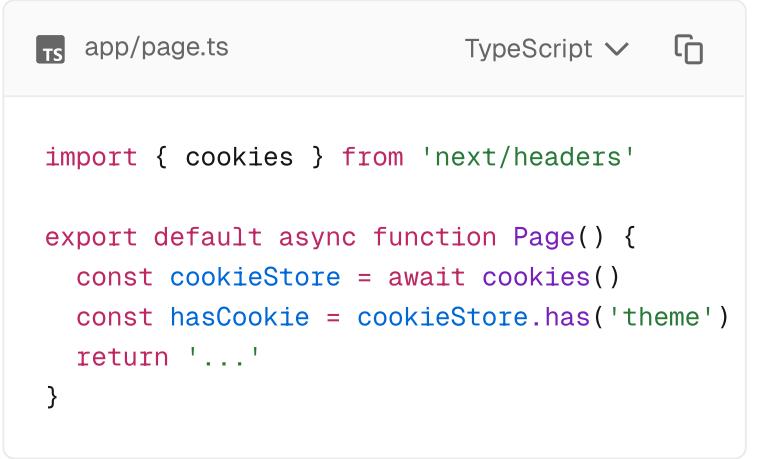
export async function create(data) {
  const cookieStore = await cookies()

  cookieStore.set('name', 'lee')
}
```

```
// or  
cookieStore.set('name', 'lee', { secure: true }  
// or  
cookieStore.set({  
  name: 'name',  
  value: 'lee',  
  httpOnly: true,  
  path: '/',  
})  
}
```

Checking if a cookie exists

You can use the `(await cookies()).has(name)` method to check if a cookie exists:



```
TS app/page.ts TypeScript ▾
```

```
import { cookies } from 'next/headers'  
  
export default async function Page() {  
  const cookieStore = await cookies()  
  const hasCookie = cookieStore.has('theme')  
  return '...'  
}
```

Deleting cookies

There are three ways you can delete a cookie.

Using the `delete()` method:



```
TS app/actions.ts TypeScript ▾
```

```
'use server'  
  
import { cookies } from 'next/headers'  
  
export async function delete(data) {  
  (await cookies()).delete('name')  
}
```

Setting a new cookie with the same name and an empty value:



```
'use server'

import { cookies } from 'next/headers'

export async function delete(data) {
  (await cookies()).set('name', '')
}
```

Setting the `maxAge` to 0 will immediately expire a cookie. `maxAge` accepts a value in seconds.



```
'use server'

import { cookies } from 'next/headers'

export async function delete(data) {
  (await cookies()).set('name', 'value', { ma
}
```

Version History

Version	Changes
v15.0.0-RC	<code>cookies</code> is now an async function. A codemod is available.
v13.0.0	<code>cookies</code> introduced.

Was this helpful?

Using App Router
Features available in /app

Latest Version
15.5.4

draftMode

`draftMode` is an **async** function allows you to enable and disable **Draft Mode**, as well as check if Draft Mode is enabled in a [Server Component](#).

TS app/page.ts

TypeScript



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

export default async function Page() {
  const { isEnabled } = await draftMode()
}
```

Reference

The following methods and properties are available:

Method	Description
<code>isEnabled</code>	A boolean value that indicates if Draft Mode is enabled.
<code>enable()</code>	Enables Draft Mode in a Route Handler by setting a cookie (<code>__prerender_bypass</code>).
<code>disable()</code>	Disables Draft Mode in a Route Handler by deleting a cookie.

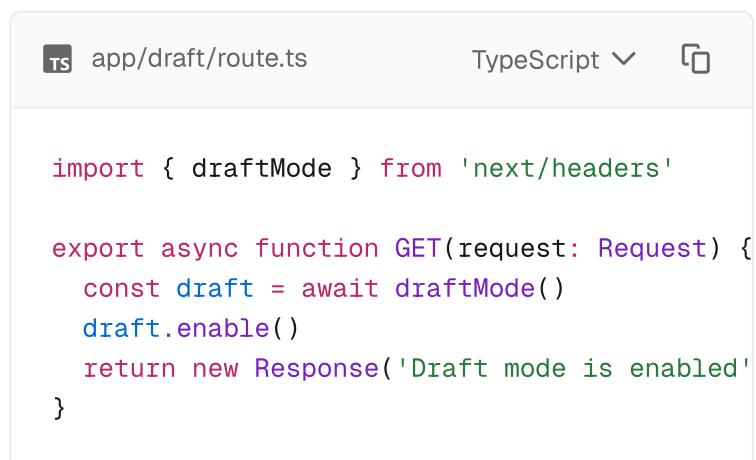
Good to know

- `draftMode` is an **asynchronous** function that returns a promise. You must use `async/await` or React's `use` ↗ function.
 - In version 14 and earlier, `draftMode` was a synchronous function. To help with backwards compatibility, you can still access it synchronously in Next.js 15, but this behavior will be deprecated in the future.
- A new bypass cookie value will be generated each time you run `next build`. This ensures that the bypass cookie can't be guessed.
- To test Draft Mode locally over HTTP, your browser will need to allow third-party cookies and local storage access.

Examples

Enabling Draft Mode

To enable Draft Mode, create a new [Route Handler](#) and call the `enable()` method:



The screenshot shows a code editor window with the following details:

- File path: `app/draft/route.ts`
- TypeScript version: `TypeScript` (dropdown menu)
- Code content:

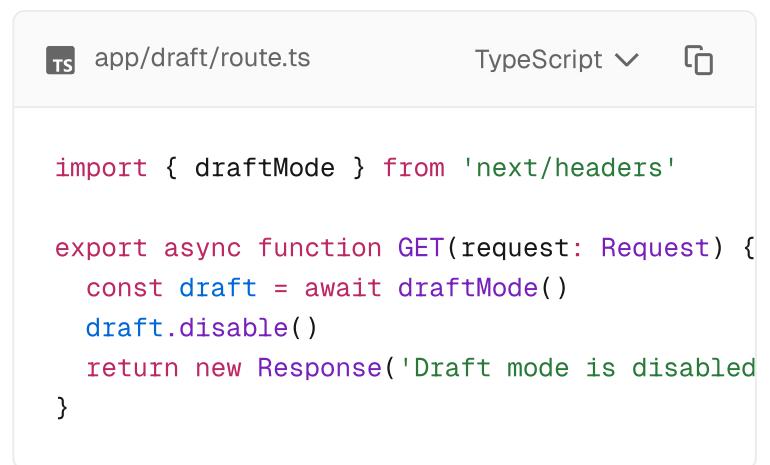
```
import { draftMode } from 'next/headers'

export async function GET(request: Request) {
  const draft = await draftMode()
  draft.enable()
  return new Response('Draft mode is enabled')
}
```

Disabling Draft Mode

By default, the Draft Mode session ends when the browser is closed.

To disable Draft Mode manually, call the `disable()` method in your [Route Handler](#):



The screenshot shows a code editor window with a TypeScript file named `route.ts`. The code imports `draftMode` from `'next/headers'` and defines a `GET` handler function. Inside the function, it calls `const draft = await draftMode()`, then `draft.disable()`, and finally returns a `Response` with the message 'Draft mode is disabled'.

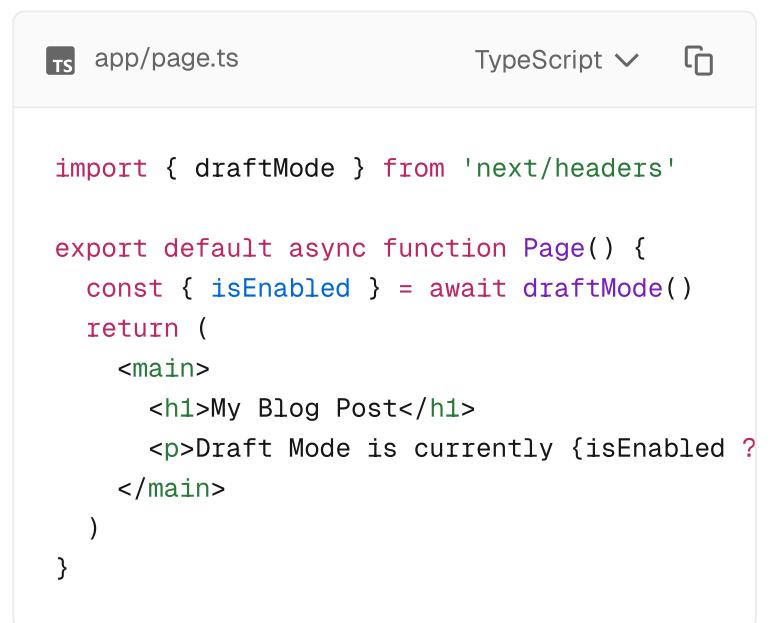
```
import { draftMode } from 'next/headers'

export async function GET(request: Request) {
  const draft = await draftMode()
  draft.disable()
  return new Response('Draft mode is disabled')
}
```

Then, send a request to invoke the Route Handler. If calling the route using the [`<Link>` component](#), you must pass `prefetch={false}` to prevent accidentally deleting the cookie on prefetch.

Checking if Draft Mode is enabled

You can check if Draft Mode is enabled in a Server Component with the `isEnabled` property:



The screenshot shows a code editor window with a TypeScript file named `page.ts`. The code imports `draftMode` from `'next/headers'` and defines a `Page` component. It uses the `isEnabled` property of the `draftMode` object to conditionally render a `<p>` element with the message 'Draft Mode is currently {isEnabled ? ...}'.

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

export default async function Page() {
  const { isEnabled } = await draftMode()
  return (
    <main>
      <h1>My Blog Post</h1>
      <p>Draft Mode is currently {isEnabled ? ...}</p>
    </main>
  )
}
```

Version History

Version Changes

v15.0.0-
RC

`draftMode` is now an async function. A
[codemod](#) is available.

v13.4.0

`draftMode` introduced.

Next Steps

Learn how to use Draft Mode with this step-by-step guide.

Draft Mode

Next.js has draft mode to toggle between static...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

fetch

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

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

You can call `fetch` with `async` and `await` directly within Server Components.

TS app/page.tsx TypeScript ▾

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

fetch(url, options)

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

options.cache

Configure how the request should interact with Next.js [Data Cache](#).

```
fetch(`https://...`, { cache: 'force-cache' |
```

- **auto no cache** (default): Next.js fetches the resource from the remote server on every request in development, but will fetch once during `next build` because the route will be statically prerendered. If [Dynamic APIs](#) are detected on the route, Next.js will fetch the resource on every request.
- **no-store**: Next.js fetches the resource from the remote server on every request, even if Dynamic APIs are not detected on the route.
- **force-cache**: Next.js looks for a matching request in its Data Cache.
 - If there is a match and it is fresh, it will be returned from the cache.
 - If there is no match or a stale match, Next.js will fetch the resource from the remote server and update the cache with the downloaded resource.

options.next.revalidate

```
fetch(`https://...`, { next: { revalidate: fa
```

Set the cache lifetime of a resource (in seconds). [Data Cache](#).

- **false** - Cache the resource indefinitely. Semantically equivalent to `revalidate: Infinity`. The HTTP cache may evict older resources over time.

- `0` - Prevent the resource from being cached.
- `number` - (in seconds) Specify the resource should have a cache lifetime of at most `n` seconds.

Good to know:

- If an individual `fetch()` request sets a `revalidate` number lower than the `default revalidate` of a route, the whole route revalidation interval will be decreased.
- If two fetch requests with the same URL in the same route have different `revalidate` values, the lower value will be used.
- Conflicting options such as `{ revalidate: 3600, cache: 'no-store' }` are not allowed, both will be ignored, and in development mode a warning will be printed to the terminal.

options.next.tags

```
fetch(`https://...`), { next: { tags: ['collect...
```

Set the cache tags of a resource. Data can then be revalidated on-demand using `revalidateTag`. The max length for a custom tag is 256 characters and the max tag items is 128.

Troubleshooting

Fetch default `auto no store` and `cache: 'no-store'` not showing fresh data in development

Next.js caches `fetch` responses in Server Components across Hot Module Replacement (HMR) in local development for faster responses and to reduce costs for billed API calls.

By default, the [HMR cache](#) applies to all fetch requests, including those with the default `auto no cache` and `cache: 'no-store'` option. This means uncached requests will not show fresh data between HMR refreshes. However, the cache will be cleared on navigation or full-page reloads.

See the [serverComponentsHmrCache](#) docs for more information.

Hard refresh and caching in development

In development mode, if the request includes the `cache-control: no-cache` header, `options.cache`, `options.next.revalidate`, and `options.next.tags` are ignored, and the `fetch` request is served from the source.

Browsers typically include `cache-control: no-cache` when the cache is disabled in developer tools or during a hard refresh.

Version History

Version	Changes
v13.0.0	<code>fetch</code> introduced.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

forbidden

 This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

The `forbidden` function throws an error that renders a Next.js 403 error page. It's useful for handling authorization errors in your application. You can customize the UI using the [`forbidden.js` file](#).

To start using `forbidden`, enable the experimental `authInterrupts` configuration option in your `next.config.js` file:

```
TS next.config.ts TypeScript ▾   
  
import type { NextConfig } from 'next'  
  
const nextConfig: NextConfig = {  
  experimental: {  
    authInterrupts: true,  
  },  
}  
  
export default nextConfig
```

`forbidden` can be invoked in [Server Components](#), [Server Actions](#), and [Route Handlers](#).

```
TS app/auth/page.tsx TypeScript ▾   
  
import { verifySession } from '@/app/lib/dal'  
import { forbidden } from 'next/navigation'  
  
export default async function AdminPage() {
```

```
const session = await verifySession()

// Check if the user has the 'admin' role
if (session.role !== 'admin') {
  forbidden()
}

// Render the admin page for authorized use
return <></>
}
```

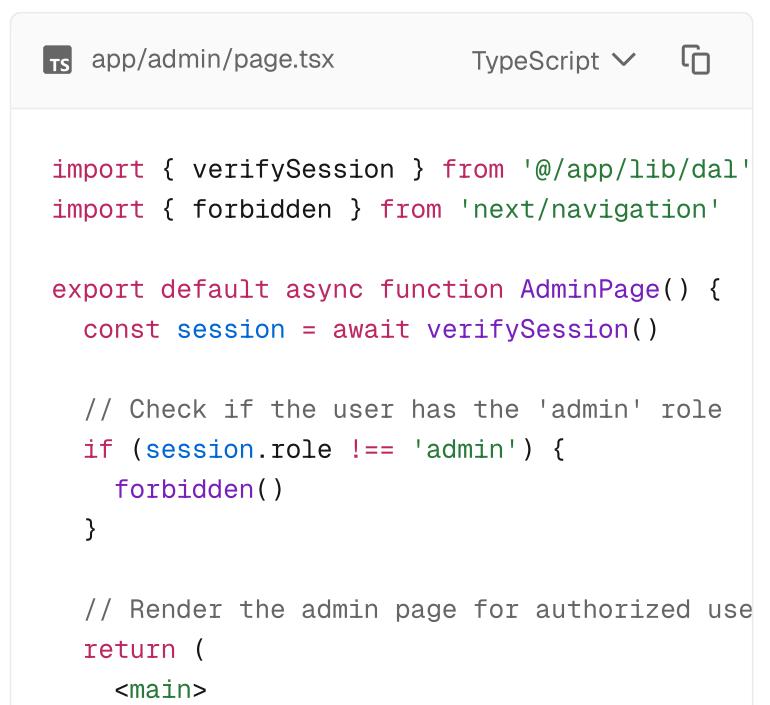
Good to know

- The `forbidden` function cannot be called in the [root layout](#).

Examples

Role-based route protection

You can use `forbidden` to restrict access to certain routes based on user roles. This ensures that users who are authenticated but lack the required permissions cannot access the route.



The screenshot shows a code editor window with a TypeScript file named `app/admin/page.tsx`. The file contains the following code:

```
import { verifySession } from '@/app/lib/dal'
import { forbidden } from 'next/navigation'

export default async function AdminPage() {
  const session = await verifySession()

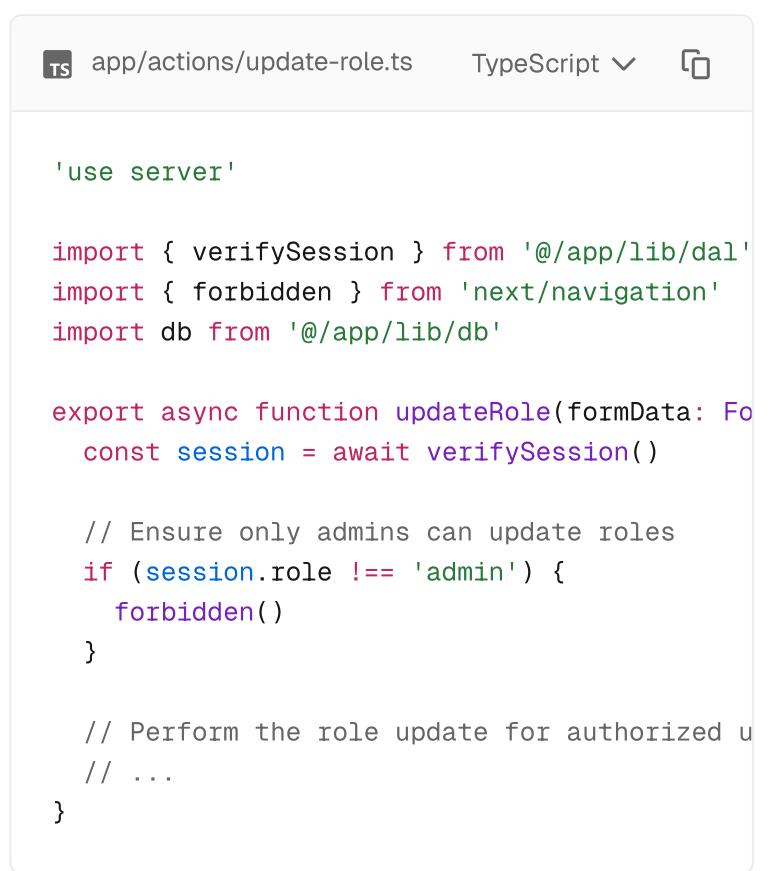
  // Check if the user has the 'admin' role
  if (session.role !== 'admin') {
    forbidden()
  }

  // Render the admin page for authorized use
  return (
    <main>
```

```
<h1>Admin Dashboard</h1>
<p>Welcome, {session.user.name}!</p>
</main>
)
}
```

Mutations with Server Actions

When implementing mutations in Server Actions, you can use `forbidden` to only allow users with a specific role to update sensitive data.



The screenshot shows a code editor window with the following details:

- File path: app/actions/update-role.ts
- Language: TypeScript
- Code content:

```
'use server'

import { verifySession } from '@/app/lib/dal'
import { forbidden } from 'next/navigation'
import db from '@/app/lib/db'

export async function updateRole(formData: FormData) {
  const session = await verifySession()

  // Ensure only admins can update roles
  if (session.role !== 'admin') {
    forbidden()
  }

  // Perform the role update for authorized users
  // ...
}
```

Version History

Version	Changes
v15.1.0	<code>forbidden</code> introduced.

Next Steps

forbidden.js

API reference for
the forbidden.js
special file.

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

generateImageMetadata

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

Parameters

`generateImageMetadata` function accepts the following parameters:

`params` (optional)

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

icon.tsx

TypeScript

```
export function generateImageMetadata({
  params,
}: {
  params: { slug: string }
}) {
  // ...
}
```

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

Returns

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

Image Metadata

Object	Type
<code>id</code>	<code>string</code> (required)
<code>alt</code>	<code>string</code>
<code>size</code>	<code>{ width: number; height: number }</code>
<code>contentType</code>	<code>string</code>

```
TS icon.tsx TypeScript ▾ ⌂
import { ImageResponse } from 'next/og'

export function generateImageMetadata() {
  return [
    {
      contentType: 'image/png',

```

```

        size: { width: 48, height: 48 },
        id: 'small',
    },
    {
        contentType: 'image/png',
        size: { width: 72, height: 72 },
        id: 'medium',
    },
]
}
}

export default function Icon({ id }: { id: string }): { id: string; url: string } {
    return new ImageResponse(
        (
            <div
                style={{
                    width: '100%',
                    height: '100%',
                    display: 'flex',
                    alignItems: 'center',
                    justifyContent: 'center',
                    fontSize: 88,
                    background: '#000',
                    color: '#fafafa',
                }}>
                >
                    Icon {id}
                </div>
            )
        )
    )
}
}

```

Examples

Using external data

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



The screenshot shows a code editor with a tab bar labeled "app/products/[id]/opengraph.ts" and "TypeScript". The code is written in TypeScript and uses the `next/og` package to generate Open Graph images. It imports `ImageResponse` and `getCaptionForImage`, `getOGImages`. The `generateImageMetadata` function takes a `params` object with an `id` string. It then calls `getOGImages` with the `params.id` to get images.

```

import { ImageResponse } from 'next/og'
import { getCaptionForImage, getOGImages } from 'next/og'

export async function generateImageMetadata({
    params,
}: {
    params: { id: string }
}) {
    const images = await getOGImages(params.id)
}

```

```

        return images.map((image, idx) => ({
          id: idx,
          size: { width: 1200, height: 600 },
          alt: image.text,
          contentType: 'image/png',
        }))
      }

      export default async function Image({
        params,
        id,
      }: {
        params: { id: string }
        id: number
      }) {
        const productId = (await params).id
        const imageId = id
        const text = await getCaptionForImage(product
      }

      return new ImageResponse(
        (
          <div
            style={{
              ...
            }}
          >
            {text}
          </div>
        )
      )
    }
  
```

Version History

Version	Changes
v13.3.0	generateImageMetadata introduced.

Next Steps

View all the Metadata API options.

Metadata Fil...

API documentation
for the metadata
file conventions.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

generateMetadata

You can use the `metadata` object or the `generateMetadata` function to define metadata.

The `metadata` object

To define static metadata, export a [Metadata object](#) from a `layout.js` or `page.js` file.

```
TS layout.tsx | page.tsx TypeScript ▾ ⌂

import type { Metadata } from 'next'

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

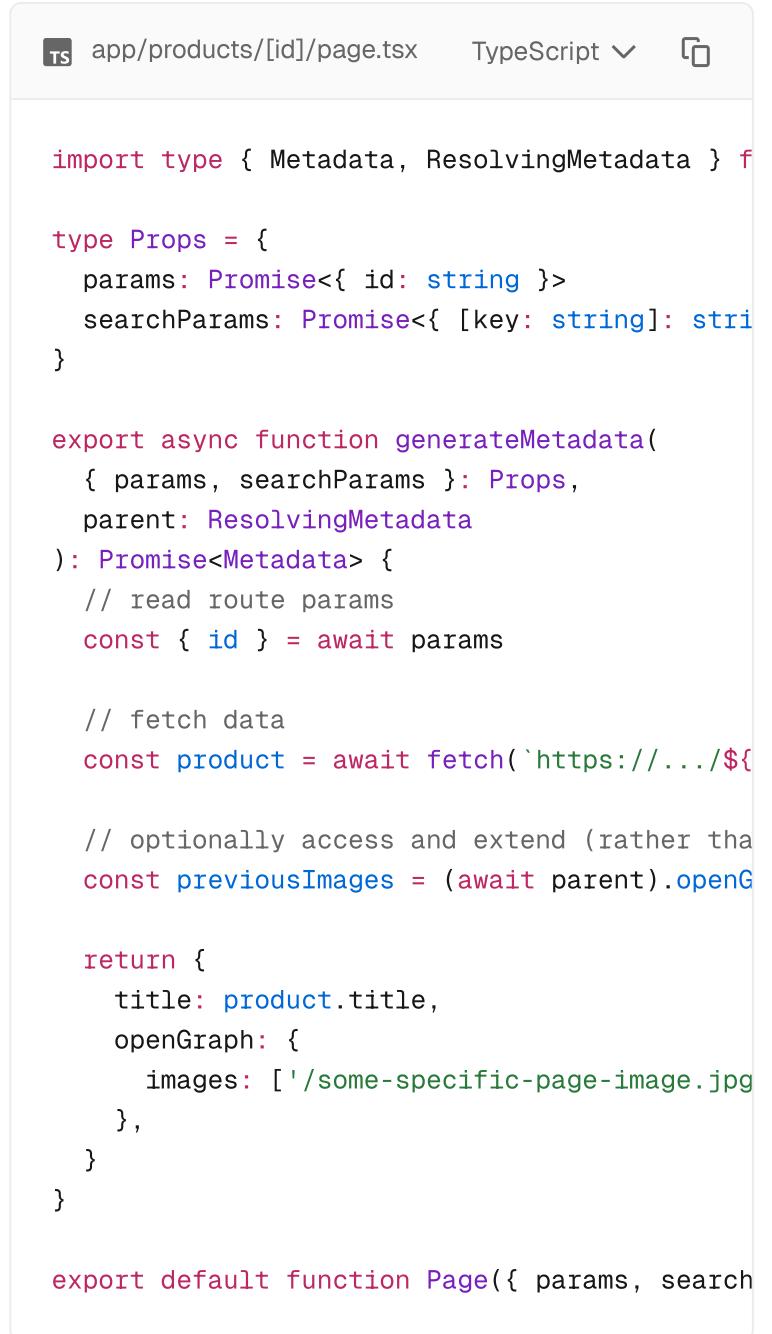
export default function Page() {}
```

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

generateMetadata function

Dynamic metadata depends on **dynamic information**, such as the current route parameters, external data, or `metadata` in parent segments,

can be set by exporting a `generateMetadata` function that returns a `Metadata` object.



```
TS app/products/[id]/page.tsx TypeScript ▾
```

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

type Props = {
  params: Promise<{ id: string }>
  searchParams: Promise<{ [key: string]: string }>
}

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

  // fetch data
  const product = await fetch(`https://.../${id}`)

  // optionally access and extend (rather than replace)
  const previousImages = (await parent).openGraph?.images

  return {
    title: product.title,
    openGraph: {
      images: ['/some-specific-page-image.jpg'],
    },
  }
}

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

For type completion of `params` and `searchParams`, you can type the first argument with `PageProps<'/route'>` or `LayoutProps<'/route'>` for pages and layouts respectively.

Good to know:

- Metadata can be added to `layout.js` and `page.js` files.
- Next.js will automatically resolve the metadata, and create the relevant `<head>` tags for the page.
- The `metadata` object and `generateMetadata` function exports are **only supported in Server**

Components.

- You cannot export both the `metadata` object and `generateMetadata` function from the same route segment.
- `fetch` requests inside `generateMetadata` are automatically **memoized** for the same data across `generateMetadata`, `generateStaticParams`, Layouts, Pages, and Server Components.
- Resolving `generateMetadata` is part of rendering the page. If the page can be pre-rendered and `generateMetadata` doesn't introduce dynamic behavior, its result is included in the page's initial HTML.
- React **cache** can be used if `fetch` is unavailable.
- **File-based metadata** has the higher priority and will override the `metadata` object and `generateMetadata` function.

Reference

Parameters

`generateMetadata` function accepts the following parameters:

- `props` - An object containing the parameters of the current route:
 - `params` - An object containing the **dynamic route parameters** object from the root segment down to the segment `generateMetadata` is called from.

Examples:

Route

URL

`app/shop/[slug]/page.js`

`/shop/1`

Route

URL

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

/shop/1/2

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

/shop/1/2

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

URL

searchParams

/shop?a=1

{ a: '1' }

/shop?a=1&b=2

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

/shop?a=1&a=2

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

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

Returns

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

Good to know:

- If metadata doesn't depend on runtime information, it should be defined using the static [metadata object](#) rather than `generateMetadata`.
- `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.
- `searchParams` are only available in `page.js` segments.

- The `redirect()` and `notFound()` Next.js methods can also be used inside `generateMetadata`.

Metadata Fields

The following fields are supported:

`title`

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

`String`

`JS` layout.js | page.js 

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

 <head> output

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

`default`

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

`TS` app/layout.tsx 

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

export const metadata: Metadata = {
  title: {
    default: 'Acme',
  },
}
```



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

export const metadata: Metadata = {}

// Output: <title>Acme</title>
```

template

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

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

export const metadata: Metadata = {
  title: {
    template: '%s | Acme',
    default: 'Acme', // a default is required
  },
}
```

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

export const metadata: Metadata = {
  title: 'About',
}
```

// Output: <title>About | Acme</title>

Good to know:

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

terminating segment (it doesn't have any children route segments).

- `title.template` has **no effect** if a route has not defined a `title` or `title.default`.

absolute

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

app/layout.tsx

TypeScript

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

export const metadata: Metadata = {
  title: {
    template: '%s | Acme',
  },
}
```

app/about/page.tsx

TypeScript

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

export const metadata: Metadata = {
  title: {
    absolute: 'About',
  },
}

// Output: <title>About</title>
```

Good to know:

- `layout.js`
 - `title` (string) and `title.default` define the default title for child segments (that do not define their own `title`). It will augment `title.template` from the closest parent segment if it exists.
 - `title.absolute` defines the default title for child segments. It ignores `title.template` from parent segments.

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

description

`JS` layout.js | page.js 

```
export const metadata = {
  description: 'The React Framework for the Web'
}
```

 <head> output 

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

Other fields

`JS` layout.js | page.js 

```
export const metadata = {
  generator: 'Next.js',
  applicationName: 'Next.js',
  referrer: 'origin-when-cross-origin',
  keywords: ['Next.js', 'React', 'JavaScript'],
  authors: [{ name: 'Seb' }, { name: 'Josh' }],
  creator: 'Jiachi Liu',
  publisher: 'Sebastian Markbåge',
  formatDetection: {
    email: false,
    address: false,
    telephone: false,
  },
}
```

```
}
```

📄 <head> output

```
<meta name="application-name" content="Next.js" />
<meta name="author" content="Seb" />
<link rel="author" href="https://nextjs.org" />
<meta name="author" content="Josh" />
<meta name="generator" content="Next.js" />
<meta name="keywords" content="Next.js,React,Node.js" />
<meta name="referrer" content="origin-when-cross-origin" />
<meta name="color-scheme" content="dark" />
<meta name="creator" content="Jiachi Liu" />
<meta name="publisher" content="Sebastian Marhan" />
<meta name="format-detection" content="telephone" />
```

metadataBase

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

- `metadataBase` allows URL-based `metadata` fields defined in the **current route segment and below** to use a **relative path** instead of an otherwise required absolute URL.
- The field's relative path will be composed with `metadataBase` to form a fully qualified URL.

JS layout.js | page.js

```
export const metadata = {
  metadataBase: new URL('https://acme.com'),
  alternates: {
    canonical: '/',
    languages: {
      'en-US': '/en-US',
      'de-DE': '/de-DE',
    },
  },
  openGraph: {
    images: '/og-image.png',
  },
}
```



```
<link rel="canonical" href="https://acme.com">
<link rel="alternate" hreflang="en-US" href=">
<link rel="alternate" hreflang="de-DE" href=">
<meta property="og:image" content="https://ac
```

Good to know:

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

URL Composition

URL composition favors developer intent over default directory traversal semantics.

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

For example, given the following `metadataBase`:



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

```
export const metadata: Metadata = {
  metadataBase: new URL('https://acme.com'),
}
```

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

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

openGraph

js layout.js | page.js



```
export const metadata = {
  openGraph: {
    title: 'Next.js',
    description: 'The React Framework for the',
    url: 'https://nextjs.org',
    siteName: 'Next.js',
    images: [
      {
        url: 'https://nextjs.org/og.png', //
        width: 800,
        height: 600,
      },
      {
        url: 'https://nextjs.org/og-alt.png',
        width: 1800,
        height: 1600,
        alt: 'My custom alt',
      }
    ]
  }
}
```

```
        },
      ],
      videos: [
        {
          url: 'https://nextjs.org/video.mp4',
          width: 800,
          height: 600,
        },
      ],
      audio: [
        {
          url: 'https://nextjs.org/audio.mp3',
        },
      ],
      locale: 'en_US',
      type: 'website',
    },
  }
}
```

📄 <head> output

```
<meta property="og:title" content="Next.js" />
<meta property="og:description" content="The
<meta property="og:url" content="https://next
<meta property="og:site_name" content="Next.j
<meta property="og:locale" content="en_US" />
<meta property="og:image" content="https://ne
<meta property="og:image:width" content="800"
<meta property="og:image:height" content="600"
<meta property="og:image" content="https://ne
<meta property="og:image:width" content="1800
<meta property="og:image:height" content="160
<meta property="og:image:alt" content="My cus
<meta property="og:video" content="https://ne
<meta property="og:video:width" content="800"
<meta property="og:video:height" content="600"
<meta property="og:audio" content="https://ne
<meta property="og:type" content="website" />
```

JS layout.js | page.js

```
export const metadata = {
  openGraph: {
    title: 'Next.js',
    description: 'The React Framework for the',
    type: 'article',
    publishedTime: '2023-01-01T00:00:00.000Z',
    authors: ['Seb', 'Josh'],
  },
}
```

└ <head> output

```
<meta property="og:title" content="Next.js" />
<meta property="og:description" content="The" />
<meta property="og:type" content="article" />
<meta property="article:published_time" content="2023-01-01T00:00:00.000Z" />
<meta property="article:author" content="Seb" />
<meta property="article:author" content="Josh" />
```

Good to know:

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

robots

layout.tsx | page.tsx

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

export const metadata: Metadata = {
  robots: {
    index: true,
    follow: true,
    nocache: false,
    googleBot: {
      index: true,
      follow: true,
      noimageindex: false,
      'max-video-preview': -1,
      'max-image-preview': 'large',
      'max-snippet': -1,
    },
  },
}
```

```
}
```

 <head> output



```
<meta name="robots" content="index, follow" />
<meta
  name="googlebot"
  content="index, follow, max-video-preview:-1" />
```

icons

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

 layout.js | page.js



```
export const metadata = {
  icons: {
    icon: '/icon.png',
    shortcut: '/shortcut-icon.png',
    apple: '/apple-icon.png',
    other: {
      rel: 'apple-touch-icon-precomposed',
      url: '/apple-touch-icon-precomposed.png'
    },
  },
}
```

 <head> output



```
<link rel="shortcut icon" href="/shortcut-icon.png" />
<link rel="icon" href="/icon.png" />
<link rel="apple-touch-icon" href="/apple-icon.png" />
<link
  rel="apple-touch-icon-precomposed"
  href="/apple-touch-icon-precomposed.png" />
```

 layout.js | page.js



```
export const metadata = {
  icons: {
    icon: [
      { url: '/icon.png' },
      new URL('/icon.png', 'https://example.c
      { url: '/icon-dark.png', media: '(prefe
    ],
    shortcut: ['/shortcut-icon.png'],
    apple: [
      { url: '/apple-icon.png' },
      { url: '/apple-icon-x3.png', sizes: '18
    ],
    other: [
      {
        rel: 'apple-touch-icon-precomposed',
        url: '/apple-touch-icon-precomposed.p
      },
    ],
    },
  }
}
```

<head> output

```
<link rel="shortcut icon" href="/shortcut-ico
<link rel="icon" href="/icon.png" />
<link rel="icon" href="https://example.com/ic
<link rel="icon" href="/icon-dark.png" media=
<link rel="apple-touch-icon" href="/apple-ico
<link
  rel="apple-touch-icon-precomposed"
  href="/apple-touch-icon-precomposed.png"
/>
<link
  rel="apple-touch-icon"
  href="/apple-icon-x3.png"
  sizes="180x180"
  type="image/png"
/>
```

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

themeColor

Deprecated: The `themeColor` option in `metadata` is deprecated as of Next.js 14. Please use the `viewport`

configuration instead.

colorScheme

Deprecated: The `colorScheme` option in `metadata` is deprecated as of Next.js 14. Please use the [viewport configuration](#) instead.

manifest

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

`JS` layout.js | page.js 

```
export const metadata = {
  manifest: 'https://nextjs.org/manifest.json'
}
```

 <head> output 

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

twitter

The Twitter specification is (surprisingly) used for more than just X (formerly known as Twitter).

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

`JS` layout.js | page.js 

```
export const metadata = {
  twitter: {
    card: 'summary_large_image',
    title: 'Next.js',
    description: 'The React Framework for the',
    siteId: '1467726470533754880',
    creator: '@nextjs',
    creatorId: '1467726470533754880',
    images: ['https://nextjs.org/og.png'], //
```

```
 },  
 }
```

📄 <head> output

```
<meta name="twitter:card" content="summary_lan  
<meta name="twitter:site:id" content="14677264726470533754880"  
<meta name="twitter:creator" content="@nextjs"  
<meta name="twitter:creator:id" content="1467726470533754880"  
<meta name="twitter:title" content="Next.js"  
<meta name="twitter:description" content="The React Framework for the Web"  
<meta name="twitter:image" content="https://nextjs.org/og.png"
```

JSImport | page.js

```
export const metadata = {  
  twitter: {  
    card: 'app',  
    title: 'Next.js',  
    description: 'The React Framework for the Web',  
    siteId: '1467726470533754880',  
    creator: '@nextjs',  
    creatorId: '1467726470533754880',  
    images: {  
      url: 'https://nextjs.org/og.png',  
      alt: 'Next.js Logo',  
    },  
    app: {  
      name: 'twitter_app',  
      id: {  
        iphone: 'twitter_app://iphone',  
        ipad: 'twitter_app://ipad',  
        googleplay: 'twitter_app://googleplay',  
      },  
      url: {  
        iphone: 'https://iphone_url',  
        ipad: 'https://ipad_url',  
      },  
    },  
  },  
}
```

📄 <head> output

```
<meta name="twitter:site:id" content="14677264726470533754880"  
<meta name="twitter:creator" content="@nextjs"  
<meta name="twitter:creator:id" content="1467726470533754880"  
<meta name="twitter:title" content="Next.js"
```

```
<meta name="twitter:description" content="The  
<meta name="twitter:card" content="app" />  
<meta name="twitter:image" content="https://n  
<meta name="twitter:image:alt" content="Next.  
<meta name="twitter:app:name:iphone" content=  
<meta name="twitter:app:id:iphone" content="t  
<meta name="twitter:app:id:ipad" content="twi  
<meta name="twitter:app:id:googleplay" conten  
<meta name="twitter:app:url:iphone" content="  
<meta name="twitter:app:url:ipad" content="ht  
<meta name="twitter:app:name:ipad" content="t  
<meta name="twitter:app:name:googleplay" cont
```

viewport

Deprecated: The `viewport` option in `metadata` is deprecated as of Next.js 14. Please use the [viewport configuration](#) instead.

verification

`js` layout.js | page.js

```
export const metadata = {  
  verification: {  
    google: 'google',  
    yandex: 'yandex',  
    yahoo: 'yahoo',  
    other: {  
      me: ['my-email', 'my-link'],  
    },  
  },  
}
```

`js` <head> output

```
<meta name="google-site-verification" content  
<meta name="y_key" content="yahoo" />  
<meta name="yandex-verification" content="yan  
<meta name="me" content="my-email" />  
<meta name="me" content="my-link" />
```

appleWebApp



```
export const metadata = {
  itunes: {
    appId: 'myAppStoreID',
    appArgument: 'myAppArgument',
  },
  appleWebApp: {
    title: 'Apple Web App',
    statusBarStyle: 'black-translucent',
    startupImage: [
      '/assets/startup/apple-touch-startup-image.png',
      {
        url: '/assets/startup/apple-touch-startup-image.png',
        media: '(device-width: 768px) and (device-height: 1024px) and (orientation: portrait)'
      }
    ],
  },
}
```



```
<meta
  name="apple-itunes-app"
  content="app-id=myAppStoreID, app-argument=ios"/>
<meta name="mobile-web-app-capable" content="yes"/>
<meta name="apple-mobile-web-app-title" content="Next.js"/>
<link
  href="/assets/startup/apple-touch-startup-image.png"
  rel="apple-touch-startup-image"/>
<link
  href="/assets/startup/apple-touch-startup-image.png"
  media="(device-width: 768px) and (device-height: 1024px) and (orientation: portrait)"
  rel="apple-touch-startup-image"/>
<meta
  name="apple-mobile-web-app-status-bar-style"
  content="black-translucent"/>

```

alternates



```
export const metadata = {
  alternates: {
    canonical: 'https://nextjs.org',
  }
}
```

```
languages: {
  'en-US': 'https://nextjs.org/en-US',
  'de-DE': 'https://nextjs.org/de-DE',
},
media: {
  'only screen and (max-width: 600px)': '
},
types: {
  'application/rss+xml': 'https://nextjs.
},
},
}
```

🔗 <head> output

```
<link rel="canonical" href="https://nextjs.or
<link rel="alternate" hreflang="en-US" href="
<link rel="alternate" hreflang="de-DE" href="
<link
  rel="alternate"
  media="only screen and (max-width: 600px)"
  href="https://nextjs.org/mobile"
/>
<link
  rel="alternate"
  type="application/rss+xml"
  href="https://nextjs.org/rss"
/>
```

appLinks

JS layout.js | page.js

```
export const metadata = {
  appLinks: {
    ios: {
      url: 'https://nextjs.org/ios',
      app_store_id: 'app_store_id',
    },
    android: {
      package: 'com.example.android/package',
      app_name: 'app_name_android',
    },
    web: {
      url: 'https://nextjs.org/web',
      should_fallback: true,
    },
  },
}
```

📄 <head> output



```
<meta property="al:ios:url" content="https://
<meta property="al:ios:app_store_id" content=
<meta property="al:android:package" content=
<meta property="al:android:app_name" content=
<meta property="al:web:url" content="https://
<meta property="al:web:should_fallback" conte
```

archives

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

JS layout.js | page.js



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

📄 <head> output



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

assets



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



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

bookmarks



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



```
<link rel="bookmarks" href="https://nextjs.or
```

category



```
export const metadata = {
  category: 'technology',
}
```



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

facebook

You can connect a Facebook app or Facebook account to your webpage for certain Facebook Social Plugins [Facebook Documentation](#)

Good to know: You can specify either appId or admins, but not both.

 layout.js | page.js 

```
export const metadata = {
  facebook: {
    appId: '12345678',
  },
}
```

 <head> output 

```
<meta property="fb:app_id" content="12345678"
```

 layout.js | page.js 

```
export const metadata = {
  facebook: {
    admins: '12345678',
  },
}
```

 <head> output 

```
<meta property="fb:admins" content="12345678"
```

If you want to generate multiple fb:admins meta tags you can use array value.

 layout.js | page.js 

```
export const metadata = {
  facebook: {
    admins: ['12345678', '87654321'],
  },
}
```

```
}
```

 <head> output



```
<meta property="fb:admins" content="12345678"  
<meta property="fb:admins" content="87654321"
```

pinterest

You can enable or disable [Pinterest Rich Pins ↗](#) on your webpage.

 layout.js | page.js



```
export const metadata = {  
  pinterest: {  
    richPin: true,  
  },  
}
```

 <head> output



```
<meta name="pinterest-rich-pin" content="true"
```

other

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

 layout.js | page.js



```
export const metadata = {  
  other: {  
    custom: 'meta',  
  },  
}
```

```
<head> output
```



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

If you want to generate multiple same key meta tags you can use array value.

```
js layout.js | page.js
```



```
export const metadata = {
  other: {
    custom: ['meta1', 'meta2'],
  },
}
```

```
<head> output
```



```
<meta name="custom" content="meta1" /> <meta
```

Types

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

metadata object

```
ts layout.tsx | page.tsx
```



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

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

generateMetadata function

Regular function



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

export function generateMetadata(): Metadata
  return {
    title: 'Next.js',
  }
}
```

Async function



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

export async function generateMetadata(): Promise<Metadata>
  return {
    title: 'Next.js',
  }
}
```

With segment props



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

type Props = {
  params: Promise<{ id: string }>
  searchParams: Promise<{ [key: string]: string }>
}

export function generateMetadata({ params, searchParams }: Props): Metadata
  return {
    title: 'Next.js',
  }
}

export default function Page({ params, searchParams }) {
  return (
    <div>
      <h1>Hello, <span>{params.id}</span>!</h1>
      <p>This page was generated by the <code>generateMetadata</code> function</p>
    </div>
  )
}
```

With parent metadata



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

```
export async function generateMetadata(  
  { params, searchParams }: Props,  
  parent: ResolvingMetadata  
): Promise<Metadata> {  
  return {  
    title: 'Next.js',  
  }  
}
```

JavaScript Projects

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



```
JS layout.js | page.js ⌂  
  
/** @type {import("next").Metadata} */  
export const metadata = {  
  title: 'Next.js',  
}
```

Unsupported Metadata

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

Metadata	Recommendation
<meta http-equiv=" ... ">	Use appropriate HTTP Headers via redirect() , Middleware , Security Headers
<base>	Render the tag in the layout or page itself.
<noscript>	Render the tag in the layout or page itself.
<style>	Learn more about styling in Next.js .
<script>	Learn more about using scripts .

Metadata

Recommendation

```
<link  
rel="stylesheet"  
/>
```

import stylesheets directly in the layout or page itself.

```
<link rel="preload"  
/>
```

Use [ReactDOM preload method](#)

```
<link  
rel="preconnect"  
/>
```

Use [ReactDOM preconnect method](#)

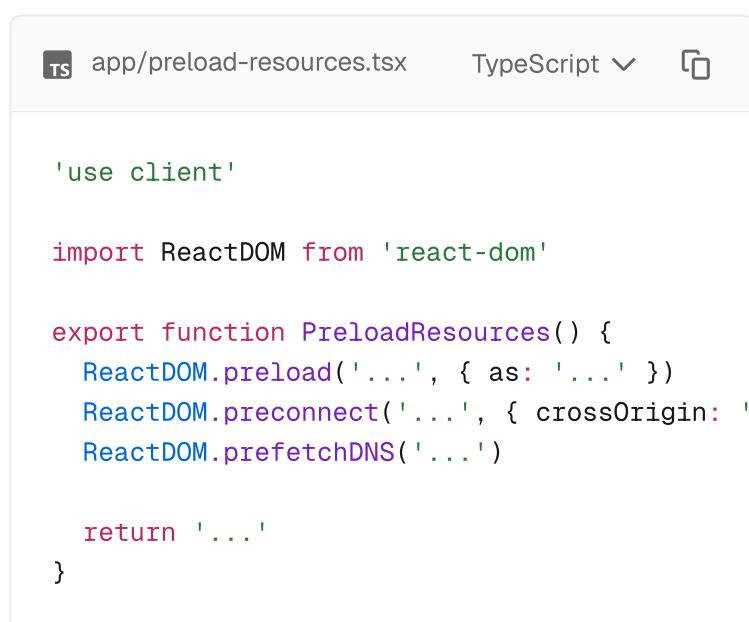
```
<link rel="dns-  
prefetch" />
```

Use [ReactDOM prefetchDNS method](#)

Resource hints

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

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



```
TS app/preload-resources.tsx TypeScript ▾ ⌂

'use client'

import ReactDOM from 'react-dom'

export function PreloadResources() {
  ReactDOM.preload('...', { as: '...' })
  ReactDOM.preconnect('...', { crossOrigin: '...' })
  ReactDOM.prefetchDNS('...')

  return '...'
}
```

```
<link rel="preload">
```

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

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

 <head> output 

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

```
<link rel="preconnect">
```

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

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

 <head> output 

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

```
<link rel="dns-prefetch">
```

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

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

 <head> output 

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

Good to know:

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

- Next.js in-built features such as `next/font`, `next/image` and `next/script` automatically handle relevant resource hints.

Behavior

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-w
```

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

Streaming metadata

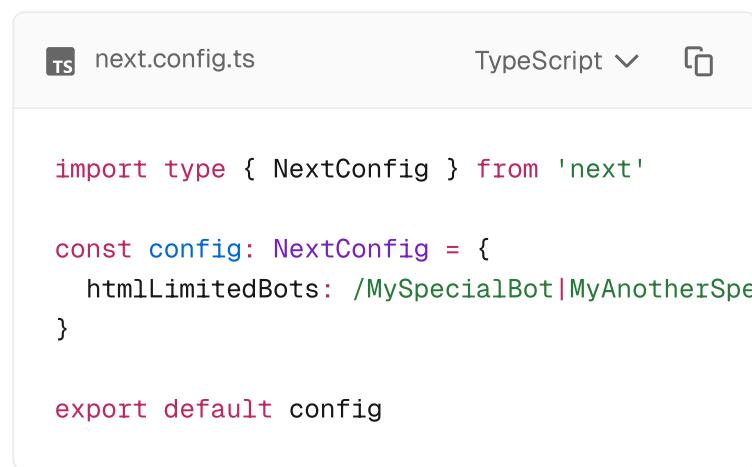
Metadata returned by `generateMetadata` is streamed to the client. This allows Next.js to inject metadata into the HTML as soon as it's resolved.

Streamed metadata is appended to the `<body>` tag. Since metadata mainly targets bots and crawlers, Next.js streams metadata for bots that can execute JavaScript and inspect the full DOM (e.g. `Googlebot`). We have verified that these bots interpret the metadata correctly.

For **HTML-limited** bots that can't run JavaScript (e.g. `Twitterbot`), metadata continues to block page rendering and is placed in the `<head>` tag.

Next.js automatically detects the user agent of incoming requests to determine whether to serve streaming metadata or fallback to blocking metadata.

If you need to customize this list, you can define them manually using the `htmlLimitedBots` option in `next.config.js`. Next.js will ensure user agents matching this regex receive blocking metadata when requesting your web page.



The screenshot shows a code editor window with a TypeScript file named `next.config.ts`. The file contains the following code:

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

const config: NextConfig = {
  htmlLimitedBots: /MySpecialBot|MyAnotherSpe
}

export default config
```

Specifying a `htmlLimitedBots` config will override the Next.js' default list, allowing you full control over what user agents should opt into this behavior.

Overriding `htmlLimitedBots` could lead to longer response times. Streaming metadata is an advanced feature, and the default should be sufficient for most cases.

Ordering

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

1. `app/layout.tsx` (Root Layout)

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

Merging

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

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

Overwriting fields

`JS app/layout.js`

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

`JS app/blog/page.js`

```
export const metadata = {
  title: 'Blog',
  openGraph: {
    title: 'Blog',
  },
}

// Output:
// <title>Blog</title>
// <meta property="og:title" content="Blog" /
```

In the example above:

- title from app/layout.js is replaced by title in app/blog/page.js.
- All openGraph fields from app/layout.js are replaced in app/blog/page.js because app/blog/page.js sets openGraph metadata. Note the absence of openGraph.description.

If you'd like to share some nested fields between segments while overwriting others, you can pull them out into a separate variable:

`JS` app/shared-metadata.js

```
export const openGraphImage = { images: [ 'htt
```

`JS` app/page.js

```
import { openGraphImage } from './shared-meta

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

`JS` app/about/page.js

```
import { openGraphImage } from '../shared-met

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="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.

Version History

Version	Changes
v15.2.0	Introduced streaming support to generateMetadata.
v13.2.0	viewport, themeColor, and colorScheme deprecated in favor of the viewport configuration.

v15.2.0 Introduced streaming support to generateMetadata.

v13.2.0 viewport, themeColor, and colorScheme deprecated in favor of the viewport configuration.

v13.2.0

metadata and generateMetadata

introduced.

Next Steps

[View all the Metadata API options.](#)

Metadata Fil...

API documentation
for the metadata
file conventions.

generateVie...

API Reference for
the
generateViewport...

Was this helpful?





Using App Router
Features available in /app



generateSitemaps



Latest Version
15.5.4



You can use the `generateSitemaps` function to generate multiple sitemaps for your application.

Returns

The `generateSitemaps` returns an array of objects with an `id` property.

URLs

Your generated sitemaps will be available at

`/ ... /sitemap/[id].xml`. For example,
`/product/sitemap/1.xml`.

Example

For example, to split a sitemap using `generateSitemaps`, return an array of objects with the sitemap `id`. Then, use the `id` to generate the unique sitemaps.

app/product/sitemap.ts TypeScript ▾

```
import { BASE_URL } from '@/app/lib/constants

export async function generateSitemaps() {
```

```
// Fetch the total number of products and c
return [{ id: 0 }, { id: 1 }, { id: 2 }, {
}

export default async function sitemap({
  id,
}: {
  id: number
}): Promise<MetadataRoute.Sitemap> {
  // Google's limit is 50,000 URLs per sitema
  const start = id * 50000
  const end = start + 50000
  const products = await getProducts(
    `SELECT id, date FROM products WHERE id B
  )
  return products.map((product) => ({
    url: `${BASE_URL}/product/${product.id}`,
    lastModified: product.date,
  }))
}
```

Version History

Version	Changes
v15.0.0	<code>generateSitemaps</code> now generates consistent URLs between development and production
v13.3.2	<code>generateSitemaps</code> introduced. In development, you can view the generated sitemap on <code>/.../sitemap.xml/[id]</code> . For example, <code>/product/sitemap.xml/1</code> .

Next Steps

Learn how to create sitemaps for your Next.js application.

sitemap.xml

API Reference for
the sitemap.xml
file.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

generateStaticParams

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

```
TS app/blog/[slug]/page.tsx TypeScript ▾   
  
// Return a list of `params` to populate the  
export async function generateStaticParams()  
  const posts = await fetch('https://.../post  
  
  return posts.map((post) => ({  
    slug: post.slug,  
  }))  
}  
  
// Multiple versions of this page will be sta  
// using the `params` returned by `generateSt  
export default async function Page({  
  params,  
}: {  
  params: Promise<{ slug: string }>  
}) {  
  const { slug } = await params  
  // ...  
}
```

Good to know:

- You can use the `dynamicParams` segment config option to control what happens when a dynamic segment is visited that was not generated with `generateStaticParams`.
- You must return an empty array from `generateStaticParams` or utilize `export const dynamic = 'force-static'` in order to revalidate (ISR) paths at runtime.

- During `next dev`, `generateStaticParams` will be called when you navigate to a route.
- During `next build`, `generateStaticParams` runs before the corresponding Layouts or Pages are generated.
- During revalidation (ISR), `generateStaticParams` will not be called again.
- `generateStaticParams` replaces the `getStaticPaths` function in the Pages Router.

Parameters

`options.params` (optional)

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

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

Returns

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

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

Example Route	generateStaticParams()	Return Type
/product/[id]		{ id: string }[]
/products/[category]/[product]		{ category: string; product: string }
/products/[...slug]		{ slug: string }[]

Single Dynamic Segment

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

```
export function generateStaticParams() {
  return [{ id: '1' }, { id: '2' }, { id: '3' }]
}

// Three versions of this page will be static
// using the `params` returned by `generateStaticParams`
// - /product/1
// - /product/2
// - /product/3
export default async function Page({
  params,
}: {
  params: Promise<{ id: string }>
}) {
  const { id } = await params
  // ...
}
```

Multiple Dynamic Segments

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

```
export function generateStaticParams() {
  return [
    { category: 'a', product: '1' },
    { category: 'a', product: '2' },
    { category: 'b', product: '1' },
    { category: 'b', product: '2' },
    { category: 'b', product: '3' },
    { category: 'c', product: '1' },
    { category: 'c', product: '2' },
  ]
}
```

```
        { category: 'b', product: '2' },
        { category: 'c', product: '3' },
    ]
}

// Three versions of this page will be static
// using the `params` returned by `generateSt
// - /products/a/1
// - /products/b/2
// - /products/c/3
export default async function Page({
    params,
}: {
    params: Promise<{ category: string; product: string }>
}) {
    const { category, product } = await params
    // ...
}
```

Catch-all Dynamic Segment

```
ts app/product/[..slug]/page.tsx TypeScript ▾ ⌂

export function generateStaticParams() {
    return [{ slug: ['a', '1'] }, { slug: ['b', '1'] }]
}

// Three versions of this page will be static
// using the `params` returned by `generateSt
// - /product/a/1
// - /product/b/2
// - /product/c/3
export default async function Page({
    params,
}: {
    params: Promise<{ slug: string[] }>
}) {
    const { slug } = await params
    // ...
}
```

Examples

Static Rendering

All paths at build time

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

```
TS app/blog/[slug]/page.tsx TypeScript ▾ ⌂  
  
export async function generateStaticParams()  
  const posts = await fetch('https://.../post')  
  
  return posts.map((post) => ({  
    slug: post.slug,  
  }))  
}
```

Subset of paths at build time

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:

```
TS app/blog/[slug]/page.tsx TypeScript ▾ ⌂  
  
export async function generateStaticParams()  
  const posts = await fetch('https://.../post')  
  
  // Render the first 10 posts at build time  
  return posts.slice(0, 10).map((post) => ({  
    slug: post.slug,  
  }))  
}
```

Then, by using the `dynamicParams` segment config option, you can control what happens when a dynamic segment is visited that was not generated with `generateStaticParams`.

```
TS app/blog/[slug]/page.tsx TypeScript ▾ ⌂  
  
// All posts besides the top 10 will be a 404  
export const dynamicParams = false  
  
export async function generateStaticParams()
```

```
const posts = await fetch('https://.../post')
const topPosts = posts.slice(0, 10)

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

All paths at runtime

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' :
```

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

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

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

`JS app/changelog/[slug]/page.js`

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

Disable rendering for unspecified paths

To prevent unspecified paths from being statically rendered at runtime, 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 unspecified routes will 404 or match (in the case of [catch-all routes](#)).

Multiple Dynamic Segments in a Route

You can generate params for dynamic segments above the current layout or page, but **not below**.

For example, given the

app/products/[category]/[product] route:

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

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

Generate params from the bottom up

Generate multiple dynamic segments from the child route segment.



The screenshot shows a code editor window with a TypeScript file open. The file contains code for generating static parameters from the bottom up. The code includes an export function named generateStaticParams() which fetches products from a URL and maps them to an array of objects containing category and product IDs. It also includes a default export for a Page component that takes params and returns a promise with category and product information.

```
TS app/products/[category]/[pr... TypeScript ▾ ⌂
// Generate segments for both [category] and
export async function generateStaticParams()
  const products = await fetch('https://.../p

    return products.map((product) => ({
      category: product.category.slug,
      product: product.id,
    }))
}

export default function Page({
  params,
}: {
  params: Promise<{ category: string; product
}) {
  // ...
}
```

Generate params from the top down

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

```
TS app/products/[category]/lay... TypeScript ▾
```

```
// Generate segments for [category]
export async function generateStaticParams()
  const products = await fetch('https://.../p

  return products.map((product) => ({
    category: product.category.slug,
  }))
}

export default function Layout({
  params,
}: {
  params: Promise<{ category: string }>
}) {
  // ...
}
```

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

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

```
TS app/products/[category]/[pr... TypeScript ▾
```

```
// Generate segments for [product] using the
// the parent segment's `generateStaticParams
export async function generateStaticParams({
  params: { category },
}: {
  params: { category: string }
}) {
  const products = await fetch(
    `https://.../products?category=${category}
  ).then((res) => res.json())

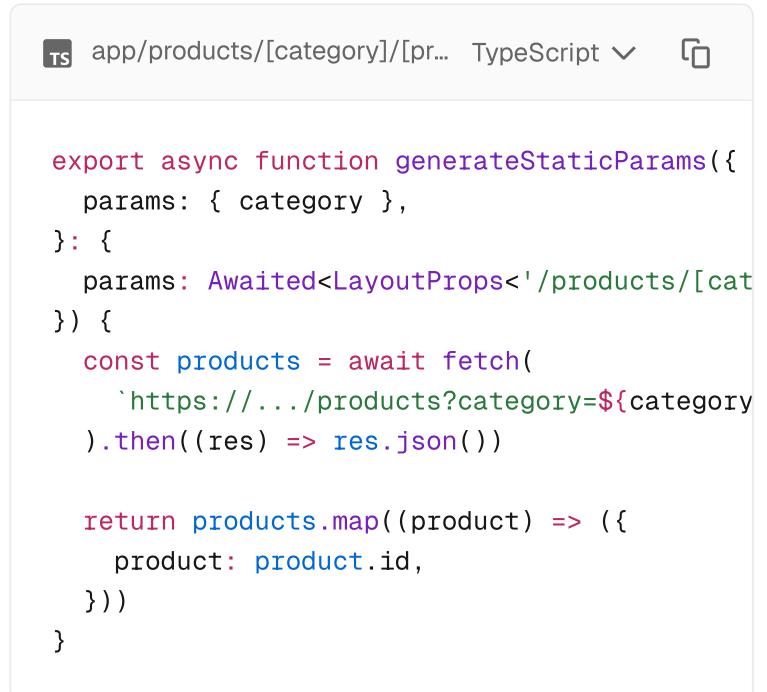
  return products.map((product) => ({
    product: product.id,
  }))
}
```

```
        }

        export default function Page({
          params,
        }: {
          params: Promise<{ category: string; product
        }) {
          // ...
        }
      }
```

Notice that the params argument can be accessed synchronously and includes only parent segment params.

For type completion, you can make use of the TypeScript `Awaited` helper in combination with either `Page Props helper` or `Layout Props helper`:



The screenshot shows a code editor window with a TypeScript file open. The file contains a function named `generateStaticParams` that takes a parameter `category`. Inside the function, it uses `Awaited<LayoutProps<'/products/[cat...]` to get products from a fetch request. It then maps over the products to return an array of objects containing `product.id`.

```
TS app/products/[category]/[pr... TypeScript ▾
```

```
export async function generateStaticParams({
  params: { category },
}: {
  params: Awaited<LayoutProps<'/products/[cat...
}) {
  const products = await fetch(
    `https://.../products?category=${category}
  ).then((res) => res.json())

  return products.map((product) => ({
    product: product.id,
  }))
}
```

Good to know: `fetch` requests are automatically memoized for the same data across all `generate-` prefixed functions, Layouts, Pages, and Server Components. React `cache` can be used if `fetch` is unavailable.

Version History

Version	Changes
---------	---------

v13.0.0	generateStaticParams introduced.
---------	----------------------------------

Was this helpful?    



Using App Router
Features available in /app



generateViewport



Latest Version
15.5.4



You can customize the initial viewport of the page with the static `viewport` object or the dynamic `generateViewport` function.

Good to know:

- The `viewport` object and `generateViewport` function exports are **only supported in Server Components**.
- You cannot export both the `viewport` object and `generateViewport` function from the same route segment.
- If you're coming from migrating `metadata` exports, you can use [metadata-to-viewport-export codemod](#) to update your changes.

The `viewport` object

To define the viewport options, export a `viewport` object from a `layout.jsx` or `page.jsx` file.

TS layout.tsx | page.tsx TypeScript

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

export const viewport: Viewport = {
  themeColor: 'black',
}

export default function Page() {}
```

generateViewport function

`generateViewport` should return a `Viewport object` containing one or more viewport fields.

layout.tsx | page.tsx

TypeScript

```
export function generateViewport({ params })  
  return {  
    themeColor: '...',  
  }  
}
```

In TypeScript, the `params` argument can be typed via `PageProps<'/route'>` or `LayoutProps<'/route'>` depending on where `generateViewport` is defined.

Good to know:

- If the viewport doesn't depend on runtime information, it should be defined using the static `viewport object` rather than `generateViewport`.

Viewport Fields

themeColor

Learn more about [theme-color ↗](#).

Simple theme color

layout.tsx | page.tsx

TypeScript

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

export const viewport: Viewport = {
  themeColor: 'black',
}
```

 <head> output 

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

With media attribute

 layout.tsx | page.tsx  

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

export const viewport: Viewport = {
  themeColor: [
    { media: '(prefers-color-scheme: light)', value: 'white' },
    { media: '(prefers-color-scheme: dark)', value: 'black' },
  ],
}
```

 <head> output 

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

width, **initialScale**, **maximumScale**
and userScalable

Good to know: The `viewport` meta tag is automatically set, and manual configuration is usually unnecessary as the default is sufficient. However, the information is provided for completeness.

 layout.tsx | page.tsx  

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

export const viewport: Viewport = {
```

```
width: 'device-width',
initialScale: 1,
maximumScale: 1,
userScalable: false,
// Also supported but less commonly used
// interactiveWidget: 'resizes-visual',
}
```

□ <head> output □

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

colorScheme

Learn more about [color-scheme ↗](#).

TS layout.tsx | page.tsx TypeScript ▾ □

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

export const viewport: Viewport = {
  colorScheme: 'dark',
}
```

□ <head> output □

```
<meta name="color-scheme" content="dark" />
```

Types

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

viewport object

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

export const viewport: Viewport = {
  themeColor: 'black',
}
```

generateViewport function

Regular function

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

export function generateViewport(): Viewport
  return {
    themeColor: 'black',
  }
}
```

With segment props

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

type Props = {
  params: Promise<{ id: string }>
  searchParams: Promise<{ [key: string]: string }>
}

export function generateViewport({ params, searchParams }: Props): Viewport
  return {
    themeColor: 'black',
  }
}

export default function Page({ params, searchParams }) {
  const viewport = generateViewport({ params, searchParams })
  return (
    <div>
      <h1>Hello, {params.id}</h1>
      <p>This page was generated by {viewport.themeColor}</p>
    </div>
  )
}
```

JavaScript Projects

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

```
/** @type {import("next").Viewport} */
export const viewport = {
  themeColor: 'black',
}
```

}

Version History

Version	Changes
---------	---------

v14.0.0	viewport and generateViewport introduced.
---------	---

Next Steps

[View all the Metadata API options.](#)

[Metadata Fil...](#)

API documentation
for the metadata
file conventions.

Was this helpful?





Using App Router
Features available in /app



Latest Version
15.5.4

headers



`headers` is an **async** function that allows you to **read** the HTTP incoming request headers from a **Server Component**.

TS app/page.tsx

TypeScript



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

export default async function Page() {
  const headersList = await headers()
  const userAgent = headersList.get('user-agent')
}
```

Reference

Parameters

`headers` does not take any parameters.

Returns

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

- `Headers.entries()` ↗: Returns an [iterator](#) ↗ allowing to go through all key/value pairs contained in this object.
- `Headers.forEach()` ↗: Executes a provided function once for each key/value pair in this `Headers` object.

- `Headers.get()` ↗ : Returns a `String` ↗ sequence of all the values of a header within a `Headers` object with a given name.
 - `Headers.has()` ↗ : Returns a boolean stating whether a `Headers` object contains a certain header.
 - `Headers.keys()` ↗ : Returns an `iterator` ↗ allowing you to go through all keys of the key/value pairs contained in this object.
 - `Headers.values()` ↗ : Returns an `iterator` ↗ allowing you to go through all values of the key/value pairs contained in this object.
-

Good to know

- `headers` is an **asynchronous** function that returns a promise. You must use `async/await` or React's `use` ↗ function.
 - In version 14 and earlier, `headers` was a synchronous function. To help with backwards compatibility, you can still access it synchronously in Next.js 15, but this behavior will be deprecated in the future.
 - Since `headers` is read-only, you cannot `set` or `delete` the outgoing request headers.
 - `headers` is a **Dynamic API** whose returned values cannot be known ahead of time. Using it in will opt a route into **dynamic rendering**.
-

Examples

Using the Authorization header

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

export default async function Page() {
  const authorization = (await headers()).get('Authorization')
  const res = await fetch('...', { headers: { authorization } })
  const user = await res.json()

  return <h1>{user.name}</h1>
}
```

Version History

Version	Changes
v15.0.0-RC	<code>headers</code> is now an async function. A codemod is available.
v13.0.0	<code>headers</code> introduced.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

ImageResponse

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

Reference

Parameters

The following parameters are available for `ImageResponse`:

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

new ImageResponse(
  element: ReactElement,
  options: {
    width?: number = 1200
    height?: number = 630
    emoji?: 'twemoji' | 'blobemoji' | 'noto' |
    fonts?: {
      name: string,
      data: ArrayBuffer,
      weight: number,
      style: 'normal' | 'italic'
    }[]
    debug?: boolean = false

    // Options that will be passed to the HTT
    status?: number = 200
    statusText?: string
    headers?: Record<string, string>
  },
)
```

Supported HTML and CSS features

`ImageResponse` supports common CSS properties including flexbox and absolute positioning, custom fonts, text wrapping, centering, and nested images.

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

Behavior

- `ImageResponse` uses [@vercel/og](#) ↗, [Satori](#) ↗, and Resvg to convert HTML and CSS into PNG.
 - 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`.
-

Examples

Route Handlers

`ImageResponse` can be used in Route Handlers to generate images dynamically at request time.

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

export async function GET() {
  try {
    return new ImageResponse(
      (
        <div
          style={{
            height: '100%',
            width: '100%',
            display: 'flex',
            flexDirection: 'column',
            alignItems: 'center',
            justifyContent: 'center',
            backgroundColor: 'white',
            padding: '40px',
          }}>
        >
        <div
          style={{
            fontSize: 60,
            fontWeight: 'bold',
            color: 'black',
            textAlign: 'center',
          }}>
        >
        Welcome to My Site
      </div>
      <div
        style={{
          fontSize: 30,
          color: '#666',
          marginTop: '20px',
        }}>
      >
      Generated with Next.js ImageRespo
    </div>
  </div>
),
{
  width: 1200,
  height: 630,
}
)
} catch (e) {
  console.log(`[${e.message}]`)
  return new Response(`Failed to generate t
    status: 500,
  ))
}
}
}
```

File-based Metadata

You can use `ImageResponse` in a `opengraph-image.tsx` file to generate Open Graph images at build time or dynamically at request time.

```
TS app/opengraph-image.tsx 
```

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

// Image metadata
export const alt = 'My site'
export const size = {
  width: 1200,
  height: 630,
}

export const contentType = 'image/png'

// Image generation
export default async function Image() {
  return new ImageResponse(
    (
      // ImageResponse JSX element
      <div
        style={{
          fontSize: 128,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        My site
      </div>
    ),
    // ImageResponse options
    {
      // For convenience, we can re-use the e
      // size config to also set the ImageRes
      ...size,
    }
  )
}
```

Custom fonts

You can use custom fonts in your `ImageResponse` by providing a `fonts` array in the options.

app/opengraph-image.tsx



```
import { ImageResponse } from 'next/og'
import { readFile } from 'node:fs/promises'
import { join } from 'node:path'

// Image metadata
export const alt = 'My site'
export const size = {
  width: 1200,
  height: 630,
}

export const contentType = 'image/png'

// Image generation
export default async function Image() {
  // Font loading, process.cwd() is Next.js p
  const interSemiBold = await readFile(
    join(process.cwd(), 'assets/Inter-SemiBol
  )

  return new ImageResponse(
    (
      // ...
    ),
    // ImageResponse options
    {
      // For convenience, we can re-use the e
      // size config to also set the ImageRes
      ...size,
      fonts: [
        {
          name: 'Inter',
          data: interSemiBold,
          style: 'normal',
          weight: 400,
        },
      ],
    }
  )
}
```

Version History

Version Changes

v14.0.0 `ImageResponse` moved from `next/server` to `next/og`

v13.3.0 `ImageResponse` can be imported from `next/server`.

v13.0.0 `ImageResponse` introduced via `@vercel/og` package.

Was this helpful?    



Using App Router
Features available in /app



NextRequest



Latest Version
15.5.4



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

cookies

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

set(name, value)

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

```
// Given incoming request /home
// Set a cookie to hide the banner
// request will have a `Set-Cookie:show-banner
request.cookies.set('show-banner', 'false')
```

get(name)

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

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

getAll()

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

```
// Given incoming request /home
// [
//   { name: 'experiments', value: 'new-prici
//   { name: 'experiments', value: 'winter-la
// ]
request.cookies.getAll('experiments')
// Alternatively, get all cookies for the req
request.cookies.getAll()
```

delete(name)

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

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

has(name)

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

```
// Returns true if cookie exists, false if it
request.cookies.has('experiments')
```

clear()

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

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

nextUrl

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

```
// Given a request to /home, pathname is /home
request.nextUrl.pathname
// Given a request to /home?name=lee, searchParams
request.nextUrl.searchParams
```

The following options are available:

Property	Type	Description
basePath	string	The base path of the URL.
buildId	string undefined	The build identifier of the Next.js application. Can be customized .
pathname	string	The pathname of the URL.
searchParams	Object	The search parameters of the URL.

Note: The internationalization properties from the Pages Router are not available for usage in the App Router. Learn more about [internationalization with the App Router](#).

Version History

Version**Changes**

v15.0.0

ip and geo removed.

Was this helpful?    



Using App Router
Features available in /app



NextResponse



Latest Version
15.5.4



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

cookies

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

set(name, value)

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

```
// Given incoming request /home
let response = NextResponse.next()
// Set a cookie to hide the banner
response.cookies.set('show-banner', 'false')
// Response will have a `Set-Cookie:show-bann
return response
```

get(name)

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

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

getAll()

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

```
// Given incoming request /home
let response = NextResponse.next()
// [
//   { name: 'experiments', value: 'new-pricing' },
//   { name: 'experiments', value: 'winter-lights' }
// ]
response.cookies.getAll('experiments')
// Alternatively, get all cookies for the response
response.cookies.getAll()
```

delete(name)

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

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

json()

Produce a response with the given JSON body.

TS app/api/route.ts

TypeScript ▾



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

```
export async function GET(request: Request) {  
  return NextResponse.json({ error: 'Internal  
  '}  
}
```

redirect()

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

```
import { NextResponse } from 'next/server'  
  
return NextResponse.redirect(new URL('/new',
```

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

```
import { NextResponse } from 'next/server'  
  
// Given an incoming request...  
const loginUrl = new URL('/login', request.url)  
// Add ?from=/incoming-url to the /login URL  
loginUrl.searchParams.set('from', request.nextUrl)  
// And redirect to the new URL  
return NextResponse.redirect(loginUrl)
```

rewrite()

Produce a response that rewrites (proxies) the given [URL ↗](#) while preserving the original URL.

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

// Incoming request: /about, browser shows /a
// Rewritten request: /proxy, browser shows /
return NextResponse.rewrite(new URL('/proxy',
```

next()

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

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

return NextResponse.next()
```

You can also forward `headers` upstream when producing the response, using

```
NextResponse.next({ request: { headers } })
:
```

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

// Given an incoming request...
const newHeaders = new Headers(request.headers)
// Add a new header
newHeaders.set('x-version', '123')
// Forward the modified request headers upstream
return NextResponse.next({
  request: {
    // New request headers
    headers: newHeaders,
  },
})
```

This forwards `newHeaders` upstream to the target page, route, or server action, and does not expose them to the client. While this pattern is useful for passing data upstream, it should be used with

caution because the headers containing this data may be forwarded to external services.

In contrast, `NextResponse.next({ headers })` is a shorthand for sending headers from middleware to the client. This is **NOT** good practice and should be avoided. Among other reasons because setting response headers like `Content-Type`, can override framework expectations (for example, the `Content-Type` used by Server Actions), leading to failed submissions or broken streaming responses.

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

async function middleware(request: NextRequest) {
  const headers = await injectAuth(request.headers)
  // DO NOT forward headers like this
  return NextResponse.next({ headers })
}
```

In general, avoid copying all incoming request headers because doing so can leak sensitive data to clients or upstream services.

Prefer a defensive approach by creating a subset of incoming request headers using an allow-list. For example, you might discard custom `x-*` headers and only forward known-safe headers:

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

function middleware(request: NextRequest) {
  const incoming = new Headers(request.headers)
  const forwarded = new Headers()

  for (const [name, value] of incoming) {
    const headerName = name.toLowerCase()
    // Keep only known-safe headers, discard others
    if (
      !headerName.startsWith('x-') &&
      headerName !== 'authorization' &&
      headerName !== 'cookie'
    ) {
      // Preserve original header name casing
      forwarded.set(name, value)
    }
  }
  return NextResponse.next({ headers: forwarded })
}
```

```
        }
    }

    return NextResponse.next({
        request: {
            headers: forwarded,
        },
    })
}
```

Was this helpful?     



Using App Router
Features available in /app



notFound



Latest Version
15.5.4



The `notFound` function allows you to render the `not-found file` within a route segment as well as inject a

```
<meta name="robots" content="noindex" />  
tag.
```

notFound()

Invoking the `notFound()` function throws a `NEXT_HTTP_ERROR_FALLBACK;404` error and terminates rendering of the route segment in which it was thrown. Specifying a `not-found file` allows you to gracefully handle such errors by rendering a Not Found UI within the segment.

app/user/[id]/page.js

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

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

export default async function Profile({ params }) {
  const { id } = params
  const user = await fetchUser(id)

  if (!user) {
    notFound()
  }

  // ...
}
```

}

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

Version History

Version	Changes
v13.0.0	<code>notFound</code> introduced.

Was this helpful?    



Using App Router
Features available in /app



permanentRedirect



Latest Version
15.5.4



The `permanentRedirect` function allows you to redirect the user to another URL.

`permanentRedirect` can be used in Server Components, Client Components, [Route Handlers](#), and [Server Actions](#).

When used in a streaming context, this will insert a meta tag to emit the redirect on the client side.

When used in a server action, it will serve a 303 HTTP redirect response to the caller. Otherwise, it will serve a 308 (Permanent) HTTP redirect response to the caller.

If a resource doesn't exist, you can use the `notFound` function instead.

Good to know: If you prefer to return a 307 (Temporary) HTTP redirect instead of 308 (Permanent), you can use the `redirect` function instead.

Parameters

The `permanentRedirect` function accepts two arguments:

```
permanentRedirect(path, type)
```

Parameter	Type	Description
path	string	The URL to redirect to. Can be a relative or absolute path.
type	'replace' (default) or 'push' (default in Server Actions)	The type of redirect to perform.

By default, `permanentRedirect` will use `push` (adding a new entry to the browser history stack) in [Server Actions](#) and `replace` (replacing the current URL in the browser history stack) everywhere else. You can override this behavior by specifying the `type` parameter.

The `type` parameter has no effect when used in Server Components.

Returns

`permanentRedirect` does not return a value.

Example

Invoking the `permanentRedirect()` function throws a `NEXT_REDIRECT` error and terminates rendering of the route segment in which it was thrown.

```
JS app/team/[id]/page.js Copy  
  
import { permanentRedirect } from 'next/navigation'
```

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

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

  // ...
}
```

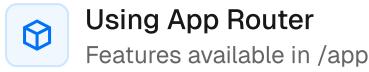
Good to know: `permanentRedirect` does not require you to use `return permanentRedirect()` as it uses the TypeScript `never` ↗ type.

Next Steps

redirect

API Reference for the redirect function.

Was this helpful?     



redirect

The `redirect` function allows you to redirect the user to another URL. `redirect` can be used while rendering in [Server and Client Components](#), [Route Handlers](#), and [Server Actions](#).

When used in a [streaming context](#), this will insert a meta tag to emit the redirect on the client side.

When used in a server action, it will serve a 303 HTTP redirect response to the caller. Otherwise, it will serve a 307 HTTP redirect response to the caller.

If a resource doesn't exist, you can use the `notFound` function instead.

Reference

Parameters

The `redirect` function accepts two arguments:

```
redirect(path, type)
```

Parameter	Type	Description
<code>path</code>	<code>string</code>	The URL to redirect to. Can be a relative or absolute path.

Parameter	Type	Description
-----------	------	-------------

<code>type</code>	'replace' (default) or 'push' (default in Server Actions)	The type of redirect to perform.
-------------------	---	-------------------------------------

By default, `redirect` will use `push` (adding a new entry to the browser history stack) in [Server Actions](#) and `replace` (replacing the current URL in the browser history stack) everywhere else. You can override this behavior by specifying the `type` parameter.

The `RedirectType` object contains the available options for the `type` parameter.

```
import { redirect, RedirectType } from 'next'

redirect('/redirect-to', RedirectType.replace
// or
redirect('/redirect-to', RedirectType.push)
```

The `type` parameter has no effect when used in Server Components.

Returns

`redirect` does not return a value.

Behavior

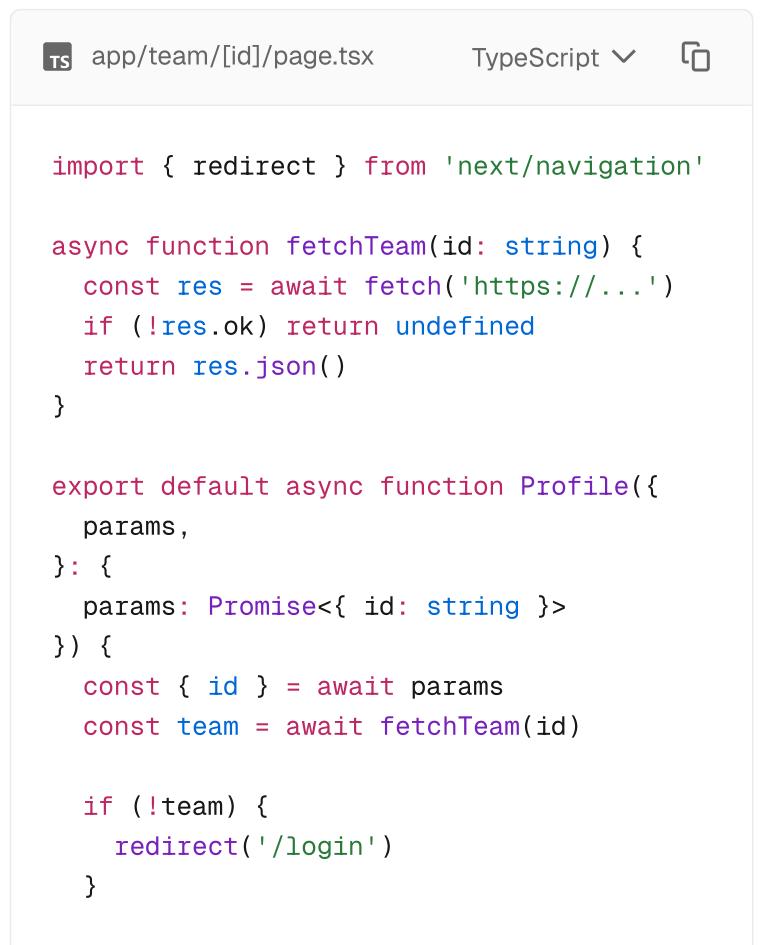
- In Server Actions and Route Handlers, `redirect` should be called **outside** the `try` block when using `try/catch` statements.
- If you prefer to return a 308 (Permanent) HTTP redirect instead of 307 (Temporary), you can use the `permanentRedirect` function instead.

- `redirect` throws an error so it should be called **outside** the `try` block when using `try/catch` statements.
 - `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](#).
-

Example

Server Component

Invoking the `redirect()` function throws a `NEXT_REDIRECT` error and terminates rendering of the route segment in which it was thrown.



```
TS app/team/[id]/page.tsx TypeScript ▾
```

```
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: Promise<{ id: string }>
}) {
  const { id } = await params
  const team = await fetchTeam(id)

  if (!team) {
    redirect('/login')
  }
}
```

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

Good to know: `redirect` does not require you to use `return redirect()` as it uses the TypeScript `never ↗ type`.

Client Component

`redirect` can be directly used in a Client Component.

```
TS components/client-redirect.t... TypeScript ▾ ⌂
use client

import { redirect, usePathname } from 'next/n

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

  if (pathname.startsWith('/admin') && !pathn
    redirect('/admin/login')
  }

  return <div>Login Page</div>
}
```

Good to know: When using `redirect` in a Client Component on initial page load during Server-Side Rendering (SSR), it will perform a server-side redirect.

`redirect` can be used in a Client Component through a Server Action. If you need to use an event handler to redirect the user, you can use the `useRouter` hook.

```
TS app/client-redirect.tsx TypeScript ▾ ⌂
use client

import { navigate } from './actions'
```

```
export function ClientRedirect() {
  return (
    <form action={navigate}>
      <input type="text" name="id" />
      <button>Submit</button>
    </form>
  )
}
```

app/actions.ts TypeScript ▾

```
'use server'

import { redirect } from 'next/navigation'

export async function navigate(data: FormData) {
  redirect(`/posts/${data.get('id')}`)
}
```

FAQ

Why does `redirect` use 307 and 308?

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

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

create a new user, you should be making a `POST` request to `/people`, and not a `GET` request.

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

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

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

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

Version History

Version	Changes
<code>v13.0.0</code>	<code>redirect</code> introduced.

Next Steps

`permanentR...`

API Reference for
the
`permanentRedire...`

Was this helpful?





Using App Router
Features available in /app



revalidatePath



Latest Version
15.5.4



`revalidatePath` allows you to invalidate [cached data](#) on-demand for a specific path.

Usage

`revalidatePath` can be called in Server Functions and Route Handlers.

`revalidatePath` cannot be called in Client Components or Middleware, as it only works in server environments.

Good to know:

- **Server Functions:** Updates the UI immediately (if viewing the affected path). Currently, it also causes all previously visited pages to refresh when navigated to again. This behavior is temporary and will be updated in the future to apply only to the specific path.
- **Route Handlers:** Marks the path for revalidation. The revalidation is done on the next visit to the specified path. This means calling `revalidatePath` with a dynamic route segment will not immediately trigger many revalidations at once. The invalidation only happens when the path is next visited.

Parameters

```
revalidatePath(path: string, type?: 'page' |
```

- `path` : Either a route pattern corresponding to the data you want to revalidate, for example `/product/[slug]`, or a specific URL, `/product/123`. Do not append `/page` or `/layout`, use the `type` parameter instead. Must not exceed 1024 characters. This value is case-sensitive.
- `type` : (optional) `'page'` or `'layout'` string to change the type of path to revalidate. If `path` contains a dynamic segment, for example `/product/[slug]`, this parameter is required. If `path` is a specific URL, `/product/1`, omit `type`.

Use a specific URL when you want to refresh a [single page](#). Use a route pattern plus `type` to refresh [multiple URLs](#).

Returns

`revalidatePath` does not return a value.

What can be invalidated

The path parameter can point to pages, layouts, or route handlers:

- **Pages:** Invalidates the specific page
- **Layouts:** Invalidates the layout (the `layout.tsx` at that segment), all nested layouts beneath it, and all pages beneath them

- **Route Handlers:** Invalidates Data Cache entries accessed within route handlers. For example `revalidatePath("/api/data")` invalidates this GET handler:

```
ts app/api/data/route.ts
```

```
export async function GET() {
  const data = await fetch('https://api.vercel
    cache: 'force-cache',
  })

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

Relationship with `revalidateTag`

`revalidatePath` and `revalidateTag` serve different purposes:

- `revalidatePath`: Invalidates a specific page or layout path
- `revalidateTag`: Invalidates data with specific tags across all pages that use those tags

When you call `revalidatePath`, only the specified path gets fresh data on the next visit. Other pages that use the same data tags will continue to serve cached data until those specific tags are also revalidated:

```
// Page A: /blog
const posts = await fetch('https://api.vercel
  next: { tags: ['posts'] },
)

// Page B: /dashboard
const recentPosts = await fetch('https://api.
  next: { tags: ['posts'] },
)
```

```
})
```

After calling `revalidatePath('/blog')`:

- **Page A (/blog)**: Shows fresh data (page re-rendered)
- **Page B (/dashboard)**: Still shows stale data (cache tag 'posts' not invalidated)

Building revalidation utilities

`revalidatePath` and `revalidateTag` are complementary primitives that are often used together in utility functions to ensure comprehensive data consistency across your application:

```
'use server'

import { revalidatePath, revalidateTag } from

export async function updatePost() {
  await updatePostInDatabase()

  revalidatePath('/blog') // Refresh the blog
  revalidateTag('posts') // Refresh all pages
}
```

This pattern ensures that both the specific page and any other pages using the same data remain consistent.

Examples

Revalidating a specific URL

```
import { revalidatePath } from 'next/cache'
revalidatePath('/blog/post-1')
```

This will invalidate one specific URL for revalidation on the next page visit.

Revalidating a Page path

```
import { revalidatePath } from 'next/cache'  
revalidatePath('/blog/[slug]', 'page')  
// or with route groups  
revalidatePath('/(main)/blog/[slug]', 'page')
```

This will invalidate any URL that matches the provided `page` file for revalidation on the next page visit. This will *not* invalidate pages beneath the specific page. For example, `/blog/[slug]` won't invalidate `/blog/[slug]/[author]`.

Revalidating a Layout path

```
import { revalidatePath } from 'next/cache'  
revalidatePath('/blog/[slug]', 'layout')  
// or with route groups  
revalidatePath('/(main)/post/[slug]', 'layout')
```

This will invalidate any URL that matches the provided `layout` file for revalidation on the next page visit. This will cause pages beneath with the same layout to be invalidated and revalidated on the next visit. For example, in the above case, `/blog/[slug]/[another]` would also be invalidated and revalidated on the next visit.

Revalidating all data

```
import { revalidatePath } from 'next/cache'  
  
revalidatePath('/', 'layout')
```

This will purge the Client-side Router Cache, and invalidate the Data Cache for revalidation on the

next page visit.

Server Function



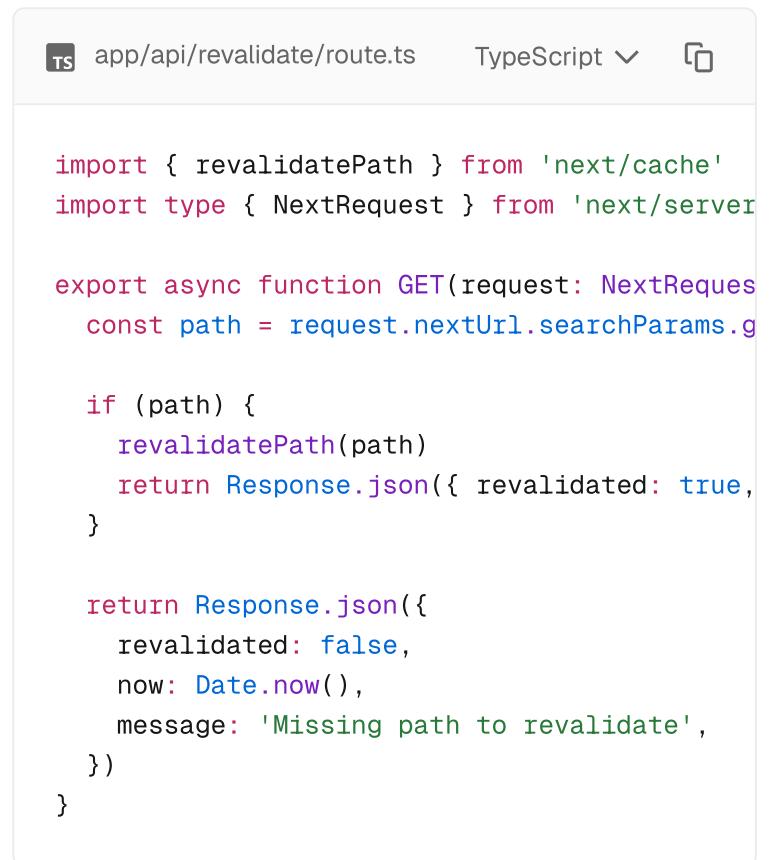
```
TS app/actions.ts TypeScript ▾ ⌂

'use server'

import { revalidatePath } from 'next/cache'

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

Route Handler



```
TS app/api/revalidate/route.ts TypeScript ▾ ⌂

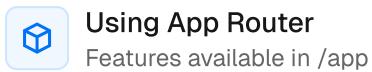
import { revalidatePath } from 'next/cache'
import type { NextRequest } from 'next/server'

export async function GET(request: NextRequest) {
  const path = request.nextUrl.searchParams.get('path')

  if (path) {
    revalidatePath(path)
    return Response.json({ revalidated: true })
  }

  return Response.json({
    revalidated: false,
    now: Date.now(),
    message: 'Missing path to revalidate',
  })
}
```

Was this helpful?    



revalidateTag

`revalidateTag` allows you to invalidate [cached data](#) on-demand for a specific cache tag.

Usage

`revalidateTag` can be called in Server Functions and Route Handlers.

`revalidateTag` cannot be called in Client Components or Middleware, as it only works in server environments.

Good to know: `revalidateTag` marks tagged data as stale, but fresh data is only fetched when pages using that tag are next visited. This means calling `revalidateTag` will not immediately trigger many revalidations at once. The invalidation only happens when any page using that tag is next visited.

Parameters

```
revalidateTag(tag: string): void;
```

- `tag`: A string representing the cache tag associated with the data you want to revalidate. Must not exceed 256 characters. This value is case-sensitive.

You can add tags to `fetch` as follows:

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

Returns

`revalidateTag` does not return a value.

Relationship with `revalidatePath`

`revalidateTag` invalidates data with specific tags across all pages that use those tags, while `revalidatePath` invalidates specific page or layout paths.

Good to know: These functions serve different purposes and may need to be used together for comprehensive data consistency. For detailed examples and considerations, see [Relationship with revalidateTag](#).

Examples

Server Action

TS app/actions.ts

TypeScript ▾



```
'use server'

import { revalidateTag } from 'next/cache'

export default async function submit() {
```

```
    await addPost()
    revalidateTag('posts')
}
```

Route Handler

```
TS app/api/revalidate/route.ts TypeScript ▾
```

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

export async function GET(request: NextRequest) {
  const tag = request.nextUrl.searchParams.get('tag')

  if (tag) {
    revalidateTag(tag)
    return Response.json({ revalidated: true })
  }

  return Response.json({
    revalidated: false,
    now: Date.now(),
    message: 'Missing tag to revalidate',
  })
}
```

Was this helpful?    



Using App Router
Features available in /app



unauthorized



Latest Version
15.5.4



This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

The `unauthorized` function throws an error that renders a Next.js 401 error page. It's useful for handling authorization errors in your application. You can customize the UI using the `unauthorized.js` file.

To start using `unauthorized`, enable the experimental `authInterrupts` configuration option in your `next.config.js` file:

next.config.ts TypeScript ▾

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

const nextConfig: NextConfig = {
  experimental: {
    authInterrupts: true,
  },
}

export default nextConfig
```

`unauthorized` can be invoked in [Server Components](#), [Server Actions](#), and [Route Handlers](#).

app/dashboard/page.tsx TypeScript ▾

```
import { verifySession } from '@/app/lib/dal'
import { unauthorized } from 'next/navigation'

export default async function DashboardPage()
```

```
const session = await verifySession()

if (!session) {
  unauthorized()
}

// Render the dashboard for authenticated users
return (
  <main>
    <h1>Welcome to the Dashboard</h1>
    <p>Hi, {session.user.name}.</p>
  </main>
)
}
```

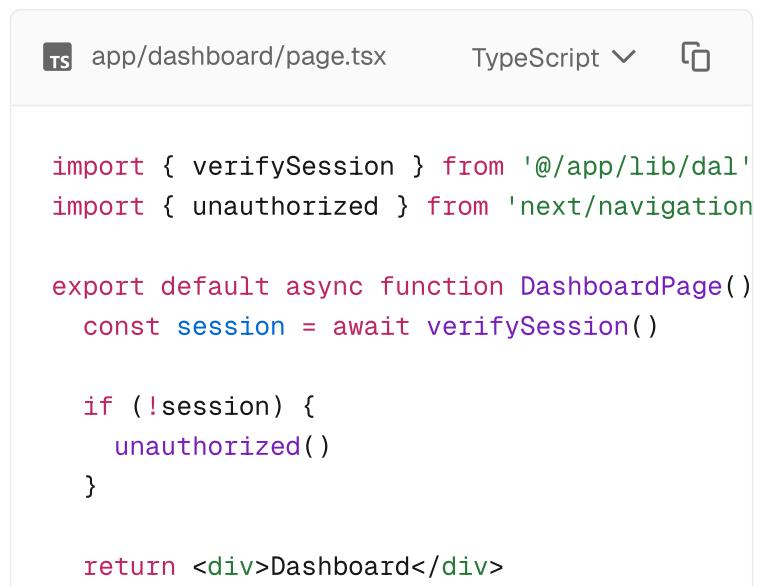
Good to know

- The `unauthorized` function cannot be called in the [root layout](#).

Examples

Displaying login UI to unauthenticated users

You can use `unauthorized` function to display the `unauthorized.js` file with a login UI.



```
TS app/dashboard/page.tsx TypeScript ▾ ⌂

import { verifySession } from '@/app/lib/dal'
import { unauthorized } from 'next/navigation'

export default async function DashboardPage() {
  const session = await verifySession()

  if (!session) {
    unauthorized()
  }

  return <div>Dashboard</div>
}
```

```
}
```

TS app/unauthorized.tsx

TypeScript ▾

```
import Login from '@/app/components/Login'

export default function UnauthorizedPage() {
  return (
    <main>
      <h1>401 - Unauthorized</h1>
      <p>Please log in to access this page.</p>
      <Login />
    </main>
  )
}
```

Mutations with Server Actions

You can invoke `unauthorized` in Server Actions to ensure only authenticated users can perform specific mutations.

TS app/actions/update-profile.ts

TypeScript ▾

```
'use server'

import { verifySession } from '@/app/lib/dal'
import { unauthorized } from 'next/navigation'
import db from '@/app/lib/db'

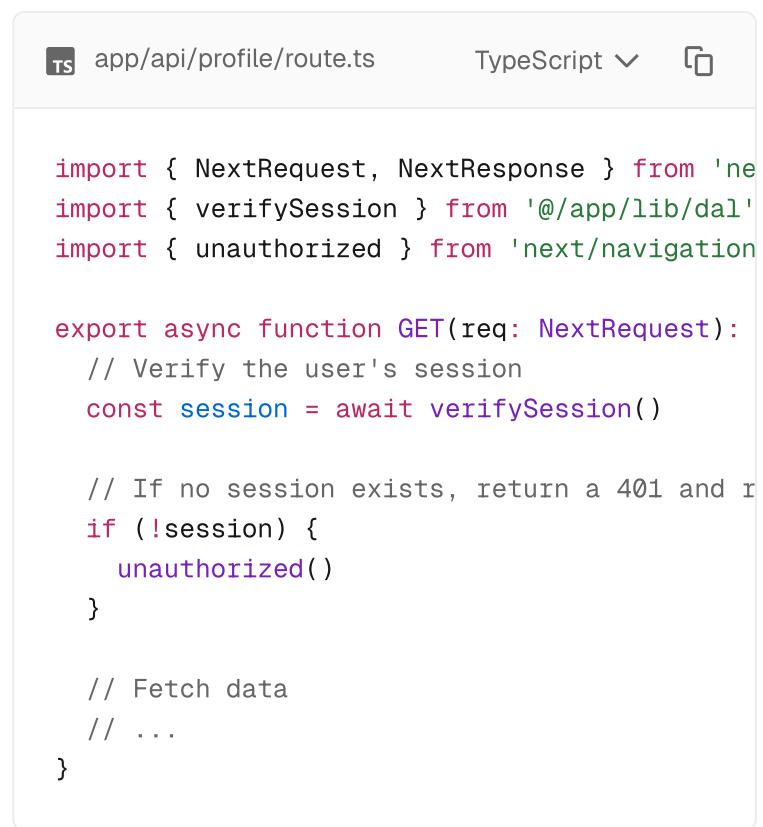
export async function updateProfile(data: For
  const session = await verifySession()

  // If the user is not authenticated, return
  if (!session) {
    unauthorized()
  }

  // Proceed with mutation
  // ...
}
```

Fetching data with Route Handlers

You can use `unauthorized` in Route Handlers to ensure only authenticated users can access the endpoint.



```
TS app/api/profile/route.ts TypeScript ▾
```

```
import { NextRequest, NextResponse } from 'next/server'
import { verifySession } from '@/app/lib/dal'
import { unauthorized } from 'next/navigation'

export async function GET(req: NextRequest): Promise<NextResponse> {
  // Verify the user's session
  const session = await verifySession()

  // If no session exists, return a 401 and return early
  if (!session) {
    return unauthorized()
  }

  // Fetch data
  // ...
}
```

Version History

Version	Changes
v15.1.0	<code>unauthorized</code> introduced.

Next Steps

unauthorize...

API reference for
the
unauthorized.js...

Was this helpful?    



Using App Router
Features available in /app



unstable_cache



Latest Version
15.5.4



Warning: This API will be replaced by [use cache](#) when it reaches stability.

`unstable_cache` allows you to cache the results of expensive operations, like database queries, and reuse them across multiple requests.

```
import { getUser } from './data';
import { unstable_cache } from 'next/cache';

const getCachedUser = unstable_cache(
  async (id) => getUser(id),
  ['my-app-user']
);

export default async function Component({ use
  const user = await getCachedUser(userID);
  ...
}
```

Good to know:

- Accessing dynamic data sources such as `headers` or `cookies` inside a cache scope is not supported. If you need this data inside a cached function use `headers` outside of the cached function and pass the required dynamic data in as an argument.
- This API uses Next.js' built-in [Data Cache](#) to persist the result across requests and deployments.

Parameters

```
const data = unstable_cache(fetchData, keyPar
```

- `fetchData` : This is an asynchronous function that fetches the data you want to cache. It must be a function that returns a `Promise`.
- `keyParts` : This is an extra array of keys that further adds identification to the cache. By default, `unstable_cache` already uses the arguments and the stringified version of your function as the cache key. It is optional in most cases; the only time you need to use it is when you use external variables without passing them as parameters. However, it is important to add closures used within the function if you do not pass them as parameters.
- `options` : This is an object that controls how the cache behaves. It can contain the following properties:
 - `tags` : An array of tags that can be used to control cache invalidation. Next.js will not use this to uniquely identify the function.
 - `revalidate` : The number of seconds after which the cache should be revalidated. Omit or pass `false` to cache indefinitely or until matching `revalidateTag()` or `revalidatePath()` methods are called.

Returns

`unstable_cache` returns a function that when invoked, returns a `Promise` that resolves to the cached data. If the data is not in the cache, the provided function will be invoked, and its result will be cached and returned.

Example

```
TS app/page.tsx TypeScript ▾ ⌂

import { unstable_cache } from 'next/cache'

export default async function Page({
  params,
}: {
  params: Promise<{ userId: string }>
}) {
  const { userId } = await params
  const getCachedUser = unstable_cache(
    async () => {
      return { id: userId }
    },
    [userId], // add the user ID to the cache
  {
    tags: ['users'],
    revalidate: 60,
  }
)

// ...
}
```

Version History

Version	Changes
v14.0.0	unstable_cache introduced.

Was this helpful?    

Using App Router

Features available in /app



Latest Version

15.5.4



unstable_noStore

This is a legacy API and no longer recommended.
It's still supported for backward compatibility.

In version 15, we recommend using [connection](#) instead of `unstable_noStore`.

`unstable_noStore` can be used to declaratively opt out of static rendering and indicate a particular component should not be cached.

```
import { unstable_noStore as noStore } from 'next'

export default async function ServerComponent() {
  const result = await db.query(...);
  ...
}
```

Good to know:

- `unstable_noStore` is equivalent to `cache: 'no-store'` on a `fetch`
- `unstable_noStore` is preferred over `export const dynamic = 'force-dynamic'` as it is more granular and can be used on a per-component basis

- Using `unstable_noStore` inside `unstable_cache` will not opt out of static generation. Instead, it will defer to the cache configuration to determine whether to cache the result or not.

Usage

If you prefer not to pass additional options to `fetch`, like `cache: 'no-store'`, `next: { revalidate: 0 }` or in cases where `fetch` is not available, you can use `noStore()` as a replacement for all of these use cases.

```
import { unstable_noStore as noStore } from 'next'

export default async function ServerComponent() {
  const result = await db.query(...);
  ...
}
```

Version History

Version	Changes
v15.0.0	<code>unstable_noStore</code> deprecated for connection.
v14.0.0	<code>unstable_noStore</code> introduced.

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

unstable_rethrow

This feature is currently unstable and subject to change, it's not recommended for production. Try it out and share your feedback on [GitHub](#).

`unstable_rethrow` can be used to avoid catching internal errors thrown by Next.js when attempting to handle errors thrown in your application code.

For example, calling the `notFound` function will throw an internal Next.js error and render the `not-found.js` component. However, if used inside the `try` block of a `try/catch` statement, the error will be caught, preventing `not-found.js` from rendering:

@/app/ui/component.tsx

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

export default async function Page() {
  try {
    const post = await fetch('https://.../pos')
    if (res.status === 404) notFound()
    if (!res.ok) throw new Error(res.statusText)
    return res.json()
  }
} catch (err) {
  console.error(err)
}
}
```

You can use `unstable_rethrow` API to re-throw the internal error and continue with the expected behavior:



```

import { notFound, unstable_rethrow } from 'next'

export default async function Page() {
  try {
    const post = await fetch('https://.../post')
    if (res.status === 404) notFound()
    if (!res.ok) throw new Error(res.statusText)
    return res.json()
  }
} catch (err) {
  unstable_rethrow(err)
  console.error(err)
}
}

```

The following Next.js APIs rely on throwing an error which should be rethrown and handled by Next.js itself:

- `notFound()`
- `redirect()`
- `permanentRedirect()`

If a route segment is marked to throw an error unless it's static, a Dynamic API call will also throw an error that should similarly not be caught by the developer. Note that Partial Prerendering (PPR) affects this behavior as well. These APIs are:

- `cookies`
- `headers`
- `searchParams`
- `fetch(... , { cache: 'no-store' })`
- `fetch(... , { next: { revalidate: 0 } })`

Good to know:

- This method should be called at the top of the catch block, passing the error object as its only argument. It can also be used within a `.catch` handler of a promise.

- You may be able to avoid using `unstable_rethrow` if you encapsulate your API calls that throw and let the **caller** handle the exception.
- Only use `unstable_rethrow` if your caught exceptions may include both application errors and framework-controlled exceptions (like `redirect()` or `notFound()`).
- Any resource cleanup (like clearing intervals, timers, etc) would have to either happen prior to the call to `unstable_rethrow` or within a `finally` block.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

useLinkStatus

The `useLinkStatus` hook lets you tracks the **pending** state of a `<Link>`. You can use it to show inline visual feedback to the user (like spinners or text glimmers) while a navigation to a new route completes.

`useLinkStatus` is useful when:

- Prefetching is disabled or in progress meaning navigation is blocked.
- The destination route is dynamic **and** doesn't include a `loading.js` file that would allow an instant navigation.

TS app/loading-indicator.tsx TypeScript [Copy](#)

```
'use client'

import { useLinkStatus } from 'next/link'

export default function LoadingIndicator() {
  const { pending } = useLinkStatus()
  return pending ? (
    <div role="status" aria-label="Loading" c
  ) : null
}
```

TS app/header.tsx TypeScript [Copy](#)

```
import Link from 'next/link'
import LoadingIndicator from './loading-indic

export default function Header() {
  return (
    <header>
      <Link href="/dashboard" prefetch={false}
```

```
Dashboard <LoadingIndicator />
</Link>
</header>
)
}
```

Good to know:

- `useLinkStatus` must be used within a descendant component of a `Link` component
- The hook is most useful when `prefetch={false}` is set on the `Link` component
- If the linked route has been prefetched, the pending state will be skipped
- When clicking multiple links in quick succession, only the last link's pending state is shown
- This hook is not supported in the Pages Router and will always return `{ pending: false }`

Parameters

```
const { pending } = useLinkStatus()
```

`useLinkStatus` does not take any parameters.

Returns

`useLinkStatus` returns an object with a single property:

Property	Type	Description
----------	------	-------------

pending	boolean	<code>true</code> before history updates, <code>false</code> after
---------	---------	---

Example

Inline loading indicator

It's helpful to add visual feedback that navigation is happening in case the user clicks a link before prefetching is complete.

```
TS app/components/loading-in... TypeScript ▾
```

```
'use client'

import { useLinkStatus } from 'next/link'

export default function LoadingIndicator() {
  const { pending } = useLinkStatus()
  return pending ? (
    <div role="status" aria-label="Loading" c
  ) : null
}
```

```
TS app/shop/layout.tsx TypeScript ▾
```

```
import Link from 'next/link'
import LoadingIndicator from './components/lo

const links = [
  { href: '/shop/electronics', label: 'Electr
  { href: '/shop/clothing', label: 'Clothing'
  { href: '/shop/books', label: 'Books' },
]

function Menubar() {
  return (
    <div>
      {links.map((link) => (
        <Link key={link.label} href={link.hre
          {link.label} <LoadingIndicator />
        </Link>
      )))
    </div>
  )
}

export default function Layout({ children }:
```

```
<Menubar />
  {children}
</div>
)
}
```

Gracefully handling fast navigation

If the navigation to a new route is fast, users may see an unnecessary flash of the loading indicator. One way to improve the user experience and only show the loading indicator when the navigation takes time to complete is to add an initial animation delay (e.g. 100ms) and start the animation as invisible (e.g. `opacity: 0`).

app/styles/global.css

```
.spinner {
/* ... */
  opacity: 0;
  animation:
    fadeIn 500ms 100ms forwards,
    rotate 1s linear infinite;
}

@keyframes fadeIn {
  from {
    opacity: 0;
  }
  to {
    opacity: 1;
  }
}

@keyframes rotate {
  to {
    transform: rotate(360deg);
  }
}
```

Version

v15.3.0

Changes

`useLinkStatus` introduced.

Next Steps

Learn more about the features mentioned in this page by reading the API Reference.

[Link Compon...](#)

Enable fast client-side navigation with the built-in...

[loading.js](#)

API reference for the loading.js file.

Was this helpful?





Using App Router
Features available in /app



useParams



Latest Version
15.5.4



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

TS app/example-client-compon... TypeScript

```
'use client'

import { useParams } from 'next/navigation'

export default function ExampleClientComponen
  const params = useParams<{ tag: string; ite

    // Route -> /shop/[tag]/[item]
    // URL -> /shop/shoes/nike-air-max-97
    // `params` -> { tag: 'shoes', item: 'nike-
      console.log(params)

    return '...'
}
```

Parameters

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

`useParams` does not take any parameters.

Returns

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

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

For example:

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

Version History

Version**Changes**

v13.3.0

useParams introduced.

Was this helpful?    



Using App Router
Features available in /app



usePathname



Latest Version
15.5.4



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

TS app/example-client-compon... TypeScript

```
'use client'

import { usePathname } from 'next/navigation'

export default function ExampleClientComponen
  const pathname = usePathname()
  return <p>Current pathname: {pathname}</p>
}
```

`usePathname` intentionally requires using a **Client Component**. It's important to note Client Components are not a de-optimization. They are an integral part of the **Server Components** architecture.

For example, a Client Component with `usePathname` will be rendered into HTML on the initial page load. When navigating to a new route, this component does not need to be re-fetched. Instead, the component is downloaded once (in the client JavaScript bundle), and re-renders based on the current state.

Good to know:

- Reading the current URL from a **Server Component** is not supported. This design is intentional to support layout state being preserved across page navigations.
- Compatibility mode:

- `usePathname` can return `null` when a [fallback route](#) is being rendered or when a `pages` directory page has been [automatically statically optimized](#) by Next.js and the router is not ready.
- When using `usePathname` with rewrites in `next.config` or [Middleware](#), `useState` and `useEffect` must also be used in order to avoid hydration mismatch errors.
- Next.js will automatically update your types if it detects both an `app` and `pages` directory in your project.

Parameters

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

`usePathname` does not take any parameters.

Returns

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

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

Examples

Do something in response to a route change

```
TS app/example-client-compon... TypeScript ▾ ⌂

'use client'

import { useEffect } from 'react'
import { usePathname, useSearchParams } from 'next/navigation'

function ExampleClientComponent() {
  const pathname = usePathname()
  const searchParams = useSearchParams()
  useEffect(() => {
    // Do something here...
  }, [pathname, searchParams])
}
```

Avoid hydration mismatch with rewrites

When a page is pre-rendered, the HTML is generated for the source pathname. If the page is then reached through a rewrite using

`next.config` or `Middleware`, the browser URL may differ, and `usePathname()` will read the rewritten pathname on the client.

To avoid hydration mismatches, design the UI so that only a small, isolated part depends on the client pathname. Render a stable fallback on the server and update that part after mount.

```
TS app/example-client-compon... TypeScript ▾ ⌂

'use client'

import { useEffect, useState } from 'react'
import { usePathname } from 'next/navigation'
```

```
export default function PathnameBadge() {
  const pathname = usePathname()
  const [clientPathname, setClientPathname] =
    useState(pathname)

  useEffect(() => {
    setClientPathname(pathname)
  }, [pathname])

  return (
    <p>
      Current pathname: <span>{clientPathname}</span>
    </p>
  )
}
```

Version	Changes
v13.0.0	usePathname introduced.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

useReportWebVitals

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

New functions passed to `useReportWebVitals` are called with the available metrics up to that point. To prevent reporting duplicated data, ensure that the callback function reference does not change (as shown in the code examples below).

 app/_components/web-vitals.js 

```
'use client'

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

const logWebVitals = (metric) => {
  console.log(metric)
}

export function WebVitals() {
  useReportWebVitals(logWebVitals)

  return null
}
```

 app/layout.js 

```
import { WebVitals } from './_components/web-vitals'

export default function Layout({ children }) {
  return (
    <html>
      <body>
        <WebVitals />
        {children}
      </body>
    </html>
  )
}
```

```
</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.

useReportWebVitals

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

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

performance. Possible values are "good", "needs-improvement", and "poor". The rating is typically determined by comparing the metric value against predefined thresholds that indicate acceptable or suboptimal performance.

- `value` : The actual value or duration of the performance entry, typically in milliseconds. The value provides a quantitative measure of the performance aspect being tracked by the metric. The source of the value depends on the specific metric being measured and can come from various [Performance API ↗](#)s.

Web Vitals

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

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

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



The screenshot shows a code editor with the following code:

```
TS app/components/web-vitals.... TypeScript ▾ ⏎
'use client'

import { useReportWebVitals } from 'next/web-
type ReportWebVitalsCallback = Parameters<typ
```

```
const handleWebVitals: ReportWebVitalsCallback = metric => {
  switch (metric.name) {
    case 'FCP':
      // handle FCP results
    case 'LCP':
      // handle LCP results
    case 'CLS':
      // ...
  }
}

export function WebVitals() {
  useReportWebVitals(handleWebVitals)
}
```

Sending results to external systems

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

```
function postWebVitals(metrics) {
  const body = JSON.stringify(metrics)
  const url = 'https://example.com/analytics'

  // Use `navigator.sendBeacon()` if available
  if (navigator.sendBeacon) {
    navigator.sendBeacon(url, body)
  } else {
    fetch(url, { body, method: 'POST', keepalive: true })
  }
}

useReportWebVitals(postWebVitals)
```

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
  // https://github.com/vercel/next.js/bl
  window.gtag('event', metric.name, {
    value: Math.round(metric.name === 'CLS'
      ? metric.value : metric.value * 100),
    event_label: metric.id, // id unique to this metric
    non_interaction: true, // avoids affecting other metrics
  });
})
```

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

Was this helpful?    



Using App Router
Features available in /app



useRouter



Latest Version
15.5.4



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

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

TS app/example-client-compon... TypeScript ▾

```
'use client'

import { useRouter } from 'next/navigation'

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

  return (
    <button type="button" onClick={() => rout
      Dashboard
    </button>
  )
}
```

useRouter()

- `router.push(href: string, { scroll: boolean })`: Perform a client-side navigation to the provided route. Adds a new entry into the [browser's history ↗](#) stack.
- `router.replace(href: string, { scroll: boolean })`: Perform a client-side navigation to

the provided route without adding a new entry into the browser's history stack [↗](#).

- `router.refresh()` : Refresh the current route. Making a new request to the server, re-fetching data requests, and re-rendering Server Components. The client will merge the updated React Server Component payload without losing unaffected client-side React (e.g. `useState`) or browser state (e.g. scroll position).
- `router.prefetch(href: string, options?: { onInvalidate?: () => void })` : Prefetch the provided route for faster client-side transitions. The optional `onInvalidate` callback is called when the **prefetched data becomes stale**.
- `router.back()` : Navigate back to the previous route in the browser's history stack.
- `router.forward()` : Navigate forwards to the next page in the browser's history stack.

Good to know:

- You must not send untrusted or unsanitized URLs to `router.push` or `router.replace`, as this can open your site to cross-site scripting (XSS) vulnerabilities. For example, `javascript:` URLs sent to `router.push` or `router.replace` will be executed in the context of your page.
- The `<Link>` component automatically prefetch routes as they become visible in the viewport.
- `refresh()` could re-produce the same result if fetch requests are cached. Other Dynamic APIs like `cookies` and `headers` could also change the response.
- The `onInvalidate` callback is called at most once per prefetch request. It signals when you may want to trigger a new prefetch for updated route data.

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

[View the full migration guide.](#)

Examples

Router events

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

```
js app/components/navigation-events.js

'use client'

import { useEffect } from 'react'
import { usePathname, useSearchParams } from 'next/navigation'

export function NavigationEvents() {
  const pathname = usePathname()
  const searchParams = useSearchParams()

  useEffect(() => {
    const url = `${pathname}?${searchParams}`
    console.log(url)
    // You can now use the current URL
    // ...
  }, [pathname, searchParams])

  return '...'
}
```

Which can be imported into a layout.

```

import { Suspense } from 'react'
import { NavigationEvents } from './component'

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

        <Suspense fallback={null}>
          <NavigationEvents />
        </Suspense>
      </body>
    </html>
  )
}

```

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

Disabling scroll to top

By default, Next.js will scroll to the top of the page when navigating to a new route. You can disable this behavior by passing `scroll: false` to `router.push()` or `router.replace()`.

```

'use client'

import { useRouter } from 'next/navigation'

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

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

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

Version History

Version	Changes
v15.4.0	Optional <code>onInvalidate</code> callback for <code>router.prefetch</code> introduced
v13.0.0	<code>useRouter</code> from <code>next/navigation</code> introduced.

Was this helpful?    



Using App Router
Features available in /app



useSearchParams



Latest Version
15.5.4



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

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

The code editor shows a TypeScript file named `app/dashboard/search-bar.ts` in a `TypeScript` environment. The code uses the `useSearchParams` hook from `'next/navigation'` to get search parameters from the URL. It then extracts the `'search'` parameter and returns it as a prop to a component.

```
'use client'

import { useSearchParams } from 'next/navigation'

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

  const search = searchParams.get('search')

  // URL -> `/dashboard?search=my-project`
  // `search` -> 'my-project'
  return <>Search: {search}</>
}
```

Parameters

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

`useSearchParams` does not take any parameters.

Returns

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

- `URLSearchParams.get()` ↗ : Returns the first value associated with the search parameter.
For example:

URL	<code>searchParams.get("a")</code>
/dashboard?a=1	'1'
/dashboard?a=	''
/dashboard?b=3	null
/dashboard? a=1&a=2	'1' - use <code>getAll()</code> ↗ to get all values

- `URLSearchParams.has()` ↗ : Returns a boolean value indicating if the given parameter exists.
For example:

URL	<code>searchParams.has("a")</code>
/dashboard?a=1	true
/dashboard?b=3	false

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

Good to know:

- `useSearchParams` is a [Client Component](#) hook and is **not supported** in [Server Components](#) to prevent stale values during [partial rendering](#).
- If you want to fetch data in a Server Component based on search params, it's often a better option

to read the `searchParams` prop of the corresponding Page. You can then pass it down by props to any component (Server or Client) within that Page.

- If an application includes the `/pages` directory, `useSearchParams` will return `ReadonlyURLSearchParams | null`. The `null` value is for compatibility during migration since search params cannot be known during pre-rendering of a page that doesn't use `getServerSideProps`

Behavior

Static Rendering

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

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

We recommend wrapping the Client Component that uses `useSearchParams` in a `<Suspense>` boundary. This will allow any Client Components above it to be statically rendered and sent as part of initial HTML. [Example](#).

For example:

```
TS app/dashboard/search-bar.t... TypeScript ▾ ⌂
'use client'

import { useSearchParams } from 'next/navigation'

export default function SearchBar() {
  const searchParams = useSearchParams()
```

```
const search = useParams('search')
```

```
// This will not be logged on the server when  
console.log(search)
```

```
return <>Search: {search}</>
```

```
}
```

TS app/dashboard/page.tsx

TypeScript

```
import { Suspense } from 'react'  
import SearchBar from './search-bar'
```

```
// This component passed as a fallback to the  
// will be rendered in place of the search bar  
// When the value is available during React hydration  
// will be replaced with the `<SearchBar>` component  
function SearchBarFallback() {
```

```
    return <>placeholder</>
```

```
}
```

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

```
    return (
```

```
<>
```

```
<nav>
```

```
<Suspense fallback=<><SearchBarFallback></><SearchBar /></Suspense>
```

```
</nav>
```

```
<h1>Dashboard</h1>
```

```
</>
```

```
)
```

```
}
```

Good to know: In development, all pages are rendered on-demand, so `useSearchParams` doesn't suspend. However, pre-rendering a [static page](#) that worked in development will fail the build if `useSearchParams` is used in a Client Component that isn't directly or indirectly wrapped in a `Suspense` boundary.

Dynamic Rendering

If a route is [dynamically rendered](#),

`useSearchParams` will be available on the server

during the initial server render of the Client Component.

For example:

```
TS app/dashboard/search-bar.tsx TypeScript ▾ ⌂

'use client'

import { useSearchParams } from 'next/navigation'

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

  const search = searchParams.get('search')

  // This will be logged on the server during
  // and on the client on subsequent navigati
  console.log(search)

  return <>Search: {search}</>
}
```

```
TS app/dashboard/page.tsx TypeScript ▾ ⌂

import SearchBar from './search-bar'

export const dynamic = 'force-dynamic'

export default function Page() {
  return (
    <>
      <nav>
        <SearchBar />
      </nav>
      <h1>Dashboard</h1>
    </>
  )
}
```

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

Server Components

Pages

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

Layouts

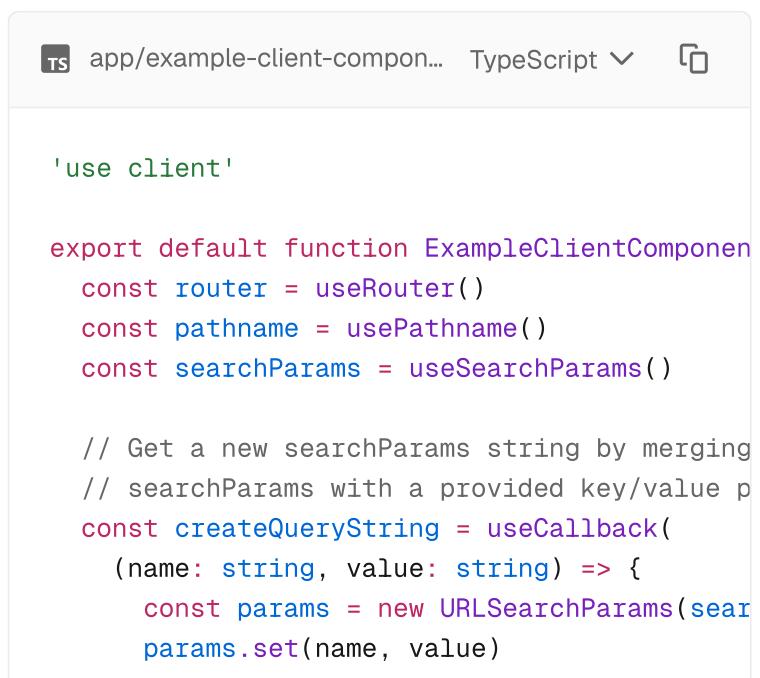
Unlike Pages, [Layouts](#) (Server Components) **do not** receive the `searchParams` prop. This is because a shared layout is [not re-rendered during navigation](#) which could lead to stale `searchParams` between navigations. View [detailed explanation](#).

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

Examples

Updating `searchParams`

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



```
'use client'

export default function ExampleClientComponen
  const router = useRouter()
  const pathname = usePathname()
  const searchParams = useSearchParams()

  // Get a new searchParams string by merging
  // searchParams with a provided key/value p
  const createQueryString = useCallback(
    (name: string, value: string) => {
      const params = new URLSearchParams(sear
        params.set(name, value)
    }
  )
  const handleQueryChange = (query: string) => {
    const parsedQuery = parseQuery(query)
    const newSearchParams = new URLSearchParams(searchParams)
    const newParams = createQueryString(parsedQuery)
    newSearchParams.append('q', newParams.get('q') || '')
    newSearchParams.append('sort', newParams.get('sort') || '')
    newSearchParams.append('order', newParams.get('order') || '')
    router.push({
      pathname: pathname,
      search: newSearchParams.toString(),
    })
  }
  useEffect(() => {
    handleQueryChange(window.location.search)
  }, [pathname, router])
  useEffect(() => {
    window.addEventListener('popstate', handleQueryChange)
    return () => {
      window.removeEventListener('popstate', handleQueryChange)
    }
  }, [handleQueryChange])
}

function parseQuery(query: string): Record<string, string> {
  const params = new URLSearchParams(query)
  const parsedQuery: Record<string, string> = {}
  for (let [key, value] of params.entries()) {
    if (key === 'q' || key === 'sort' || key === 'order') {
      parsedQuery[key] = value
    } else {
      parsedQuery[key] = value
    }
  }
  return parsedQuery
}
```

```
        return params.toString()
    },
    [searchParams]
)

return (
<>
<p>Sort By</p>

/* using useRouter */
<button
onClick={() => {
// <pathname>?sort=asc
router.push(pathname + '?'+ create
}}
>
ASC
</button>

/* using <Link> */
<Link
href={
// <pathname>?sort=desc
pathname + '?' + createQueryString(
}
>
DESC
</Link>
</>
)
}
```

Version History

Version	Changes
v13.0.0	useSearchParams introduced.

Was this helpful?    



Using App Router
Features available in /app



useSelectedLayoutSegment



Latest Version
15.5.4

`useSelectedLayoutSegment` is a **Client Component**

hook that lets you read the active route segment **one level below** the Layout it is called from.

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

TS app/example-client-compon... TypeScript

```
'use client'

import { useSelectedLayoutSegment } from 'next'

export default function ExampleClientComponen
  const segment = useSelectedLayoutSegment()

  return <p>Active segment: {segment}</p>
}
```

Good to know:

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

Parameters

```
const segment = useSelectedLayoutSegment(para
```

`useSelectedLayoutSegment` optionally accepts a `parallelRoutesKey`, which allows you to read the active route segment within that slot.

Returns

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

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

Layout	Visited URL
app/layout.js	/
app/layout.js	/dashboard
app/dashboard/layout.js	/dashboard
app/dashboard/layout.js	/dashboard/settings
app/dashboard/layout.js	/dashboard/analytics
app/dashboard/layout.js	/dashboard/analytics/mon

Examples

Creating an active link component

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

```
TS app/blog/blog-nav-link.tsx TypeScript ▾ ⌂

'use client'

import Link from 'next/link'
import { useSelectedLayoutSegment } from 'next'

// This *client* component will be imported in the sidebar
export default function BlogNavLink({
  slug,
  children,
}: {
  slug: string
  children: React.ReactNode
}) {
  // Navigating to `/blog/hello-world` will result in this component
  // for the selected layout segment
  const segment = useSelectedLayoutSegment()
  const isActive = slug === segment

  return (
    <Link
      href={`/blog/${slug}`}
      // Change style depending on whether this is the active segment
      style={{ fontWeight: isActive ? 'bold' : 'normal' }}
    >
      {children}
    </Link>
  )
}
```

```
TS app/blog/layout.tsx TypeScript ▾ ⌂

// Import the Client Component into a parent
import { BlogNavLink } from './blog-nav-link'
import getFeaturedPosts from './get-featured-posts'

export default async function Layout({
  children,
}: {
  children: React.ReactNode
}) {
  const featuredPosts = await getFeaturedPosts()
  return (
    <BlogNavLink slug="featured" >
      {featuredPosts.map(post => (
        <div key={post.id}>
          <h2>{post.title}</h2>
          <p>{post.description}</p>
        </div>
      ))}
    </BlogNavLink>
  )
}
```

```
<div>
  {featuredPosts.map((post) => (
    <div key={post.id}>
      <BlogNavLink slug={post.slug}>{post
    </div>
  ))}
  <div>{children}</div>
</div>
)
}
```

Version History

Version	Changes
v13.0.0	useSelectedLayoutSegment introduced.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

useSelectedLayoutSegments

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

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

```
TS app/example-client-compon... TypeScript ▾ ⌂

'use client'

import { useSelectedLayoutSegments } from 'next/app'

export default function ExampleClientComponen
  const segments = useSelectedLayoutSegments()

  return (
    <ul>
      {segments.map((segment, index) => (
        <li key={index}>{segment}</li>
      ))}
    </ul>
  )
}
```

Good to know:

- Since `useSelectedLayoutSegments` is a **Client Component** hook, and Layouts are **Server Components** by default, `useSelectedLayoutSegments` is usually called via a Client Component that is imported into a Layout.
- The returned segments include **Route Groups**, which you might not want to be included in your

UI. You can use the `filter` ↗ array method to remove items that start with a bracket.

Parameters

```
const segments = useSelectedLayoutSegments(pa
```

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

Returns

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

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

Layout	Visited URL	Result
app/layout.js	/	[]
app/layout.js	/dashboard	['']
app/layout.js	/dashboard/settings	['']
app/dashboard/layout.js	/dashboard	[]
app/dashboard/layout.js	/dashboard/settings	['']

Version History

Version Changes

v13.0.0

`useSelectedLayoutSegments` introduced.

Was this helpful?



 **Using App Router**
Features available in /app

 **Latest Version**
15.5.4

userAgent

The `userAgent` helper extends the [Web Request API](#) with additional properties and methods to interact with the user agent object from the request.

TS middleware.ts

TypeScript



```
import { NextRequest, NextResponse, userAgent }

export function middleware(request: NextRequest) {
  const url = request.nextUrl
  const { device } = userAgent(request)

  // device.type can be: 'mobile', 'tablet',
  // 'wearable', 'embedded', or undefined (for
  const viewport = device.type || 'desktop'

  url.searchParams.set('viewport', viewport)
  return NextResponse.rewrite(url)
}
```

isBot

A boolean indicating whether the request comes from a known bot.

browser

An object containing information about the browser used in the request.

- `name` : A string representing the browser's name, or `undefined` if not identifiable.
 - `version` : A string representing the browser's version, or `undefined`.
-

device

An object containing information about the device used in the request.

- `model` : A string representing the model of the device, or `undefined`.
 - `type` : A string representing the type of the device, such as `console`, `mobile`, `tablet`, `smarttv`, `wearable`, `embedded`, or `undefined`.
 - `vendor` : A string representing the vendor of the device, or `undefined`.
-

engine

An object containing information about the browser's engine.

- `name` : A string representing the engine's name. Possible values include: `Amaya`, `Blink`, `EdgeHTML`, `Flow`, `Gecko`, `Goanna`, `iCab`, `KHTML`, `Links`, `Lynx`, `NetFront`, `NetSurf`, `Presto`, `Tasman`, `Trident`, `w3m`, `WebKit` or `undefined`.
- `version` : A string representing the engine's version, or `undefined`.

os

An object containing information about the operating system.

- `name` : A string representing the name of the OS, or `undefined`.
- `version` : A string representing the version of the OS, or `undefined`.

cpu

An object containing information about the CPU architecture.

- `architecture` : A string representing the architecture of the CPU. Possible values include: `68k`, `amd64`, `arm`, `arm64`, `armhf`, `avr`, `ia32`, `ia64`, `irix`, `irix64`, `mips`, `mips64`, `pa-risc`, `ppc`, `sparc`, `sparc64` or `undefined`

Was this helpful?    



Using App Router
Features available in /app



Configuration



Latest Version
15.5.4



next.config.js

Learn how to
configure your
application with...

TypeScript

Next.js provides a
TypeScript-first
development...

ESLint

Learn how to use
and configure the
ESLint plugin to...

Was this helpful?





Using App Router

Features available in /app



Latest Version

15.5.4



next.config.js

Next.js can be configured through a

`next.config.js` file in the root of your project directory (for example, by `package.json`) with a default export.

`next.config.js`

```
// @ts-check

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

module.exports = nextConfig
```

ECMAScript Modules

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

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

`next.config.mjs`

```
// @ts-check

/*
```

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

export default nextConfig
```

Good to know: `next.config` with the `.cjs`, `.cts`, or `.mts` extensions are currently **not** supported.

Configuration as a Function

You can also use a function:

```
JS next.config.mjs
```

```
// @ts-check

export default (phase, { defaultConfig }) =>
  /**
   * @type {import('next').NextConfig}
   */
const nextConfig = {
  /* config options here */
}
return nextConfig
}
```

Async Configuration

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

```
JS next.config.js
```

```
// @ts-check

module.exports = async (phase, { defaultConfig }) => {
  /**
   * @type {import('next').NextConfig}
   */
const nextConfig = {
```

```
    /* config options here */
}
return nextConfig
}
```

Phase

`phase` is the current context in which the configuration is loaded. You can see the [available phases ↗](#). Phases can be imported from `next/constants`:

```
JS next.config.js Copy

// @ts-check

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

module.exports = (phase, { defaultConfig }) =>
  if (phase === PHASE_DEVELOPMENT_SERVER) {
    return {
      /* development only config options here */
    }
  }

  return {
    /* config options for all phases except d */
  }
}
```

TypeScript

If you are using TypeScript in your project, you can use `next.config.ts` to use TypeScript in your configuration:

```
TS next.config.ts Copy

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  /* config options here */
}
```

```
}
```

```
export default nextConfig
```

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

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

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

This page documents all the available configuration options:

Unit Testing (experimental)

Starting in Next.js 15.1, the `next/experimental/testing/server` package contains utilities to help unit test `next.config.js` files.

The `unstable_getResponseFromNextConfig` function runs the `headers`, `redirects`, and `rewrites` functions from `next.config.js` with the provided request information and returns `NextResponse` with the results of the routing.

The response from `unstable_getResponseFromNextConfig` only considers `next.config.js` fields and does not consider middleware or filesystem routes, so the

result in production may be different than the unit test.

```
import {
  getRedirectUrl,
  unstable_getResponseFromNextConfig,
} from 'next/experimental/testing/server'

const response = await unstable_getResponseFromNextConfig(
  url: 'https://nextjs.org/test',
  nextConfig: {
    async redirects() {
      return [{ source: '/test', destination: '/' }]
    }
  }
)
expect(response.status).toEqual(307)
expect(getRedirectUrl(response)).toEqual('https://nextjs.org/test')
```

allowedDevOrigins

Use
`allowedDevOrigins`
to configure...

appDir

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

assetPrefix

Learn how to use
the assetPrefix
config option to...

authInterrupts

Learn how to
enable the
experimental...

basePath

Use `basePath` to
deploy a Next.js
application under...

browserDeb...

Forward browser
console logs and
errors to your...

cacheCompo...

Learn how to enable the cacheComponent...

cacheLife

Learn how to set up cacheLife configurations in...

compress

Next.js provides gzip compression to compress...

crossOrigin

Use the `crossOrigin` option to add a...

cssChunking

Use the `cssChunking` option to control...

devIndicators

Configuration options for the on-screen indicator...

distDir

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

env

Learn to add and access environment...

eslint

Next.js reports ESLint errors and warnings during...

expireTime

Customize stale-while-revalidate expire time for IS...

exportPathM...

generateBuil...

Customize the pages that will be exported as HTM...

Configure the build id, which is used to identify...

generateEta...

Next.js will generate etags for every page by...

headers

Add custom HTTP headers to your Next.js app.

htmlLimited...

Specify a list of user agents that should receive...

httpAgentOp...

Next.js will automatically use HTTP Keep-Alive...

images

Custom configuration for the next/image...

cacheHandler

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

inlineCss

Enable inline CSS support.

logging

Configure how data fetches are logged to the...

mdxRs

Use the new Rust compiler to compile MDX file...

onDemandE...

Configure how Next.js will dispose and keep...

optimizePac...

API Reference for
optimizePackageIn
Next.js Config...

output

Next.js
automatically
traces which files...

pageExtensi...

Extend the default
page extensions
used by Next.js...

poweredByH...

Next.js will add
the `x-powered-by`
header by default...

ppr

Learn how to
enable Partial
Prerendering in...

productionBr...

Enables browser
source map
generation during...

reactCompiler

Enable the React
Compiler to
automatically...

reactMaxHe...

The maximum
length of the
headers that are...

reactStrictM...

The complete
Next.js runtime is
now Strict Mode-...

redirects

Add redirects to
your Next.js app.

rewrites

sassOptions

Add rewrites to
your Next.js app.

Configure Sass
options.

serverActions

Configure Server
Actions behavior in
your Next.js...

serverComp...

Configure whether
fetch responses in
Server...

serverExtern...

Opt-out specific
dependencies
from the Server...

staleTimes

Learn how to
override the
invalidation time...

staticGenera...

Learn how to
configure static
generation in you...

taint

Enable tainting
Objects and
Values.

trailingSlash

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

transpilePac...

Automatically
transpile and
bundle...

turbopack

Configure Next.js
with Turbopack-
specific options

turbopackPe...

Learn how to
enable Persistent
Caching for...

typedRoutes

Enable support for statically typed links.

typescript

Next.js reports TypeScript errors by default. Learn...

urlImports

Configure Next.js to allow importing modules from...

useCache

Learn how to enable the useCache flag in...

useLightning...

Enable experimental support for...

viewTransition

Enable ViewTransition API from React in Ap...

webpack

Learn how to customize the webpack config...

webVitalsAttribut...

Learn how to use the webVitalsAttributi...

Was this helpful?    

allowedDevOrigins

 Using App Router
Features available in /app Latest Version
15.5.4

allowedDevOrigins

Next.js does not automatically block cross-origin requests during development, but will block by default in a future major version of Next.js to prevent unauthorized requesting of internal assets/endpoints that are available in development mode.

To configure a Next.js application to allow requests from origins other than the hostname the server was initialized with (`localhost` by default) you can use the `allowedDevOrigins` config option.

`allowedDevOrigins` allows you to set additional origins that can be used in development mode. For example, to use `local-origin.dev` instead of only `localhost`, open `next.config.js` and add the `allowedDevOrigins` config:

```
JS next.config.js   
  
module.exports = {  
  allowedDevOrigins: ['local-origin.dev', '*']  
}
```

Was this helpful?    



Using App Router

Features available in /app



Latest Version

15.5.4



appDir

This is a legacy API and no longer recommended.
It's still supported for backward compatibility.

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

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

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

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

assetPrefix

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

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

Set up a CDN

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

Open `next.config.mjs` and add the `assetPrefix` config based on the [phase](#):

```
JS next.config.mjs 
```

```
// @ts-check
import { PHASE_DEVELOPMENT_SERVER } from 'next'

export default (phase) => {
  const isDev = phase === PHASE_DEVELOPMENT_SERVER
  /**
   * @type {import('next').NextConfig}
   */
  const nextConfig = {
    assetPrefix: isDev ? undefined : 'https://'
  }
  return nextConfig
}
```

}

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

```
/_next/static/chunks/4b9b41aaa062cbbfeff4add70f2
```

Would instead become:

```
https://cdn.mydomain.com/_next/static/chunks/4b9b41aaa062cbbfeff4add70f2
```

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

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

- Files in the `public` folder; if you want to serve those assets over a CDN, you'll have to introduce the prefix yourself

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

authInterrups

 This feature is currently available in the canary channel and subject to change. Try it out by upgrading Next.js, and share your feedback on GitHub.

The `authInterrups` configuration option allows you to use `forbidden` and `unauthorized` APIs in your application. While these functions are experimental, you must enable the `authInterrups` option in your `next.config.js` file to use them:

TS next.config.ts TypeScript ▾ 

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

const nextConfig: NextConfig = {
  experimental: {
    authInterrups: true,
  },
}

export default nextConfig
```

Next Steps

[forbidden](#)

[unauthorized](#)

API Reference for
the forbidden
function.

API Reference for
the unauthorized
function.

forbidden.js

API reference for
the forbidden.js
special file.

unauthorize...

API reference for
the
unauthorized.js...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

basePath

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

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

 `next.config.js`



```
module.exports = {
  basePath: '/docs',
}
```

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

Links

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

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

```
export default function HomePage() {
  return (
    <>
    <Link href="/about">About Page</Link>
  </>
}
```

```
)  
}
```

Output html:

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

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

Images

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

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

```
import Image from 'next/image'  
  
function Home() {  
  return (  
    <>  
      <h1>My Homepage</h1>  
      <Image  
        src="/docs/me.png"  
        alt="Picture of the author"  
        width={500}  
        height={500}  
      />  
      <p>Welcome to my homepage!</p>  
    </>  
  )  
}  
  
export default Home
```

Was this helpful?

 Using App Router
Features available in /app

 Latest Version
15.5.4

browserDebugInfoInTerminal

 This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

The `experimental.browserDebugInfoInTerminal` option forwards console output and runtime errors originating in the browser to the dev server terminal.

This option is disabled by default. When enabled it only works in development mode.

Usage

Enable forwarding:

```
TS next.config.ts TypeScript ▾ ⌂

import type { NextConfig } from 'next'

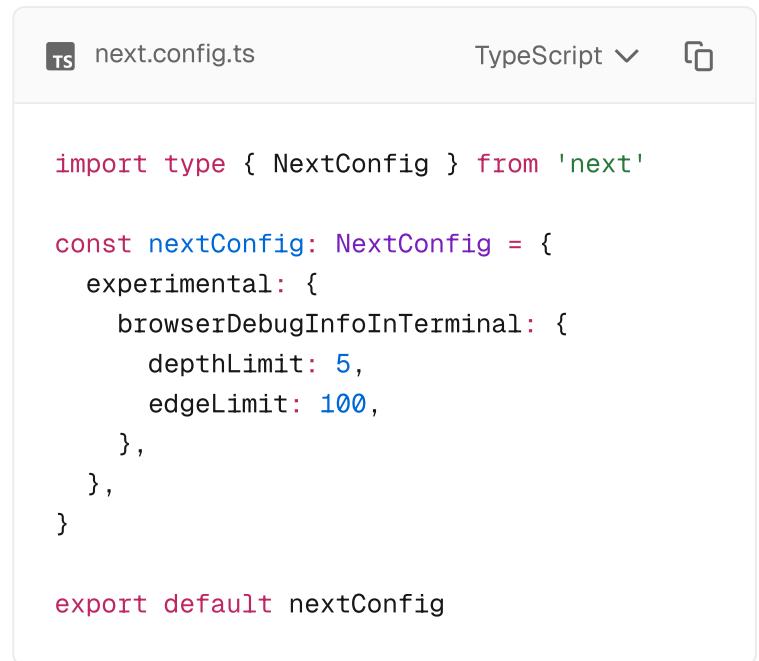
const nextConfig: NextConfig = {
  experimental: {
    browserDebugInfoInTerminal: true,
  },
}

export default nextConfig
```

Serialization limits

Deeply nested objects/arrays are truncated using **sensible defaults**. You can tweak these limits:

- **depthLimit**: (optional) Limit stringification depth for nested objects/arrays. Default: 5
- **edgeLimit**: (optional) Max number of properties or elements to include per object or array. Default: 100



The screenshot shows a code editor window with the file name "next.config.ts" at the top. The file contains the following TypeScript code:

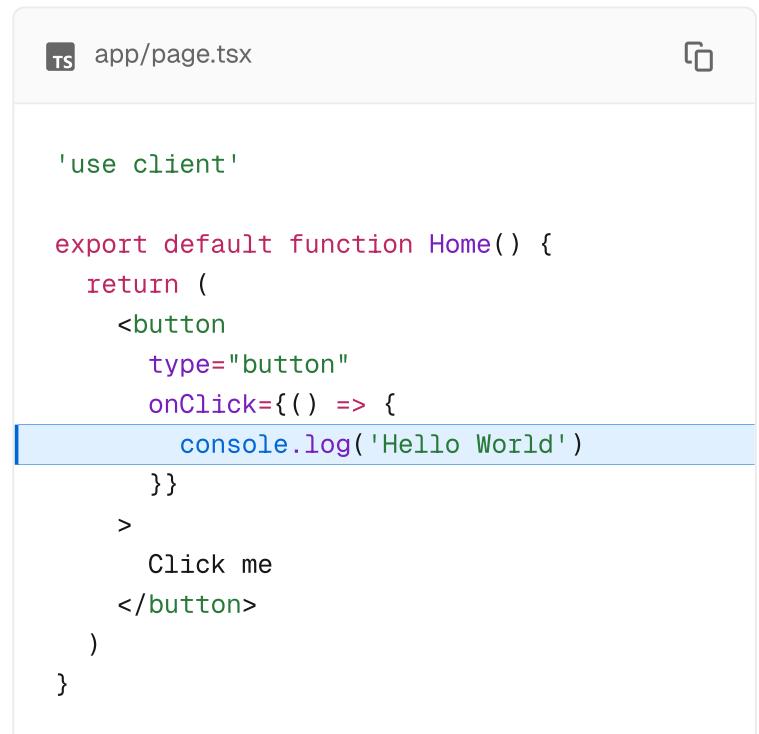
```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    browserDebugInfoInTerminal: {
      depthLimit: 5,
      edgeLimit: 100,
    },
  },
}

export default nextConfig
```

Source location

Source locations are included by default when this feature is enabled.

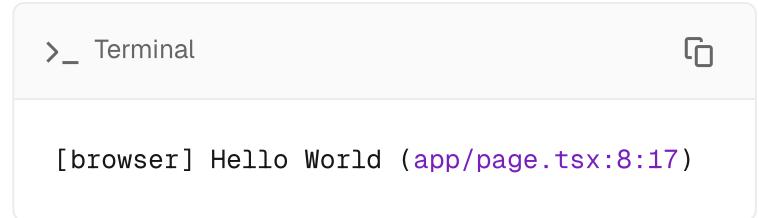


The screenshot shows a code editor window with the file name "app/page.tsx" at the top. The file contains the following TypeScript code:

```
'use client'

export default function Home() {
  return (
    <button
      type="button"
      onClick={() => {
        console.log('Hello World')
      }}
    >
      Click me
    </button>
  )
}
```

Clicking the button prints this message to the terminal.



A screenshot of a browser-based terminal window. The title bar says "Terminal". The content area shows the text "[browser] Hello World (app/page.tsx:8:17)" in purple, indicating a source location.

To suppress them, set

```
showSourceLocation: false.
```

- **showSourceLocation:** Include source location info when available.



A screenshot of a code editor showing the file "next.config.ts". The code defines a "nextConfig" object with an "experimental" key containing a "browserDebugInfoInTerminal" object. Within that object, the "showSourceLocation" key is set to "false".

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

const nextConfig: NextConfig = {
  experimental: {
    browserDebugInfoInTerminal: {
      showSourceLocation: false,
    },
  },
}

export default nextConfig
```

Version Changes

v15.4.0 experimental browserDebugInfoInTerminal introduced

Was this helpful?    



Using App Router

Features available in /app



Latest Version

15.5.4



cacheComponents

This feature is currently available in the canary channel and subject to change. Try it out by upgrading Next.js, and share your feedback on GitHub.

The `cacheComponents` flag is an experimental feature in Next.js that causes data fetching operations in the App Router to be excluded from pre-renders unless they are explicitly cached. This can be useful for optimizing the performance of dynamic data fetching in Server Components.

It is useful if your application requires fresh data fetching during runtime rather than serving from a pre-rendered cache.

It is expected to be used in conjunction with `use cache` so that your data fetching happens at runtime by default unless you define specific parts of your application to be cached with `use cache` at the page, function, or component level.

Usage

To enable the `cacheComponents` flag, set it to `true` in the `experimental` section of your `next.config.ts` file:

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

const nextConfig: NextConfig = {
  experimental: {
    cacheComponents: true,
  },
}

export default nextConfig
```

When `cacheComponents` is enabled, you can use the following cache functions and configurations:

- The `use cache` directive
- The `cacheLife` function with `use cache`
- The `cacheTag` function

Notes

- While `cacheComponents` can optimize performance by ensuring fresh data fetching during runtime, it may also introduce additional latency compared to serving pre-rendered content.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

cacheLife

 This feature is currently available in the canary channel and subject to change. Try it out by upgrading Next.js, and share your feedback on GitHub.

The `cacheLife` option allows you to define **custom cache profiles** when using the `cacheLife` function inside components or functions, and within the scope of the `use cache` directive.

Usage

To define a profile, enable the `cacheComponents flag` and add the cache profile in the `cacheLife` object in the `next.config.js` file. For example, a `blog` profile:

TS next.config.ts TypeScript ▾ 

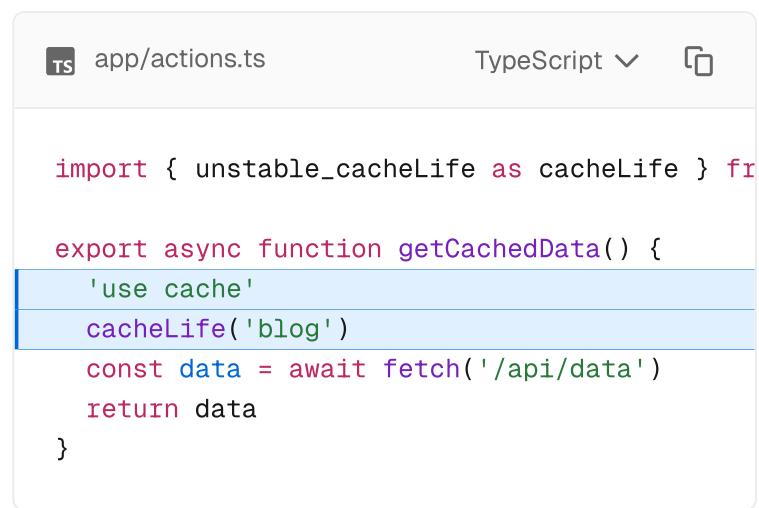
```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    cacheComponents: true,
    cacheLife: {
      blog: {
        stale: 3600, // 1 hour
        revalidate: 900, // 15 minutes
        expire: 86400, // 1 day
      },
    },
  },
}
```

```
}
```

```
export default nextConfig
```

You can now use this custom `blog` configuration in your component or function as follows:



```
TS app/actions.ts TypeScript ▾
```

```
import { unstable_cacheLife as cacheLife } from 'next/cache'

export async function getCachedData() {
  'use cache'
  cacheLife('blog')
  const data = await fetch('/api/data')
  return data
}
```

Reference

The configuration object has key values with the following format:

Property	Value	Description	Requirement
<code>stale</code>	<code>number</code>	Duration the client should cache a value without checking the server.	Optional
<code>revalidate</code>	<code>number</code>	Frequency at which the cache should refresh on the server; stale values may be served while revalidating.	Optional

Property	Value	Description	Requirement
<code>expire</code>	<code>number</code>	Maximum duration for which a value can remain stale before switching to dynamic.	Optional - Must be longer than <code>revalidate</code>

Was this helpful?     

Using App Router
Features available in /app

Latest Version
15.5.4

compress

By default, Next.js uses `gzip` to compress rendered content and static files when using `next start` or a custom server. This is an optimization for applications that do not have compression configured. If compression is *already* configured in your application via a custom server, Next.js will not add compression.

You can check if compression is enabled and which algorithm is used by looking at the `Accept-Encoding` ↗ (browser accepted options) and `Content-Encoding` ↗ (currently used) headers in the response.

Disabling compression

To disable **compression**, set the `compress` config option to `false`:

next.config.js



```
module.exports = {  
  compress: false,  
}
```

We **do not recommend disabling compression** unless you have compression configured on your server, as compression reduces bandwidth usage and improves the performance of your application.

For example, you're using [nginx](#) and want to switch to `brotli`, set the `compress` option to `false` to allow nginx to handle compression.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

crossOrigin

Use the `crossOrigin` option to add a `crossOrigin` attribute [↗](#) in all `<script>` tags generated by the `next/script` component, and define how cross-origin requests should be handled.

 next.config.js



```
module.exports = {
  crossOrigin: 'anonymous',
}
```

Options

- `'anonymous'` : Adds `crossOrigin="anonymous"` [↗](#) attribute.
- `'use-credentials'` : Adds `crossOrigin="use-credentials"` [↗](#).

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

cssChunking

 This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

CSS Chunking is a strategy used to improve the performance of your web application by splitting and re-ordering CSS files into chunks. This allows you to load only the CSS that is needed for a specific route, instead of loading all the application's CSS at once.

You can control how CSS files are chunked using the `experimental.cssChunking` option in your `next.config.js` file:

```
TS next.config.ts TypeScript ▾ ⌂

import type { NextConfig } from 'next'

const nextConfig = {
  experimental: {
    cssChunking: true, // default
  },
} satisfies NextConfig

export default nextConfig
```

Options

- `true (default)`: Next.js will try to merge CSS files whenever possible, determining explicit

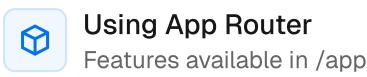
and implicit dependencies between files from import order to reduce the number of chunks and therefore the number of requests.

- `false` : Next.js will not attempt to merge or re-order your CSS files.
- `'strict'` : Next.js will load CSS files in the correct order they are imported into your files, which can lead to more chunks and requests.

You may consider using `'strict'` if you run into unexpected CSS behavior. For example, if you import `a.css` and `b.css` in different files using a different `import` order (`a` before `b`, or `b` before `a`), `true` will merge the files in any order and assume there are no dependencies between them. However, if `b.css` depends on `a.css`, you may want to use `'strict'` to prevent the files from being merged, and instead, load them in the order they are imported - which can result in more chunks and requests.

For most applications, we recommend `true` as it leads to fewer requests and better performance.

Was this helpful?    



devIndicators

`devIndicators` allows you to configure the on-screen indicator that gives context about the current route you're viewing during development.



The code editor shows a file named "Types" with the following content:

```
devIndicators: false | {
  position?: 'bottom-right'
  | 'bottom-left'
  | 'top-right'
  | 'top-left', // defaults to 'bottom-left'
},
```

Setting `devIndicators` to `false` will hide the indicator, however Next.js will continue to surface any build or runtime errors that were encountered.

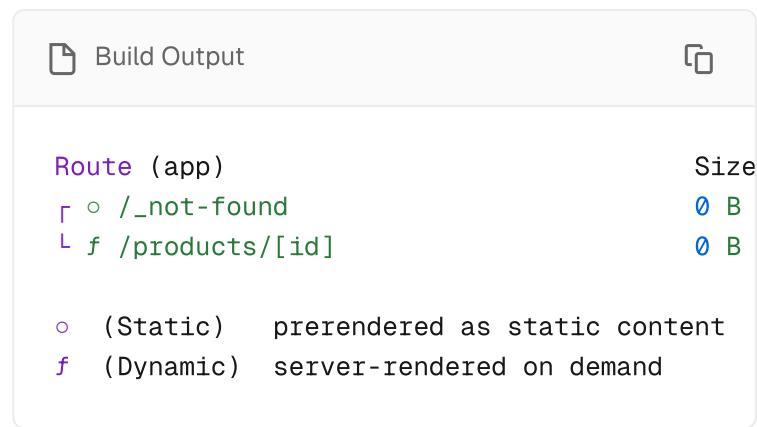
Troubleshooting

Indicator not marking a route as static

If you expect a route to be static and the indicator has marked it as dynamic, it's likely the route has opted out of static rendering.

You can confirm if a route is `static` or `dynamic` by building your application using `next build --debug`, and checking the output in your terminal. Static (or prerendered) routes will

display a `o` symbol, whereas dynamic routes will display a `f` symbol. For example:



The screenshot shows the Vite DevTools interface with the "Build Output" tab selected. It displays a list of routes under the "Route (app)" section. There are two entries: `/_not-found` (Static) and `/products/[id]` (Dynamic). Below the list, there is a legend explaining the symbols: `o` (Static) and `f` (Dynamic).

Route (app)	Size
<code>/_not-found</code>	0 B
<code>/products/[id]</code>	0 B

`o` (Static) prerendered as static content
`f` (Dynamic) server-rendered on demand

There are two reasons a route might opt out of static rendering:

- The presence of [Dynamic APIs](#) which rely on runtime information.
- An [uncached data request](#), like a call to an ORM or database driver.

Check your route for any of these conditions, and if you are not able to statically render the route, then consider using [loading.js](#) or [`<Suspense />`](#) to leverage [streaming](#).

Version History

Version	Changes
v15.2.0	Improved on-screen indicator with new <code>position</code> option. <code>appIsrStatus</code> , <code>buildActivity</code> , and <code>buildActivityPosition</code> options have been deprecated.
v15.0.0	Static on-screen indicator added with <code>appIsrStatus</code> option.

v15.2.0 Improved on-screen indicator with new `position` option. `appIsrStatus`, `buildActivity`, and `buildActivityPosition` options have been deprecated.

v15.0.0 Static on-screen indicator added with `appIsrStatus` option.

Was this helpful?  



distDir

 Using App Router
Features available in /app

 Latest Version
15.5.4

distDir

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

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

 next.config.js



```
module.exports = {  
  distDir: 'build',  
}
```

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

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

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

env

This is a legacy API and no longer recommended.
It's still supported for backward compatibility.

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

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

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

`next.config.js`



```
module.exports = {
  env: {
    customKey: 'my-value',
  },
}
```

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

```
function Page() {
  return <h1>The value of customKey is: {process.env.customKey}
}

export default Page
```

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

For example, the following line:

```
return <h1>The value of customKey is: {process
```

Will end up being:

```
return <h1>The value of customKey is: {'my-va
```

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

eslint

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

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

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

`next.config.js`

```
module.exports = {
  eslint: {
    // Warning: This allows production builds
    // your project has ESLint errors.
    ignoreDuringBuilds: true,
  },
}
```

Was this helpful?

Using App Router
Features available in /app

Latest Version
15.5.4

expireTime

You can specify a custom

`stale-while-revalidate` expire time for CDNs to consume in the `Cache-Control` header for ISR enabled pages.

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

next.config.js



```
module.exports = {
  // one hour in seconds
  expireTime: 3600,
}
```

Now when sending the `Cache-Control` header the expire time will be calculated depending on the specific revalidate period.

For example, if you have a revalidate of 15 minutes on a path and the expire time is one hour the generated `Cache-Control` header will be `s-maxage=900, stale-while-revalidate=2700` so that it can stay stale for 15 minutes less than the configured expire time.

Was this helpful?



exportPathMap

 Using App Router
Features available in /app

 Latest Version
15.5.4

exportPathMap

 This is a legacy API and no longer recommended.
It's still supported for backward compatibility.

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

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

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

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

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

 next.config.js



```
module.exports = {
  exportPathMap: async function (
    defaultPathMap,
    { dev, dir, outDir, distDir, buildId }
  ) {
    return {
      '/': { page: '/' },
      '/about': { page: '/about' },
      '/p/hello-nextjs': { page: '/post', que
```

```
'/p/learn-nextjs': { page: '/post', query: {} },
'/p/deploy-nextjs': { page: '/post', query: {} },
},
}
```

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

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

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

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

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

- `page : String` - the page inside the `pages` directory to render

- `query`: Object - the `query` object passed to `getInitialProps` when prerendering. Defaults to `{}`

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

Adding a trailing slash

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

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

```
js next.config.js
```

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

Customizing the output directory

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

```
>_ Terminal
```

```
next export -o outdir
```

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

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

generateBuildId

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`:

next.config.js



```
module.exports = {
  generateBuildId: async () => {
    // This could be anything, using the late
    return process.env.GIT_HASH
  },
}
```

Was this helpful?



[generateEtags](#)

 Using App Router
Features available in /app

 Latest Version
15.5.4

generateEtags

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

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

 `next.config.js`



```
module.exports = {  
  generateEtags: false,  
}
```

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

headers

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

To set custom HTTP headers you can use the

`headers` key in `next.config.js`:

 next.config.js



```
module.exports = {
  async headers() {
    return [
      {
        source: '/about',
        headers: [
          {
            key: 'x-custom-header',
            value: 'my custom header value',
          },
          {
            key: 'x-another-custom-header',
            value: 'my other custom header va
          },
        ],
      },
    ],
  },
}
```

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

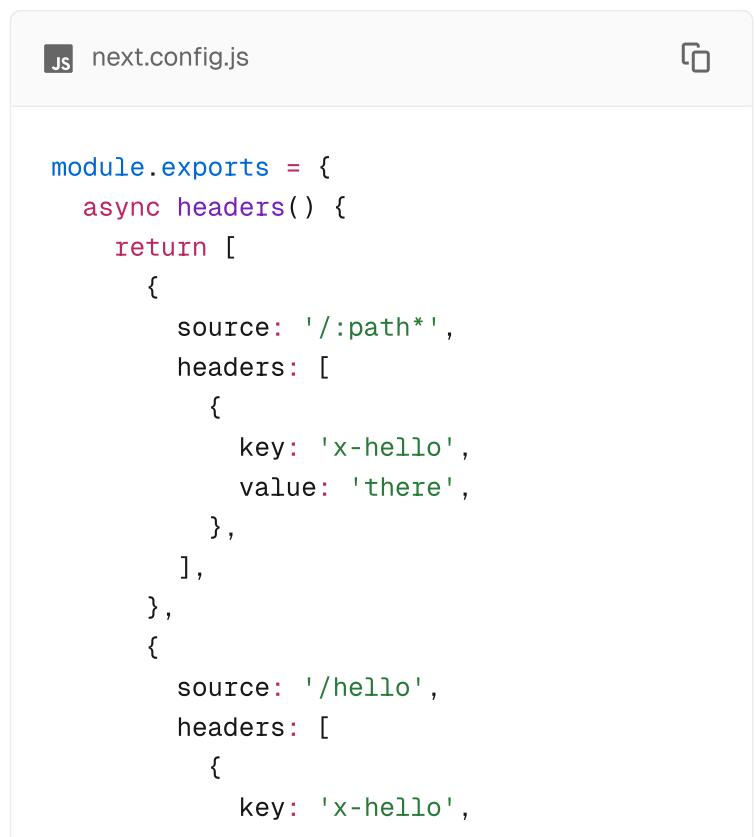
- `source` is the incoming request path pattern.
- `headers` is an array of response header objects, with `key` and `value` properties.

- `basePath`: `false` or `undefined` - if false the `basePath` won't be included when matching, can be used for external rewrites only.
- `locale`: `false` or `undefined` - whether the `locale` should not be included when matching.
- `has` is an array of `has objects` with the `type`, `key` and `value` properties.
- `missing` is an array of `missing objects` with the `type`, `key` and `value` properties.

Headers are checked before the filesystem which includes pages and `/public` files.

Header Overriding Behavior

If two headers match the same path and set the same header key, the last header key will override the first. Using the below headers, the path `/hello` will result in the header `x-hello` being `world` due to the last header value set being `world`.



```
JS next.config.js

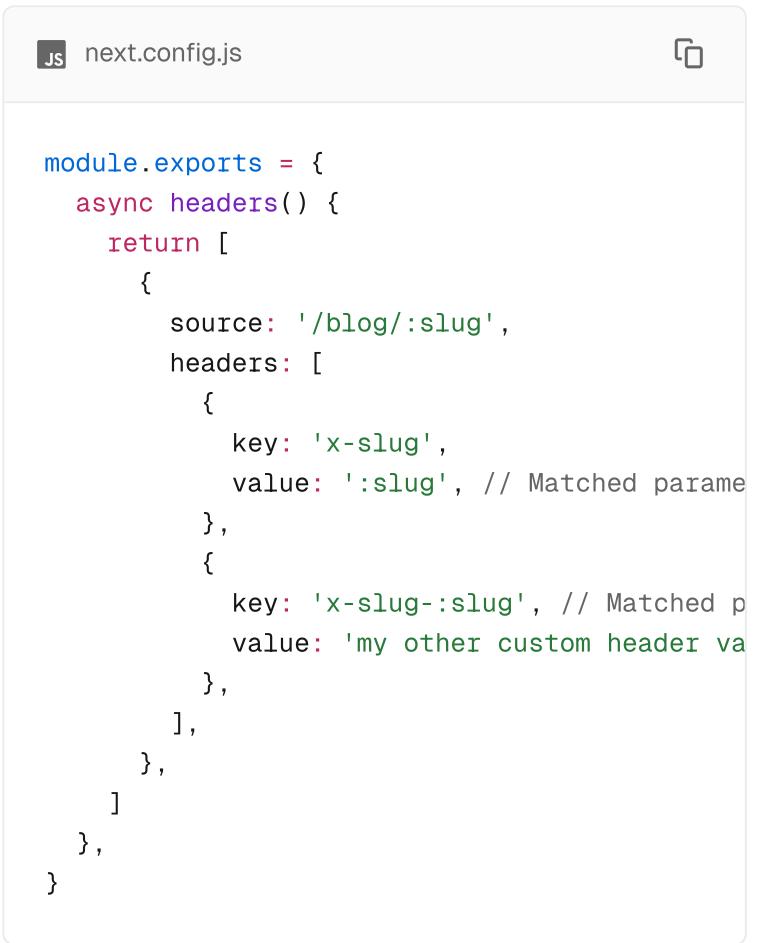
module.exports = {
  async headers() {
    return [
      {
        source: '/:path*',
        headers: [
          {
            key: 'x-hello',
            value: 'there',
          },
        ],
      },
      {
        source: '/hello',
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
    ];
  }
};
```

```
        value: 'world',
    },
],
},
],
}
}
```

Path Matching

Path matches are allowed, for example

`/blog/:slug` will match `/blog/hello-world` (no nested paths):

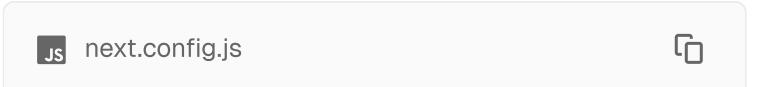


```
JS next.config.js

module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:slug',
        headers: [
          {
            key: 'x-slug',
            value: ':slug', // Matched parameter
          },
          {
            key: 'x-slug-:slug', // Matched parameter
            value: 'my other custom header value',
          },
        ],
      },
    ],
  },
}
```

Wildcard Path Matching

To match a wildcard path you can use `*` after a parameter, for example `/blog/:slug*` will match `/blog/a/b/c/d/hello-world`:



```
JS next.config.js
```

```
module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:slug*', // Matched parameter
        headers: [
          {
            key: 'x-slug',
            value: ':slug*', // Matched parameter
          },
          {
            key: 'x-slug-:slug*', // Matched parameter
            value: 'my other custom header value',
          },
        ],
      },
    ],
  },
}
```

Regex Path Matching

To match a regex path you can wrap the regex in parenthesis after a parameter, for example

/blog/:slug(\d{1,}) will match /blog/123

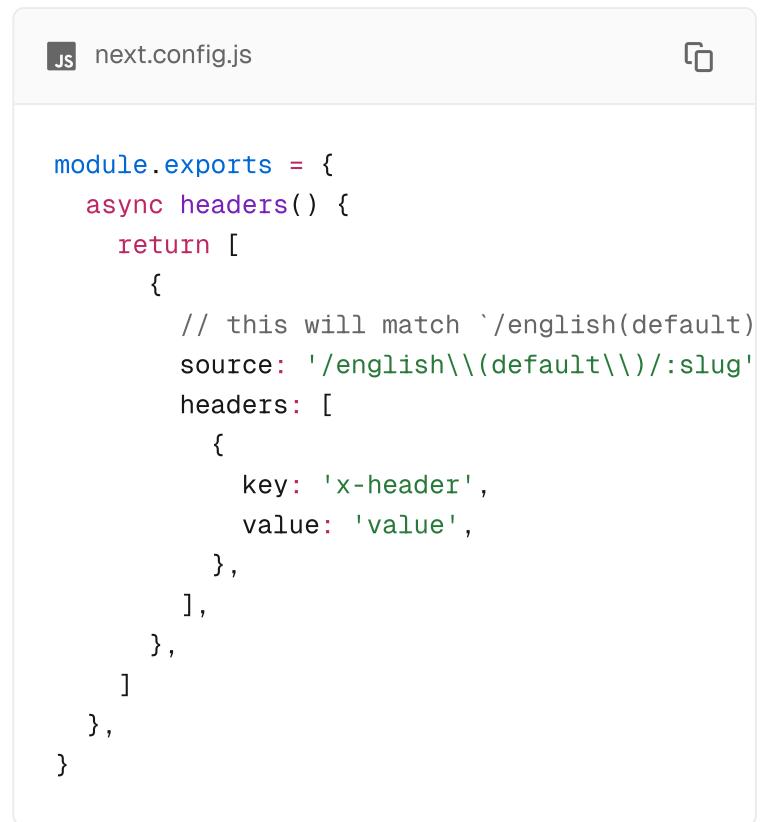
but not /blog/abc :

```
next.config.js
```

```
module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:post(\d{1,})',
        headers: [
          {
            key: 'x-post',
            value: ':post',
          },
        ],
      },
    ],
  },
}
```

The following characters (,), {, }, :, *, +, ? are used for regex path matching, so when

used in the `source` as non-special values they must be escaped by adding `\\"` before them:



```
JS next.config.js

module.exports = {
  async headers() {
    return [
      {
        // this will match `/english(default)
        source: '/english\\(default\\)\\/:slug'
        headers: [
          {
            key: 'x-header',
            value: 'value',
          },
        ],
      },
    ],
  },
}
```

Header, Cookie, and Query Matching

To only apply a header when header, cookie, or query values also match the `has` field or don't match the `missing` field can be used. Both the `source` and all `has` items must match and all `missing` items must not match for the header to be applied.

`has` and `missing` items can have the following fields:

- `type: String` - must be either `header`, `cookie`, `host`, or `query`.
- `key: String` - the key from the selected type to match against.

- `value`: `String` or `undefined` - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value `first-(?<paramName>.*)` is used for `first-second` then `second` will be usable in the destination with `:paramName`.



A screenshot of a code editor showing a file named `next.config.js`. The code defines an asynchronous function `headers()` that returns an array of objects. These objects represent headers to be added to responses based on specific conditions. The conditions include checking for the presence of `x-add-header` and the absence of `x-no-header`, and specifying a specific source path for certain headers.

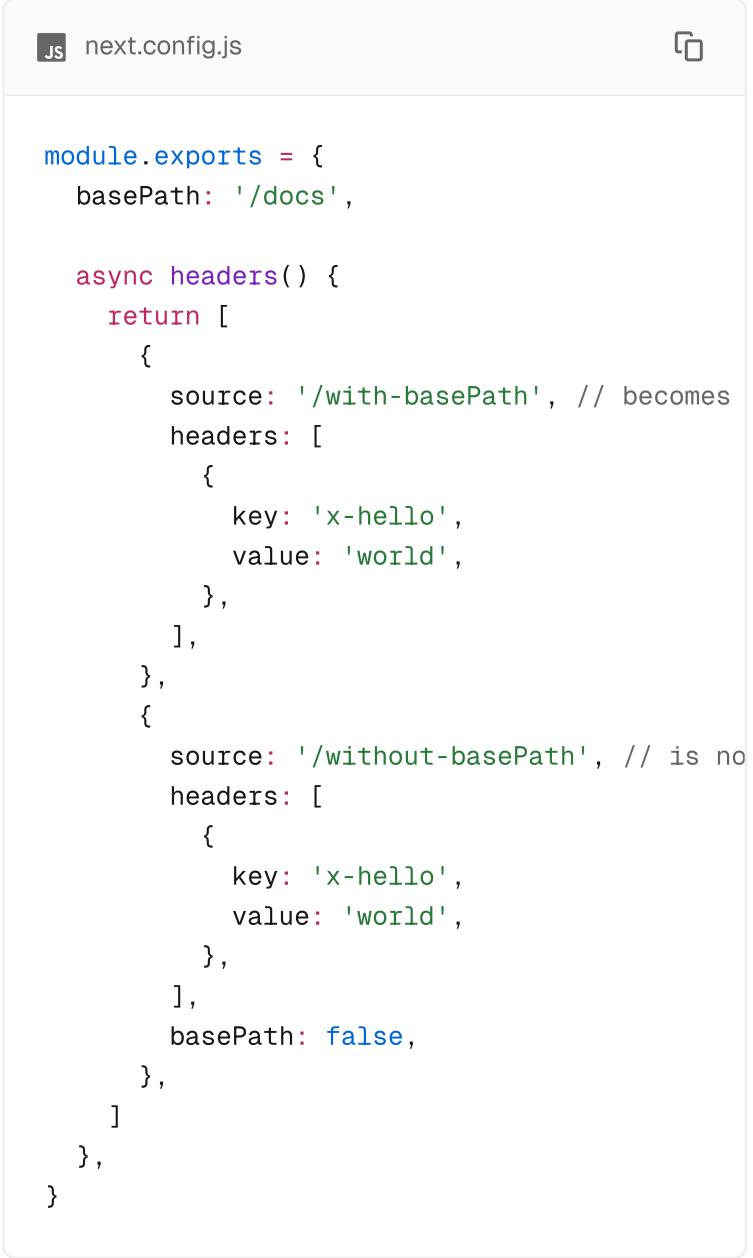
```
module.exports = {
  async headers() {
    return [
      // if the header `x-add-header` is present
      // the `x-another-header` header will be added
      {
        source: '/:path*',
        has: [
          {
            type: 'header',
            key: 'x-add-header',
          },
        ],
        headers: [
          {
            key: 'x-another-header',
            value: 'hello',
          },
        ],
      },
      // if the header `x-no-header` is not present
      // the `x-another-header` header will be added
      {
        source: '/:path*',
        missing: [
          {
            type: 'header',
            key: 'x-no-header',
          },
        ],
        headers: [
          {
            key: 'x-another-header',
            value: 'hello',
          },
        ],
      },
      // if the source, query, and cookie are present
      // the `x-authorized` header will be appended
      {
        source: '/specific/:path*',
        has: [
          {
            type: 'header',
            key: 'x-authorized',
          },
        ],
      },
    ];
  }
};
```

```
has: [
  {
    type: 'query',
    key: 'page',
    // the page value will not be ava
    // header key/values since value
    // doesn't use a named capture gr
    value: 'home',
  },
  {
    type: 'cookie',
    key: 'authorized',
    value: 'true',
  },
],
headers: [
  {
    key: 'x-authorized',
    value: ':authorized',
  },
],
// if the header `x-authorized` is pres
// contains a matching value, the `x-an
{
  source: '/:path*',
  has: [
    {
      type: 'header',
      key: 'x-authorized',
      value: '(?<authorized>yes|true)',
    },
  ],
  headers: [
    {
      key: 'x-another-header',
      value: ':authorized',
    },
  ],
},
// if the host is `example.com`,
// this header will be applied
{
  source: '/:path*',
  has: [
    {
      type: 'host',
      value: 'example.com',
    },
  ],
  headers: [
    {
      key: 'x-another-header',
      value: ':authorized',
    },
  ],
}
```

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

Headers with basePath support

When leveraging `basePath` support with headers each `source` is automatically prefixed with the `basePath` unless you add `basePath: false` to the header:



The screenshot shows a code editor window with a file named `next.config.js`. The code defines an object with a `basePath` key set to `'/docs'`. It also contains an `async headers()` function that returns an array of objects. Each object has a `source` key and a `headers` key. The first object's `source` is `'/with-basePath'` and its `headers` array contains one element with `key: 'x-hello'` and `value: 'world'`. The second object's `source` is `'/without-basePath'` and its `headers` array also contains one element with the same key-value pair. Both objects have a `basePath: false` property.

```
module.exports = {
  basePath: '/docs',

  async headers() {
    return [
      {
        source: '/with-basePath', // becomes
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        source: '/without-basePath', // is no
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
        basePath: false,
      },
    ]
  },
}
```

Headers with i18n support

When leveraging [i18n support](#) with headers each `source` is automatically prefixed to handle the configured `locales` unless you add `locale: false` to the header. If `locale: false` is used you must prefix the `source` with a locale for it to be matched correctly.

```
JS next.config.js

module.exports = {
  i18n: {
    locales: ['en', 'fr', 'de'],
    defaultLocale: 'en',
  },
  async headers() {
    return [
      {
        source: '/with-locale', // automatica
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // does not handle locales automatica
        source: '/nl/with-locale-manual',
        locale: false,
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // this matches '/' since 'en' is the
        source: '/en',
        locale: false,
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
    ],
  },
}
```

```
        ],
      },
      {
        // this gets converted to /(en|fr|de)
        // `/` or `/fr` routes like `/:path*` w
        source: '/(.*)',
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
    ],
  },
}
```

Cache-Control

Next.js sets the `Cache-Control` header of

`public, max-age=31536000, immutable` for 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 cannot set `Cache-Control` headers in `next.config.js` for these assets.

However, you can set `Cache-Control` headers for other responses or data.

Learn more about [caching](#) with the App Router.

Options

CORS

[Cross-Origin Resource Sharing \(CORS\)](#) ↗ is a security feature that allows you to control which

sites can access your resources. You can set the `Access-Control-Allow-Origin` header to allow a specific origin to access your Route Handlers.

```
async headers() {
  return [
    {
      source: "/api/:path*",
      headers: [
        {
          key: "Access-Control-Allow-Origin",
          value: "*", // Set your origin
        },
        {
          key: "Access-Control-Allow-Methods",
          value: "GET, POST, PUT, DELETE, OPTIONS",
        },
        {
          key: "Access-Control-Allow-Headers",
          value: "Content-Type, Authorization",
        },
      ],
    },
  ];
},
```

X-DNS-Prefetch-Control

This header [↗](#) controls DNS prefetching, allowing browsers to proactively perform domain name resolution on external links, images, CSS, JavaScript, and more. This prefetching is performed in the background, so the [DNS ↗](#) is more likely to be resolved by the time the referenced items are needed. This reduces latency when the user clicks a link.

```
{
  key: 'X-DNS-Prefetch-Control',
  value: 'on'
}
```

Strict-Transport-Security

This header ↗ informs browsers it should only be accessed using HTTPS, instead of using HTTP. Using the configuration below, all present and future subdomains will use HTTPS for a `max-age` of 2 years. This blocks access to pages or subdomains that can only be served over HTTP.

```
{  
  key: 'Strict-Transport-Security',  
  value: 'max-age=63072000; includeSubDomains'  
}
```

X-Frame-Options

This header ↗ indicates whether the site should be allowed to be displayed within an `iframe`. This can prevent against clickjacking attacks.

This header has been superseded by CSP's `frame-ancestors` option, which has better support in modern browsers (see [Content Security Policy](#) for configuration details).

```
{  
  key: 'X-Frame-Options',  
  value: 'SAMEORIGIN'  
}
```

Permissions-Policy

This header ↗ allows you to control which features and APIs can be used in the browser. It was previously named `Feature-Policy`.

```
{  
  key: 'Permissions-Policy',  
  value: 'camera=(), microphone=(), geolocati  
}'
```

X-Content-Type-Options

This header [↗](#) prevents the browser from attempting to guess the type of content if the `Content-Type` header is not explicitly set. This can prevent XSS exploits for websites that allow users to upload and share files.

For example, a user trying to download an image, but having it treated as a different `Content-Type` like an executable, which could be malicious. This header also applies to downloading browser extensions. The only valid value for this header is `nosniff`.

```
{  
  key: 'X-Content-Type-Options',  
  value: 'nosniff'  
}
```

Referrer-Policy

This header [↗](#) controls how much information the browser includes when navigating from the current website (origin) to another.

```
{  
  key: 'Referrer-Policy',  
  value: 'origin-when-cross-origin'  
}
```

Content-Security-Policy

Learn more about adding a [Content Security Policy](#) to your application.

Version History

Version**Changes**

v13.3.0

missing added.

v10.2.0

has added.

v9.5.0

Headers added.

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

htmlLimitedBots

The `htmlLimitedBots` config allows you to specify a list of user agents that should receive blocking metadata instead of [streaming metadata](#).

next.config.ts

TypeScript



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

const config: NextConfig = {
  htmlLimitedBots: '/MySpecialBot|MyAnotherSpe'
}

export default config
```

Default list

Next.js includes [a default list of HTML limited bots](#) ↗.

Specifying a `htmlLimitedBots` config will override the Next.js' default list, allowing you full control over what user agents should opt into this behavior. However, this is advanced behavior, and the default should be sufficient for most cases.

Version History

Version	Changes
---------	---------

15.2.0	<code>htmlLimitedBots</code> option introduced.
--------	---

Was this helpful?    

[httpAgentOptions](#)

 Using App Router
Features available in /app

 Latest Version
15.5.4

httpAgentOptions

In Node.js versions prior to 18, Next.js automatically polyfills `fetch()` with `undici` and enables [HTTP Keep-Alive ↗](#) by default.

To disable HTTP Keep-Alive for all `fetch()` calls on the server-side, open `next.config.js` and add the `httpAgentOptions` config:

 `next.config.js`



```
module.exports = {  
  httpAgentOptions: {  
    keepAlive: false,  
  },  
}
```

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

images

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure

`next.config.js` with the following:

 next.config.js

```
module.exports = {
  images: {
    loader: 'custom',
    loaderFile: './my/image/loader.js',
  },
}
```

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

 my/image/loader.js

```
'use client'

export default function myImageLoader({ src,
  return `https://example.com/${src}?w=${width}`
})
```

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

Good to know: Customizing the image loader file, which accepts a function, requires using `Client`

To learn more about configuring the behavior of the built-in [Image Optimization API](#) and the [Image Component](#), see [Image Configuration Options](#) for available options.

Example Loader Configuration

- [Akamai](#)
- [AWS CloudFront](#)
- [Cloudinary](#)
- [Cloudflare](#)
- [Contentful](#)
- [Fastly](#)
- [Gumlet](#)
- [ImageEngine](#)
- [Imgix](#)
- [PixelBin](#)
- [Sanity](#)
- [Sirv](#)
- [Supabase](#)
- [Thumbor](#)
- [Imagekit](#)
- [Nitrogen AIO](#)

Akamai

```
// Docs: https://techdocs.akamai.com/ivm/referential-image-optimization-api#image-loader-configuration
export default function akamaiLoader({ src, width }) {
  return `https://example.com/${src}?imwidth=${width}`;
}
```

AWS CloudFront

```
// Docs: https://aws.amazon.com/developer/app
export default function cloudfrontLoader({ sr
  const url = new URL(`https://example.com${s
  urlSearchParams.set('format', 'auto')
  urlSearchParams.set('width', width.toString())
  urlSearchParams.set('quality', (quality || 75).t
  return url.href
}
```

Cloudinary

```
// Demo: https://res.cloudinary.com/demo/imag
export default function cloudinaryLoader({ sr
  const params = ['f_auto', 'c_limit', `w_${w
  return `https://example.com/${params.join(
} }
```

Cloudflare

```
// Docs: https://developers.cloudflare.com/im
export default function cloudflareLoader({ sr
  const params = [`width=${width}`, `quality=
  return `https://example.com/cdn-cgi/image/$
}
```

Contentful

```
// Docs: https://www.contentful.com/developer
export default function contentfulLoader({ sr
  const url = new URL(`https://example.com${s
  urlSearchParams.set('fm', 'webp')
  urlSearchParams.set('w', width.toString())
  urlSearchParams.set('q', (quality || 75).t
  return url.href
}
```

Fastly

```
// Docs: https://developer.fastly.com/referen
```

```
export default function fastlyLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`);
  url.searchParams.set('auto', 'webp');
  url.searchParams.set('width', width.toString());
  url.searchParams.set('quality', (quality || 75).toString());
  return url.href;
}
```

Gumlet

```
// Docs: https://docs.gumlet.com/reference/image-optimization
export default function gumletLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`);
  url.searchParams.set('format', 'auto');
  url.searchParams.set('w', width.toString());
  url.searchParams.set('q', (quality || 75).toString());
  return url.href;
}
```

ImageEngine

```
// Docs: https://support.imageengine.io/hc/en-us/articles/360000000000
export default function imageengineLoader({ src, width, quality }) {
  const compression = 100 - (quality || 50);
  const params = [ `w_${width}` , `cmpr_${compression}` ];
  return `https://example.com${src}?imgeng=/${params.join('&')}`;
}
```

Imgix

```
// Demo: https://static.imgix.net/daisy.png?fit=auto&w=500&q=50
export default function imgixLoader({ src, width, height, quality }) {
  const url = new URL(`https://example.com${src}`);
  const params = url.searchParams;
  params.set('auto', params.getAll('auto').join('+'));
  params.set('fit', params.get('fit') || 'max');
  params.set('w', params.get('w') || width.toString());
  params.set('q', (quality || 50).toString());
  return url.href;
}
```

PixelBin

```
// Doc (Resize): https://www.pixelbin.io/docs
// Doc (Optimise): https://www.pixelbin.io/do
// Doc (Auto Format Delivery): https://www.pi
export default function pixelBinLoader({ src,
  const name = '<your-cloud-name>'
  const opt = `t.resize(w:${width})~t.compres
return `https://cdn.pixelbin.io/v2/${name}/
}
```

Sanity

```
// Docs: https://www.sanity.io/docs/image-url
export default function sanityLoader({ src, w
  const prj = 'zp7mbokg'
  const dataset = 'production'
  const url = new URL(`https://cdn.sanity.io/
urlSearchParams.set('auto', 'format')
urlSearchParams.set('fit', 'max')
urlSearchParams.set('w', width.toString())
if (quality) {
  urlSearchParams.set('q', quality.toStrin
}
return url.href
}
```

Sirv

```
// Docs: https://sirv.com/help/articles/dynam
export default function sirvLoader({ src, wid
  const url = new URL(`https://example.com${s
  const params = url.searchParams
  params.set('format', params.getAll('format')
  params.set('w', params.get('w') || width.to
  params.set('q', (quality || 85).toString())
  return url.href
}
```

Supabase

```
// Docs: https://supabase.com/docs/guides/sto
export default function supabaseLoader({ src,
  const url = new URL(`https://example.com${s
  urlSearchParams.set('width', width.toStrin
  urlSearchParams.set('quality', (quality ||
```

```
    return url.href
}
```

Thumbor

```
// Docs: https://thumbor.readthedocs.io/en/la
export default function thumborLoader({ src,
  const params = [`w-${width}x0`, `filters:qual
  return `https://example.com${params.join('/
}`
```

ImageKit.io

```
// Docs: https://imagekit.io/docs/image-trans
export default function imageKitLoader({ src,
  const params = [`w-${width}`, `q-${quality}
  return `https://ik.imagekit.io/your_imageki
`
```

Nitrogen AIO

```
// Docs: https://docs.n7.io/aio/intergrations
export default function aioLoader({ src, width
  const url = new URL(src, window.location.href)
  const params = url.searchParams
  const aioParams = params.getAll('aio')
  aioParams.push(`w-${width}`)
  if (quality) {
    aioParams.push(`q-${quality.toString()}`)
  }
  params.set('aio', aioParams.join(';'))
  return url.href
}
```

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

Custom Next.js Cache Handler

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.

next.config.js



```
module.exports = {
  cacheHandler: require.resolve('./cache-hand
  cacheMaxMemorySize: 0, // disable default i
}
```

View an example of a [custom cache handler](#) and learn more about the implementation.

API Reference

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

`get()`

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

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

`set()`

Parameter	Type	Description
<code>key</code>	<code>string</code>	The key to store the data under.
<code>data</code>	Data or <code>null</code>	The data to be cached.
<code>ctx</code>	<code>{ tags: [] }</code>	The cache tags provided.

Returns `Promise<void>`.

`revalidateTag()`

Parameter	Type	Description
<code>tag</code>	<code>string</code> or <code>string[]</code>	The cache tags to revalidate.

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

`resetRequestCache()`

This method resets the temporary in-memory cache for a single request before the next request.

Returns `void`.

Good to know:

- `revalidatePath` is a convenience layer on top of cache tags. Calling `revalidatePath` will call your `revalidateTag` function, which you can

then choose if you want to tag cache keys based on the path.

Platform Support

Deployment Option	Supported
Node.js server	Yes
Docker container	Yes
Static export	No
Adapters	Platform-specific

Learn how to [configure ISR](#) when self-hosting Next.js.

Version History

Version	Changes
v14.1.0	Renamed to <code>cacheHandler</code> and became stable.
v13.4.0	<code>incrementalCacheHandlerPath</code> support for <code>revalidateTag</code> .
v13.4.0	<code>incrementalCacheHandlerPath</code> support for standalone output.
v12.2.0	Experimental <code>incrementalCacheHandlerPath</code> added.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

inlineCss

 This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

Usage

Experimental support for inlining CSS in the `<head>`. When this flag is enabled, all places where we normally generate a `<link>` tag will instead have a generated `<style>` tag.

TS

next.config.ts

TypeScript



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

const nextConfig: NextConfig = {
  experimental: {
    inlineCss: true,
  },
}

export default nextConfig
```

Trade-Offs

When to Use Inline CSS

Inlining CSS can be beneficial in several scenarios:

- **First-Time Visitors:** Since CSS files are render-blocking resources, inlining eliminates the initial download delay that first-time visitors experience, improving page load performance.
- **Performance Metrics:** By removing the additional network requests for CSS files, inlining can significantly improve key metrics like First Contentful Paint (FCP) and Largest Contentful Paint (LCP).
- **Slow Connections:** For users on slower networks where each request adds considerable latency, inlining CSS can provide a noticeable performance boost by reducing network roundtrips.
- **Atomic CSS Bundles (e.g., Tailwind):** With utility-first frameworks like Tailwind CSS, the size of the styles required for a page is often O(1) relative to the complexity of the design. This makes inlining a compelling choice because the entire set of styles for the current page is lightweight and doesn't grow with the page size. Inlining Tailwind styles ensures minimal payload and eliminates the need for additional network requests, which can further enhance performance.

When Not to Use Inline CSS

While inlining CSS offers significant benefits for performance, there are scenarios where it may not be the best choice:

- **Large CSS Bundles:** If your CSS bundle is too large, inlining it may significantly increase the size of the HTML, resulting in slower Time to First Byte (TTFB) and potentially worse performance for users with slow connections.

- **Dynamic or Page-Specific CSS:** For applications with highly dynamic styles or pages that use different sets of CSS, inlining may lead to redundancy and bloat, as the full CSS for all pages may need to be inlined repeatedly.
- **Browser Caching:** In cases where visitors frequently return to your site, external CSS files allow browsers to cache styles efficiently, reducing data transfer for subsequent visits. Inlining CSS eliminates this benefit.

Evaluate these trade-offs carefully, and consider combining inlining with other strategies, such as critical CSS extraction or a hybrid approach, for the best results tailored to your site's needs.

Good to know:

This feature is currently experimental and has some known limitations:

- CSS inlining is applied globally and cannot be configured on a per-page basis
- Styles are duplicated during initial page load - once within `<style>` tags for SSR and once in the RSC payload
- When navigating to statically rendered pages, styles will use `<link>` tags instead of inline CSS to avoid duplication
- This feature is not available in development mode and only works in production builds

Was this helpful?



logging

 Using App Router
Features available in /app

 Latest Version
15.5.4

logging

Options

Fetching

You can configure the logging level and whether the full URL is logged to the console when running Next.js in development mode.

Currently, `logging` only applies to data fetching using the `fetch` API. It does not yet apply to other logs inside of Next.js.

 next.config.js 

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

Any `fetch` requests that are restored from the [Server Components HMR cache](#) are not logged by default. However, this can be enabled by setting `logging.fetches.hmrRefreshes` to `true`.

 next.config.js 

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

```
 },  
 }
```

Incoming Requests

By default all the incoming requests will be logged in the console during development. You can use the `incomingRequests` option to decide which requests to ignore. Since this is only logged in development, this option doesn't affect production builds.



```
JS next.config.js Copy  
  
module.exports = {  
  logging: {  
    incomingRequests: {  
      ignore: [/^api\/v1\/health/],  
    },  
  },  
}
```

Or you can disable incoming request logging by setting `incomingRequests` to `false`.



```
JS next.config.js Copy  
  
module.exports = {  
  logging: {  
    incomingRequests: false,  
  },  
}
```

Disabling Logging

In addition, you can disable the development logging by setting `logging` to `false`.



```
JS next.config.js Copy  
  
module.exports = {  
  logging: false,
```

}

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

mdxRs

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

 next.config.js 

```
const withMDX = require('@next/mdx')()

/** @type {import('next').NextConfig} */
const nextConfig = {
  pageExtensions: ['ts', 'tsx', 'mdx'],
  experimental: {
    mdxRs: true,
  },
}

module.exports = withMDX(nextConfig)
```

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

onDemandEntries

Next.js exposes some options that give you some control over how the server will dispose or keep in memory built pages in development.

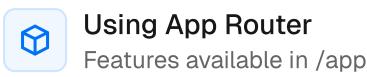
To change the defaults, open `next.config.js` and add the `onDemandEntries` config:

next.config.js



```
module.exports = {
  onDemandEntries: {
    // period (in ms) where the server will k
    maxInactiveAge: 25 * 1000,
    // number of pages that should be kept si
    pagesBufferLength: 2,
  },
}
```

Was this helpful?



optimizePackageImports

Some packages can export hundreds or thousands of modules, which can cause performance issues in development and production.

Adding a package to

`experimental.optimizePackageImports` will only load the modules you are actually using, while still giving you the convenience of writing import statements with many named exports.

```
JS next.config.js 
```

```
module.exports = {
  experimental: {
    optimizePackageImports: ['package-name'],
  },
}
```

The following libraries are optimized by default:

- `lucide-react`
- `date-fns`
- `lodash-es`
- `ramda`
- `antd`
- `react-bootstrap`
- `ahooks`
- `@ant-design/icons`

- `@headlessui/react`
- `@headlessui-float/react`
- `@heroicons/react/20/solid`
- `@heroicons/react/24/solid`
- `@heroicons/react/24/outline`
- `@visx/visx`
- `@tremor/react`
- `rxjs`
- `@mui/material`
- `@mui/icons-material`
- `recharts`
- `react-use`
- `@material-ui/core`
- `@material-ui/icons`
- `@tabler/icons-react`
- `mui-core`
- `react-icons/*`
- `effect`
- `@effect/*`

Was this helpful?    

[Copy page](#)

Using App Router

Features available in /app

Latest Version

15.5.4

During a build, Next.js will automatically trace each page and its dependencies to determine all of the files that are needed for deploying a production version of your application.

This feature helps reduce the size of deployments drastically. Previously, when deploying with Docker you would need to have all files from your package's `dependencies` installed to run `next start`. Starting with Next.js 12, you can leverage Output File Tracing in the `.next/` directory to only include the necessary files.

Furthermore, this removes the need for the deprecated `serverless` target which can cause various issues and also creates unnecessary duplication.

How it Works

During `next build`, Next.js will use `@vercel/nft` to statically analyze `import`, `require`, and `fs` usage to determine all files that a page might load.

Next.js' production server is also traced for its needed files and output at `.next/next-server.js.nft.json` which can be leveraged in production.

To leverage the `.nft.json` files emitted to the `.next` output directory, you can read the list of files in each trace that are relative to the `.nft.json` file and then copy them to your deployment location.

Automatically Copying Traced Files

Next.js can automatically create a `standalone` folder that copies only the necessary files for a production deployment including select files in `node_modules`.

To leverage this automatic copying you can enable it in your `next.config.js`:

```
JS next.config.js Copy  
  
module.exports = {  
  output: 'standalone',  
}  
  
This will create a folder at .next/standalone which can then be deployed on its own without installing node_modules.
```

Additionally, a minimal `server.js` file is also output which can be used instead of `next start`. This minimal server does not copy the `public` or `.next/static` folders by default as these should ideally be handled by a CDN instead, although these folders can be copied to the `standalone/public` and `standalone/.next/static` folders manually, after which `server.js` file will serve these automatically.

To copy these manually, you can use the `cp` command-line tool after you `next build`:

>_ Terminal



```
cp -r public .next/standalone/ && cp -r .next
```

To start your minimal `server.js` file locally, run the following command:

>_ Terminal



```
node .next/standalone/server.js
```

Good to know:

- If your project needs to listen to a specific port or hostname, you can define `PORT` or `HOSTNAME` environment variables before running `server.js`. For example, run `PORT=8080 HOSTNAME=0.0.0.0 node server.js` to start the server on `http://0.0.0.0:8080`.

Caveats

- While tracing in monorepo setups, the project directory is used for tracing by default. For `next build packages/web-app`, `packages/web-app` would be the tracing root and any files outside of that folder will not be included. To include files outside of this folder you can set `outputFileTracingRoot` in your `next.config.js`.

JS packages/web-app/next.config.js



```
const path = require('path')
```

```
module.exports = {
  // this includes files from the monorepo ba
  outputFileTracingRoot: path.join(__dirname,
}
```

- There are some cases in which Next.js might fail to include required files, or might incorrectly include unused files. In those cases, you can leverage `outputFileTracingExcludes` and `outputFileTracingIncludes` respectively in `next.config.js`. Each option accepts an object whose keys are **route globs** (matched with [picomatch ↗](#) against the route path, e.g. `/api/hello`) and whose values are **glob patterns resolved from the project root** that specify files to include or exclude in the trace.

JS next.config.js

```
module.exports = {
  outputFileTracingExcludes: {
    '/api/hello': ['./un-necessary-folder/**/*'],
  },
  outputFileTracingIncludes: {
    '/api/another': ['./necessary-folder/**/*'],
    '/api/login/\\"[\\"[\\".\\".\\".\\".slug\"\"]]\\"': [
      './node_modules/aws-crt/dist/bin/**/*',
    ],
  },
}
```

Using a `src/` directory does not change how you write these options:

- **Keys** still match the route path (`'/api/hello'`, `'/products/[id]'`, etc.).
- **Values** can reference paths under `src/` since they are resolved relative to the project root.

JS next.config.js

```
module.exports = {
  outputFileTracingIncludes: {
```

```
'/products/*': ['src/lib/payments/**/*'],
'/*': ['src/config/runtime/**/*.json'],
},
outputFileTracingExcludes: {
  '/api/*': ['src/temp/**/*', 'public/large'],
},
}
```

You can also target all routes using a global key like `'/*'`:

next.config.js



```
module.exports = {
  outputFileTracingIncludes: {
    '/*': ['src/i18n/locales/**/*.json'],
  },
}
```

These options are applied to server traces and do not affect routes that do not produce a server trace file:

- Edge Runtime routes are not affected.
- Fully static pages are not affected.

In monorepos or when you need to include files outside the app folder, combine

`outputFileTracingRoot` with includes:

next.config.js



```
const path = require('path')

module.exports = {
  // Trace from the monorepo root
  outputFileTracingRoot: path.join(__dirname,
  outputFileTracingIncludes: {
    '/route1': ['./shared/assets/**/*'],
  },
}
```

Good to know:

- Prefer forward slashes (/) in patterns for cross-platform compatibility.
- Keep patterns as narrow as possible to avoid oversized traces (avoid **/* at the repo root).

Common include patterns for native/runtime assets:

next.config.js



```
module.exports = {
  outputFileTracingIncludes: {
    '/**': ['node_modules/sharp/**/*', 'node_m
  },
}
```

Was this helpful?





Using App Router

Features available in /app



pageExtensions



Latest Version

15.5.4

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

`.md`, `.mdx`).

next.config.js



```
const withMDX = require('@next/mdx')()

/** @type {import('next').NextConfig} */
const nextConfig = {
  pageExtensions: ['js', 'jsx', 'ts', 'tsx'],
}

module.exports = withMDX(nextConfig)
```

Was this helpful?



**Using App Router**

Features available in /app

**Latest Version**

15.5.4



poweredByHeader

By default Next.js will add the `x-powered-by` header. To opt-out of it, open `next.config.js` and disable the `poweredByHeader` config:

`JS next.config.js`

```
module.exports = {
  poweredByHeader: false,
}
```

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

ppr

 This feature is currently available in the canary channel and subject to change. Try it out by upgrading Next.js, and share your feedback on GitHub.

Partial Prerendering (PPR) enables you to combine static and dynamic components together in the same route. Learn more about [PPR](#).

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:

```
TS next.config.ts TypeScript ▾   
  
import type { NextConfig } from 'next'  
  
const nextConfig: NextConfig = {  
  experimental: {  
    ppr: 'incremental',  
  },  
}  
  
export default nextConfig
```

TypeScript ▾

```
import { Suspense } from "react"
import { StaticComponent, DynamicComponent, F

    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.

Version Changes

v15.0.0 experimental `incremental` value introduced

v14.0.0 experimental `ppr` introduced

Learn more about Partial Prerendering

Partial Prere...

Learn how to use
Partial
Prerendering and...

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

productionBrowser- SourceMaps

Source Maps are enabled by default during development. During production builds, they are disabled to prevent you leaking your source on the client, unless you specifically opt-in with the configuration flag.

Next.js provides a configuration flag you can use to enable browser source map generation during the production build:

```
JS next.config.js   
  
module.exports = {  
  productionBrowserSourceMaps: true,  
}
```

When the `productionBrowserSourceMaps` option is enabled, the source maps will be output in the same directory as the JavaScript files. Next.js will automatically serve these files when requested.

- Adding source maps can increase `next build` time
- Increases memory usage during `next build`

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

reactCompiler

 This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

Next.js includes support for the [React Compiler ↗](#), a tool designed to improve performance by automatically optimizing component rendering. This reduces the need for manual memoization using `useMemo` and `useCallback`.

Next.js includes a custom performance optimization written in SWC that makes the React Compiler more efficient. Instead of running the compiler on every file, Next.js analyzes your project and only applies the React Compiler to relevant files. This avoids unnecessary work and leads to faster builds compared to using the Babel plugin on its own.

How It Works

The React Compiler runs through a Babel plugin. To keep builds fast, Next.js uses a custom SWC optimization that only applies the React Compiler to relevant files—like those with JSX or React Hooks.

This avoids compiling everything and keeps the performance cost minimal. You may still see slightly slower builds compared to the default

Rust-based compiler, but the impact is small and localized.

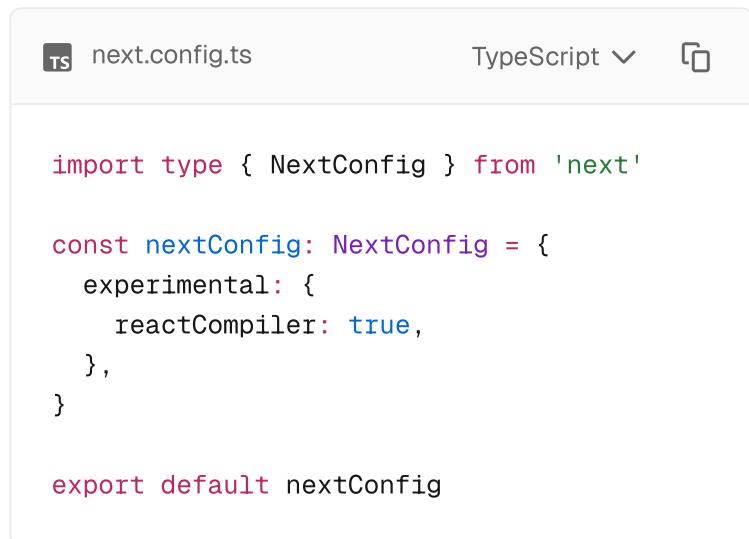
To use it, install the

```
babel-plugin-react-compiler:
```



A screenshot of a terminal window titled "Terminal". It contains the command `npm install babel-plugin-react-compiler` in green text, indicating successful execution.

Then, add `experimental.reactCompiler` option in `next.config.js`:



A screenshot of a code editor window titled "next.config.ts". The code is written in TypeScript and defines a `nextConfig` object with an `experimental` key containing a `reactCompiler` key set to `true`. The code is also annotated with `import type` from 'next' and `export default`.

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

const nextConfig: NextConfig = {
  experimental: {
    reactCompiler: true,
  },
}

export default nextConfig
```

Annotations

You can configure the compiler to run in "opt-in" mode as follows:



A screenshot of a code editor window titled "next.config.ts". The code is similar to the previous example but includes an additional `compilationMode` key under the `experimental/reactCompiler` object, set to `'annotation'`.

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

const nextConfig: NextConfig = {
  experimental: {
    reactCompiler: {
      compilationMode: 'annotation',
    },
  },
}
```

```
export default nextConfig
```

Then, you can annotate specific components or hooks with the "use memo" directive from React to opt-in:

app/page.tsx

TypeScript



```
export default function Page() {  
  'use memo'  
  // ...  
}
```

Note: You can also use the "use no memo" directive from React for the opposite effect, to opt-out a component or hook.

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

reactMaxHeaderLength

During static rendering, React can emit headers that can be added to the response. These can be used to improve performance by allowing the browser to preload resources like fonts, scripts, and stylesheets. The default value is `6000`, but you can override this value by configuring the `reactMaxHeadersLength` option in `next.config.js`:

next.config.js



```
module.exports = {  
  reactMaxHeadersLength: 1000,  
}
```

Good to know: This option is only available in App Router.

Depending on the type of proxy between the browser and the server, the headers can be truncated. For example, if you are using a reverse proxy that doesn't support long headers, you should set a lower value to ensure that the headers are not truncated.

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

reactStrictMode

Good to know: Since Next.js 13.5.1, Strict Mode is `true` by default with `app` router, so the above configuration is only necessary for `pages`. You can still disable Strict Mode by setting `reactStrictMode: false`.

Suggested: We strongly suggest you enable Strict Mode in your Next.js application to better prepare your application for the future of React.

React's [Strict Mode](#) is a development mode only feature for highlighting potential problems in an application. It helps to identify unsafe lifecycles, legacy API usage, and a number of other features.

The Next.js runtime is Strict Mode-compliant. To opt-in to Strict Mode, configure the following option in your `next.config.js`:

next.config.js



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

If you or your team are not ready to use Strict Mode in your entire application, that's OK! You can incrementally migrate on a page-by-page basis using `<React.StrictMode>`.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

redirects

Redirects allow you to redirect an incoming request path to a different destination path.

To use redirects you can use the `redirects` key in `next.config.js`:

 next.config.js



```
module.exports = {
  async redirects() {
    return [
      {
        source: '/about',
        destination: '/',
        permanent: true,
      },
    ]
  },
}
```

`redirects` is an `async` function that expects an array to be returned holding objects with `source`, `destination`, and `permanent` properties:

- `source` is the incoming request path pattern.
- `destination` is the path you want to route to.
- `permanent` `true` or `false` - if `true` will use the 308 status code which instructs clients/search engines to cache the redirect forever, if `false` will use the 307 status code which is temporary and is not cached.

Why does Next.js use 307 and 308? Traditionally a 302 was used for a temporary redirect, and a 301 for

a permanent redirect, but many browsers changed the request method of the redirect to `GET`, regardless of the original method. For example, if the browser made a request to `POST /v1/users` which returned status code `302` with location `/v2/users`, the subsequent request might be `GET /v2/users` instead of the expected `POST /v2/users`. Next.js uses the `307` temporary redirect, and `308` permanent redirect status codes to explicitly preserve the request method used.

- `basePath`: `false` or `undefined` - if false the `basePath` won't be included when matching, can be used for external redirects only.
- `locale`: `false` or `undefined` - whether the locale should not be included when matching.
- `has` is an array of `has` objects with the `type`, `key` and `value` properties.
- `missing` is an array of `missing` objects with the `type`, `key` and `value` properties.

Redirects are checked before the filesystem which includes pages and `/public` files.

When using the Pages Router, redirects are not applied to client-side routing (`Link`, `router.push`) unless `Middleware` is present and matches the path.

When a redirect is applied, any query values provided in the request will be passed through to the redirect destination. For example, see the following redirect configuration:

```
{  
  source: '/old-blog/:path*',  
  destination: '/blog/:path*',  
  permanent: false  
}
```

Good to know: Remember to include the forward slash `/` before the colon `:` in path parameters of the

`source` and `destination` paths, otherwise the path will be treated as a literal string and you run the risk of causing infinite redirects.

When `/old-blog/post-1?hello=world` is requested, the client will be redirected to `/blog/post-1?hello=world`.

Path Matching

Path matches are allowed, for example

`/old-blog/:slug` will match

`/old-blog/hello-world` (no nested paths):

`next.config.js`

```
module.exports = {
  async redirects() {
    return [
      {
        source: '/old-blog/:slug',
        destination: '/news/:slug', // Matches
        permanent: true,
      },
    ]
  },
}
```

Wildcard Path Matching

To match a wildcard path you can use `*` after a parameter, for example `/blog/:slug*` will match `/blog/a/b/c/d/hello-world`:

`next.config.js`

```
module.exports = {
  async redirects() {
    return [
      {
        source: '/blog/:slug*',
```

```
        destination: '/news/:slug*', // Match
        permanent: true,
    },
]
},
}
```

Regex Path Matching

To match a regex path you can wrap the regex in parentheses after a parameter, for example

/post/:slug(\d{1,}) will match /post/123

but not /post/abc :

next.config.js

```
module.exports = {
  async redirects() {
    return [
      {
        source: '/post/:slug(\d{1,})',
        destination: '/news/:slug', // Match
        permanent: false,
      },
    ]
  },
}
```

The following characters (,), {, }, :, *, +, ? are used for regex path matching, so when used in the source as non-special values they must be escaped by adding \ before them:

next.config.js

```
module.exports = {
  async redirects() {
    return [
      {
        // this will match `/english(default)`
        source: '/english\\(default\\)\\/:slug',
        destination: '/en-us/:slug',
        permanent: false,
      },
    ]
  },
}
```

}

Header, Cookie, and Query Matching

To only match a redirect when header, cookie, or query values also match the `has` field or don't match the `missing` field can be used. Both the `source` and all `has` items must match and all `missing` items must not match for the redirect to be applied.

`has` and `missing` items can have the following fields:

- `type : String` - must be either `header`, `cookie`, `host`, or `query`.
- `key : String` - the key from the selected type to match against.
- `value : String` or `undefined` - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value `first-(?<paramName>.*)` is used for `first-second` then `second` will be usable in the destination with `:paramName`.

next.config.js



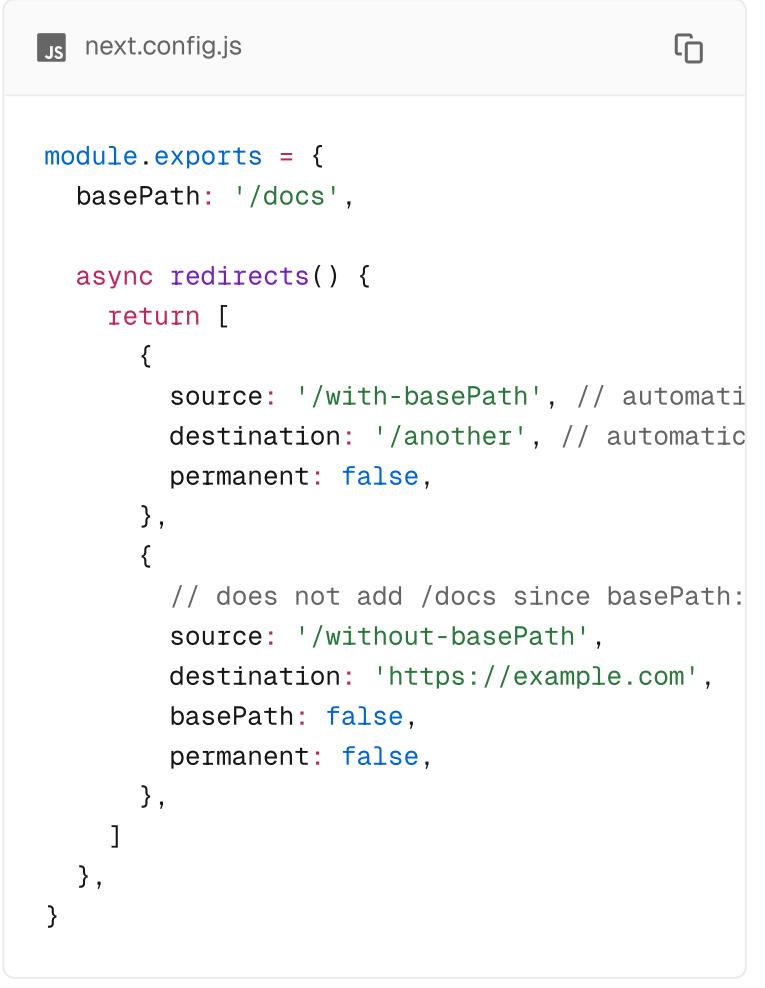
```
module.exports = {
  async redirects() {
    return [
      // if the header `x-redirect-me` is pre
      // this redirect will be applied
      {
        source: '/:path((?!another-page$).*)'
        has: [
          {
            type: 'header',
```

```
        key: 'x-redirect-me',
    },
],
permanent: false,
destination: '/another-page',
},
// if the header `x-dont-redirect` is present
// this redirect will NOT be applied
{
source: '/:path((?!another-page$).*)'
missing: [
{
type: 'header',
key: 'x-do-not-redirect',
},
],
permanent: false,
destination: '/another-page',
},
// if the source, query, and cookie are present
// this redirect will be applied
{
source: '/specific/:path*',
has: [
{
type: 'query',
key: 'page',
// the page value will not be available in the
// destination since value is provided via query
// use a named capture group e.g.
value: 'home',
},
{
type: 'cookie',
key: 'authorized',
value: 'true',
},
],
permanent: false,
destination: '/another/:path*',
},
// if the header `x-authorized` is present and
// contains a matching value, this redirect is applied
{
source: '/',
has: [
{
type: 'header',
key: 'x-authorized',
value: '(?<authorized>yes|true)',
},
],
permanent: false,
destination: '/home?authorized=:authorized',
},
```

```
// if the host is `example.com`,  
// this redirect will be applied  
{  
  source: '/:path((?!another-page$).*)'  
  has: [  
    {  
      type: 'host',  
      value: 'example.com',  
    },  
    ],  
    permanent: false,  
    destination: '/another-page',  
  },  
}  
]  
},  
}
```

Redirects with basePath support

When leveraging `basePath` support with redirects each `source` and `destination` is automatically prefixed with the `basePath` unless you add `basePath: false` to the redirect:



```
js next.config.js
```

```
module.exports = {  
  basePath: '/docs',  
  
  async redirects() {  
    return [  
      {  
        source: '/with-basePath', // automatic  
        destination: '/another', // automatic  
        permanent: false,  
      },  
      {  
        // does not add /docs since basePath:  
        source: '/without-basePath',  
        destination: 'https://example.com',  
        basePath: false,  
        permanent: false,  
      },  
    ]  
  },  
}
```

Redirects with i18n support

When implementing redirects with internationalization in the App Router, you can include locales in `next.config.js` redirects, but only as hardcoded paths.

For dynamic or per-request locale handling, use [dynamic route segments and middleware](#), which can redirect based on the user's preferred language.

```
JS next.config.js

module.exports = {
  async redirects() {
    return [
      {
        // Manually handle locale prefixes for
        source: '/en/old-path',
        destination: '/en/new-path',
        permanent: false,
      },
      {
        // Redirect for all locales using a placeholder
        source: '/:locale/old-path',
        destination: '/:locale/new-path',
        permanent: false,
      },
      {
        // Redirect from one locale to another
        source: '/de/old-path',
        destination: '/en/new-path',
        permanent: false,
      },
      {
        // Catch-all redirect for multiple locales
        source: '/:locale(en|fr|de)/:path*',
        destination: '/:locale/new-section/:path',
        permanent: false,
      },
    ]
  }
}
```

In some rare cases, you might need to assign a custom status code for older HTTP Clients to properly redirect. In these cases, you can use the `statusCode` property instead of the `permanent`

property, but not both. To ensure IE11 compatibility, a `Refresh` header is automatically added for the 308 status code.

Other Redirects

- Inside [API Routes](#) and [Route Handlers](#), you can redirect based on the incoming request.
 - Inside `getStaticProps` and `getServerSideProps`, you can redirect specific pages at request-time.
-

Version History

Version	Changes
v13.3.0	<code>missing</code> added.
v10.2.0	<code>has</code> added.
v9.5.0	<code>redirects</code> added.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

rewrites

Rewrites allow you to map an incoming request path to a different destination path.

Rewrites act as a URL proxy and mask the destination path, making it appear the user hasn't changed their location on the site. In contrast, [redirects](#) will reroute to a new page and show the URL changes.

To use rewrites you can use the `rewrites` key in `next.config.js`:

 next.config.js



```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/about',
        destination: '/',
      },
    ],
  },
}
```

Rewrites are applied to client-side routing. In the example above, navigating to `<Link href="/about">` will serve content from `/` while keeping the URL as `/about`.

`rewrites` is an `async` function that expects to return either an array or an object of arrays (see below) holding objects with `source` and `destination` properties:

- `source`: `String` - is the incoming request path pattern.
- `destination`: `String` is the path you want to route to.
- `basePath`: `false` or `undefined` - if false the basePath won't be included when matching, can be used for external rewrites only.
- `locale`: `false` or `undefined` - whether the locale should not be included when matching.
- `has` is an array of [has objects](#) with the `type`, `key` and `value` properties.
- `missing` is an array of [missing objects](#) with the `type`, `key` and `value` properties.

When the `rewrites` function returns an array, rewrites are applied after checking the filesystem (pages and `/public` files) and before dynamic routes. When the `rewrites` function returns an object of arrays with a specific shape, this behavior can be changed and more finely controlled, as of `v10.1` of Next.js:

```
JS next.config.js
```

```
module.exports = {
  async rewrites() {
    return {
      beforeFiles: [
        // These rewrites are checked after pages
        // and before all files including _next
        // allows overriding page files
        {
          source: '/some-page',
          destination: '/somewhere-else',
          has: [{ type: 'query', key: 'override' }],
        },
      ],
      afterFiles: [
        // These rewrites are checked after pages
        // are checked but before dynamic routes
        {
          source: '/non-existent',
          destination: '/somewhere-else',
        },
      ],
    };
  }
};
```

```
        },
      ],
      fallback: [
        // These rewrites are checked after dynamic routes
        // and static files are checked
        {
          source: '/:path*',
          destination: `https://my-old-site.com${req.url}`
        },
      ],
    },
  ],
}
```

Good to know: rewrites in `beforeFiles` do not check the filesystem/dynamic routes immediately after matching a source, they continue until all `beforeFiles` have been checked.

The order Next.js routes are checked is:

1. `headers` are checked/applied
2. `redirects` are checked/applied
3. `beforeFiles` rewrites are checked/applied
4. static files from the [public directory](#), `_next/static` files, and non-dynamic pages are checked/served
5. `afterFiles` rewrites are checked/applied, if one of these rewrites is matched we check dynamic routes/static files after each match
6. `fallback` rewrites are checked/applied, these are applied before rendering the 404 page and after dynamic routes/all static assets have been checked. If you use `fallback: true/'blocking'` in `getStaticPaths`, the fallback `rewrites` defined in your `next.config.js` will *not* be run.

Rewrite parameters

When using parameters in a rewrite the parameters will be passed in the query by default when none of the parameters are used in the destination.

next.config.js

```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/old-about/:path*',
        destination: '/about', // The :path p
      },
    ]
  },
}
```

If a parameter is used in the destination none of the parameters will be automatically passed in the query.

next.config.js

```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/docs/:path*',
        destination: '/:path*', // The :path
      },
    ]
  },
}
```

You can still pass the parameters manually in the query if one is already used in the destination by specifying the query in the destination.

next.config.js

```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/old-about/:path*',
        destination: '/about?query=:path'
      },
    ]
  },
}
```

```
{  
  source: '/:first/:second',  
  destination: '/:first?second=:second'  
  // Since the :first parameter is used  
  // will not automatically be added in  
  // as shown above  
},  
]  
,  
}
```

Good to know: Static pages from [Automatic Static Optimization](#) or [prerendering](#) params from rewrites will be parsed on the client after hydration and provided in the query.

Path Matching

Path matches are allowed, for example

`/blog/:slug` will match `/blog/hello-world` (no nested paths):

next.config.js

```
module.exports = {  
  async rewrites() {  
    return [  
      {  
        source: '/blog/:slug',  
        destination: '/news/:slug', // Matches  
      },  
    ]  
  },  
}
```

Wildcard Path Matching

To match a wildcard path you can use `*` after a parameter, for example `/blog/:slug*` will match `/blog/a/b/c/d/hello-world`:

JS next.config.js

```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/blog/:slug*',
        destination: '/news/:slug*', // Match
      },
    ],
  },
}
```

Regex Path Matching

To match a regex path you can wrap the regex in parenthesis after a parameter, for example

/blog/:slug(\d{1,}) will match /blog/123

but not /blog/abc :

JS next.config.js

```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/old-blog/:post(\d{1,})',
        destination: '/blog/:post', // Match
      },
    ],
  },
}
```

The following characters (,), {, }, [,], |, \, ^, ., :, *, +, -, ?, \$ are used for regex path matching, so when used in the `source` as non-special values they must be escaped by adding \\ before them:

JS next.config.js

```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/english\\(default\\)',
        destination: '/english'
      }
    ]
  }
}
```

```
        source: '/english\\(default\\)/:slug',
        destination: '/en-us/:slug',
    },
]
},
}
```

Header, Cookie, and Query Matching

To only match a rewrite when header, cookie, or query values also match the `has` field or don't match the `missing` field can be used. Both the `source` and all `has` items must match and all `missing` items must not match for the rewrite to be applied.

`has` and `missing` items can have the following fields:

- `type: String` - must be either `header`, `cookie`, `host`, or `query`.
- `key: String` - the key from the selected type to match against.
- `value: String` or `undefined` - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value `first-(?<paramName>.*)` is used for `first-second` then `second` will be usable in the destination with `:paramName`.

next.config.js

```
module.exports = {
  async rewrites() {
    return [
      // if the header `x-rewrite-me` is present
      // this rewrite will be applied
    ]
  }
}
```

```

        },
        source: '/:path*',
        has: [
            {
                type: 'header',
                key: 'x-rewrite-me',
            },
        ],
        destination: '/another-page',
    },
    // if the header `x-rewrite-me` is not
    // this rewrite will be applied
{
    source: '/:path*',
    missing: [
        {
            type: 'header',
            key: 'x-rewrite-me',
        },
    ],
    destination: '/another-page',
},
// if the source, query, and cookie are
// this rewrite will be applied
{
    source: '/specific/:path*',
    has: [
        {
            type: 'query',
            key: 'page',
            // the page value will not be ava
            // destination since value is pro
            // use a named capture group e.g.
            value: 'home',
        },
        {
            type: 'cookie',
            key: 'authorized',
            value: 'true',
        },
    ],
    destination: '/:path*/home',
},
// if the header `x-authorized` is pres
// contains a matching value, this rew
{
    source: '/:path*',
    has: [
        {
            type: 'header',
            key: 'x-authorized',
            value: '(?<authorized>yes|true)',
        },
    ],
    destination: '/home?authorized=:autho

```

```
,  
  // if the host is `example.com`,  
  // this rewrite will be applied  
{  
  source: '/:path*',  
  has: [  
    {  
      type: 'host',  
      value: 'example.com',  
    },  
    ],  
  destination: '/another-page',  
},  
]  
},  
}
```

Rewriting to an external URL

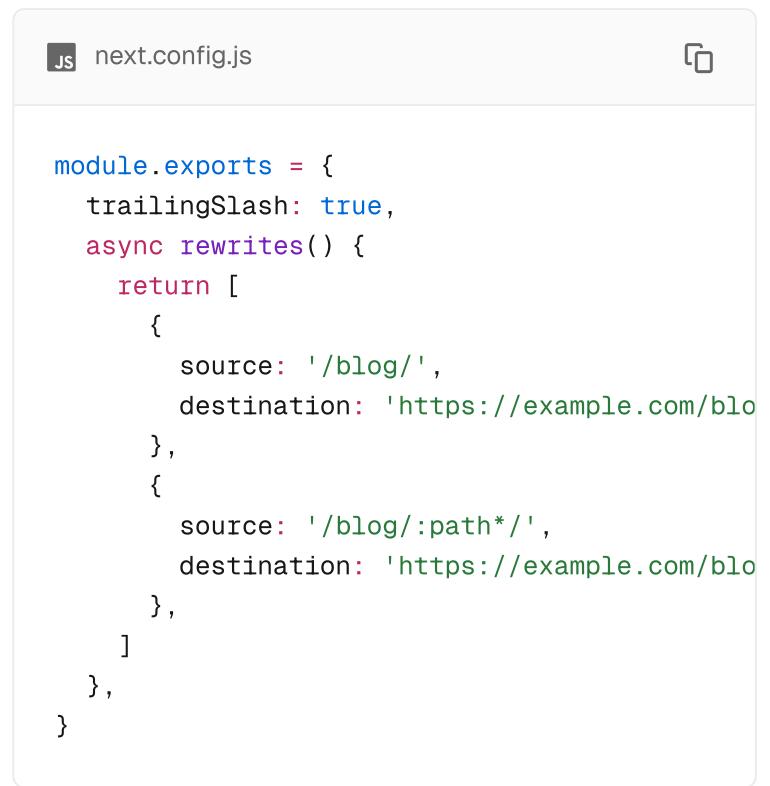
► Examples

Rewrites allow you to rewrite to an external URL. This is especially useful for incrementally adopting Next.js. The following is an example rewrite for redirecting the `/blog` route of your main app to an external site.

```
next.config.js
```

```
module.exports = {  
  async rewrites() {  
    return [  
      {  
        source: '/blog',  
        destination: 'https://example.com/blo  
      },  
      {  
        source: '/blog/:slug',  
        destination: 'https://example.com/blo  
      },  
    ]  
  },  
}
```

If you're using `trailingSlash: true`, you also need to insert a trailing slash in the `source` parameter. If the destination server is also expecting a trailing slash it should be included in the `destination` parameter as well.



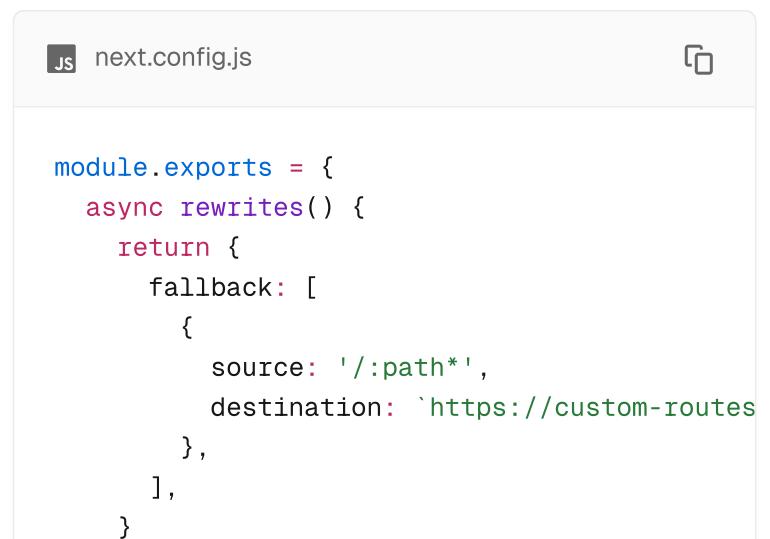
```
next.config.js
```

```
module.exports = {
  trailingSlash: true,
  async rewrites() {
    return [
      {
        source: '/blog/',
        destination: 'https://example.com/blog',
      },
      {
        source: '/blog/:path*/',
        destination: 'https://example.com/blog/:path',
      },
    ]
  },
}
```

Incremental adoption of Next.js

You can also have Next.js fall back to proxying to an existing website after checking all Next.js routes.

This way you don't have to change the rewrites configuration when migrating more pages to Next.js



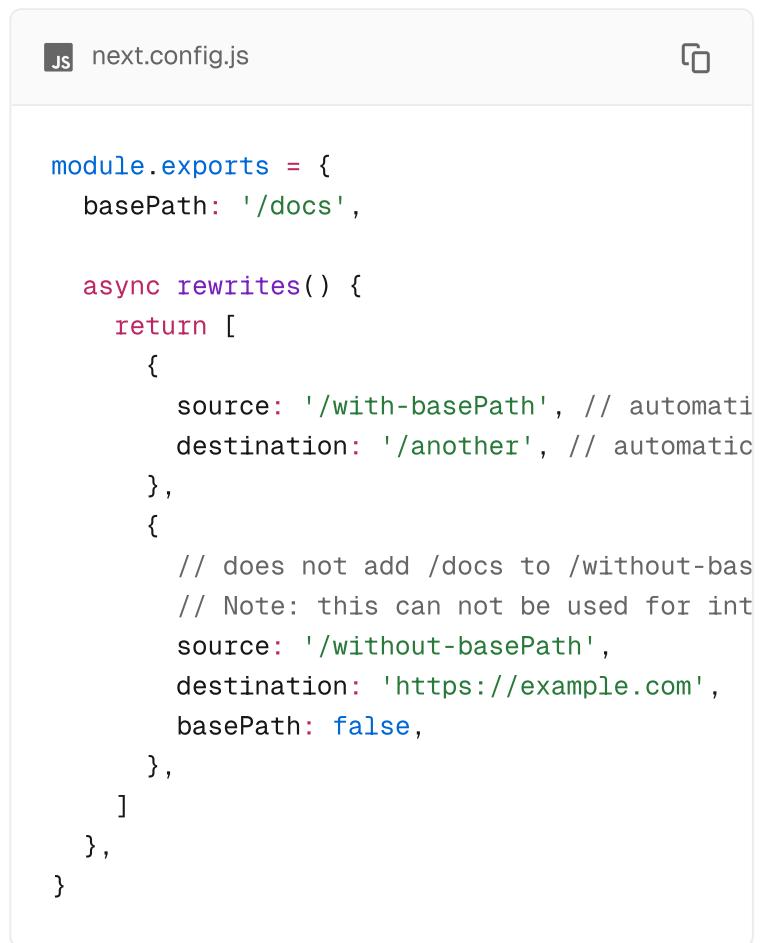
```
next.config.js
```

```
module.exports = {
  async rewrites() {
    return {
      fallback: [
        {
          source: '/:path*',
          destination: `https://custom-routes${process.env.CUSTOM_ROUTES}`,
        },
      ],
    }
  }
}
```

```
},  
}
```

Rewrites with basePath support

When leveraging `basePath` support with rewrites each `source` and `destination` is automatically prefixed with the `basePath` unless you add `basePath: false` to the rewrite:



The screenshot shows a code editor window with a file named `next.config.js`. The code defines a `module.exports` object with a `basePath` set to `'/docs'`. It contains an `async rewrites()` function that returns an array of rewrite rules. One rule has a `source` of `'/with-basePath'` and a `destination` of `'/another'`. Another rule has a `source` of `'/without-basePath'` and a `destination` of `'https://example.com'`, with `basePath: false` specified.

```
JS next.config.js

module.exports = {
  basePath: '/docs',

  async rewrites() {
    return [
      {
        source: '/with-basePath', // automatic
        destination: '/another', // automatic
      },
      {
        // does not add /docs to /without-bas
        // Note: this can not be used for int
        source: '/without-basePath',
        destination: 'https://example.com',
        basePath: false,
      },
    ]
  },
}
```

Version History

Version	Changes
v13.3.0	<code>missing</code> added.
v10.2.0	<code>has</code> added.
v9.5.0	Headers added.

Was this helpful?  



 Using App Router
Features available in /app

 Latest Version
15.5.4

sassOptions

`sassOptions` allow you to configure the Sass compiler.

 next.config.ts TypeScript ▾

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

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

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

export default nextConfig
```

Good to know: `sassOptions` are not typed outside of `implementation` because Next.js does not maintain the other possible properties.

Was this helpful?





Using App Router

Features available in /app



Latest Version

15.5.4



serverActions

Options for configuring Server Actions behavior in your Next.js application.

allowedOrigins

A list of extra safe origin domains from which Server Actions can be invoked. Next.js compares the origin of a Server Action request with the host domain, ensuring they match to prevent CSRF attacks. If not provided, only the same origin is allowed.

next.config.js



```
/** @type {import('next').NextConfig} */

module.exports = {
  experimental: {
    serverActions: {
      allowedOrigins: ['my-proxy.com', '*.my-'],
    },
  }
}
```

bodySizeLimit

By default, the maximum size of the request body sent to a Server Action is 1MB, to prevent the

consumption of excessive server resources in parsing large amounts of data, as well as potential DDoS attacks.

However, you can configure this limit using the `serverActions.bodySizeLimit` option. It can take the number of bytes or any string format supported by bytes, for example `1000`, `'500kb'` or `'3mb'`.

```
JS next.config.js

/** @type {import('next').NextConfig} */

module.exports = {
  experimental: {
    serverActions: {
      bodySizeLimit: '2mb',
    },
  },
}
```

Enabling Server Actions (v13)

Server Actions became a stable feature in Next.js 14, and are enabled by default. However, if you are using an earlier version of Next.js, you can enable them by setting `experimental.serverActions` to `true`.

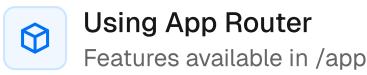
```
JS next.config.js

/** @type {import('next').NextConfig} */
const config = {
  experimental: {
    serverActions: true,
  },
}

module.exports = config
```

Was this helpful?  





serverComponentsHmrCache

This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

The experimental `serverComponentsHmrCache` option allows you to cache `fetch` responses in Server Components across Hot Module Replacement (HMR) refreshes in local development. This results in faster responses and reduced costs for billed API calls.

By default, the HMR cache applies to all `fetch` requests, including those with the `cache: 'no-store'` option. This means uncached requests will not show fresh data between HMR refreshes. However, the cache will be cleared on navigation or full-page reloads.

You can disable the HMR cache by setting `serverComponentsHmrCache` to `false` in your `next.config.js` file:

```
TS next.config.ts TypeScript ▾
```

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    serverComponentsHmrCache: false, // default
  },
}

export default nextConfig
```

Good to know: For better observability, we recommend using the `logging.fetches` option which logs fetch cache hits and misses in the console during development.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

serverExternalPack- ages

Dependencies used inside [Server Components](#) and [Route Handlers](#) will automatically be bundled by Next.js.

If a dependency is using Node.js specific features, you can choose to opt-out specific dependencies from the Server Components bundling and use native Node.js `require`.

 next.config.js



```
/* @type {import('next').NextConfig} */
const nextConfig = {
  serverExternalPackages: ['@acme/ui'],
}

module.exports = nextConfig
```

Next.js includes a [short list of popular packages](#) that currently are working on compatibility and automatically opt-ed out:

- `@appsignal/nodejs`
- `@aws-sdk/client-s3`
- `@aws-sdk/s3-presigned-post`
- `@blockfrost/blockfrost-js`
- `@highlight-run/node`
- `@huggingface/transformers`
- `@jpg-store/lucid-cardano`

- `@libsql/client`
- `@mikro-orm/core`
- `@mikro-orm/knex`
- `@node-rs/argon2`
- `@node-rs/bcrypt`
- `@prisma/client`
- `@react-pdf/renderer`
- `@sentry/profiling-node`
- `@sparticuz/chromium`
- `@sparticuz/chromium-min`
- `@swc/core`
- `@xenova/transformers`
- `argon2`
- `autoprefixer`
- `aws-crt`
- `bcrypt`
- `better-sqlite3`
- `canvas`
- `chromadb-default-embed`
- `config`
- `cpu-features`
- `cypress`
- `dd-trace`
- `eslint`
- `express`
- `firebase-admin`
- `htmlrewriter`
- `import-in-the-middle`
- `isolated-vm`
- `jest`

- `jsdom`
- `keyv`
- `libsql`
- `mdx-bundler`
- `mongodb`
- `mongoose`
- `newrelic`
- `next-mdx-remote`
- `next-seo`
- `node-cron`
- `node-pty`
- `node-web-audio-api`
- `onnxruntime-node`
- `oslo`
- `pg`
- `playwright`
- `playwright-core`
- `postcss`
- `prettier`
- `prisma`
- `puppeteer-core`
- `puppeteer`
- `ravendb`
- `require-in-the-middle`
- `rimraf`
- `sharp`
- `shiki`
- `sqlite3`
- `ts-node`
- `ts-morph`

- `typescript`
- `vscode-oniguruma`
- `webpack`
- `websocket`
- `zeromq`

Version	Changes
v15.0.0	Moved from experimental to stable. Renamed from <code>serverComponentsExternalPackages</code> to <code>serverExternalPackages</code>

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

staleTimes

This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

`staleTimes` is an experimental feature that enables caching of page segments in the [client-side router cache](#).

You can enable this experimental feature and provide custom revalidation times by setting the experimental `staleTimes` flag:

`next.config.js`

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    staleTimes: {
      dynamic: 30,
      static: 180,
    },
  },
}

module.exports = nextConfig
```

The `static` and `dynamic` properties correspond with the time period (in seconds) based on different types of [link prefetching](#).

- The `dynamic` property is used when the page is neither statically generated nor fully prefetched (e.g. with `prefetch={true}`).
- Default: 0 seconds (not cached)

- The `static` property is used for statically generated pages, or when the `prefetch` prop on `Link` is set to `true`, or when calling `router.prefetch`.
 - Default: 5 minutes

Good to know:

- [Loading boundaries](#) are considered reusable for the `static` period defined in this configuration.
- This doesn't affect [partial rendering](#), meaning **shared layouts won't automatically be refetched on every navigation, only the page segment that changes.**
- This doesn't change [back/forward caching](#) behavior to prevent layout shift and to prevent losing the browser scroll position.

You can learn more about the Client Router Cache [here](#).

Version History

Version	Changes
v15.0.0	The <code>dynamic staleTimes</code> default changed from 30s to 0s.
v14.2.0	Experimental <code>staleTimes</code> introduced.

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

staticGeneration*

This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

The `staticGeneration*` options allow you to configure the Static Generation process for advanced use cases.

TS next.config.ts

TypeScript



```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    staticGenerationRetryCount: 1,
    staticGenerationMaxConcurrency: 8,
    staticGenerationMinPagesPerWorker: 25,
  },
}

export default nextConfig
```

Config Options

The following options are available:

- `staticGenerationRetryCount` : The number of times to retry a failed page generation before failing the build.
- `staticGenerationMaxConcurrency` : The maximum number of pages to be processed

per worker.

- `staticGenerationMinPagesPerWorker` : The minimum number of pages to be processed before starting a new worker.

Was this helpful?



taint

 Using App Router
Features available in /app

 Latest Version
15.5.4

taint

 This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

Usage

The `taint` option enables support for experimental React APIs for tainting objects and values. This feature helps prevent sensitive data from being accidentally passed to the client. When enabled, you can use:

- [experimental_taintObjectReference](#) ↗ taint objects references.
- [experimental_taintUniqueValue](#) ↗ to taint unique values.

Good to know: Activating this flag also enables the React `experimental` channel for `app` directory.

 next.config.ts

TypeScript ▼



```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    taint: true,
  },
}

export default nextConfig
```

Warning: Do not rely on the taint API as your only mechanism to prevent exposing sensitive data to the client. See our [security recommendations](#).

The taint APIs allows you to be defensive, by declaratively and explicitly marking data that is not allowed to pass through the Server-Client boundary. When an object or value, is passed through the Server-Client boundary, React throws an error.

This is helpful for cases where:

- The methods to read data are out of your control
- You have to work with sensitive data shapes not defined by you
- Sensitive data is accessed during Server Component rendering

It is recommended to model your data and APIs so that sensitive data is not returned to contexts where it is not needed.

Caveats

- Tainting can only keep track of objects by reference. Copying an object creates an

untainted version, which loses all guarantees given by the API. You'll need to taint the copy.

- Tainting cannot keep track of data derived from a tainted value. You also need to taint the derived value.
- Values are tainted for as long as their lifetime reference is within scope. See the [experimental_taintUniqueValue](#) parameters reference ↗, for more information.

Examples

Tainting an object reference

In this case, the `getUserDetails` function returns data about a given user. We taint the user object reference, so that it cannot cross a Server-Client boundary. For example, assuming `UserCard` is a Client Component.

```
import { experimental_taintObjectReference }

function getUserDetails(id: string): UserDetail
  const user = await db.queryUserById(id)

  experimental_taintObjectReference(
    'Do not use the entire user info object.
    user
  )

  return user
}
```

We can still access individual fields from the tainted `userDetails` object.

```
export async function ContactPage({
  params,
}: {
  params: Promise<{ id: string }>
})
```

```

}) {
  const { id } = await params
  const userDetails = await getUserDetails(id)

  return (
    <UserCard
      firstName={userDetails.firstName}
      lastName={userDetails.lastName}
    />
  )
}

```

Now, passing the entire object to the Client Component will throw an error.

```

export async function ContactPage({
  params,
}: {
  params: Promise<{ id: string }>
}) {
  const userDetails = await getUserDetails(id)

  // Throws an error
  return <UserCard user={userDetails} />
}

```

Tainting a unique value

In this case, we can access the server configuration by awaiting calls to `config.getConfigDetails`. However, the system configuration contains the `SERVICE_API_KEY` that we don't want to expose to clients.

We can taint the `config.SERVICE_API_KEY` value.

```

import { experimental_taintUniqueValue } from

function getSystemConfig(): SystemConfig {
  const config = await config.getConfigDetail

  experimental_taintUniqueValue(
    'Do not pass configuration tokens to the
    config,
    config.SERVICE_API_KEY
  )
}

```

```
    return config  
}
```

We can still access other properties of the `systemConfig` object.

```
export async function Dashboard() {  
  const systemConfig = await getSystemConfig()  
  
  return <ClientDashboard version={systemConfig.version} />  
}
```

However, passing `SERVICE_API_KEY` to `ClientDashboard` throws an error.

```
export async function Dashboard() {  
  const systemConfig = await getSystemConfig()  
  // Someone makes a mistake in a PR  
  const version = systemConfig.SERVICE_API_KEY  
  
  return <ClientDashboard version={version} />  
}
```

Note that, even though, `systemConfig.SERVICE_API_KEY` is reassigned to a new variable. Passing it to a Client Component still throws an error.

Whereas, a value derived from a tainted unique value, will be exposed to the client.

```
export async function Dashboard() {  
  const systemConfig = await getSystemConfig()  
  // Someone makes a mistake in a PR  
  const version = `version:${systemConfig.SERVICE_API_KEY}`  
  
  return <ClientDashboard version={version} />  
}
```

A better approach is to remove `SERVICE_API_KEY` from the data returned by `getSystemConfig`.

Was this helpful?    

trailingSlash

 Using App Router
Features available in /app

 Latest Version
15.5.4

trailingSlash

By default Next.js will redirect URLs with trailing slashes to their counterpart without a trailing slash. For example `/about/` will redirect to `/about`. You can configure this behavior to act the opposite way, where URLs without trailing slashes are redirected to their counterparts with trailing slashes.

Open `next.config.js` and add the `trailingSlash` config:

```
JS next.config.js
```

```
module.exports = {  
  trailingSlash: true,  
}
```

With this option set, URLs like `/about` will redirect to `/about/`.

When using `trailingSlash: true`, certain URLs are exceptions and will not have a trailing slash appended:

- Static file URLs, such as files with extensions.
- Any paths under `.well-known/`.

For example, the following URLs will remain unchanged: `/file.txt`, `images/photos/picture.png`, and `.well-known/subfolder/config.json`.

When used with `output: "export"` configuration,
the `/about` page will output `/about/index.html`
(instead of the default `/about.html`).

Version History

Version	Changes
v9.5.0	<code>trailingSlash</code> added.

Was this helpful? 



Using App Router

Features available in /app



Latest Version

15.5.4



transpilePackages

Next.js can automatically transpile and bundle dependencies from local packages (like monorepos) or from external dependencies (`node_modules`). This replaces the `next-transpile-modules` package.

next.config.js



```
/** @type {import('next').NextConfig} */
const nextConfig = {
  transpilePackages: ['package-name'],
}

module.exports = nextConfig
```

Version History

Version

Changes

v13.0.0

transpilePackages added.

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4

turbopack

The `turbopack` option lets you customize [Turbopack](#) to transform different files and change how modules are resolved.

TS

next.config.ts

TypeScript



```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  turbopack: {
    // ...
  },
}

export default nextConfig
```

Good to know:

- Turbopack for Next.js does not require loaders or loader configuration for built-in functionality. Turbopack has built-in support for CSS and compiling modern JavaScript, so there's no need for `css-loader`, `postcss-loader`, or `babel-loader` if you're using `@babel/preset-env`.

Reference

Options

The following options are available for the `turbo` configuration:

Option	Description
<code>root</code>	Sets the application root directory. Should be an absolute path.
<code>rules</code>	List of supported webpack loaders to apply when running with Turbopack.
<code>resolveAlias</code>	Map aliased imports to modules to load in their place.
<code>resolveExtensions</code>	List of extensions to resolve when importing files.

Supported loaders

The following loaders have been tested to work
with Turbopack's webpack loader implementation,
but many other webpack loaders should work as
well even if not listed here:

- [babel-loader ↗](#)
- [@svgr/webpack ↗](#)
- [svg-inline-loader ↗](#)
- [yaml-loader ↗](#)
- [string-replace-loader ↗](#)
- [raw-loader ↗](#)
- [sass-loader ↗](#)
- [graphql-tag/loader ↗](#)

Examples

Root directory

Turbopack uses the root directory to resolve modules. Files outside of the project root are not resolved.

Next.js automatically detects the root directory of your project. It does so by looking for one of these files:

- `pnpm-lock.yaml`
- `package-lock.json`
- `yarn.lock`
- `bun.lock`
- `bun.lockb`

If you have a different project structure, for example if you don't use workspaces, you can manually set the `root` option:



A screenshot of a code editor showing a file named `next.config.js`. The code defines a module export for the `turbopack` configuration, specifically setting the `root` to the parent directory of the current file.

```
const path = require('path')
module.exports = {
  turbopack: {
    root: path.join(__dirname, '..'),
  },
}
```

Configuring webpack loaders

If you need loader support beyond what's built in, many webpack loaders already work with Turbopack. There are currently some limitations:

- Only a core subset of the webpack loader API is implemented. Currently, there is enough coverage for some popular loaders, and we'll expand our API support in the future.
- Only loaders that return JavaScript code are supported. Loaders that transform files like

stylesheets or images are not currently supported.

- Options passed to webpack loaders must be plain JavaScript primitives, objects, and arrays.
For example, it's not possible to pass `require()` plugin modules as option values.

To configure loaders, add the names of the loaders you've installed and any options in

`next.config.js`, mapping file extensions to a list of loaders.

Here is an example below using the `@svgr/webpack` loader, which enables importing `.svg` files and rendering them as React components.

```
JS next.config.js

module.exports = {
  turbopack: {
    rules: {
      '*.svg': {
        loaders: ['@svgr/webpack'],
        as: '*.js',
      },
    },
  },
}
```

For loaders that require configuration options, you can use an object format instead of a string:

```
JS next.config.js

module.exports = {
  turbopack: {
    rules: {
      '*.svg': {
        loaders: [
          {
            loader: '@svgr/webpack',
            options: {
              icon: true,
            }
          }
        ]
      }
    }
  }
}
```

```
        },
        },
        ],
        as: '*.js',
    },
},
},
}
```

Good to know: Prior to Next.js version 13.4.4, `turbo.rules` was named `turbo.loaders` and only accepted file extensions like `.mdx` instead of `*.mdx`.

Resolving aliases

Turbopack can be configured to modify module resolution through aliases, similar to webpack's [resolve.alias](#) configuration.

To configure resolve aliases, map imported patterns to their new destination in

`next.config.js`:

```
JS next.config.js
```

```
module.exports = {
  turbopack: {
    resolveAlias: {
      underscore: 'lodash',
      mocha: { browser: 'mocha/browser-entry.' },
    },
  },
}
```

This aliases imports of the `underscore` package to the `lodash` package. In other words, `import underscore from 'underscore'` will load the `lodash` module instead of `underscore`.

Turbopack also supports conditional aliasing through this field, similar to Node.js' [conditional exports](#). At the moment only the `browser` condition is supported. In the case above, imports

of the `mocha` module will be aliased to `mocha/browser-entry.js` when Turbopack targets browser environments.

Resolving custom extensions

Turbopack can be configured to resolve modules with custom extensions, similar to webpack's [resolve.extensions](#) configuration.

To configure resolve extensions, use the `resolveExtensions` field in `next.config.js`:

```
JS next.config.js Copy  
  
module.exports = {  
  turbopack: {  
    resolveExtensions: ['.mdx', '.tsx', '.ts']  
  },  
}
```

This overwrites the original resolve extensions with the provided list. Make sure to include the default extensions.

For more information and guidance for how to migrate your app to Turbopack from webpack, see [Turbopack's documentation on webpack compatibility](#).

Version History

Version Changes

`15.3.0` `experimental.turbo` is changed to `turbopack`.

`13.0.0` `experimental.turbo` introduced.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

turbopackPersistent-Caching

 This feature is currently available in the canary channel and subject to change. Try it out by upgrading Next.js, and share your feedback on GitHub.

Usage

Turbopack Persistent Caching enables Turbopack to reduce work across `next dev` or `next build` commands. When enabled, Turbopack will save and restore data to the `.next` folder between builds, which can greatly speed up subsequent builds and dev sessions.

Warning: Persistent Caching is still under development and is not yet stable. Users adopting should expect some stability issues.

Good to know: Note that while `next dev` and `next build` can share cached data with each other, most cache entries are command-specific due to different configuration and environment variables.

 next.config.ts

TypeScript ▾



```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
```

```
turbopackPersistentCaching: true,  
},  
}  
  
export default nextConfig
```

Version Changes

Version	Changes
---------	---------

v15.5.0	Persistent caching released as experimental
---------	---

Was this helpful?    

typedRoutes

 Using App Router
Features available in /app Latest Version
15.5.4

typedRoutes

Note: This option has been marked as stable, so you should use `typedRoutes` instead of `experimental.typedRoutes`.

Support for [statically typed links](#). This feature requires using TypeScript in your project.

 next.config.js

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  typedRoutes: true,
}

module.exports = nextConfig
```

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

typescript

Next.js fails your **production build** (`next build`) when TypeScript errors are present in your project.

If you'd like Next.js to dangerously produce production code even when your application has errors, you can disable the built-in type checking step.

If disabled, be sure you are running type checks as part of your build or deploy process, otherwise this can be very dangerous.

Open `next.config.js` and enable the `ignoreBuildErrors` option in the `typescript` config:

```
JS next.config.js 
```

```
module.exports = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```

Was this helpful?

Using App Router
Features available in /app

Latest Version
15.5.4

urlImports

This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

URL imports are an experimental feature that allows you to import modules directly from external servers (instead of from the local disk).

Warning: Only use domains that you trust to download and execute on your machine. Please exercise discretion, and caution until the feature is flagged as stable.

To opt-in, add the allowed URL prefixes inside `next.config.js`:

`next.config.js`



```
module.exports = {
  experimental: {
    urlImports: ['https://example.com/assets/'],
  },
}
```

Then, you can import modules directly from URLs:

```
import { a, b, c } from 'https://example.com/
```

URL Imports can be used everywhere normal package imports can be used.

Security Model

This feature is being designed with **security as the top priority**. To start, we added an experimental flag forcing you to explicitly allow the domains you accept URL imports from. We're working to take this further by limiting URL imports to execute in the browser sandbox using the [Edge Runtime](#).

Lockfile

When using URL imports, Next.js will create a `next.lock` directory containing a lockfile and fetched assets. This directory **must be committed to Git**, not ignored by `.gitignore`.

- When running `next dev`, Next.js will download and add all newly discovered URL Imports to your lockfile.
- When running `next build`, Next.js will use only the lockfile to build the application for production.

Typically, no network requests are needed and any outdated lockfile will cause the build to fail. One exception is resources that respond with `Cache-Control: no-cache`. These resources will have a `no-cache` entry in the lockfile and will always be fetched from the network on each build.

Examples

Skypack

```
import confetti from 'https://cdn.skypack.dev'
import { useEffect } from 'react'

export default () => {
  useEffect(() => {
    confetti()
  })
  return <p>Hello</p>
}
```

Static Image Imports

```
import Image from 'next/image'
import logo from 'https://example.com/assets/'

export default () => (
  <div>
    <Image src={logo} placeholder="blur" />
  </div>
)
```

URLs in CSS

```
.className {
  background: url('https://example.com/assets')
```

Asset Imports

```
const logo = new URL('https://example.com/ass
console.log(logo.pathname)

// prints "/_next/static/media/file.a9727b5d.
```

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

useCache

 This feature is currently available in the canary channel and subject to change. Try it out by upgrading Next.js, and share your feedback on GitHub.

The `useCache` flag is an experimental feature in Next.js that enables the `use cache` directive to be used independently of `cacheComponents`. When enabled, you can use `use cache` in your application even if `cacheComponents` is turned off.

Usage

To enable the `useCache` flag, set it to `true` in the `experimental` section of your `next.config.ts` file:

 next.config.ts

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    useCache: true,
  },
}

export default nextConfig
```

When `useCache` is enabled, you can use the following cache functions and configurations:

- The `use cache` directive
- The `cacheLife` function with `use cache`
- The `cacheTag` function

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

useLightningcss

This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

Experimental support for using [Lightning CSS](#), a fast CSS bundler and minifier, written in Rust.

TS

next.config.ts

TypeScript



```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    useLightningcss: true,
  },
}

export default nextConfig
```

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

viewTransition

This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

`viewTransition` is an experimental flag that enables the new experimental [View Transitions API ↗](#) in React. This API allows you to leverage the native View Transitions browser API to create seamless transitions between UI states.

To enable this feature, you need to set the `viewTransition` property to `true` in your `next.config.js` file.

`next.config.js`

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    viewTransition: true,
  },
}

module.exports = nextConfig
```

Important Notice: This feature is not developed or maintained by the Next.js team — it is an experimental API from the React team. It is still in **early stages and not recommended for production use**. The implementation is still being iterated on, and its behavior may change in future React releases. Enabling this feature requires understanding the experimental nature of the API. To fully grasp its behavior, refer to the [React pull request ↗](#) and related discussions.

Usage

Once enabled, you can import the `ViewTransition` component from React in your application:

```
import { unstable_ViewTransition as ViewTrans
```

However, documentation and examples are currently limited, and you will need to refer directly to React's source code and discussions to understand how this works.

Live Demo

Check out our [Next.js View Transition Demo ↗](#) to see this feature in action.

As this API evolves, we will update our documentation and share more examples. However, for now, we strongly advise against using this feature in production.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

Custom Webpack Config

Good to know: changes to webpack config are not covered by semver so proceed at your own risk

Before continuing to add custom webpack configuration to your application make sure Next.js doesn't already support your use-case:

- [CSS imports](#)
- [CSS modules](#)
- [Sass/SCSS imports](#)
- [Sass/SCSS modules](#)

Some commonly asked for features are available as plugins:

- [@next/mdx ↗](#)
- [@next/bundle-analyzer ↗](#)

In order to extend our usage of `webpack`, you can define a function that extends its config inside `next.config.js`, like so:

`js` next.config.js



```
module.exports = {
  webpack: (
    config,
    { buildId, dev, isServer, defaultLoaders,
  ) => {
    // Important: return the modified config
    return config
```

```
},  
}
```

The `webpack` function is executed three times, twice for the server (nodejs / edge runtime) and once for the client. This allows you to distinguish between client and server configuration using the `isServer` property.

The second argument to the `webpack` function is an object with the following properties:

- `buildId : String` - The build id, used as a unique identifier between builds.
- `dev : Boolean` - Indicates if the compilation will be done in development.
- `isServer : Boolean` - It's `true` for server-side compilation, and `false` for client-side compilation.
- `nextRuntime : String | undefined` - The target runtime for server-side compilation; either `"edge"` or `"nodejs"`, it's `undefined` for client-side compilation.
- `defaultLoaders : Object` - Default loaders used internally by Next.js:
 - `babel : Object` - Default `babel-loader` configuration.

Example usage of `defaultLoaders.babel`:

```
// Example config for adding a loader that de  
// This source was taken from the @next/mdx p  
// https://github.com/vercel/next.js/tree/can  
module.exports = {  
  webpack: (config, options) => {  
    config.module.rules.push({  
      test: /\.mdx/,  
      use: [  
        options.defaultLoaders.babel,  
        {  
          loader: '@mdx-js/loader',
```

```
        options: pluginOptions.options,  
        ],  
    })  
  
    return config  
},  
}  
}
```

nextRuntime

Notice that `isServer` is `true` when `nextRuntime` is `"edge"` or `"nodejs"`, `nextRuntime` `"edge"` is currently for middleware and Server Components in edge runtime only.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4

webVitalsAttribution

 This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

When debugging issues related to Web Vitals, it is often helpful if we can pinpoint the source of the problem. For example, in the case of Cumulative Layout Shift (CLS), we might want to know the first element that shifted when the single largest layout shift occurred. Or, in the case of Largest Contentful Paint (LCP), we might want to identify the element corresponding to the LCP for the page. If the LCP element is an image, knowing the URL of the image resource can help us locate the asset we need to optimize.

Pinpointing the biggest contributor to the Web Vitals score, aka [attribution ↗](#), allows us to obtain more in-depth information like entries for [PerformanceEventTiming ↗](#), [PerformanceNavigationTiming ↗](#) and [PerformanceResourceTiming ↗](#).

Attribution is disabled by default in Next.js but can be enabled **per metric** by specifying the following in `next.config.js`.

 next.config.js

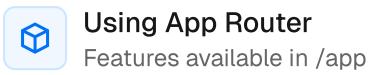


```
module.exports = {
  experimental: {
    webVitalsAttribution: ['CLS', 'LCP'],
  },
}
```

}

Valid attribution values are all `web-vitals` metrics specified in the [NextWebVitalsMetric](#)  type.

Was this helpful?    



TypeScript

Next.js comes with built-in TypeScript, automatically installing the necessary packages and configuring the proper settings when you create a new project with `create-next-app`.

To add TypeScript to an existing project, rename 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.

Good to know: If you already have 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.

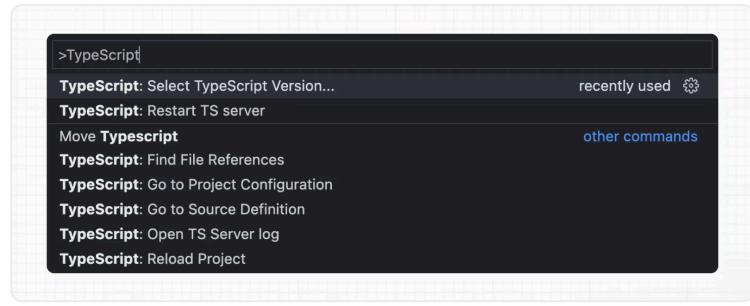
IDE Plugin

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.

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.

Watch: Learn about the built-in TypeScript plugin
→ [YouTube \(3 minutes\)](#) ↗

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

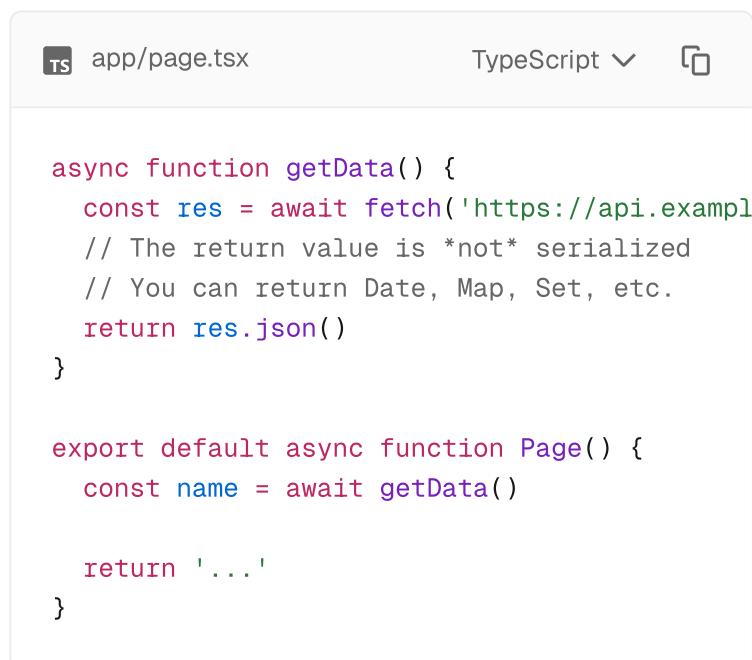
consumption in React. Instead, since `app` uses Server Components by default, we can use values like `Date`, `Map`, `Set`, and more without any extra steps. Previously, you needed to manually type the boundary between server and client with Next.js-specific types.

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:



The screenshot shows a code editor window with the following details:

- File name: `app/page.tsx`
- TypeScript version: `TypeScript v`
- Code content:

```
TS app/page.tsx TypeScript v

async function getData() {
  const res = await fetch('https://api.example.com/data')
  // 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.

Route-Aware Type Helpers

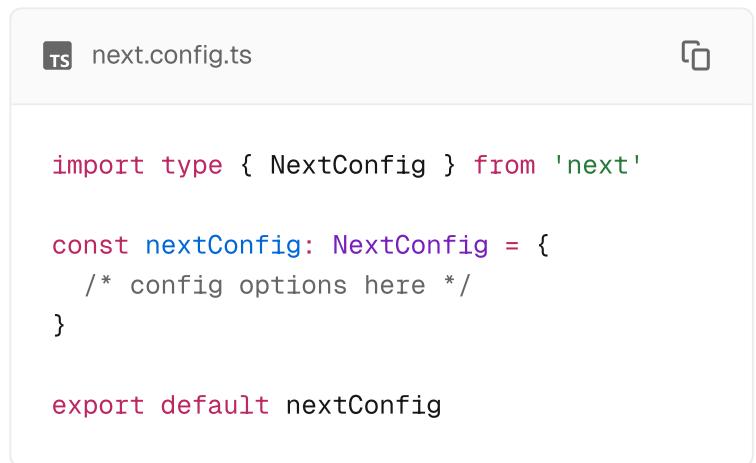
Next.js generates global helpers for App Router route types. These are available without imports and are generated during `next dev`, `next build`, or via `next typegen`:

- `PageProps`
- `LayoutProps`
- `RouteContext`

Examples

Type checking `next.config.ts`

You can use TypeScript and import types in your Next.js configuration by using `next.config.ts`.



```
TS next.config.ts

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  /* config options here */
}

export default nextConfig
```

Good to know: 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`.

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
```

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.

Works in both the Pages and App Router for the `href` prop in `next/link`. In the App Router, it also types `next/navigation` methods like `push`, `replace`, and `prefetch`. It does not type `next/router` methods in Pages Router.

Literal `href` strings are validated, while non-literal `href`s may require a cast with `as Route`.

To opt-into this feature, `typedRoutes` need to be enabled and the project needs to be using TypeScript.



```
import type { NextConfig } from 'next'

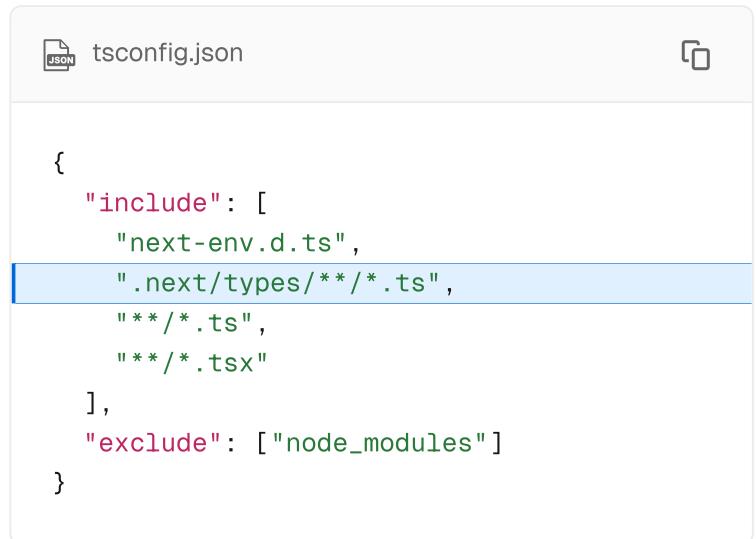
const nextConfig: NextConfig = {
  typedRoutes: true,
}

export default nextConfig
```

Next.js will generate a link definition in `.next/types` that contains information about all

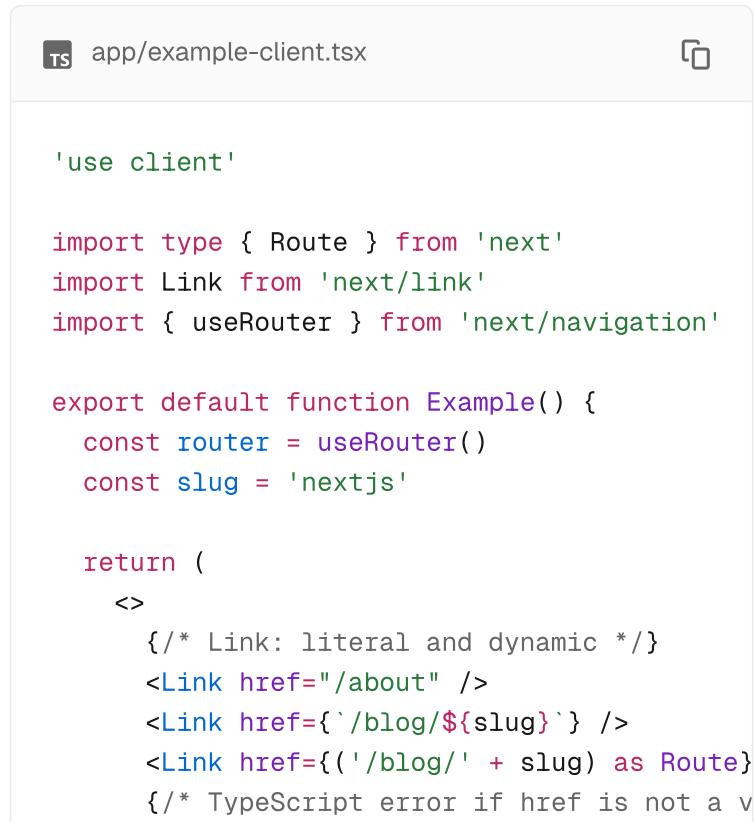
existing routes in your application, which TypeScript can then use to provide feedback in your editor about invalid links.

Good to know: If you set up your project without `create-next-app`, ensure the generated Next.js types are included by adding `.next/types/**/*.ts` to the `include` array in your `tsconfig.json`:



```
{  
  "include": [  
    "next-env.d.ts",  
    ".next/types/**/*.ts",  
    "**/*.ts",  
    "**/*.tsx"  
  ],  
  "exclude": ["node_modules"]  
}
```

Currently, support includes any string literal, including dynamic segments. For non-literal strings, you need to manually cast with `as Route`. The example below shows both `next/link` and `next/navigation` usage:



```
'use client'  
  
import type { Route } from 'next'  
import Link from 'next/link'  
import { useRouter } from 'next/navigation'  
  
export default function Example() {  
  const router = useRouter()  
  const slug = 'nextjs'  
  
  return (  
    <>  
      /* Link: literal and dynamic */  
      <Link href="/about" />  
      <Link href={`/blog/${slug}`} />  
      <Link href={{`/blog/${slug}`}} as={Route} />  
      /* TypeScript error if href is not a v
```

```
<Link href="/about" />
```

```
/* Router: literal and dynamic strings
<button onClick={() => router.push('/about')}>
  Replace Blog
</button>
<button onClick={() => router.replace('/about')}>
  Prefetch Contact
</button>
```

```
/* For non-literal strings, cast to RouterLink
<button onClick={() => router.push('/about')}>
  Push Non-literal Blog
</button>
```

```
)
```

```
}
```

The same applies for redirecting routes defined by middleware:

middleware.ts



```
import { NextRequest, NextResponse } from 'next'

export function middleware(request: NextRequest) {
  if (request.nextUrl.pathname === '/middleware-redirect') {
    return NextResponse.redirect(new URL('/', request.url))
  }

  return NextResponse.next()
}
```

app/some/page.tsx



```
import type { Route } from 'next'

export default function Page() {
  return <Link href={'/middleware-redirect'}>
    Go to middleware-redirect
  </Link>
}
```

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 }: { h
  return (
    <Link href={href}>
      <div>My Card</div>
    </Link>
  )
}
```

You can also type a simple data structure and iterate to render links:

```
components/nav-items.ts
```

```
import type { Route } from 'next'

type NavItem<T extends string = string> = {
  href: T
  label: string
}

export const navItems: NavItem<Route>[] = [
  { href: '/', label: 'Home' },
  { href: '/about', label: 'About' },
  { href: '/blog', label: 'Blog' },
]
```

Then, map over the items to render `Links`:

```
components/nav.tsx
```

```
import Link from 'next/link'
import { navItems } from './nav-items'

export function Nav() {
  return (
    <nav>
      {navItems.map((item) => (
        <Link key={item.href} href={item.href}
          {item.label}
        </Link>
      ))}
    </nav>
  )
}
```

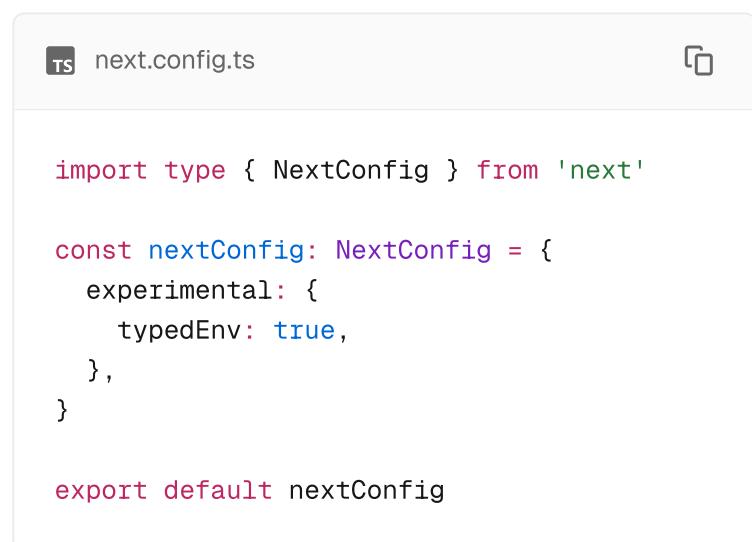
How does it work?

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.

Type IntelliSense for Environment Variables

During development, Next.js generates a `.d.ts` file in `.next/types` that contains information about the loaded environment variables for your editor's IntelliSense. If the same environment variable key is defined in multiple files, it is deduplicated according to the [Environment Variable Load Order](#).

To opt-into this feature, `experimental.typedEnv` needs to be enabled and the project needs to be using TypeScript.



```
TS next.config.ts

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    typedEnv: true,
  },
}

export default nextConfig
```

Good to know: Types are generated based on the environment variables loaded at development runtime, which excludes variables from `.env.production*` files by default. To include production-specific variables, run `next dev` with `NODE_ENV=production`.

With Async Server Components

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.

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.

Custom `tsconfig` path

In some cases, you might want to use a different TypeScript configuration for builds or tooling. To do that, set `typescript.tsconfigPath` in `next.config.ts` to point Next.js to another `tsconfig` file.



```
TS next.config.ts

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  typescript: {
    tsconfigPath: 'tsconfig.build.json',
  },
}

export default nextConfig
```

For example, switch to a different config for production builds:



A screenshot of a code editor showing a file named `next.config.ts`. The code defines a function `nextConfig` that returns a configuration object. This object includes a `typescript` field which points to either `'tsconfig.json'` or `'tsconfig.build.json'` based on the `NODE_ENV` environment variable. The code also imports `NextConfig` from the `'next'` module.

```
import type { NextConfig } from 'next'

const isProd = process.env.NODE_ENV === 'prod'

const nextConfig: NextConfig = {
  typescript: {
    tsconfigPath: isProd ? 'tsconfig.build.json' : 'tsconfig.json'
  },
}

export default nextConfig
```

► Why you might use a separate `tsconfig` for builds

Good to know:

- IDEs typically read `tsconfig.json` for diagnostics and IntelliSense, so you can still see IDE warnings while production builds use the alternate config. Mirror critical options if you want parity in the editor.
- In development, only `tsconfig.json` is watched for changes. If you edit a different file name via `typescript.tsconfigPath`, restart the dev server to apply changes.
- The configured file is used in `next dev`, `next build`, `next lint`, and `next typegen`.

Disabling TypeScript errors in production

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:

```
TS next.config.ts

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to
    // 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`:

```
JSON 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 .

Was this helpful?    

Using App Router
Features available in /app

Latest Version
15.5.4

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.

Reference

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`.

Rules

The full set of rules is as follows:

Enabled in recommended config			Rule	Description
	@next/next/google-font-display			Enforce font-display behavior with Google Fonts.

Enabled in
recommended

config

Rule

Description



@next/next/google-
font-preconnect

Ensure `preconnect`
is used with Google
Fonts.



@next/next/inline-
script-id

Enforce `id` attrib-
ute on `next/script`
components with
inline content.



@next/next/next-
script-for-ga

Prefer `next/script`
component when
using the inline sc-
ript for Google Analyti



@next/next/no-
assign-module-
variable

Prevent assignme-
nt 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 c-
pages/_document



@next/next/no-
css-tags

Prevent manual
stylesheet tags.



@next/next/no-
document-import-
in-page

Prevent importing
`next/document`
outside of
pages/_document



@next/next/no-
duplicate-head

Prevent duplicate
usage of `<Head>`
pages/_document

Enabled in recommended config	Rule	Description
✓	@next/next/no-head-element	Prevent usage of <code><head></code> element.
✓	@next/next/no-head-import-in-document	Prevent usage of <code>next/head</code> in <code>pages/_document</code>
✓	@next/next/no-html-link-for-pages	Prevent usage of <code><a></code> elements to navigate to internal Next.js pages.
✓	@next/next/no-img-element	Prevent usage of <code></code> element due to slower LCP and higher bandwidth.
✓	@next/next/no-page-custom-font	Prevent page-only custom fonts.
✓	@next/next/no-script-component-in-head	Prevent usage of <code>next/script</code> in <code>next/head</code> component.
✓	@next/next/no-styled-jsx-in-document	Prevent usage of <code>styled-jsx</code> in <code>pages/_document</code>
✓	@next/next/no-sync-scripts	Prevent synchronous scripts.
✓	@next/next/no-title-in-document-head	Prevent usage of <code><title></code> with <code>Head</code> component from <code>next/document</code> .
✓	@next/next/no-typos	Prevent common typos in Next.js's data fetching functions

Enabled in recommended config	Rule	Description
	@next/next/no-unwanted-polyfillio	Prevent duplicate polyfills from Polyfill.io.

We recommend using an appropriate [integration](#) to view warnings and errors directly in your code editor during development.

Examples

Linting custom directories and files

By default, Next.js will run ESLint for all files in the `pages/`, `app/`, `components/`, `lib/`, and `src/` directories. However, you can specify which directories using the `dirs` option in the `eslint` config in `next.config.js` for production builds:

```
JS next.config.js

module.exports = {
  eslint: {
    dirs: ['pages', 'utils'], // Only run ESL
  },
}
```

Similarly, the `--dir` and `--file` flags can be used for `next lint` to lint specific directories and files:

```
>_ Terminal

next lint --dir pages --dir utils --file bar.
```

Specifying a root directory within a monorepo

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`:

```
JS eslint.config.mjs

import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta.dirname is available after N
  baseDirectory: import.meta.dirname,
})

const eslintConfig = [
  ...compat.config({
    extends: ['next'],
    settings: {
      next: {
        rootDir: 'packages/my-app/',
      },
    },
  }),
]

export default eslintConfig
```

`rootDir` can be a path (relative or absolute), a glob (i.e. `"packages/*/"`), or an array of paths and/or globs.

Disabling the cache

To improve performance, information of files processed by ESLint are cached by default. This is stored in `.next/cache` or in your defined [build directory](#). If you include any ESLint rules that depend on more than the contents of a single

source file and need to disable the cache, use the `--no-cache` flag with `next lint`.

> Terminal



```
next lint --no-cache
```

Disabling rules

If you would like to modify or disable any rules provided by the supported plugins (`react`, `react-hooks`, `next`), you can directly change them using the `rules` property in your `.eslintrc`:

eslint.config.mjs



```
import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta.dirname is available after N
  baseDirectory: import.meta.dirname,
})

const eslintConfig = [
  ...compat.config({
    extends: ['next'],
    rules: {
      'react/no-unescaped-entities': 'off',
      '@next/next/no-page-custom-font': 'off'
    },
  }),
]

export default eslintConfig
```

With 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.

eslint.config.mjs



```
import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta.dirname is available after N
  baseDirectory: import.meta.dirname,
})

const eslintConfig = [
  ...compat.config({
    extends: ['next/core-web-vitals'],
  }),
]

export default eslintConfig
```

`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](#).

With 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:

`eslint.config.mjs`



```
import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta.dirname is available after N
  baseDirectory: import.meta.dirname,
})

const eslintConfig = [
  ...compat.config({
    extends: ['next/core-web-vitals', 'next/t'],
  }),
]
```

```
export default eslintConfig
```

Those rules are based on

[plugin:@typescript-eslint/recommended](#) . See [typescript-eslint > Configs](#)  for more details.

With Prettier

ESLint also contains code formatting rules, which can conflict with your existing [Prettier](#)  setup. We recommend including [eslint-config-prettier](#)  in your ESLint config to make ESLint and Prettier work together.

First, install the dependency:

```
>_ Terminal   
  
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:

```
eslint.config.mjs 
```

```
import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta.dirname is available after N
  baseDirectory: import.meta.dirname,
})

const eslintConfig = [
  ...compat.config({
    extends: ['next', 'prettier'],
  }),
]

export default eslintConfig
```

Running lint on staged files

If you would like to use `next lint` with `lint-staged` ↗ to run the linter on staged git files, you'll have to add the following to the `.lintstagedrc.js` file in the root of your project in order to specify usage of the `--file` flag.

```
JS .lintstagedrc.js

const path = require('path')

const buildEslintCommand = (filenames) =>
  `next lint --fix --file ${filenames
    .map((f) => path.relative(process.cwd(),
      .join(' --file ')))}

module.exports = {
  '*.{js,jsx,ts,tsx}': [buildEslintCommand],
}
```

Disabling linting during production builds

If you do not want ESLint to run during `next build`, you can set the

`eslint.ignoreDuringBuilds` option in
`next.config.js` to `true`:



The screenshot shows a code editor window with a tab labeled "next.config.ts". The file contains the following TypeScript code:

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  eslint: {
    // Warning: This allows production builds
    // your project has ESLint errors.
    ignoreDuringBuilds: true,
  },
}

export default nextConfig
```

Migrating existing config

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.

Recommended plugin ruleset

If 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`:

```
>_ Terminal   
  
npm install --save-dev @next/eslint-plugin-ne  
  
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:

```
 eslint.config.mjs 
```

```
import js from '@eslint/js'
import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta.dirname is available after N
  baseDirectory: import.meta.dirname,
  recommendedConfig: js.configs.recommended,
})

const eslintConfig = [
  ...compat.config({
    extends: ['eslint:recommended', 'next'],
  }),
]

export default eslintConfig
```

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.

Was this helpful?    



Using App Router

Features available in /app



CLI



Latest Version

15.5.4



Next.js comes with **two** Command Line Interface (CLI) tools:

- `create-next-app` : Quickly create a new Next.js application using the default template or an [example ↗](#) from a public GitHub repository.
- `next` : Run the Next.js development server, build your application, and more.

`create-next-app`

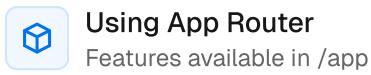
Create Next.js apps using one command with th...

`next` CLI

Learn how to run and build your application with...

Was this helpful?





create-next-app

The `create-next-app` CLI allows you to create a new Next.js application using the default template or an [example ↗](#) from a public GitHub repository. It is the easiest way to get started with Next.js.

Basic usage:

> Terminal

```
npx create-next-app@latest [project-name] [op
```

Reference

The following options are available:

Options	Description
<code>-h</code> or <code>--help</code>	Show all available options
<code>-v</code> or <code>--version</code>	Output the version number
<code>--no-*</code>	Negate default options. E.g. <code>--no-ts</code>
<code>--ts</code> or <code>--typescript</code>	Initialize as a TypeScript project (default)
<code>--js</code> or <code>--javascript</code>	Initialize as a JavaScript project
<code>--tailwind</code>	Initialize with Tailwind CSS config (default)

Options	Description
--eslint	Initialize with ESLint config
--biome	Initialize with Biome config
--no-linter	Skip linter configuration
--app	Initialize as an App Router project
--api	Initialize a project with only route handlers
--src-dir	Initialize inside a <code>src/</code> directory
--turbopack	Enable Turbopack by default for development
--import-alias <code><alias-to-configure></code>	Specify import alias to use (default "@/*")
--empty	Initialize an empty project
--use-npm	Explicitly tell the CLI to bootstrap the application using npm
--use-pnpm	Explicitly tell the CLI to bootstrap the application using pnpm
--use-yarn	Explicitly tell the CLI to bootstrap the application using Yarn
--use-bun	Explicitly tell the CLI to bootstrap the application using Bun
-e or --example [name] [github-url]	An example to bootstrap the app with
--example-path <code><path-to-example></code>	Specify the path to the example separately

Options	Description
--reset-preferences	Explicitly tell the CLI to reset any stored preferences
--skip-install	Explicitly tell the CLI to skip installing packages
--disable-git	Explicitly tell the CLI to disable git initialization
--yes	Use previous preferences or defaults for all options

Examples

With the default template

To create a new app using the default template, run the following command in your terminal:



```
npx create-next-app@latest
```

You will then be asked the following prompts:



```
What is your project named? my-app
Would you like to use TypeScript? No / Yes
Which linter would you like to use? ESLint /
Would you like to use Tailwind CSS? No / Yes
Would you like your code inside a `src/` dire
Would you like to use App Router? (recommende
Would you like to use Turbopack? (recommended
Would you like to customize the import alias
```

Linter Options

ESLint: The traditional and most popular JavaScript linter. Includes Next.js-specific rules from `eslint-plugin-next`.

Biome: A fast, modern linter and formatter that combines the functionality of ESLint and Prettier. Includes built-in Next.js and React domain support for optimal performance.

None: Skip linter configuration entirely. You can always add a linter later.

Note: `next lint` command will be deprecated in Next.js 16. New projects should use the chosen linter directly (e.g., `biome check` or `eslint`).

Once you've answered the prompts, a new project will be created with your chosen configuration.

With an official Next.js example

To create a new app using an official Next.js example, use the `--example` flag. For example:

```
>_ Terminal   
npx create-next-app@latest --example [example]
```

You can view a list of all available examples along with setup instructions in the [Next.js repository ↗](#).

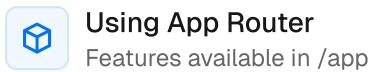
With any public GitHub example

To create a new app using any public GitHub example, use the `--example` option with the GitHub repo's URL. For example:

```
>_ Terminal   
npx create-next-app@latest --example https://github.com/nextjs/next
```

```
npx create-next-app@latest --example "https:/
```

Was this helpful?    



next CLI

The Next.js CLI allows you to develop, build, start your application, and more.

Basic usage:

```
>_ Terminal   
  
npx next [command] [options]
```

Reference

The following options are available:

Options	Description
<code>-h</code> or <code>--help</code>	Shows all available options
<code>-v</code> or <code>--version</code>	Outputs the Next.js version number

Commands

The following commands are available:

Command	Description
<code>dev</code>	Starts Next.js in development mode with Hot Module Reloading, error reporting, and more.

Command	Description
<code>build</code>	Creates an optimized production build of your application. Displaying information about each route.
<code>start</code>	Starts Next.js in production mode. The application should be compiled with <code>next build</code> first.
<code>info</code>	Prints relevant details about the current system which can be used to report Next.js bugs.
<code>lint</code>	Runs ESLint for all files in the <code>/src</code> , <code>/app</code> , <code>/pages</code> , <code>/components</code> , and <code>/lib</code> directories. It also provides a guided setup to install any required dependencies if ESLint it is not already configured in your application.
<code>telemetry</code>	Allows you to enable or disable Next.js' completely anonymous telemetry collection.
<code>typegen</code>	Generates TypeScript definitions for routes, pages, layouts, and route handlers without running a full build.

Good to know: Running `next` without a command is an alias for `next dev`.

next dev options

`next dev` starts the application in development mode with Hot Module Reloading (HMR), error reporting, and more. The following options are available when running `next dev`:

Option	Description
<code>-h, --help</code>	Show all available options.

Option	Description
[directory]	A directory in which to build the application. If not provided, current directory is used.
--turbopack	Starts development mode using Turbopack . Also available as --turbo .
-p or --port <port>	Specify a port number on which to start the application. Default: 3000, env: PORT
-H or --hostname <hostname>	Specify a hostname on which to start the application. Useful for making the application available for other devices on the network. Default: 0.0.0.0
--experimental-https	Starts the server with HTTPS and generates a self-signed certificate.
--experimental-https-key <path>	Path to a HTTPS key file.
--experimental-https-cert <path>	Path to a HTTPS certificate file.
--experimental-https-ca <path>	Path to a HTTPS certificate authority file.
--experimental-upload-trace <traceUrl>	Reports a subset of the debugging trace to a remote HTTP URL.

next build options

`next build` creates an optimized production build of your application. The output displays information about each route. For example:

```
>_ Terminal
Route (app) Size
  ↗ ◉ /_not-found 0 B
```

L f /products/[id]

0 B

- o (Static) prerendered as static content
- f (Dynamic) server-rendered on demand

- **Size:** The size of assets downloaded when navigating to the page client-side. The size for each route only includes its dependencies.
- **First Load JS:** The size of assets downloaded when visiting the page from the server. The amount of JS shared by all is shown as a separate metric.

Both of these values are [compressed with gzip](#).
The first load is indicated by green, yellow, or red.
Aim for green for performant applications.

The following options are available for the `next build` command:

Option	Description
<code>-h, --help</code>	Show all available options.
<code>[directory]</code>	A directory on which to build the application. If not provided, the current directory will be used.
<code>--turbopack</code>	Build using Turbopack (beta). Also available as <code>--turbo</code> .
<code>-d</code> or <code>--debug</code>	Enables a more verbose build output. With this flag enabled additional build output like rewrites, redirects, and headers will be shown.
<code>--profile</code>	Enables production profiling for React .
<code>--no-lint</code>	Disables linting. <i>Note: linting will be removed from <code>next build</code> in Next 16. If you're using Next 15.5+ with a</i>

Option

Description

linter other than eslint, linting during build will not occur.

--no-mangling

Disables mangling [↗](#). This may affect performance and should only be used for debugging purposes.

--experimental-app-only

Builds only App Router routes.

--experimental-build-mode
[mode]

Uses an experimental build mode.
(choices: "compile", "generate",
default: "default")

--debug-prerender

Debug prerender errors in development.

next start options

`next start` starts the application in production mode. The application should be compiled with `next build` first.

The following options are available for the `next start` command:

Option

Description

-h or --help

Show all available options.

[directory]

A directory on which to start the application. If no directory is provided, the current directory will be used.

-p or --port <port>

Specify a port number on which to start the application. (default: 3000, env: PORT)

-H or

--hostname <hostname>

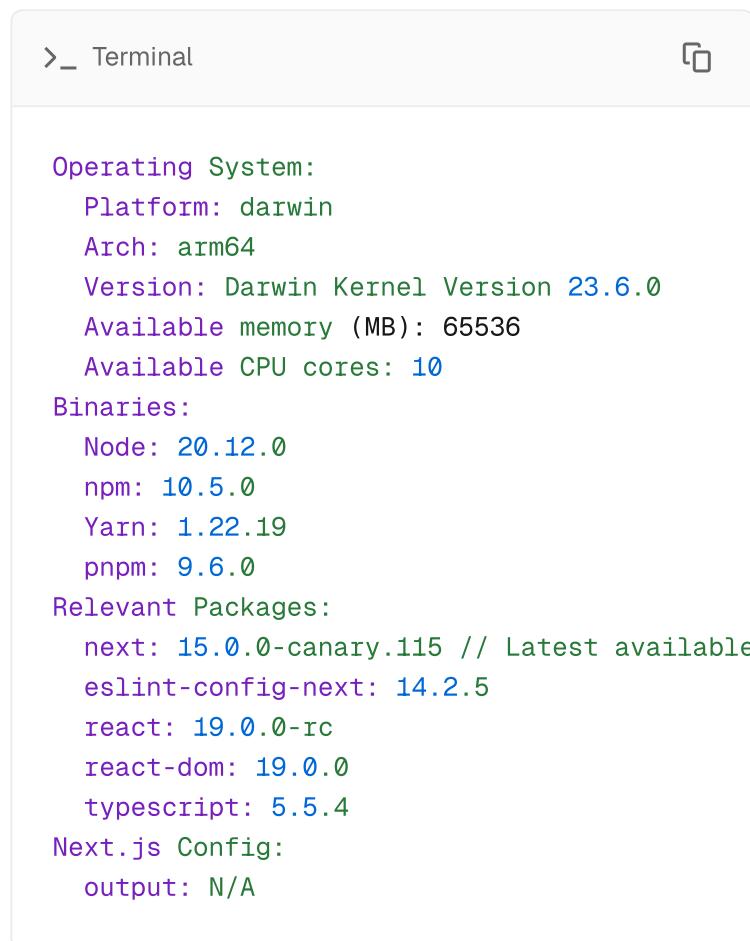
Specify a hostname on which to start the application

Option	Description
	(default: 0.0.0.0).
--keepAliveTimeout <keepAliveTimeout>	Specify the maximum amount of milliseconds to wait before closing the inactive connections.

next info options

`next info` prints relevant details about the current system which can be used to report Next.js bugs when opening a [GitHub issue](#). This information includes Operating System platform/arch/version, Binaries (Node.js, npm, Yarn, pnpm), package versions (`next`, `react`, `react-dom`), and more.

The output should look like this:



A screenshot of a terminal window titled "Terminal". The window displays the output of the `next info` command. The output is color-coded and organized into sections: Operating System, Binaries, Relevant Packages, and Next.js Config. The "Operating System" section shows Darwin Kernel Version 23.6.0. The "Binaries" section lists Node.js 20.12.0, npm 10.5.0, Yarn 1.22.19, and pnpm 9.6.0. The "Relevant Packages" section lists next 15.0.0-canary.115, eslint-config-next 14.2.5, react 19.0.0-rc, react-dom 19.0.0, and typescript 5.5.4. The "Next.js Config" section shows output N/A.

```
Operating System:
  Platform: darwin
  Arch: arm64
  Version: Darwin Kernel Version 23.6.0
  Available memory (MB): 65536
  Available CPU cores: 10
Binaries:
  Node: 20.12.0
  npm: 10.5.0
  Yarn: 1.22.19
  pnpm: 9.6.0
Relevant Packages:
  next: 15.0.0-canary.115 // Latest available
  eslint-config-next: 14.2.5
  react: 19.0.0-rc
  react-dom: 19.0.0
  typescript: 5.5.4
Next.js Config:
  output: N/A
```

The following options are available for the `next info` command:

Option	Description
<code>-h</code> or <code>--help</code>	Show all available options
<code>--verbose</code>	Collects additional information for debugging.

next lint options

Warning: This option is deprecated and will be removed in Next 16. A [codemod](#) is available to migrate to ESLint CLI.

`next lint` runs ESLint for all files in the `pages/`, `app/`, `components/`, `lib/`, and `src/` directories. It also provides a guided setup to install any required dependencies if ESLint is not already configured in your application.

The following options are available for the `next lint` command:

Option	Description
<code>[directory]</code>	A base directory on which to lint the application. If not provided, the current directory will be used.
<code>-d</code> , <code>--dir</code> , <code><dirs ... ></code>	Include directory, or directories, to run ESLint.
<code>--file</code> , <code><files ... ></code>	Include file, or files, to run ESLint.
<code>--ext</code> , <code>[exts ...]</code>	Specify JavaScript file extensions. (default: <code>[".js", ".mjs", ".cjs", ".jsx", ".ts", ".mts", ".cts", ".tsx"]</code>)

Option	Description
<code>-c, --config, <config></code>	Uses this configuration file, overriding all other configuration options.
<code>--resolve-plugins-relative-to, <rprt></code>	Specify a directory where plugins should be resolved from.
<code>--strict</code>	Creates a <code>.eslintrc.json</code> file using the Next.js strict configuration.
<code>--rulesdir, <rulesdir...></code>	Uses additional rules from this directory(s).
<code>--fix</code>	Automatically fix linting issues.
<code>--fix-type <fixType></code>	Specify the types of fixes to apply (e.g., problem, suggestion, layout).
<code>--ignore-path <path></code>	Specify a file to ignore.
<code>--no-ignore <path></code>	Disables the <code>--ignore-path</code> option.
<code>--quiet</code>	Reports errors only.
<code>--max-warnings [maxWarnings]</code>	Specify the number of warnings before triggering a non-zero exit code. (default: -1)
<code>-o, --output-file, <outputFile></code>	Specify a file to write report to.
<code>-f, --format, <format></code>	Uses a specific output format.
<code>--no-inline-config</code>	Prevents comments from changing config or rules.
<code>--report-unused-disable-directives-severity <level></code>	Specify severity level for unused eslint-disable

Option	Description
	directives. (choices: "error", "off", "warn")
--no-cache	Disables caching.
--cache-location, <cacheLocation>	Specify a location for cache.
--cache-strategy, [cacheStrategy]	Specify a strategy to use for detecting changed files in the cache. (default: "metadata")
--error-on-unmatched-pattern	Reports errors when any file patterns are unmatched.
-h, --help	Displays this message.

next telemetry options

Next.js collects **completely anonymous** telemetry data about general usage. Participation in this anonymous program is optional, and you can opt-out if you prefer not to share information.

The following options are available for the `next telemetry` command:

Option	Description
-h, --help	Show all available options.
--enable	Enables Next.js' telemetry collection.
--disable	Disables Next.js' telemetry collection.

Learn more about [Telemetry](#).

next typegen Options

`next typegen` generates TypeScript definitions for your application's routes without performing a

full build. This is useful for IDE autocomplete and CI type-checking of route usage.

Previously, route types were only generated during `next dev` or `next build`, which meant running `tsc --noEmit` directly wouldn't validate your route types. Now you can generate types independently and validate them externally:

>_ Terminal

```
# Generate route types first, then validate w
next typegen && tsc --noEmit

# Or in CI workflows for type checking without
next typegen && npm run type-check
```

The following options are available for the `next typegen` command:

Option	Description
<code>-h, --help</code>	Show all available options.
<code>[directory]</code>	A directory on which to generate types. If not provided, the current directory will be used.

Output files are written to `<distDir>/types` (default: `.next/types`):

>_ Terminal

```
next typegen
# or for a specific app
next typegen ./apps/web
```

Good to know: `next typegen` loads your Next.js config (`next.config.js`, `next.config.mjs`, or `next.config.ts`) using the production build phase.

Ensure any required environment variables and dependencies are available so the config can load correctly.

Examples

Debugging prerender errors

If you encounter prerendering errors during `next build`, you can pass the `--debug-prerender` flag to get more detailed output:



A screenshot of a terminal window titled "Terminal". Inside the terminal, the command `next build --debug-prerender` is typed in green text. The terminal interface includes a close button in the top right corner.

This enables several experimental options to make debugging easier:

- Disables server code minification:
 - `experimental.serverMinification = false`
 - `experimental.turbopackMinify = false`
- Generates source maps for server bundles:
 - `experimental.serverSourceMaps = true`
- Enables source map consumption in child processes used for prerendering:
 - `experimental.enablePrerenderSourceMaps = true`
- Continues building even after the first prerender error, so you can see all issues at once:
 - `experimental.prerenderEarlyExit = false`

This helps surface more readable stack traces and code frames in the build output.

Warning: `--debug-prerender` is for debugging in development only. Do not deploy builds generated with `--debug-prerender` to production, as it may impact performance.

Changing the default port

By default, Next.js uses `http://localhost:3000` during development and with `next start`. The default port can be changed with the `-p` option, like so:

```
>_ Terminal
```

```
next dev -p 4000
```

Or using the `PORT` environment variable:

```
>_ Terminal
```

```
PORT=4000 next dev
```

Good to know: `PORT` cannot be set in `.env` as booting up the HTTP server happens before any other code is initialized.

Using HTTPS during development

For certain use cases like webhooks or authentication, you can use [HTTPS ↗](#) to have a secure environment on `localhost`. Next.js can generate a self-signed certificate with `next dev` using the `--experimental-https` flag:

```
>_ Terminal
```

```
next dev --experimental-https
```

With the generated certificate, the Next.js development server will exist at

`https://localhost:3000`. The default port `3000` is used unless a port is specified with `-p`, `--port`, or `PORT`.

You can also provide a custom certificate and key with `--experimental-https-key` and `--experimental-https-cert`. Optionally, you can provide a custom CA certificate with `--experimental-https-ca` as well.

> Terminal



```
next dev --experimental-https --experimental-
```

`next dev --experimental-https` is only intended for development and creates a locally trusted certificate with `mkcert` [↗]. In production, use properly issued certificates from trusted authorities.

Configuring a timeout for downstream proxies

When deploying Next.js behind a downstream proxy (e.g. a load-balancer like AWS ELB/ALB), it's important to configure Next's underlying HTTP server with [keep-alive timeouts](#) [↗] that are *larger* than the downstream proxy's timeouts. Otherwise, once a keep-alive timeout is reached for a given TCP connection, Node.js will immediately terminate that connection without notifying the downstream proxy. This results in a proxy error whenever it attempts to reuse a connection that Node.js has already terminated.

To configure the timeout values for the production Next.js server, pass `--keepAliveTimeout` (in milliseconds) to `next start`, like so:

```
>_ Terminal   
  
next start --keepAliveTimeout 70000
```

Passing Node.js arguments

You can pass any [node arguments](#) to `next` commands. For example:

```
>_ Terminal   
  
NODE_OPTIONS='--throw-deprecation' next  
NODE_OPTIONS='-r esm' next  
NODE_OPTIONS='--inspect' next
```

Version Changes

v15.5.0 Add the `next typegen` command

v15.4.0 Add `--debug-prerender` option for `next build` to help debug prerender errors.

Was this helpful?    



Using App Router
Features available in /app



Edge Runtime



Latest Version
15.5.4



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 is used for rendering your application.
- The **Edge Runtime** which contains a more limited [set of APIs](#), used in [Middleware](#).

Caveats

- The Edge Runtime does not support all Node.js APIs. Some packages may not work as expected.
- The Edge Runtime does not support Incremental Static Regeneration (ISR).
- Both runtimes can support [streaming](#) depending on your deployment adapter.

Reference

The Edge Runtime supports the following APIs:

Network APIs

API	Description
Blob	Represents a blob
fetch	Fetches a resource
FetchEvent	Represents a fetch event
File	Represents a file
FormData	Represents form data
Headers	Represents HTTP headers
Request	Represents an HTTP request
Response	Represents an HTTP response
URLSearchParams	Represents URL search parameters
WebSocket	Represents a websocket connection

Encoding APIs

API	Description
atob	Decodes a base-64 encoded string
btoa	Encodes a string in base-64
TextDecoder	Decodes a Uint8Array into a string
TextDecoderStream	Chainable decoder for streams
TextEncoder	Encodes a string into a Uint8Array
TextEncoderStream	Chainable encoder for streams

Stream APIs

API	Description
ReadableStream ↗	Represents a readable stream
ReadableStreamBYOBReader ↗	Represents a reader of a ReadableStream
ReadableStreamDefaultReader ↗	Represents a reader of a ReadableStream
TransformStream ↗	Represents a transform stream
WritableStream ↗	Represents a writable stream
WritableStreamDefaultWriter ↗	Represents a writer of a WritableStream

Crypto APIs

API	Description
crypto ↗	Provides access to the cryptographic functionality of the platform
CryptoKey ↗	Represents a cryptographic key
SubtleCrypto ↗	Provides access to common cryptographic primitives, like hashing, signing, encryption or decryption

Web Standard APIs

API	Description
AbortController ↗	Allows you to abort one or more DOM requests as and when desired
Array ↗	Represents an array of values
ArrayBuffer ↗	Represents a generic, fixed-length raw binary data buffer
Atomics ↗	Provides atomic operations as static methods
BigInt ↗	Represents a whole number with arbitrary precision
BigInt64Array ↗	Represents a typed array of 64-bit signed integers
BigUint64Array ↗	Represents a typed array of 64-bit unsigned integers
Boolean ↗	Represents a logical entity and can have two values: <code>true</code> and <code>false</code>
clearInterval ↗	Cancels a timed, repeating action which was previously established by a call to <code>setInterval()</code>
clearTimeout ↗	Cancels a timed, repeating action which was previously established by a call to <code>setTimeout()</code>
console ↗	Provides access to the browser's debugging console
DataView ↗	Represents a generic view of an ArrayBuffer
Date ↗	Represents a single moment in time in a platform-independent format
decodeURI ↗	Decodes a Uniform Resource Identifier (URI) previously created

API

Description

by `encodeURI` or by a similar routine

`decodeURIComponent` ↗
Decodes a Uniform Resource Identifier (URI) component previously created by `encodeURIComponent` or by a similar routine

`DOMException` ↗
Represents an error that occurs in the DOM

`encodeURI` ↗
Encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character

`encodeURIComponent` ↗
Encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character

`Error` ↗
Represents an error when trying to execute a statement or accessing a property

`EvalError` ↗
Represents an error that occurs regarding the global function `eval()`

`Float32Array` ↗
Represents a typed array of 32-bit floating point numbers

`Float64Array` ↗
Represents a typed array of 64-bit floating point numbers

`Function` ↗
Represents a function

`Infinity` ↗
Represents the mathematical Infinity value

API	Description
Int8Array	Represents a typed array of 8-bit signed integers
Int16Array	Represents a typed array of 16-bit signed integers
Int32Array	Represents a typed array of 32-bit signed integers
Intl	Provides access to internationalization and localization functionality
isFinite	Determines whether a value is a finite number
isNaN	Determines whether a value is NaN or not
JSON	Provides functionality to convert JavaScript values to and from the JSON format
Map	Represents a collection of values, where each value may occur only once
Math	Provides access to mathematical functions and constants
Number	Represents a numeric value
Object	Represents the object that is the base of all JavaScript objects
parseFloat	Parses a string argument and returns a floating point number
parseInt	Parses a string argument and returns an integer of the specified radix
Promise	Represents the eventual completion (or failure) of an asynchronous operation, and its resulting value

API	Description
Proxy	Represents an object that is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc)
queueMicrotask	Queues a microtask to be executed
RangeError	Represents an error when a value is not in the set or range of allowed values
ReferenceError	Represents an error when a non-existent variable is referenced
Reflect	Provides methods for interceptable JavaScript operations
RegExp	Represents a regular expression, allowing you to match combinations of characters
Set	Represents a collection of values, where each value may occur only once
setInterval	Repeatedly calls a function, with a fixed time delay between each call
setTimeout	Calls a function or evaluates an expression after a specified number of milliseconds
SharedArrayBuffer	Represents a generic, fixed-length raw binary data buffer
String	Represents a sequence of characters
structuredClone	Creates a deep copy of a value
Symbol	Represents a unique and immutable data type that is used

API	Description
	as the key of an object property
SyntaxError ↗	Represents an error when trying to interpret syntactically invalid code
TypeError ↗	Represents an error when a value is not of the expected type
Uint8Array ↗	Represents a typed array of 8-bit unsigned integers
Uint8ClampedArray ↗	Represents a typed array of 8-bit unsigned integers clamped to 0–255
Uint32Array ↗	Represents a typed array of 32-bit unsigned integers
URIError ↗	Represents an error when a global URI handling function was used in a wrong way
URL ↗	Represents an object providing static methods used for creating object URLs
URLPattern ↗	Represents a URL pattern
URLSearchParams ↗	Represents a collection of key/value pairs
WeakMap ↗	Represents a collection of key/value pairs in which the keys are weakly referenced
WeakSet ↗	Represents a collection of objects in which each object may occur only once
WebAssembly ↗	Provides access to WebAssembly

Next.js Specific Polyfills

- [AsyncLocalStorage](#) ↗

Environment Variables

You can use `process.env` to access Environment Variables for both `next dev` and `next build`.

Unsupported APIs

The Edge Runtime has some restrictions including:

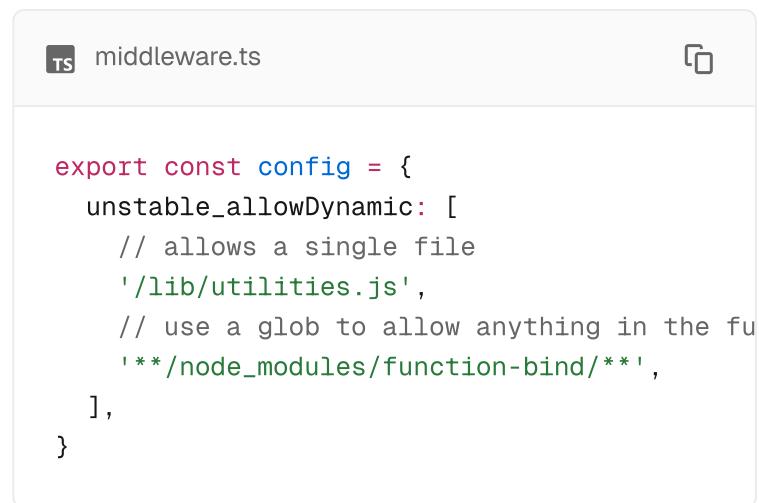
- Native Node.js APIs **are not supported**. For example, you can't read or write to the filesystem.
- `node_modules` *can* be used, as long as they implement ES Modules and do not use native Node.js APIs.
- Calling `require` directly is **not allowed**. Use ES Modules instead.

The following JavaScript language features are disabled, and **will not work**:

API	Description
<code>eval</code> ↗	Evaluates JavaScript code represented as a string
<code>new Function(evalString)</code> ↗	Creates a new function with the code provided as an argument
<code>WebAssembly.compile</code> ↗	Compiles a WebAssembly module from a buffer source
<code>WebAssembly.instantiate</code> ↗	Compiles and instantiates a WebAssembly module from a buffer source

In rare cases, your code could contain (or import) some dynamic code evaluation statements which *can not be reached at runtime* and which can not

be removed by treeshaking. You can relax the check to allow specific files with your Middleware configuration:



A screenshot of a code editor showing a file named "middleware.ts". The code contains a single export statement defining a constant "config" with a property "unstable_allowDynamic" set to an array of globs. The globs include a single file path and a glob pattern for node modules.

```
export const config = {
  unstable_allowDynamic: [
    // allows a single file
    '/lib/utilities.js',
    // use a glob to allow anything in the function-bind directory
    '**/node_modules/function-bind/**',
  ],
}
```

`unstable_allowDynamic` is a [glob ↗](#), or an array of globs, ignoring dynamic code evaluation for specific files. The globs are relative to your application root folder.

Be warned that if these statements are executed on the Edge, *they will throw and cause a runtime error.*

Was this helpful?    



Using App Router
Features available in /app



Turbopack



Latest Version
15.5.4



Turbopack is an **incremental bundler** optimized for JavaScript and TypeScript, written in Rust, and built into **Next.js**. You can use Turbopack with both the Pages and App Router for a **much faster** local development experience.

Why Turbopack?

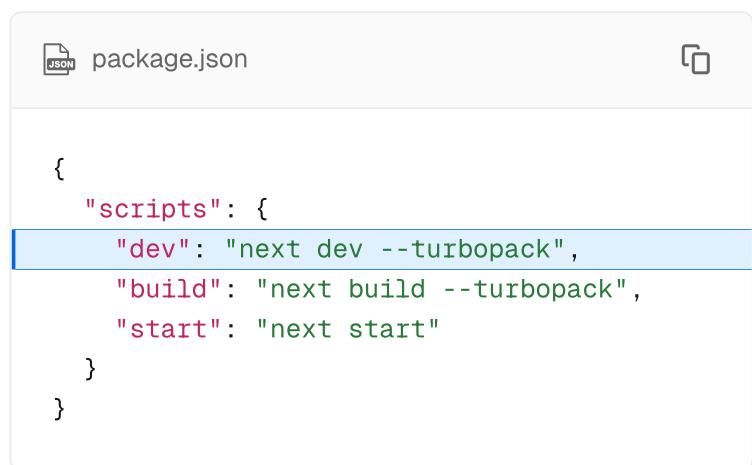
We built Turbopack to push the performance of Next.js, including:

- **Unified Graph:** Next.js supports multiple output environments (e.g., client and server). Managing multiple compilers and stitching bundles together can be tedious. Turbopack uses a **single, unified graph** for all environments.
- **Bundling vs Native ESM:** Some tools skip bundling in development and rely on the browser's native ESM. This works well for small apps but can slow down large apps due to excessive network requests. Turbopack **bundles** in dev, but in an optimized way to keep large apps fast.
- **Incremental Computation:** Turbopack parallelizes work across cores and **caches** results down to the function level. Once a piece of work is done, Turbopack won't repeat it.

- **Lazy Bundling:** Turbopack only bundles what is actually requested by the dev server. This lazy approach can reduce initial compile times and memory usage.
-

Getting started

To enable Turbopack in your Next.js project, add the `--turbopack` flag to the `dev` and `build` scripts in your `package.json` file:



```
JSON package.json
```

```
{  
  "scripts": {  
    "dev": "next dev --turbopack",  
    "build": "next build --turbopack",  
    "start": "next start"  
  }  
}
```

Currently, Turbopack for `dev` is stable, while `build` is in beta. We are actively working on production support as Turbopack moves closer to stability.

Supported features

Turbopack in Next.js has **zero-configuration** for the common use cases. Below is a summary of what is supported out of the box, plus some references to how you can configure Turbopack further when needed.

Language features

Feature	Status	Notes
JavaScript & TypeScript	Supported	Uses SWC under the hood. Type-checking is not done by Turbopack (run <code>tsc --watch</code> or rely on your IDE for type checks).
ECMAScript (ESNext)	Supported	Turbopack supports the latest ECMAScript features, matching SWC's coverage.
CommonJS	Supported	<code>require()</code> syntax is handled out of the box.
ESM	Supported	Static and dynamic <code>import</code> are fully supported.
Babel	Partially Unsupported	Turbopack does not include Babel by default. However, you can configure <code>babel-loader</code> via the Turbopack config .

Framework and React features

Feature	Status	Notes
JSX / TSX	Supported	SWC handles JSX/TSX compilation.
Fast Refresh	Supported	No configuration needed.
React Server Components (RSC)	Supported	For the Next.js App Router. Turbopack ensures correct server/client bundling.

Feature	Status	Notes
Root layout creation	Unsupported	Automatic creation of a root layout in App Router is not supported. Turbopack will instruct you to create it manually.

CSS and styling

Feature	Status	Notes
Global CSS	Supported	Import <code>.css</code> files directly in your application.
CSS Modules	Supported	<code>.module.css</code> files work natively (Lightning CSS).
CSS Nesting	Supported	Lightning CSS supports modern CSS nesting .
@import syntax	Supported	Combine multiple CSS files.
PostCSS	Supported	Automatically processes <code>postcss.config.js</code> in a Node.js worker pool. Useful for Tailwind, Autoprefixer, etc.
Sass / SCSS (Next.js)	Supported	For Next.js, Sass is supported out of the box. In the future, Turbopack standalone usage will likely require a loader config.
Less	Planned via plugins	Not yet supported by default. Will likely require a loader config once custom loaders are stable.
Lightning CSS	In Use	Handles CSS transformations. Some low-usage CSS Modules features (like <code>:local/:global</code>) as standalone pseudo-classes

Feature	Status	Notes
		are not yet supported. See below for more details.

Assets

Feature	Status	Notes
Static Assets (images, fonts)	Supported	<p>Importing <code>import img from './img.png'</code> works out of the box. In Next.js, returns an object for the <code><Image /></code> component.</p>
JSON Imports	Supported	Named or default imports from <code>.json</code> are supported.

Module resolution

Feature	Status	Notes
Path Aliases	Supported	Reads <code>tsconfig.json</code> 's <code>paths</code> and <code>baseUrl</code> , matching Next.js behavior.
Manual Aliases	Supported	Configure <code>resolveAlias</code> in <code>next.config.js</code> (similar to <code>webpack.resolve.alias</code>).
Custom Extensions	Supported	Configure <code>resolveExtensions</code> in <code>next.config.js</code> .
AMD	Partially Supported	Basic transforms work; advanced AMD usage is limited.

Performance and Fast Refresh

Feature	Status	Notes
Fast Refresh	Supported	Updates JavaScript, TypeScript, and CSS without a full refresh.
Incremental Bundling	Supported	Turbopack lazily builds only what's requested by the dev server, speeding up large apps.

Known gaps with webpack

There are a number of non-trivial behavior differences between webpack and Turbopack that are important to be aware of when migrating an application. Generally, these are less of a concern for new applications.

CSS Module Ordering

Turbopack will follow JS import order to order [CSS modules](#) which are not otherwise ordered. For example:



```
components/BlogPost.jsx
```

```
import utilStyles from './utils.module.css'
import buttonStyles from './button.module.css'
export default function BlogPost() {
  return (
    <div className={utilStyles.container}>
      <button className={buttonStyles.primary}>
    </div>
  )
}
```

In this example, Turbopack will ensure that `utils.module.css` will appear before

`button.module.css` in the produced CSS chunk,
following the import order

Webpack generally does this as well, but there are cases where it will ignore JS inferred ordering, for example if it infers the JS file is side-effect-free.

This can lead to subtle rendering changes when adopting Turbopack, if applications have come to rely on an arbitrary ordering. Generally, the solution is easy, e.g. have `button.module.css` `@import utils.module.css` to force the ordering, or identify the conflicting rules and change them to not target the same properties.

Bundle Sizes

Turbopack does not yet have an equivalent to the [Inner Graph Optimization ↗](#) in webpack. This optimization is useful to tree shake large modules. For example:

```
import heavy from 'some-heavy-dependency.js'

export function usesHeavy() {
    return heavy.run()
}

export const CONSTANT_VALUE = 3
```

If an application only uses `CONSTANT_VALUE` Turbopack will detect this and delete the `usesHeavy` export but not the corresponding `import`. However, with the `optimization.innerGraph = true` option enabled, webpack can delete the `import` too.

We are planning to offer an equivalent to the `innerGraph` optimization in Turbopack but it is still under development. If you are affected by this gap, consider manually splitting these modules.

Build Caching

Webpack supports [disk build caching ↗](#) to speed up builds. We are planning to support an analogous feature in Turbopack but it is not ready yet. On the `next@canary` release you can experiment with our solution by enabling the `experimental.turbopackPersistentCaching` flag.

Good to know: For this reason, when comparing webpack and Turbopack performance, make sure to delete the `.next` folder between builds to see a fair comparison.

Webpack plugins

Turbopack does not support webpack plugins. This affects third-party tools that rely on webpack's plugin system for integration. We do support [webpack loaders](#). If you depend on webpack plugins, you'll need to find Turbopack-compatible alternatives or continue using webpack until equivalent functionality is available.

Unsupported and unplanned features

Some features are not yet implemented or not planned:

- **Legacy CSS Modules features**
 - Standalone `:local` and `:global` pseudo-classes (only the function variant `:global(...)` is supported).
 - The `@value` rule (superseded by CSS variables).

- `:import` and `:export` ICSS rules.
- `composes` in `.module.css` composing a `.css` file. In webpack this would treat the `.css` file as a CSS Module, with Turbopack the `.css` file will always be global. This means that if you want to use `composes` in a CSS Module, you need to change the `.css` file to a `.module.css` file.
- `@import` in CSS Modules importing `.css` as a CSS Module. In webpack this would treat the `.css` file as a CSS Module, with Turbopack the `.css` file will always be global. This means that if you want to use `@import` in a CSS Module, you need to change the `.css` file to a `.module.css` file.
- `webpack()` configuration in `next.config.js` Turbopack replaces webpack, so `webpack()` configs are not recognized. Use the `turbopack config` instead.
- **AMP** Not planned for Turbopack support in Next.js.
- **Yarn PnP** Not planned for Turbopack support in Next.js.
- `experimental.urlImports` Not planned for Turbopack.
- `experimental.esmExternals` Not planned. Turbopack does not support the legacy `esmExternals` configuration in Next.js.
- **Some Next.js Experimental Flags**
 - `experimental.nextScriptWorkers`
 - `experimental.sri.algorithm`
 - `experimental.fallbackNodePolyfills` We plan to implement these in the future.

For a full, detailed breakdown of each feature flag and its status, see the [Turbopack API Reference](#).

Configuration

Turbopack can be configured via `next.config.js` (or `next.config.ts`) under the `turbopack` key. Configuration options include:

- `rules` Define additional [webpack loaders](#) for file transformations.
- `resolveAlias` Create manual aliases (like `resolve.alias` in webpack).
- `resolveExtensions` Change or extend file extensions for module resolution.
- `moduleIds` Set how module IDs are generated (`'named'` vs `'deterministic'`).
- `memoryLimit` Set a memory limit (in bytes) for Turbopack.

```
js next.config.js
```

```
module.exports = {
  turbopack: {
    // Example: adding an alias and custom fi
    resolveAlias: {
      underscore: 'lodash',
    },
    resolveExtensions: ['.mdx', '.tsx', '.ts'
  },
}
```

For more in-depth configuration examples, see the [Turbopack config documentation](#).

Generating trace files for performance debugging

If you encounter performance or memory issues and want to help the Next.js team diagnose them, you can generate a trace file by appending `NEXT_TURBOPACK_TRACING=1` to your dev command:

```
NEXT_TURBOPACK_TRACING=1 next dev --turbopack
```

This will produce a `.next/trace-turbopack` file. Include that file when creating a GitHub issue on the [Next.js repo ↗](#) to help us investigate.

Summary

Turbopack is a **Rust-based, incremental** bundler designed to make local development and builds fast—especially for large applications. It is integrated into Next.js, offering zero-config CSS, React, and TypeScript support.

Stay tuned for more updates as we continue to improve Turbopack and add production build support. In the meantime, give it a try with `next dev --turbopack` and let us know your feedback.

Version Changes

Version	Changes
v15.5.0	Turbopack support for <code>build beta</code>
v15.3.0	Experimental support for <code>build</code>
v15.0.0	Turbopack for <code>dev stable</code>

Was this helpful?  



Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

Latest Version

15.5.4



Getting Started - Pages Router

Installation

How to create a new Next.js application with...

Project Struc...

Learn about the folder and file conventions in a...

Images

Optimize your images with the built-in...

Fonts

Learn how to use fonts in Next.js

CSS

Learn about the different ways to add CSS to your...

Deploying

Learn how to deploy your Next.js application.

Was this helpful?





(i) You are currently viewing the documentation for Pages Router.

Create a new Next.js application

System requirements

Before you begin, make sure your system meets the following requirements:

- [Node.js 18.18 ↗](#) or later.
- macOS, Windows (including WSL), or Linux.

Automatic installation

The quickest way to create a new Next.js app is using [create-next-app](#), which sets up everything automatically for you. To create a project, run:

```
>_ Terminal   
  
npx create-next-app@latest
```

On installation, you'll see the following prompts:

```
>_ Terminal   
  
What is your project named? my-app  
Would you like to use TypeScript? No / Yes
```

```
Would you like to use ESLint? No / Yes  
Would you like to use Tailwind CSS? No / Yes  
Would you like your code inside a `src/` dire  
Would you like to use App Router? (recommende  
Would you like to use Turbopack? (recommended  
Would you like to customize the import alias  
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.

Manual installation

To manually create a new Next.js app, install the required packages:

```
pnpm   npm   yarn   bun  
>_ Terminal  ⌂  
  
pnpm i next@latest react@latest react-dom@lat
```

Good to know: The App Router uses [React canary releases](#) ↗ built-in, which include all the stable React 19 changes, as well as newer features being validated in frameworks. The Pages Router uses the React version you install in `package.json`.

Then, add the following scripts to your `package.json` file:

```
package.json  ⌂  
  
{  
  "scripts": {  
    "dev": "next dev --turbopack",  
    "build": "next build",  
    "start": "next start",  
    "lint": "eslint"  
  }  
}
```

```
}
```

These scripts refer to the different stages of developing an application:

- `next dev --turbopack` : Starts the development server using Turbopack.
- `next build` : Builds the application for production.
- `next start` : Starts the production server.
- `eslint` : Runs ESLint.

Turbopack is stable for `dev`. For production builds, Turbopack is in beta. To try it, run `next build --turbopack`. See the [Turbopack docs](#) for status and caveats.

Create the `pages` directory

Next.js uses file-system routing, which means the routes in your application are determined by how you structure your files.

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 (`/`):



```
TS pages/index.tsx TypeScript ▾ ⌂
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](#).

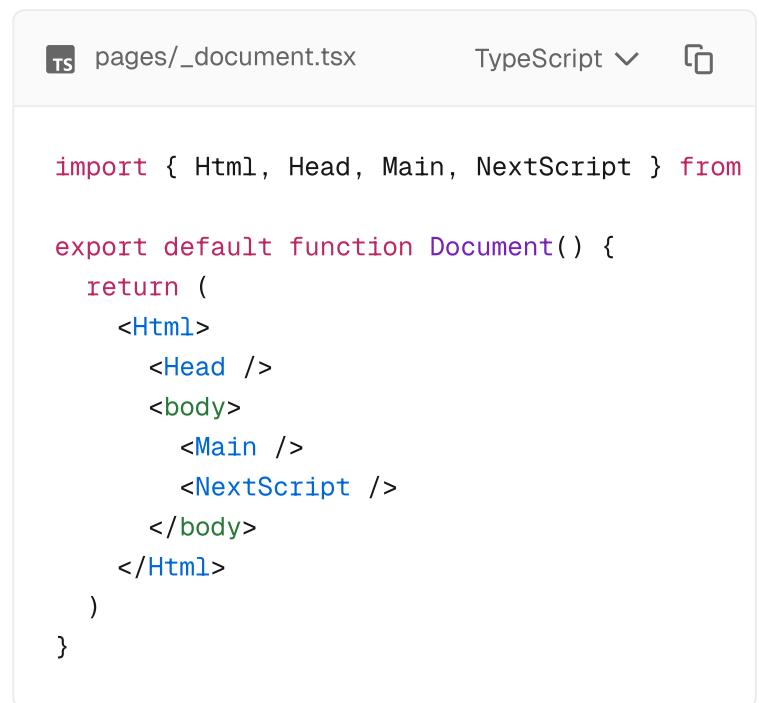


```
TS pages/_app.tsx TypeScript ▾ ⌂
```

```
import type { AppProps } from 'next/app'

export default function App({ Component, page
    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](#).



```
pages/_document.tsx TypeScript ▾ ⌂

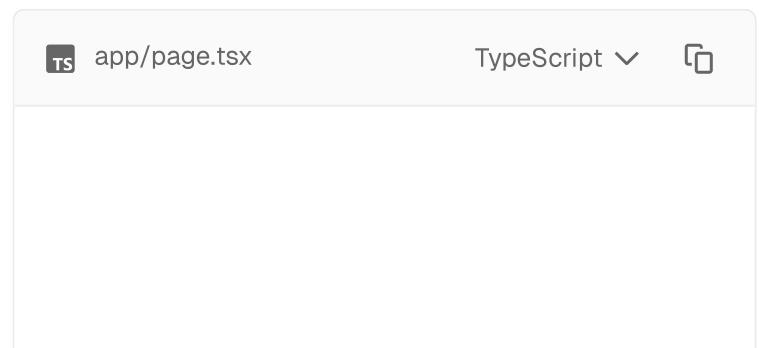
import { Html, Head, Main, NextScript } from 'next/document'

export default function Document() {
  return (
    <Html>
      <Head />
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

Create the `public` folder (optional)

Create a `public` folder at the root of your project to store static assets such as images, fonts, etc. Files inside `public` can then be referenced by your code starting from the base URL (`/`).

You can then reference these assets using the root path (`/`). For example, `public/profile.png` can be referenced as `/profile.png`:



```
app/page.tsx TypeScript ▾ ⌂
```

```
import Image from 'next/image'

export default function Page() {
  return <Image src="/profile.png" alt="Profile picture" />
}
```

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 the `pages/index.tsx` file and save it to see the updated result in your browser.

Set up TypeScript

Minimum TypeScript version: v4.5.2

Next.js comes with built-in TypeScript support. To add TypeScript to your project, rename a file to `.ts` / `.tsx` and run `next dev`. Next.js will automatically install the necessary dependencies and add a `tsconfig.json` file with the recommended config options.

See the [TypeScript reference](#) page for more information.

Set up ESLint

Next.js comes with built-in ESLint. It automatically installs the necessary packages and configures the proper settings when you create a new project with `create-next-app`.

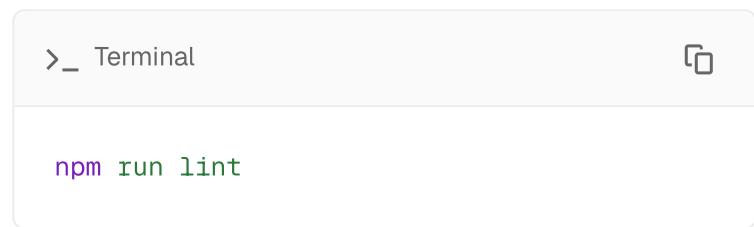
To manually add ESLint to an existing project, add `next lint` as a script to `package.json`:



```
JSON package.json
```

```
{  
  "scripts": {  
    "lint": "next lint"  
  }  
}
```

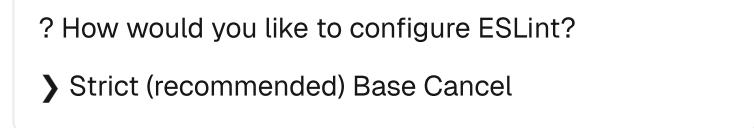
Then, run `npm run lint` and you will be guided through the installation and configuration process.



```
Terminal
```

```
npm run lint
```

You'll see a prompt like this:



```
? How would you like to configure ESLint?  
❯ Strict (recommended) Base Cancel
```

- **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.
- **Base:** Includes Next.js' base ESLint configuration.
- **Cancel:** Skip configuration. Select this option if you plan on setting up your own custom ESLint configuration.

If `Strict` or `Base` are selected, Next.js will automatically install `eslint` and `eslint-config-next` as dependencies in your application and create a configuration file in the root of your project.

The ESLint config generated by `next lint` uses the older `.eslintrc.json` format. ESLint supports both [the legacy `.eslintrc.json`](#) and [the newer `eslint.config.mjs` format ↗](#).

You can manually replace `.eslintrc.json` with an `eslint.config.mjs` file using the setup recommended in our [ESLint API reference](#), and installing the [`@eslint/eslintrc` ↗](#) package. This more closely matches the ESLint setup used by `create-next-app`.

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.

See the [ESLint Plugin](#) page for more information.

Set up Absolute Imports and Module Path Aliases

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 and cleaner to import modules. For example:

```
// Before
```

```
import { Button } from '../../../../../components/b
```

```
// After
```

```
import { Button } from '@/components/button'
```

To configure absolute imports, add the `baseUrl` configuration option to your `tsconfig.json` or `jsconfig.json` file. For example:

 tsconfig.json or jsconfig.json

```
{
  "compilerOptions": {
    "baseUrl": "src/"
  }
}
```

In addition to configuring the `baseUrl` path, you can use the `"paths"` option to `"alias"` module paths.

For example, the following configuration maps `@/components/*` to `components/*`:

 tsconfig.json or jsconfig.json

```
{
  "compilerOptions": {
    "baseUrl": "src/",
    "paths": {
      "@/styles/*": ["styles/*"],
      "@/components/*": ["components/*"]
    }
  }
}
```

Each of the `"paths"` are relative to the `baseUrl` location.

Was this helpful?



 Using Pages Router
Features available in /pages

 Latest Version
15.5.4

(i) You are currently viewing the documentation for Pages Router.

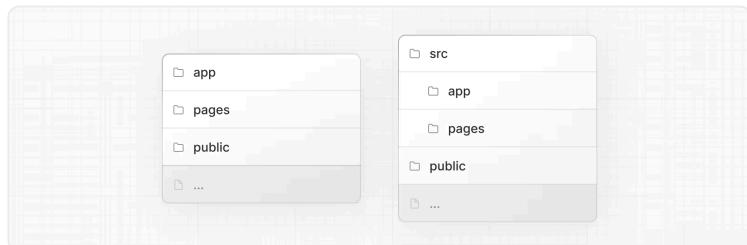
Project Structure and Organization

This page provides an overview of **all** the folder and file conventions in Next.js, and recommendations for organizing your project.

Folder and file conventions

Top-level folders

Top-level folders are used to organize your application's code and static assets.



 App Router

 Pages Router

 Static assets to be served

 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

<code>next.config.js</code>	Configuration file for Next.js
<code>package.json</code>	Project dependencies and scripts
<code>instrumentation.ts</code>	OpenTelemetry and Instrumentation file
<code>middleware.ts</code>	Next.js request middleware
<code>.env</code>	Environment variables
<code>.env.local</code>	Local environment variables
<code>.env.production</code>	Production environment variables
<code>.env.development</code>	Development environment variables
<code>.eslintrc.json</code>	Configuration file for ESLint
<code>.gitignore</code>	Git files and folders to ignore
<code>next-env.d.ts</code>	TypeScript declaration file for Next.js
<code>tsconfig.json</code>	Configuration file for TypeScript
<code>jsconfig.json</code>	Configuration file for JavaScript

Files conventions

<code>_app</code>	<code>.js</code>	<code>.jsx</code>	<code>.ts</code>	Custom App
<code>_document</code>	<code>.js</code>	<code>.jsx</code>	<code>.ts</code>	Custom Document

<code>_error</code>	<code>.js</code> <code>.jsx</code> <code>.tsx</code>	Custom Error Page
<code>404</code>	<code>.js</code> <code>.jsx</code> <code>.tsx</code>	404 Error Page
<code>500</code>	<code>.js</code> <code>.jsx</code> <code>.tsx</code>	500 Error Page

Routes

Folder convention

<code>index</code>	<code>.js</code> <code>.jsx</code> <code>.tsx</code>	Home page
<code>folder/index</code>	<code>.js</code> <code>.jsx</code> <code>.tsx</code>	Nested page

File convention

<code>index</code>	<code>.js</code> <code>.jsx</code> <code>.tsx</code>	Home page
<code>file</code>	<code>.js</code> <code>.jsx</code> <code>.tsx</code>	Nested page

Dynamic routes

Folder convention

<code>[folder]/index</code>	<code>.js</code>	Dynamic route segment
<code>[... folder]/index</code>	<code>.js</code>	Catch-all route segment
<code>[[... folder]]/index</code>	<code>.js</code>	Optional catch-all route segment

File convention

<code>[file]</code>	<code>.js</code>	Dynamic route segment
	<code>.jsx</code>	
	<code>.tsx</code>	

[... file]

.js

Catch-all route

.jsx

segment

.tsx

[[... file]]

.js

Optional catch-all

.jsx

route segment

.tsx

Was this helpful?



 Using Pages Router
Features available in /pages

 Latest Version
15.5.4

(i) You are currently viewing the documentation for Pages Router.

Image Optimization

The Next.js `<Image>` component extends the HTML `` element to provide:

- **Size optimization:** Automatically serving correctly sized images for each device, using modern image formats like WebP.
- **Visual stability:** Preventing [layout shift ↗](#) automatically when images are loading.
- **Faster page loads:** Only loading images when they enter the viewport using native browser lazy loading, with optional blur-up placeholders.
- **Asset flexibility:** Resizing images on-demand, even images stored on remote servers.

To start using `<Image>`, import it from `next/image` and render it within your component.

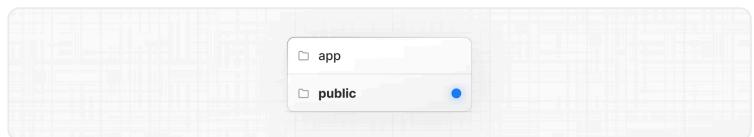
```
TS app/page.tsx TypeScript ▾   
  
import Image from 'next/image'  
  
export default function Page() {  
  return <Image src="" alt="" />  
}
```

The `src` property can be a [local](#) or [remote](#) image.

 **Watch:** Learn more about how to use `next/image` → [YouTube \(9 minutes\)](#).

Local images

You can store static files, like images and fonts, under a folder called `public` in the root directory. Files inside `public` can then be referenced by your code starting from the base URL (`/`).



TS app/page.tsx TypeScript

```
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="/profile.png"
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

If the image is statically imported, Next.js will automatically determine the intrinsic `width` and `height`. These values are used to determine the image ratio and prevent [Cumulative Layout Shift](#) while your image is loading.

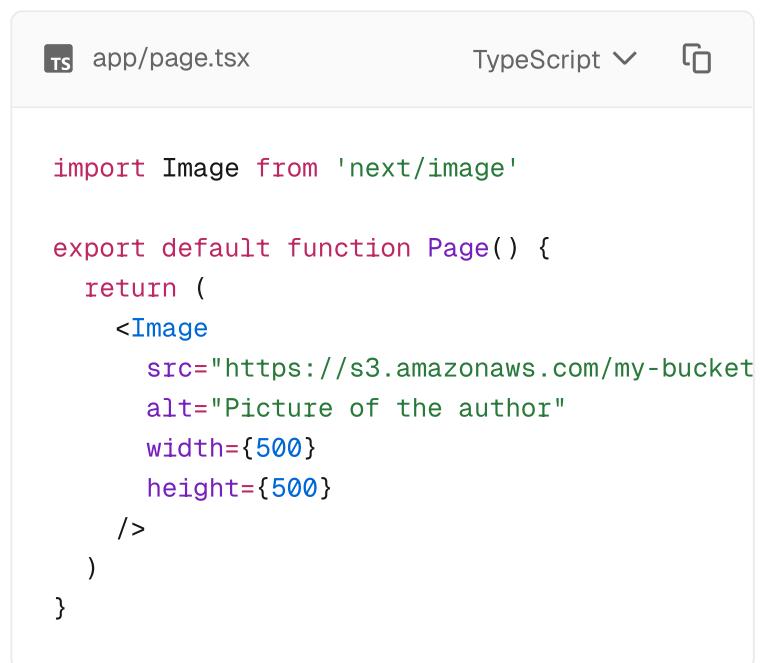
TS app/page.tsx TypeScript

```
import Image from 'next/image'
import ProfileImage from './profile.png'
```

```
export default function Page() {
  return (
    <Image
      src={ProfileImage}
      alt="Picture of the author"
      // width={500} automatically provided
      // height={500} automatically provided
      // blurDataURL="data:..." automatically
      // placeholder="blur" // Optional blur-
    />
  )
}
```

Remote images

To use a remote image, you can provide a URL string for the `src` property.



```
TS app/page.tsx TypeScript ▾
```

```
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="https://s3.amazonaws.com/my-bucket
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

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` are used to infer the correct aspect ratio of image and avoid layout shift from the image loading in. Alternatively, you can use the `fill` property to make the image fill the size of the parent element.

To safely allow images from remote servers, you need to 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:

```
ts next.config.ts TypeScript ▾
```

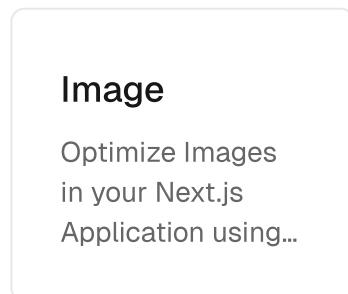
```
import type { NextConfig } from 'next'

const config: NextConfig = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 's3.amazonaws.com',
        port: '',
        pathname: '/my-bucket/**',
        search: ''
      },
    ],
  },
}

export default config
```

API Reference

See the API Reference for the full feature set of Next.js Image



Was this helpful?



Using Pages Router
Features available in /pages Latest Version
15.5.4

(i) You are currently viewing the documentation for Pages Router.

How to use fonts

The `next/font` module automatically optimizes your fonts and removes external network requests for improved privacy and performance.

It includes **built-in self-hosting** for any font file. This means you can optimally load web fonts with no layout shift.

To start using `next/font`, import it from `next/font/local` or `next/font/google`, call it as a function with the appropriate options, and set the `className` of the element you want to apply the font to. For example:

app/layout.tsx

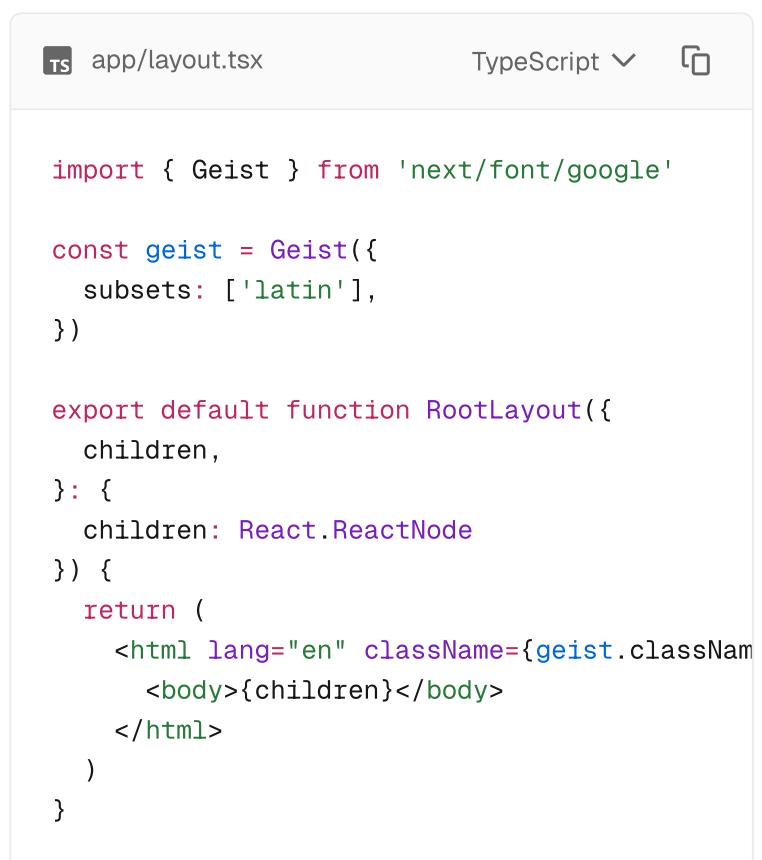
```
TS app/layout.tsx TypeScript ⓘ  
  
import { Geist } from 'next/font/google'  
  
const geist = Geist({  
  subsets: ['latin'],  
})  
  
export default function Layout({ children }: { children: React.ReactNode }) {  
  return (  
    <html lang="en" className={geist.className}>  
      <body>{children}</body>  
    </html>  
  )  
}
```

Fonts are scoped to the component they're used in. To apply a font to your entire application, add it to the [Root Layout](#).

Google fonts

You can automatically self-host any Google Font. Fonts are included stored as static assets and served from the same domain as your deployment, meaning no requests are sent to Google by the browser when the user visits your site.

To start using a Google Font, import your chosen font from `next/font/google`:



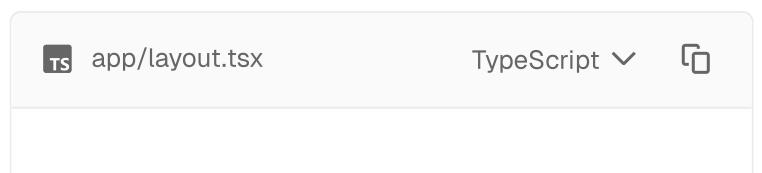
```
TS app/layout.tsx TypeScript ▾
```

```
import { Geist } from 'next/font/google'

const geist = Geist({
  subsets: ['latin'],
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={geist.className}>
      <body>{children}</body>
    </html>
  )
}
```

We recommend using [variable fonts ↗](#) for the best performance and flexibility. But if you can't use a variable font, you will need to specify a weight:



```
TS app/layout.tsx TypeScript ▾
```

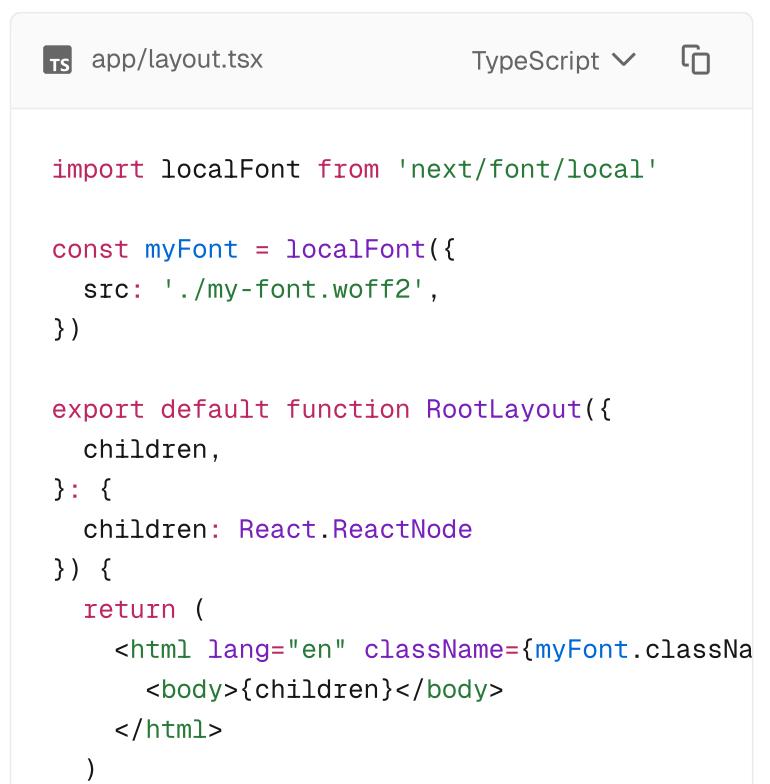
```
import { Roboto } from 'next/font/google'

const roboto = Roboto({
  weight: '400',
  subsets: ['latin'],
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={roboto.className}>
      <body>{children}</body>
    </html>
  )
}
```

Local fonts

To use a local font, import your font from `next/font/local` and specify the `src` of your local font file. Fonts can be stored in the `public` folder or co-located inside the `app` folder. For example:



```
TS app/layout.tsx TypeScript ▾ ⌂

import localFont from 'next/font/local'

const myFont = localFont({
  src: './my-font.woff2',
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={myFont.className}>
      <body>{children}</body>
    </html>
  )
}
```

}

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',
    },
  ],
})
```

API Reference

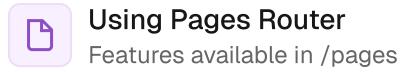
See the API Reference for the full feature set of Next.js Font

Font

API Reference for
the Font Module

Was this helpful?





ⓘ You are currently viewing the documentation for Pages Router.

How to use CSS in your application

Next.js provides several ways to style your application using CSS, including:

- [Tailwind CSS](#)
- [CSS Modules](#)
- [Global CSS](#)
- [External Stylesheets](#)
- [Sass](#)
- [CSS-in-JS](#)

Tailwind CSS

[Tailwind CSS ↗](#) is a utility-first CSS framework that provides low-level utility classes to build custom designs.

Install Tailwind CSS:

```
pnpm npm yarn bun
>_ Terminal
pnpm add -D tailwindcss @tailwindcss/postcss
```

Add the PostCSS plugin to your
postcss.config.mjs file:

JS postcss.config.mjs



```
export default {
  plugins: {
    '@tailwindcss/postcss': {},
  },
}
```

Import Tailwind in your global CSS file:

Folder styles/globals.css



```
@import 'tailwindcss';
```

Import the CSS file in your pages/_app.js file:

JS pages/_app.js



```
import '@/styles/globals.css'

export default function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

Now you can start using Tailwind's utility classes in your application:

TS pages/index.tsx

TypeScript ▾



```
export default function Home() {
  return (
    <main className="flex min-h-screen flex-col items-center justify-center p-4 sm:p-0">
      <h1 className="text-4xl font-bold">Welcome to Next.js!</h1>
    </main>
  )
}
```

Good to know: If you need broader browser support for very old browsers, see the [Tailwind CSS v3 setup instructions](#).

CSS Modules

CSS Modules locally scope CSS by generating unique class names. This allows you to use the same class in different files without worrying about naming collisions.

To start using CSS Modules, create a new file with the extension `.module.css` and import it into any component inside the `pages` directory:

 /styles/blog.module.css 

```
.blog {  
  padding: 24px;  
}
```

 pages/blog/index.tsx  

```
import styles from './blog.module.css'  
  
export default function Page() {  
  return <main className={styles.blog}></main>  
}
```

Global CSS

You can use global CSS to apply styles across your application.

Import the stylesheet in the `pages/_app.js` file to apply the styles to **every route** in your application:



```
import '@/styles/global.css'

export default function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

Due to the global nature of stylesheets, and to avoid conflicts, you should import them inside `pages/_app.js`.

External stylesheets

Next.js allows you to import CSS files from a JavaScript file. This is possible because Next.js extends the concept of `import` beyond JavaScript.

Import styles from `node_modules`

Since Next.js **9.5.4**, importing a CSS file from `node_modules` is permitted anywhere in your application.

For global stylesheets, like `bootstrap` or `nprogress`, you should import the file inside `pages/_app.js`. For example:



```
import 'bootstrap/dist/css/bootstrap.css'

export default function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

To import 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>x</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 **order of your CSS** depends on the **order you import styles in your code**.

For example, `base-button.module.css` will be ordered before `page.module.css` since `<BaseButton>` is imported before `page.module.css`:



```
import { BaseButton } from './base-button'
```

```
import styles from './page.module.css'

export default function Page() {
  return <BaseButton className={styles.primary} />
}
```

```
base-button.tsx TypeScript ▾
```

```
import styles from './base-button.module.css'

export function BaseButton() {
  return <button className={styles.primary} />
}
```

Recommendations

To keep CSS ordering predictable:

- Try to contain CSS imports to a single JavaScript or TypeScript entry file
- Import global styles and Tailwind stylesheets in the root of your application.
- **Use Tailwind CSS** for most styling needs as it covers common design patterns with utility classes.
- Use CSS Modules for component-specific styles when Tailwind utilities aren't sufficient.
- Use a consistent naming convention for your CSS modules. For example, using `<name>.module.css` over `<name>.tsx`.
- Extract shared styles into shared components to avoid duplicate imports.
- Turn off linters or formatters that auto-sort imports like ESLint's `sort-imports` ↗.
- You can use the `cssChunking` option in `next.config.js` to control how CSS is chunked.

Development vs Production

- In development (`next dev`), CSS updates apply instantly with [Fast Refresh](#).
- In production (`next build`), all CSS files are automatically concatenated into **many minified and code-split** `.css` files, ensuring the minimal amount of CSS is loaded for a route.
- CSS still loads with JavaScript disabled in production, but JavaScript is required in development for Fast Refresh.
- CSS ordering can behave differently in development, always ensure to check the build (`next build`) to verify the final CSS order.

Next Steps

Learn more about the features mentioned in this page.

Tailwind CSS

Style your Next.js Application using Tailwind CSS.

Sass

Learn how to use Sass in your Next.js application.

CSS-in-JS

Use CSS-in-JS libraries with Next.js



Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

How to deploy your Next.js application

Next.js can be deployed as a Node.js server, Docker container, static export, or adapted to run on different platforms.

Deployment Option	Feature Support
Node.js server	All
Docker container	All
Static export	Limited
Adapters	Platform-specific

Node.js server

Next.js can be deployed to any provider that supports Node.js. Ensure your `package.json` has the `"build"` and `"start"` scripts:



```
        "build": "next build",
        "start": "next start"
    }
}
```

Then, run `npm run build` to build your application and `npm run start` to start the Node.js server.

This server supports all Next.js features. If needed, you can also eject to a [custom server](#).

Node.js deployments support all Next.js features.

Learn how to [configure them](#) for your infrastructure.

Templates

- [Flightcontrol ↗](#)
- [Railway ↗](#)
- [Replit ↗](#)

Docker

Next.js can be deployed to any provider that supports [Docker ↗](#) containers. This includes container orchestrators like Kubernetes or a cloud provider that runs Docker.

Docker deployments support all Next.js features. Learn how to [configure them](#) for your infrastructure.

Note for development: While Docker is excellent for production deployments, consider using local development (`npm run dev`) instead of Docker during development on Mac and Windows for better performance. [Learn more about optimizing local development](#).

Templates

- [Docker ↗](#)
 - [Docker Multi-Environment ↗](#)
 - [DigitalOcean ↗](#)
 - [Fly.io ↗](#)
 - [Google Cloud Run ↗](#)
 - [Render ↗](#)
 - [SST ↗](#)
-

Static 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 [static exports](#), it can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets. This includes tools like AWS S3, Nginx, or Apache.

Running as a [static export does not support](#) Next.js features that require a server. [Learn more.](#)

Templates

- [GitHub Pages ↗](#)
-

Adapters

Next.js can be adapted to run on different platforms to support their infrastructure capabilities.

Refer to each provider's documentation for information on supported Next.js features:

- [AWS Amplify Hosting ↗](#)
- [Cloudflare ↗](#)
- [Deno Deploy ↗](#)
- [Netlify ↗](#)
- [Vercel ↗](#)

Note: We are working on a [Deployment Adapters API ↗](#) for all platforms to adopt. After completion, we will add documentation on how to write your own adapters.

Was this helpful?    



Using Pages Router

Features available in /pages



Latest Version

15.5.4



(i) You are currently viewing the documentation for
Pages Router.

Guides

AMP

With minimal config, and without leaving...

Analytics

Measure and track page performance using Next.js

Authenticati...

Learn how to implement authentication in...

Babel

Extend the babel preset added by Next.js with your...

CI Build Cac...

Learn how to configure CI to cache Next.js...

Content Sec...

Learn how to set a Content Security Policy (CSP) for...

CSS-in-JS

Custom Serv...

Use CSS-in-JS
libraries with
Next.js

Start a Next.js app
programmatically
using a custom...

Debugging

Learn how to
debug your Next.js
application with...

Draft Mode

Next.js has draft
mode to toggle
between static...

Environment ...

Learn to add and
access
environment...

Forms

Learn how to
handle form
submissions and...

ISR

Learn how to
create or update
static pages at...

Instrumentat...

Learn how to use
instrumentation to
run code at serve...

International...

Next.js has built-in
support for
internationalized...

Lazy Loading

Lazy load
imported libraries
and React...

MDX

Learn how to
configure MDX to
write JSX in your...

Migrating

Learn how to
migrate from
popular...

Multi-Zones

Learn how to build micro-frontends using Next.js...

OpenTelemetry

Learn how to instrument your Next.js app with...

Package Bundling

Learn how to optimize your application's...

PostCSS

Extend the PostCSS config and plugins added...

Preview Mode

Next.js has the preview mode for statically...

Production

Recommendations to ensure the best performance and...

Redirecting

Learn the different ways to handle redirects in...

Sass

Learn how to use Sass in your Next.js application.

Scripts

Optimize 3rd party scripts with the built-in Script...

Self-Hosting

Learn how to self-host your Next.js application on a...

Static Exports

Tailwind CSS

Next.js enables starting as a static site or Single-Pag...

Style your Next.js Application using Tailwind CSS.

Testing

Learn how to set up Next.js with three commonly...

Third Party Li...

Optimize the performance of third-party...

Upgrading

Learn how to upgrade to the latest versions of...

Was this helpful?





Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

How to create AMP pages in Next.js

► Examples

Warning: Built-in AMP support will be removed in Next.js 16.

With Next.js you can turn any React page into an AMP page, with minimal config, and without leaving React.

You can read more about AMP in the official [amp.dev](#) site.

Enabling AMP

To enable AMP support for a page, and to learn more about the different AMP configs, read the [API documentation for `next/amp`](#).

Caveats

- Only CSS-in-JS is supported. [CSS Modules](#) aren't supported by AMP pages at the moment.

Adding AMP Components

The AMP community provides [many components](#) ↗ to make AMP pages more interactive. Next.js will automatically import all components used on a page and there is no need to manually import AMP component scripts:

```
export const config = { amp: true }

function MyAmpPage() {
  const date = new Date()

  return (
    <div>
      <p>Some time: {date.toJSON()}</p>
      <amp-timeago
        width="0"
        height="15"
        datetime={date.toJSON()}
        layout="responsive"
      >
      .
      </amp-timeago>
    </div>
  )
}

export default MyAmpPage
```

The above example uses the [amp-timeago](#) ↗ component.

By default, the latest version of a component is always imported. If you want to customize the version, you can use `next/head`, as in the following example:

```
import Head from 'next/head'
```

```
export const config = { amp: true }

function MyAmpPage() {
  const date = new Date()

  return (
    <div>
      <Head>
        <script
          async
          key="amp-timeago"
          custom-element="amp-timeago"
          src="https://cdn.ampproject.org/v0/
        />
      </Head>

      <p>Some time: {date.toJSON()}</p>
      <amp-timeago
        width="0"
        height="15"
        datetime={date.toJSON()}
        layout="responsive"
      >
      .
      </amp-timeago>
    </div>
  )
}

export default MyAmpPage
```

AMP Validation

AMP pages are automatically validated with [amphtml-validator](#) during development. Errors and warnings will appear in the terminal where you started Next.js.

Pages are also validated during [Static HTML export](#) and any warnings / errors will be printed to the terminal. Any AMP errors will cause the export to exit with status code 1 because the export is not valid AMP.

Custom Validators

You can set up custom AMP validator in `next.config.js` as shown below:

```
module.exports = {
  amp: {
    validator: './custom_validator.js',
  },
}
```

Skip AMP Validation

To turn off AMP validation add the following code to `next.config.js`

```
experimental: {
  amp: {
    skipValidation: true
  }
}
```

AMP in Static HTML Export

When using [Static HTML export](#) statically prerender pages, Next.js will detect if the page supports AMP and change the exporting behavior based on that.

For example, the hybrid AMP page `pages/about.js` would output:

- `out/about.html` - HTML page with client-side React runtime
- `out/about.amp.html` - AMP page

And if `pages/about.js` is an AMP-only page, then it would output:

- `out/about.html` - Optimized AMP page

Next.js will automatically insert a link to the AMP version of your page in the HTML version, so you don't have to, like so:

```
<link rel="amphtml" href="/about.amp.html" />
```

And the AMP version of your page will include a link to the HTML page:

```
<link rel="canonical" href="/about" />
```

When `trailingSlash` is enabled the exported pages for `pages/about.js` would be:

- `out/about/index.html` - HTML page
- `out/about.amp/index.html` - AMP page

TypeScript

AMP currently doesn't have built-in types for TypeScript, but it's in their roadmap ([#13791 ↗](#)).

As a workaround you can manually create a file called `amp.d.ts` inside your project and add these [custom types ↗](#).

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to set up analytics

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.

Client Instrumentation

For more advanced analytics and monitoring needs, Next.js provides a

`instrumentation-client.js|ts` file that runs before your application's frontend code starts executing. This is ideal for setting up global analytics, error tracking, or performance monitoring tools.

To use it, create an `instrumentation-client.js` or `instrumentation-client.ts` file in your application's root directory:

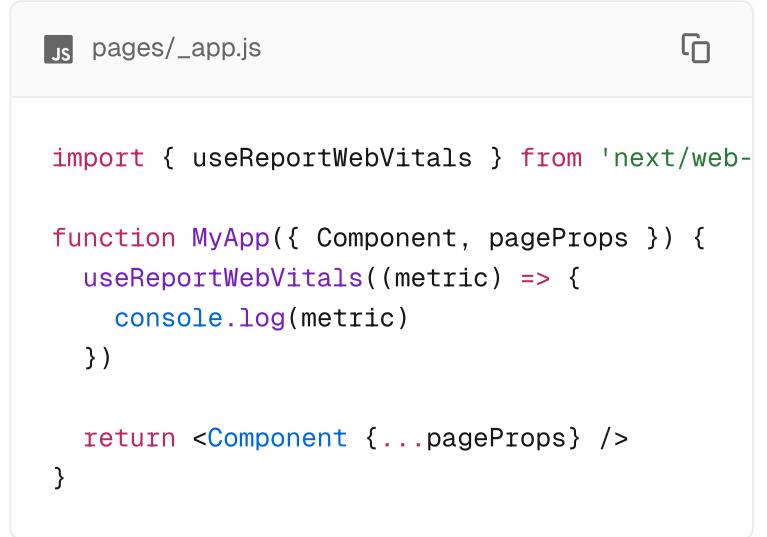
`JS instrumentation-client.js`



```
// Initialize analytics before the app starts
console.log('Analytics initialized')
```

```
// Set up global error tracking
window.addEventListener('error', (event) => {
  // Send to your error tracking service
  reportError(event.error)
})
```

Build Your Own



```
js pages/_app.js

import { useReportWebVitals } from 'next/web'

function MyApp({ Component, pageProps }) {
  useReportWebVitals((metric) => {
    console.log(metric)
  })

  return <Component {...pageProps} />
}
```

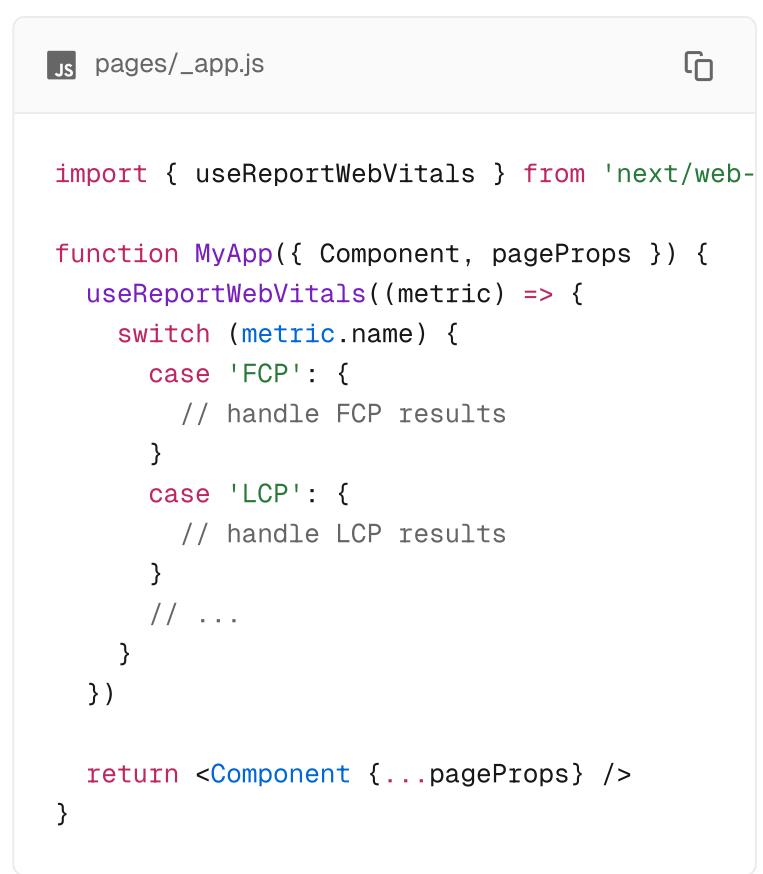
View the [API Reference](#) for more information.

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.



```
JS pages/_app.js

import { useReportWebVitals } from 'next/web-vitals'

function MyApp({ Component, pageProps }) {
  useReportWebVitals((metric) => {
    switch (metric.name) {
      case 'FCP': {
        // handle FCP results
      }
      case 'LCP': {
        // handle LCP results
      }
      // ...
    }
  })
}

return <Component {...pageProps} />
}
```

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 result
      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
  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
  // https://github.com/vercel/next.js/bl
  window.gtag('event', metric.name, {
    value: Math.round(
      metric.name === 'CLS' ? metric.value
    ), // values must be integers
    event_label: metric.id, // id unique to
    non_interaction: true, // avoids affecting
  })
})
```

Read more about [sending results to Google Analytics](#) ↗.

Was this helpful?    

 Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.

 Latest Version

15.5.4



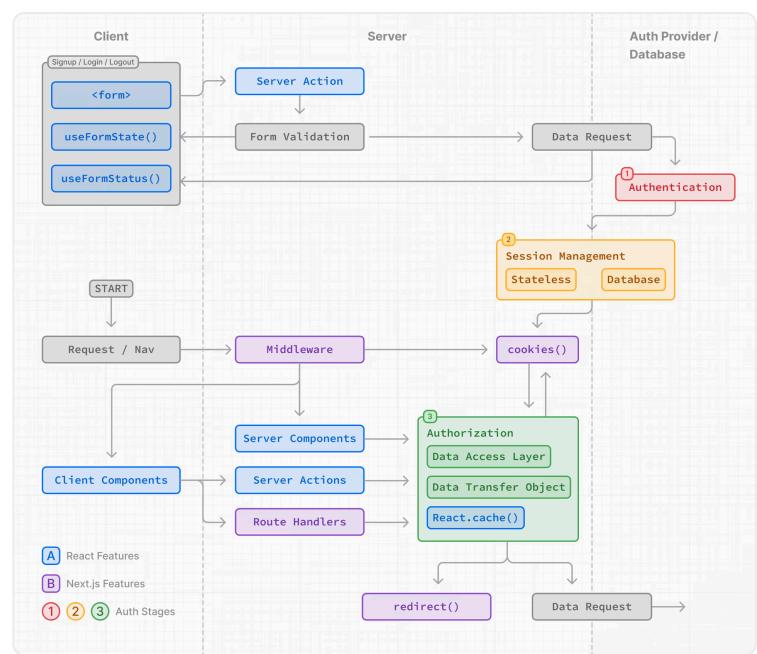
How to implement authentication in Next.js

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.

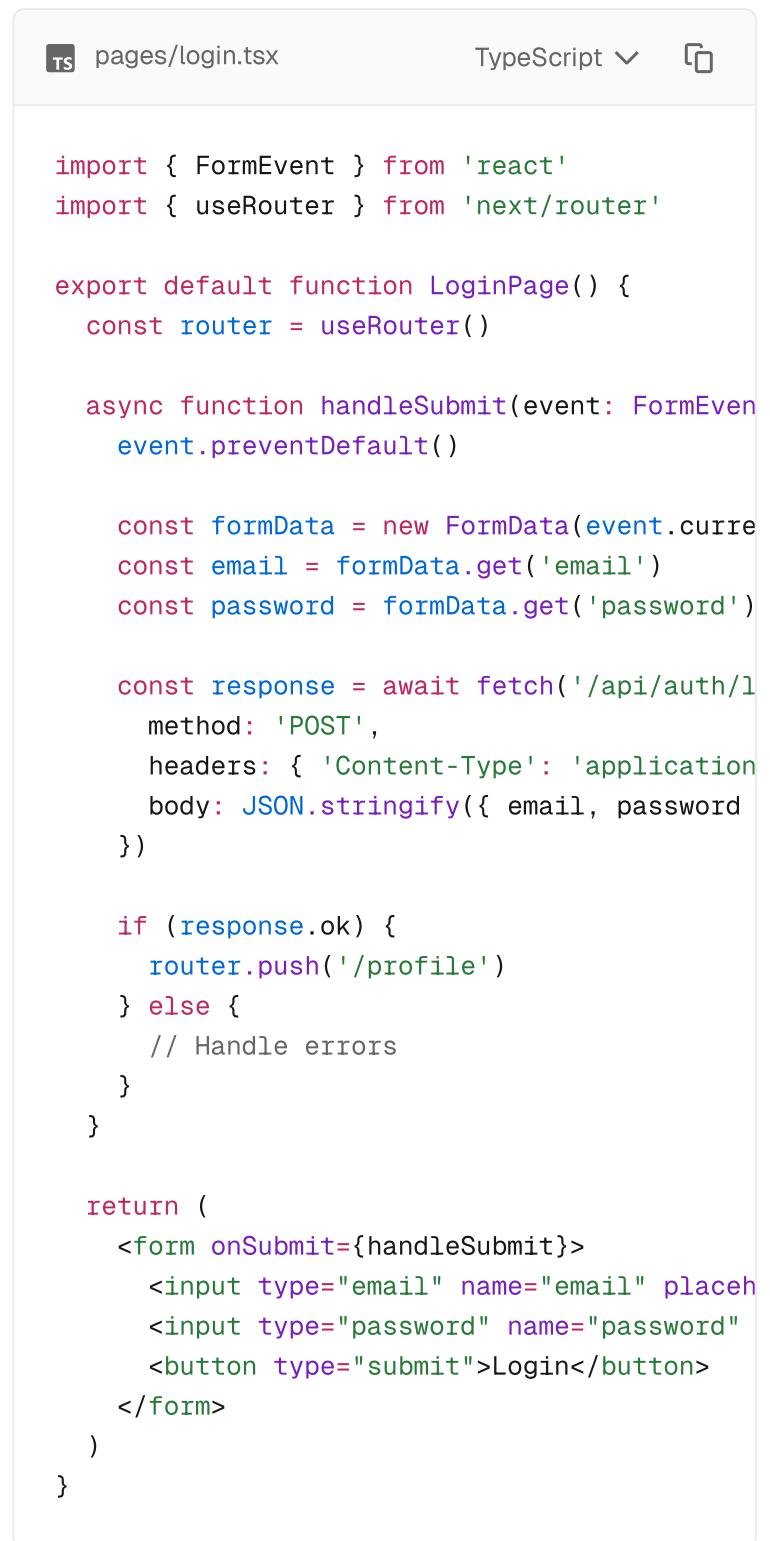
Authentication

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:



The screenshot shows a code editor window with the following details:

- File path: pages/login.tsx
- Language: TypeScript (indicated by the 'TS' icon)
- Toolbar: Includes a dropdown for 'TypeScript' and a refresh/circular arrow icon.

```
import { FormEvent } from 'react'
import { useRouter } from 'next/router'

export default function LoginPage() {
  const router = useRouter()

  async function handleSubmit(event: FormEvent) {
    event.preventDefault()

    const formData = new FormData(event.currentTarget)
    const email = formData.get('email')
    const password = formData.get('password')

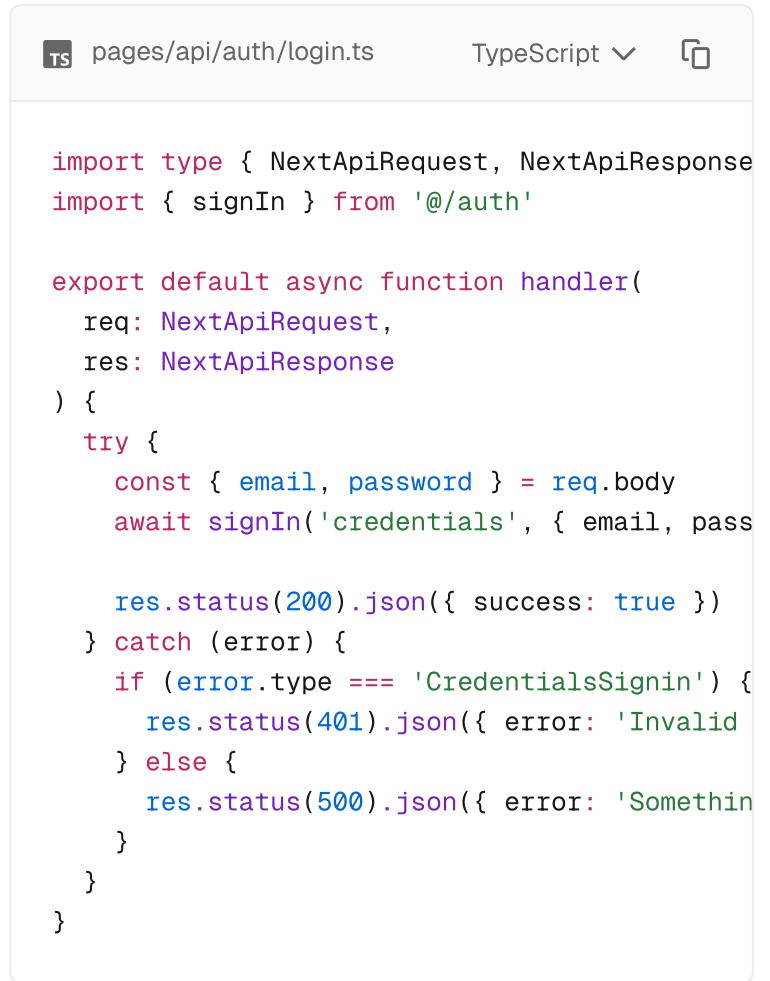
    const response = await fetch('/api/auth/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ email, password })
    })

    if (response.ok) {
      router.push('/profile')
    } else {
      // Handle errors
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="email" name="email" placeholder="Email" />
      <input type="password" name="password" placeholder="Password" />
      <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' })
      }
    }
  }
}
```

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

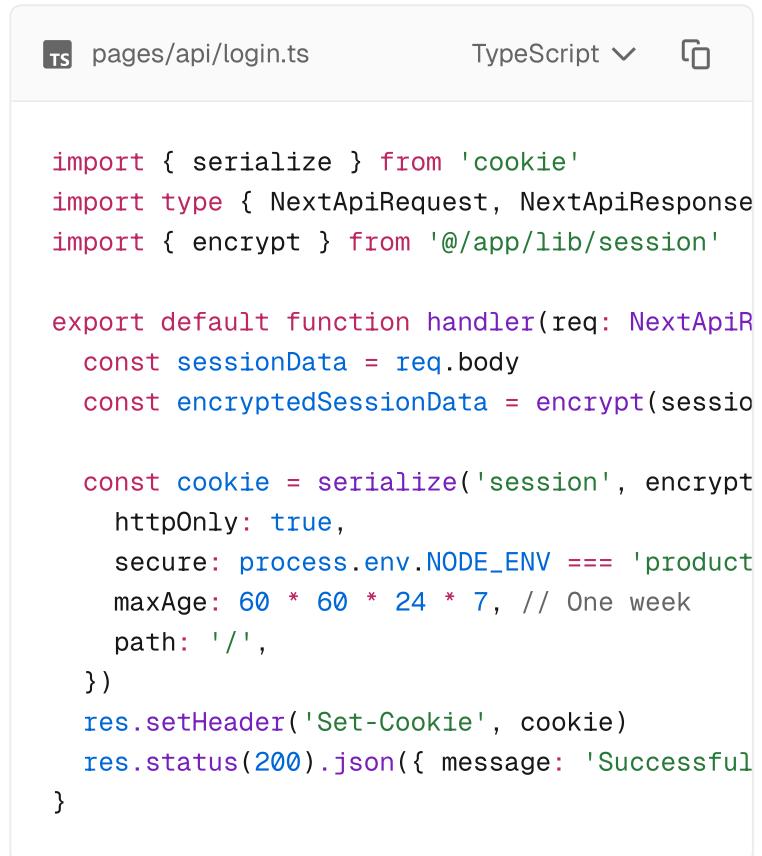
the encrypted session ID. This method is more secure, but can be complex and use more server resources.

Good to know: While you can use either method, or both, we recommend using a session management library such as [iron-session](#) ↗ or [Jose](#) ↗.

Stateless Sessions

Setting and deleting cookies

You can use [API Routes](#) to set the session as a cookie on the server:



The screenshot shows a code editor window with a TypeScript file named `pages/api/login.ts`. The code uses the `cookie` and `encrypt` modules to handle session data. It defines a function `handler` that takes a `NextApiRequest` and `NextApiResponse`. It extracts the `sessionData` from the request body, encrypts it, and then serializes it into a cookie object with various options like `httpOnly`, `secure`, and `maxAge`. Finally, it sets the `Set-Cookie` header and returns a successful response.

```
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: 'Successful' })
}
```

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](#)).

Creating a Session on the Server:



The screenshot shows a code editor window with the following details:

- File path: pages/api/create-session.ts
- Language: TypeScript
- Code content:

```
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' })
  }
}
```

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)

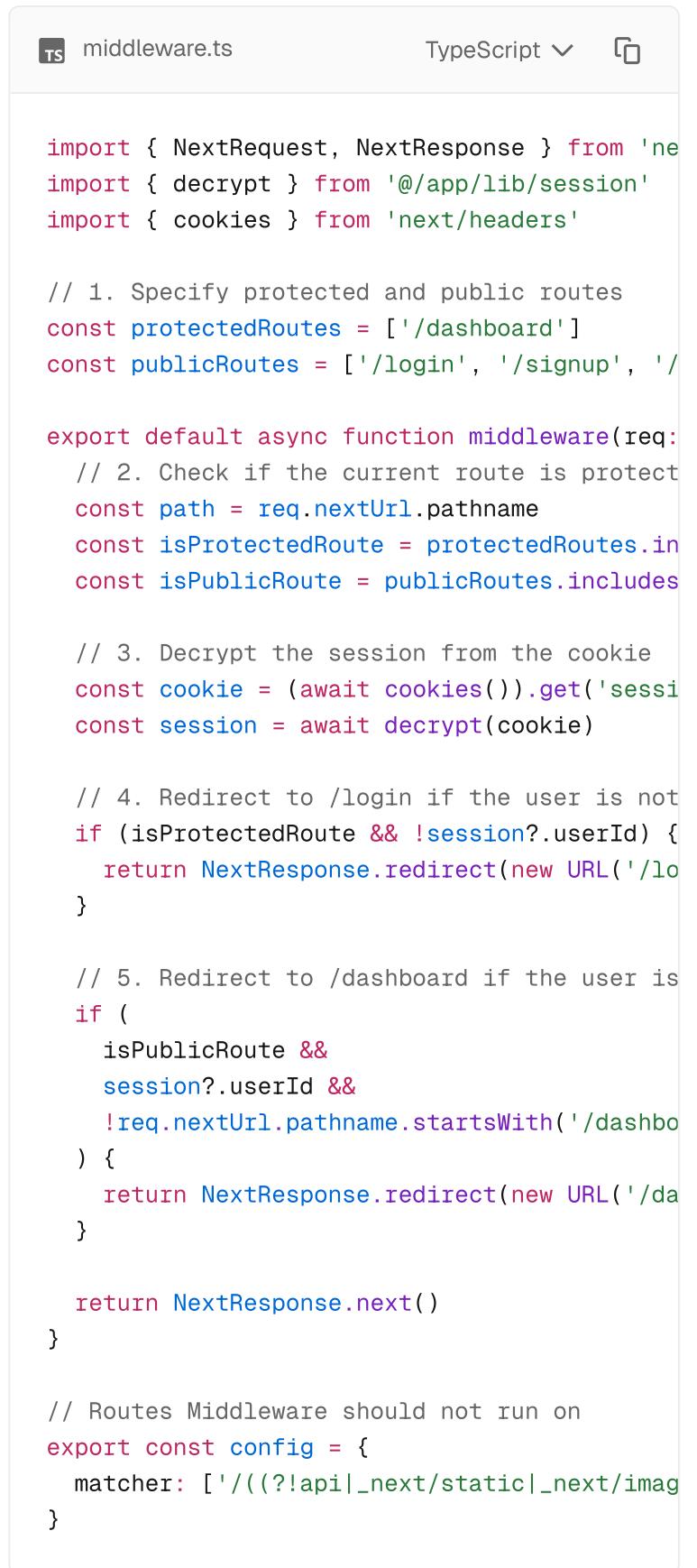
There are some cases where you may want to use [Middleware](#) and redirect users based on permissions:

- 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:



```
TS middleware.ts TypeScript ▾
```

```
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', '/about']

export default async function middleware(req: Request) {
  // 2. Check if the current route is protected
  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')
  const session = await decrypt(cookie)

  // 4. Redirect to /login if the user is not logged in
  if (isProtectedRoute && !session?.userId) {
    return NextResponse.redirect(new URL('/login'))
  }

  // 5. Redirect to /dashboard if the user is not logged in and requested a protected route
  if (
    isPublicRoute &&
    session?.userId &&
    !req.nextUrl.pathname.startsWith('/dashboard')
  ) {
    return NextResponse.redirect(new URL('/dashboard'))
  }

  return NextResponse.next()
}

// Routes Middleware should not run on static files
export const config = {
  matcher: [ '/((?!api|_next/static|_next/image).*)' ]
}
```

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:

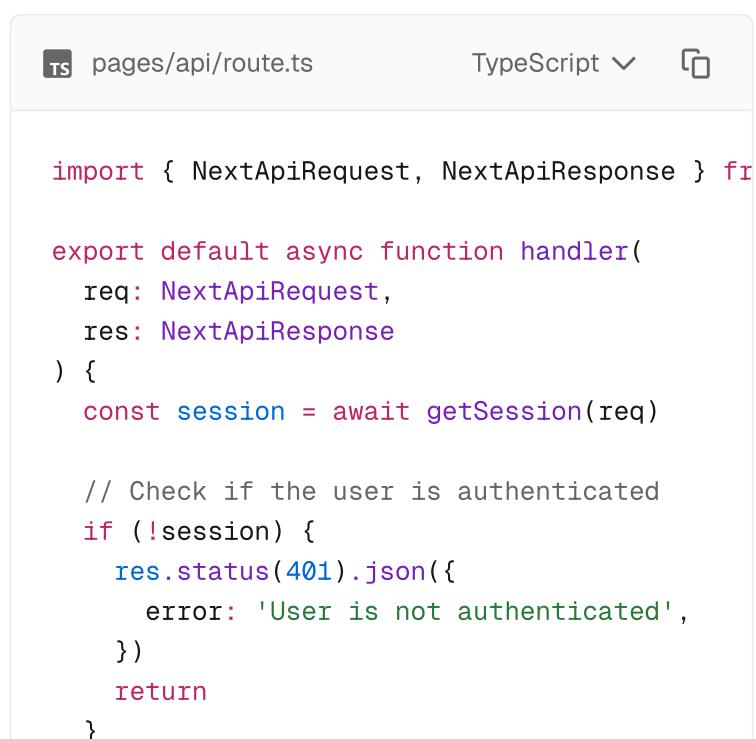
- In Middleware, you can also read cookies using `req.cookies.get('session').value`.
- Middleware uses the [Edge Runtime](#), check if your Auth library and session management library are compatible.
- You can use the `matcher` property in the Middleware to specify which routes Middleware should run on. Although, for auth, it's recommended Middleware runs on all routes.

Creating a Data Access Layer (DAL)

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:



```
TS pages/api/route.ts TypeScript ▾ ⌂

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
  })
  return
}

// Proceed with the route for authorized us
// ... 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 ↗](#)
- [Better Auth ↗](#)
- [Clerk ↗](#)
- [Kinde ↗](#)
- [Logto ↗](#)
- [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 ↗](#)

Was this helpful?    



Using Pages Router
Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version
15.5.4



How to configure Babel in Next.js

► Examples

Next.js includes the `next/babel` preset to your app, which includes everything needed to compile React applications and server-side code. But if you want to extend the default Babel configs, it's also possible.

Adding Presets and Plugins

To start, you only need to define a `.babelrc` file (or `babel.config.js`) in the root directory of your project. If such a file is found, it will be considered as the *source of truth*, and therefore it needs to define what Next.js needs as well, which is the `next/babel` preset.

Here's an example `.babelrc` file:

```
 .babelrc   
  
{  
  "presets": ["next/babel"],  
  "plugins": []  
}
```

You can [take a look at this file ↗](#) to learn about the presets included by `next/babel`.

To add presets/plugins **without configuring them**, you can do it this way:

```
📄 .babelrc
```

```
{  
  "presets": ["next/babel"],  
  "plugins": ["@babel/plugin-proposal-do-expr"]}
```

Customizing Presets and Plugins

To add presets/plugins **with custom configuration**, do it on the `next/babel` preset like so:

```
📄 .babelrc
```

```
{  
  "presets": [  
    [  
      "next/babel",  
      {  
        "preset-env": {},  
        "transform-runtime": {},  
        "styled-jsx": {},  
        "class-properties": {}  
      }  
    ]  
  ],  
  "plugins": []  
}
```

To learn more about the available options for each config, visit babel's [documentation ↗](#) site.

Good to know:

- Next.js uses the [current Node.js version ↗](#) for server-side compilations.
- The `modules` option on `"preset-env"` should be kept to `false`, otherwise webpack code splitting is turned off.

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to configure Continuous Integration (CI) build caching

To improve build performance, Next.js saves a cache to `.next/cache` that is shared between builds.

To take advantage of this cache in Continuous Integration (CI) environments, your CI workflow will need to be configured to correctly persist the cache between builds.

If your CI is not configured to persist `.next/cache` between builds, you may see a [No Cache Detected](#) error.

Here are some example cache configurations for common CI providers:

Vercel

Next.js caching is automatically configured for you. There's no action required on your part. If you are using Turborepo on Vercel, [learn more here ↗](#).

CircleCI

Edit your `save_cache` step in

`.circleci/config.yml` to include `.next/cache`:

```
steps:  
  - save_cache:  
    key: dependency-cache-{{ checksum "yarn  
    paths:  
      - ./node_modules  
      - ./next/cache
```

If you do not have a `save_cache` key, please follow CircleCI's [documentation on setting up build caching ↗](#).

Travis CI

Add or merge the following into your

`.travis.yml`:

```
cache:  
  directories:  
    - $HOME/.cache/yarn  
    - node_modules  
    - .next/cache
```

GitLab CI

Add or merge the following into your

`.gitlab-ci.yml`:

```
cache:  
  key: ${CI_COMMIT_REF_SLUG}  
  paths:
```

```
- node_modules/  
- .next/cache/
```

Netlify CI

Use [Netlify Plugins ↗](#) with
[@netlify/plugin-nextjs ↗](#).

AWS CodeBuild

Add (or merge in) the following to your
`buildspec.yml`:

```
cache:  
  paths:  
    - 'node_modules/**/*' # Cache `node_modules`  
    - '.next/cache/**/*' # Cache Next.js files
```

GitHub Actions

Using GitHub's [actions/cache ↗](#), add the following step in your workflow file:

```
uses: actions/cache@v4  
with:  
  # See here for caching with `yarn`, `bun` or  
  path: |  
    ~/.npm  
    ${{ github.workspace }}/.next/cache  
  # Generate a new cache whenever packages or  
  key: ${{ runner.os }}-nextjs-${{ hashFiles(  
    # If source files changed but packages didn't  
    restore-keys: |  
    ${{ runner.os }}-nextjs-${{ hashFiles('**
```

Bitbucket Pipelines

Add or merge the following into your `bitbucket-pipelines.yml` at the top level (same level as `pipelines`):

```
definitions:  
  caches:  
    nextcache: .next/cache
```

Then reference it in the `caches` section of your pipeline's `step`:

```
- step:  
  name: your_step_name  
  caches:  
    - node  
    - nextcache
```

Heroku

Using Heroku's [custom cache](#), add a `cacheDirectories` array in your top-level `package.json`:

```
"cacheDirectories": [".next/cache"]
```

Azure Pipelines

Using Azure Pipelines' [Cache task](#), add the following task to your pipeline yaml file

somewhere prior to the task that executes

next build:

```
- task: Cache@2
  displayName: 'Cache .next/cache'
  inputs:
    key: next | $(Agent.OS) | yarn.lock
    path: '$(System.DefaultWorkingDirectory)/'
```

Jenkins (Pipeline)

Using Jenkins' [Job Cacher](#) plugin, add the following build step to your `Jenkinsfile` where you would normally run `next build` or `npm install`:

```
stage("Restore npm packages") {
  steps {
    // Writes lock-file to cache based on
    writeFile file: "next-lock.cache", te

    cache(caches: [
      arbitraryFileCache(
        path: "node_modules",
        includes: "**/*",
        cacheValidityDecidingFile: "p
      )
    ])
    sh "npm install"
  }
}

stage("Build") {
  steps {
    // Writes lock-file to cache based on
    writeFile file: "next-lock.cache", te

    cache(caches: [
      arbitraryFileCache(
        path: ".next/cache",
        includes: "**/*",
        cacheValidityDecidingFile: "n
      )
    ])
    // aka `next build`
```

```
        sh "npm run build"
    }
}
```

Was this helpful?     



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to set a Content Security Policy (CSP) for your Next.js application

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?

CSP can block both inline and external scripts to prevent attacks. A nonce lets you safely allow specific scripts to run—only if they include the matching nonce value.

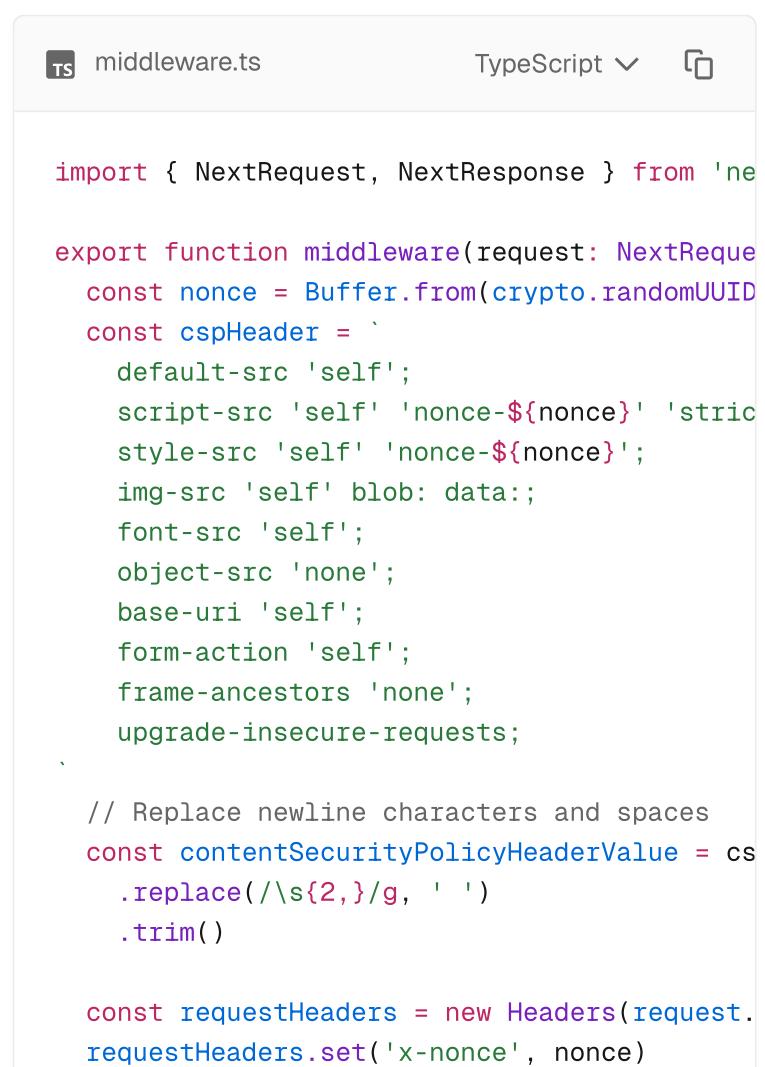
If an attacker wanted to load a script into your page, they'd need to guess the nonce value. That's why the nonce must be unpredictable and unique for every request.

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:



```
TS middleware.ts TypeScript ▾
```

```
import { NextRequest, NextResponse } from 'next'

export function middleware(request: NextRequest) {
  const nonce = Buffer.from(crypto.randomUUID())
  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 = cs
    .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.



The screenshot shows a code editor window with the following details:

- File Name:** middleware.ts
- TypeScript Version:** indicated by a dropdown menu.
- Code Content:**

```

export const config = {
  matcher: [
    /*
     * Match all request paths except for the
     * - api (API routes)
     * - _next/static (static files)
     * - _next/image (image optimization file)
     * - favicon.ico (favicon file)
    */
    {
      source: '/((?!api|_next/static|_next/im
      missing: [
        { type: 'header', key: 'next-router-p
          { type: 'header', key: 'purpose', val
        ],
      },
    ],
  ],
}

```

How nonces work in Next.js

To use a nonce, your page must be **dynamically rendered**. This is because Next.js applies nonces during **server-side rendering**, based on the CSP header present in the request. Static pages are generated at build time, when no request or response headers exist—so no nonce can be injected.

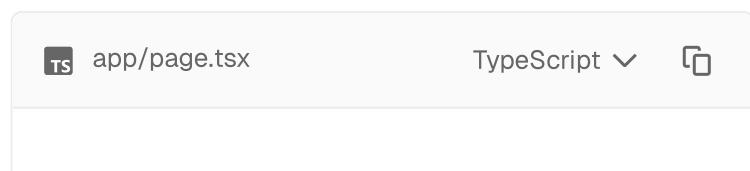
Here's how nonce support works in a dynamically rendered page:

1. **Middleware generates a nonce:** Your middleware creates a unique nonce for the request, adds it to your `Content-Security-Policy` header, and also sets it in a custom `x-nonce` header.
2. **Next.js extracts the nonce:** During rendering, Next.js parses the `Content-Security-Policy` header and extracts the nonce using the `'nonce-{value}'` pattern.
3. **Nonce is applied automatically:** Next.js attaches the nonce to:
 4. Framework scripts (React, Next.js runtime)
 5. Page-specific JavaScript bundles
 6. Inline styles and scripts generated by Next.js
 7. Any `<Script>` components using the `nonce` prop

Because of this automatic behavior, you don't need to manually add a nonce to each tag.

Forcing dynamic rendering

If you're using nonces, you may need to explicitly opt pages into dynamic rendering:



The screenshot shows a code editor interface. At the top, there is a tab labeled "app/page.tsx". To the left of the tab, there is a small "TS" icon indicating that the file is written in TypeScript. To the right of the tab, there is a "TypeScript" dropdown menu and a "File" icon. The main area of the editor is currently empty, showing a light gray background.

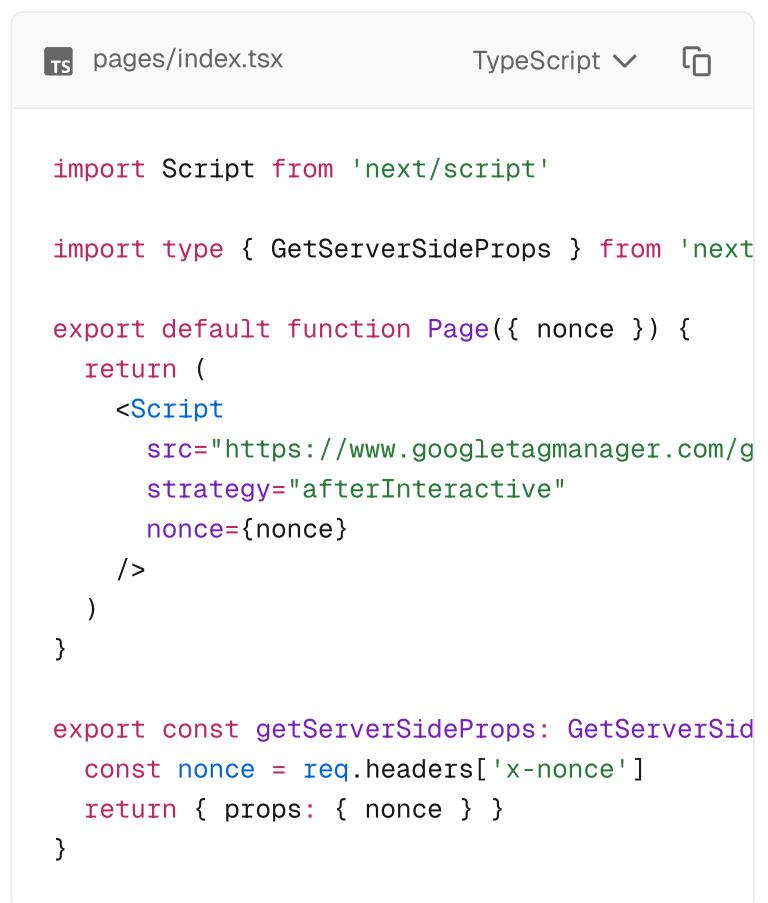
```
import { connection } from 'next/server'

export default async function Page() {
  // wait for an incoming request to render t
  await connection()
  // Your page content
}
```

Reading the nonce

You can provide the nonce to your page using

`getServerSideProps`:



The screenshot shows a code editor window with the file name "pages/index.tsx" at the top. The code is written in TypeScript and includes a `Script` component and a `getServerSideProps` function.

```
import Script from 'next/script'

import type { GetServerSideProps } from 'next'

export default function Page({ nonce }) {
  return (
    <Script
      src="https://www.googletagmanager.com/g
      strategy="afterInteractive"
      nonce={nonce}
    />
  )
}

export const getServerSideProps: GetServerSid
const nonce = req.headers['x-nonce']
return { props: { nonce } }
}
```

You can also access the nonce in `_document.tsx` for Pages Router applications:



The screenshot shows a code editor window with the file name "pages/_document.tsx" at the top. The code imports various components from the "Document" module.

```
import Document, {
  Html,
  Head,
  Main,
  NextScript,
  DocumentContext,
  DocumentInitialProps,
```

```

} from 'next/document'

interface ExtendedDocumentProps extends DocumentProps {
  nonce?: string
}

class MyDocument extends Document<ExtendedDocumentProps> {
  static async getInitialProps(
    ctx: DocumentContext
  ): Promise<ExtendedDocumentProps> {
    const initialProps = await Document.getInitialProps(ctx)
    const nonce = ctx.req?.headers?.['x-nonce']

    return {
      ...initialProps,
      nonce,
    }
  }

  render() {
    const { nonce } = this.props

    return (
      <Html lang="en">
        <Head nonce={nonce} />
        <body>
          <Main />
          <NextScript nonce={nonce} />
        </body>
      </Html>
    )
  }
}

export default MyDocument

```

Static vs Dynamic Rendering with CSP

Using nonces has important implications for how your Next.js application renders:

Dynamic Rendering Requirement

When you use nonces in your CSP, **all pages must be dynamically rendered**. This means:

- Pages will build successfully but may encounter runtime errors if not properly configured for dynamic rendering
- Each request generates a fresh page with a new nonce
- Static optimization and Incremental Static Regeneration (ISR) are disabled
- Pages cannot be cached by CDNs without additional configuration
- **Partial Prerendering (PPR) is incompatible** with nonce-based CSP since static shell scripts won't have access to the nonce

Performance Implications

The shift from static to dynamic rendering affects performance:

- **Slower initial page loads:** Pages must be generated on each request
- **Increased server load:** Every request requires server-side rendering
- **No CDN caching:** Dynamic pages cannot be cached at the edge by default
- **Higher hosting costs:** More server resources needed for dynamic rendering

When to use nonces

Consider nonces when:

- You have strict security requirements that prohibit '`'unsafe-inline'`'
- Your application handles sensitive data
- You need to allow specific inline scripts while blocking others
- Compliance requirements mandate strict CSP

Without Nonces

For applications that do not require nonces, you can set the CSP header directly in your `next.config.js` file:

```
JS next.config.js Copy  
  
const cspHeader = `  
    default-src 'self';  
    script-src 'self' 'unsafe-eval' 'unsafe-i  
    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, '  
            '),  
          ],  
        ],  
      },  
    ]  
  },  
}
```

Development vs Production Considerations

CSP implementation differs between development and production environments:

Development Environment

In development, you will need to enable '`'unsafe-eval'`' to support APIs that provide additional debugging information:

A screenshot of a code editor showing a file named 'middleware.ts'. The file contains TypeScript code for a middleware function. The code includes a nonce generation, a Content-Security-Policy header configuration, and a comment indicating the rest of the middleware implementation. The code editor has a 'TypeScript' dropdown and a save icon.

```
TS middleware.ts TypeScript ▾
```

```
export function middleware(request: NextRequest) {
  const nonce = Buffer.from(crypto.randomUUID().toString());
  const isDev = process.env.NODE_ENV === 'development';

  const cspHeader = `

    default-src 'self';
    script-src 'self' 'nonce-${nonce}' 'strictDynamic';
    style-src 'self' ${isDev ? "'unsafe-inline'" : "'self'"};
    img-src 'self' blob: data:;
    font-src 'self';
    object-src 'none';
    base-uri 'self';
    form-action 'self';
    frame-ancestors 'none';
    upgrade-insecure-requests;

  `;

  // Rest of middleware implementation
}
```

Production Deployment

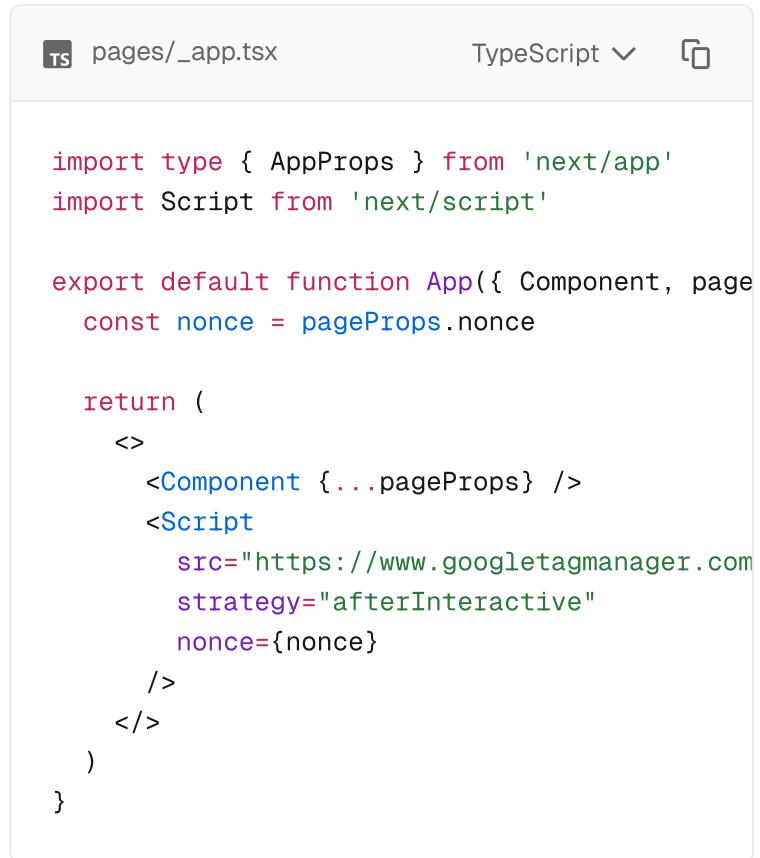
Common issues in production:

- **Nonce not applied:** Ensure your middleware runs on all necessary routes
- **Static assets blocked:** Verify your CSP allows Next.js static assets
- **Third-party scripts:** Add necessary domains to your CSP policy

Troubleshooting

Third-party Scripts

When using third-party scripts with CSP, ensure you add the necessary domains and pass the nonce:



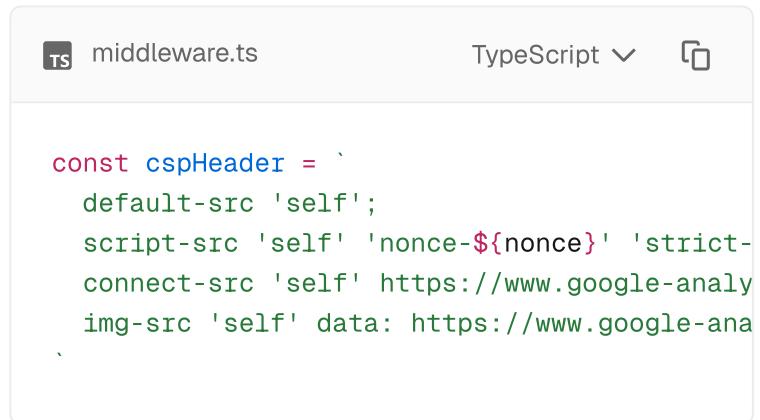
The screenshot shows a code editor window for a file named `pages/_app.tsx`. The file contains the following TypeScript code:

```
import type { AppProps } from 'next/app'
import Script from 'next/script'

export default function App({ Component, pageProps }) {
  const nonce = pageProps.nonce

  return (
    <>
      <Component {...pageProps} />
      <Script
        src="https://www.googletagmanager.com"
        strategy="afterInteractive"
        nonce={nonce}
      />
    </>
  )
}
```

Update your CSP to allow third-party domains:



The screenshot shows a code editor window for a file named `middleware.ts`. The file contains the following TypeScript code:

```
const cspHeader = `
  default-src 'self';
  script-src 'self' 'nonce-${nonce}' 'strict-
  connect-src 'self' https://www.google-analy
  img-src 'self' data: https://www.google-ana`
```

Common CSP Violations

- Inline styles:** Use CSS-in-JS libraries that support nonces or move styles to external files
- Dynamic imports:** Ensure dynamic imports are allowed in your script-src policy
- WebAssembly:** Add `'wasm-unsafe-eval'` if using WebAssembly

4. **Service workers:** Add appropriate policies for service worker scripts
-

Version History

Version	Changes
v14.0.0	Experimental SRI support added for hash-based CSP
v13.4.20	Recommended for proper nonce handling and CSP header parsing.

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to use CSS-in-JS libraries

► 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
}

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;
        }
      </style>
    </div>
  )
}
```

```
        }
        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 ↗](#).

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to set up a custom server in Next.js

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](#).
- 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:

TS server.ts TypeScript ▾

```
import { createServer } from 'http'
import { parse } from 'url'
import next from 'next'
```

```
const port = parseInt(process.env.PORT || '3000')
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:${
    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"  
  }  
}
```

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
<code>conf</code>	<code>Object</code>	The same object you would use in <code>next.config.js</code> . Defaults to <code>{}</code>
<code>dev</code>	<code>Boolean</code>	(Optional) Whether or not to launch Next.js in dev mode. Defaults to <code>false</code>
<code>dir</code>	<code>String</code>	(Optional) Location of the Next.js project. Defaults to <code>'.'</code>
<code>quiet</code>	<code>Boolean</code>	(Optional) Hide error messages containing server information. Defaults to <code>false</code>
<code>hostname</code>	<code>String</code>	(Optional) The hostname the server is running behind
<code>port</code>	<code>Number</code>	(Optional) The port the server is running behind
<code>httpServer</code>	<code>node:http#Server</code>	(Optional) The HTTP Server that Next.js is running behind
<code>turbo</code>	<code>Boolean</code>	(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:

```
JS next.config.js ✖  
  
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](#).

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to use debugging tools with Next.js

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:

```
launch.json
```

```
{  
  "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 (Fi
  "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_mod
  "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.

In the "Next.js: debug full stack" configuration, `serverReadyAction.action` specifies which browser to open when the server is ready. `debugWithEdge` means to launch the Edge

browser. If you are using Chrome, change this value to `debugWithChrome`.

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/_/`.

For React-specific debugging, install the [React Developer Tools](#) browser extension. This essential tool helps you:

- Inspect React components
- Edit props and state
- Identify performance problems

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
```

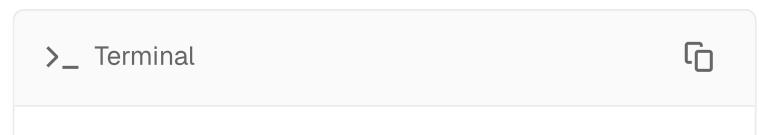
Good to know: Use `NODE_OPTIONS='--inspect=0.0.0.0'` to allow remote debugging access outside localhost, such as when running the app in a Docker container.

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:



```
NODE_OPTIONS='--inspect' next dev
```

```
Debugger listening on ws://127.0.0.1:9229/0cf
For help, see: https://nodejs.org/en/docs/ins
ready - started server on 0.0.0.0:3000, url:
```

For Chrome:

1. Open a new tab and visit `chrome://inspect`
2. Click **Configure...** to ensure both debugging ports are listed
3. Add both `localhost:9229` and `localhost:9230` if they're not already present
4. Look for your Next.js application in the **Remote Target** section
5. Click **inspect** to open a separate DevTools window
6. 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 icon underneath the Next.js version indicator on the error overlay. By clicking that icon, 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'  
  }  
}
```

`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 ↗](#)

Was this helpful?    



Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

How to preview content with Draft Mode in Next.js

In the [Pages documentation](#) and the [Data Fetching documentation](#), we talked about how to pre-render a page at build time (**Static Generation**) using `getStaticProps` and `getStaticPaths`.

Static Generation is useful when your pages fetch data from a headless CMS. However, it's not ideal when you're writing a draft on your headless CMS and want to view the draft immediately on your page. You'd want Next.js to render these pages at **request time** instead of build time and fetch the draft content instead of the published content. You'd want Next.js to bypass Static Generation only for this specific case.

Next.js has a feature called **Draft Mode** which solves this problem. Here are instructions on how to use it.

Step 1: Create and access the API route

Take a look at the [API Routes documentation](#) first if

First, create the **API route**. It can have any name -
e.g. `pages/api/draft.ts`

In this API route, you need to call `setDraftMode` on the response object.

```
export default function handler(req, res) {  
  // ...  
  res.setDraftMode({ enable: true })  
  // ...  
}
```

This will set a **cookie** to enable draft mode.

Subsequent requests containing this cookie will trigger **Draft Mode** changing the behavior for statically generated pages (more on this later).

You can test this manually by creating an API route like below and accessing it from your browser manually:



```
TS pages/api/draft.ts  
  
// simple example for testing it manually from browser  
export default function handler(req, res) {  
  res.setDraftMode({ enable: true })  
  res.end('Draft mode is enabled')  
}
```

If you open your browser's developer tools and visit `/api/draft`, you'll notice a `Set-Cookie` response header with a cookie named `--prerender_bypass`.

Securely accessing it from your Headless CMS

In practice, you'd want to call this API route *securely* from your headless CMS. The specific

steps will vary depending on which headless CMS you're using, but here are some common steps you could take.

These steps assume that the headless CMS you're using supports setting **custom draft URLs**. If it doesn't, you can still use this method to secure your draft URLs, but you'll need to construct and access the draft URL manually.

First, you should create a **secret token string** using a token generator of your choice. This secret will only be known by your Next.js app and your headless CMS. This secret prevents people who don't have access to your CMS from accessing draft URLs.

Second, if your headless CMS supports setting custom draft URLs, specify the following as the draft URL. This assumes that your draft API route is located at `pages/api/draft.ts`.



A screenshot of a terminal window titled "Terminal". Inside the terminal, the URL `https://<your-site>/api/draft?secret=<token>&` is displayed in purple text, indicating it is a placeholder or a command.

- `<your-site>` should be your deployment domain.
- `<token>` should be replaced with the secret token you generated.
- `<path>` should be the path for the page that you want to view. If you want to view `/posts/foo`, then you should use `&slug=/posts/foo`.

Your headless CMS might allow you to include a variable in the draft URL so that `<path>` can be set dynamically based on the CMS's data like so: `&slug=/posts/{entry.fields.slug}`

Finally, in the draft API route:

- Check that the secret matches and that the `slug` parameter exists (if not, the request should fail).
- Call `res.setDraftMode`.
- Then redirect the browser to the path specified by `slug`. (The following example uses a [307 redirect ↗](#)).

```
export default async (req, res) => {
  // Check the secret and next parameters
  // This secret should only be known to this
  if (req.query.secret !== 'MY_SECRET_TOKEN')
    return res.status(401).json({ message: 'I
  }

  // Fetch the headless CMS to check if the p
  // getPostBySlug would implement the requir
  const post = await getPostBySlug(req.query.

  // If the slug doesn't exist prevent draft
  if (!post) {
    return res.status(401).json({ message: 'I
  }

  // Enable Draft Mode by setting the cookie
  res.setDraftMode({ enable: true })

  // Redirect to the path from the fetched po
  // We don't redirect to req.query.slug as t
  res.redirect(post.slug)
}
```

If it succeeds, then the browser will be redirected to the path you want to view with the draft mode cookie.

Step 2: Update `getStaticProps`

The next step is to update `getStaticProps` to support draft mode.

If you request a page which has `getStaticProps` with the cookie set (via `res.setDraftMode`), then `getStaticProps` will be called at **request time** (instead of at build time).

Furthermore, it will be called with a `context` object where `context.draftMode` will be `true`.

```
export async function getStaticProps(context)
  if (context.draftMode) {
    // dynamic data
  }
}
```

We used `res.setDraftMode` in the draft API route, so `context.draftMode` will be `true`.

If you're also using `getStaticPaths`, then `context.params` will also be available.

Fetch draft data

You can update `getStaticProps` to fetch different data based on `context.draftMode`.

For example, your headless CMS might have a different API endpoint for draft posts. If so, you can modify the API endpoint URL like below:

```
export async function getStaticProps(context)
  const url = context.draftMode
    ? 'https://draft.example.com'
    : 'https://production.example.com'
  const res = await fetch(url)
  // ...
}
```

That's it! If you access the draft API route (with `secret` and `slug`) from your headless CMS or

manually, you should now be able to see the draft content. And if you update your draft without publishing, you should be able to view the draft.

Set this as the draft URL on your headless CMS or access manually, and you should be able to see the draft.

```
>_ Terminal ┌ └  
https://<your-site>/api/draft?secret=<token>&
```

More Details

Clear the Draft Mode cookie

By default, the Draft Mode session ends when the browser is closed.

To clear the Draft Mode cookie manually, create an API route that calls

```
setDraftMode({ enable: false }):
```

```
TS pages/api/disable-draft.ts ┌ └  
export default function handler(req, res) {  
  res.setDraftMode({ enable: false })  
}
```

Then, send a request to `/api/disable-draft` to invoke the API Route. If calling this route using `next/link`, you must pass `prefetch={false}` to prevent accidentally deleting the cookie on prefetch.

Works with `getServerSideProps`

Draft Mode works with `getServerSideProps`, and is available as a `draftMode` key in the `context` object.

Good to know: You shouldn't set the `Cache-Control` header when using Draft Mode because it cannot be bypassed. Instead, we recommend using [ISR](#).

Works with API Routes

API Routes will have access to `draftMode` on the `request` object. For example:

```
export default function myApiRoute(req, res)
  if (req.draftMode) {
    // get draft data
  }
}
```

Unique per `next build`

A new bypass cookie value will be generated each time you run `next build`.

This ensures that the bypass cookie can't be guessed.

Good to know: To test Draft Mode locally over HTTP, your browser will need to allow third-party cookies and local storage access.

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to use environment variables in Next.js

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_`

Warning: The default `create-next-app` template ensures all `.env` files are added to your `.gitignore`. You almost never want to commit these files to your repository.

Loading Environment Variables

Next.js has built-in support for loading environment variables from `.env*` files into `process.env`.

```
 .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`:

```
JS pages/index.js Copy  
  
export async function getStaticProps() {  
  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
```

```
TS envConfig.ts TypeScript ▾ Copy
```

```
import { loadEnvConfig } from '@next/env'

const projectDir = process.cwd()
loadEnvConfig(projectDir)
```

Then, you can import the configuration where needed. For example:



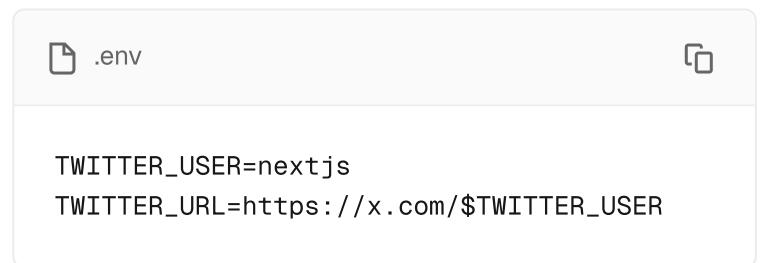
```
TS orm.config.ts TypeScript ▾
```

```
import './envConfig.ts'

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:



```
.env
```

```
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:

```
>_ Terminal ┌─────────┐  
NEXT_PUBLIC_ANALYTICS_ID=abcdefghijklm ┘
```

This will tell Next.js to replace all references to `process.env.NEXT_PUBLIC_ANALYTICS_ID` in the Node.js environment with the value from the environment in which you run `next build`, allowing you to use it anywhere in your code. It will be inlined into any JavaScript sent to the browser.

Note: After being built, your app will no longer respond to changes to these environment variables. For instance, if you use a Heroku pipeline to promote slugs built in one environment to another environment, or if you build and deploy a single Docker image to multiple environments, all `NEXT_PUBLIC_` variables will be frozen with the value evaluated at build time, so these values need to be set appropriately when the project is built. If you need access to runtime environment values, you'll have to setup your own API to provide them to the client (either on demand or during initialization).

```
js pages/index.js ┌─────────┐  
import setupAnalyticsService from '../lib/my- ┘
```

```
// 'NEXT_PUBLIC_ANALYTICS_ID' can be used here  
// It will be transformed at build time to `setupAnalyticsService(process.env.NEXT_PUBLIC  
  
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  
const varName = 'NEXT_PUBLIC_ANALYTICS_ID'  
setupAnalyticsService(process.env[varName])  
  
// This will NOT be inlined, because it uses  
const env = process.env  
setupAnalyticsService(env.NEXT_PUBLIC_ANALYTIC
```

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](#).

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 if you need this feature.
-

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
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 <code>.env</code> and <code>NEXT_PUBLIC_</code> introduced.
--------	---

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to create forms with API Routes

Forms enable you to create and update data in web applications. Next.js provides a powerful way to handle data mutations using **API Routes**. This guide will walk you through how to handle form submission on the server.

Server Forms

To handle form submissions on the server, create an API endpoint that securely mutates data.

The code editor shows a file named `pages/api/submit.ts` with the following content:

```
import type { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const data = req.body
  const id = await createItem(data)
  res.status(200).json({ id })
}
```

Then, call the API Route from the client with an event handler:



```
import { FormEvent } from 'react'

export default function Page() {
  async function onSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault()

    const formData = new FormData(event.currentTarget)
    const response = await fetch('/api/submit', {
      method: 'POST',
      body: formData,
    })

    // Handle response if necessary
    const data = await response.json()
    // ...
  }

  return (
    <form onSubmit={onSubmit}>
      <input type="text" name="name" />
      <button type="submit">Submit</button>
    </form>
  )
}
```

Good to know:

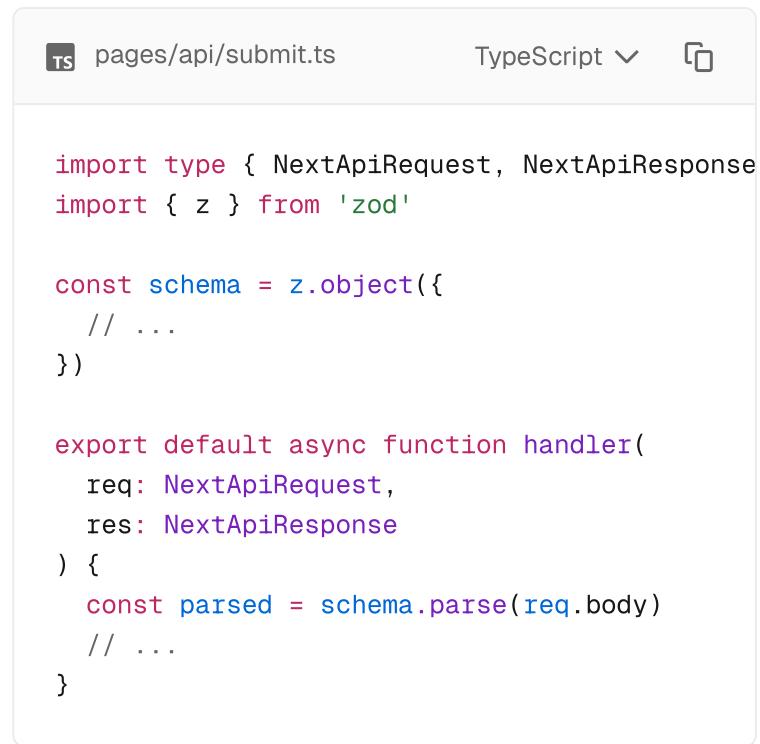
- API Routes [do not specify CORS headers ↗](#), meaning they are same-origin only by default.
- Since API Routes run on the server, we're able to use sensitive values (like API keys) through [Environment Variables](#) without exposing them to the client. This is critical for the security of your application.

Form validation

We recommend using HTML validation like `required` and `type="email"` for basic client-side form validation.

For more advanced server-side validation, you can use a schema validation library like [zod ↗](#) to

validate the form fields before mutating the data:



pages/api/submit.ts TypeScript

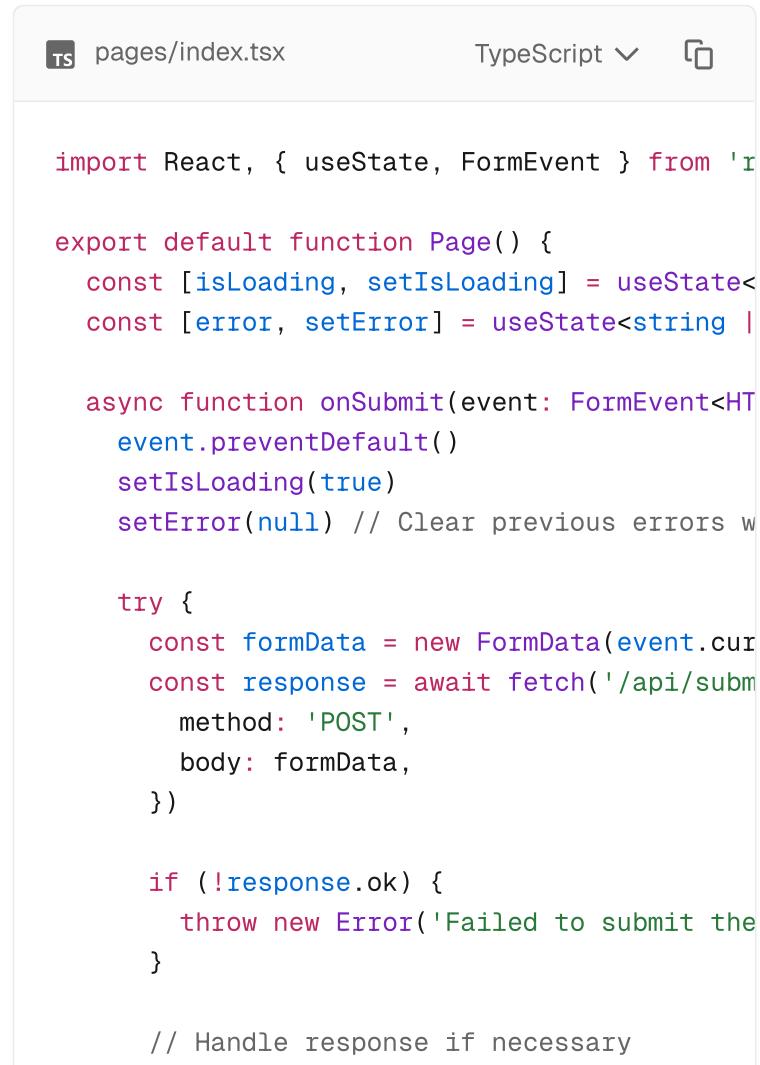
```
import type { NextApiRequest, NextApiResponse } from 'zod'

const schema = z.object({
  // ...
})

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const parsed = schema.parse(req.body)
  // ...
}
```

Error handling

You can use React state to show an error message when a form submission fails:



pages/index.tsx TypeScript

```
import React, { useState, FormEvent } from 'react'

export default function Page() {
  const [isLoading, setIsLoading] = useState(false)
  const [error, setError] = useState<string>('')

  async function onSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault()
    setIsLoading(true)
    setError(null) // Clear previous errors when starting a new submission

    try {
      const formData = new FormData(event.currentTarget)
      const response = await fetch('/api/submit', {
        method: 'POST',
        body: formData,
      })
      if (!response.ok) {
        throw new Error('Failed to submit the form')
      }
    } catch (err) {
      setError(err.message)
    }
  }

  return (
    <div>
      <h1>Submit a form</h1>
      <form onSubmit={onSubmit}>
        <input type="text" name="name" />
        <input type="text" name="email" />
        <button type="submit">Submit</button>
      </form>
      {error ? <p>{error}</p> : null}
    </div>
  )
}
```

```

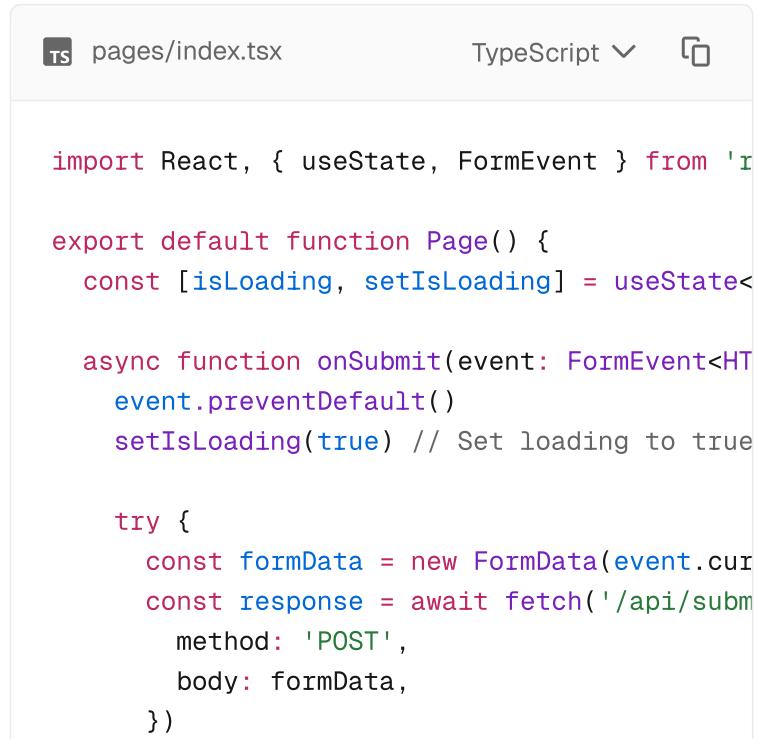
        const data = await response.json()
        // ...
    } catch (error) {
        // Capture the error message to display
        setError(error.message)
        console.error(error)
    } finally {
        setIsLoading(false)
    }
}

return (
    <div>
        {error && <div style={{ color: 'red' }}>
            <form onSubmit={onSubmit}>
                <input type="text" name="name" />
                <button type="submit" disabled={isLoa
                    isLoading ? 'Loading...' : 'Submit'
                </button>
            </form>
        </div>
    )
}

```

Displaying loading state

You can use React state to show a loading state when a form is submitting on the server:



The screenshot shows a code editor interface with a TypeScript file named `pages/index.tsx`. The code implements a `Page` component that handles form submission and displays a loading state.

```

import React, { useState, FormEvent } from 'react'

export default function Page() {
    const [isLoading, setIsLoading] = useState(false)

    async function onSubmit(event: FormEvent<HTMLFormElement>)
        event.preventDefault()
        setIsLoading(true) // Set loading to true

        try {
            const formData = new FormData(event.currentTarget)
            const response = await fetch('/api/submit', {
                method: 'POST',
                body: formData,
            })
        }
    }
}

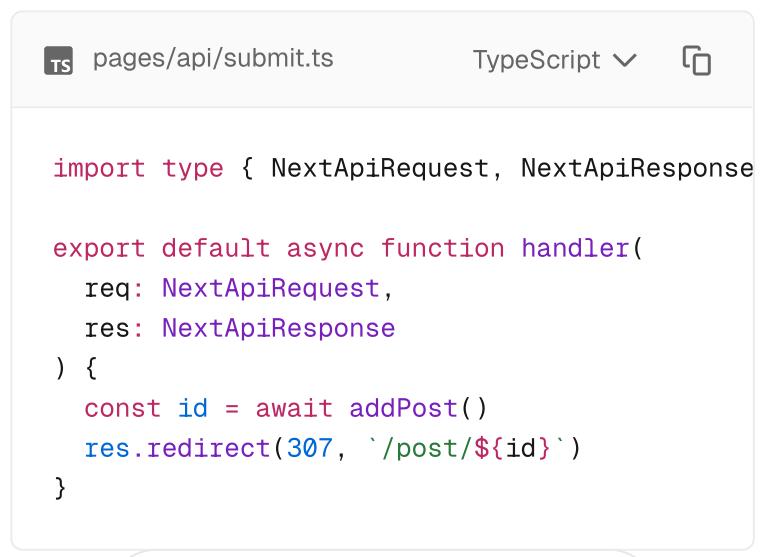
```

```
// Handle response if necessary
const data = await response.json()
// ...
} catch (error) {
  // Handle error if necessary
  console.error(error)
} finally {
  setIsLoading(false) // Set loading to false
}
}

return (
  <form onSubmit={onSubmit}>
    <input type="text" name="name" />
    <button type="submit" disabled={isLoading}><span>{isLoading ? 'Loading...' : 'Submit'}`</span></button>
  </form>
)
}
```

Redirecting

If you would like to redirect the user to a different route after a mutation, you can `redirect` to any absolute or relative URL:



```
pages/api/submit.ts  TypeScript ▾
```

```
import type { NextApiRequest, NextApiResponse }

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const id = await addPost()
  res.redirect(307, `/post/${id}`)
}
```

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to implement Incremental Static Regeneration (ISR)

► 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:

```
TS pages/blog/[id].tsx TypeScript ▾ ⌂  
  
import type { GetStaticPaths, GetStaticProps }  
  
interface Post {  
  id: string  
  title: string  
  content: string  
}  
  
interface Props {  
  post: Post  
}
```

```

        export const getStaticPaths: GetStaticPaths = () => {
          const posts = await fetch('https://api.veralabs.com/posts')
          .then(res => res.json())
        }
        const paths = posts.map((post: Post) => ({
          params: { id: String(post.id) },
        }))
      }

      return { paths, fallback: 'blocking' }
    }

    export const getStaticProps: GetStaticProps<Post> = async ({ params }) => {
      const post = await fetch(`https://api.veralabs.com/posts/${params.id}`)
      .then(res => res.json())
    }

    return {
      props: { post },
      // Next.js will invalidate the cache when
      // 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>
    )
  }
}

```

Here's how this example works:

1. During `next build`, all known blog posts are generated
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 return the cached (now stale) page
4. The cache is invalidated and a new version of the page begins generating in the background

- Once generated successfully, the next request will return the updated page and cache it for subsequent requests
 - If `/blog/26` is requested, and it exists, the page will be generated on-demand. This behavior can be changed by using a different `fallback` value. However, if the post does not exist, then 404 is returned.
-

Reference

Functions

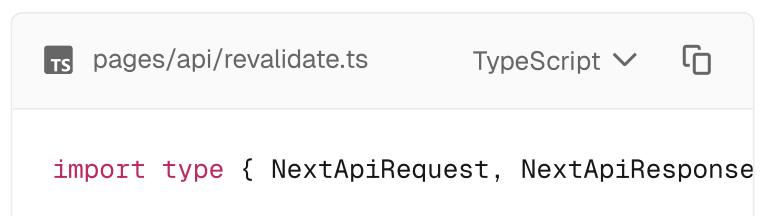
- `getStaticProps`
 - `res.revalidate`
-

Examples

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.



```
TS pages/api/revalidate.ts TypeScript ▾ ⌂
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)
            return res.status(401).json({ message: 'Unauthorized' })

          try {
            // This should be the actual path not a regular route
            // 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 consider it
            // 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 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<P
  params,
>: {
  params: { id: string }
}) => {
  // If this request throws an uncaught error
  // not invalidate the currently shown page
  // retry getStaticProps on the next request
  const res = await fetch(`https://api.vercel
  const post: Post = await res.json()

  if (!res.ok) {
    // If there is a server error, you might
    // throw an error instead of returning so
    // until the next successful request.
    throw new Error(`Failed to fetch posts, r
  }

  return {
    props: { post },
    // Next.js will invalidate the cache when
    // request comes in, at most once every 6
    revalidate: 60,
  }
}
```

Customizing the cache location

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.](#)



A screenshot of a code editor showing the file `next.config.js`. The code defines a module export object with a `logging` key. The `logging` object contains a `fetches` key, which is set to an object with a `fullUrl` key set to `true`.

```
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:



A screenshot of a code editor showing the file `.env`. It contains the environment variable `NEXT_PRIVATE_DEBUG_CACHE=1`.

```
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](#).
- 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`.

Platform Support

Deployment Option	Supported
Node.js server	Yes
Docker container	Yes
Static export	No
Adapters	Platform-specific

Learn how to [configure ISR](#) when self-hosting Next.js.

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](#).

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to set up instrumentation

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](#) ↗ :

```
TS instrumentation.ts TypeScript ▾
```

```
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:



instrumentation.ts

TypeScript ▾



```
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:



```
TS instrumentation.ts TypeScript ▾
```

```
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs')
    await import('./instrumentation-node')
  }

  if (process.env.NEXT_RUNTIME === 'edge')
    await import('./instrumentation-edge')
}
```

Was this helpful?    



Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

How to implement internationalization in Next.js

► Examples

Next.js has built-in support for internationalized ([i18n ↗](#)) routing since `v10.0.0`. You can provide a list of locales, the default locale, and domain-specific locales and Next.js will automatically handle the routing.

The i18n routing support is currently meant to complement existing i18n library solutions like [react-intl ↗](#), [react-i18next ↗](#), [lingui ↗](#), [rosetta ↗](#), [next-intl ↗](#), [next-translate ↗](#), [next-multilingual ↗](#), [tolgee ↗](#), [paraglide-next ↗](#), [next-intlayer ↗](#) and others by streamlining the routes and locale parsing.

Getting started

To get started, add the `i18n` config to your `next.config.js` file.

Locales are [UTS Locale Identifiers ↗](#), a standardized format for defining locales.

Generally a Locale Identifier is made up of a language, region, and script separated by a dash: `language-region-script`. The region and script are optional. An example:

- `en-US` - English as spoken in the United States
- `nl-NL` - Dutch as spoken in the Netherlands
- `nl` - Dutch, no specific region

If user locale is `nl-BE` and it is not listed in your configuration, they will be redirected to `nl` if available, or to the default locale otherwise. If you don't plan to support all regions of a country, it is therefore a good practice to include country locales that will act as fallbacks.



A screenshot of a code editor showing a file named `next.config.js`. The code defines an `i18n` object with various properties and nested objects for different domains and subdomains, specifying locales and default locales.

```
module.exports = {
  i18n: {
    // These are all the locales you want to
    // your application
    locales: ['en-US', 'fr', 'nl-NL'],
    // This is the default locale you want to
    // a non-locale prefixed path e.g. `/hell
    defaultLocale: 'en-US',
    // This is a list of locale domains and t
    // should handle (these are only required
    // Note: subdomains must be included in t
    domains: [
      {
        domain: 'example.com',
        defaultLocale: 'en-US',
      },
      {
        domain: 'example.nl',
        defaultLocale: 'nl-NL',
      },
      {
        domain: 'example.fr',
        defaultLocale: 'fr',
        // an optional http field can also be
        // locale domains locally with http i
        http: true,
      },
    ],
  },
};
```

```
}
```

Locale Strategies

There are two locale handling strategies: Sub-path Routing and Domain Routing.

Sub-path Routing

Sub-path Routing puts the locale in the url path.

```
JS next.config.js
```

```
module.exports = {
  i18n: {
    locales: ['en-US', 'fr', 'nl-NL'],
    defaultLocale: 'en-US',
  },
}
```

With the above configuration `en-US`, `fr`, and `nl-NL` will be available to be routed to, and `en-US` is the default locale. If you have a `pages/blog.js` the following urls would be available:

- `/blog`
- `/fr/blog`
- `/nl-nl/blog`

The default locale does not have a prefix.

Domain Routing

By using domain routing you can configure locales to be served from different domains:

```
JS next.config.js
```

```

module.exports = {
  i18n: {
    locales: ['en-US', 'fr', 'nl-NL', 'nl-BE'],
    defaultLocale: 'en-US',
  },
  domains: [
    {
      // Note: subdomains must be included
      // e.g. www.example.com should be used
      domain: 'example.com',
      defaultLocale: 'en-US',
    },
    {
      domain: 'example.fr',
      defaultLocale: 'fr',
    },
    {
      domain: 'example.nl',
      defaultLocale: 'nl-NL',
      // specify other locales that should
      // go to this domain
      locales: ['nl-BE'],
    },
  ],
},
}

```

For example if you have `pages/blog.js` the following urls will be available:

- `example.com/blog`
- `www.example.com/blog`
- `example.fr/blog`
- `example.nl/blog`
- `example.nl/nl-BE/blog`

Automatic Locale Detection

When a user visits the application root (generally `/`), Next.js will try to automatically detect which locale the user prefers based on the

`Accept-Language` ↗ header and the current domain.

If a locale other than the default locale is detected, the user will be redirected to either:

- **When using Sub-path Routing:** The locale prefixed path
- **When using Domain Routing:** The domain with that locale specified as the default

When using Domain Routing, if a user with the

`Accept-Language` header `fr;q=0.9` visits `example.com`, they will be redirected to `example.fr` since that domain handles the `fr` locale by default.

When using Sub-path Routing, the user would be redirected to `/fr`.

Prefixing the Default Locale

With Next.js 12 and [Middleware](#), we can add a prefix to the default locale with a [workaround](#) ↗.

For example, here's a `next.config.js` file with support for a few languages. Note the `"default"` locale has been added intentionally.

```
JS next.config.js Copy  
  
module.exports = {  
  i18n: {  
    locales: ['default', 'en', 'de', 'fr'],  
    defaultLocale: 'default',  
    localeDetection: false,  
  },  
  trailingSlash: true,  
}
```

Next, we can use [Middleware](#) to add custom routing rules:

```
import { NextRequest, NextResponse } from 'next'

const PUBLIC_FILE = /\.(\.*$)/

export async function middleware(req: NextRequest) {
  if (
    req.nextUrl.pathname.startsWith('/_next')
    req.nextUrl.pathname.includes('/api/') ||
    PUBLIC_FILE.test(req.nextUrl.pathname)
  ) {
    return
  }

  if (req.nextUrl.locale === 'default') {
    const locale = req.cookies.get('NEXT_LOCA
      return NextResponse.redirect(
        new URL(`/${locale}${req.nextUrl.pathname}`))
    }
  }
}
```

This [Middleware](#) skips adding the default prefix to [API Routes](#) and [public](#) files like fonts or images. If a request is made to the default locale, we redirect to our prefix `/en`.

Disabling Automatic Locale Detection

The automatic locale detection can be disabled with:

```
module.exports = {
  i18n: {
    localeDetection: false,
  },
}
```

When `localeDetection` is set to `false` Next.js will no longer automatically redirect based on the

user's preferred locale and will only provide locale information detected from either the locale based domain or locale path as described above.

Accessing the locale information

You can access the locale information via the Next.js router. For example, using the

`useRouter()` hook the following properties are available:

- `locale` contains the currently active locale.
- `locales` contains all configured locales.
- `defaultLocale` contains the configured default locale.

When [pre-rendering](#) pages with `getStaticProps` or `getServerSideProps`, the locale information is provided in [the context](#) provided to the function.

When leveraging `getStaticPaths`, the configured locales are provided in the context parameter of the function under `locales` and the configured defaultLocale under `defaultLocale`.

Transition between locales

You can use `next/link` or `next/router` to transition between locales.

For `next/link`, a `locale` prop can be provided to transition to a different locale from the currently active one. If no `locale` prop is provided, the

currently active `locale` is used during client-transitions. For example:

```
import Link from 'next/link'

export default function IndexPage(props) {
  return (
    <Link href="/another" locale="fr">
      To /fr/another
    </Link>
  )
}
```

When using the `next/router` methods directly, you can specify the `locale` that should be used via the transition options. For example:

```
import { useRouter } from 'next/router'

export default function IndexPage(props) {
  const router = useRouter()

  return (
    <div
      onClick={() => {
        router.push('/another', '/another', {
          locale: 'fr'
        })
      }}
    >
      to /fr/another
    </div>
  )
}
```

Note that to handle switching only the `locale` while preserving all routing information such as `dynamic route` query values or hidden href query values, you can provide the `href` parameter as an object:

```
import { useRouter } from 'next/router'
const router = useRouter()
const { pathname, asPath, query } = router
// change just the locale and maintain all other
// query values
router.push({ pathname, query }, asPath, { locale: 'fr' })
```

See [here](#) for more information on the object structure for `router.push`.

If you have a `href` that already includes the locale you can opt-out of automatically handling the locale prefixing:

```
import Link from 'next/link'

export default function IndexPage(props) {
  return (
    <Link href="/fr/another" locale={false}>
      To /fr/another
    </Link>
  )
}
```

Leveraging the `NEXT_LOCALE` cookie

Next.js allows setting a `NEXT_LOCALE=the-locale` cookie, which takes priority over the accept-language header. This cookie can be set using a language switcher and then when a user comes back to the site it will leverage the locale specified in the cookie when redirecting from `/` to the correct locale location.

For example, if a user prefers the locale `fr` in their accept-language header but a `NEXT_LOCALE=en` cookie is set the `en` locale when visiting `/` the user will be redirected to the `en` locale location until the cookie is removed or expired.

Search Engine Optimization

Since Next.js knows what language the user is visiting it will automatically add the `lang` attribute to the `<html>` tag.

Next.js doesn't know about variants of a page so it's up to you to add the `hreflang` meta tags using `next/head`. You can learn more about `hreflang` in the [Google Webmasters documentation ↗](#).

How does this work with Static Generation?

Note that Internationalized Routing does not integrate with `output: 'export'` as it does not leverage the Next.js routing layer. Hybrid Next.js applications that do not use `output: 'export'` are fully supported.

Dynamic Routes and `getStaticProps` Pages

For pages using `getStaticProps` with [Dynamic Routes](#), all locale variants of the page desired to be prerendered need to be returned from `getStaticPaths`. Along with the `params` object returned for `paths`, you can also return a `locale` field specifying which locale you want to render.

For example:

```
JS pages/blog/[slug].js

export const getStaticPaths = ({ locales }) =>
  return {
    paths: [
      // if no `locale` is provided only the
      { params: { slug: 'post-1' }, locale: 'en' },
      { params: { slug: 'post-1' }, locale: 'fr' },
    ],
    fallback: true,
  }
```

}

For [Automatically Statically Optimized](#) and non-dynamic `getStaticProps` pages, a **version of the page will be generated for each locale**. This is important to consider because it can increase build times depending on how many locales are configured inside `getStaticProps`.

For example, if you have 50 locales configured with 10 non-dynamic pages using

`getStaticProps`, this means `getStaticProps` will be called 500 times. 50 versions of the 10 pages will be generated during each build.

To decrease the build time of dynamic pages with `getStaticProps`, use a [fallback mode](#). This allows you to return only the most popular paths and locales from `getStaticPaths` for prerendering during the build. Then, Next.js will build the remaining pages at runtime as they are requested.

Automatically Statically Optimized Pages

For pages that are [automatically statically optimized](#), a version of the page will be generated for each locale.

Non-dynamic `getStaticProps` Pages

For non-dynamic `getStaticProps` pages, a version is generated for each locale like above.

`getStaticProps` is called with each `locale` that is being rendered. If you would like to opt-out of a certain locale from being pre-rendered, you can return `notFound: true` from `getStaticProps` and this variant of the page will not be generated.

```
export async function getStaticProps({ locale
  // Call an external API endpoint to get posts
  // You can use any data fetching library
  const res = await fetch(`https://.../posts?`)
  const posts = await res.json()

  if (posts.length === 0) {
    return {
      notFound: true,
    }
  }

  // By returning { props: posts }, the Blog
  // will receive `posts` as a prop at build
  return {
    props: {
      posts,
    },
  }
}
```

Limits for the i18n config

- locales : 100 total locales
- domains : 100 total locale domain items

Good to know: These limits have been added initially to prevent potential [performance issues at build time](#). You can workaround these limits with custom routing using [Middleware](#) in Next.js 12.

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to lazy load Client Components and libraries

[Lazy loading ↗](#) in Next.js helps improve the initial loading performance of an application by decreasing the amount of JavaScript needed to render a route.

It allows you to defer loading of **Client Components** and imported libraries, and only include them in the client bundle when they're needed. For example, you might want to defer loading a modal until a user clicks to open it.

There are two ways you can implement lazy loading in Next.js:

1. Using [Dynamic Imports](#) with `next/dynamic`
2. Using `React.lazy()` ↗ with [Suspense ↗](#)

By default, Server Components are automatically [code split ↗](#), and you can use [streaming](#) to progressively send pieces of UI from the server to the client. Lazy loading applies to Client Components.

`next/dynamic`

`next/dynamic` is a composite of `React.lazy()` [↑](#) and `Suspense` [↑](#). It behaves the same way in the `app` and `pages` directories to allow for incremental migration.

Examples

By using `next/dynamic`, the header component will not be included in the page's initial JavaScript bundle. The page will render the `Suspense` `fallback` first, followed by the `Header` component when the `Suspense` boundary is resolved.

```
import dynamic from 'next/dynamic'

const DynamicHeader = dynamic(() => import('.
  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()`

↗ :

JS components/hello.js



```
export function Hello() {
  return <p>Hello!</p>
}

// pages/index.js
import dynamic from 'next/dynamic'

const DynamicComponent = dynamic(() =>
  import('../components/hello').then((mod) =>
)
```

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('.
  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'))
          const fuse = new Fuse(names)

          setResults(fuse.search(value))
        }}
      />
      <pre>Results: {JSON.stringify(results,
      </div>
    )
  }
}
```

Was this helpful?    



Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to use mark-down and MDX in Next.js

[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="htt
```

[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:

> Terminal



```
npm install @next-mdx @mdx-js/loader @mdx-js/
```

Configure `next.config.mjs`

Update the `next.config.mjs` file at your project's root to configure it to use MDX:

JS next.config.mjs



```
import createMDX from '@next-mdx'

/** @type {import('next').NextConfig} */
const nextConfig = {
  // Configure `pageExtensions` to include markdown
  pageExtensions: ['js', 'jsx', 'md', 'mdx'],
  // Optionally, add any other Next.js config
}
```

```
const withMDX = createMDX({
  // Add markdown plugins here, as desired
})

// Merge MDX config with Next.js config
export default withMDX(nextConfig)
```

This allows `.mdx` files to act as pages, routes, or imports in your application.

Handling `.md` files

By default, `next/mdx` only compiles files with the `.mdx` extension. To handle `.md` files with webpack, update the `extension` option:

`JS` next.config.mjs 

```
const withMDX = createMDX({
  extension: /\.(md|mdx)$/,
})
```

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.

`TS` mdx-components.tsx  

```
import type { MDXComponents } from 'mdx/types'

const components: MDXComponents = {}

export function useMDXComponents(): MDXCompon
    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.

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 `/pages` directory and an MDX file wherever you'd like:

```
.
├── markdown/
│   └── welcome.(mdx/md)
└── pages/
    └── mdx-page.(tsx/js)
    └── mdx-components.(tsx/js)
    └── package.json
```

You can use MDX in these files, and even import React components, directly inside your MDX page:

Import the MDX file inside the page to display the content:

TS pages/mdl-page.tsx TypeScript ▾ ⌂

```
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>
```

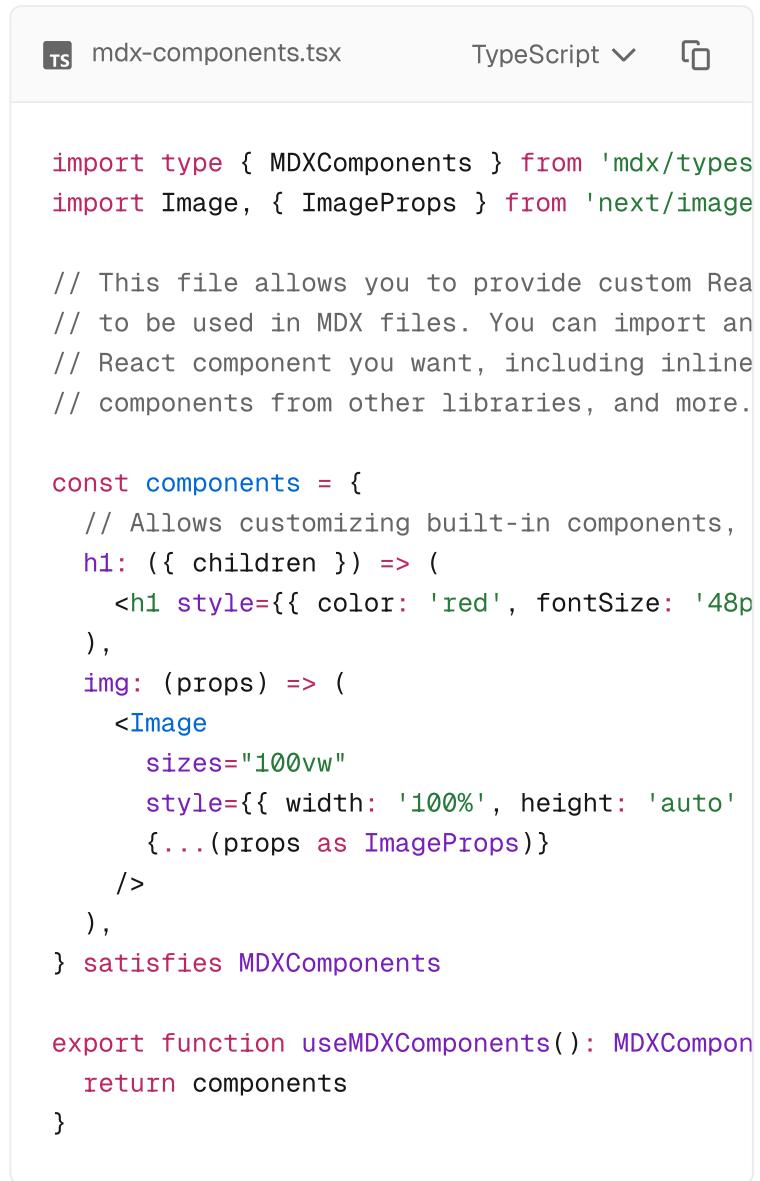
To style your markdown, you can provide custom components that map to the generated HTML elements. Styles and components can be

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.



```
TS  mdx-components.tsx  TypeScript ▾  ⌂
import type { MDXComponents } from 'mdx/types'
import Image, { ImageProps } from 'next/image'

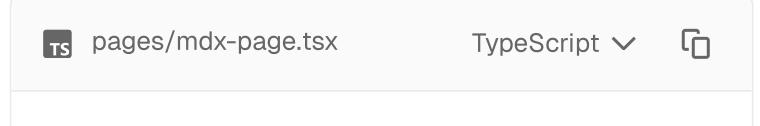
// This file allows you to provide custom React components to be used in MDX files. You can import an
// React component you want, including inline components from other libraries, and more.

const components = {
  // Allows customizing built-in components,
  h1: ({ children }) => (
    <h1 style={{ color: 'red', fontSize: '48px' }}>{children}</h1>
  ),
  img: (props) => (
    <Image
      sizes="100vw"
      style={{ width: '100%', height: 'auto' }}
      {...(props as ImageProps)}
    />
  ),
} satisfies MDXComponents

export function useMDXComponents(): MDXComponents {
  return 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](#).



```
TS  pages/mdx-page.tsx  TypeScript ▾  ⌂
import { MDXPage } from 'mdx'

const components = {
  h1: ({ children }) => (
    <h1 style={{ color: 'blue', fontSize: '32px' }}>{children}</h1>
  ),
}

MDXPage({ components })
```

```
import Welcome from '@/markdown/welcome.mdx'

function CustomH1({ children }) {
  return <h1 style={{ color: 'blue', fontSize }}>
}

const overrideComponents = {
  h1: CustomH1,
}

export default function Page() {
  return <Welcome components={overrideComponents}>
}
```

Shared layouts

To share a layout around MDX pages, create a layout component:

```
TS components/mdx-layout.tsx TypeScript ▾ ⌂

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.

To share a layout around MDX pages, create a layout component:



```
TS components/mdx-layout.tsx TypeScript ▾ ⌂

export default function MdxLayout({ children
  // Create any shared layout or styles here
  return (
    <div className="prose prose-headings:mt-8"
      {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-layo

# Welcome to my MDX page!

export default function MDXPage({ children })
  return <MdxLayout>{children}</MdxLayout>

}
```

Frontmatter

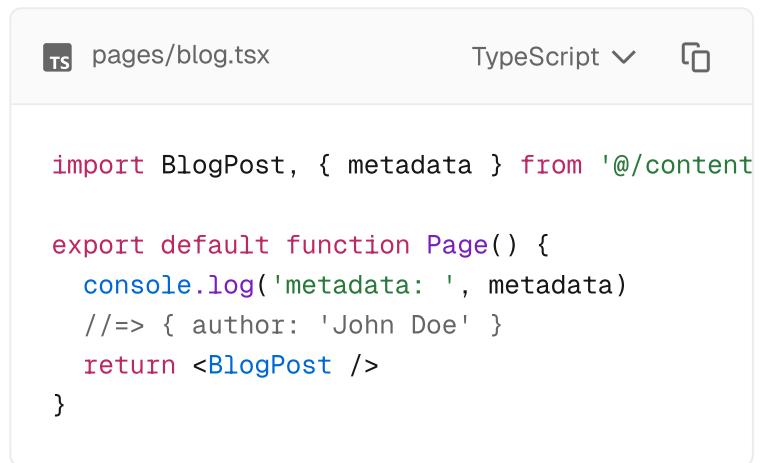
Frontmatter is a YAML like key/value pairing that can be used to store data about a page.

`@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:

Metadata can now be referenced outside of the MDX file:



```
pages/blog.tsx
TypeScript ▾
```

```
import BlogPost, { metadata } from '@/content'

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` or `next.config.ts` as the configuration file.

```
JS next.config.mjs

import remarkGfm from 'remark-gfm'
import createMDX from '@next/mdx'

/** @type {import('next').NextConfig} */
const nextConfig = {
    // Allow .mdx extensions for files
    pageExtensions: ['js', 'jsx', 'md', 'mdx'],
    // Optionally, add any other Next.js config
}

const withMDX = createMDX({
    // Add markdown plugins here, as desired
    options: {
        remarkPlugins: [remarkGfm],
        rehypePlugins: [],
    },
})

// Combine MDX and Next.js config
export default withMDX(nextConfig)
```

Using Plugins with Turbopack

To use plugins with [Turbopack](#), upgrade to the latest `@next/mdx` and specify plugin names using a string:

```
JS next.config.mjs
```

```
import createMDX from '@next/mdx'

/** @type {import('next').NextConfig} */
const nextConfig = {
  pageExtensions: ['js', 'jsx', 'md', 'mdx'],
}

const withMDX = createMDX({
  options: {
    remarkPlugins: [
      // Without options
      'remark-gfm',
      // With options
      ['remark-toc', { heading: 'The Table' }],
    ],
    rehypePlugins: [
      // Without options
      'rehype-slug',
      // With options
      ['rehype-katex', { strict: true, throwOnWarning: false }],
    ],
  },
})

export default withMDX(nextConfig)
```

Good to know:

remark and rehype plugins without serializable options cannot be used yet with [TurboPack](#), because JavaScript functions can't be passed to Rust.

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 CMS, database, or anywhere else. A community package for this use is [next-mdx-remote-client](#).

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-client :

```
  TS pages/mdx-page-remote.tsx TypeScript ▾ ⌂
import {
  serialize,
  type SerializeResult,
} from 'next-mdx-remote-client/serialize'
import { MDXClient } from 'next-mdx-remote-cl

type Props = {
  mdxSource: SerializeResult
}

export default function RemoteMdxPage({ mdxSo
  if ('error' in mdxSource) {
    // either render error UI or throw `mdxSo
  }
  return <MDXClient {...mdxSource} />
}

export async function getStaticProps() {
  // MDX text - can be from a database, CMS,
  const res = await fetch('https:....')
  const mdxText = await res.text()
  const mdxSource = await serialize({ source:
  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
    .use(remarkRehype) // Transform to HTML AST
    .use(rehypeSanitize) // Sanitize HTML input
    .use(rehypeStringify) // Convert AST into string
    .process('Hello, Next.js!')

  console.log(String(file)) // <p>Hello, Next.js!
}
```

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`:

JS next.config.js

2

```
module.exports = withMDX({
  experimental: {
    mdxRs: true,
  },
})
```

`mdxRs` also accepts an object to configure how to transform mdx files.

 next.config.js

6

```
module.exports = withMDX({
  experimental: {
    mdxRs: {
      jsxRuntime?: string // Custo
      jsxImportSource?: string // Custo
      mdxType?: 'gfm' | 'commonmark' // Config
    },
  },
})
```

Helpful Links

- MDX ↗
 - `@next/mdi` ↗
 - remark ↗
 - rehype ↗
 - Markdoc ↗

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



Migrating

App Router

Learn how to upgrade your existing Next.js...

Create React...

Learn how to migrate your existing React...

Vite

Learn how to migrate your existing React...

Was this helpful?



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to migrate from Pages 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:

>_ Terminal



```
npm install next@latest react@latest react-do
```

ESLint Version

If you're using ESLint, you need to upgrade your ESLint version:

> Terminal



```
npm install -D eslint-config-next@latest
```

Good to know: You may need to restart the ESLint server in VS Code for the ESLint changes to take effect. Open the Command Palette (`cmd+shift+p` on Mac; `ctrl+shift+p` on Windows) and search for `ESLint: Restart ESLint Server`.

Next Steps

After you've updated, see the following sections for next steps:

- [Upgrade new features](#): 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 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
<Link href="/about">
  <a>About</a>
</Link>

// Next.js 13: `<Link>` always renders `<a>`
<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

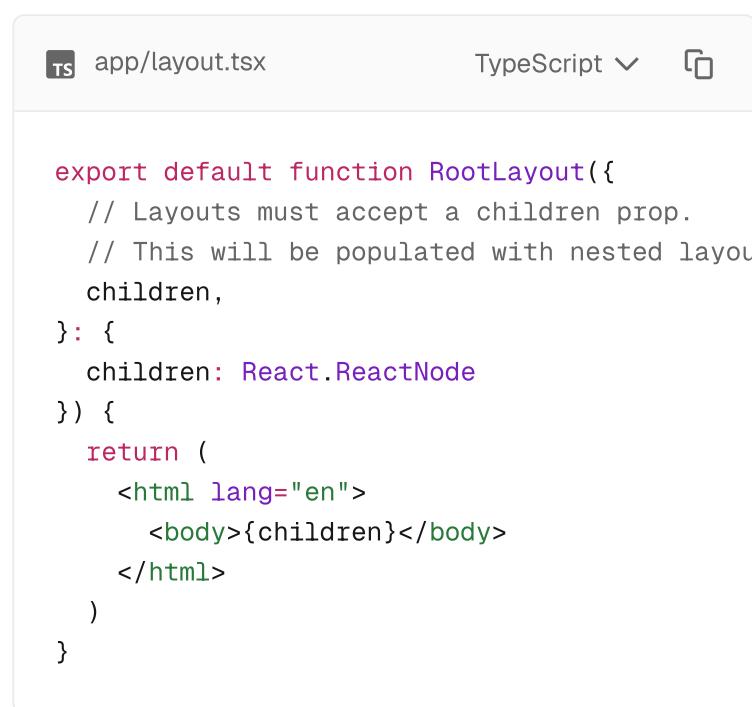
Update to the latest Next.js version (requires 13.4 or greater):

```
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`.

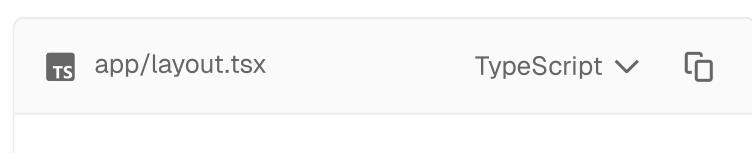


```
TS app/layout.tsx TypeScript ▾ ⌂

export default function RootLayout({
  // Layouts must accept a children prop.
  // This will be populated with nested layout
  // children,
}: {
  children: React.ReactNode
}) {
  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](#):



```
TS app/layout.tsx TypeScript ▾ ⌂
```

```
import type { Metadata } from 'next'

export const metadata: 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](#).

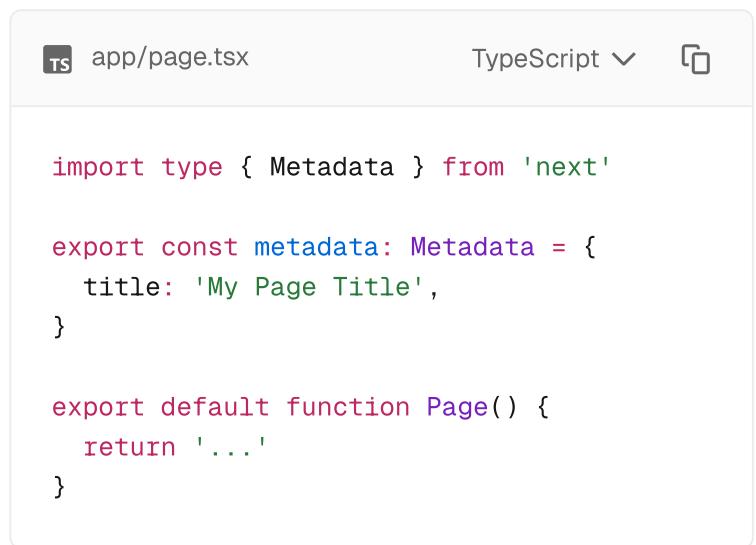
Before:



```
import Head from 'next/head'

export default function Page() {
  return (
    <>
    <Head>
      <title>My page title</title>
    </Head>
    </>
  )
}
```

After:



The screenshot shows a code editor window with the file name "app/page.tsx" at the top. The code is as follows:

```
import type { Metadata } from 'next'

export const metadata: 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/p 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`.

TS app/home-page.tsx

TypeScript ▾

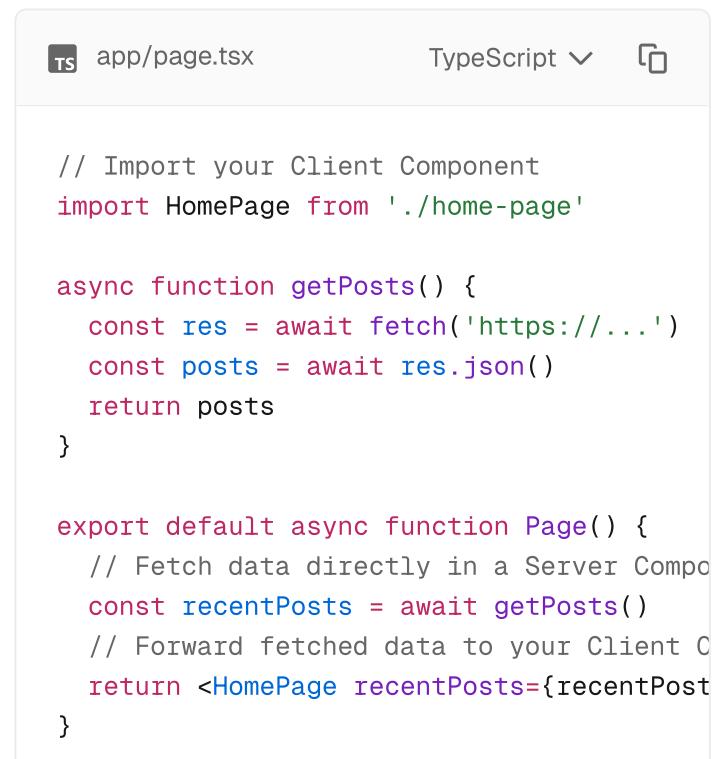


```
'use client'
```

```
// This is a Client Component (same as component)
// It receives data as props, has access to state
// prerendered on the server during the initial render
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.



```
TS app/page.tsx TypeScript ▾ ⌂

// 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.

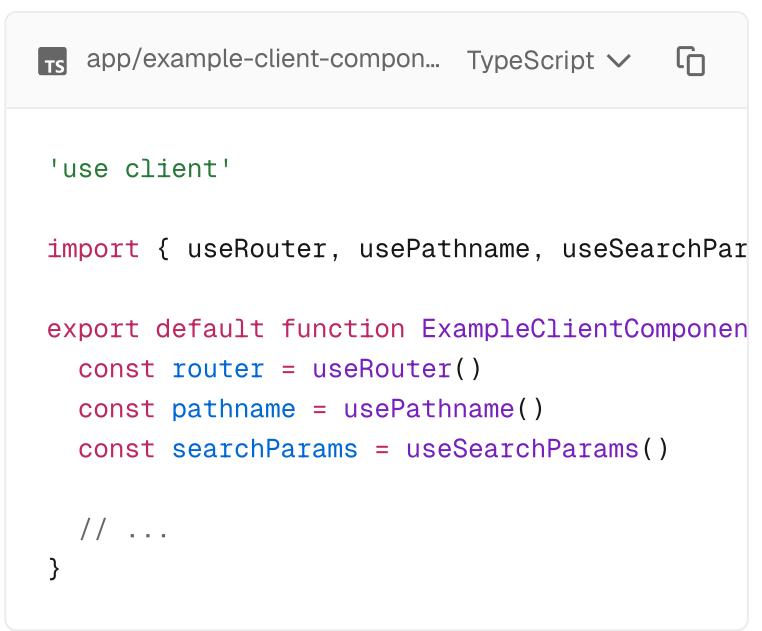
Step 5: Migrating Routing Hooks

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 }

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.

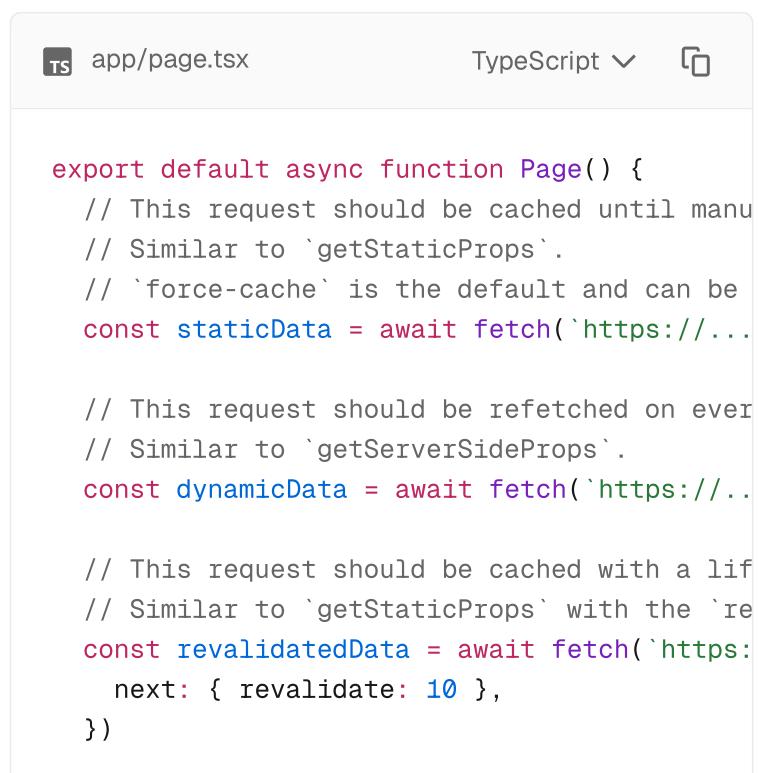
[View the `useRouter\(\)` API reference.](#)

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.



```
TS app/page.tsx TypeScript ▾ ⌂

export default async function Page() {
  // This request should be cached until manual
  // Similar to `getStaticProps`.
  // `force-cache` is the default and can be
  const staticData = await fetch(`https://...`)

  // This request should be refetched on every
  // Similar to `getServerSideProps`.
  const dynamicData = await fetch(`https://...`)

  // This request should be cached with a lifetime
  // Similar to `getStaticProps` with the `revalidate` option.
  const revalidatedData = await fetch(`https://...`)

  return (
    <div>
      <p>Static Data: {staticData}</p>
      <p>Dynamic Data: {dynamicData}</p>
      <p>Revalidated Data: {revalidatedData}</p>
    </div>
  )
}
```

```
        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).

```
JS pages/dashboard.js
```

```
// `pages` directory

export async function getServerSideProps() {
  const res = await fetch(`https://...`)
  const projects = await res.json()

  return { props: { projects } }
}

export default function Dashboard({ projects
  return (
    <ul>
      {projects.map((project) => (
        <li key={project.id}>{project.name}</li>
      ))}
    </ul>
  )
}
```

In the App 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.

```
TS app/dashboard/page.tsx
```

```
// `app` directory

// This function can be named anything
async function getProjects() {
  const res = await fetch(`https://...`, { ca
  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}</
      ))}
    </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.

`js` pages/index.js

```
// `pages` directory

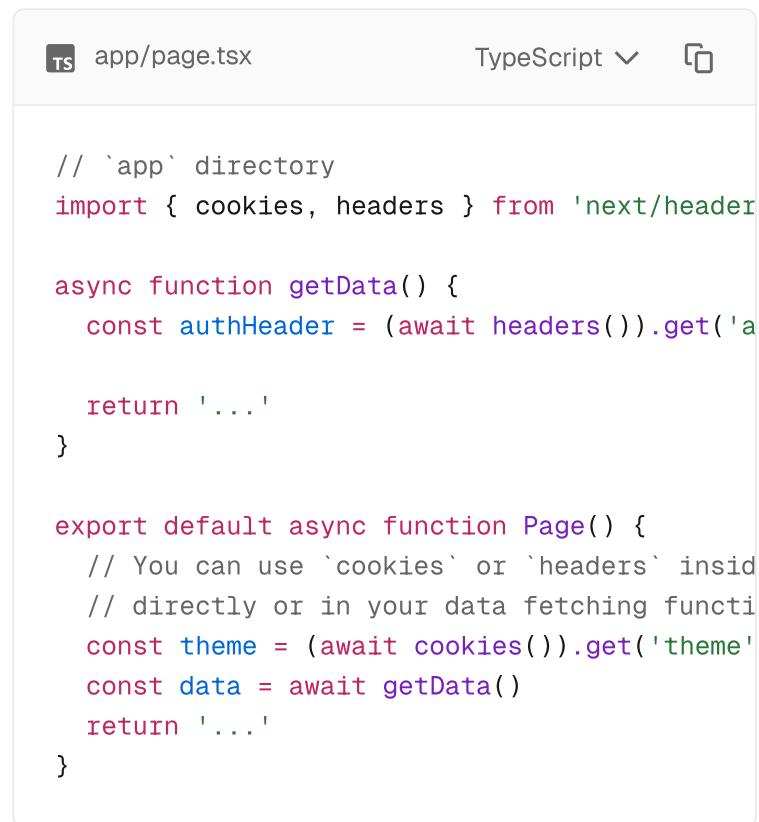
export async function getServerSideProps({ re
  const authHeader = req.getHeaders()['author
  const theme = req.cookies['theme'];

  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.



The screenshot shows a code editor window with the file name `app/page.tsx` at the top. The code uses TypeScript and includes imports for `cookies` and `headers` from `'next/header'`. It defines two functions: `getData()` which retrieves an auth header from the headers object, and `Page()` which uses `cookies` to get a theme and then calls `getData()`.

```
// `app` directory
import { cookies, headers } from 'next/header'

async function getData() {
  const authHeader = (await headers()).get('auth')

  return '...'
}

export default async function Page() {
  // You can use `cookies` or `headers` inside
  // 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.



The screenshot shows a code editor window with the file name `pages/index.js` at the top. The code uses JavaScript and defines an `async` function `getStaticProps()` which performs a `fetch` operation to retrieve data from a URL, converts the response to JSON, and returns it.

```
// `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>{proj
}
```

In the `app` directory, data fetching with `fetch()` will default to `cache: 'force-cache'`, which will cache the request data until manually invalidated. This is similar to `getStaticProps` in the `pages` directory.

```
JS app/page.js

// `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>{proj
}
```

Dynamic paths (`getStaticPaths`)

In the `pages` directory, the `getStaticPaths` function is used to define the dynamic paths that should be pre-rendered at build time.

```
JS pages/posts/[id].js

// `pages` directory
import PostLayout from '@/components/post-lay

export async function getStaticPaths() {
    return {
        paths: [{ params: { id: '1' } }, { params
    }
```

```
}

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.

```
JS app/posts/[id]/page.js

// `app` directory
import PostLayout from '@/components/post-layout'

export async function generateStaticParams() {
  return [{ id: '1' }, { id: '2' }]
}

async function getPost(params) {
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  return post
}

export default async function Post({ params }) {
  const post = await getPost(params)

  return <PostLayout post={post} />
}
```

Using the name `generateStaticParams` is more appropriate than `getStaticPaths` for the new model in the `app` directory. The `get` prefix is replaced with a more descriptive `generate`, which sits better alone now that `getStaticProps` and `getServerSideProps` are no longer necessary. The `Paths` suffix is replaced by `Params`, which is more appropriate for nested routing with multiple dynamic segments.

Replacing `fallback`

In the `pages` directory, the `fallback` property returned from `getStaticPaths` is used to define the behavior of a page that isn't pre-rendered at build time. This property can be set to `true` to show a fallback page while the page is being generated, `false` to show a 404 page, or `blocking` to generate the page at request time.

```
JS pages/posts/[id].js

// `pages` directory

export async function getStaticPaths() {
  return {
    paths: [],
    fallback: 'blocking'
  };
}

export async function getStaticProps({ params
  ...
})

export default function Post({ post }) {
  return ...
}
```

In the `app` directory the `config.dynamicParams` property controls how params outside of

`generateStaticParams` are handled:

- `true` : (default) Dynamic segments not included in `generateStaticParams` are generated on demand.
- `false` : Dynamic segments not included in `generateStaticParams` will return a 404.

This replaces the

`fallback: true | false | 'blocking'` option of `getStaticPaths` in the `pages` directory. The `fallback: 'blocking'` option is not included in `dynamicParams` because the difference between `'blocking'` and `true` is negligible with streaming.

```
JS app/posts/[id]/page.js

// `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.

```
JS pages/index.js

// `pages` directory

export async function getStaticProps() {
  const res = await fetch(`https://.../posts`)
  const posts = await res.json()

  return {
    props: { posts },
    revalidate: 60,
  }
}

export default function Index({ posts }) {
  return (
    <Layout>
      <PostList posts={posts} />
    </Layout>
  )
}
```

In the `app` directory, data fetching with `fetch()` can use `revalidate`, which will cache the request for the specified amount of seconds.

```
JS app/page.js

// `app` directory

async function getPosts() {
  const res = await fetch(`https://.../posts`)
  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.



```
TS app/api/route.ts TypeScript ▾ ⌂
export async function GET(request: 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](#).

Single-Page Applications

If you are also migrating to Next.js from a Single-Page Application (SPA) at the same time, see our [documentation](#) to learn more.

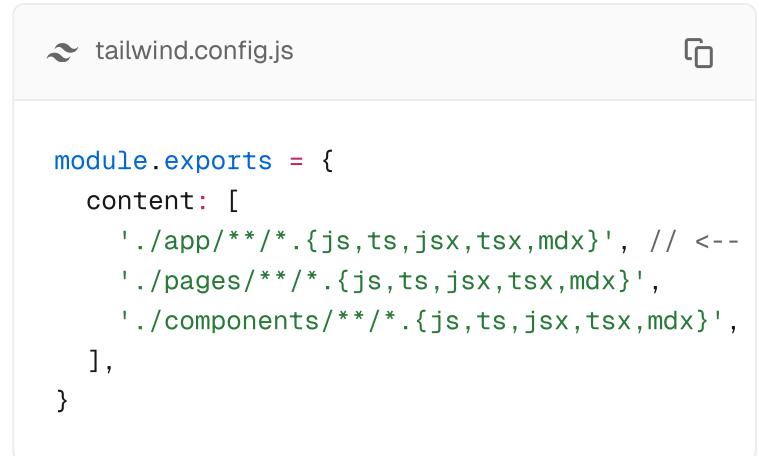
Step 7: Styling

In the `pages` directory, global stylesheets are restricted to only `pages/_app.js`. With the `app` directory, this restriction has been lifted. Global styles can be added to any layout, page, or component.

- [CSS Modules](#)
- [Tailwind CSS](#)
- [Global Styles](#)
- [CSS-in-JS](#)
- [External Stylesheets](#)
- [Sass](#)

Tailwind CSS

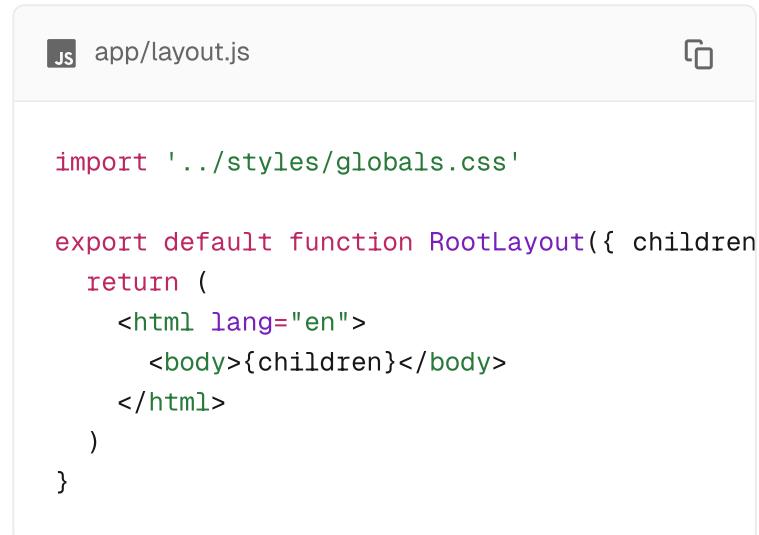
If you're using Tailwind CSS, you'll need to add the `app` directory to your `tailwind.config.js` file:



```
tailwind.config.js
```

```
module.exports = {
  content: [
    './app/**/*.{js,ts,jsx,tsx,mdx}', // <-- here
    './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:



```
app/layout.js
```

```
import '../styles/globals.css'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

Learn more about [styling with Tailwind CSS](#)

Using App Router together with Pages Router

When navigating between routes served by the different Next.js routers, there will be a hard navigation. Automatic link prefetching with `next/link` will not prefetch across routers.

Instead, you can [optimize navigations ↗](#) between App Router and Pages Router to retain the prefetched and fast page transitions. [Learn more ↗](#)

Codemods

Next.js provides Codemod transformations to help upgrade your codebase when a feature is deprecated. See [Codemods](#) for more information.

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to migrate from Create React App to Next.js

This guide will help you migrate an existing Create React App (CRA) 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 rendering. 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 mitigated with code splitting. However, if you try to do code splitting manually, you can inadvertently introduce network waterfalls. Next.js provides automatic code splitting and tree-shaking built into its router and build pipeline.

Network waterfalls

A common cause of poor performance occurs when applications make sequential client-server requests to fetch data. One pattern for data fetching in a [SPA](#) is to render a placeholder, and then fetch data after the component has mounted. Unfortunately, a child component can only begin fetching data after its parent has finished loading its own data, resulting in a “waterfall” of requests.

While client-side data fetching is supported in Next.js, Next.js also lets you move data fetching to the server. This often eliminates client-server waterfalls altogether.

Fast and intentional loading states

With built-in support for [streaming through React Suspense](#), you can define which parts of your UI load first and in what order, without creating 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 or component-level basis. For example, you could fetch data from your CMS and render blog posts at

build time (SSG) for quick load speeds, or fetch data at request time (SSR) when necessary.

Middleware

[Next.js Middleware](#) allows you to run code on the server before a request is completed. For instance, you can avoid a flash of unauthenticated content by redirecting a user to a login page in the middleware for authenticated-only pages. You can also use it for features like A/B testing, experimentation, and [internationalization](#).

Built-in Optimizations

[Images](#), [fonts](#), and [third-party scripts](#) often have a large impact on an application's performance. Next.js includes specialized components and APIs that automatically optimize them for you.

Migration Steps

Our goal 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 treat your application as a purely client-side application ([SPA](#)) without immediately replacing your existing router. This reduces complexity and merge conflicts.

Note: If you are using advanced CRA configurations such as a custom `homepage` field in your `package.json`, a custom service worker, or specific Babel/webpack tweaks, please see the **Additional Considerations** section at the end of this guide for tips on replicating or adapting these features in Next.js.

Step 1: Install the Next.js Dependency

Install Next.js in your existing project:

```
>_ Terminal
```

```
npm install next@latest
```

Step 2: Create the Next.js Configuration File

Create a `next.config.ts` at the root of your project (same level as your `package.json`). This file holds your [Next.js configuration options](#).

```
TS next.config.ts
```

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  output: 'export', // Outputs a Single-Page
  distDir: 'build', // Changes the build output
}

export default nextConfig
```

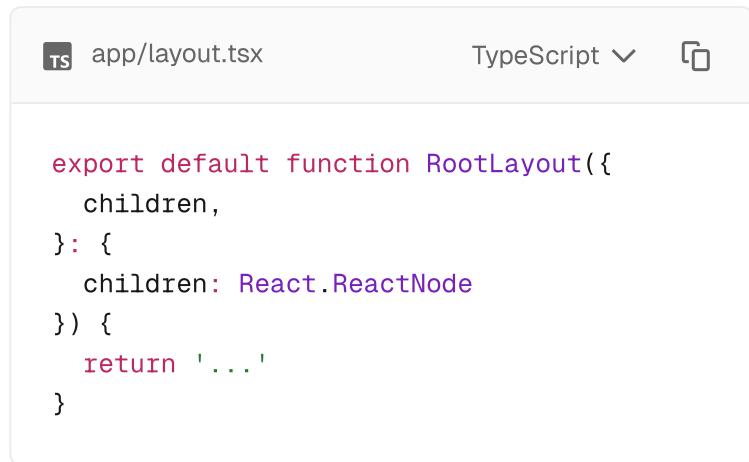
Note: Using `output: 'export'` means you're doing a static export. You will **not** have access to server-side features like SSR or APIs. You can remove this line to leverage Next.js server features.

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 your pages.

The closest equivalent of the root layout file in a CRA application is `public/index.html`, which includes your `<html>`, `<head>`, and `<body>` tags.

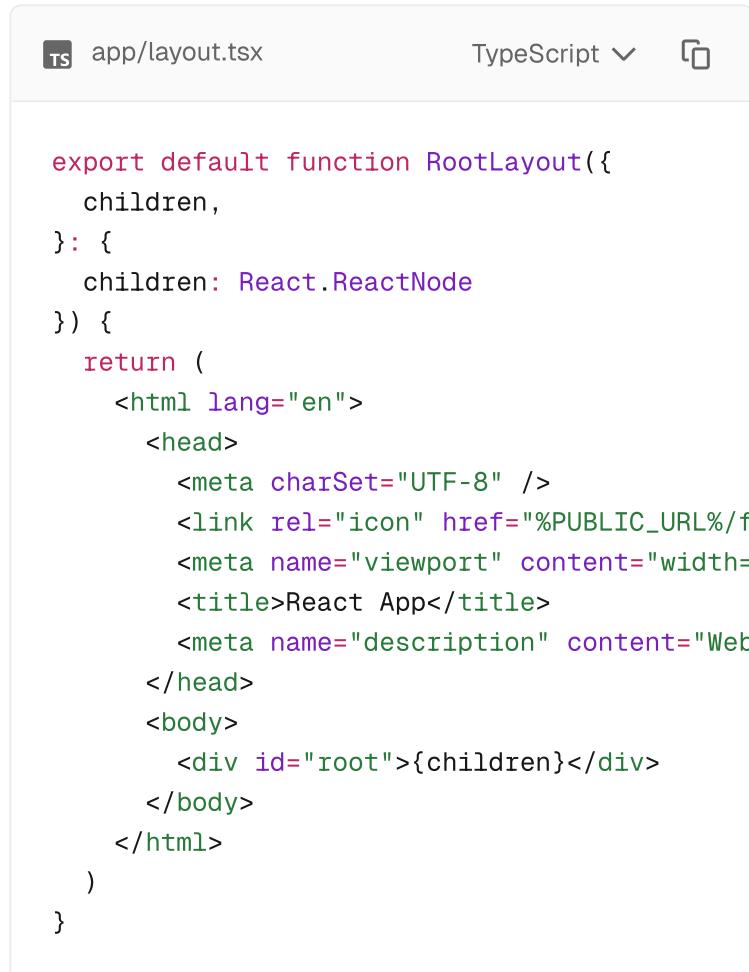
1. Create a new `app` directory inside your `src` folder (or at your project root if you prefer `app` at the root).
2. Inside the `app` directory, create a `layout.tsx` (or `layout.js`) file:



```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
    children,  
}: {  
    children: React.ReactNode  
) {  
    return '...'  
}
```

Now copy the content of your old `index.html` into this `<RootLayout>` component. Replace `body div#root` (and `body noscript`) with `<div id="root">{children}</div>`.



```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
    children,  
}: {  
    children: React.ReactNode  
) {  
    return (  
        <html lang="en">  
            <head>  
                <meta charSet="UTF-8" />  
                <link rel="icon" href="%PUBLIC_URL%/  
                <meta name="viewport" content="width=  
                <title>React App</title>  
                <meta name="description" content="Web  
            </head>  
            <body>  
                <div id="root">{children}</div>  
            </body>  
        </html>  
    )  
}
```

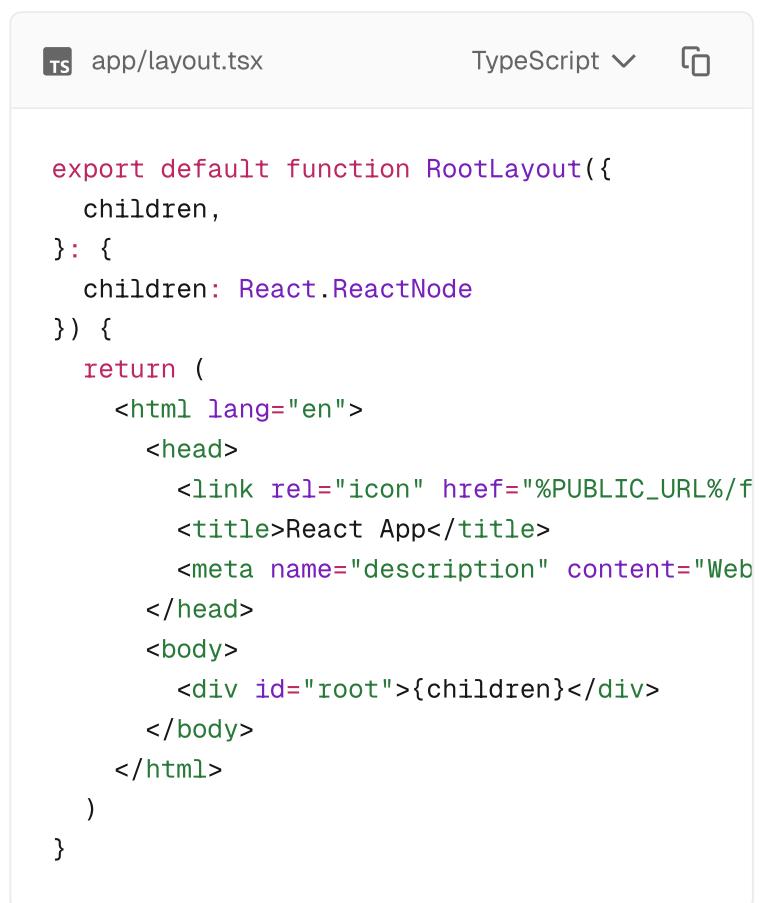
Good to know: Next.js ignores CRA's `public/manifest.json`, additional iconography, and [testing configuration](#) by default. If you need these, Next.js has support with its [Metadata API](#) and [Testing](#) setup.

Step 4: Metadata

Next.js automatically includes the

```
<meta charset="UTF-8" /> and  
<meta name="viewport"  
content="width=device-width, initial-  
scale=1" />
```

tags, so you can remove them from `<head>`:



```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
) {  
  return (  
    <html lang="en">  
      <head>  
        <link rel="icon" href="%PUBLIC_URL%/" />  
        <title>React App</title>  
        <meta name="description" content="Web  
      </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:



```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
) {  
  return (  
    <html lang="en">  
      <head>  
        <title>React App</title>  
        <meta name="description" content="Web  
      </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:



```
TS app/layout.tsx TypeScript ▾
```

```
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>  
  )  
}
```

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 CRA, Next.js supports [CSS Modules](#) out of the box. It also supports [global CSS imports](#).

If you have a global CSS file, import it into your `app/layout.tsx`:



```
TS app/layout.tsx TypeScript ⓘ  
  
import './index.css'  
  
export const 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>  
  )  
}
```

If you're using Tailwind CSS, see our [installation docs](#).

Step 6: Create the Entrypoint Page

Create React App uses `src/index.tsx` (or `index.js`) as the entry point. In Next.js (App Router), each folder inside the `app` directory corresponds to a route, and each folder should have a `page.tsx`.

Since we want to keep the app as an SPA for now and intercept **all** routes, we'll use an [optional catch-all route](#).

1. Create a `[[...slug]]` directory inside `app`.

```
app
└ [[...slug]]
  └ page.tsx
  └ layout.tsx
```

1. Add the following to `page.tsx`:

TS app/[[...slug]]/page.tsx

TypeScript

```
export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return '...' // We'll update this
}
```

This tells Next.js to generate a single route for the empty slug (`/`), effectively mapping **all** routes to the same page. This page is a [Server Component](#), prerendered into static HTML.

Step 7: Add a Client-Only Entrypoint

Next, we'll embed your CRA's root App component inside a [Client Component](#) so that all logic remains client-side. If this is your first time using Next.js, it's worth knowing that clients components (by default) are still prerendered on the server. You can think about them as having the additional capability of running client-side JavaScript.

Create a `client.tsx` (or `client.js`) in `app/[[...slug]]/`:

```
'use client'

import dynamic from 'next/dynamic'

const App = dynamic(() => import('../App'))

export function ClientOnly() {
  return <App />
}
```

- The `'use client'` directive makes this file a **Client Component**.
- The `dynamic` import with `ssr: false` disables server-side rendering for the `<App />` component, making it truly client-only (SPA).

Now update your `page.tsx` (or `page.js`) to use your new component:

```
import { ClientOnly } from './client'

export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return <ClientOnly />
}
```

Step 8: Update Static Image Imports

In CRA, importing an image file returns 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 `` 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 `` 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 `` dimension attribute's value, which can cause the image to appear distorted.

Keeping the `` 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 `` 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.

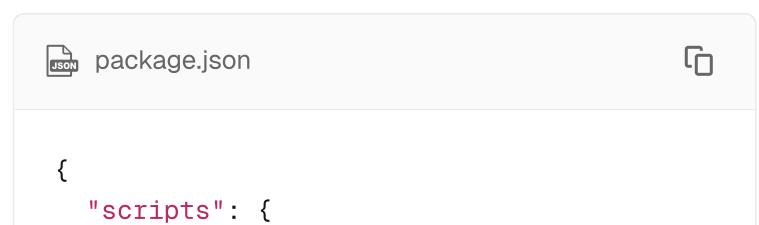
Step 9: Migrate Environment Variables

Next.js supports [environment variables](#) similarly to CRA but **requires** a `NEXT_PUBLIC_` prefix for any variable you want to expose in the browser.

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 10: Update Scripts in `package.json`

Update your `package.json` scripts to use Next.js commands. Also, add `.next` and `next-env.d.ts` to your `.gitignore`:



The image shows a code editor interface with two tabs: "package.json" and ".gitignore". The "package.json" tab contains the following JSON code:

```
{  
  "scripts": {  
    "start": "next dev"  
  }  
}
```

The ".gitignore" tab contains the following ignore patterns:

```
node_modules  
.next  
next-env.d.ts
```

```
"dev": "next dev --turbopack",
"build": "next build",
"start": "npx serve@latest ./build"
}
}
```

 .gitignore



```
# ...
.next
next-env.d.ts
```

Now you can run:

```
npm run dev
```

Open <http://localhost:3000>. You should see your application now running on Next.js (in SPA mode).

Step 11: Clean Up

You can now remove artifacts that are specific to Create React App:

- `public/index.html`
- `src/index.tsx`
- `src/react-app-env.d.ts`
- The `reportWebVitals` setup
- The `react-scripts` dependency (uninstall it from `package.json`)

Additional Considerations

`Using a Custom homepage in CRA`

If you used the `homepage` field in your CRA `package.json` to serve the app under a specific subpath, you can replicate that in Next.js using the `basePath` configuration in `next.config.ts`:

TS next.config.ts

```
import { NextConfig } from 'next'

const nextConfig: NextConfig = {
  basePath: '/my-subpath',
  // ...
}

export default nextConfig
```

Handling a Custom Service Worker

If you used CRA's service worker (e.g., `serviceWorker.js` from `create-react-app`), you can learn how to create [Progressive Web Applications \(PWAs\)](#) with Next.js.

Proxying API Requests

If your CRA app used the `proxy` field in `package.json` to forward requests to a backend server, you can replicate this with [Next.js rewrites](#) in `next.config.ts`:

TS next.config.ts

```
import { NextConfig } from 'next'

const nextConfig: NextConfig = {
  async rewrites() {
    return [
      {
        source: '/api/:path*',
        destination: 'https://your-backend.co
      },
    ]
  },
}
```

Custom Webpack / Babel Config

If you had a custom webpack or Babel configuration in CRA, you can extend Next.js's config in `next.config.ts`:

```
next.config.ts
```

```
import { NextConfig } from 'next'

const nextConfig: NextConfig = {
  webpack: (config, { isServer }) => {
    // Modify the webpack config here
    return config
  },
}

export default nextConfig
```

Note: This will require disabling Turbopack by removing `--turbopack` from your `dev` script.

TypeScript Setup

Next.js automatically sets up TypeScript if you have a `tsconfig.json`. Make sure `next-env.d.ts` is listed in your `tsconfig.json` `include` array:

```
{
```

```
"include": [ "next-env.d.ts", "app/**/*", "s  
}
```

Bundler Compatibility

Both Create React App and Next.js default to webpack for bundling. Next.js also offers [Turbopack](#) for faster local development with:

```
next dev --turbopack
```

You can still provide a [custom webpack configuration](#) if you need to migrate advanced webpack settings from CRA.

Next Steps

If everything worked, you now have a functioning Next.js application running as a single-page application. You aren't yet leveraging Next.js features like server-side rendering or file-based routing, but you can now do so incrementally:

- **Migrate from React Router** to the [Next.js App Router](#) for:
 - 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
- Enable ESLint with Next.js recommended rules by running `npx next lint` and configuring it to match your project's needs

Note: Using a static export (`output: 'export'`) **does not currently support ↗** the `useParams` hook or other server features. To use all Next.js features, remove `output: 'export'` from your `next.config.ts`.

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to migrate 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:

> Terminal



```
npm install next@latest
```

Step 2: Create the Next.js Configuration File

Create a `next.config.mjs` at the root of your project. This file will hold your [Next.js configuration options](#).

JS next.config.mjs



```
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export', // Outputs a Single-Page
  distDir: './dist', // Changes the build out
}

export default nextConfig
```

Good to know: You can use either `.js` or `.mjs` for your Next.js configuration file.

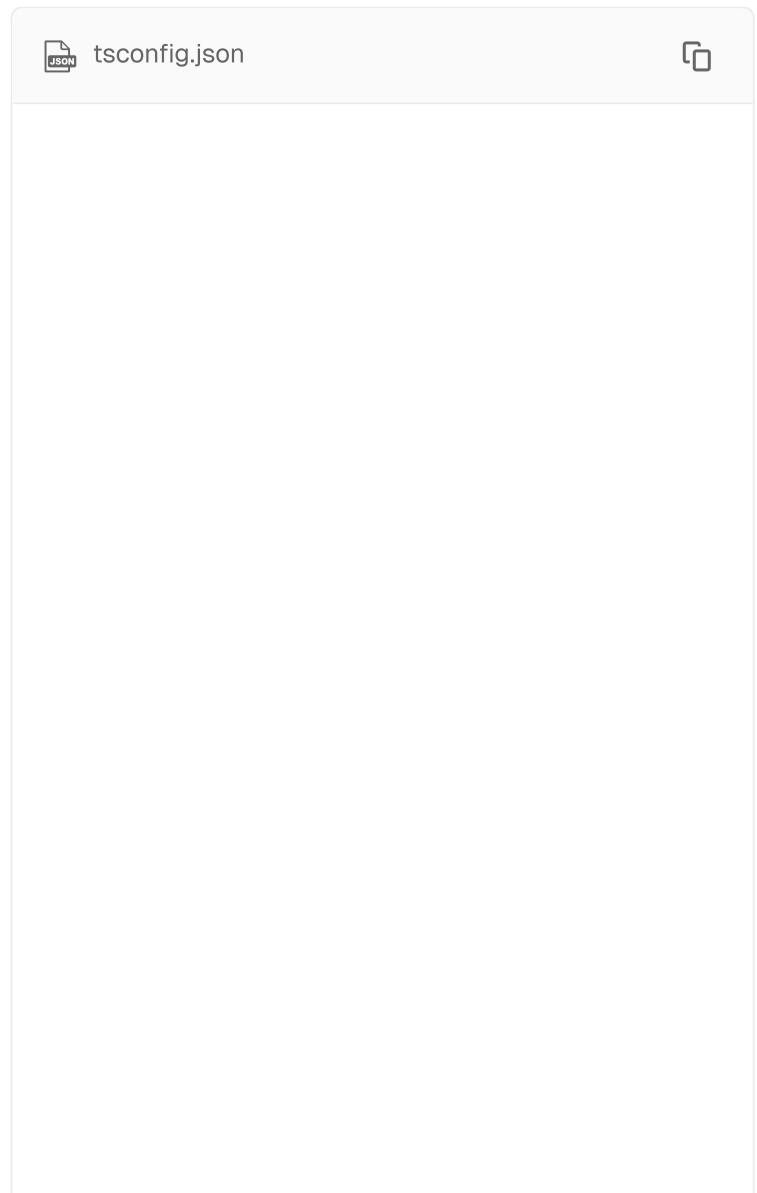
Step 3: Update TypeScript Configuration

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"]  
  "exclude": [".node_modules"]  
}
```

You can find more information about configuring TypeScript on the [Next.js docs](#).

Step 4: Create the Root Layout

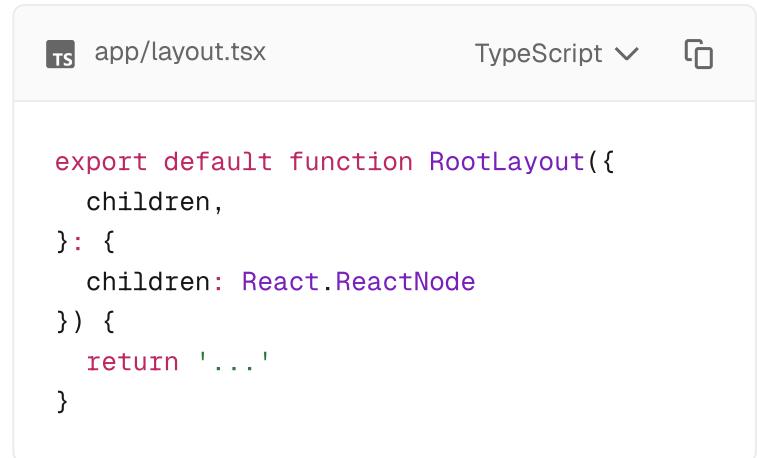
A Next.js [App Router](#) application must include a [root layout file](#), which is a [React Server Component](#) that will wrap all pages in your application. This file is defined at the top level of the `app` directory.

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` folder.

2. Create a new `layout.tsx` file inside that `app` directory:

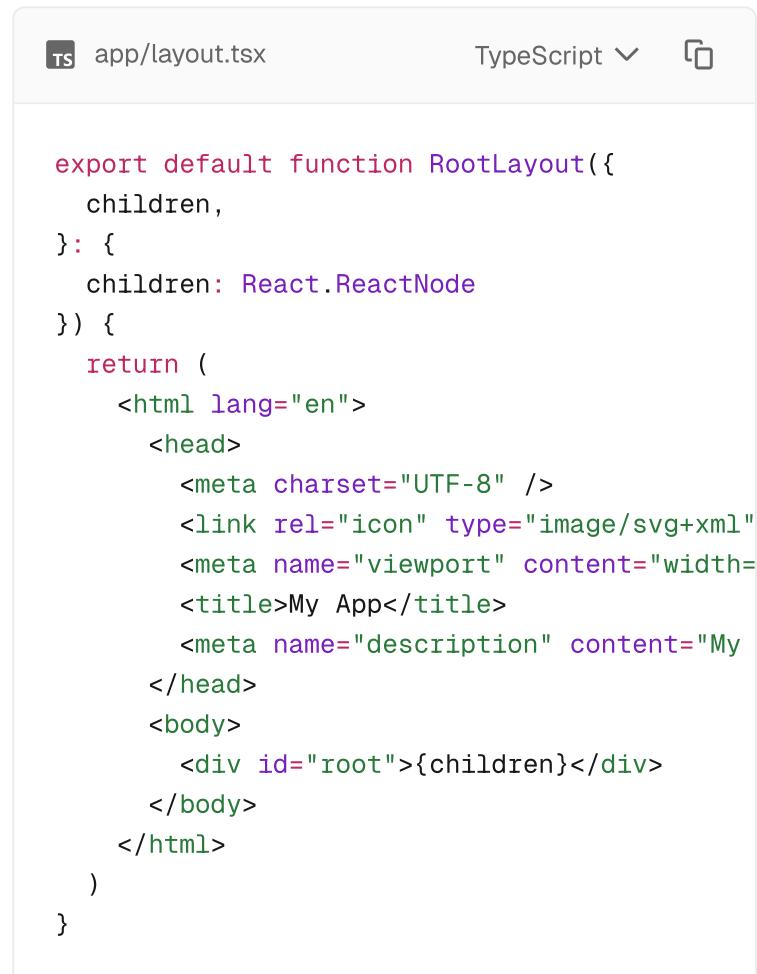


```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
) {  
  return '<...>'  
}
```

Good to know: `.js`, `.jsx`, or `.tsx` extensions can be used for Layout files.

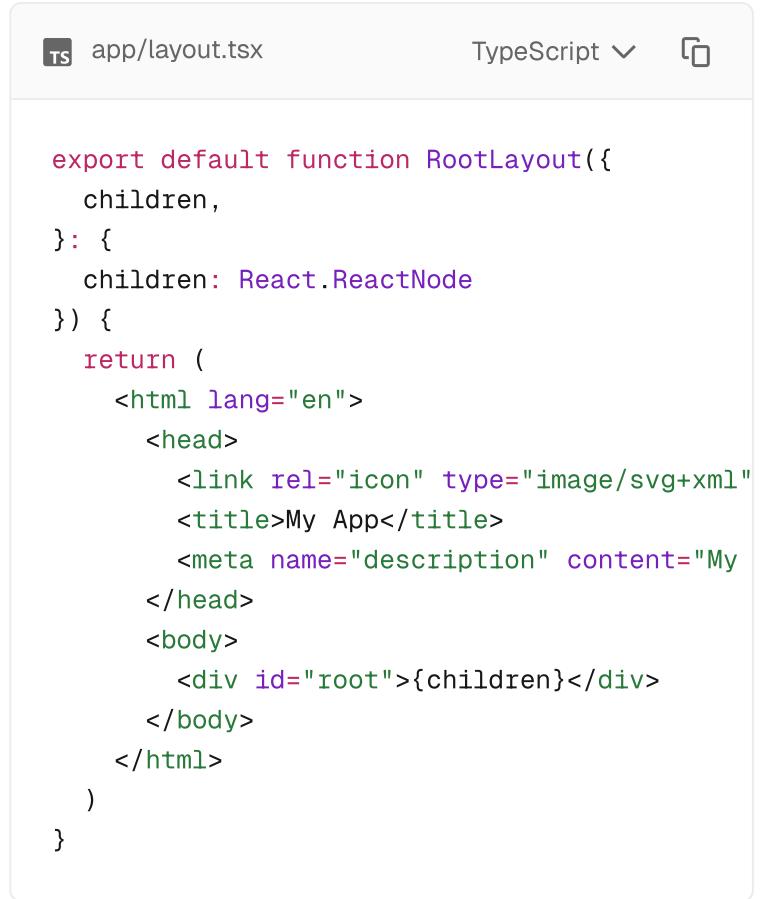
1. Copy the content of your `index.html` file into the previously created `<RootLayout>` component while replacing the `body.div#root` and `body.script` tags with `<div id="root">{children}</div>`:



```
TS app/layout.tsx TypeScript ▾
```

```
import { ReactNode } from 'react'  
  
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=""/>  
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
        <title>My App</title>  
        <meta name="description" content="My App" />  
      </head>  
      <body>  
        <div id="root">{children}</div>  
      </body>  
    </html>  
  )  
}
```

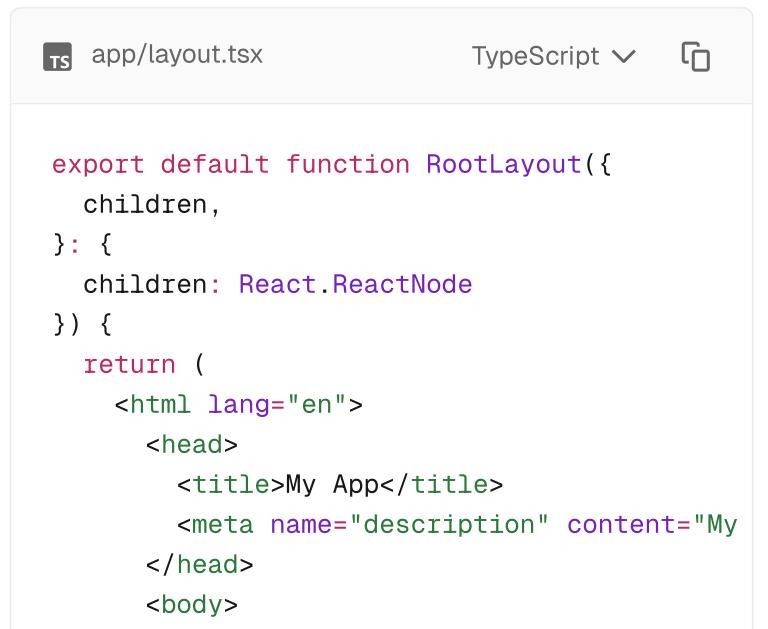
1. Next.js already includes by default the `meta charset` and `meta viewport` tags, so you can safely remove those from your `<head>`:



```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
}) {  
  return (  
    <html lang="en">  
      <head>  
        <link rel="icon" type="image/svg+xml"  
        <title>My App</title>  
        <meta name="description" content="My  
      </head>  
      <body>  
        <div id="root">{children}</div>  
      </body>  
    </html>  
  )  
}
```

1. 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:



```
TS app/layout.tsx TypeScript ▾
```

```
export default function RootLayout({  
  children,  
}: {  
  children: React.ReactNode  
}) {  
  return (  
    <html lang="en">  
      <head>  
        <title>My App</title>  
        <meta name="description" content="My  
      </head>  
      <body>  
    </html>  
  )  
}
```

```
<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`:



```
TS app/layout.tsx TypeScript ▾
```

```
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>
  )
}
```

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

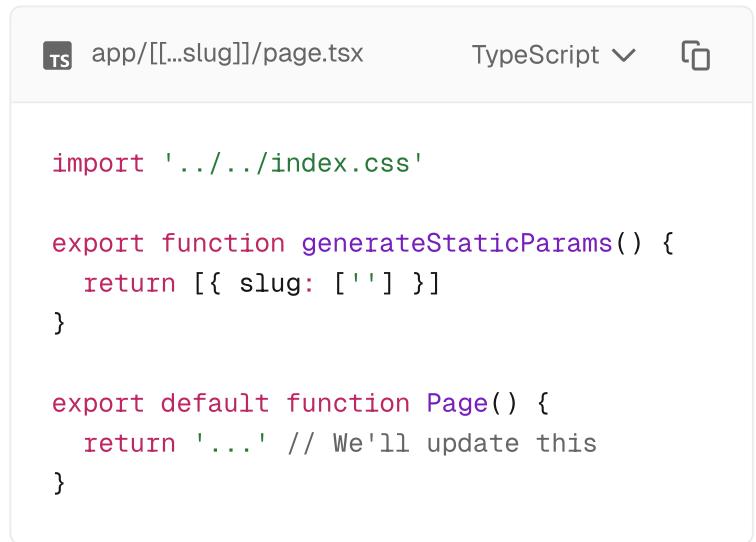
`main.tsx` file. In this step, you'll set up the entrypoint of your application.

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 folders 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.

1. Create a new `page.tsx` file inside the `app/[[... slug]]` directory with the following content:



```
TS app/[[...slug]]/page.tsx TypeScript ▾ ⌂

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.

```
TS app/[...slug]/client.tsx TypeScript ▾ ⌂

use client

import React from 'react'
import dynamic from 'next/dynamic'

const App = dynamic(() => import('../App'))

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'))
```

Now, update your entrypoint page to use the new component:

```
TS app/[...slug]/page.tsx TypeScript ▾ ⌂

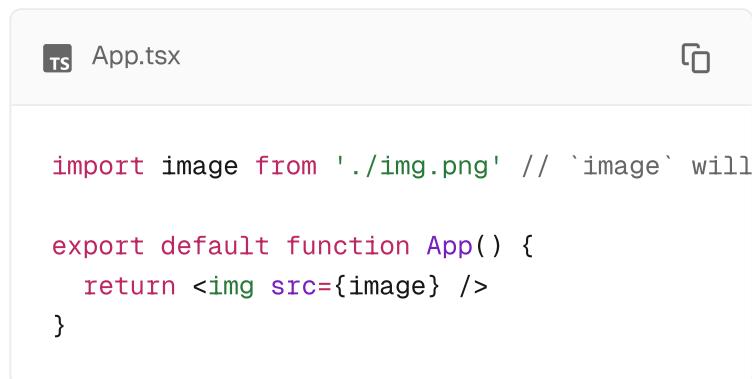
import '../index.css'
import { ClientOnly } from './client'

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:



```
TS App.tsx

import image from './img.png' // `image` will be a string

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 `` 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 `` 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 `` dimension attribute's value, which can cause the image to appear distorted.

Keeping the `` 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'
```

1. Pass the image `src` property instead of the whole image object to your `` 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"
```

1. 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
  distDir: './dist', // Changes the build out
  basePath: process.env.NEXT_PUBLIC_BASE_PATH
}
```

```
export default nextConfig
```

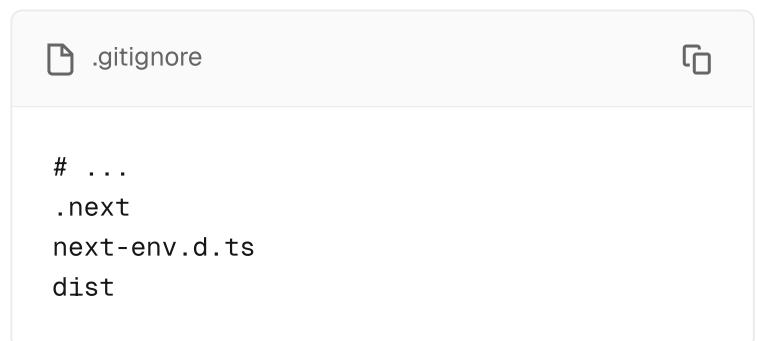
1. Update `import.meta.env.BASE_URL` usages to `process.env.NEXT_PUBLIC_BASE_PATH`

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

Was this helpful?    

 Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.

 Latest Version

15.5.4



How to build micro-frontends using multi-zones and Next.js

▼ Examples

- [With Zones ↗](#)

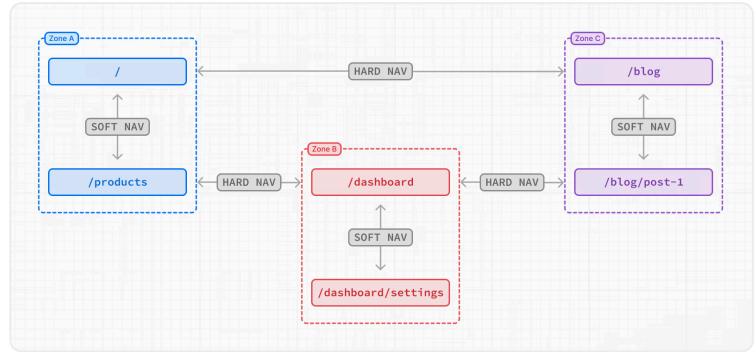
Multi-Zones are an approach to micro-frontends that separate a large application on a domain into smaller Next.js applications that each serve a set of paths. This is useful when there are collections of pages unrelated to the other pages in the application. By moving those pages to a separate zone (i.e., 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.

```
JS next.config.js
```

```
/** @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, and you also need to rewrite the requests for the static assets. For example:

```
JS next.config.js

async rewrites() {
  return [
    {
      source: '/blog',
      destination: `${process.env.BLOG_}`,
    },
    {
      source: '/blog/:path+',
      destination: `${process.env.BLOG_}`,
    },
    {
      source: '/blog-static/:path+',
      destination: `${process.env.BLOG_}`
    }
  ];
}
```

`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.



```
JS middleware.js

export async function middleware(request) {
  const { pathname, search } = req.nextUrl;
  if (pathname === '/your-path' && myFeatureF
    return NextResponse.rewrite(`${
      rewriteDom
    })
}
```

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.

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to instrument your Next.js app with OpenTelemetry

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:

```
>_ Terminal ✖  
  
npm install @vercel/otel @opentelemetry/sdk-1
```

Next, create a custom `instrumentation.ts` (or `.js`) file in the **root directory** of the project (or inside `src` folder if using one):

```
TS your-project/instrumentation... TypeScript ✖  
  
// Your instrumentation code here
```

```
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.

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:

> Terminal

```
npm install @opentelemetry/sdk-node @opentele
```

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:

```
TS instrumentation.ts TypeScript ▾
```

```
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs')
    await import('./instrumentation.node.ts')
}
```

```
TS instrumentation.node.ts TypeScript ▾
```

```
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace'
import { Resource } from '@opentelemetry/resource'
import { NodeSDK } from '@opentelemetry/sdk-node'
import { SimpleSpanProcessor } from '@opentelemetry/processor-span'
import { ATTR_SERVICE_NAME } from '@opentelemetry/const'

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 ↗](#).

> Terminal

```
npm install @opentelemetry/api
```

The following example demonstrates a function that fetches GitHub stars and adds a custom `fetchGithubStars` span to track the fetch request's result:

```
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.

Was this helpful?    



Using Pages Router
Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version
15.5.4



How to optimize package bundling

Bundling external packages can significantly improve the performance of your application. 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`.

next.config.js



```
/** @type {import('next').NextConfig} */  
const nextConfig = {}  
  
const withBundleAnalyzer = require('@next/bun  
    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.

JS next.config.js

```
/** @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`.

JS next.config.js

```
/** @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`.

`JS` `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`:

`JS` `next.config.js`

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  // Automatically bundle external packages
  bundlePagesRouterDependencies: true,
  // Opt specific packages out of bundling for
  serverExternalPackages: ['package-name'],
}
```

```
module.exports = nextConfig
```

Next Steps

Learn more about optimizing your application for production.

Production

Recommendations
to ensure the best
performance and...

Was this helpful?



 Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.

 Latest Version

15.5.4



How to configure PostCSS in Next.js

Default Behavior

Next.js compiles CSS for its [built-in CSS support](#) using PostCSS.

Out of the box, with no configuration, Next.js compiles CSS with the following transformations:

- [Autoprefixer](#) ↗ automatically adds vendor prefixes to CSS rules (back to IE11).
- [Cross-browser Flexbox bugs](#) ↗ are corrected to behave like [the spec](#) ↗.
- New CSS features are automatically compiled for Internet Explorer 11 compatibility:
 - [all Property](#) ↗
 - [Break Properties](#) ↗
 - [font-variant Property](#) ↗
 - [Gap Properties](#) ↗
 - [Media Query Ranges](#) ↗

By default, [CSS Grid](#) ↗ and [Custom Properties](#) ↗ (CSS variables) are **not compiled** for IE11 support.

To compile [CSS Grid Layout](#) ↗ for IE11, you can place the following comment at the top of your

CSS file:

```
/* autoprefixer grid: autoplacement */
```

You can also enable IE11 support for [CSS Grid Layout](#) ↗ in your entire project by configuring autoprefixer with the configuration shown below (collapsed). See "[Customizing Plugins](#)" below for more information.

► [Click to view the configuration to enable CSS Grid Layout](#)

CSS variables are not compiled because it is [not possible to safely do so](#) ↗. If you must use variables, consider using something like [Sass variables](#) ↗ which are compiled away by [Sass](#) ↗.

Customizing Target Browsers

Next.js allows you to configure the target browsers (for [Autoprefixer](#) ↗ and compiled css features) through [Browserslist](#) ↗.

To customize browserslist, create a `browserslist` key in your `package.json` like so:



```
JSON package.json
```

```
{  
  "browserslist": [">0.3%", "not dead", "not]  
}
```

You can use the [browserslist](#) ↗ tool to visualize what browsers you are targeting.

CSS Modules

No configuration is needed to support CSS Modules. To enable CSS Modules for a file, rename the file to have the extension `.module.css`.

You can learn more about [Next.js' CSS Module support here](#).

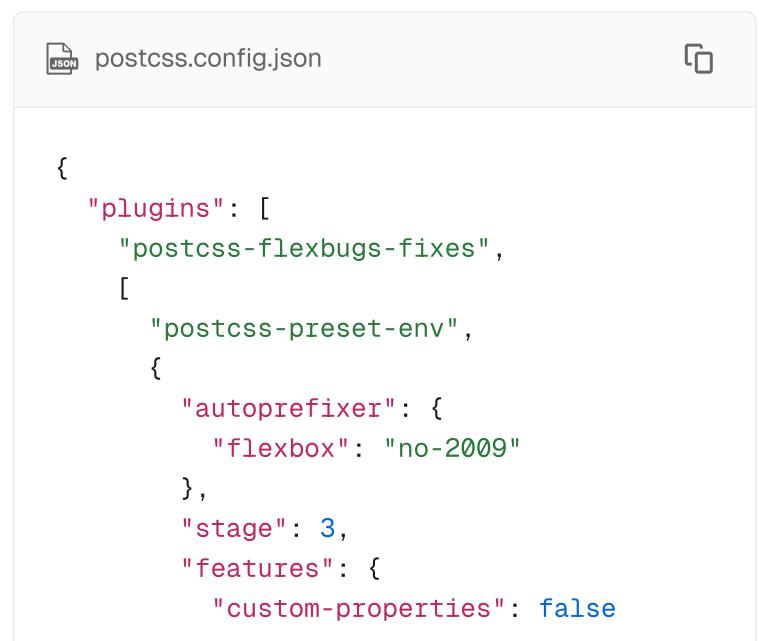
Customizing Plugins

Warning: When you define a custom PostCSS configuration file, Next.js **completely disables** the [default behavior](#). Be sure to manually configure all the features you need compiled, including [Autoprefixer](#) ↗. You also need to install any plugins included in your custom configuration manually, i.e.

```
npm install postcss-flexbugs-fixes postcss-preset-env
```

To customize the PostCSS configuration, create a `postcss.config.json` file in the root of your project.

This is the default configuration used by Next.js:



```
{  
  "plugins": [  
    "postcss-flexbugs-fixes",  
    [  
      "postcss-preset-env",  
      {  
        "autoprefixer": {  
          "flexbox": "no-2009"  
        },  
        "stage": 3,  
        "features": {  
          "custom-properties": false  
        }  
      }  
    ]  
  ]  
}
```

```
        }
    ]
}
}
```

Good to know: Next.js also allows the file to be named `.postcssrc.json`, or, to be read from the `postcss` key in `package.json`.

It is also possible to configure PostCSS with a `postcss.config.js` file, which is useful when you want to conditionally include plugins based on environment:

```
JS postcss.config.js
```

```
module.exports = {
  plugins:
    process.env.NODE_ENV === 'production'
      ? [
          'postcss-flexbugs-fixes',
          [
            'postcss-preset-env',
            {
              autoprefixer: {
                flexbox: 'no-2009',
              },
              stage: 3,
              features: {
                'custom-properties': false,
              },
            },
          ],
        ]
      : [
        // No transformations in development
      ],
}
```

Good to know: Next.js also allows the file to be named `.postcssrc.js`.

Do not use `require()` to import the PostCSS Plugins. Plugins must be provided as strings.

Good to know: If your `postcss.config.js` needs to support other non-Next.js tools in the same project, you must use the interoperable object-based format instead:

```
module.exports = {
  plugins: {
    'postcss-flexbugs-fixes': {},
    'postcss-preset-env': {
      autoprefixer: {
        flexbox: 'no-2009',
      },
      stage: 3,
      features: {
        'custom-properties': false,
      },
    },
  },
}
```

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to preview content with Preview Mode in Next.js



This is a legacy API and no longer recommended. It's still supported for backward compatibility.

Note: This feature is superseded by [Draft Mode](#).

► Examples

In the [Pages documentation](#) and the [Data Fetching documentation](#), we talked about how to pre-render a page at build time (**Static Generation**) using `getStaticProps` and `getStaticPaths`.

Static Generation is useful when your pages fetch data from a headless CMS. However, it's not ideal when you're writing a draft on your headless CMS and want to **preview** the draft immediately on your page. You'd want Next.js to render these pages at **request time** instead of build time and fetch the draft content instead of the published content. You'd want Next.js to bypass Static Generation only for this specific case.

Next.js has a feature called **Preview Mode** which solves this problem. Here are instructions on how to use it.

Step 1: Create and access a preview API route

Take a look at the [API Routes documentation](#) first if you're not familiar with Next.js API Routes.

First, create a **preview API route**. It can have any name - e.g. `pages/api/preview.js` (or `.ts` if using TypeScript).

In this API route, you need to call `setPreviewData` on the response object. The argument for `setPreviewData` should be an object, and this can be used by `getStaticProps` (more on this later). For now, we'll use `{}`.

```
export default function handler(req, res) {
  // ...
  res.setPreviewData({})
  // ...
}
```

`res.setPreviewData` sets some **cookies** on the browser which turns on the preview mode. Any requests to Next.js containing these cookies will be considered as the **preview mode**, and the behavior for statically generated pages will change (more on this later).

You can test this manually by creating an API route like below and accessing it from your browser manually:

`JS` `pages/api/preview.js`



```
// simple example for testing it manually from browser
export default function handler(req, res) {
  res.setPreviewData({})
  res.end('Preview mode enabled')
```

If you open your browser's developer tools and visit `/api/preview`, you'll notice that the `--prerender_bypass` and `--next_preview_data` cookies will be set on this request.

Securely accessing it from your Headless CMS

In practice, you'd want to call this API route *securely* from your headless CMS. The specific steps will vary depending on which headless CMS you're using, but here are some common steps you could take.

These steps assume that the headless CMS you're using supports setting **custom preview URLs**. If it doesn't, you can still use this method to secure your preview URLs, but you'll need to construct and access the preview URL manually.

First, you should create a **secret token string** using a token generator of your choice. This secret will only be known by your Next.js app and your headless CMS. This secret prevents people who don't have access to your CMS from accessing preview URLs.

Second, if your headless CMS supports setting custom preview URLs, specify the following as the preview URL. This assumes that your preview API route is located at `pages/api/preview.js`.

> Terminal



`https://<your-site>/api/preview?secret=<token>`

- <your-site> should be your deployment domain.
- <token> should be replaced with the secret token you generated.
- <path> should be the path for the page that you want to preview. If you want to preview /posts/foo, then you should use &slug=/posts/foo.

Your headless CMS might allow you to include a variable in the preview URL so that <path> can be set dynamically based on the CMS's data like so:

&slug=/posts/{entry.fields.slug}

Finally, in the preview API route:

- Check that the secret matches and that the slug parameter exists (if not, the request should fail).
- Call res.setPreviewData.
- Then redirect the browser to the path specified by slug. (The following example uses a [307 redirect ↗](#)).

```
export default async (req, res) => {
  // Check the secret and next parameters
  // This secret should only be known to this
  if (req.query.secret !== 'MY_SECRET_TOKEN')
    return res.status(401).json({ message: 'I' })
}

// Fetch the headless CMS to check if the p
// getPostBySlug would implement the requir
const post = await getPostBySlug(req.query.

// If the slug doesn't exist prevent previe
if (!post) {
  return res.status(401).json({ message: 'I' })
}

// Enable Preview Mode by setting the cooki
res.setPreviewData({})

// Redirect to the path from the fetched po
```

```
// We don't redirect to req.query.slug as t  
res.redirect(post.slug)  
}
```

If it succeeds, then the browser will be redirected to the path you want to preview with the preview mode cookies being set.

Step 2: Update `getStaticProps`

The next step is to update `getStaticProps` to support the preview mode.

If you request a page which has `getStaticProps` with the preview mode cookies set (via `res.setPreviewData`), then `getStaticProps` will be called at **request time** (instead of at build time).

Furthermore, it will be called with a `context` object where:

- `context.preview` will be `true`.
- `context.previewData` will be the same as the argument used for `setPreviewData`.

```
export async function getStaticProps(context)  
  // If you request this page with the previe  
  //  
  // - context.preview will be true  
  // - context.previewData will be the same a  
  //   the argument used for `setPreviewData`  
}
```

We used `res.setPreviewData({})` in the preview API route, so `context.previewData` will be `{}`. You can use this to pass session information from the preview API route to `getStaticProps` if necessary.

If you're also using `getStaticPaths`, then `context.params` will also be available.

Fetch preview data

You can update `getStaticProps` to fetch different data based on `context.preview` and/or `context.previewData`.

For example, your headless CMS might have a different API endpoint for draft posts. If so, you can use `context.preview` to modify the API endpoint URL like below:

```
export async function getStaticProps(context)
  // If context.preview is true, append "/pre"
  // to request draft data instead of publish
  // based on which headless CMS you're using
  const res = await fetch(`https://.../${cont
  // ...
}
```

That's it! If you access the preview API route (with `secret` and `slug`) from your headless CMS or manually, you should now be able to see the preview content. And if you update your draft without publishing, you should be able to preview the draft.

Set this as the preview URL on your headless CMS or access manually, and you should be able to see the preview.

> Terminal



```
https://<your-site>/api/preview?secret=<token
```

More Details

Good to know: during rendering `next/router` exposes an `isPreview` flag, see the [router object docs](#) for more info.

Specify the Preview Mode duration

`setPreviewData` takes an optional second parameter which should be an options object. It accepts the following keys:

- `maxAge` : Specifies the number (in seconds) for the preview session to last for.
- `path` : Specifies the path the cookie should be applied under. Defaults to `/` enabling preview mode for all paths.

```
setPreviewData(data, {  
  maxAge: 60 * 60, // The preview mode cookie  
  path: '/about', // The preview mode cookies  
})
```

Clear the Preview Mode cookies

By default, no expiration date is set for Preview Mode cookies, so the preview session ends when the browser is closed.

To clear the Preview Mode cookies manually, create an API route that calls

`clearPreviewData()` :

```
js pages/api/clear-preview-mode-cookies.js
```

```
export default function handler(req, res) {  
  res.clearPreviewData({})  
}
```

Then, send a request to

`/api/clear-preview-mode-cookies` to invoke the API Route. If calling this route using [next/link](#),

you must pass `prefetch={false}` to prevent calling `clearPreviewData` during link prefetching.

If a path was specified in the `setPreviewData` call, you must pass the same path to `clearPreviewData`:

```
JS pages/api/clear-preview-mode-cookies.js
```

```
export default function handler(req, res) {
  const { path } = req.query

  res.clearPreviewData({ path })
}
```

previewData size limits

You can pass an object to `setPreviewData` and have it be available in `getStaticProps`. However, because the data will be stored in a cookie, there's a size limitation. Currently, preview data is limited to 2KB.

Works with `getServerSideProps`

The preview mode works on `getServerSideProps` as well. It will also be available on the `context` object containing `preview` and `previewData`.

Good to know: You shouldn't set the `Cache-Control` header when using Preview Mode because it cannot be bypassed. Instead, we recommend using [ISR](#).

Works with API Routes

API Routes will have access to `preview` and `previewData` under the request object. For example:

```
export default function myApiRoute(req, res)
  const isPreview = req.preview
```

```
const previewData = req.previewData  
// ...  
}
```

Unique per next build

Both the bypass cookie value and the private key for encrypting the `previewData` change when `next build` is completed. This ensures that the bypass cookie can't be guessed.

Good to know: To test Preview Mode locally over HTTP your browser will need to allow third-party cookies and local storage access.

Was this helpful?    

 Using Pages Router	Features available in /pages
 Latest Version	15.5.4

ⓘ You are currently viewing the documentation for Pages Router.

How to optimize your Next.js application for 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](#) and [before going to production](#), 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:

- **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

- **<Link> component:** Use the `<Link>` component for client-side navigation and prefetching.
- **Custom Errors:** Gracefully handle `500` and `404` errors

Data fetching and caching

- **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

- **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.
- **<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

- **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

- `<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).

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 ↗](#)

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



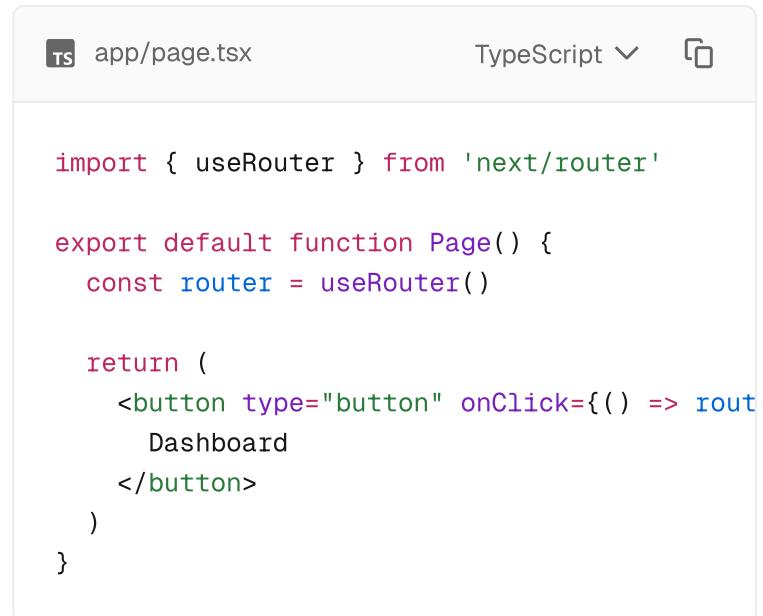
How to handle redirects in Next.js

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
<code>useRouter</code>	Perform a client-side navigation	Components
<code>redirects in next.config.js</code>	Redirect an incoming request based on a path	<code>next.config.js</code> file
<code>NextResponse.redirect</code>	Redirect an incoming request based on a condition	Middleware

useRouter() hook

If you need to redirect inside a component, you can use the `push` method from the `useRouter` hook. For example:



```
TS app/page.tsx TypeScript ▾
```

```
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.

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:



```
TS next.config.ts TypeScript ▾ ⌂

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  async redirects() {
    return [
      // Basic redirect
      {
        source: '/about',
        destination: '/',
        permanent: true,
      },
      // Wildcard path matching
      {
        source: '/blog/:slug',
        destination: '/news/:slug',
        permanent: true,
      },
    ]
  },
}

export default nextConfig
```

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:

```
TS middleware.ts TypeScript ▾ ⌂

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'))
}

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:



The screenshot shows a code editor window with a tab labeled "middleware.ts" and a status bar indicating "TypeScript". The code itself is a TypeScript file for a Next.js middleware. It imports NextResponse and NextRequest from 'next', and get from '@vercel/edge-config'. It defines a type RedirectEntry with properties destination (string) and permanent (boolean). The export async function middleware takes a request and returns a response. It checks if redirectData exists and is a JSON object. If so, it extracts the redirectEntry and statusCode, then uses NextResponse.redirect. If not, it continues without redirecting. Finally, it calls NextResponse.next() to handle the rest of the request.

```
import { NextResponse, NextRequest } from 'next'
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 === 'object') {
    const redirectEntry: RedirectEntry = JSON.parse(redirectData)
    const statusCode = redirectEntry.permanent ? 301 : 302
    return NextResponse.redirect(redirectEntry.destination, { status: statusCode })
  }

  // No redirect found, continue without redirecting
  return NextResponse.next()
}
```

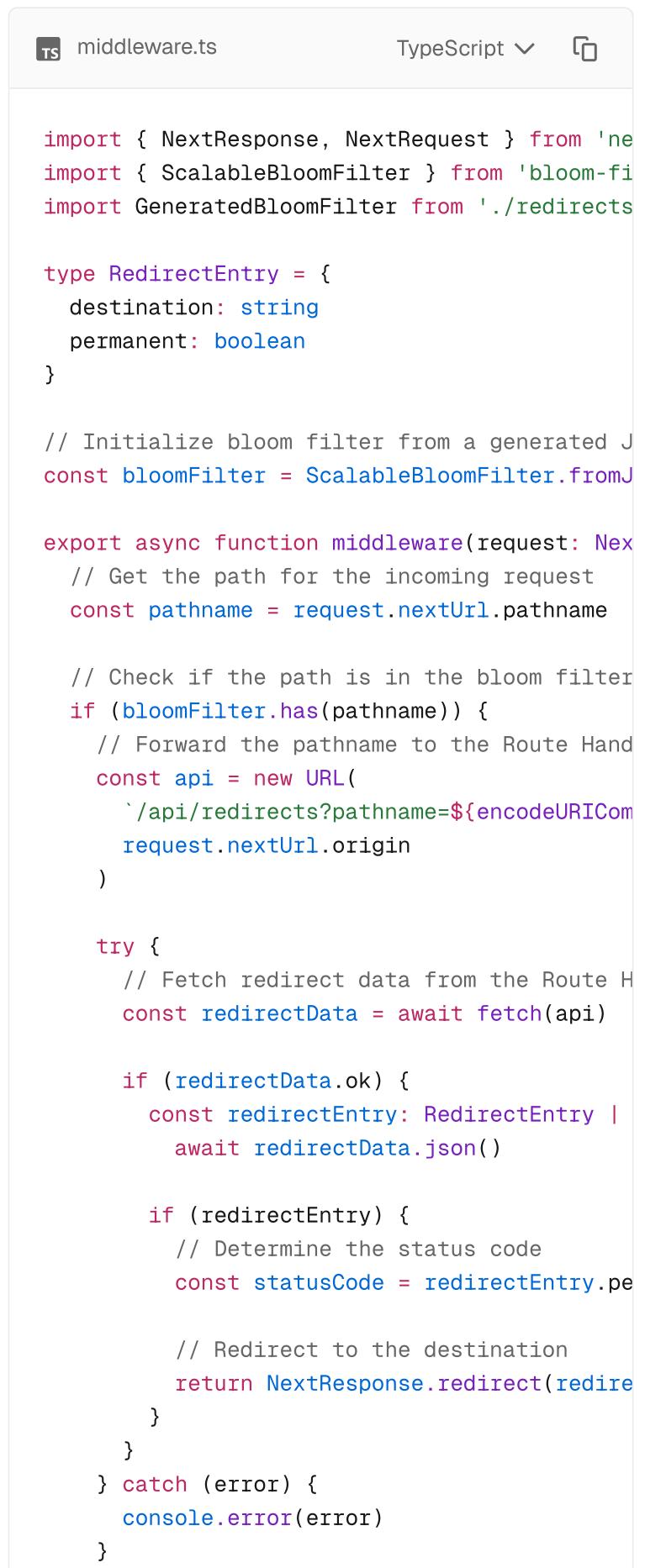
2. Optimizing data lookup performance

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
- 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 [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.



The screenshot shows a code editor window with the file name "middleware.ts" and a "TypeScript" dropdown menu. The code itself is a TypeScript file that imports NextResponse and NextRequest from 'next', ScalableBloomFilter from 'bloom-filters', and GeneratedBloomFilter from './redirects'. It defines a RedirectEntry type with properties destination (string) and permanent (boolean). The export async function middleware takes a NextRequest as an argument. It initializes a bloom filter from a generated JSON file, gets the pathname from the request, and checks if the pathname is in the bloom filter. If it is, it forwards the pathname to a Route Handler using a new URL to the '/api/redirects?pathname' endpoint with the origin from the request. It then tries to fetch redirect data from this endpoint. If successful, it checks if the redirect entry exists and determines the status code. Finally, it returns a redirect response with the destination from the redirect entry.

```
import { NextResponse, NextRequest } from 'next'
import { ScalableBloomFilter } from 'bloom-filters'
import GeneratedBloomFilter from './redirects'

type RedirectEntry = {
  destination: string
  permanent: boolean
}

// Initialize bloom filter from a generated JSON file
const bloomFilter = ScalableBloomFilter.fromJSONFile('bloom.json')

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=${encodeURIComponent(pathname)}&origin=${request.nextUrl.origin}`
    )

    try {
      // Fetch redirect data from the Route Handler
      const redirectData = await fetch(api)

      if (redirectData.ok) {
        const redirectEntry: RedirectEntry | null = await redirectData.json()

        if (redirectEntry) {
          // Determine the status code
          const statusCode = redirectEntry.permanent ? 301 : 302

          // Redirect to the destination
          return NextResponse.redirect(`https://${request.nextUrl.host}${redirectEntry.destination}`, { statusCode })
        }
      }
    } catch (error) {
      console.error(error)
    }
  }
}
```

```
}
```

```
// No redirect found, continue the request  
return NextResponse.next()
```

```
}
```

Then, in the API Route:

```
ts pages/api/redirects.ts TypeScript ▾
```

```
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 pathname' })  
  }  
  
  // Get the redirect entry from the redirect 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 found' })  
  }  
  
  // 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.

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to use Sass in Next.js

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` :

>_ Terminal



```
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.js`.

next.config.ts

TypeScript



```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  sassOptions: {
    additionalData: `$var: red;`,
  },
}

export default nextConfig
```

Implementation

You can use the `implementation` property to specify the Sass implementation to use. By default, Next.js uses the [sass](#) package.



next.config.ts TypeScript

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  sassOptions: {
    implementation: 'sass-embedded',
  },
}

export default nextConfig
```

Sass Variables

Next.js supports Sass variables exported from CSS Module files.

For example, using the exported `primaryColor` Sass variable:



app/variables.module.scss

```
$primary-color: #64ff00;

:export {
  primaryColor: $primary-color;
}
```



```
import variables from '../styles/variables.m

export default function MyApp({ Component, pa
    return (
        <Layout color={variables.primaryColor}>
            <Component {...pageProps} />
        </Layout>
    )
}
```

Was this helpful?



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to load and optimize scripts

Application Scripts

To load a third-party script for all routes, import `next/script` and include the script directly in your custom `_app`:

pages/_app.js

```
import Script from 'next/script'

export default function MyApp({ Component, pageProps }) {
  return (
    <>
      <Component {...pageProps} />
      <Script src="https://example.com/script" />
    </>
  )
}
```

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 Router. 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.

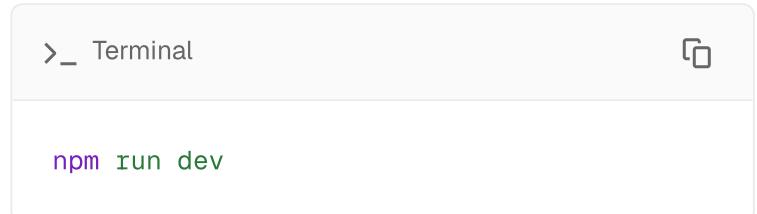
This strategy is still experimental and can only be used if the `nextScriptWorkers` flag is enabled in `next.config.js`:

`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:



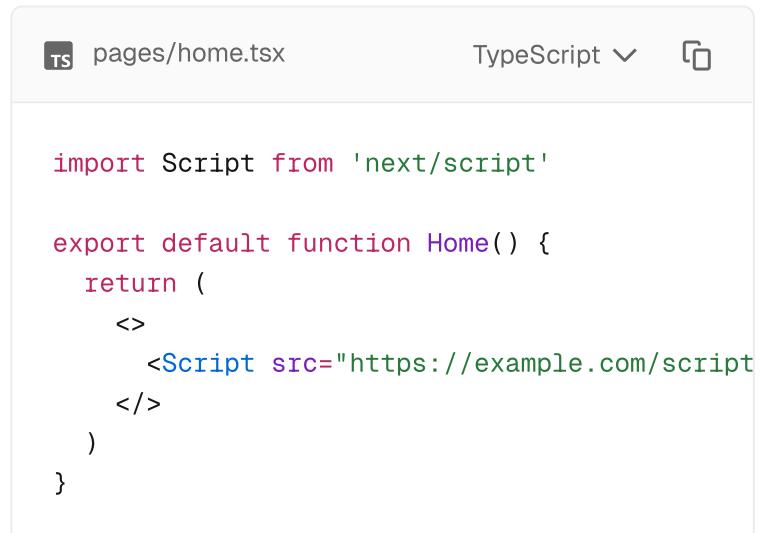
A screenshot of a terminal window titled "Terminal". It contains a single command: `npm run dev`, which is highlighted in green.

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.



A screenshot of a code editor showing a file named `pages/home.tsx`. The file contains the following code:

```
import Script from 'next/script'

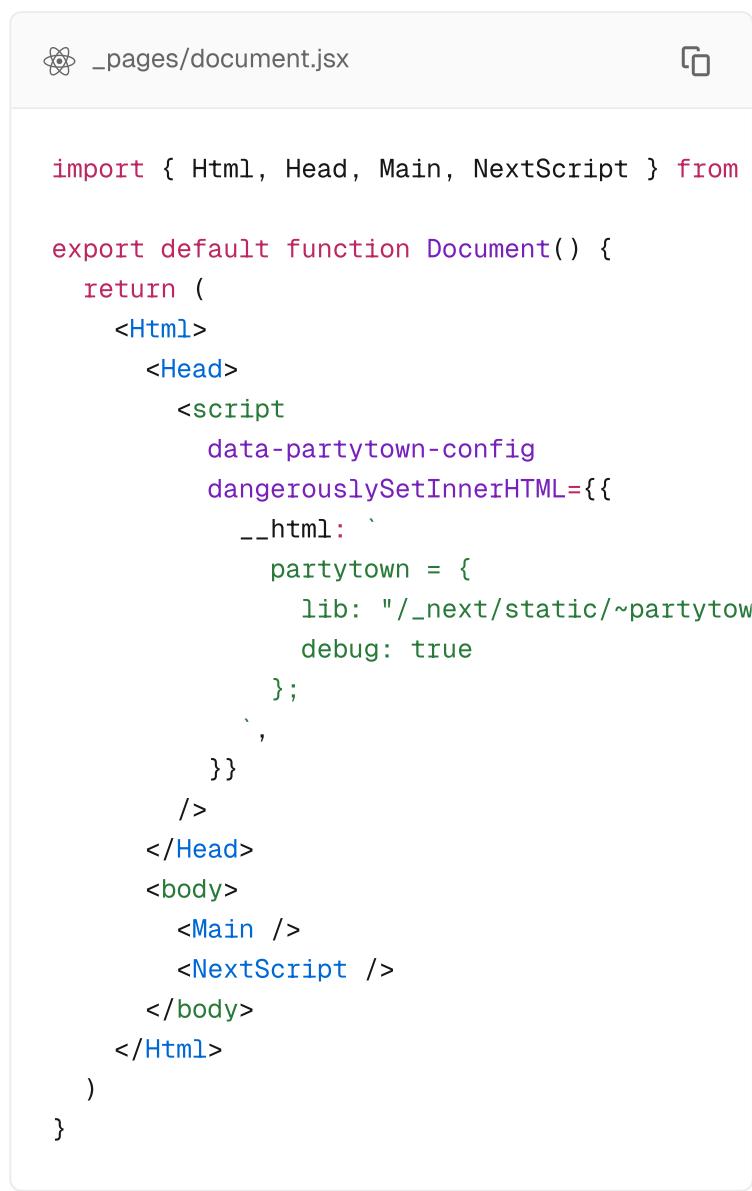
export default function Home() {
  return (
    <>
      <Script src="https://example.com/script" />
    </>
  )
}
```

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`:



The screenshot shows a code editor window with the file name '_pages/document.jsx' at the top. The code inside the file is as follows:

```
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.add('show')`}
</Script>
```

Or by using the `dangerouslySetInnerHTML` property:

```
<Script
  id="show-banner"
  dangerouslySetInnerHTML={{
    __html: `document.getElementById('banner').classList.add('show')`
  }}
/>
```

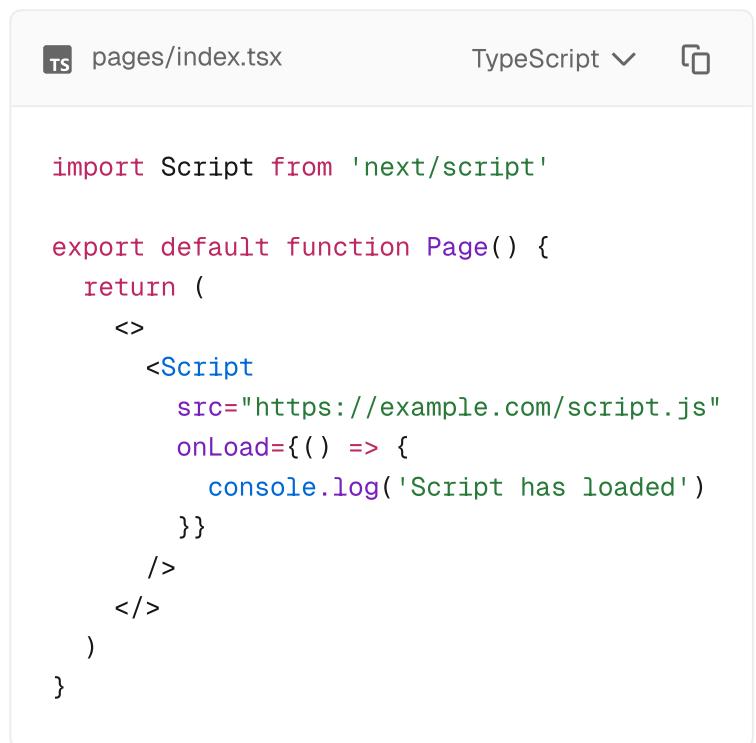
Warning: An `id` property must be assigned for inline scripts in order for Next.js to track and optimize the script.

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:



The screenshot shows a code editor window with a TypeScript file named `pages/index.tsx`. The code defines a `Page` component that contains a `<Script>` element. The `src` attribute is set to `'https://example.com/script.js'`. An `onLoad` event handler is attached to the `<Script>` element, which logs the message `'Script has loaded'` to the console.

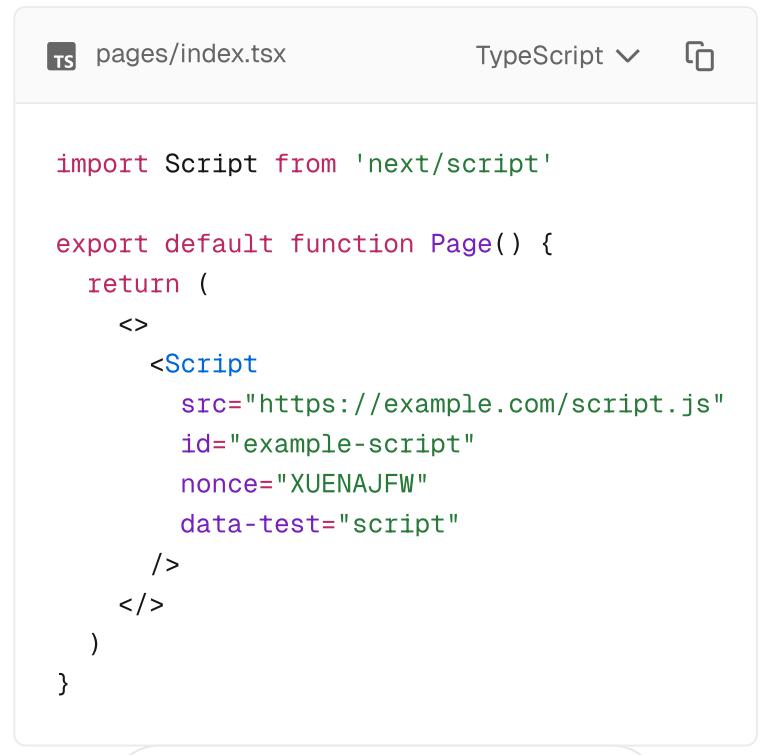
```
import Script from 'next/script'

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.



The screenshot shows a code editor window with the following details:

- File: pages/index.tsx
- TypeScript version: v
- Code content:

```
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"
    />
    </>
  )
}
```

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to self-host your Next.js application

When [deploying](#) your Next.js app, you may want to configure how different features are handled based on your infrastructure.



Watch: Learn more about self-hosting Next.js → [YouTube \(45 minutes\)](#) ↗.

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 the [Edge runtime](#), 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. If you do not want this, you can use the [full Node.js runtime](#) to run Middleware.

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](#).

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:

```
JS next.config.js

module.exports = {
  cacheHandler: require.resolve('./cache-hand
  cacheMaxMemorySize: 0, // disable default i
}
```

Then, create `cache-handler.js` in the root of your project, for example:

```
JS cache-handler.js

const cache = new Map()

module.exports = class CacheHandler {
  constructor(options) {
    this.options = options
  }

  async get(key) {
    // This could be stored anywhere, like du
    return cache.get(key)
  }

  async set(key, data, ctx) {
    // This could be stored anywhere, like du
    cache.set(key, {
      value: data,
      lastModified: Date.now(),
      tags: ctx.tags,
    })
  }

  async revalidateTag(tags) {
    // tags is either a string or an array of
```

```
tags = [tags].flat()
// Iterate over all entries in the cache
for (let [key, value] of cache) {
  // If the value's tags include the spec
  if (value.tags.some((tag) => tags.inclu
    cache.delete(key)
  }
}
}

// If you want to have temporary in memory
// before the next request you can leverage
resetRequestCache() {}

}
```

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`:

`next.config.js`



```
module.exports = {
  generateBuildId: async () => {
    // This could be anything, using the late
    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.

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`.

 package.json



```
{  
  "scripts": {  
    "dev": "next dev",  
    "build": "next build",  
    "start": "NEXT_MANUAL_SIG_HANDLE=true nex  
  }  
}
```

 pages/_document.js



```
if (process.env.NEXT_MANUAL_SIG_HANDLE) {  
  process.on('SIGTERM', () => {  
    console.log('Received SIGTERM: cleaning up')  
    process.exit(0)  
  })  
  process.on('SIGINT', () => {  
    console.log('Received SIGINT: cleaning up')  
    process.exit(0)  
  })  
}
```

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to create a static export of your Next.js application

Next.js enables starting as a static site or Single-Page Application (SPA), then later optionally upgrading to use features that require a server.

When running `next build`, Next.js generates an HTML file per route. By breaking a strict SPA into individual HTML files, Next.js can avoid loading unnecessary JavaScript code on the client-side, reducing the bundle size and enabling faster page loads.

Since Next.js supports this static export, it can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets.

Configuration

To enable a static export, change the output mode inside `next.config.js`:

```
JS next.config.js 
```

```
/**  
 * @type {import('next').NextConfig}  
 */
```

```
const nextConfig = {
  output: 'export',
  // Optional: Change links `/me` -> `/me/` a
  // trailingSlash: true,
  // Optional: Prevent automatic `/me` -> `/m
  // skipTrailingSlashRedirect: true,
  // Optional: Change the output directory `o
  // distDir: 'dist',
}

module.exports = nextConfig
```

After running `next build`, Next.js will create an `out` folder with the HTML/CSS/JS assets for your application.

You can utilize `getStaticProps` and `getStaticPaths` to generate an HTML file for each page in your `pages` directory (or more for [dynamic routes](#)).

Supported Features

The majority of core Next.js features needed to build a static site are supported, including:

- [Dynamic Routes when using `getStaticPaths`](#)
- Prefetching with `next/link`
- Preloading JavaScript
- [Dynamic Imports](#)
- Any styling options (e.g. CSS Modules, styled-`jsx`)
- [Client-side data fetching](#)
- [`getStaticProps`](#)
- [`getStaticPaths`](#)

Image Optimization

[Image Optimization](#) through `next/image` can be used with a static export by defining a custom image loader in `next.config.js`. For example, you can optimize images with a service like Cloudinary:

```
JS next.config.js

/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export',
  images: {
    loader: 'custom',
    loaderFile: './my-loader.ts',
  },
}

module.exports = nextConfig
```

This custom loader will define how to fetch images from a remote source. For example, the following loader will construct the URL for Cloudinary:

```
TS my-loader.ts TypeScript ▾

export default function cloudinaryLoader({
  src,
  width,
  quality,
}: {
  src: string
  width: number
  quality?: number
}) {
  const params = ['f_auto', 'c_limit', `w_${width}`]
  return `https://res.cloudinary.com/demo/image/f_auto,c_limit,w_${width}` + 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.j
}
```

Unsupported Features

Features that require a Node.js server, or dynamic logic that cannot be computed during the build process, are **not** supported:

- Internationalized Routing
- API Routes
- Rewrites
- Redirects
- Headers
- Middleware
- Incremental Static Regeneration
- 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:

```
nginx.conf

server {
  listen 80;
  server_name acme.com;

  root /var/www/out;

  location / {
    try_files $uri $uri.html $uri/ =404;
  }

  # This is necessary when `trailingSlash: false`
  # You can omit this when `trailingSlash: true`
  location /blog/ {
    rewrite ^/blog/(.*)$ /blog/$1.html break;
  }

  error_page 404 /404.html;
  location = /404.html {
    internal;
  }
}
```

Version History

Version Changes

v14.0.0 `next export` has been removed in favor of
`"output": "export"`

v13.4.0 App Router (Stable) adds enhanced static
export support, including using React Server
Components and Route Handlers.

v13.3.0 `next export` is deprecated and replaced
with `"output": "export"`

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



Tailwind CSS

This guide will walk you through how to install Tailwind CSS v3 [↗](#) in your Next.js application.

Good to know: For the latest Tailwind 4 setup, see the [Tailwind CSS setup instructions](#).

Installing Tailwind v3

Install Tailwind CSS and its peer dependencies, then run the `init` command to generate both `tailwind.config.js` and `postcss.config.js` files:

```
pnpm npm yarn bun  
>_ Terminal   
pnpm add -D tailwindcss@^3 postcss autoprefixer  
npx tailwindcss init -p
```

Configuring Tailwind v3

Configure your template paths in your `tailwind.config.js` file:

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    './pages/**/*.{js,ts,jsx,tsx,mdx}',
    './components/**/*.{js,ts,jsx,tsx,mdx}',
    './app/**/*.{js,ts,jsx,tsx,mdx}',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Add the Tailwind directives to your global CSS file:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Import the CSS file in your `pages/_app.js` file:

```
import '@/styles/globals.css'

export default function MyApp({ 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 un  
}
```

Usage with Turbopack

As of Next.js 13.1, Tailwind CSS and PostCSS are supported with [Turbopack ↗](#).

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



Testing

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.

Guides

See the guides below to learn how to set up Next.js with these commonly used testing tools:

Cypress

Learn how to set up Next.js with Cypress for End-to-End Testing.

Jest

Learn how to set up Next.js with Jest for Unit Testing.

Playwright

Learn how to set up Next.js with Playwright for Headless UI Testing.

Vitest

Learn how to set up Next.js with Vitest and React Testing Library.

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to set up Cypress with Next.js

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:

- 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 ↗](#)

Manual setup

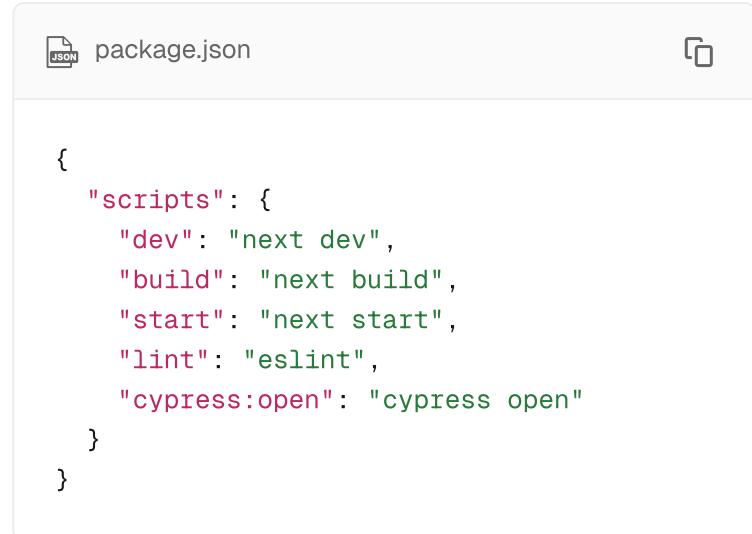
To manually set up Cypress, install `cypress` as a dev dependency:

>_ Terminal



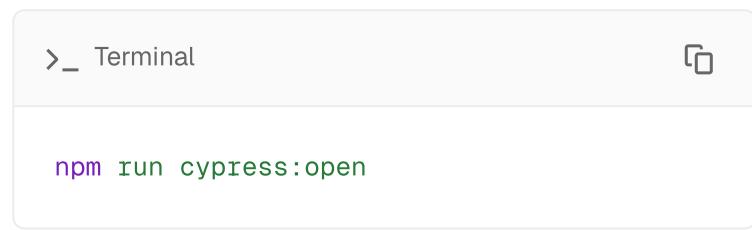
```
npm install -D cypress
# or
yarn add -D cypress
# or
pnpm install -D cypress
```

Add the Cypress `open` command to the `package.json` scripts field:



```
{  
  "scripts": {  
    "dev": "next dev",  
    "build": "next build",  
    "start": "next start",  
    "lint": "eslint",  
    "cypress:open": "cypress open"  
  }  
}
```

Run Cypress for the first time to open the Cypress testing suite:



```
>_ Terminal  
  
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` file has the following configuration:



```
TS cypress.config.ts TypeScript ▾  
  
import { defineConfig } from 'cypress'  
  
export default defineConfig({
```

```
e2e: {
  setupNodeEvents(on, config) {},
  },
})
```

Then, create two new Next.js files:

JS pages/index.js

```
import Link from 'next/link'

export default function Home() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

JS pages/about.js

```
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:

JS cypress/e2e/app.cy.js

```
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"
    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
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.

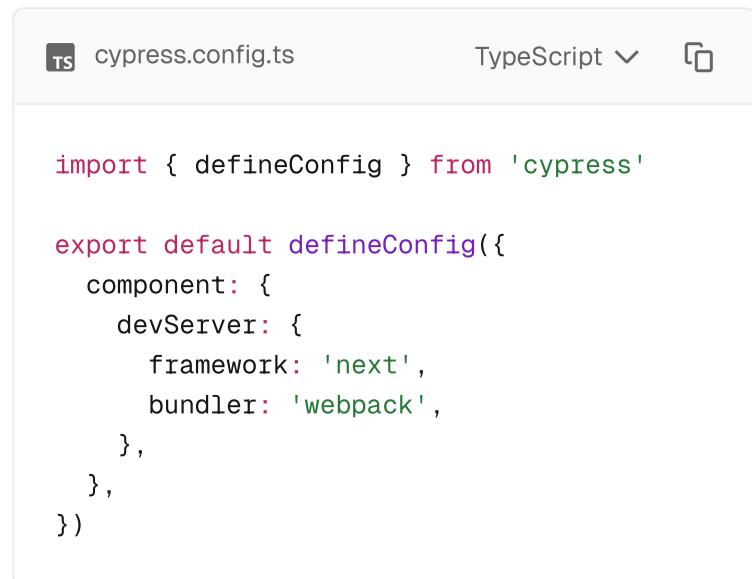
Creating your first Cypress component test

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` file has the following configuration:

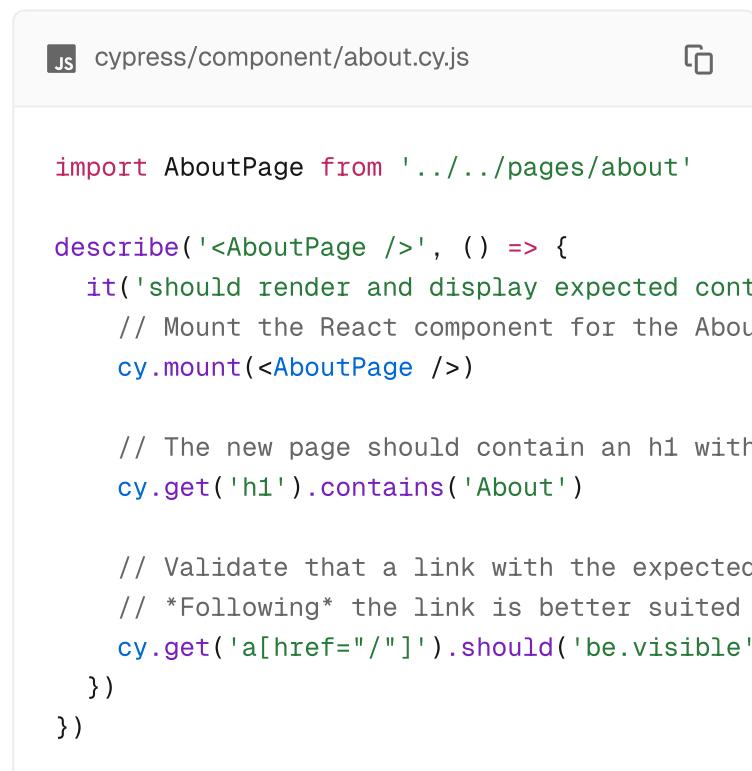


```
TS cypress.config.ts TypeScript ▾
```

```
import { defineConfig } from 'cypress'

export default 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:



```
JS cypress/component/about.cy.js
```

```
import AboutPage from '../../../../../pages/about'

describe('<AboutPage />', () => {
  it('should render and display expected content')
  // Mount the React component for the AboutPage
  cy.mount(<AboutPage />

  // The new page should contain an h1 with
  // cy.get('h1').contains('About')

  // Validate that a link with the expected href
  // *Following* the link is better suited
  // 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:



A screenshot of a code editor showing a `package.json` file. The file contains the following JSON code:

```
{  
  "scripts": {  
    //...  
    "e2e": "start-server-and-test dev http://  
    "e2e:headless": "start-server-and-test de  
    "component": "cypress open --component",  
    "component:headless": "cypress run --comp  
  }  
}
```

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 ↗](#)

Was this helpful?





Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

How to set up Jest with Next.js

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:

>_ Terminal



```
npx create-next-app@latest --example with-jes
```

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:

>_ Terminal

```
npm install -D jest jest-environment-jsdom @t
# or
yarn add -D jest jest-environment-jsdom @test
# or
pnpm install -D jest jest-environment-jsdom @
```

Generate a basic Jest configuration file by running the following command:

>_ Terminal

```
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 jest.config.ts

TypeScript ▾

```
import type { Config } from 'jest'
import nextJest from 'next/jest'

const createJestConfig = nextJest({
  // Provide the path to your Next.js app to
  dir: './',
})

// Add any custom config to be passed to Jest
```

```
const config: Config = {
  coverageProvider: 'v8',
  testEnvironment: 'jsdom',
  // Add more setup options before each test
  // setupFilesAfterEnv: ['<rootDir>/jest.set
}

// createJestConfig is exported this way to e
export default 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:

```
js jest.config.js

module.exports = {
  collectCoverage: true,
  // on node 14.x coverage provider v8 offers
  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
    '^.+\\.module\\.(css|sass|scss)$': 'ident'
      // Handle CSS imports (without CSS module
      // https://jestjs.io/docs/webpack#mocking
      '^.+\\.\\.(css|sass|scss)$': '<rootDir>/_mo
        // Handle image imports
        // https://jestjs.io/docs/webpack#handling-images
        '^.+\\.(png|jpg|jpeg|gif|webp|avif|ico|bm
          // Handle module aliases
          '^@/components/(.*)$': '<rootDir>/compone
            // Handle @next/font
            '@next/font/(.*)': `<rootDir>/__mocks__/_n
              // Handle next/font
              'next/font/(.*)': `<rootDir>/__mocks__/_ne
              // Disable server-only
              'server-only': `<rootDir>/__mocks__/_empty
            },
            // Add more setup options before each test
            // setupFilesAfterEnv: ['<rootDir>/jest.set
            testPathIgnorePatterns: ['<rootDir>/node_mo
            testEnvironment: 'jsdom',
            transform: {
              // Use babel-jest to transpile tests with
              // https://jestjs.io/docs/configuration#t
              '^.+\\.(js|jsx|ts|tsx)$': ['babel-jest',
            },
            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:

```
JS __mocks__/fileMock.js
```

```
module.exports = 'test-file-stub'
```

```
JS __mocks__/styleMock.js
```

```
module.exports = {}
```

For more information on handling static assets, please refer to the [Jest Docs](#).

Handling Fonts

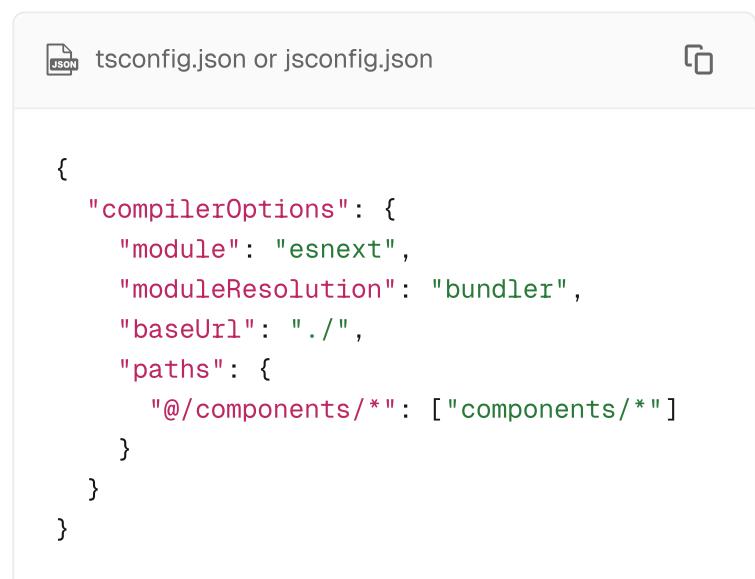
To handle fonts, create the `nextFontMock.js` file inside the `__mocks__` directory, and add the following configuration:

```
JS __mocks__/nextFontMock.js
```

```
module.exports = new Proxy(
  {},
  {
    get: function getter() {
      return () => ({
        className: 'className',
        variable: 'variable',
        style: { fontFamily: 'fontFamily' },
      })
    },
  }
)
```

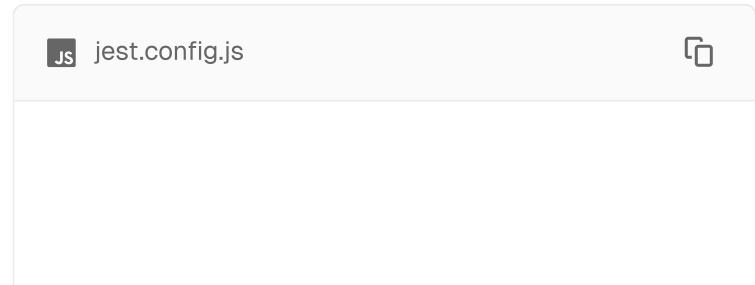
Optional: Handling Absolute Imports and Module Path Aliases

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:



The screenshot shows a code editor window with a JSON configuration file. The file contains the following code:

```
{
  "compilerOptions": {
    "module": "esnext",
    "moduleResolution": "bundler",
    "baseUrl": "./",
    "paths": {
      "@components/*": ["components/*"]
    }
  }
}
```



The screenshot shows a code editor window with a JavaScript configuration file. The file contains the following code:

```
module.exports = {
  moduleNameMapper: {
    '^@/(.*)$': '/src/$1'
  }
}
```

```
moduleNameMapper: {  
  // ...  
  '^@/components/(.*)$': '<rootDir>/component'  
}
```

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:

```
TS jest.config.ts TypeScript ▾   
  
setupFilesAfterEnv: [<rootDir>/jest.setup.ts]
```

Then, inside `jest.setup`, add the following import:

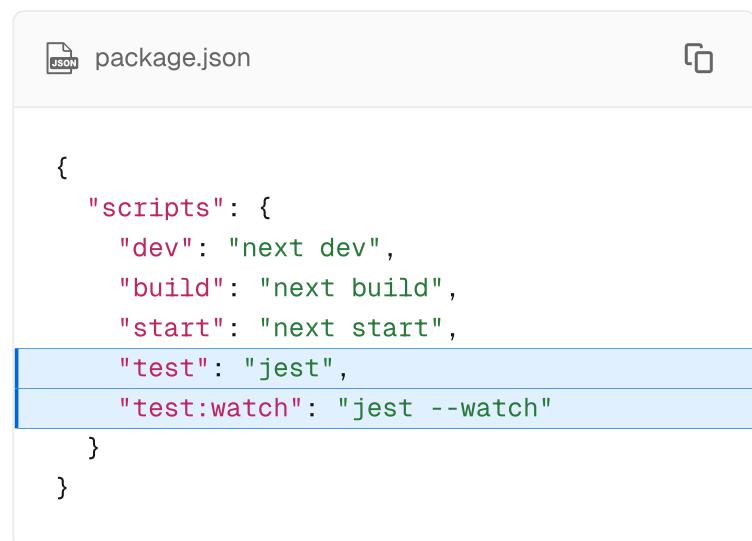
```
TS jest.setup.ts TypeScript ▾   
  
import '@testing-library/jest-dom'
```

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:



```
JSON package.json

{
  "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>
}
```



```
JS __tests__/index.test.js

import '@testing-library/jest-dom'
import { render, screen } from '@testing-libr
```

```
import Home from '../pages/index'

describe('Home', () => {
  it('renders a heading', () => {
    render(<Home />

    const heading = screen.getByRole('heading')

    expect(heading).toBeInTheDocument()
  })
})
```

Optionally, add a [snapshot test ↗](#) to keep track of any unexpected changes in your component:

JS __tests__/snapshot.js

```
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.

Running your tests

Then, run the following command to run your tests:

>_ Terminal

```
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.

Was this helpful?    

[Using Pages Router](#)

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.

[Latest Version](#)

15.5.4



How to set up Playwright with Next.js

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.

> Terminal



```
npx create-next-app@latest --example with-pla
```

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 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:

tests/example.spec.ts

```
import { test, expect } from '@playwright/test'

test('should navigate to the about page', async () => {
  // Start from the index page (the baseURL is automatically set to http://localhost:3000)
  await page.goto('http://localhost:3000/')
  // Find an element with the text 'About' and click it
  await page.click('text=About')
  // The new URL should be "/about" (baseURL is automatically set to http://localhost:3000)
  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 ↗](#)

Was this helpful?    



Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

How to set up Vitest with Next.js

Vitest 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 **E2E tests** for `async` components.

Quickstart

You can use `create-next-app` with the Next.js [with-vitest ↗](#) example to quickly get started:

```
>_ Terminal   
npx create-next-app@latest --example with-vit
```

Manual Setup

To manually set up Vitest, install `vitest` and the following packages as dev dependencies:

> Terminal

```
# Using TypeScript
npm install -D vitest @vitejs/plugin-react js
# Using JavaScript
npm install -D vitest @vitejs/plugin-react js
```

Create a `vitest.config.mts|js` file in the root of your project, and add the following options:

vitest.config.mts

TypeScript ▾

```
import { defineConfig } from 'vitest/config'
import react from '@vitejs/plugin-react'
import tsconfigPaths from 'vite-tsconfig-path'

export default defineConfig({
  plugins: [tsconfigPaths(), 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`:

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:

```
TS pages/index.tsx TypeScript ▾
```

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
TS __tests__/index.test.tsx TypeScript ▾
```

```
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 })).toBeInTheDocument()
})
```

Running your tests

Then, run the following command to run your tests:

```
>_ Terminal
```

```
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 ↗](#)

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to optimize third-party libraries

`@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:

>_ Terminal



```
npm install @next/third-parties@latest next@1
```

`@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 custom `_app` and pass in your GTM container ID:

```
js pages/_app.js

import { GoogleTagManager } from '@next/third

export default function MyApp({ Component, pa
  return (
    <>
      <Component {...pageProps} />
      <GoogleTagManager gtmId="GTM-XYZ" />
    </>
  )
}
```

To load Google Tag Manager for a single route, include the component in your page file:

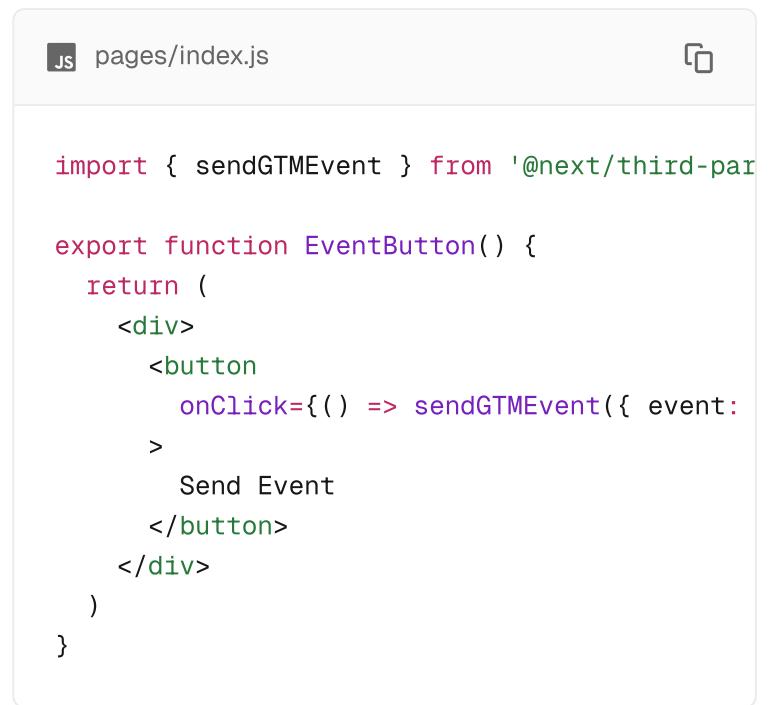
```
js pages/index.js

import { GoogleTagManager } from '@next/third

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.



```
JS pages/index.js

import { sendGTMEvent } from '@next/third-par

export function EventButton() {
  return (
    <div>
      <button
        onClick={() => sendGTMEvent({ event:
      >
        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 in the form GTM- .
gtmScriptUrl	Optional	GTM script URL. Defaults to https://www.googletagmanager.com
dataLayer	Optional	Data layer object to instantiate.
dataLayerName	Optional	Name of the data layer. Defaults to <code>dataLayer</code> .
auth	Optional	Value of authentication parameter () for environment snippets.
preview	Optional	Value of preview parameter 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 custom `_app` and pass in your measurement ID:

`JS pages/_app.js`

```
import { GoogleAnalytics } from '@next/third-
```

```
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:

JS pages/index.js

```
import { GoogleAnalytics } from '@next/third-party'

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.

JS pages/index.js

```
import { sendGAEvent } from '@next/third-party'

export function EventButton() {
  return (
    <div>
      <button
        onClick={() => sendGAEvent('event', 'button-clicked')}
      >
        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.

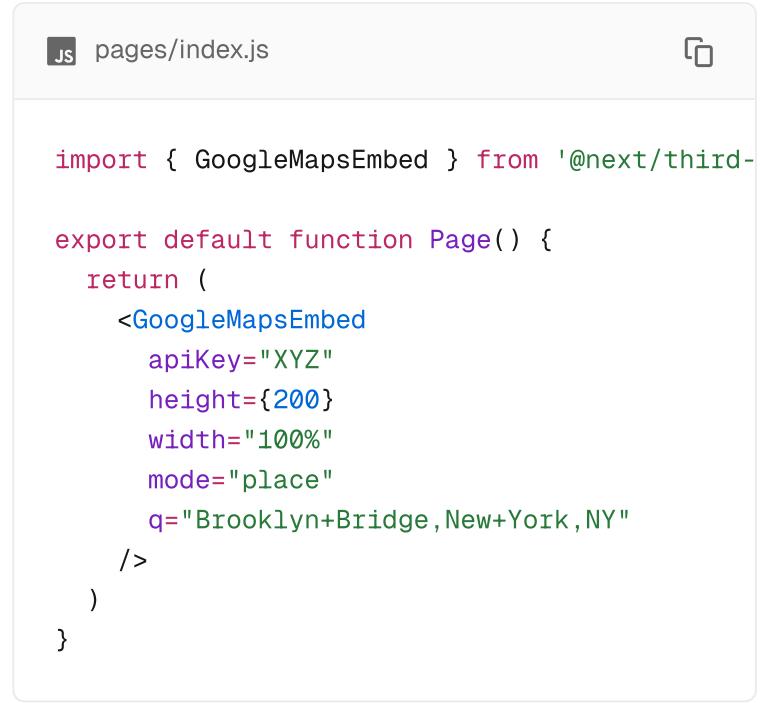
Options

Options to pass to the `<GoogleAnalytics>` component.

Name	Type	Description
<code>gaId</code>	Required	Your measurement ID ↗. Usually starts with G-.
<code>dataLayerName</code>	Optional	Name of the data layer. Defaults to <code>dataLayer</code> .
<code>nonce</code>	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.



```
JS pages/index.js

import { GoogleMapsEmbed } from '@next/third-party'

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
<code>apiKey</code>	Required	Your api key.
<code>mode</code>	Required	Map mode ↗
<code>height</code>	Optional	Height of the embed. Defaults to <code>auto</code> .
<code>width</code>	Optional	Width of the embed. Defaults to <code>auto</code> .
<code>style</code>	Optional	Pass styles to the iframe.
<code>allowfullscreen</code>	Optional	Property to allow certain map parts to go full screen.
<code>loading</code>	Optional	Defaults to <code>lazy</code> . Consider changing if you know

Name	Type	Description
		your embed will be above the fold.
q	Optional	Defines map marker location. <i>This may be required depending on the map mode.</i>
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 geopolitical 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.

```
JS pages/index.js

import { YouTubeEmbed } from '@next/third-par

export default function Page() {
  return <YouTubeEmbed videoid="ogfYd705cRs"
}
```

Options

Name	Type	Description
videoid	Required	YouTube video id.
width	Optional	Width of the video container. Default: auto
height	Optional	Height of the video container. Default: auto
playlabel	Optional	A visually hidden label for the play button for accessibility.
params	Optional	The video player params defined. Params are passed as a query parameter string. Eg: <code>params="controls=0&start=108"</code>
style	Optional	Used to apply styles to the video container.

Was this helpful?     



Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

Upgrading

Learn how to upgrade to the latest versions of Next.js following the versions-specific guides:

Codemods

Use codemods to upgrade your Next.js codebase...

Version 10

Upgrade your Next.js Application from Version 9 to...

Version 11

Upgrade your Next.js Application from Version 10 to...

Version 12

Upgrade your Next.js Application from Version 11 to...

Version 13

Upgrade your Next.js Application from Version 12 to...

Version 14

Upgrade your Next.js Application from Version 13 to...

Version 9

Upgrade your
Next.js Application
from Version 8 to...

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



Codemods

Codemods are transformations that run on your codebase programmatically. This allows a large number of changes to be programmatically applied without having to manually go through every file.

Next.js provides Codemod transformations to help upgrade your Next.js codebase when an API is updated or deprecated.

Usage

In your terminal, navigate (`cd`) into your project's folder, then run:

```
>_ Terminal   
npx @next/codemod <transform> <path>
```

Replacing `<transform>` and `<path>` with appropriate values.

- `transform` - name of transform
- `path` - files or directory to transform
- `--dry` Do a dry-run, no code will be edited

- `--print` Prints the changed output for comparison
-

Codemods

16.0

Migrate from `next lint` to ESLint CLI

`next-lint-to-eslint-cli`

```
>_ Terminal   
npx @next/codemod@canary next-lint-to-eslint-
```

This codemod migrates projects from using `next lint` to using the ESLint CLI with your local ESLint config. It:

- Creates an `eslint.config.mjs` file with Next.js recommended configurations
- Updates `package.json` scripts to use `eslint .` instead of `next lint`
- Adds necessary ESLint dependencies to `package.json`
- Preserves existing ESLint configurations when found

For example:

```
 package.json   
{  
  "scripts": {  
    "lint": "next lint"  
  }  
}
```

Becomes:



```
JSON package.json
```

```
{  
  "scripts": {  
    "lint": "eslint ."  
  }  
}
```

And creates:



```
JS eslint.config.mjs
```

```
import { dirname } from 'path'  
import { fileURLToPath } from 'url'  
import { FlatCompat } from '@eslint/eslintrc'  
  
const __filename = fileURLToPath(import.meta.url)  
const __dirname = dirname(__filename)  
  
const compat = new FlatCompat({  
  baseDirectory: __dirname,  
})  
  
const eslintConfig = [  
  ...compat.extends('next/core-web-vitals', {  
    ignores: [  
      'node_modules/**',  
      '.next/**',  
      'out/**',  
      'build/**',  
      'next-env.d.ts',  
    ],  
  }),  
]  
  
export default eslintConfig
```

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.

>_ Terminal



```
npx @next/codemod@latest app-dir-runtime-conf
```

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`

>_ Terminal



```
npx @next/codemod@latest next-async-request-a
```

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/header'
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,
  type UnsafeUnwrappedHeaders,
} from 'next/headers'
const token = (cookies() as unknown as Unsafe

function useToken() {
  const token = use(cookies()).get('token')
  return token
}

export default async function Page() {
  const name = (await cookies()).get('name')
}

function getHeader() {
  return (headers() as unknown as UnsafeUnwra
}
```

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 | str
}) {
  const { value } = searchParams
  if (value === 'foo') {
    // ...
  }
}

export function generateMetadata({ params }: {
  const { slug } = params
  return {
    title: `My Page - ${slug}`,
  }
})
```

Transforms into:

```
// page.tsx
export default async function Page(props: {
  params: Promise<{ slug: string }>
  searchParams: Promise<{ [key: string]: str
}) {
  const searchParams = await props.searchPara
  const { value } = searchParams
  if (value === 'foo') {
    // ...
  }
}

export async function generateMetadata(props: {
  params: Promise<{ slug: string }>
}) {
  const params = await props.params
  const { slug } = 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
```

>_ Terminal



```
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 '@verc

export function GET(req: NextRequest) {
  const geo = geolocation(req)
  const ip = ipAddress(req)
}
```

14.0

Migrate `ImageResponse` imports

`next-og-import`

>_ Terminal



```
npx @next/codemod@latest next-og-import .
```

This codemod moves transforms imports from `next/server` to `next/og` for usage of [Dynamic OG Image Generation](#).

For example:

```
import { ImageResponse } from 'next/server'
```

Transforms into:

```
import { ImageResponse } from 'next/og'
```

Use `viewport` export

`metadata-to-viewport-export`

>_ Terminal



```
npx @next/codemod@latest metadata-to-viewport
```

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`

>_ Terminal



```
npx @next/codemod@latest built-in-next-font .
```

This codemod uninstalls the `@next/font` package and transforms `@next/font` imports into the built-in `next/font`.

For example:

```
import { Inter } from '@next/font/google'
```

Transforms into:

```
import { Inter } from 'next/font/google'
```

13.0

Rename Next Image Imports

```
next-image-to-legacy-image
```

>_ Terminal



```
npx @next/codemod@latest next-image-to-legacy
```

Safely renames `next/image` imports in existing Next.js 10, 11, or 12 applications to `next/legacy/image` in Next.js 13. Also renames `next/future/image` to `next/image`.

For example:

js pages/index.js



```
import Image1 from 'next/image'
import Image2 from 'next/future/image'

export default function Home() {
  return (
    <div>
      <Image1 src="/test.jpg" width="200" hei
      <Image2 src="/test.png" width="500" hei
    </div>
  )
}
```

Transforms into:

js pages/index.js



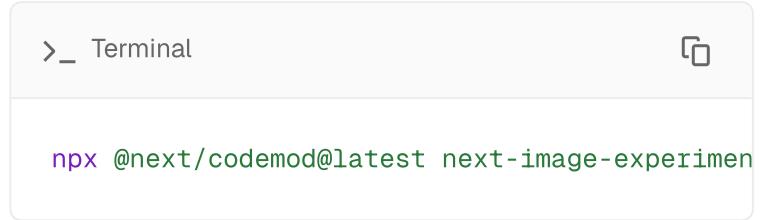
```
// 'next/image' becomes 'next/legacy/image'
import Image1 from 'next/legacy/image'
// 'next/future/image' becomes 'next/image'
import Image2 from 'next/image'

export default function Home() {
```

```
    return (
      <div>
        <Image1 src="/test.jpg" width="200" hei
        <Image2 src="/test.png" width="500" hei
      </div>
    )
}
```

Migrate to the New Image Component

next-image-experimental



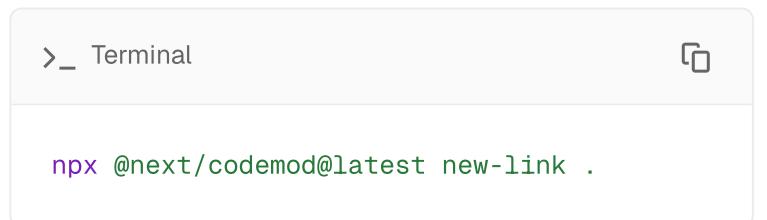
```
>_ Terminal
npx @next/codemod@latest next-image-experimental
```

Dangerously migrates from `next/legacy/image` to the new `next/image` by adding inline styles and removing unused props.

- Removes `layout` prop and adds `style`.
- Removes `objectFit` prop and adds `style`.
- Removes `objectPosition` prop and adds `style`.
- Removes `lazyBoundary` prop.
- Removes `lazyRoot` prop.

Remove `<a>` Tags From Link Components

new-link



```
>_ Terminal
npx @next/codemod@latest new-link .
```

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')}>A
</Link>
// transforms into
<Link href="/about" onClick={() => console.lo
  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

>_ Terminal



```
npx @next/codemod cra-to-next
```

Migrates a Create React App project to Next.js; creating a Pages Router and necessary config to match behavior. Client-side only rendering is

leveraged initially to prevent breaking compatibility due to `window` usage during SSR and can be enabled seamlessly to allow the gradual adoption of Next.js specific features.

Please share any feedback related to this transform [in this discussion ↗](#).

10

Add React imports

`add-missing-react-import`

```
>_ Terminal   
npx @next/codemod add-missing-react-import
```

Transforms files that do not import `React` to include the import in order for the new [React JSX transform ↗](#) to work.

For example:

```
JS my-component.js   
  
export default class Home extends React.Component {  
  render() {  
    return <div>Hello World</div>  
  }  
}
```

Transforms into:

```
JS my-component.js   
  
import React from 'react'  
export default class Home extends React.Component {  
  render() {  
    return <div>Hello World</div>  
  }  
}
```

Transform Anonymous Components into Named Components

`name-default-component`

> Terminal



```
npx @next/codemod name-default-component
```

Versions 9 and above.

Transforms anonymous components into named components to make sure they work with [Fast Refresh ↗](#).

For example:

JS my-component.js



```
export default function () {
  return <div>Hello World</div>
}
```

Transforms into:

JS my-component.js



```
export default function MyComponent() {
  return <div>Hello World</div>
}
```

The component will have a camel-cased name based on the name of the file, and it also works with arrow functions.

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:

```
From
```

```
import React from 'react'
export default class extends React.Component {
  render() {
    const { pathname } = this.props.url
    return <div>Current pathname: {pathname}<
      >
  }
}
```

```
To
```

```
import React from 'react'
import { withRouter } from 'next/router'
export default withRouter(
  class extends React.Component {
    render() {
      const { pathname } = this.props.router
      return <div>Current pathname: {pathname}
    }
  }
)
```

This is one case. All the cases that are transformed (and tested) can be found in the [__testfixtures__ directory ↗](#).

Was this helpful?    



Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

How to upgrade to version 10

There were no breaking changes between versions 9 and 10.

To upgrade to version 10, run the following command:

>_ Terminal



```
npm i next@10
```

>_ Terminal



```
yarn add next@10
```

>_ Terminal



```
pnpm up next@10
```

>_ Terminal



```
bun add next@10
```

Good to know: If you are using TypeScript, ensure you also upgrade `@types/react` and `@types/react-dom` to their corresponding versions.

Was this helpful?    



Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to upgrade to version 11

To upgrade to version 11, run the following command:

>_ Terminal



```
npm i next@11 react@17 react-dom@17
```

>_ Terminal



```
yarn add next@11 react@17 react-dom@17
```

>_ Terminal



```
pnpm up next@11 react@17 react-dom@17
```

>_ Terminal



```
bun add next@11 react@17 react-dom@17
```

Good to know: If you are using TypeScript, ensure you also upgrade `@types/react` and `@types/react-dom` to their corresponding versions.

Webpack 5 is now the default for all Next.js applications. If you did not have a custom webpack configuration, your application is already using webpack 5. If you do have a custom webpack configuration, you can refer to the [Next.js webpack 5 documentation](#) for upgrade guidance.

Cleaning the `distDir` is now a default

The build output directory (defaults to `.next`) is now cleared by default except for the Next.js caches. You can refer to [the cleaning `distDir` RFC ↗](#) for more information.

If your application was relying on this behavior previously you can disable the new default behavior by adding the `cleanDistDir: false` flag in `next.config.js`.

`PORT` is now supported for `next dev` and `next start`

Next.js 11 supports the `PORT` environment variable to set the port the application runs on. Using `-p / --port` is still recommended but if you were prohibited from using `-p` in any way you can now use `PORT` as an alternative:

Example:

```
PORT=4000 next start
```

`next.config.js` customization to import images

Next.js 11 supports static image imports with `next/image`. This new feature relies on being able

to process image imports. If you previously added the `next-images` or `next-optimized-images` packages you can either move to the new built-in support using `next/image` or disable the feature:



```
JS next.config.js

module.exports = {
  images: {
    disableStaticImages: true,
  },
}
```

Remove `super.componentDidCatch()` from `pages/_app.js`

The `next/app` component's `componentDidCatch` was deprecated in Next.js 9 as it's no longer needed and has since been a no-op. In Next.js 11, it was removed.

If your `pages/_app.js` has a custom `componentDidCatch` method you can remove `super.componentDidCatch` as it is no longer needed.

Remove `Container` from `pages/_app.js`

This export was deprecated in Next.js 9 as it's no longer needed and has since been a no-op with a warning during development. In Next.js 11 it was removed.

If your `pages/_app.js` imports `Container` from `next/app` you can remove `Container` as it was removed. Learn more in [the documentation](#).

Remove `props.url` usage from page components

This property was deprecated in Next.js 4 and has since shown a warning during development. With the introduction of `getStaticProps` / `getServerSideProps` these methods already disallowed the usage of `props.url`. In Next.js 11, it was removed completely.

You can learn more in [the documentation](#).

Remove `unsized` property on `next/image`

The `unsized` property on `next/image` was deprecated in Next.js 10.0.1. You can use `layout="fill"` instead. In Next.js 11 `unsized` was removed.

Remove `modules` property on `next/dynamic`

The `modules` and `render` option for `next/dynamic` were deprecated in Next.js 9.5. This was done in order to make the `next/dynamic` API closer to `React.lazy`. In Next.js 11, the `modules` and `render` options were removed.

This option hasn't been mentioned in the documentation since Next.js 8 so it's less likely that your application is using it.

If your application does use `modules` and `render` you can refer to [the documentation](#).

Remove `Head.rewind`

`Head.rewind` has been a no-op since Next.js 9.5, in Next.js 11 it was removed. You can safely remove your usage of `Head.rewind`.

Moment.js locales excluded by default

Moment.js includes translations for a lot of locales by default. Next.js now automatically excludes these locales by default to optimize bundle size for applications using Moment.js.

To load a specific locale use this snippet:

```
import moment from 'moment'  
import 'moment/locale/ja'  
  
moment.locale('ja')
```

You can opt-out of this new default by adding `excludeDefaultMomentLocales: false` to `next.config.js` if you do not want the new behavior, do note it's highly recommended to not disable this new optimization as it significantly reduces the size of Moment.js.

Update usage of `router.events`

In case you're accessing `router.events` during rendering, in Next.js 11 `router.events` is no longer provided during pre-rendering. Ensure you're accessing `router.events` in `useEffect`:

```
useEffect(() => {  
  const handleRouteChange = (url, { shallow }  
  console.log(  
    `App is changing to ${url} ${  
      shallow ? 'with' : 'without'  
    } shallow routing`  
  )  
}  
  
router.events.on('routeChangeStart', handle  
  
// If the component is unmounted, unsubscribe  
// from the event with the `off` method:  
return () => {  
  router.events.off('routeChangeStart', han
```

```
    }  
}, [router])
```

If your application uses `router.router.events` which was an internal property that was not public please make sure to use `router.events` as well.

React 16 to 17

React 17 introduced a new [JSX Transform](#) that brings a long-time Next.js feature to the wider React ecosystem: Not having to

```
import React from 'react'
```

When using React 17 Next.js will automatically use the new transform. This transform does not make the `React` variable global, which was an unintended side-effect of the previous Next.js implementation. A [codemod is available](#) to automatically fix cases where you accidentally used `React` without importing it.

Most applications already use the latest version of React, with Next.js 11 the minimum React version has been updated to 17.0.2.

To upgrade you can run the following command:

```
npm install react@latest react-dom@latest
```

Or using `yarn`:

```
yarn add react@latest react-dom@latest
```

Was this helpful?





Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to upgrade to version 12

To upgrade to version 12, run the following command:

>_ Terminal



```
npm i next@12 react@17 react-dom@17 eslint-co
```

>_ Terminal



```
yarn add next@12 react@17 react-dom@17 eslint-
```

>_ Terminal



```
pnpm up next@12 react@17 react-dom@17 eslint-
```

>_ Terminal



```
bun add next@12 react@17 react-dom@17 eslint-
```

Good to know: If you are using TypeScript, ensure you also upgrade `@types/react` and `@types/react-dom` to their corresponding versions.

Upgrading to 12.2

[Middleware](#) - If you were using Middleware prior to 12.2, please see the [upgrade guide](#) for more information.

Upgrading to 12.0

[Minimum Node.js Version ↗](#) - The minimum Node.js version has been bumped from 12.0.0 to 12.22.0 which is the first version of Node.js with native ES Modules support.

[Minimum React Version ↗](#) - The minimum required React version is 17.0.2. To upgrade you can run the following command in the terminal:

A screenshot of a terminal window titled "Terminal". It contains four lines of text representing package managers: npm, yarn, pnpm, and bun, each followed by the command to install react@latest and react-dom@latest.

```
npm install react@latest react-dom@latest

yarn add react@latest react-dom@latest

pnpm update react@latest react-dom@latest

bun add react@latest react-dom@latest
```

SWC replacing Babel

Next.js now uses the Rust-based compiler [SWC ↗](#) to compile JavaScript/TypeScript. This new compiler is up to 17x faster than Babel when compiling individual files and up to 5x faster Fast Refresh.

Next.js provides full backward compatibility with applications that have [custom Babel configuration](#). All transformations that Next.js handles by default like styled-jsx and tree-shaking of `getStaticProps` / `getStaticPaths` / `getServerSideProps` have been ported to Rust.

When an application has a custom Babel configuration, Next.js will automatically opt-out of

using SWC for compiling JavaScript/TypeScript and will fall back to using Babel in the same way that it was used in Next.js 11.

Many of the integrations with external libraries that currently require custom Babel transformations will be ported to Rust-based SWC transforms in the near future. These include but are not limited to:

- Styled Components
- Emotion
- Relay

In order to prioritize transforms that will help you adopt SWC, please provide your `.babelrc` on [this feedback thread ↗](#).

SWC replacing Terser for minification

You can opt-in to replacing Terser with SWC for minifying JavaScript up to 7x faster using a flag in `next.config.js`:

```
JS next.config.js ⚙️  
  
module.exports = {  
  swcMinify: true,  
}
```

Minification using SWC is an opt-in flag to ensure it can be tested against more real-world Next.js applications before it becomes the default in Next.js 12.1. If you have feedback about minification, please leave it on [this feedback thread ↗](#).

Improvements to styled-jsx CSS parsing

On top of the Rust-based compiler we've implemented a new CSS parser based on the one

used for the styled-jsx Babel transform. This new parser has improved handling of CSS and now errors when invalid CSS is used that would previously slip through and cause unexpected behavior.

Because of this change invalid CSS will throw an error during development and `next build`. This change only affects styled-jsx usage.

`next/image` changed wrapping element

`next/image` now renders the `` inside a `` instead of `<div>`.

If your application has specific CSS targeting span such as `.container span`, upgrading to Next.js 12 might incorrectly match the wrapping element inside the `<Image>` component. You can avoid this by restricting the selector to a specific class such as `.container span.item` and updating the relevant component with that className, such as ``.

If your application has specific CSS targeting the `next/image <div>` tag, for example `.container div`, it may not match anymore. You can update the selector `.container span`, or preferably, add a new

`<div className="wrapper">` wrapping the `<Image>` component and target that instead such as `.container .wrapper`.

The `className` prop is unchanged and will still be passed to the underlying `` element.

See the [documentation](#) for more info.

HMR connection now uses a WebSocket

Previously, Next.js used a [server-sent events ↗](#) connection to receive HMR events. Next.js 12 now

uses a WebSocket connection.

In some cases when proxying requests to the Next.js dev server, you will need to ensure the upgrade request is handled correctly. For example, in `nginx` you would need to add the following configuration:

```
location /_next/webpack-hmr {  
    proxy_pass http://localhost:3000/_next/we  
    proxy_http_version 1.1;  
    proxy_set_header Upgrade $http_upgrade;  
    proxy_set_header Connection "upgrade";  
}
```

If you are using Apache (2.x), you can add the following configuration to enable web sockets to the server. Review the port, host name and server names.

```
<VirtualHost *:443>  
    # ServerName yourwebsite.local  
    ServerName "${WEBSITE_SERVER_NAME}"  
    ProxyPass / http://localhost:3000/  
    ProxyPassReverse / http://localhost:3000/  
    # Next.js 12 uses websocket  
    <Location /_next/webpack-hmr>  
        RewriteEngine On  
        RewriteCond %{QUERY_STRING} transport=webso  
        RewriteCond %{HTTP:Upgrade} websocket [NC]  
        RewriteCond %{HTTP:Connection} upgrade [NC]  
        RewriteRule /(.*) ws://localhost:3000/_next/  
        ProxyPass ws://localhost:3000/_next/webpack-  
        ProxyPassReverse ws://localhost:3000/_next/v  
    </Location>  
</VirtualHost>
```

For custom servers, such as `express`, you may need to use `app.all` to ensure the request is passed correctly, for example:

```
app.all('/_next/webpack-hmr', (req, res) => {
  nextjsRequestHandler(req, res)
})
```

Webpack 4 support has been removed

If you are already using webpack 5 you can skip this section.

Next.js has adopted webpack 5 as the default for compilation in Next.js 11. As communicated in the [webpack 5 upgrading documentation](#) Next.js 12 removes support for webpack 4.

If your application is still using webpack 4 using the opt-out flag, you will now see an error linking to the [webpack 5 upgrading documentation](#).

`target` option deprecated

If you do not have `target` in `next.config.js` you can skip this section.

The target option has been deprecated in favor of built-in support for tracing what dependencies are needed to run a page.

During `next build`, Next.js will automatically trace each page and its dependencies to determine all of the files that are needed for deploying a production version of your application.

If you are currently using the `target` option set to `serverless`, please read the [documentation on how to leverage the new output](#).

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to upgrade to version 13

Upgrading from 12 to 13

To update to Next.js version 13, run the following command using your preferred package manager:

>_ Terminal



```
npm i next@13 react@latest react-dom@latest e
```

>_ Terminal



```
yarn add next@13 react@latest react-dom@lates
```

>_ Terminal



```
pnpm i next@13 react@latest react-dom@latest
```

>_ Terminal



```
bun add next@13 react@latest react-dom@latest
```

Good to know: If you are using TypeScript, ensure you also upgrade `@types/react` and `@types/react-dom` to their latest versions.

v13 Summary

- The [Supported Browsers](#) have been changed to drop Internet Explorer and target modern browsers.
- The minimum Node.js version has been bumped from 12.22.0 to 16.14.0, since 12.x and 14.x have reached end-of-life.
- The minimum React version has been bumped from 17.0.2 to 18.2.0.
- The `swcMinify` configuration property was changed from `false` to `true`. See [Next.js Compiler](#) for more info.
- The `next/image` import was renamed to `next/legacy/image`. The `next/future/image` import was renamed to `next/image`. A [codemod is available](#) to safely and automatically rename your imports.
- The `next/link` child can no longer be `<a>`. Add the `legacyBehavior` prop to use the legacy behavior or remove the `<a>` to upgrade. A [codemod is available](#) to automatically upgrade your code.
- The `target` configuration property has been removed and superseded by [Output File Tracing](#).

Migrating shared features

Next.js 13 introduces a new `app` directory with new features and conventions. However, upgrading to Next.js 13 does **not** require using the new `app` 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 many improvements to the Image Component with a temporary import:

`next/future/image`. These improvements included less client-side JavaScript, easier ways to extend and style images, better accessibility, and native browser lazy loading.

Starting in Next.js 13, this new behavior is now the default for `next/image`.

There are two codemods to help you migrate to the new Image Component:

- [next-image-to-legacy-image](#): This codemod will safely and automatically rename `next/image` imports to `next/legacy/image` to maintain the same behavior as Next.js 12. We recommend running this codemod to quickly update to Next.js 13 automatically.
- [next-image-experimental](#): After running the previous codemod, you can optionally run this experimental codemod to upgrade `next/legacy/image` to the new `next/image`, which will remove unused props and add inline styles. Please note this codemod is experimental and only covers static usage (such as `<Image src={img} layout="responsive" />`) but not dynamic usage (such as `<Image {...props} />`).

Alternatively, you can manually update by following the [migration guide](#) and also see the [legacy comparison](#).

<Link> Component

The `<Link>` Component no longer requires manually adding an `<a>` tag as a child. This behavior was added as an experimental option in [version 12.2 ↗](#) and is now the default. In Next.js 13, `<Link>` always renders `<a>` and allows you to forward props to the underlying tag.

For example:

```
import Link from 'next/link'

// Next.js 12: `<a>` has to be nested otherwise
<Link href="/about">
  <a>About</a>
</Link>

// Next.js 13: `<Link>` always renders `<a>`
<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`. If incrementally adopting `app`, read the [upgrade guide](#).

Font Optimization

Previously, Next.js helped you optimize fonts by inlining font CSS. Version 13 introduces the new `next/font` module which gives you the ability to customize your font loading experience while still ensuring great performance and privacy.

See [Optimizing Fonts](#) to learn how to use `next/font`.

Was this helpful?





Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

How to upgrade to version 14

Upgrading from 13 to 14

To update to Next.js version 14, run the following command using your preferred package manager:

>_ Terminal



```
npm i next@next-14 react@18 react-dom@18 && n
```

>_ Terminal



```
yarn add next@next-14 react@18 react-dom@18 &
```

>_ Terminal



```
pnpm i next@next-14 react@18 react-dom@18 &&
```

>_ Terminal



```
bun add next@next-14 react@18 react-dom@18 &&
```

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.

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to upgrade to version 9

To upgrade to version 9, run the following command:

>_ Terminal

```
npm i next@9
```

>_ Terminal

```
yarn add next@9
```

>_ Terminal

```
pnpm up next@9
```

>_ Terminal

```
bun add next@9
```

Good to know: If you are using TypeScript, ensure you also upgrade `@types/react` and `@types/react-dom` to their corresponding versions.

Check your Custom App File (pages/_app.js)

If you previously copied the [Custom <App>](#) example, you may be able to remove your `getInitialProps`.

Removing `getInitialProps` from `pages/_app.js` (when possible) is important to leverage new Next.js features!

The following `getInitialProps` does nothing and may be removed:

```
class MyApp extends App {  
  // Remove me, I do nothing!  
  static async getInitialProps({ Component, c  
    let pageProps = {}  
  
    if (Component.getInitialProps) {  
      pageProps = await Component.getInitialP  
    }  
  
    return { pageProps }  
  }  
  
  render() {  
    // ... etc  
  }  
}
```

Breaking Changes

[@zeit/next-typescript](#) is no longer necessary

Next.js will now ignore usage [@zeit/next-typescript](#) and warn you to remove

it. Please remove this plugin from your `next.config.js`.

Remove references to `@zeit/next-typescript/babel` from your custom `.babelrc` (if present).

The usage of `fork-ts-checker-webpack-plugin` ↗ should also be removed from your `next.config.js`.

TypeScript Definitions are published with the `next` package, so you need to uninstall `@types/next` as they would conflict.

The following types are different:

This list was created by the community to help you upgrade, if you find other differences please send a pull-request to this list to help other users.

From:

```
import { NextContext } from 'next'  
import { NextAppContext, DefaultAppIProps } f  
import { NextDocumentContext, DefaultDocument
```

to

```
import { NextPageContext } from 'next'  
import { ApplicationContext, AppInitialProps } from '  
import { DocumentContext, DocumentInitialProp
```

The `config` key is now an export on a page

You may no longer export a custom variable named `config` from a page (i.e. `export { config }` / `export const config ...`). This exported variable is now used to specify page-level Next.js

configuration like Opt-in AMP and API Route features.

You must rename a non-Next.js-purposed `config` export to something different.

`next/dynamic` no longer renders "loading..." by default while loading

Dynamic components will not render anything by default while loading. You can still customize this behavior by setting the `loading` property:

```
import dynamic from 'next/dynamic'

const DynamicComponentWithCustomLoading = dynamic(() => import('../components/hello2'), {
  loading: () => <p>Loading</p>,
})
```

`withAmp` has been removed in favor of an exported configuration object

Next.js now has the concept of page-level configuration, so the `withAmp` higher-order component has been removed for consistency.

This change can be **automatically migrated by running the following commands in the root of your Next.js project:**

> Terminal



```
curl -L https://github.com/vercel/next-codemod
```

To perform this migration by hand, or view what the codemod will produce, see below:

Before

```
import { withAmp } from 'next/amp'

function Home() {
  return <h1>My AMP Page</h1>
}

export default withAmp(Home)
// or
export default withAmp(Home, { hybrid: true })
```

After

```
export default function Home() {
  return <h1>My AMP Page</h1>
}

export const config = {
  amp: true,
  // or
  amp: 'hybrid',
}
```

next export no longer exports pages as index.html

Previously, exporting `pages/about.js` would result in `out/about/index.html`. This behavior has been changed to result in `out/about.html`.

You can revert to the previous behavior by creating a `next.config.js` with the following content:

```
JS next.config.js
```

```
module.exports = {
  trailingSlash: true,
}
```

pages/api/ is treated differently

Pages in `pages/api/` are now considered [API Routes](#). Pages in this directory will no longer

Deprecated Features

`next/dynamic` has deprecated loading multiple modules at once

The ability to load multiple modules at once has been deprecated in `next/dynamic` to be closer to React's implementation (`React.lazy` and `Suspense`).

Updating code that relies on this behavior is relatively straightforward! We've provided an example of a before/after to help you migrate your application:

Before

```
import dynamic from 'next/dynamic'

const HelloBundle = dynamic({
  modules: () => {
    const components = {
      Hello1: () => import('../components/hel
      Hello2: () => import('../components/hel
    }
  }

  return components
},
  render: (props, { Hello1, Hello2 }) => (
    <div>
      <h1>{props.title}</h1>
      <Hello1 />
      <Hello2 />
    </div>
  ),
})
}

function DynamicBundle() {
  return <HelloBundle title="Dynamic Bundle">
}

export default DynamicBundle
```

After

```
import dynamic from 'next/dynamic'

const Hello1 = dynamic(() => import('../components/Hello1'))
const Hello2 = dynamic(() => import('../components/Hello2'))

function HelloBundle({ title }) {
  return (
    <div>
      <h1>{title}</h1>
      <Hello1 />
      <Hello2 />
    </div>
  )
}

function DynamicBundle() {
  return <HelloBundle title="Dynamic Bundle" />
}

export default DynamicBundle
```

Was this helpful?    

[Copy page](#)

Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

Latest Version

15.5.4



Building Your Application

Routing

Learn the fundamentals of routing for front-...

Rendering

Learn the fundamentals of rendering in Reac...

Data Fetching

Next.js allows you to fetch data in multiple ways,...

Configuring

Learn how to configure your Next.js application.

Was this helpful?





Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



Routing

The Pages Router has a file-system based router built on concepts of pages. When a file is added to the `pages` directory it's automatically available as a route. Learn more about routing in the Pages Router:

Pages and L...

Create your first page and shared layout with the...

Dynamic Rou...

Dynamic Routes are pages that allow you to add...

Linking and ...

Learn how navigation works in Next.js, and ho...

Custom App

Control page initialization and add a layout that...

Custom Doc...

Extend the default document markup added by Next.js.

API Routes

Next.js supports API Routes, which allow you to build...

Custom Errors

Override and
extend the built-in
Error page to...

Was this helpful?



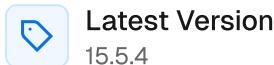


Using Pages Router

Features available in `/pages`



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



Pages and Layouts

The Pages Router has a file-system based router built on the concept of pages.

When a file is added to the `pages` directory, it's automatically available as a route.

In Next.js, a **page** is a [React Component ↗](#) exported from a `.js`, `.jsx`, `.ts`, or `.tsx` file in the `pages` directory. Each page is associated with a route based on its file name.

Example: If you create `pages/about.js` that exports a React component like below, it will be accessible at `/about`.

```
export default function About() {  
  return <div>About</div>  
}
```

Index routes

The router will automatically route files named `index` to the root of the directory.

- `pages/index.js` → `/`
- `pages/blog/index.js` → `/blog`

Nested routes

The router supports nested files. If you create a nested folder structure, files will automatically be routed in the same way still.

- `pages/blog/first-post.js` → `/blog/first-post`
- `pages/dashboard/settings/username.js` → `/dashboard/settings/username`

Pages with Dynamic Routes

Next.js supports pages with dynamic routes. For example, if you create a file called `pages/posts/[id].js`, then it will be accessible at `posts/1`, `posts/2`, etc.

To learn more about dynamic routing, check the [Dynamic Routing documentation](#).

Layout Pattern

The React model allows us to deconstruct a [page](#) into a series of components. Many of these components are often reused between pages. For example, you might have the same navigation bar and footer on every page.

`JS` `components/layout.js`



```
import Navbar from './navbar'  
import Footer from './footer'
```

```
export default function Layout({ children })  
  return (  
    <>  
      <Navbar />  
      <main>{children}</main>  
      <Footer />  
    </>  
  )  
}
```

Examples

Single Shared Layout with Custom App

If you only have one layout for your entire application, you can create a [Custom App](#) and wrap your application with the layout. Since the `<Layout />` component is re-used when changing pages, its component state will be preserved (e.g. input values).

```
js pages/_app.js
```

```
import Layout from '../components/layout'  
  
export default function MyApp({ Component, pa  
  return (  
    <Layout>  
      <Component {...pageProps} />  
    </Layout>  
  )  
}
```

Per-Page Layouts

If you need multiple layouts, you can add a property `getLayout` to your page, allowing you to return a React component for the layout. This allows you to define the layout on a *per-page*

basis. Since we're returning a function, we can have complex nested layouts if desired.

JS pages/index.js

```
import Layout from '../components/layout'
import NestedLayout from '../components/neste

export default function Page() {
  return (
    /** Your content */
  )
}

Page.getLayout = function getLayout(page) {
  return (
    <Layout>
      <NestedLayout>{page}</NestedLayout>
    </Layout>
  )
}
```

JS pages/_app.js

```
export default function MyApp({ Component, pa
  // Use the layout defined at the page level
  const getLayout = Component.getLayout ?? ((

    return getLayout(<Component {...pageProps}
  )
```

When navigating between pages, we want to *persist* page state (input values, scroll position, etc.) for a Single-Page Application (SPA) experience.

This layout pattern enables state persistence because the React component tree is maintained between page transitions. With the component tree, React can understand which elements have changed to preserve state.

Good to know: This process is called [reconciliation ↗](#), which is how React understands which elements have

changed.

With TypeScript

When using TypeScript, you must first create a new type for your pages which includes a `getLayout` function. Then, you must create a new type for your `AppProps` which overrides the `Component` property to use the previously created type.

```
TS pages/index.tsx TypeScript ▾ ⌂

import type { ReactElement } from 'react'
import Layout from '../components/layout'
import NestedLayout from '../components/neste
import type { NextPageWithLayout } from './_a

const Page: NextPageWithLayout = () => {
  return <p>hello world</p>
}

Page.getLayout = function getLayout(page: Re
  return (
    <Layout>
      <NestedLayout>{page}</NestedLayout>
    </Layout>
  )
}

export default Page
```

```
TS pages/_app.tsx TypeScript ▾ ⌂

import type { ReactElement, ReactNode } from
import type { NextPage } from 'next'
import type { AppProps } from 'next/app'

export type NextPageWithLayout<P = {}, IP = P
  getLayout?: (page: ReactElement) => ReactNo
}

type AppPropsWithLayout = AppProps & {
  Component: NextPageWithLayout
}

export default function MyApp({ Component, pa
```

```
// Use the layout defined at the page level
const getLayout = Component.getLayout ?? () =>

  return getLayout(<Component {...pageProps}>
}
```

Data Fetching

Inside your layout, you can fetch data on the client-side using `useEffect` or a library like [SWR](#). Because this file is not a [Page](#), you cannot use `getStaticProps` or `getServerSideProps` currently.

components/layout.js



```
import useSWR from 'swr'
import Navbar from './navbar'
import Footer from './footer'

export default function Layout({ children }) {
  const { data, error } = useSWR('/api/navbar')

  if (error) return <div>Failed to load</div>
  if (!data) return <div>Loading...</div>

  return (
    <>
      <Navbar links={data.links} />
      <main>{children}</main>
      <Footer />
    </>
  )
}
```

Was this helpful?

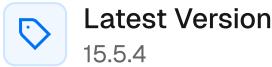


Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



Linking and Navigating

The Next.js router allows you to do client-side route transitions between pages, similar to a single-page application.

A React component called `Link` is provided to do this client-side route transition.

```
import Link from 'next/link'

function Home() {
  return (
    <ul>
      <li>
        <Link href="/">Home</Link>
      </li>
      <li>
        <Link href="/about">About Us</Link>
      </li>
      <li>
        <Link href="/blog/hello-world">Blog P
      </li>
    </ul>
  )
}

export default Home
```

The example above uses multiple links. Each one maps a path (`href`) to a known page:

- / → pages/index.js

- /about → pages/about.js
- /blog/hello-world → pages/blog/[slug].js

Any `<Link />` in the viewport (initially or through scroll) will be prefetched by default (including the corresponding data) for pages using **Static Generation**. The corresponding data for **server-rendered** routes is fetched *only when* the `<Link />` is clicked.

Linking to dynamic paths

You can also use interpolation to create the path, which comes in handy for **dynamic route segments**. For example, to show a list of posts which have been passed to the component as a prop:

```
import Link from 'next/link'

function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={`/blog/${encodeURIComponent(post.title)}}>
            </Link>
          </li>
      ))}
    </ul>
  )
}

export default Posts
```

`encodeURIComponent ↗` is used in the example to keep the path utf-8 compatible.

Alternatively, using a URL Object:

```
import Link from 'next/link'

function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={{ pathname: '/blog/[slug]' ,
            query: { slug: post.slug } ,
          }}>
            {post.title}
          </Link>
        </li>
      ))}
    </ul>
  )
}

export default Posts
```

Now, instead of using interpolation to create the path, we use a URL object in `href` where:

- `pathname` is the name of the page in the `pages` directory. `/blog/[slug]` in this case.
- `query` is an object with the dynamic segment. `slug` in this case.

Injecting the router

To access the `router` object in a React component you can use `useRouter` or `withRouter`.

In general we recommend using `useRouter`.

Imperative Routing

`next/link` should be able to cover most of your routing needs, but you can also do client-side navigations without it, take a look at the documentation for `next/router`.

The following example shows how to do basic page navigations with `useRouter`:

```
import { useRouter } from 'next/router'

export default function ReadMore() {
  const router = useRouter()

  return (
    <button onClick={() => router.push('/about')}
      Click here to read more
    </button>
  )
}
```

Shallow Routing

► Examples

Shallow routing allows you to change the URL without running data fetching methods again, that includes `getServerSideProps`, `getStaticProps`, and `getInitialProps`.

You'll receive the updated `pathname` and the `query` via the `router object` (added by `useRouter` or `withRouter`), without losing state.

To enable shallow routing, set the `shallow` option to `true`. Consider the following example:

```
import { useEffect } from 'react'
import { useRouter } from 'next/router'

// Current URL is '/'
function Page() {
  const router = useRouter()

  useEffect(() => {
    // Always do navigations after the first
    router.push('/?counter=10', undefined, {
      }, [])
  })

  useEffect(() => {
    // The counter changed!
    }, [router.query.counter])
}

export default Page
```

The URL will get updated to `/?counter=10` and the page won't get replaced, only the state of the route is changed.

You can also watch for URL changes via

`componentDidUpdate` ↗ as shown below:

```
componentDidUpdate(prevProps) {
  const { pathname, query } = this.props.router
  // verify props have changed to avoid an infinite loop
  if (query.counter !== prevProps.router.query.counter)
    // fetch data based on the new query
}
}
```

Caveats

Shallow routing **only** works for URL changes in the current page. For example, let's assume we have another page called `pages/about.js`, and you run this:

```
router.push('/?counter=10', '/about?counter=10')
```

Since that's a new page, it'll unload the current page, load the new one and wait for data fetching even though we asked to do shallow routing.

When shallow routing is used with middleware it will not ensure the new page matches the current page like previously done without middleware. This is due to middleware being able to rewrite dynamically and can't be verified client-side without a data fetch which is skipped with shallow, so a shallow route change must always be treated as shallow.

Was this helpful?    

 **Using Pages Router**
Features available in /pages

 **Latest Version**
15.5.4

(i) You are currently viewing the documentation for Pages Router.

Custom App

Next.js uses the `App` component to initialize pages. You can override it and control the page initialization and:

- Create a shared layout between page changes
- Inject additional data into pages
- [Add global CSS](#)

Usage

To override the default `App`, create the file `pages/_app` as shown below:

```
TS pages/_app.tsx TypeScript ⓘ  
  
import type { AppProps } from 'next/app'  
  
export default function MyApp({ Component, pa  
    return <Component {...pageProps} />  
}
```

The `Component` prop is the active `page`, so whenever you navigate between routes, `Component` will change to the new `page`. Therefore, any props you send to `Component` will be received by the `page`.

`pageProps` is an object with the initial props that were preloaded for your page by one of our [data fetching methods](#), otherwise it's an empty object.

Good to know:

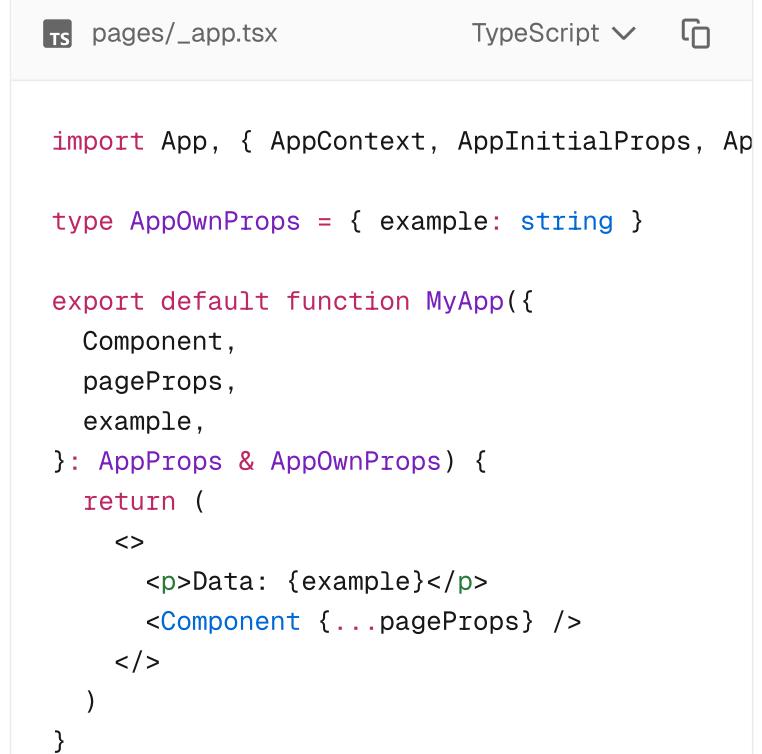
- If your app is running and you added a custom `App`, you'll need to restart the development server. Only required if `pages/_app.js` didn't exist before.
- `App` does not support Next.js [Data Fetching methods](#) like `getStaticProps` or `getServerSideProps`.

getInitialProps with App

Using `getInitialProps` in `App` will disable [Automatic Static Optimization](#) for pages without `getStaticProps`.

We do not recommend using this pattern.

Instead, consider [incrementally adopting](#) the App Router, which allows you to more easily fetch data for pages and layouts.



The screenshot shows a code editor interface with the file `pages/_app.tsx` open. The code uses TypeScript and includes imports for `App`, `Component`, `AppProps`, and `AppInitialProps`. It defines a type `AppOwnProps` and an export default function `MyApp` that takes `Component`, `pageProps`, and `example` as arguments, along with `AppProps & AppOwnProps`. The function returns a component that renders an `<p>` tag with the value of `example` and then renders the provided `Component` with `pageProps` as its props.

```
import App, { AppContext, AppInitialProps, AppProps } from 'next/app'

type AppOwnProps = { example: string }

export default function MyApp({ Component, pageProps, example }: AppProps & AppOwnProps) {
  return (
    <>
      <p>Data: {example}</p>
      <Component {...pageProps} />
    </>
  )
}
```

```
MyApp.getInitialProps = async (context: ApplicationContext): Promise<AppOwnProps & AppInitialProps> =>
  const ctx = await App.getInitialProps(context)

  return { ...ctx, example: 'data' }
```

Was this helpful?



Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

Latest Version

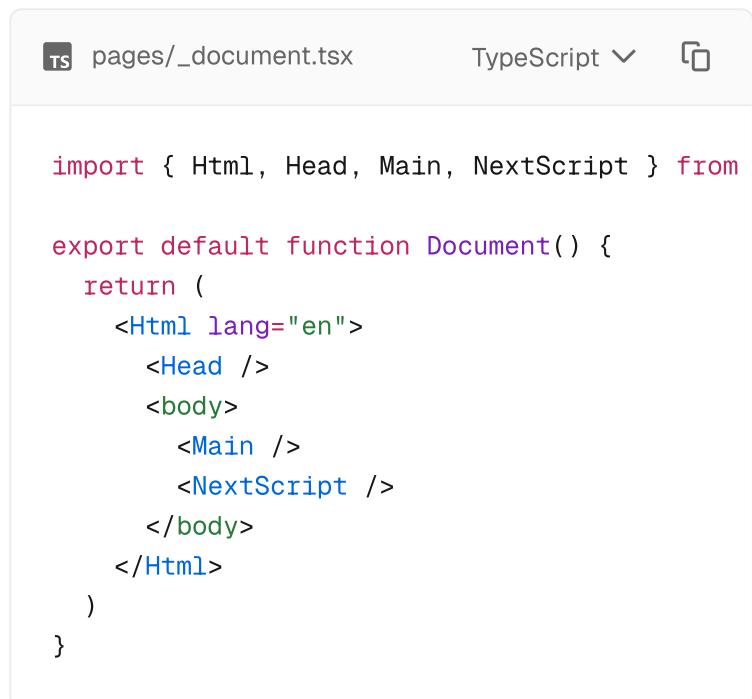
15.5.4



Custom Document

A custom `Document` can update the `<html>` and `<body>` tags used to render a [Page](#).

To override the default `Document`, create the file `pages/_document` as shown below:



TS pages/_document.tsx TypeScript

```
import { Html, Head, Main, NextScript } from 'next/document'

export default function Document() {
  return (
    <Html lang="en">
      <Head />
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

Good to know:

- `_document` is only rendered on the server, so event handlers like `onClick` cannot be used in this file.
- `<Html>`, `<Head />`, `<Main />` and `<NextScript />` are required for the page to be properly rendered.

Caveats

- The `<Head />` component used in `_document` is not the same as `next/head`. The `<Head />` component used here should only be used for any `<head>` code that is common for all pages. For all other cases, such as `<title>` tags, we recommend using `next/head` in your pages or components.
- React components outside of `<Main />` will not be initialized by the browser. Do *not* add application logic here or custom CSS (like `styled-jsx`). If you need shared components in all your pages (like a menu or a toolbar), read [Layouts](#) instead.
- `Document` currently does not support Next.js [Data Fetching](#) methods like `getStaticProps` or `getServerSideProps`.

Customizing `renderPage`

Customizing `renderPage` is advanced and only needed for libraries like CSS-in-JS to support server-side rendering. This is not needed for built-in `styled-jsx` support.

We do not recommend using this pattern.

Instead, consider [incrementally adopting](#) the App Router, which allows you to more easily fetch data for pages and layouts.



The screenshot shows a code editor interface with the following details:

- File name: `pages/_document.tsx`
- TypeScript support is enabled.
- The code block contains the following TypeScript code:

```
import Document, {  
  Html,  
  Head,
```

```

Main,
NextScript,
DocumentContext,
DocumentInitialProps,
} from 'next/document'

class MyDocument extends Document {
  static async getInitialProps(
    ctx: DocumentContext
  ): Promise<DocumentInitialProps> {
    const originalRenderPage = ctx.renderPage

    // Run the React rendering logic synchronously
    ctx.renderPage = () =>
      originalRenderPage({
        // Useful for wrapping the whole react app
        enhanceApp: (App) => App,
        // Useful for wrapping in a per-page component
        enhanceComponent: (Component) => Component
      })

    // Run the parent `getInitialProps`, it now has the correct `ctx`
    const initialProps = await Document.getInitialProps(ctx)

    return initialProps
  }

  render() {
    return (
      <Html lang="en">
        <Head />
        <body>
          <Main />
          <NextScript />
        </body>
      </Html>
    )
  }
}

export default MyDocument

```

Good to know:

- `getInitialProps` in `_document` is not called during client-side transitions.
- The `ctx` object for `_document` is equivalent to the one received in `getInitialProps`, with the addition of `renderPage`.



Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



API Routes

► Examples

Good to know: If you are using the App Router, you can use [Server Components](#) or [Route Handlers](#) instead of API Routes.

API routes provide a solution to build a **public API** with Next.js.

Any file inside the folder `pages/api` is mapped to `/api/*` and will be treated as an API endpoint instead of a `page`. They are server-side only bundles and won't increase your client-side bundle size.

For example, the following API route returns a JSON response with a status code of `200`:

TS pages/api/hello.ts TypeScript ▾

```
import { NextApiRequest, NextApiResponse } from 'next'

export default function handler(req: NextApiRequest, res: NextApiResponse) {
  res.status(200).json({ message: 'Hello from API!' })
}
```

```
import type { NextApiRequest, NextApiResponse } from 'next'

type ResponseData = {
  message: string
}

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<ResponseData>
) {
  res.status(200).json({ message: 'Hello from API' })
}
```

Good to know:

- API Routes [do not specify CORS headers](#), meaning they are **same-origin only** by default. You can customize such behavior by wrapping the request handler with the [CORS request helpers](#).
- API Routes can't be used with [static exports](#). However, [Route Handlers](#) in the App Router can.

- API Routes will be affected by [pageExtensions configuration](#) in `next.config.js`.

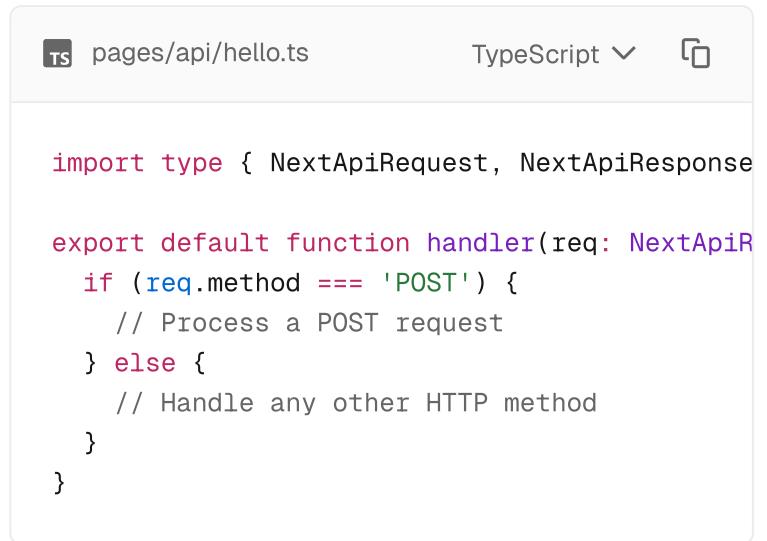
Parameters

```
export default function handler(req: NextApiRequest) {
  // ...
}
```

- `req`: An instance of [http.IncomingMessage](#)
- `res`: An instance of [http.ServerResponse](#)

HTTP Methods

To handle different HTTP methods in an API route, you can use `req.method` in your request handler, like so:



```
TS pages/api/hello.ts TypeScript ▾ ⌂

import type { NextApiRequest, NextApiResponse }

export default function handler(req: NextApiR
  if (req.method === 'POST') {
    // Process a POST request
  } else {
    // Handle any other HTTP method
  }
}
```

Request Helpers

API Routes provide built-in request helpers which parse the incoming request (`req`):

- `req.cookies` - An object containing the cookies sent by the request. Defaults to `{}`
- `req.query` - An object containing the `query string`. Defaults to `{}`
- `req.body` - An object containing the body parsed by `content-type`, or `null` if no body was sent

Custom config

Every API Route can export a `config` object to change the default configuration, which is the following:

```
export const config = {
  api: {
    bodyParser: {
      sizeLimit: '1mb',
    },
  },
  // Specifies the maximum allowed duration for requests
  maxDuration: 5,
}
```

`bodyParser` is automatically enabled. If you want to consume the body as a `Stream` or with `raw-body` [↗](#), you can set this to `false`.

One use case for disabling the automatic `bodyParsing` is to allow you to verify the raw body of a `webhook` request, for example [from GitHub](#) [↗](#).

```
export const config = {
  api: {
    bodyParser: false,
  },
}
```

`bodyParser.sizeLimit` is the maximum size allowed for the parsed body, in any format supported by `bytes` [↗](#), like so:

```
export const config = {
  api: {
    bodyParser: {
      sizeLimit: '500kb',
    },
  },
}
```

`externalResolver` is an explicit flag that tells the server that this route is being handled by an external resolver like `express` or `connect`. Enabling this option disables warnings for unresolved requests.

```
export const config = {
  api: {
    externalResolver: true,
  },
}
```

`responseLimit` is automatically enabled, warning when an API Routes' response body is over 4MB.

If you are not using Next.js in a serverless environment, and understand the performance implications of not using a CDN or dedicated media host, you can set this limit to `false`.

```
export const config = {
  api: {
    responseLimit: false,
  },
}
```

`responseLimit` can also take the number of bytes or any string format supported by `bytes`, for example `1000`, `'500kb'` or `'3mb'`. This value will be the maximum response size before a warning is displayed. Default is 4MB. (see above)

```
export const config = {
  api: {
    responseLimit: '8mb',
  },
}
```

Response Helpers

The [Server Response object ↗](#), (often abbreviated as `res`) includes a set of Express.js-like helper methods to improve the developer experience and increase the speed of creating new API endpoints.

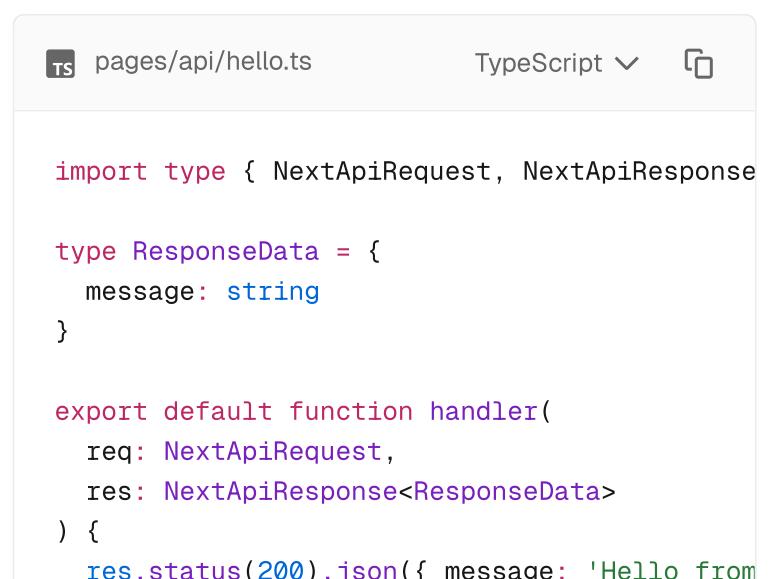
The included helpers are:

- `res.status(code)` - A function to set the status code. `code` must be a valid [HTTP status code](#) ↗
- `res.json(body)` - Sends a JSON response. `body` must be a [serializable object](#) ↗
- `res.send(body)` - Sends the HTTP response. `body` can be a `string`, an `object` or a `Buffer`
- `res.redirect([status,] path)` - Redirects to a specified path or URL. `status` must be a valid [HTTP status code](#) ↗. If not specified, `status` defaults to "307" "Temporary redirect".
- `res.revalidate(urlPath)` - [Revalidate a page on demand](#) using `getStaticProps`. `urlPath` must be a `string`.

Setting the status code of a response

When sending a response back to the client, you can set the status code of the response.

The following example sets the status code of the response to `200` (`OK`) and returns a `message` property with the value of `Hello from Next.js!` as a JSON response:



A screenshot of a code editor showing a file named `pages/api/hello.ts`. The code is written in TypeScript and defines a handler function for an API request. The handler takes `req` and `res` parameters. It uses the `res.status(200).json()` method to send a JSON response with a `message` property containing the string `'Hello from Next.js!'`.

```
import type { NextApiRequest, NextApiResponse } from 'next'

type ResponseData = {
  message: string
}

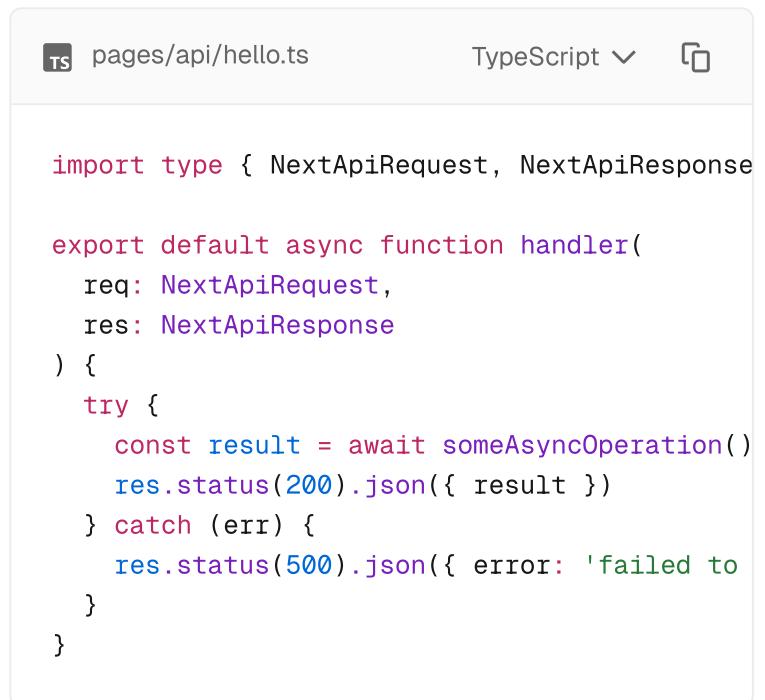
export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<ResponseData>
) {
  res.status(200).json({ message: 'Hello from Next.js!' })
}
```

```
}
```

Sending a JSON response

When sending a response back to the client you can send a JSON response, this must be a [serializable object ↗](#). In a real world application you might want to let the client know the status of the request depending on the result of the requested endpoint.

The following example sends a JSON response with the status code `200` (`OK`) and the result of the async operation. It's contained in a try catch block to handle any errors that may occur, with the appropriate status code and error message caught and sent back to the client:



A screenshot of a code editor window titled "pages/api/hello.ts". The file contains the following TypeScript code:

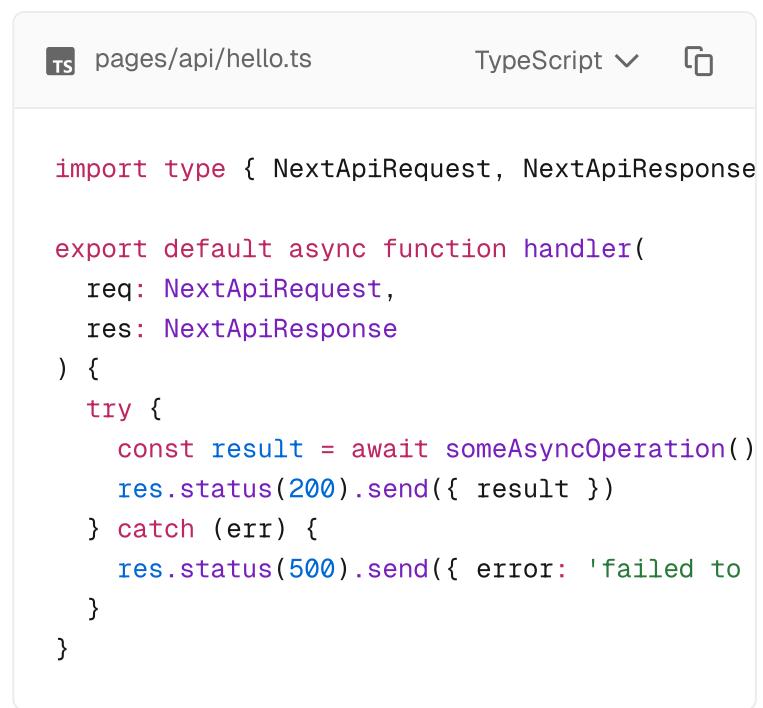
```
import type { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  try {
    const result = await someAsyncOperation()
    res.status(200).json({ result })
  } catch (err) {
    res.status(500).json({ error: 'Failed to' })
  }
}
```

Sending a HTTP response

Sending an HTTP response works the same way as when sending a JSON response. The only difference is that the response body can be a `string`, an `object` or a `Buffer`.

The following example sends a HTTP response with the status code `200` (OK) and the result of the async operation.



A screenshot of a code editor window titled "pages/api/hello.ts". The file contains the following TypeScript code:

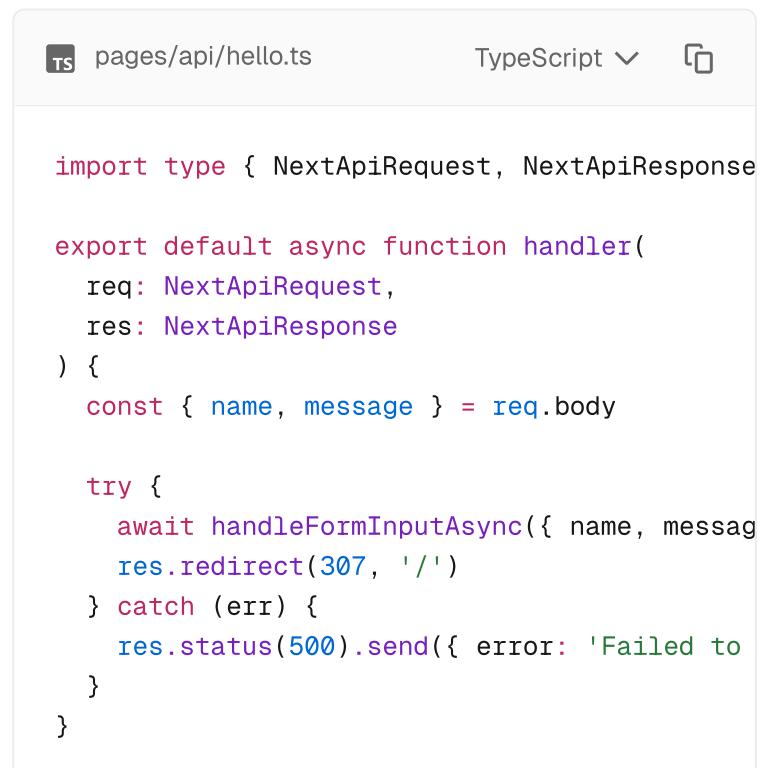
```
import type { NextApiRequest, NextApiResponse }

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  try {
    const result = await someAsyncOperation()
    res.status(200).send({ result })
  } catch (err) {
    res.status(500).send({ error: 'Failed to' })
  }
}
```

Redirects to a specified path or URL

Taking a form as an example, you may want to redirect your client to a specified path or URL once they have submitted the form.

The following example redirects the client to the `/` path if the form is successfully submitted:



A screenshot of a code editor window titled "pages/api/hello.ts". The file contains the following TypeScript code:

```
import type { NextApiRequest, NextApiResponse }

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const { name, message } = req.body

  try {
    await handleFormInputAsync({ name, message })
    res.redirect(307, '/')
  } catch (err) {
    res.status(500).send({ error: 'Failed to' })
  }
}
```

Adding TypeScript types

You can make your API Routes more type-safe by importing the `NextApiRequest` and `NextApiResponse` types from `next`, in addition to those, you can also type your response data:

```
import type { NextApiRequest, NextApiResponse }

type ResponseData = {
  message: string
}

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<ResponseData>
) {
  res.status(200).json({ message: 'Hello from' })
}
```

Good to know: The body of `NextApiRequest` is `any` because the client may include any payload. You should validate the type/shape of the body at runtime before using it.

Dynamic API Routes

API Routes support [dynamic routes](#), and follow the same file naming rules used for `pages/`.



The screenshot shows a code editor with a TypeScript file named `pages/api/post/[pid].ts`. The file contains the following code:

```
import type { NextApiRequest, NextApiResponse }

export default function handler(req: NextApiRequest) {
  const { pid } = req.query
  res.end(`Post: ${pid}`)
}
```

Now, a request to `/api/post/abc` will respond with the text: `Post: abc`.

Catch all API routes

API Routes can be extended to catch all paths by adding three dots (`...`) inside the brackets. For example:

- `pages/api/post/[... slug].js` matches `/api/post/a`, but also `/api/post/a/b`, `/api/post/a/b/c` and so on.

Good to know: You can use names other than `slug`, such as: `[... param]`

Matched parameters will be sent as a query parameter (`slug` in the example) to the page, and it will always be an array, so, the path `/api/post/a` will have the following `query` object:

```
{ "slug": [ "a" ] }
```

And in the case of `/api/post/a/b`, and any other matching path, new parameters will be added to the array, like so:

```
{ "slug": [ "a", "b" ] }
```

For example:

```
TS pages/api/post/[...slug].ts TypeScript ▾ ⌂  
  
import type { NextApiRequest, NextApiResponse }  
  
export default function handler(req: NextApiRequest)  
  const { slug } = req.query  
  res.end(`Post: ${slug.join(' ', '')}`)  
}
```

Now, a request to `/api/post/a/b/c` will respond with the text: Post: a, b, c.

Optional catch all API routes

Catch all routes can be made optional by including the parameter in double brackets (`[[... slug]]`).

For example, `pages/api/post/[[... slug]].js` will match `/api/post`, `/api/post/a`, `/api/post/a/b`, and so on.

The main difference between catch all and optional catch all routes is that with optional, the route without the parameter is also matched (`/api/post` in the example above).

The `query` objects are as follows:

```
{ } // GET `/api/post` (empty object)
{ "slug": ["a"] } // `GET /api/post/a` (single)
{ "slug": ["a", "b"] } // `GET /api/post/a/b`
```

Caveats

- Predefined API routes take precedence over dynamic API routes, and dynamic API routes over catch all API routes. Take a look at the following examples:
 - `pages/api/post/create.js` - Will match `/api/post/create`
 - `pages/api/post/[pid].js` - Will match `/api/post/1`, `/api/post/abc`, etc. But not `/api/post/create`
 - `pages/api/post/[... slug].js` - Will match `/api/post/1/2`, `/api/post/a/b/c`, etc. But not `/api/post/create`, `/api/post/abc`

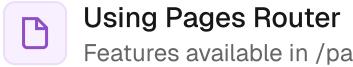
Streaming responses

While the Pages Router does support streaming responses with API Routes, we recommend incrementally adopting the App Router and using [Route Handlers](#) if you are on Next.js 14+.

Here's how you can stream a response from an API Route with `writeHead`:

```
JS pages/api/hello.js Copy  
  
import { NextApiRequest, NextApiResponse } from 'next'  
  
export default async function handler(  
  req: NextApiRequest,  
  res: NextApiResponse  
) {  
  res.writeHead(200, {  
    'Content-Type': 'text/event-stream',  
    'Cache-Control': "no-store",  
  })  
  let i = 0  
  while (i < 10) {  
    res.write(`data: ${i}\n\n`)  
    i++  
    await new Promise((resolve) => setTimeout(resolve, 100))  
  }  
  res.end()  
}
```

Was this helpful? Helpful Unhelpful Smile Sad



(i) You are currently viewing the documentation for Pages Router.



Custom Errors

404 Page

A 404 page may be accessed very often. Server-rendering an error page for every visit increases the load of the Next.js server. This can result in increased costs and slow experiences.

To avoid the above pitfalls, Next.js provides a static 404 page by default without having to add any additional files.

Customizing The 404 Page

To create a custom 404 page you can create a `pages/404.js` file. This file is statically generated at build time.

`JS` `pages/404.js`



```
export default function Custom404() {
  return <h1>404 - Page Not Found</h1>
}
```

Good to know: You can use `getStaticProps` inside this page if you need to fetch data at build time.

500 Page

Server-rendering an error page for every visit adds complexity to responding to errors. To help users get responses to errors as fast as possible, Next.js provides a static 500 page by default without having to add any additional files.

Customizing The 500 Page

To customize the 500 page you can create a `pages/500.js` file. This file is statically generated at build time.

`JS` `pages/500.js`

```
export default function Custom500() {
  return <h1>500 - Server-side error occurred
}
```

Good to know: You can use `getStaticProps` inside this page if you need to fetch data at build time.

More Advanced Error Page Customizing

500 errors are handled both client-side and server-side by the `Error` component. If you wish to override it, define the file `pages/_error.js` and add the following code:

```
function Error({ statusCode }) {
  return (
    <p>
      {statusCode
        ? `An error ${statusCode} occurred on
          : 'An error occurred on client'`}
    </p>
  )
}
```

```
Error.getInitialProps = ({ res, err }) => {
  const statusCode = res ? res.statusCode : e
  return { statusCode }
}

export default Error
```

`pages/_error.js` is only used in production. In development you'll get an error with the call stack to know where the error originated from.

Reusing the built-in error page

If you want to render the built-in error page you can by importing the `Error` component:

```
import Error from 'next/error'

export async function getServerSideProps() {
  const res = await fetch('https://api.github.com/repos/vercel/next.js')
  const errorCode = res.ok ? false : res.status
  const json = await res.json()

  return {
    props: { errorCode, stars: json.stargazers_count }
  }
}

export default function Page({ errorCode, stars }) {
  if (errorCode) {
    return <Error statusCode={errorCode} />
  }

  return <div>Next stars: {stars}</div>
}
```

The `Error` component also takes `title` as a property if you want to pass in a text message along with a `statusCode`.

If you have a custom `Error` component be sure to import that one instead. `next/error` exports the default component used by Next.js.

Caveats

- `Error` does not currently support Next.js Data Fetching methods like `getStaticProps` or `getServerSideProps`.
- `_error`, like `_app`, is a reserved pathname. `_error` is used to define the customized layouts and behaviors of the error pages. `/_error` will render 404 when accessed directly via [routing](#) or rendering in a [custom server](#).

Was this helpful?    

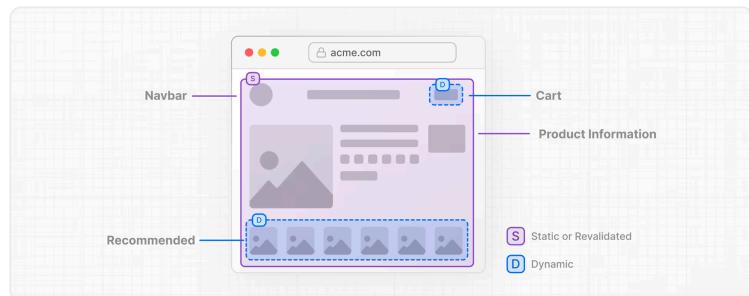
 Using App Router
Features available in /app

 Latest Version
15.5.4

Partial Prerendering

 This feature is currently experimental and subject to change, it's not recommended for production.
[Try it out and share your feedback on GitHub.](#)

Partial Prerendering (PPR) is a rendering strategy that allows you to combine static and dynamic content in the same route. This improves the initial page performance while still supporting personalized, dynamic data.



When a user visits a route:

- The server sends a **shell** containing the static content, ensuring a fast initial load.
- The shell leaves **holes** for the dynamic content that will load in asynchronously.
- The dynamic holes are **streamed in parallel**, reducing the overall load time of the page.

 **Watch:** Why PPR and how it works → [YouTube \(10 minutes\)](#) ↗.

How does Partial Prerendering work?

To understand Partial Prerendering, it helps to be familiar with the rendering strategies available in Next.js.

Static Rendering

With Static Rendering, HTML is generated ahead of time—either at build time or through [revalidation](#). The result is cached and shared across users and requests.

In Partial Prerendering, Next.js prerenders a **static shell** for a route. This can include the layout and any other components that don't depend on request-time data.

Dynamic Rendering

With Dynamic Rendering, HTML is generated at **request time**. This allows you to serve personalized content based on request-time data.

A component becomes dynamic if it uses the following APIs:

- `cookies`
- `headers`
- `connection`
- `draftMode`
- `searchParams` prop
- `unstable_noStore`
- `fetch` with `{ cache: 'no-store' }`

In Partial Prerendering, using these APIs throws a special React error that informs Next.js the component cannot be statically rendered, causing a build error. You can use a [Suspense](#) boundary to wrap your component to defer rendering until runtime.

Suspense

React [Suspense](#) ↗ is used to defer rendering parts of your application until some condition is met.

In Partial Prerendering, Suspense is used to mark **dynamic boundaries** in your component tree.

At build time, Next.js prerenders the static content and the `fallback` UI. The dynamic content is **postponed** until the user requests the route.

Wrapping a component in Suspense doesn't make the component itself dynamic (your API usage does), but rather Suspense is used as a boundary that encapsulates dynamic content and enable [streaming](#)

```
JS app/page.js

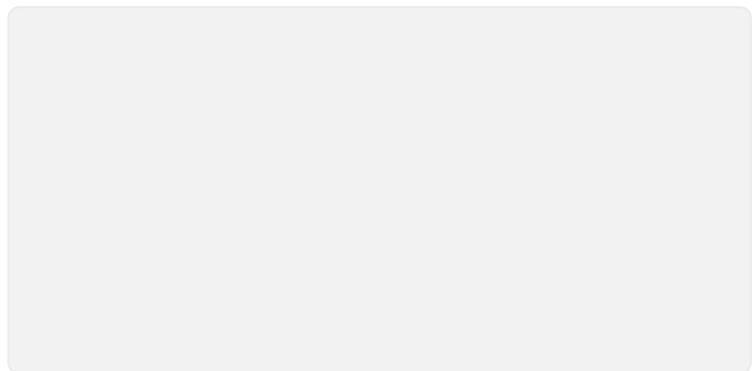
import { Suspense } from 'react'
import StaticComponent from './StaticComponent'
import DynamicComponent from './DynamicComponent'
import Fallback from './Fallback'

export const experimental_ppr = true

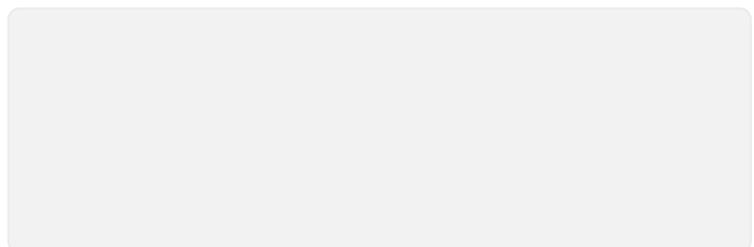
export default function Page() {
  return (
    <>
      <StaticComponent />
      <Suspense fallback={<Fallback />}>
        <DynamicComponent />
      </Suspense>
    </>
  )
}
```

Streaming

Streaming splits the route into chunks and progressively streams them to the client as they become ready. This allows the user to see parts of the page immediately, before the entire content has finished rendering.



In Partial Prerendering, dynamic components wrapped in Suspense start streaming from the server in parallel.



To reduce network overhead, the full response—including static HTML and streamed dynamic parts—is sent in a **single HTTP request**. This avoids extra roundtrips and improves both initial load and overall performance.

Enabling Partial Prerendering

You can enable PPR by adding the `ppr` option to your `next.config.ts` file:

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    ppr: 'incremental',
  },
}

export default nextConfig
```

The `'incremental'` value allows you to adopt PPR for specific routes:

```
TS /app/dashboard/layout.tsx TypeScript ▾ ⌂
export const experimental_ppr = true

export default function Layout({ children }: ...)
```

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.

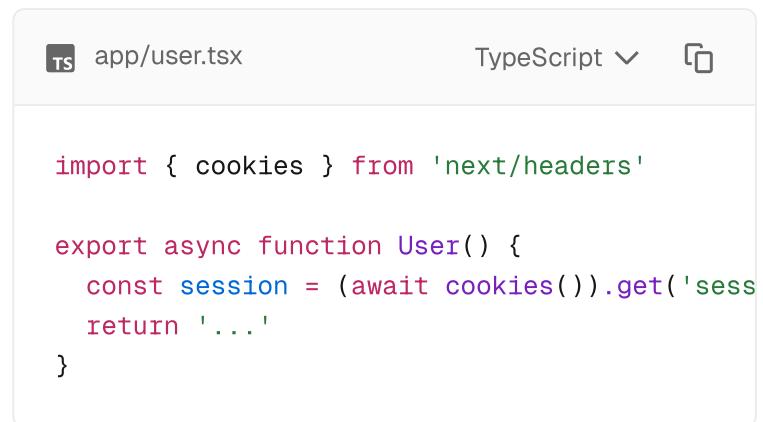
Good to know:

- `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.

Examples

Dynamic APIs

When using Dynamic APIs that require looking at the incoming request, Next.js will opt into dynamic rendering for the route. To continue using PPR, wrap the component with Suspense. For example, the `<User />` component is dynamic because it uses the `cookies` API:

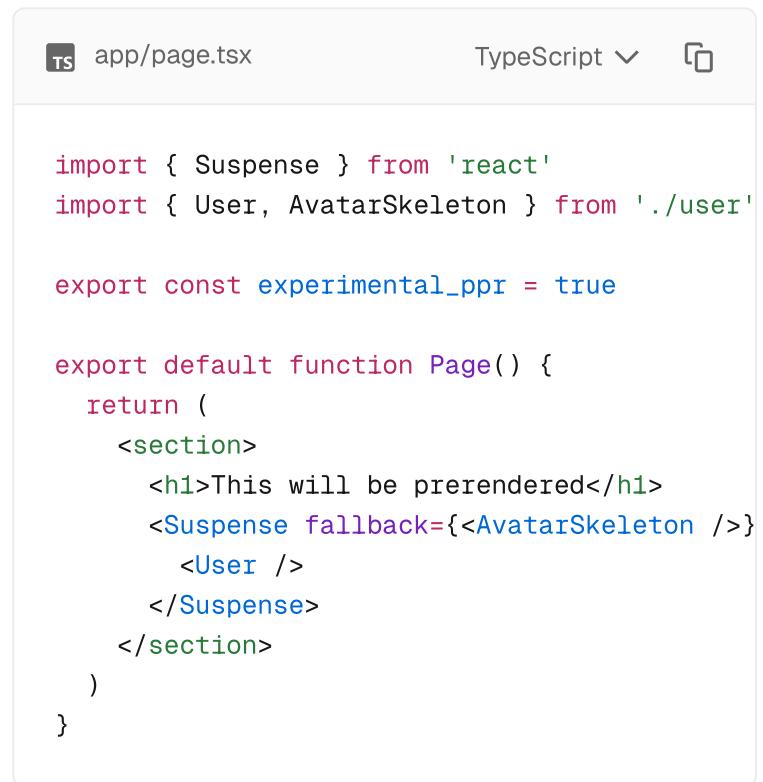


The screenshot shows a code editor window with the file name "app/user.tsx" and a TypeScript dropdown menu. The code is as follows:

```
import { cookies } from 'next/headers'

export async function User() {
  const session = (await cookies()).get('sess
  return '...'
}
```

The `<User />` component will be streamed while any other content inside `<Page />` will be prerendered and become part of the static shell.



The screenshot shows a code editor window with the file name "app/page.tsx" and a TypeScript dropdown menu. The code is as follows:

```
import { Suspense } from 'react'
import { User, AvatarSkeleton } from './user'

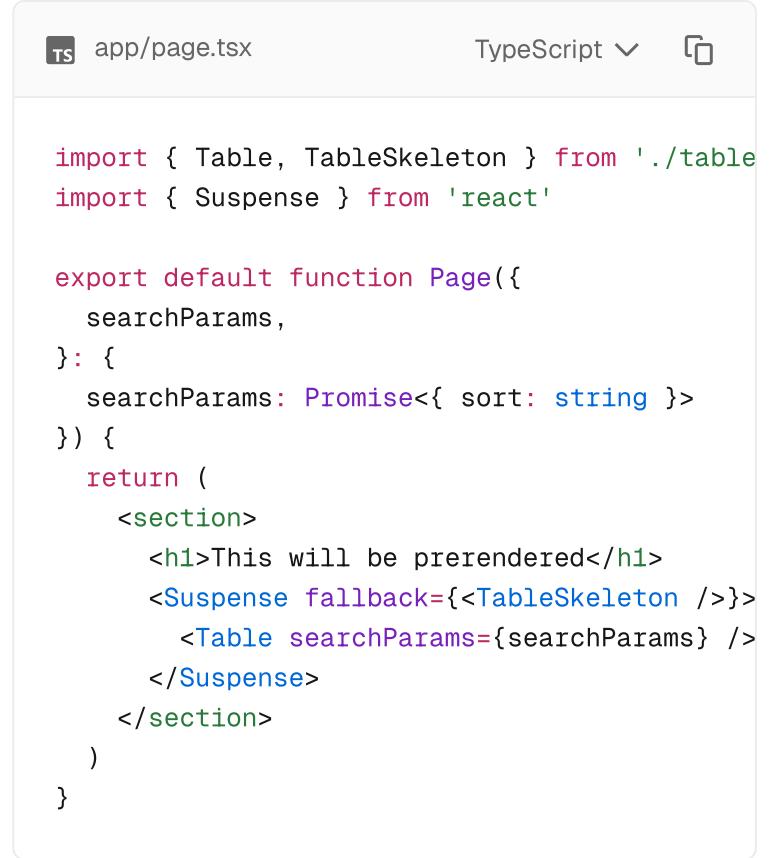
export const experimental_ppr = true

export default function Page() {
  return (
    <section>
      <h1>This will be prerendered</h1>
      <Suspense fallback={<AvatarSkeleton />}>
        <User />
      </Suspense>
    </section>
  )
}
```

Passing dynamic props

Components only opt into dynamic rendering when the value is accessed. For example, if you are reading `searchParams` from a `<Page />`

component, you can forward this value to another component as a prop:



```
import { Table, TableSkeleton } from './table'
import { Suspense } from 'react'

export default function Page({
  searchParams,
}: {
  searchParams: Promise<{ sort: string }>
}) {
  return (
    <section>
      <h1>This will be prerendered</h1>
      <Suspense fallback={<TableSkeleton />}>
        <Table searchParams={searchParams} />
      </Suspense>
    </section>
  )
}
```

Inside of the table component, accessing the value from `searchParams` will make the component dynamic while the rest of the page will be prerendered.



```
export async function Table({
  searchParams,
}: {
  searchParams: Promise<{ sort: string }>
}) {
  const sort = (await searchParams).sort ===
  return '...'
}
```

Next Steps

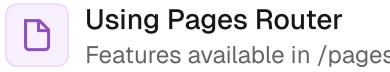
Learn more about the config option for Partial
Prerendering.

ppr

Learn how to
enable Partial
Prerendering in...

Was this helpful?





(i) You are currently viewing the documentation for Pages Router.

Static Site Generation (SSG)

► Examples

If a page uses **Static Generation**, the page HTML is generated at **build time**. That means in production, the page HTML is generated when you run `next build`. This HTML will then be reused on each request. It can be cached by a CDN.

In Next.js, you can statically generate pages **with or without data**. Let's take a look at each case.

Static Generation without data

By default, Next.js pre-renders pages using Static Generation without fetching data. Here's an example:

```
function About() {  
  return <div>About</div>  
}  
  
export default About
```

Note that this page does not need to fetch any external data to be pre-rendered. In cases like this, Next.js generates a single HTML file per page during build time.

Static Generation with data

Some pages require fetching external data for pre-rendering. There are two scenarios, and one or both might apply. In each case, you can use these functions that Next.js provides:

1. Your page **content** depends on external data:

Use `getStaticProps`.

2. Your page **paths** depend on external data: Use

`getStaticPaths` (usually in addition to

`getStaticProps`).

Scenario 1: Your page content depends on external data

Example: Your blog page might need to fetch the list of blog posts from a CMS (content management system).

```
// TODO: Need to fetch `posts` (by calling so
//        before this page can be pre-rendered
export default function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>{post.title}</li>
      ))}
    </ul>
  )
}
```

To fetch this data on pre-render, Next.js allows you to `export` an `async` function called `getStaticProps` from the same file. This function gets called at build time and lets you pass fetched data to the page's `props` on pre-render.

```
export default function Blog({ posts }) {
  // Render posts...
}

// This function gets called at build time
export async function getStaticProps() {
```

```
// Call an external API endpoint to get posts
const res = await fetch('https://.../posts')
const posts = await res.json()

// By returning { props: { posts } }, the B
// will receive `posts` as a prop at build
return {
  props: {
    posts,
  },
}
```

To learn more about how `getStaticProps` works, check out the [Data Fetching documentation](#).

Scenario 2: Your page paths depend on external data

Next.js allows you to create pages with **dynamic routes**. For example, you can create a file called `pages/posts/[id].js` to show a single blog post based on `id`. This will allow you to show a blog post with `id: 1` when you access `posts/1`.

To learn more about dynamic routing, check the [Dynamic Routing documentation](#).

However, which `id` you want to pre-render at build time might depend on external data.

Example: suppose that you've only added one blog post (with `id: 1`) to the database. In this case, you'd only want to pre-render `posts/1` at build time.

Later, you might add the second post with `id: 2`. Then you'd want to pre-render `posts/2` as well.

So your page **paths** that are pre-rendered depend on external data. To handle this, Next.js lets you `export` an `async` function called `getStaticPaths` from a dynamic page (`pages/posts/[id].js` in this case). This function

gets called at build time and lets you specify which paths you want to pre-render.

```
// This function gets called at build time
export async function getStaticPaths() {
  // Call an external API endpoint to get posts
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to pre-render based on posts
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

  // We'll pre-render only these paths at build time
  // { fallback: false } means other routes should be
  // handled by the server
  return { paths, fallback: false }
}
```

Also in `pages/posts/[id].js`, you need to export `getStaticProps` so that you can fetch the data about the post with this `id` and use it to pre-render the page:

```
export default function Post({ post }) {
  // Render post...
}

export async function getStaticPaths() {
  // ...
}

// This also gets called at build time
export async function getStaticProps({ params }) {
  // params contains the post `id`.
  // If the route is like /posts/1, then params.id is `1`
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  // Pass post data to the page via props
  return { props: { post } }
}
```

To learn more about how `getStaticPaths` works, check out the [Data Fetching documentation](#).

When should I use Static Generation?

We recommend using **Static Generation** (with and without data) whenever possible because your page can be built once and served by CDN, which makes it much faster than having a server render the page on every request.

You can use Static Generation for many types of pages, including:

- Marketing pages
- Blog posts and portfolios
- E-commerce product listings
- Help and documentation

You should ask yourself: "Can I pre-render this page **ahead** of a user's request?" If the answer is yes, then you should choose Static Generation.

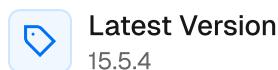
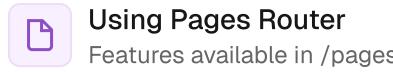
On the other hand, Static Generation is **not** a good idea if you cannot pre-render a page ahead of a user's request. Maybe your page shows frequently updated data, and the page content changes on every request.

In cases like this, you can do one of the following:

- Use Static Generation with **Client-side data fetching**: You can skip pre-rendering some parts of a page and then use client-side JavaScript to populate them. To learn more about this approach, check out the [Data Fetching documentation](#).
- Use **Server-Side Rendering**: Next.js pre-renders a page on each request. It will be slower because the page cannot be cached by a CDN, but the pre-rendered page will always

be up-to-date. We'll talk about this approach below.

Was this helpful?    



(i) You are currently viewing the documentation for Pages Router.

Automatic Static Optimization

Next.js automatically determines that a page is static (can be prerendered) if it has no blocking data requirements. This determination is made by the absence of `getServerSideProps` and `getInitialProps` in the page.

This feature allows Next.js to emit hybrid applications that contain **both server-rendered and statically generated pages**.

Good to know: Statically generated pages are still reactive. Next.js will hydrate your application client-side to give it full interactivity.

One of the main benefits of this feature is that optimized pages require no server-side computation, and can be instantly streamed to the end-user from multiple CDN locations. The result is an *ultra fast* loading experience for your users.

How it works

If `getServerSideProps` or `getInitialProps` is present in a page, Next.js will switch to render the

page on-demand, per-request (meaning [Server-Side Rendering](#)).

If the above is not the case, Next.js will **statically optimize** your page automatically by prerendering the page to static HTML.

During prerendering, the router's `query` object will be empty since we do not have `query` information to provide during this phase. After hydration, Next.js will trigger an update to your application to provide the route parameters in the `query` object.

The cases where the query will be updated after hydration triggering another render are:

- The page is a [dynamic-route](#).
- The page has query values in the URL.
- [Rewrites](#) are configured in your `next.config.js` since these can have parameters that may need to be parsed and provided in the `query`.

To be able to distinguish if the query is fully updated and ready for use, you can leverage the `isReady` field on [next/router](#).

Good to know: Parameters added with [dynamic routes](#) to a page that's using `getStaticProps` will always be available inside the `query` object.

`next build` will emit `.html` files for statically optimized pages. For example, the result for the page `pages/about.js` would be:

```
>_ Terminal   
  
.next/server/pages/about.html
```

And if you add `getServerSideProps` to the page, it will then be JavaScript, like so:

```
>_ Terminal   
.next/server/pages/about.js
```

Caveats

- If you have a `custom App` with `getInitialProps` then this optimization will be turned off in pages without `Static Generation`.
- If you have a `custom Document` with `getInitialProps` be sure you check if `ctx.req` is defined before assuming the page is server-side rendered. `ctx.req` will be `undefined` for pages that are prerendered.
- Avoid using the `asPath` value on `next/router` in the rendering tree until the router's `isReady` field is `true`. Statically optimized pages only know `asPath` on the client and not the server, so using it as a prop may lead to mismatch errors. The [active-class-name example ↗](#) demonstrates one way to use `asPath` as a prop.

Was this helpful?    

 **Using Pages Router**

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

 **Latest Version**

15.5.4



Client-side Rendering (CSR)

In Client-Side Rendering (CSR) with React, the browser downloads a minimal HTML page and the JavaScript needed for the page. The JavaScript is then used to update the DOM and render the page. When the application is first loaded, the user may notice a slight delay before they can see the full page, this is because the page isn't fully rendered until all the JavaScript is downloaded, parsed, and executed.

After the page has been loaded for the first time, navigating to other pages on the same website is typically faster, as only necessary data needs to be fetched, and JavaScript can re-render parts of the page without requiring a full page refresh.

In Next.js, there are two ways you can implement client-side rendering:

1. Using React's `useEffect()` hook inside your pages instead of the server-side rendering methods (`getStaticProps` and `getServerSideProps`).
2. Using a data fetching library like [SWR](#) or [TanStack Query](#) to fetch data on the client (recommended).

Here's an example of using `useEffect()` inside a Next.js page:

JS pages/index.js

```
import React, { useState, useEffect } from 'react'

export function Page() {
  const [data, setData] = useState(null)

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch('https://api.example.com/data')
      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`)
      }
      const result = await response.json()
      setData(result)
    }

    fetchData().catch((e) => {
      // handle the error as needed
      console.error('An error occurred while fetching data')
    })
  }, [])

  return <p>{data ? `Your data: ${data}` : 'Loading ...'}
```

In the example above, the component starts by rendering `Loading ...`. Then, once the data is fetched, it re-renders and displays the data.

Although fetching data in a `useEffect` is a pattern you may see in older React Applications, we recommend using a data-fetching library for better performance, caching, optimistic updates, and more. Here's a minimum example using [SWR ↗](#) to fetch data on the client:

JS pages/index.js

```
import useSWR from 'swr'

export function Page() {
  const { data, error, isLoading } = useSWR('https://api.example.com/data')
```

```
'https://api.example.com/data' ,
```

```
fetcher
```

```
)
```

```
if (error) return <p>Failed to load.</p>
if (isLoading) return <p>Loading...</p>
```

```
return <p>Your Data: {data}</p>
```

```
}
```

Good to know:

Keep in mind that CSR can impact SEO. Some search engine crawlers might not execute JavaScript and therefore only see the initial empty or loading state of your application. It can also lead to performance issues for users with slower internet connections or devices, as they need to wait for all the JavaScript to load and run before they can see the full page. Next.js promotes a hybrid approach that allows you to use a combination of [server-side rendering](#), [static site generation](#), and client-side rendering, **depending on the needs of each page** in your application. In the App Router, you can also use [Loading UI with Suspense](#) to show a loading indicator while the page is being rendered.

Next Steps

Learn about the alternative rendering methods in Next.js.

Server-side ...

Use Server-side Rendering to render pages on...

Static Site G...

Use Static Site Generation (SSG) to pre-render...

ISR

Learn how to
create or update
static pages at...

Was this helpful?    



Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4

Data Fetching

Data fetching in Next.js allows you to render your content in different ways, depending on your application's use case. These include pre-rendering with **Server-side Rendering** or **Static Generation**, and updating or creating content at runtime with **Incremental Static Regeneration**.

Examples

- [Agility CMS Example ↗ \(Demo ↗\)](#)
- [Builder.io Example ↗ \(Demo ↗\)](#)
- [ButterCMS Example ↗ \(Demo ↗\)](#)
- [Contentful Example ↗ \(Demo ↗\)](#)
- [Cosmic Example ↗ \(Demo ↗\)](#)
- [DatoCMS Example ↗ \(Demo ↗\)](#)
- [DotCMS Example ↗ \(Demo ↗\)](#)
- [Drupal Example ↗ \(Demo ↗\)](#)
- [Enterspeed Example ↗ \(Demo ↗\)](#)
- [GraphCMS Example ↗ \(Demo ↗\)](#)
- [Keystone Example ↗ \(Demo ↗\)](#)
- [Kontent.ai Example ↗ \(Demo ↗\)](#)
- [Makeswift Example ↗ \(Demo ↗\)](#)

- [Plasmic Example ↗ \(Demo ↗\)](#)
- [Prepr Example ↗ \(Demo ↗\)](#)
- [Prismic Example ↗ \(Demo ↗\)](#)
- [Sanity Example ↗ \(Demo ↗\)](#)
- [Sitecore XM Cloud Example ↗ \(Demo ↗\)](#)
- [Storyblok Example ↗ \(Demo ↗\)](#)
- [Strapi Example ↗ \(Demo ↗\)](#)
- [TakeShape Example ↗ \(Demo ↗\)](#)
- [Tina Example ↗ \(Demo ↗\)](#)
- [Umbraco Example ↗ \(Demo ↗\)](#)
- [Umbraco Heartcore Example ↗ \(Demo ↗\)](#)
- [Webiny Example ↗ \(Demo ↗\)](#)
- [WordPress Example ↗ \(Demo ↗\)](#)
- [Blog Starter Example ↗ \(Demo ↗\)](#)
- [Static Tweet \(Demo\) ↗](#)

getStaticPro...

Fetch data and generate static pages with...

getStaticPat...

Fetch data and generate static pages with...

Forms and M...

Learn how to handle form submissions and...

getServerSid...

Fetch data on each request with `getServerSideProp

Client-side F...

Learn about client-side data fetching,
and how to use...

Was this helpful?    

 Using Pages Router
Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

 Latest Version
15.5.4



getStaticProps

If you export a function called `getStaticProps` (Static Site Generation) from a page, Next.js will pre-render this page at build time using the props returned by `getStaticProps`.

```
TS pages/index.tsx TypeScript ▾
```

```
import type { InferGetStaticPropsType, GetStaticProps } from 'next'

type Repo = {
  name: string
  stargazers_count: number
}

export const getStaticProps = (async (context) => {
  const res = await fetch('https://api.github.com/repos/vercel/next.js')
  const repo = await res.json()
  return { props: { repo } }
}) satisfies GetStaticProps<{
  repo: Repo
}>

export default function Page({
  repo,
}: InferGetStaticPropsType<typeof getStaticProps>) {
  return repo.stargazers_count
}
```

Note that irrespective of rendering type, any `props` will be passed to the page component and can be viewed on the client-side in the initial HTML. This is to allow the page to be hydrated ↗ correctly. Make sure

that you don't pass any sensitive information that shouldn't be available on the client in `props`.

The [getStaticProps API reference](#) covers all parameters and props that can be used with `getStaticProps`.

When should I use `getStaticProps`?

You should use `getStaticProps` if:

- The data required to render the page is available at build time ahead of a user's request
 - The data comes from a headless CMS
 - The page must be pre-rendered (for SEO) and be very fast — `getStaticProps` generates `HTML` and `JSON` files, both of which can be cached by a CDN for performance
 - The data can be publicly cached (not user-specific). This condition can be bypassed in certain specific situations by using a Middleware to rewrite the path.
-

When does `getStaticProps` run

`getStaticProps` always runs on the server and never on the client. You can validate code written inside `getStaticProps` is removed from the client-side bundle [with this tool ↗](#).

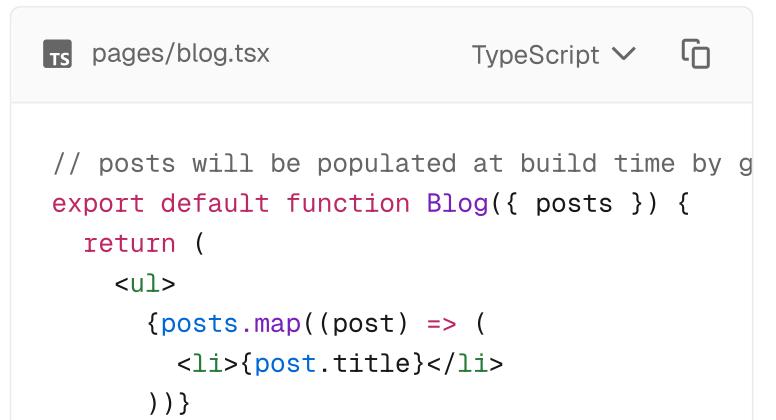
- `getStaticProps` always runs during `next build`
- `getStaticProps` runs in the background when using `fallback: true`
- `getStaticProps` is called before initial render when using `fallback: blocking`
- `getStaticProps` runs in the background when using `revalidate`
- `getStaticProps` runs on-demand in the background when using `revalidate()`

When combined with [Incremental Static Regeneration](#), `getStaticProps` will run in the background while the stale page is being revalidated, and the fresh page served to the browser.

`getStaticProps` does not have access to the incoming request (such as query parameters or HTTP headers) as it generates static HTML. If you need access to the request for your page, consider using [Middleware](#) in addition to `getStaticProps`.

Using `getStaticProps` to fetch data from a CMS

The following example shows how you can fetch a list of blog posts from a CMS.



```
TS pages/blog.tsx TypeScript ▾ ⌂
// posts will be populated at build time by g
export default function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>{post.title}</li>
      ))}
    </ul>
  )
}
```

```
</ul>
)
}

// This function gets called at build time on
// It won't be called on client-side, so you
// direct database queries.
export async function getStaticProps() {
  // Call an external API endpoint to get pos
  // You can use any data fetching library
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // By returning { props: { posts } }, the B
  // will receive `posts` as a prop at build
  return {
    props: {
      posts,
    },
  }
}
```

The `getStaticProps` API reference covers all parameters and props that can be used with `getStaticProps`.

Write server-side code directly

As `getStaticProps` runs only on the server-side, it will never run on the client-side. It won't even be included in the JS bundle for the browser, so you can write direct database queries without them being sent to browsers.

This means that instead of fetching an **API route** from `getStaticProps` (that itself fetches data from an external source), you can write the server-side code directly in `getStaticProps`.

Take the following example. An API route is used to fetch some data from a CMS. That API route is then called directly from `getStaticProps`. This

produces an additional call, reducing performance.

Instead, the logic for fetching the data from the CMS can be shared by using a `lib/` directory.

Then it can be shared with `getStaticProps`.

`JS lib/load-posts.js`

```
// The following function is shared
// with getStaticProps and API routes
// from a `lib/` directory
export async function loadPosts() {
  // Call an external API endpoint to get pos
  const res = await fetch('https://.../posts')
  const data = await res.json()

  return data
}
```

`JS pages/blog.js`

```
// pages/blog.js
import { loadPosts } from '../lib/load-posts'

// This function runs only on the server side
export async function getStaticProps() {
  // Instead of fetching your `/api` route yo
  // function directly in `getStaticProps`
  const posts = await loadPosts()

  // Props returned will be passed to the pag
  return { props: { posts } }
}
```

Alternatively, if you are **not** using API routes to fetch data, then the `fetch()` [↗] API can be used directly in `getStaticProps` to fetch data.

To verify what Next.js eliminates from the client-side bundle, you can use the [next-code-elimination tool](#) [↗].

Statically generates both HTML and JSON

When a page with `getStaticProps` is pre-rendered at build time, in addition to the page HTML file, Next.js generates a JSON file holding the result of running `getStaticProps`.

This JSON file will be used in client-side routing through `next/link` or `next/router`. When you navigate to a page that's pre-rendered using `getStaticProps`, Next.js fetches this JSON file (pre-computed at build time) and uses it as the props for the page component. This means that client-side page transitions will **not** call `getStaticProps` as only the exported JSON is used.

When using Incremental Static Generation, `getStaticProps` will be executed in the background to generate the JSON needed for client-side navigation. You may see this in the form of multiple requests being made for the same page, however, this is intended and has no impact on end-user performance.

Where can I use `getStaticProps`

`getStaticProps` can only be exported from a **page**. You **cannot** export it from non-page files, `_app`, `_document`, or `_error`.

One of the reasons for this restriction is that React needs to have all the required data before the page is rendered.

Also, you must use `export getStaticProps` as a standalone function — it will **not** work if you add `getStaticProps` as a property of the page component.

Good to know: if you have created a [custom app](#), ensure you are passing the `pageProps` to the page component as shown in the linked document, otherwise the props will be empty.

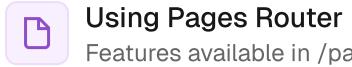
Runs on every request in development

In development (`next dev`), `getStaticProps` will be called on every request.

Preview Mode

You can temporarily bypass static generation and render the page at **request time** instead of build time using [Preview Mode](#). For example, you might be using a headless CMS and want to preview drafts before they're published.

Was this helpful?    



Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



getStaticPaths

If a page has [Dynamic Routes](#) and uses `getStaticProps`, it needs to define a list of paths to be statically generated.

When you export a function called `getStaticPaths` (Static Site Generation) from a page that uses dynamic routes, Next.js will statically pre-render all the paths specified by `getStaticPaths`.

TS pages/repo/[name].tsx

TypeScript ▾



```
import type {
  InferGetStaticPropsType,
  GetStaticProps,
  GetStaticPaths,
} from 'next'

type Repo = {
  name: string
  stargazers_count: number
}

export const getStaticPaths = (async () => {
  return {
    paths: [
      {
        params: {
          name: 'next.js',
        },
        // See the "paths" section below
      ],
      fallback: true, // false or "blocking"
    }
  }) satisfies GetStaticPaths
```

```
export const getStaticProps = (async (context) => {
  const res = await fetch('https://api.github.com/repos/vercel/next.js')
  const repo = await res.json()
  return { props: { repo } }
}) satisfies GetStaticProps<{
  repo: Repo
}>

export default function Page({repo}) {
  const { name, stargazers_count } = repo
  return (
    <div>
      <h1>{name}</h1>
      <p>{stargazers_count} stars</p>
    </div>
  )
}
```

The `getStaticPaths` API reference covers all parameters and props that can be used with `getStaticPaths`.

When should I use `getStaticPaths`?

You should use `getStaticPaths` if you're statically pre-rendering pages that use dynamic routes and:

- The data comes from a headless CMS
- The data comes from a database
- The data comes from the filesystem
- The data can be publicly cached (not user-specific)
- The page must be pre-rendered (for SEO) and be very fast — `getStaticProps` generates `HTML` and `JSON` files, both of which can be cached by a CDN for performance

When does `getStaticPaths` run

`getStaticPaths` will only run during build in production, it will not be called during runtime. You can validate code written inside `getStaticPaths` is removed from the client-side bundle [with this tool ↗](#).

How does `getStaticProps` run with regards to `getStaticPaths`

- `getStaticProps` runs during `next build` for any `paths` returned during build
- `getStaticProps` runs in the background when using `fallback: true`
- `getStaticProps` is called before initial render when using `fallback: blocking`

Where can I use `getStaticPaths`

- `getStaticPaths` **must** be used with `getStaticProps`
- You **cannot** use `getStaticPaths` with `getServerSideProps`
- You can export `getStaticPaths` from a [Dynamic Route](#) that also uses `getStaticProps`
- You **cannot** export `getStaticPaths` from non-page file (e.g. your `components` folder)
- You must export `getStaticPaths` as a standalone function, and not a property of the page component

Runs on every request in development

In development (`next dev`), `getStaticPaths` will be called on every request.

Generating paths on-demand

`getStaticPaths` allows you to control which pages are generated during the build instead of on-demand with `fallback`. Generating more pages during a build will cause slower builds.

You can defer generating all pages on-demand by returning an empty array for `paths`. This can be especially helpful when deploying your Next.js application to multiple environments. For example, you can have faster builds by generating all pages on-demand for previews (but not production builds). This is helpful for sites with hundreds/thousands of static pages.

```
JS pages/posts/[id].js Copy  
  
export async function getStaticPaths() {  
  // When this is true (in preview environment)  
  // prerender any static pages  
  // (faster builds, but slower initial page load)  
  if (process.env.SKIP_BUILD_STATIC_GENERATION)  
    return {  
      paths: [],  
      fallback: 'blocking',  
    }  
  }  
  
  // Call an external API endpoint to get posts  
  const res = await fetch('https://.../posts')  
  const posts = await res.json()  
  
  // Get the paths we want to prerender based on the posts
```

```
// In production environments, prerender all
// (slower builds, but faster initial page
const paths = posts.map((post) => ({
  params: { id: post.id },
}))  
  
// { fallback: false } means other routes skip
return { paths, fallback: false }
}
```

Was this helpful?     



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



How to create forms with API Routes

Forms enable you to create and update data in web applications. Next.js provides a powerful way to handle data mutations using **API Routes**. This guide will walk you through how to handle form submission on the server.

Server Forms

To handle form submissions on the server, create an API endpoint that securely mutates data.

The code editor shows a file named `pages/api/submit.ts` with the following content:

```
import type { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const data = req.body
  const id = await createItem(data)
  res.status(200).json({ id })
}
```

Then, call the API Route from the client with an event handler:



```
import { FormEvent } from 'react'

export default function Page() {
  async function onSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault()

    const formData = new FormData(event.currentTarget)
    const response = await fetch('/api/submit', {
      method: 'POST',
      body: formData,
    })

    // Handle response if necessary
    const data = await response.json()
    // ...
  }

  return (
    <form onSubmit={onSubmit}>
      <input type="text" name="name" />
      <button type="submit">Submit</button>
    </form>
  )
}
```

Good to know:

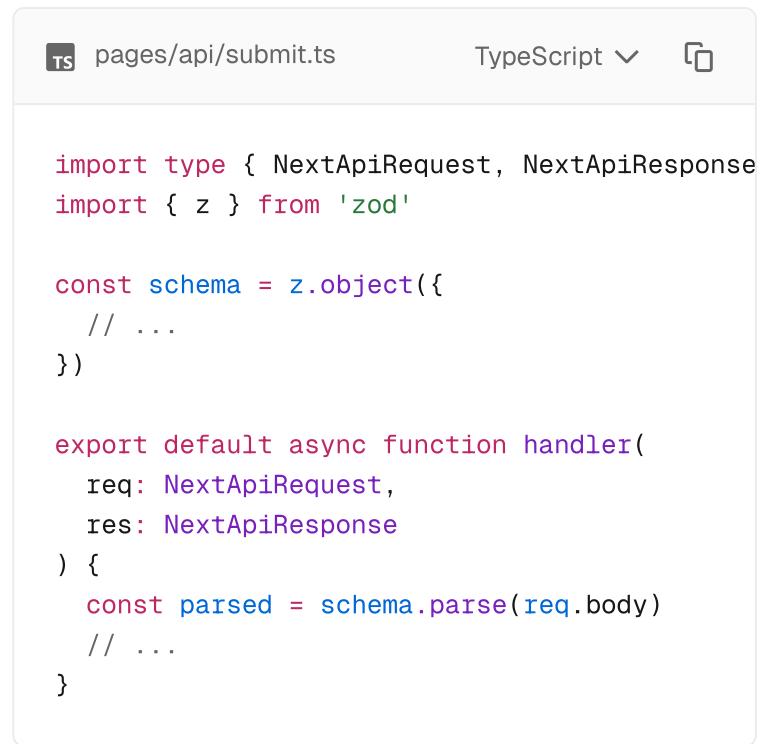
- API Routes [do not specify CORS headers ↗](#), meaning they are same-origin only by default.
- Since API Routes run on the server, we're able to use sensitive values (like API keys) through [Environment Variables](#) without exposing them to the client. This is critical for the security of your application.

Form validation

We recommend using HTML validation like `required` and `type="email"` for basic client-side form validation.

For more advanced server-side validation, you can use a schema validation library like [zod ↗](#) to

validate the form fields before mutating the data:



pages/api/submit.ts TypeScript

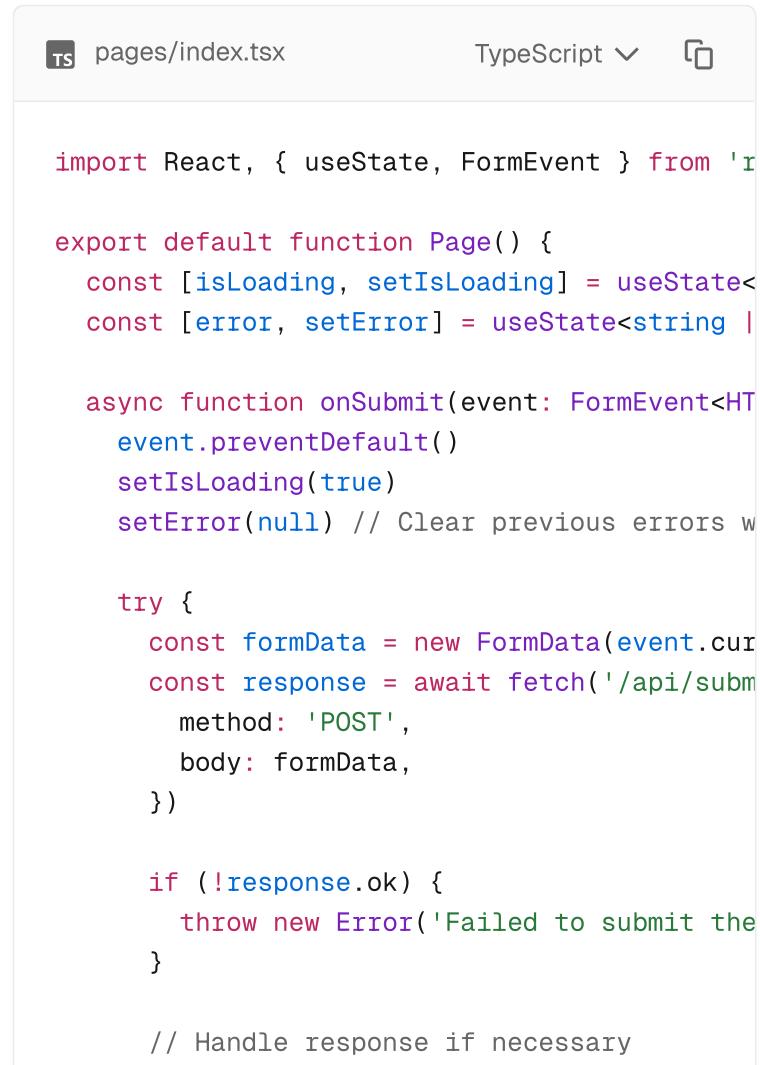
```
import type { NextApiRequest, NextApiResponse } from 'zod'

const schema = z.object({
  // ...
})

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const parsed = schema.parse(req.body)
  // ...
}
```

Error handling

You can use React state to show an error message when a form submission fails:



pages/index.tsx TypeScript

```
import React, { useState, FormEvent } from 'react'

export default function Page() {
  const [isLoading, setIsLoading] = useState(false)
  const [error, setError] = useState<string>('')

  async function onSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault()
    setIsLoading(true)
    setError(null) // Clear previous errors when starting a new submission

    try {
      const formData = new FormData(event.currentTarget)
      const response = await fetch('/api/submit', {
        method: 'POST',
        body: formData,
      })
      if (!response.ok) {
        throw new Error('Failed to submit the form')
      }
    } catch (err) {
      setError(err.message)
    }
  }

  return (
    <div>
      <h1>Submit a form</h1>
      <form onSubmit={onSubmit}>
        <input type="text" name="name" />
        <input type="text" name="email" />
        <button type="submit">Submit</button>
      </form>
      {error ? <p>{error}</p> : null}
    </div>
  )
}
```

```

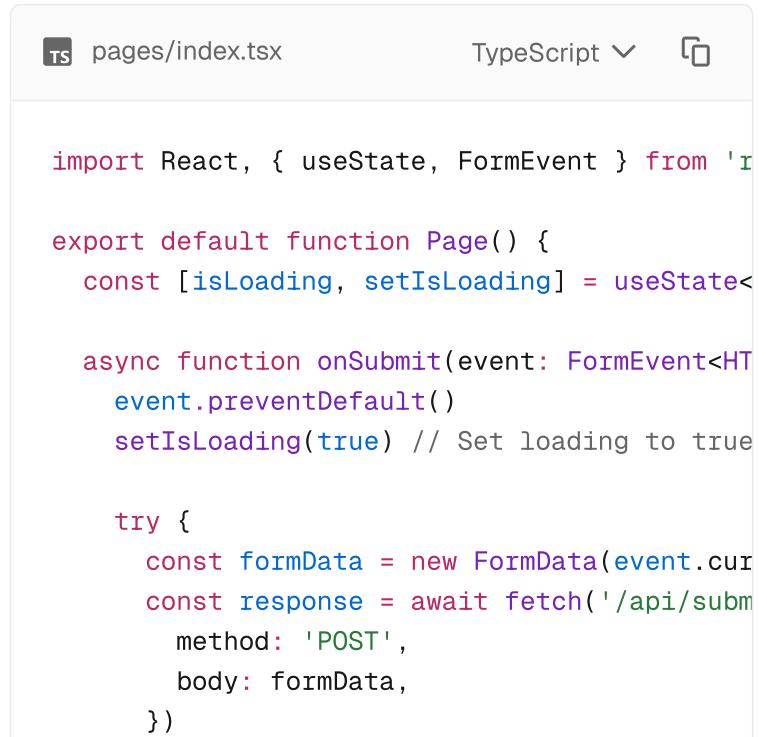
        const data = await response.json()
        // ...
    } catch (error) {
        // Capture the error message to display
        setError(error.message)
        console.error(error)
    } finally {
        setIsLoading(false)
    }
}

return (
    <div>
        {error && <div style={{ color: 'red' }}>
            <form onSubmit={onSubmit}>
                <input type="text" name="name" />
                <button type="submit" disabled={isLoa
                    isLoading ? 'Loading...' : 'Submit'
                </button>
            </form>
        </div>
    )
}

```

Displaying loading state

You can use React state to show a loading state when a form is submitting on the server:



The screenshot shows a code editor interface with a TypeScript file named `pages/index.tsx`. The code implements a `Page` component that handles form submission and displays a loading state.

```

import React, { useState, FormEvent } from 'react'

export default function Page() {
    const [isLoading, setIsLoading] = useState(false)

    async function onSubmit(event: FormEvent<HTMLFormElement>)
        event.preventDefault()
        setIsLoading(true) // Set loading to true

        try {
            const formData = new FormData(event.currentTarget)
            const response = await fetch('/api/submit', {
                method: 'POST',
                body: formData,
            })
        }
    }
}

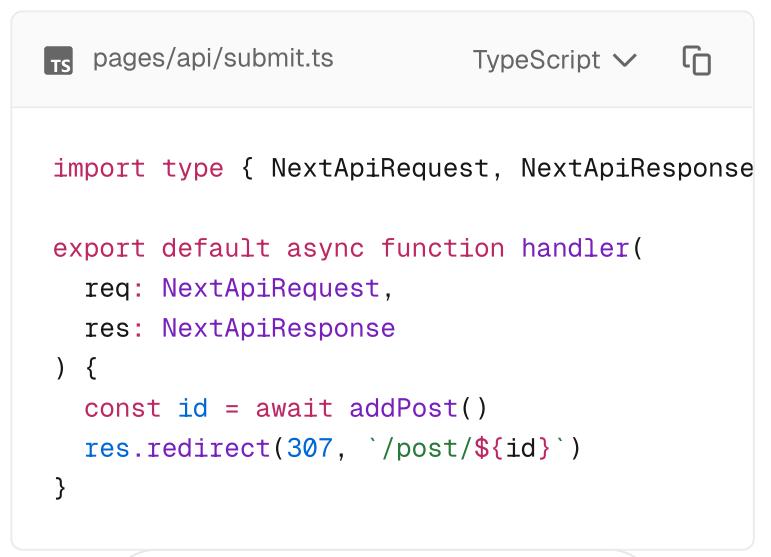
```

```
// Handle response if necessary
const data = await response.json()
// ...
} catch (error) {
  // Handle error if necessary
  console.error(error)
} finally {
  setIsLoading(false) // Set loading to false
}
}

return (
  <form onSubmit={onSubmit}>
    <input type="text" name="name" />
    <button type="submit" disabled={isLoading}><span>{isLoading ? 'Loading...' : 'Submit'}</span>
    </button>
  </form>
)
}
```

Redirecting

If you would like to redirect the user to a different route after a mutation, you can `redirect` to any absolute or relative URL:

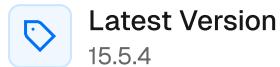


```
pages/api/submit.ts  TypeScript ▾
```

```
import type { NextApiRequest, NextApiResponse }

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const id = await addPost()
  res.redirect(307, `/post/${id}`)
}
```

Was this helpful?    



ⓘ You are currently viewing the documentation for Pages Router.

getServerSideProps

`getServerSideProps` is a Next.js function that can be used to fetch data and render the contents of a page at request time.

Example

You can use `getServerSideProps` by exporting it from a Page Component. The example below shows how you can fetch data from a 3rd party API in `getServerSideProps`, and pass the data to the page as props:

```
TS pages/index.tsx TypeScript ▾ ⌂

import type { InferGetServerSidePropsType, Ge

type Repo = {
  name: string
  stargazers_count: number
}

export const getServerSideProps = (async () =
  // Fetch data from external API
  const res = await fetch('https://api.github
  const repo: Repo = await res.json()
  // Pass data to the page via props
  return { props: { repo } }
) satisfies GetServerSideProps<{ repo: Repo

export default function Page({
```

```
repo,  
}: InferGetServerSidePropsType<typeof getServ  
return (  
  <main>  
    <p>{repo.stargazers_count}</p>  
  </main>  
)  
}
```

When should I use `getServerSideProps`?

You should use `getServerSideProps` if you need to render a page that relies on personalized user data, or information that can only be known at request time. For example, `authorization` headers or a geolocation.

If you do not need to fetch the data at request time, or would prefer to cache the data and pre-rendered HTML, we recommend using `getStaticProps`.

Behavior

- `getServerSideProps` runs on the server.
- `getServerSideProps` can only be exported from a `page`.
- `getServerSideProps` returns JSON.
- When a user visits a page, `getServerSideProps` will be used to fetch data at request time, and the data is used to render the initial HTML of the page.
- `props` passed to the page component can be viewed on the client as part of the initial HTML.

This is to allow the page to be [hydrated](#) correctly. Make sure that you don't pass any sensitive information that shouldn't be available on the client in `props`.

- When a user visits the page through `next/link` or `next/router`, Next.js sends an API request to the server, which runs `getServerSideProps`.
- You do not have to call a Next.js [API Route](#) to fetch data when using `getServerSideProps` since the function runs on the server. Instead, you can call a CMS, database, or other third-party APIs directly from inside `getServerSideProps`.

Good to know:

- See [getServerSideProps API reference](#) for parameters and props that can be used with `getServerSideProps`.
- You can use the [next-code-elimination tool](#) to verify what Next.js eliminates from the client-side bundle.

Error Handling

If an error is thrown inside `getServerSideProps`, it will show the `pages/500.js` file. Check out the documentation for [500 page](#) to learn more on how to create it. During development, this file will not be used and the development error overlay will be shown instead.

Edge Cases

Caching with Server-Side Rendering (SSR)

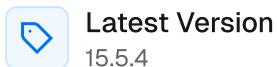
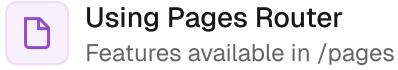
You can use caching headers (`Cache-Control`) inside `getServerSideProps` to cache dynamic responses. For example, using `stale-while-revalidate` [↗](#).

```
// This value is considered fresh for ten sec
// If a request is repeated within the next 1
// cached value will still be fresh. If the r
// the cached value will be stale but still r
//
// In the background, a revalidation request
// with a fresh value. If you refresh the pag
export async function getServerSideProps({ re
  res.setHeader(
    'Cache-Control',
    'public, s-maxage=10, stale-while-revalid
  )

  return {
    props: {},
  }
}
```

However, before reaching for `cache-control`, we recommend seeing if `getStaticProps` with ISR is a better fit for your use case.

Was this helpful?    



(i) You are currently viewing the documentation for Pages Router.

Client-side Fetching

Client-side data fetching is useful when your page doesn't require SEO indexing, when you don't need to pre-render your data, or when the content of your pages needs to update frequently. Unlike the server-side rendering APIs, you can use client-side data fetching at the component level.

If done at the page level, the data is fetched at runtime, and the content of the page is updated as the data changes. When used at the component level, the data is fetched at the time of the component mount, and the content of the component is updated as the data changes.

It's important to note that using client-side data fetching can affect the performance of your application and the load speed of your pages. This is because the data fetching is done at the time of the component or pages mount, and the data is not cached.

Client-side data fetching with useEffect

The following example shows how you can fetch data on the client side using the useEffect hook.

```
import { useState, useEffect } from 'react'

function Profile() {
  const [data, setData] = useState(null)
  const [isLoading, setLoading] = useState(true)

  useEffect(() => {
    fetch('/api/profile-data')
      .then((res) => res.json())
      .then((data) => {
        setData(data)
        setLoading(false)
      })
  }, [])

  if (isLoading) return <p>Loading...</p>
  if (!data) return <p>No profile data</p>

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

Client-side data fetching with SWR

The team behind Next.js has created a React Hook library for data fetching called [SWR](#). It is **highly recommended** if you are fetching data on the client-side. It handles caching, revalidation, focus tracking, refetching on intervals, and more.

Using the same example as above, we can now use SWR to fetch the profile data. SWR will automatically cache the data for us and will revalidate the data if it becomes stale.

For more information on using SWR, check out the [SWR docs](#).

```
import useSWR from 'swr'

const fetcher = (...args) => fetch(...args).then(res =>
  res.json())

function Profile() {
  const { data, error } = useSWR('/api/profile', fetcher)

  if (error) return <div>Failed to load</div>
  if (!data) return <div>Loading...</div>

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4

Configuring

Next.js allows you to customize your project to meet specific requirements. This includes integrations with TypeScript, ESLint, and more, as well as internal configuration options such as Absolute Imports and Environment Variables.

Error Handling

Handle errors in your Next.js app.

Was this helpful?



Using App Router
Features available in /app

Latest Version
15.5.4

Error Handling

Errors can be divided into two categories:

[expected errors](#) and [uncaught exceptions](#). This page will walk you through how you can handle these errors in your Next.js application.

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.

Server Functions

You can use the [useActionState](#) hook to handle expected errors in [Server Functions](#).

For these errors, avoid using `try / catch` blocks and throw errors. Instead, model expected errors as return values.

TS app/actions.ts TypeScript

```
'use server'

export async function createPost(prevState: a
  const title = formData.get('title')
  const content = formData.get('content')

  const res = await fetch('https://api.vercel
```

```
        method: 'POST',
        body: { title, content },
    })
const json = await res.json()

if (!res.ok) {
    return { message: 'Failed to create post' }
}
}
```

You can pass your action to the `useActionState` hook and use the returned `state` to display an error message.

```
TS app/ui/form.tsx TypeScript ▾
```

```
'use client'

import { useActionState } from 'react'
import { createPost } from '@/app/actions'

const initialState = {
    message: '',
}

export function Form() {
    const [state, formAction, pending] = useAct

    return (
        <form action={formAction}>
            <label htmlFor="title">Title</label>
            <input type="text" id="title" name="tit
            <label htmlFor="content">Content</label>
            <textarea id="content" name="content" r
                {state?.message && <p aria-live="polite
                    <button disabled={pending}>Create Post<
                </form>
            )
        }
    }
```

Server Components

When fetching data inside of a Server Component, you can use the response to conditionally render an error message or `redirect`.

```
TS app/page.tsx TypeScript ▾
```

```
export default async function Page() {
  const res = await fetch(`https://...`)
  const data = await res.json()

  if (!res.ok) {
    return 'There was an error.'
  }

  return '...'
}
```

Not found

You can call the `notFound` function within a route segment and use the `not-found.js` file to show a 404 UI.

```
TS app/blog/[slug]/page.tsx TypeScript ▾ ⌂

import {getPostBySlug} from '@/lib/posts'

export default async function Page({ params }) {
  const { slug } = await params
  const post = getPostBySlug(slug)

  if (!post) {
    notFound()
  }

  return <div>{post.title}</div>
}
```

```
TS app/blog/[slug]/not-found.tsx TypeScript ▾ ⌂
```

```
export default function NotFound() {
  return <div>404 - Page Not Found</div>
}
```

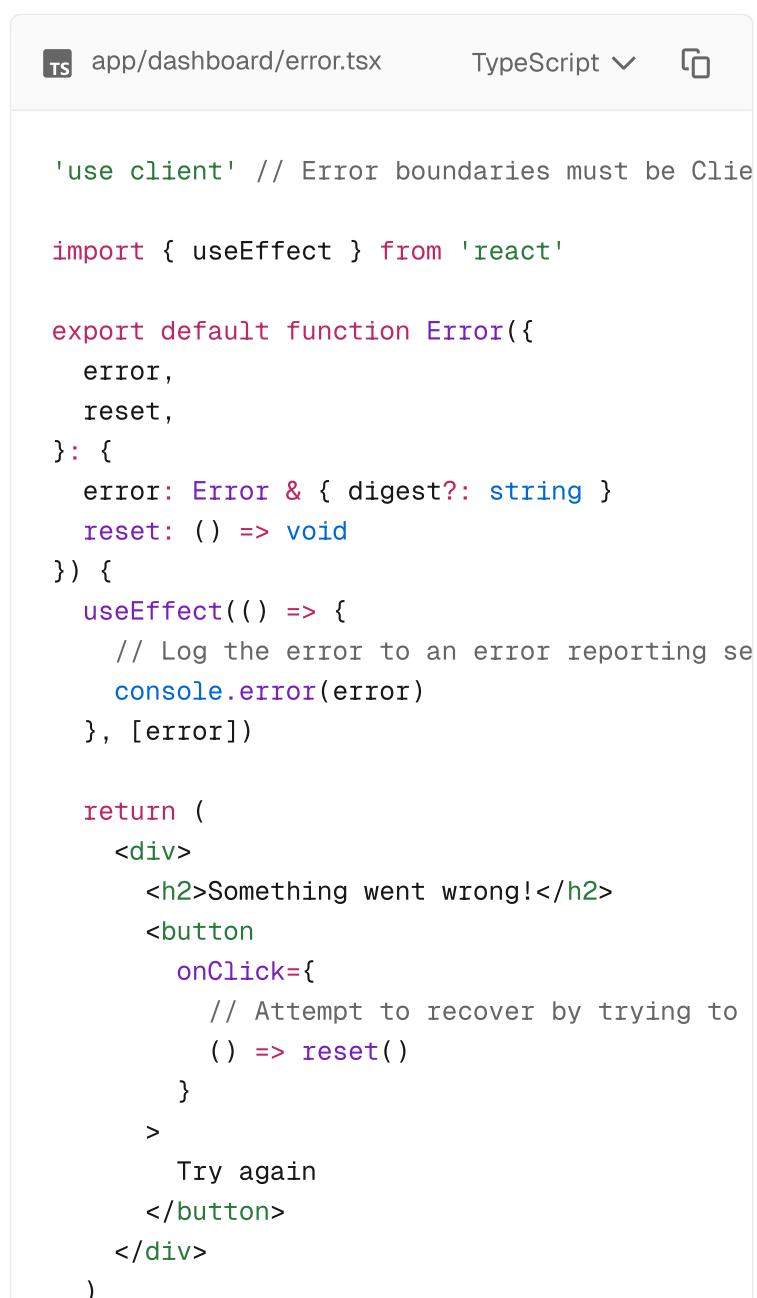
Handling uncaught exceptions

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.

Nested 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.js` file inside a route segment and exporting a React component:



```
'use client' // Error boundaries must be Client Components

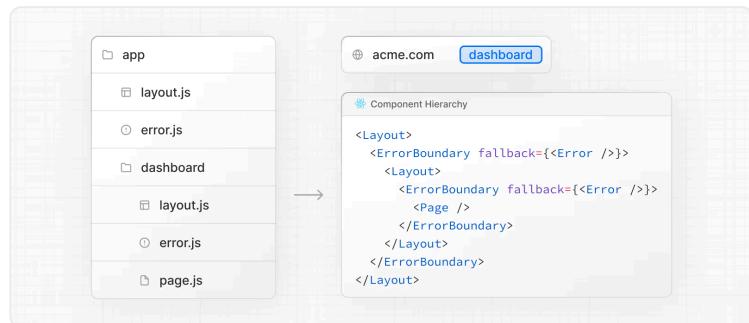
import { useEffect } from 'react'

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
        () => reset()
      }
    >
      Try again
    </button>
  </div>
)
```

```
}
```

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](#).



Error boundaries don't catch errors inside event handlers. They're designed to catch errors [during rendering](#) ↗ to show a **fallback UI** instead of crashing the whole app.

In general, errors in event handlers or async code aren't handled by error boundaries because they run after rendering.

To handle these cases, catch the error manually and store it using `useState` or `useReducer`, then update the UI to inform the user.

```
'use client'

import { useState } from 'react'

export function Button() {
  const [error, setError] = useState(null)

  const handleClick = () => {
    try {
      // do some work that might fail
      throw new Error('Exception')
    } catch (reason) {
      setError(reason)
    }
  }

  if (error) {
```

```
    /* render fallback UI */
}

return (
  <button type="button" onClick={handleClick}
    Click me
  </button>
)
}
```

Note that unhandled errors inside `startTransition` from `useTransition`, will bubble up to the nearest error boundary.

```
'use client'

import { useTransition } from 'react'

export function Button() {
  const [pending, startTransition] = useTrans

  const handleClick = () =>
    startTransition(() => {
      throw new Error('Exception')
    })

  return (
    <button type="button" onClick={handleClick}
      Click me
    </button>
  )
}
```

Global errors

While less common, you can handle errors in the root layout using the `global-error.js` file, 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
    <html>
      <body>
        <h2>Something went wrong!</h2>
        <button onClick={() => reset()}>Try again</button>
      </body>
    </html>
  )
}
```

API Reference

Learn more about the features mentioned in this page by reading the API Reference.

redirect

API Reference for the redirect function.

error.js

API reference for the error.js special file.

notFound

API Reference for the notFound function.

not-found.js

API reference for the not-found.js file.

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



API Reference

Components

API Reference for Next.js built-in components in th...

File-system ...

API Reference for Next.js file-system conventions.

Functions

API Reference for Functions and Hooks in Pages...

Configuration

Learn how to configure your Next.js application.

CLI

API Reference for the Next.js Command Line...

Edge Runtime

API Reference for the Edge Runtime.

Turbopack

Turbopack is an
incremental
bundler optimize...

Was this helpful?    

Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.

Latest Version

15.5.4



Components

Font

API Reference for the Font Module

Form

Learn how to use the `<Form>` component to...

Head

Add custom elements to the `head` of your...

Image

Optimize Images in your Next.js Application using...

Image (Legacy)

Backwards compatible Image Optimization with...

Link

API reference for the `<Link>` component.

Script

Optimize third-party scripts in your Next.js...

Was this helpful?    

 Using Pages Router
Features available in /pages

 Latest Version
15.5.4

 You are currently viewing the documentation for Pages Router.

Font Module

`next/font` automatically optimizes your fonts (including custom fonts) and removes external network requests for improved privacy and performance.

It includes **built-in automatic self-hosting** for any font file. This means you can optimally load web fonts with no [layout shift ↗](#).

You can also conveniently use all [Google Fonts ↗](#). 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.**

To use the font in all your pages, add it to `_app.js` file under `/pages` as shown below:

```
js pages/_app.js

import { Inter } from 'next/font/google'

// If loading a variable font, you don't need
const inter = Inter({ subsets: ['latin'] })

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={inter.className}>
      <Component {...pageProps} />
    </main>
  )
}
```

 Watch: Learn more about using `next/font` →
[YouTube \(6 minutes\)](#).

Reference

Key	font/google	font/local	Ty
<code>src</code>	X	✓	St or of Ol
<code>weight</code>	✓	✓	St or
<code>style</code>	✓	✓	St or
<code>subsets</code>	✓	X	Ar St
<code>axes</code>	✓	X	Ar St
<code>display</code>	✓	✓	St
<code>preload</code>	✓	✓	Bo
<code>fallback</code>	✓	✓	Ar St
<code>adjustFontFallback</code>	✓	✓	Bo or St
<code>variable</code>	✓	✓	St
<code>declarations</code>	X	✓	Ar Ol

`src`

The path of the font file as a string or an array of objects (with type

```
Array<{path: string, weight?: string, style?: string}>
```

) relative to the directory where the font loader function is called.

Used in `next/font/local`

- Required

Examples:

- `src: './fonts/my-font.woff2'` where `my-font.woff2` is placed in a directory named `fonts` inside the `app` directory
- `src:[{path: './inter/Inter-Thin.ttf', weight: '100',},{path: './inter/Inter-Regular.ttf',weight: '400',},{path: './inter/Inter-Bold-Italic.ttf', weight: '700',style: 'italic',},]`
- if the font loader function is called in `app/page.tsx` using `src:'../styles/fonts/my-font.ttf'`, then `my-font.ttf` is placed in `styles/fonts` at the root of the project

weight

The font `weight` ↗ with the following possibilities:

- A string with possible values of the weights available for the specific font or a range of values if it's a [variable](#) ↗ font
- An array of weight values if the font is not a [variable google font](#) ↗ . It applies to `next/font/google` only.

Used in `next/font/google` and `next/font/local`

- Required if the font being used is **not** variable ↗

Examples:

- `weight: '400'`: A string for a single weight value - for the font `Inter` ↗, the possible values are `'100'`, `'200'`, `'300'`, `'400'`, `'500'`, `'600'`, `'700'`, `'800'`, `'900'` or `'variable'` where `'variable'` is the default)
- `weight: '100 900'`: A string for the range between `100` and `900` for a variable font
- `weight: ['100', '400', '900']`: An array of 3 possible values for a non variable font

style

The font `style` ↗ with the following possibilities:

- A string `value` ↗ with default value of `'normal'`
- An array of style values if the font is not a **variable google font** ↗. It applies to `next/font/google` only.

Used in `next/font/google` and

`next/font/local`

- Optional

Examples:

- `style: 'italic'`: A string - it can be `normal` or `italic` for `next/font/google`
- `style: 'oblique'`: A string - it can take any value for `next/font/local` but is expected to come from **standard font styles** ↗
- `style: ['italic', 'normal']`: An array of 2 values for `next/font/google` - the values are from `normal` and `italic`

subsets

The font `subsets` ↗ defined by an array of string values with the names of each subset you would like to be [preloaded](#). Fonts specified via `subsets` will have a `link preload` tag injected into the head when the `preload` option is true, which is the default.

Used in `next/font/google`

- Optional

Examples:

- `subsets: ['latin']`: An array with the subset `latin`

You can find a list of all subsets on the Google Fonts page for your font.

axes

Some variable fonts have extra `axes` that can be included. By default, only the font weight is included to keep the file size down. The possible values of `axes` depend on the specific font.

Used in `next/font/google`

- Optional

Examples:

- `axes: ['sln1']`: An array with value `sln1` for the `Inter` variable font which has `sln1` as additional `axes` as shown [here](#) ↗. You can find the possible `axes` values for your font by using the filter on the [Google variable fonts page](#) ↗ and looking for axes other than `wght`

display

The font `display` ↗ with possible string `values` ↗ of `'auto'`, `'block'`, `'swap'`, `'fallback'` or `'optional'` with default value of `'swap'`.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `display: 'optional'`: A string assigned to the `optional` value

preload

A boolean value that specifies whether the font should be `preloaded` or not. The default is `true`.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `preload: false`

fallback

The fallback font to use if the font cannot be loaded. An array of strings of fallback fonts with no default.

- Optional

Used in `next/font/google` and `next/font/local`

Examples:

- `fallback: ['system-ui', 'arial']`: An array setting the fallback fonts to `system-ui` or

arial

adjustFontFallback

- For `next/font/google`: A boolean value that sets whether an automatic fallback font should be used to reduce [Cumulative Layout Shift ↗](#). The default is `true`.
- For `next/font/local`: A string or boolean `false` value that sets whether an automatic fallback font should be used to reduce [Cumulative Layout Shift ↗](#). The possible values are `'Arial'`, `'Times New Roman'` or `false`. The default is `'Arial'`.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `adjustFontFallback: false`: for `next/font/google`
- `adjustFontFallback: 'Times New Roman'`: for `next/font/local`

variable

A string value to define the CSS variable name to be used if the style is applied with the [CSS variable method](#).

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- variable: '--my-font' : The CSS variable --my-font is declared

declarations

An array of font face descriptor ↗ key-value pairs that define the generated @font-face further.

Used in next/font/local

- Optional

Examples:

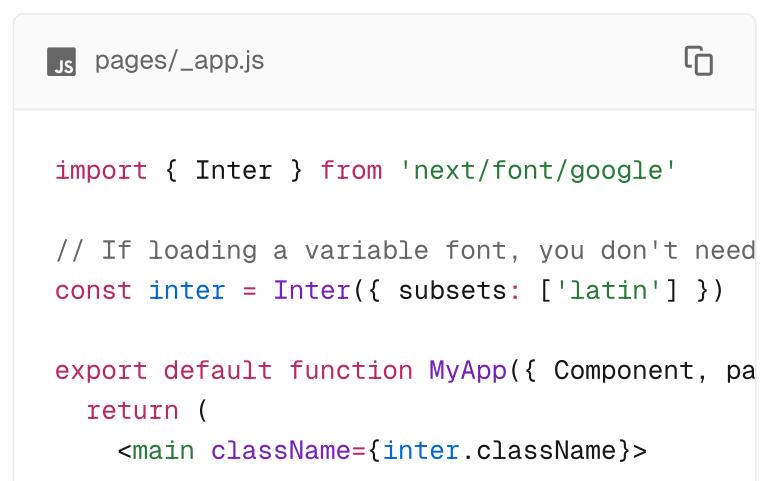
- declarations: [{ prop: 'ascent-override', value: '90%' }]

Examples

Google Fonts

To use a Google font, import it from next/font/google as a function. We recommend using variable fonts ↗ for the best performance and flexibility.

To use the font in all your pages, add it to _app.js file under /pages as shown below:



```
js pages/_app.js

import { Inter } from 'next/font/google'

// If loading a variable font, you don't need
const inter = Inter({ subsets: ['latin'] })

export default function MyApp({ Component, page }) {
  return (
    <main className={inter.className}>
      {page}
    </main>
  )
}
```

```
<Component {...pageProps} />
</main>
)
}
```

If you can't use a variable font, you will **need to specify a weight**:

js pages/_app.js

```
import { Roboto } from 'next/font/google'

const roboto = Roboto({
  weight: '400',
  subsets: ['latin'],
})

export default function MyApp({ Component, pa
  return (
    <main className={roboto.className}>
      <Component {...pageProps} />
    </main>
  )
}
```

You can specify multiple weights and/or styles by using an array:

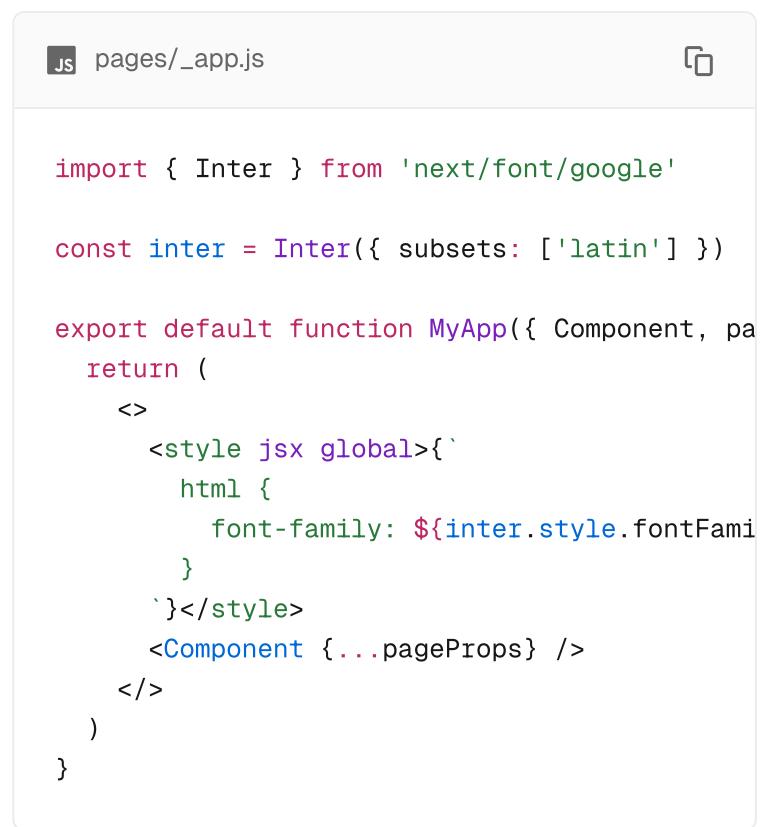
js app/layout.js

```
const roboto = Roboto({
  weight: ['400', '700'],
  style: ['normal', 'italic'],
  subsets: ['latin'],
  display: 'swap',
})
```

Good to know: Use an underscore (_) for font names with multiple words. E.g. Roboto Mono should be imported as Roboto_Mono .

Apply the font in <head>

You can also use the font without a wrapper and `className` by injecting it inside the `<head>` as follows:



The screenshot shows a code editor window with the file name `pages/_app.js` at the top. The code inside the file is 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:



The screenshot shows a code editor window with the file name `pages/index.js` at the top. The code inside the file is as follows:

```
import { Inter } from 'next/font/google'

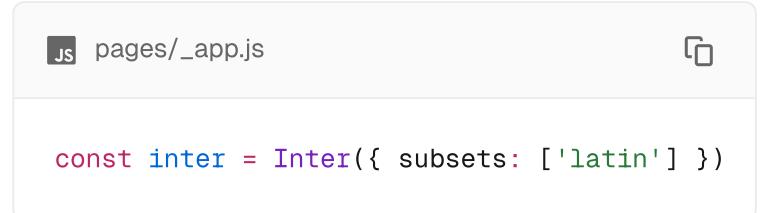
const inter = Inter({ subsets: ['latin'] })

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:



```
JS pages/_app.js

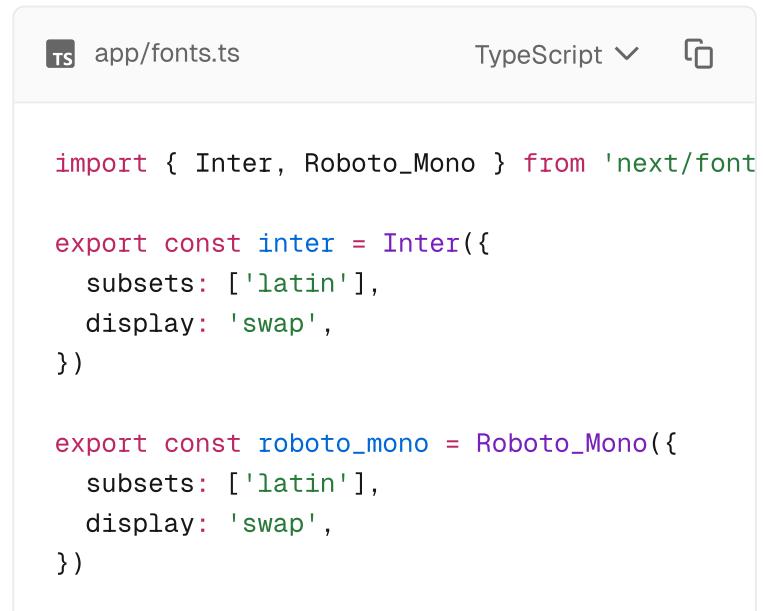
const inter = Inter({ subsets: ['latin'] })
```

View the [Font API Reference](#) for more information.

Using Multiple Fonts

You can import and use multiple fonts in your application. There are two approaches you can take.

The first approach is to create a utility function that exports a font, imports it, and applies its `className` where needed. This ensures the font is preloaded only when it's rendered:



```
TS app/fonts.ts TypeScript ▾

import { Inter, Roboto_Mono } from 'next/font'

export const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
})

export const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  display: 'swap',
})
```

In the example above, `Inter` will be applied globally, and `Roboto Mono` can be imported and applied as needed.

Alternatively, you can create a [CSS variable](#) and use it with your preferred CSS solution:



```
app/global.css

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.



```
pages/_app.js

import localFont from 'next/font/local'

// Font files can be colocated inside of `page` components
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` integrates seamlessly with [Tailwind CSS](#) ↗ using [CSS variables](#).

In the example below, we use the `Inter` and `Roboto_Mono` fonts from `next/font/google` (you can use any Google Font or Local Font). Use the `variable` option to define a CSS variable name, such as `inter` and `roboto_mono` for these fonts, respectively. Then, apply `inter.variable` and `roboto_mono.variable` to include the CSS variables in your HTML document.

Good to know: You can add these variables to the `<html>` or `<body>` tag, depending on your preference, styling needs or project requirements.

js pages/_app.js

```
import { Inter } from 'next/font/google'

const inter = Inter({
  subsets: ['latin'],
  variable: '--font-inter',
})

const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  display: 'swap',
  variable: '--font-roboto-mono',
})

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={`${inter.variable} ${roboto_mono.variable}`}>
      <Component {...pageProps} />
    </main>
  )
}
```

Finally, add the CSS variable to your [Tailwind CSS config](#):

global.css

```
@import 'tailwindcss';

@theme inline {
  --font-sans: var(--font-inter);
  --font-mono: var(--font-roboto-mono);
}
```

Tailwind CSS v3

tailwind.config.js

```
/** @type {import('tailwindcss').Config} */
module.exports = {
```

```
content: [
  './pages/**/*.{js,ts,jsx,tsx}',
  './components/**/*.{js,ts,jsx,tsx}',
  './app/**/*.{js,ts,jsx,tsx}',
],
theme: {
  extend: {
    fontFamily: {
      sans: ['var(--font-inter)'],
      mono: ['var(--font-roboto-mono)'],
    },
  },
},
plugins: [],
}
```

You can now use the `font-sans` and `font-mono` utility classes to apply the font to your elements.

```
<p class="font-sans ...>The quick brown fox
<p class="font-mono ...>The quick brown fox
```

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
```

`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:



The screenshot shows a code editor window for a file named "app/page.tsx". The file contains the following TypeScript code:

```
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.



The screenshot shows a code editor window for a file named "app/page.tsx". The file contains the following TypeScript code:

```
<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:



The screenshot shows a code editor window for a file named "styles/component.module.css". The file contains the following CSS code:

```
.text {
  font-family: var(--font-inter);
```

```
    font-weight: 200;
    font-style: italic;
}
```

In the example above, the text `Hello World` is styled using the `Inter` font and the generated font fallback with `font-weight: 200` and `font-style: italic`.

Using a font definitions file

Every time you call the `localFont` or Google font function, that font will be hosted as one instance in your application. Therefore, if you need to use the same font in multiple places, you should load it in one place and import the related font object where you need it. This is done using a font definitions file.

For example, create a `fonts.ts` file in a `styles` folder at the root of your app directory.

Then, specify your font definitions as follows:



```
TS styles/fonts.ts TypeScript ▾ ⌂

import { Inter, Lora, Source_Sans_3 } from 'next/font/local'
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 GreatVibe is the file name
const greatVibes = localFont({ src: './GreatVibe.woff2' })

export { inter, lora, sourceCodePro400, sourceCodePro700, greatVibes }
```

You can now use these definitions in your code as follows:

```

import { inter, lora, sourceCodePro700, greatVibes } from './fonts'

export default function Page() {
  return (
    <div>
      <p className={inter.className}>Hello world using Source_Sans_3 font
      <p style={lora.style}>Hello world using Source_Sans_3 font
      <p className={sourceCodePro700.className}>Hello world using Source_Sans_3 font
      <p className={greatVibes.className}>My font
    </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'

```

Preloading

When a font function is called on a page of your site, it is not globally available and preloaded on all routes. Rather, the font is only preloaded on the

related route/s based on the type of file where it is used:

- if it's a [unique page](#), it is preloaded on the unique route for that page
 - if it's in the [custom App](#), it is preloaded on all the routes of the site under `/pages`
-

Version Changes

Version	Changes
---------	---------

v13.2.0	<code>@next/font</code> renamed to <code>next/font</code> . Installation no longer required.
---------	---

v13.0.0	<code>@next/font</code> was added.
---------	------------------------------------

Was this helpful?    

Using Pages Router
Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

Latest Version
15.5.4



Form

The `<Form>` component extends the HTML `<form>` element to provide **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:

```
JS /ui/search.js TypeScript   
  
import Form from 'next/form'  
  
export default function Page() {  
  return (  
    <Form action="/search">  
      /* On submission, the input value will  
       * be added 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:
 - 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
<code>action</code>	<code>action="/search"</code>	<code>string</code> (URL or relative path)	Yes
<code>replace</code>	<code>replace={false}</code>	<code>boolean</code>	-
<code>scroll</code>	<code>scroll={true}</code>	<code>boolean</code>	-

- **`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.

Caveats

- `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.

Was this helpful?    

Using Pages Router
Features available in /pages



(i) You are currently viewing the documentation for
Pages Router.

Latest Version
15.5.4



Head

We expose a built-in component for appending elements to the `head` of the page:

```
import Head from 'next/head'

function IndexPage() {
  return (
    <div>
      <Head>
        <title>My page title</title>
      </Head>
      <p>Hello world!</p>
    </div>
  )
}

export default IndexPage
```

Avoid duplicated tags

To avoid duplicate tags in your `head` you can use the `key` property, which will make sure the tag is only rendered once, as in the following example:

```
import Head from 'next/head'

function IndexPage() {
  return (
    <div>
      <Head>
        <link key="css" href="/styles.css" />
      </Head>
    </div>
  )
}

export default IndexPage
```

```
<title>My page title</title>
<meta property="og:title" content="My
</Head>
<Head>
<meta property="og:title" content="My
</Head>
<p>Hello world!</p>
</div>
)
}

export default IndexPage
```

In this case only the second

`<meta property="og:title" />` is rendered.
`meta` tags with duplicate `key` attributes are automatically handled.

Good to know: `<title>` and `<base>` tags are automatically checked for duplicates by Next.js, so using `key` is not necessary for these tags.

The contents of `head` get cleared upon unmounting the component, so make sure each page completely defines what it needs in `head`, without making assumptions about what other pages added.

Use minimal nesting

`title` , `meta` or any other elements (e.g. `script`) need to be contained as **direct** children of the `Head` element, or wrapped into maximum one level of `<React.Fragment>` or arrays—otherwise the tags won't be correctly picked up on client-side navigations.

Use `next/script` for scripts

We recommend using `next/script` in your component instead of manually creating a `<script>` in `next/head`.

No `html` or `body` tags

You **cannot** use `<Head>` to set attributes on `<html>` or `<body>` tags. This will result in an `next-head-count is missing` error. `next/head` can only handle tags inside the HTML `<head>` tag.

Was this helpful?    

Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

Latest Version

15.5.4



Image

The Next.js Image component extends the HTML `` element for automatic image optimization.

app/page.js



```
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="/profile.png"
      width={500}
      height={500}
      alt="Picture of the author"
    />
  )
}
```

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.

Reference

Props

The following props are available:

Prop

Example

src

src="/profile.png"

alt

alt="Picture of the author"

width

width={500}

height

height={500}

fill

fill={true}

loader

loader={imageLoader}

sizes

sizes="(max-width: 768px) 100v
33vw"

quality

quality={80}

priority

priority={true}

placeholder

placeholder="blur"

style

style={{objectFit: "contain"}}

onLoadingComplete

onLoadingComplete={img =>
done()}

onLoad

onLoad={event => done()}

onError

onError(event => fail())

loading

loading="lazy"

blurDataURL

blurDataURL="data:image/jpeg .."

overrideSrc

overrideSrc="/seo.png"

src

The source of the image. Can be one of the following:

An internal path string.

```
<Image src="/profile.png" />
```

An absolute external URL (must be configured with [remotePatterns](#)).

```
<Image src="https://example.com/profile.png"
```

A static import.

```
import profile from './profile.png'

export default function Page() {
  return <Image src={profile} />
}
```

Good to know: For security reasons, the Image Optimization API using the default [loader](#) will *not* forward headers when fetching the `src` image. If the `src` image requires authentication, consider using the [unoptimized](#) property to disable Image Optimization.

`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 about [image accessibility guidelines](#).

width and height

The `width` and `height` properties represent the [intrinsic](#) image size in pixels. This property is used to infer the correct **aspect ratio** used by browsers to reserve space for the image and avoid layout shift during loading. It does not determine the *rendered size* of the image, which is controlled by CSS.

```
<Image src="/profile.png" width={500} height=
```

You **must** set both `width` and `height` properties unless:

- The image is statically imported.
- The image has the [fill](#) property

If the height and width are unknown, we recommend using the [fill](#) property.

fill

A boolean that causes the image to expand to the size of the parent element.

```
<Image src="/profile.png" fill={true} />
```

Positioning:

- The parent element **must** assign `position: "relative"`, `"fixed"`, `"absolute"`.
- By default, the `` element uses `position: "absolute"`.

Object Fit:

If no styles are applied to the image, the image will stretch to fit the container. You can use `objectFit` to control cropping and scaling.

- `"contain"` : The image will be scaled down to fit the container and preserve aspect ratio.
- `"cover"` : The image will fill the container and be cropped.

Learn more about [position](#) ↗ and [object-fit](#) ↗.

loader

A custom function used to generate the image URL. The function receives the following parameters, and returns a URL string for the image:

- `src`
- `width`
- `quality`

```
import Image from 'next/image'

const imageLoader = ({ src, width, quality }) =>
  return `https://example.com/${src}?w=${width}&q=${quality}`

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.

sizes

Define the sizes of the image at different breakpoints. Used by the browser to choose the most appropriate size from the generated `srcset`.

```
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) 500px, 300px"
      />
    </div>
  )
}
```

`sizes` should be used when:

- The image is using the `fill` prop
- CSS is used to make the image responsive

If `sizes` is missing, the browser assumes the image will be as wide as the viewport (`100vw`). This can cause unnecessarily large images to be downloaded.

In addition, `sizes` affects how `srcset` is generated:

- Without `sizes`: Next.js generates a limited `srcset` (e.g. `1x`, `2x`), suitable for fixed-size images.
- With `sizes`: Next.js generates a full `srcset` (e.g. `640w`, `750w`, etc.), optimized for responsive layouts.

Learn more about `srcset` and `sizes` on [web.dev](#) ↗ and [mdn](#) ↗.

quality

An integer between `1` and `100` that sets the quality of the optimized image. Higher values increase file size and visual fidelity. Lower values reduce file size but may affect sharpness.

```
// Default quality is 75
<Image quality={75} />
```

If you've configured `qualities` in `next.config.js`, the value must match one of the allowed entries.

Good to know: If the original image is already low quality, setting a high quality value will increase the file size without improving appearance.

style

Allows passing CSS styles to the underlying image element.

```
const imageStyle = {
  borderRadius: '50%',
  border: '1px solid #fff',
  width: '100px',
  height: 'auto',
}

export default function ProfileImage() {
  return <Image src="..." style={imageStyle} />
}
```

Good to know: If you're using the `style` prop to set a custom width, be sure to also set `height: 'auto'` to preserve the image's aspect ratio.

priority

A boolean that indicates if the image should be preloaded.

```
// Default priority is false  
<Image priority={false} />
```

- `true` : Preloads ↗ the image. Disables lazy loading.
- `false` : Lazy loads the image.

When to use it:

- The image is above the fold.
- The image is the [Largest Contentful Paint \(LCP\)](#) ↗ element.
- You want to improve the initial loading performance of your page.

When not to use it:

- When the `loading` prop is used (will trigger warnings).

loading

Controls when the image should start loading.

```
// Defaults to lazy  
<Image loading="lazy" />
```

- `lazy` : Defer loading the image until it reaches a calculated distance from the viewport.
- `eager` : Load the image immediately, regardless of its position in the page.

Use `eager` only when you want to ensure the image is loaded immediately.

Learn more about the [loading attribute ↗](#).

placeholder

Specifies a placeholder to use while the image is loading, improving the perceived loading performance.

```
// defaults to empty
<Image placeholder="empty" />
```

- `empty` : No placeholder while the image is loading.
- `blur` : Use a blurred version of the image as a placeholder. Must be used with the `blurDataURL` property.
- `data:image/ ...` : Uses the [Data URL ↗](#) as the placeholder.

Examples:

- [blur placeholder ↗](#)
- [Shimmer effect with data URL placeholder prop ↗](#)
- [Color effect with blurDataURL prop ↗](#)

Learn more about the [placeholder attribute ↗](#).

blurDataURL

A [Data URL ↗](#) to be used as a placeholder image before the image successfully loads. Can be automatically set or used with the `placeholder="blur"` property.

```
<Image placeholder="blur" blurDataURL="..." />
```

The image is automatically enlarged and blurred, so a very small image (10px or less) is recommended.

Automatic

If `src` is a static import of a `jpg`, `png`, `webp`, or `avif` file, `blurDataURL` is added automatically—unless the image is animated.

Manually set

If the image is dynamic or remote, you must provide `blurDataURL` yourself. To generate one, you can use:

- A online tool like [png-pixel.com](#) ↗
- A library like [Placeholder](#) ↗

A large `blurDataURL` may hurt performance. Keep it small and simple.

Examples:

- Default `blurDataURL` prop ↗
- Color effect with `blurDataURL` prop ↗

onLoad

A callback function that is invoked once the image is completely loaded and the `placeholder` has been removed.

```
<Image onLoad={(e) => console.log(e.target.na}
```

The callback function will be called with one argument, the event which has a `target` that references the underlying `` element.

onError

A callback function that is invoked if the image fails to load.

```
<Image onError={(e) => console.error(e.target)}
```

unoptimized

A boolean that indicates if the image should be optimized. This is useful for images that do not benefit from optimization such as small images (less than 1KB), vector images (SVG), or animated images (GIF).

```
import Image from 'next/image'

const UnoptimizedImage = (props) => {
  // Default is false
  return <Image {...props} unoptimized />
}
```

- `true` : The source image will be served as-is from the `src` instead of changing quality, size, or format.
- `false` : The source image will be optimized.

Since Next.js 12.3.0, this prop can be assigned to all images by updating `next.config.js` with the following configuration:

next.config.js

```
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 ``.

input.js

```
<Image src="/profile.jpg" />
```

output.html

```

```

decoding

A hint to the browser indicating if it should wait for the image to be decoded before presenting other content updates or not.

```
// Default is async  
<Image decoding="async" />
```

- `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. The browser chooses the best approach.

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.

Deprecated props

`onLoadingComplete`

Warning: Deprecated in Next.js 14, use [onLoad](#) instead.

A callback function that is invoked once the image is completely loaded and the [placeholder](#) has been removed.

The callback function will be called with one argument, a reference to the underlying `` element.

```
<Image onLoadingComplete={(img) => console.lo
```

Configuration options

You can configure the Image Component in `next.config.js`. The following options are available:

localPatterns

Use `localPatterns` in your `next.config.js` file to allow images from specific local paths to be optimized and block all others.

```
JS next.config.js Copy  
  
module.exports = {  
  images: {  
    localPatterns: [  
      {  
        pathname: '/assets/images/**',  
        search: '',  
      },  
    ],  
  },  
}
```

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` error.

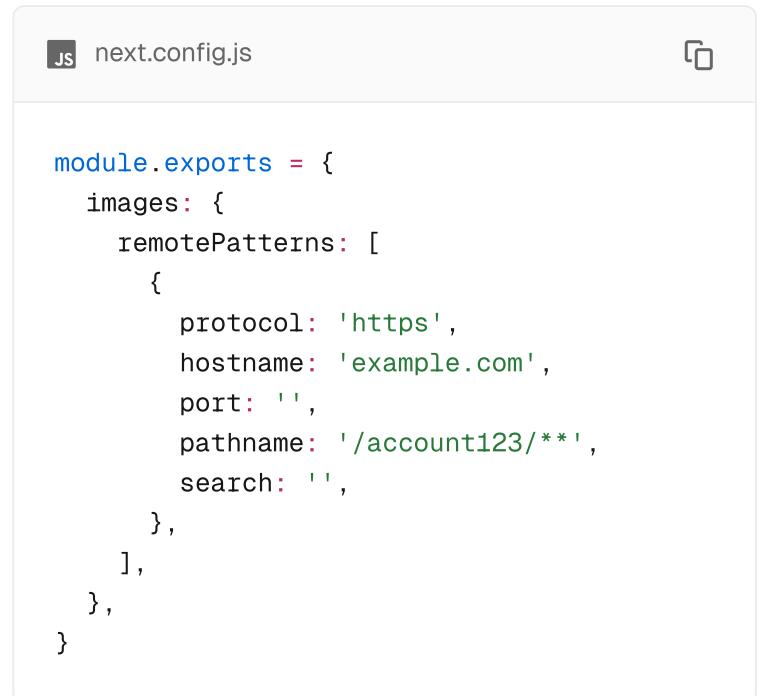
remotePatterns

Use `remotePatterns` in your `next.config.js` file to allow images from specific external paths and block all others. This ensures that only external images from your account can be served.

```
JS next.config.js Copy  
  
module.exports = {  
  images: {  
    remotePatterns: [  
      {  
        pathname: 'https://image.pollinations.ai/*',  
      },  
    ],  
  },  
}
```

```
module.exports = {
  images: {
    remotePatterns: [new URL('https://example
  },
}
```

You can also configure `remotePatterns` using the object:



```
next.config.js
```

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'example.com',
        port: '',
        pathname: '/account123/**',
        search: ''
      }
    ],
  },
}
```

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`.

Wildcard Patterns:

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. This syntax does not work in the middle of the pattern.



```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
        port: '',
        search: ''
      },
    ],
  },
}
```

This allows subdomains like `image.example.com`.
Query strings and custom ports are still blocked.

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.

Query Strings:

You can also restrict query strings using the `search` property:



```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'assets.example.com',
        search: '?v=1727111025337'
      },
    ],
  },
}
```

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.

loaderFile

`loaderFile` allows you to use a custom image optimization service instead of Next.js.

```
JS next.config.js

module.exports = {
  images: {
    loader: 'custom',
    loaderFile: './my/image/loader.js',
  },
}
```

The path must be relative to the project root. The file must export a default function that returns a URL string:

```
JS my/image/loader.js

export default function myImageLoader({ src,
  return `https://example.com/${src}?w=${width}`
})
```

Example:

- [Custom Image Loader Configuration](#)

Alternatively, you can use the `loader` prop to configure each instance of `next/image`.

path

If you want to change or prefix the default path for the Image Optimization API, you can do so with the `path` property. The default value for `path` is `/_next/image`.

```
module.exports = {
  images: {
    path: '/my-prefix/_next/image',
  },
}
```

deviceSizes

`deviceSizes` allows you to specify a list of device width breakpoints. These widths are used when the `next/image` component uses `sizes` prop to ensure the correct image is served for the user's device.

If no configuration is provided, the default below is used:

```
module.exports = {
  images: {
    deviceSizes: [640, 750, 828, 1080, 1200],
  },
}
```

imageSizes

`imageSizes` allows you to specify a list of image widths. These widths are concatenated with the array of `device sizes` to form the full array of sizes used to generate image [srcset ↗](#).

If no configuration is provided, the default below is used:

```
module.exports = {
  images: {
    imageSizes: [16, 32, 48, 64, 96, 128, 256],
  },
}
```

`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`.

qualities

`qualities` allows you to specify a list of image quality values.

```
JS next.config.js

module.exports = {
  images: {
    qualities: [25, 50, 75],
  },
}
```

In the example above, only three qualities are allowed: 25, 50, and 75. If the `quality` prop does not match a value in this array, the image will fail with a `400` Bad Request.

formats

`formats` allows you to specify a list of image formats to be used.

```
JS next.config.js

module.exports = {
  images: {
    // Default
    formats: ['image/webp'],
  },
}
```

Next.js automatically detects 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), it will use the original image's format.

You can enable AVIF support, which will fallback to the original format of the `src` image if the browser [does not support AVIF](#) :

`next.config.js`

```
module.exports = {
  images: {
    formats: ['image/avif'],
  },
}
```

Good to know:

- We still recommend using WebP for most use cases.
- 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, but 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.

`minimumCacheTTL`

`minimumCacheTTL` allows you to 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`.

If no configuration is provided, the default below is used.

```
module.exports = {
  images: {
    minimumCacheTTL: 60, // 1 minute
  },
}
```

You can increase the TTL to reduce the number of revalidations and potentially lower cost:

```
module.exports = {
  images: {
    minimumCacheTTL: 2678400, // 31 days
  },
}
```

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 the cached file `<distDir>/cache/images`.

`disableStaticImages`

`disableStaticImages` allows you to disable static image imports.

The default behavior allows you to import static files such as `import icon from './icon.png'` and then pass that to the `src` property. In some

cases, you may wish to disable this feature if it conflicts with other plugins that expect the import to behave differently.

You can disable static image imports inside your `next.config.js`:

```
JS next.config.js ✖  
  
module.exports = {  
  images: {  
    disableStaticImages: true,  
  },  
}
```

`dangerouslyAllowSVG`

`dangerouslyAllowSVG` allows you to serve SVG images.

```
JS next.config.js ✖  
  
module.exports = {  
  images: {  
    dangerouslyAllowSVG: true,  
  },  
}
```

By default, Next.js does not optimize SVG images for a few reasons:

- SVG is a vector format meaning it can be resized losslessly.
- SVG has many of the same features as HTML/CSS, which can lead to vulnerabilities without proper [Content Security Policy \(CSP\) headers](#).

We recommend using the `unoptimized` prop when the `src` prop is known to be SVG. This happens automatically when `src` ends with `".svg"`.

```
<Image src="/my-image.svg" unoptimized />
```

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.

`next.config.js`

```
module.exports = {
  images: {
    dangerouslyAllowSVG: true,
    contentDispositionType: 'attachment',
    contentSecurityPolicy: "default-src 'self'
  },
}
```

`contentDispositionType`

`contentDispositionType` allows you to configure the [Content-Disposition](#) ↗ header.

`next.config.js`

```
module.exports = {
  images: {
    contentDispositionType: 'inline',
  },
}
```

`contentSecurityPolicy`

`contentSecurityPolicy` allows you to configure the [Content-Security-Policy](#) ↗ header for images. This is particularly important when using [dangerouslyAllowSVG](#) to prevent scripts embedded in the image from executing.

`next.config.js`

```
module.exports = {
```

```
  images: {
    contentSecurityPolicy: "default-src 'self'
  },
}
```

By default, the `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.

Deprecated configuration options

`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:

`next.config.js`



```
module.exports = {
  images: {
    domains: ['assets.acme.com'],
  }
}
```

```
},  
}
```

Functions

getImageProps

The `getImageProps` function can be used to get the props that would be passed to the underlying `` element, and instead pass them to another component, style, canvas, etc.

```
import { getImageProps } from 'next/image'  
  
const { props } = getImageProps({  
  src: 'https://example.com/image.jpg',  
  alt: 'A scenic mountain view',  
  width: 1200,  
  height: 800,  
})  
  
function ImageWithCaption() {  
  return (  
    <figure>  
      <img {...props} />  
      <figcaption>A scenic mountain view</figcaption>  
    </figure>  
  )  
}
```

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.

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](#)

Examples

Styling images

Styling the Image component is similar to styling a normal `` element, but there are a few guidelines to keep in mind:

Use `className` or `style`, not `styled-jsx`. In most cases, we recommend using the `className` prop. This can be an imported [CSS Module](#), a [global stylesheet](#), etc.

```
import styles from './styles.module.css'

export default function MyImage() {
  return <Image className={styles.image} src=
}
```

You can also use the `style` prop to assign inline styles.

```
export default function MyImage() {
  return (
    <Image style={{ borderRadius: '8px' }} sr
  )
}
```

When using `fill`, the parent element must have `position: relative` or `display: block`. This is necessary for the proper rendering of the image element in that layout mode.

```
<div style={{ position: 'relative' }}>
  <Image fill src="/my-image.png" alt="My Ima
</div>
```

You cannot use `styled-jsx` because it's scoped to the current component (unless you mark the style as `global`).

Responsive images with a static export

When you import a static image, Next.js automatically sets its width and height based on the file. You can make the image responsive by setting the style:



```
import Image from 'next/image'  
import mountains from '../public/mountains.jp  
  
export default function Responsive() {  
  return (  
    <div style={{ display: 'flex', flexDirect  
      <Image  
        alt="Mountains"  
        // Importing an image will  
        // automatically set the width and he  
        src={mountains}  
        sizes="100vw"  
        // Make the image display full width  
        // and preserve its aspect ratio  
        style={{  
          width: '100%',  
          height: 'auto',  
        }}  
      />  
    </div>  
  )  
}
```

Responsive images with a remote URL

If the source image is a dynamic or a remote URL, you must provide the width and height props so Next.js can calculate the aspect ratio:

```
JS components/page.js  
  
import Image from 'next/image'  
  
export default function Page({ photoUrl }) {  
  return (  
    <Image  
      src={photoUrl}  
      alt="Picture of the author"  
      sizes="100vw"
```

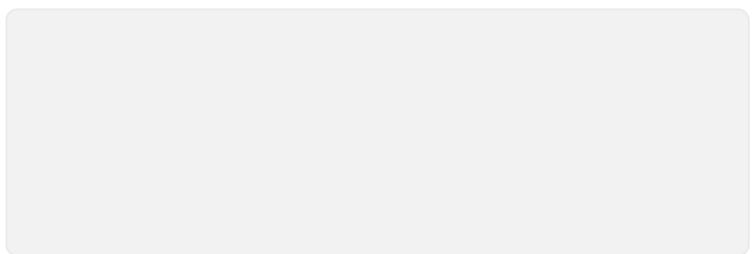
```
        style={{
          width: '100%',
          height: 'auto',
        }}
      width={500}
      height={300}
    />
  )
}
```

Try it out:

- Demo the image responsive to viewport ↗

Responsive image with `fill`

If you don't know the aspect ratio of the image, you can add the `fill` prop with the `objectFit` prop set to `cover`. This will make the image fill the full width of its parent container.



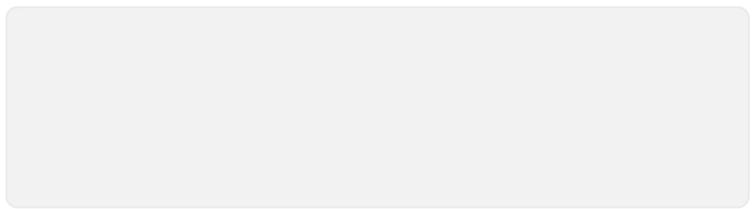
```
import Image from 'next/image'
import mountains from '../public/mountains.jp

export default function Fill() {
  return (
    <div
      style={{
        display: 'grid',
        gridGap: '8px',
        gridTemplateColumns: 'repeat(auto-fit
      )}
    >
      <div style={{ position: 'relative', wid
        <Image
          alt="Mountains"
          src={mountains}
          fill
          sizes="(min-width: 808px) 50vw, 100
          style={{
            objectFit: 'cover', // cover, con
```

```
        }})
      />
    </div>
    /* And more images in the grid... */
  </div>
)
}
```

Background Image

Use the `fill` prop to make the image cover the entire screen area:



```
import Image from 'next/image'
import mountains from '../public/mountains.jp

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 ↗](#).

Remote images

To use a remote image, the `src` property should be a URL string.



```
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="https://s3.amazonaws.com/my-bucket
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

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.

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/**',
        search: ''
      },
    ],
  }
}
```

```
 },  
 }
```

Theme detection

If you want to display a different image for light and dark mode, you can create a new component that wraps two `<Image>` components and reveals the correct one based on a CSS media query.

components/theme-image.module.css

```
.imgDark {  
  display: none;  
}  
  
@media (prefers-color-scheme: dark) {  
  .imgLight {  
    display: none;  
  }  
  .imgDark {  
    display: unset;  
  }  
}
```

components/theme-image.tsx TypeScript

```
import styles from './theme-image.module.css'  
import Image, { ImageProps } from 'next/image'  
  
type Props = Omit<ImageProps, 'src' | 'priori  
srcLight: string  
srcDark: string  
}  
  
const ThemeImage = (props: Props) => {  
  const { srcLight, srcDark, ...rest } = prop  
  
  return (  
    <>  
      <Image {...rest} src={srcLight} className="imgLight" />  
      <Image {...rest} src={srcDark} className="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 ↗](#)

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()`.



```
JS app/page.js

import { getImageProps } from 'next/image'

export default function Home() {
  const common = { alt: 'Art Direction Example' }
  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)" src={desktop} />
      <source media="(min-width: 500px)" src={mobile} />
      <img {...rest} style={{ width: '100%' }} alt={common.alt} />
    </picture>
  )
}
```

```
    </picture>
)
}
```

Background CSS

You can even convert the `srcSet` string to the [image-set\(\)](#) CSS function to optimize a background image.

```
JS app/page.js

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 })
  const backgroundImage = getBackgroundImage(srcSet)
  const style = { height: '100vh', width: '100%' }

  return (
    <main style={style}>
      <h1>Hello World</h1>
    </main>
  )
}
```

Version History

Version	Changes
v15.3.0	<code>remotePatterns</code> added support for array of <code>URL</code> objects.
v15.0.0	<code>contentDispositionType</code> configuration default changed to <code>attachment</code> .
v14.2.23	<code>qualities</code> configuration added.
v14.2.15	<code>decoding</code> prop added and <code>localPatterns</code> configuration added.
v14.2.14	<code>remotePatterns.search</code> prop added.
v14.2.0	<code>overrideSrc</code> prop added.
v14.1.0	<code>getImageProps()</code> is stable.
v14.0.0	<code>onLoadingComplete</code> prop and <code>domains</code> config deprecated.
v13.4.14	<code>placeholder</code> prop support for <code>data:/image ...</code>
v13.2.0	<code>contentDispositionType</code> configuration added.
v13.0.6	<code>ref</code> prop added.
v13.0.0	The <code>next/image</code> import was renamed to <code>next/legacy/image</code> . The <code>next/future/image</code> import was renamed to <code>next/image</code> . A codemod is available to safely and automatically rename your imports. <code></code> wrapper removed. <code>layout</code> , <code>objectFit</code> , <code>objectPosition</code> , <code>lazyBoundary</code> , <code>lazyRoot</code> props removed. <code>alt</code> is required. <code>onLoadingComplete</code> receives reference to <code>img</code> element. Built-in loader config removed.
v12.3.0	<code>remotePatterns</code> and <code>unoptimized</code> configuration is stable.
v12.2.0	Experimental <code>remotePatterns</code> and experimental <code>unoptimized</code> configuration added. <code>layout="raw"</code> removed.

Version Changes

v12.1.1	<code>style</code> prop added. Experimental support for <code>layout="raw"</code> added.
v12.1.0	<code>dangerouslyAllowSVG</code> and <code>contentSecurityPolicy</code> configuration added.
v12.0.9	<code>lazyRoot</code> prop added.
v12.0.0	<code>formats</code> configuration added. AVIF support added. Wrapper <code><div></code> changed to <code></code> .
v11.1.0	<code>onLoadingComplete</code> and <code>lazyBoundary</code> props added.
v11.0.0	<code>src</code> prop support for static import. <code>placeholder</code> prop added. <code>blurDataURL</code> prop added.
v10.0.5	<code>loader</code> prop added.
v10.0.1	<code>layout</code> prop added.
v10.0.0	next/image introduced.

Was this helpful?    

 Using Pages Router
Features available in /pages

 Latest Version
15.5.4

 You are currently viewing the documentation for Pages Router.

Image (Legacy)

 This is a legacy API and no longer recommended.
It's still supported for backward compatibility.

Starting with Next.js 13, the `next/image` component was rewritten to improve both the performance and developer experience. In order to provide a backwards compatible upgrade solution, the old `next/image` was renamed to `next/legacy/image`.

View the new [next/image API Reference](#)

Comparison

Compared to `next/legacy/image`, the new `next/image` component has the following changes:

- Removes `` wrapper around `` in favor of [native computed aspect ratio ↗](#)
- Adds support for canonical `style` prop
 - Removes `layout` prop in favor of `style` or `className`
 - Removes `objectFit` prop in favor of `style` or `className`

- Removes `objectPosition` prop in favor of `style` or `className`
 - Removes `IntersectionObserver` implementation in favor of [native lazy loading](#)
 - ↗
 - Removes `lazyBoundary` prop since there is no native equivalent
 - Removes `lazyRoot` prop since there is no native equivalent
 - Removes `loader` config in favor of `loader` prop
 - Changed `alt` prop from optional to required
 - Changed `onLoadingComplete` callback to receive reference to `` element
-

Required Props

The `<Image />` component requires the following properties.

src

Must be one of the following:

- A [statically imported](#) image file
- A path string. This can be either an absolute external URL, or an internal path depending on the `loader` prop or [loader configuration](#).

When using 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 can represent either the *rendered* width or *original* width in pixels, depending on the `layout` and `sizes` properties.

When using `layout="intrinsic"` or `layout="fixed"` the `width` property represents the *rendered* width in pixels, so it will affect how large the image appears.

When using `layout="responsive"`, `layout="fill"`, the `width` property represents the *original* width in pixels, so it will only affect the aspect ratio.

The `width` property is required, except for [statically imported images](#), or those with `layout="fill"`.

height

The `height` property can represent either the *rendered* height or *original* height in pixels, depending on the `layout` and `sizes` properties.

When using `layout="intrinsic"` or `layout="fixed"` the `height` property represents the *rendered* height in pixels, so it will affect how large the image appears.

When using `layout="responsive"`, `layout="fill"`, the `height` property represents the *original* height in pixels, so it will only affect the aspect ratio.

The `height` property is required, except for statically imported images, or those with `layout="fill"`.

Optional Props

The `<Image />` component accepts a number of additional properties beyond those which are required. This section describes the most commonly-used properties of the Image component. Find details about more rarely-used properties in the [Advanced Props](#) section.

layout

The layout behavior of the image as the viewport changes size.

Has wrap and	layout	Behavior	srcSet	sizes	sizer
	intrinsic (default)	Scale <i>down to</i> fit width of container, up to image size	1x , 2x (based on <code>imageSizes</code>)	N/A	yes
	fixed	Sized to <code>width</code> and <code>height</code> exactly	1x , 2x (based on <code>imageSizes</code>)	N/A	yes
	responsive	Scale to fit width	640w , 750w , ...	100vw	yes

Has
wrapp
and

layout	Behavior	srcSet	sizes	sizer
of container	(based on imageSizes and deviceSizes)	2048w , 3840w		
fill	(based on container)	640w , 750w , ... 2048w , 3840w	100vw	yes

- Demo the [intrinsic](#) layout (default) ↗
 - When `intrinsic`, the image will scale the dimensions down for smaller viewports, but maintain the original dimensions for larger viewports.
- Demo the [fixed](#) layout ↗
 - When `fixed`, the image dimensions will not change as the viewport changes (no responsiveness) similar to the native `img` element.
- Demo the [responsive](#) layout ↗
 - When `responsive`, the image will scale the dimensions down for smaller viewports and scale up for larger viewports.
 - Ensure the parent element uses `display: block` in their stylesheet.
- Demo the [fill](#) layout ↗

- When `fill`, the image will stretch both width and height to the dimensions of the parent element, provided the parent element is relative.
- This is usually paired with the `objectFit` property.
- Ensure the parent element has `position: relative` in their stylesheet.
- [Demo background image ↗](#)

loader

A custom function used to resolve URLs. Setting the loader as a prop on the Image component overrides the default loader defined in the [images section of `next.config.js`](#).

A `loader` is a function returning a URL string for the image, given the following parameters:

- `src`
- `width`
- `quality`

Here is an example of using a custom loader:

```
import Image from 'next/legacy/image'

const myLoader = ({ src, width, quality }) =>
  return `https://example.com/${src}?w=${width}`

const MyImage = (props) => {
  return (
    <Image
      loader={myLoader}
      src="me.png"
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

sizes

A string that provides information about how wide the image will be at different breakpoints. The value of `sizes` will greatly affect performance for images using `layout="responsive"` or `layout="fill"`. It will be ignored for images using `layout="intrinsic"` or `layout="fixed"`.

The `sizes` property serves two important purposes related to image performance:

First, the value of `sizes` is used by the browser to determine which size of the image to download, from `next/legacy/image`'s automatically-generated source set. When the browser chooses, it does not yet know the size of the image on the page, so it selects an image that is the same size or larger than the viewport. The `sizes` property allows you to tell the browser that the image will actually be smaller than full screen. If you don't specify a `sizes` value, a default value of `100vw` (full screen width) is used.

Second, the `sizes` value is parsed and used to trim the values in the automatically-created source set. If the `sizes` property includes sizes such as `50vw`, which represent a percentage of the viewport width, then the source set is trimmed to not include any values which are too small to ever be necessary.

For example, if you know your styling will cause an image to be full-width on mobile devices, in a 2-column layout on tablets, and a 3-column layout on desktop displays, you should include a `sizes` property such as the following:

```
import Image from 'next/legacy/image'
const Example = () => (
  <div className="grid-element">
    <Image
      src="/example.png"
      layout="fill"
      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

The quality of the optimized image, an integer between `1` and `100` where `100` is the best quality. Defaults to `75`.

priority

When true, the image will be considered high priority and [preload ↗](#). Lazy loading is automatically disabled for images using `priority`.

You should use the `priority` property on any image detected as the [Largest Contentful Paint \(LCP\) ↗](#) element. It may be appropriate to have multiple priority images, as different images may be the LCP element for different viewport sizes.

Should only be used when the image is visible above the fold. Defaults to `false`.

placeholder

A placeholder to use while the image is loading. Possible values are `blur` or `empty`. Defaults to `empty`.

When `blur`, the [blurDataURL](#) property will be used as the placeholder. If `src` is an object from a [static import](#) and the imported image is `.jpg`, `.png`, `.webp`, or `.avif`, then `blurDataURL` will be automatically populated.

For dynamic images, you must provide the [blurDataURL](#) property. Solutions such as [Placeholder ↗](#) can help with `base64` generation.

When `empty`, there will be no placeholder while the image is loading, only empty space.

Try it out:

- Demo the [blur placeholder ↗](#)

- Demo the shimmer effect with `blurDataURL` prop ↗
 - Demo the color effect with `blurDataURL` prop ↗
-

Advanced Props

In some cases, you may need more advanced usage. The `<Image />` component optionally accepts the following advanced properties.

style

Allows [passing CSS styles ↗](#) to the underlying image element.

Note that all `layout` modes apply their own styles to the image element, and these automatic styles take precedence over the `style` prop.

Also keep in mind that the required `width` and `height` props can interact with your styling. If you use styling to modify an image's `width`, you must set the `height="auto"` style as well, or your image will be distorted.

objectFit

Defines how the image will fit into its parent container when using `layout="fill"`.

This value is passed to the [object-fit CSS property ↗](#) for the `src` image.

objectPosition

Defines how the image is positioned within its parent element when using `layout="fill"`.

This value is passed to the [object-position CSS property](#) applied to the image.

onLoadingComplete

A callback function that is invoked once the image is completely loaded and the [placeholder](#) has been removed.

The `onLoadingComplete` function accepts one parameter, an object with the following properties:

- [naturalWidth](#)
- [naturalHeight](#)

loading

The loading behavior of the image. Defaults to `lazy`.

When `lazy`, defer loading the image until it reaches a calculated distance from the viewport.

When `eager`, load the image immediately.

[Learn more](#)

blurDataURL

A [Data URL](#) to be used as a placeholder image before the `src` image successfully loads. Only takes effect when combined with `placeholder="blur"`.

Must be a base64-encoded image. It will be enlarged and blurred, so a very small image (10px or less) is recommended. Including larger images as placeholders may harm your application performance.

Try it out:

- Demo the default `blurDataURL` prop ↗
- Demo the shimmer effect with `blurDataURL` prop ↗
- Demo the color effect with `blurDataURL` prop ↗

You can also [generate a solid color Data URL ↗](#) to match the image.

lazyBoundary

A string (with similar syntax to the margin property) that acts as the bounding box used to detect the intersection of the viewport with the image and trigger lazy [loading](#). Defaults to `"200px"`.

If the image is nested in a scrollable parent element other than the root document, you will also need to assign the `lazyRoot` prop.

[Learn more ↗](#)

lazyRoot

A React [Ref ↗](#) pointing to the scrollable parent element. Defaults to `null` (the document viewport).

The Ref must point to a DOM element or a React component that [forwards the Ref ↗](#) to the underlying DOM element.

Example pointing to a DOM element

```
import Image from 'next/legacy/image'
import React from 'react'

const Example = () => {
  const lazyRoot = React.useRef(null)

  return (
    <Image alt="A placeholder image" ref={lazyRoot} />
  )
}

export default Example
```

```
<div ref={lazyRoot} style={{ overflowX: 'scroll' }}>
  <Image lazyRoot={lazyRoot} src="/one.jpg" />
  <Image lazyRoot={lazyRoot} src="/two.jpg" />
</div>
)
```

Example pointing to a React component

```
import Image from 'next/legacy/image'
import React from 'react'

const Container = React.forwardRef((props, ref) => {
  return (
    <div ref={ref} style={{ overflowX: 'scroll' }}>
      {props.children}
    </div>
  )
})

const Example = () => {
  const lazyRoot = React.useRef(null)

  return (
    <Container ref={lazyRoot}>
      <Image lazyRoot={lazyRoot} src="/one.jpg" />
      <Image lazyRoot={lazyRoot} src="/two.jpg" />
    </Container>
  )
}
```

[Learn more ↗](#)

unoptimized

When true, the source image will be served as-is from the `src` instead of changing quality, size, or format. Defaults to `false`.

This is useful for images that do not benefit from optimization such as small images (less than 1KB), vector images (SVG), or animated images (GIF).

```
import Image from 'next/image'

const UnoptimizedImage = (props) => {
  return <Image {...props} unoptimized />
```

}

Since Next.js 12.3.0, this prop can be assigned to all images by updating `next.config.js` with the following configuration:



```
JS next.config.js

module.exports = {
  images: {
    unoptimized: true,
  },
}
```

Other Props

Other properties on the `<Image />` component will be passed to the underlying `img` element with the exception of the following:

- `srcSet`. Use [Device Sizes](#) instead.
- `ref`. Use [onLoadingComplete](#) instead.
- `decoding`. It is always `"async"`.

Configuration Options

Remote Patterns

To protect your application from malicious users, configuration is required in order to use external images. This ensures that only external images from your account can be served from the Next.js Image Optimization API. These external images can be configured with the `remotePatterns`

property in your `next.config.js` file, as shown below:

```
JS next.config.js

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/legacy/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`:

```
JS next.config.js

module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
        port: '',
        search: ''
      },
    ],
  },
}
```

Good to know: The example above will ensure the `src` property of `next/legacy/image` must start with `https://img1.example.com` or `https://me.avatar.example.com` or any number of subdomains. 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`:

```
JS next.config.js ✖  
  
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/legacy/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:

next.config.js

```
module.exports = {
  images: {
    domains: ['assets.acme.com'],
  },
}
```

Loader Configuration

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure the `loader` and `path` prefix in your `next.config.js` file. This allows you to use relative URLs for the Image `src`

and automatically generate the correct absolute URL for your provider.

```
JS next.config.js
```

```
module.exports = {
  images: {
    loader: 'imgix',
    path: 'https://example.com/myaccount/',
  },
}
```

Customizing the Built-in Image Path

If you want to change or prefix the default path for the built-in Next.js image optimization, you can do so with the `path` property. The default value for `path` is `/_next/image`.

```
JS next.config.js
```

```
module.exports = {
  images: {
    path: '/my-prefix/_next/image',
  },
}
```

Built-in Loaders

The following Image Optimization cloud providers are included:

- Default: Works automatically with `next dev`, `next start`, or a custom server
- [Vercel ↗](#): Works automatically when you deploy on Vercel, no configuration necessary. [Learn more ↗](#)
- [Imgix ↗](#): `loader: 'imgix'`
- [Cloudinary ↗](#): `loader: 'cloudinary'`
- [Akamai ↗](#): `loader: 'akamai'`

- Custom: `loader: 'custom'` use a custom cloud provider by implementing the `loader` prop on the `next/legacy/image` component

If you need a different provider, you can use the `loader` prop with `next/legacy/image`.

Images can not be optimized at build time using `output: 'export'`, only on-demand. To use `next/legacy/image` with `output: 'export'`, you will need to use a different loader than the default. [Read more in the discussion.](#)

Advanced

The following configuration is for advanced use cases and is usually not necessary. If you choose to configure the properties below, you will override any changes to the Next.js defaults in future updates.

Device Sizes

If you know the expected device widths of your users, you can specify a list of device width breakpoints using the `deviceSizes` property in `next.config.js`. These widths are used when the `next/legacy/image` component uses `layout="responsive"` or `layout="fill"` to ensure the correct image is served for user's device.

If no configuration is provided, the default below is used.

`next.config.js`



```
module.exports = {
  images: {
```

```
    deviceSizes: [640, 750, 828, 1080, 1200],  
},  
}
```

Image Sizes

You can specify a list of image widths using the `images.imageSizes` property in your `next.config.js` file. These widths are concatenated with the array of `device sizes` to form the full array of sizes used to generate image `srcset` s.

The reason there are two separate lists is that `imageSizes` is only used for images which provide a `sizes` prop, which indicates that the image is less than the full width of the screen. **Therefore, the sizes in `imageSizes` should all be smaller than the smallest size in `deviceSizes`.**

If no configuration is provided, the default below is used.

```
js next.config.js  
  
module.exports = {  
  images: {  
    imageSizes: [16, 32, 48, 64, 96, 128, 256]  
  },  
}
```

Acceptable 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.

```
JS next.config.js Copy  
  
module.exports = {  
  images: {  
    formats: ['image/webp'],  
  },  
}
```

You can enable AVIF support, which will fallback to the original format of the src image if the browser [does not support AVIF ↗](#):

```
JS next.config.js Copy  
  
module.exports = {  
  images: {  
    formats: ['image/avif'],  
  },  
}
```

Good to know:

- We still recommend using WebP for most use cases.
- 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` (`x-vercel-cache` when deployed on Vercel) response header. The possible values are the following:

- `MISS` - the path is not in the cache (occurs at most once, on the first visit)
- `STALE` - the path is in the cache but exceeded the revalidate time so it will be updated in the background
- `HIT` - the path is in the cache and has not exceeded the revalidate time

The expiration (or rather Max Age) is defined by either the `minimumCacheTTL` configuration or the upstream image `Cache-Control` header, whichever is larger. Specifically, the `max-age` value of the `Cache-Control` header is used. If both `s-maxage` and `max-age` are found, then `s-maxage` is preferred. The `max-age` is also passed-through to any downstream clients including CDNs and browsers.

- You can configure `minimumCacheTTL` to increase the cache duration when the upstream image does not include `Cache-Control` header or the value is very low.
- You can configure `deviceSizes` and `imageSizes` to reduce the total number of possible generated images.
- You can configure `formats` to disable multiple formats in favor of a single image format.

Minimum Cache TTL

You can configure the Time to Live (TTL) in seconds for cached optimized images. In many cases, it's better to use a [Static Image Import](#) which will automatically hash the file contents and cache the image forever with a `Cache-Control` header of `immutable`.

If no configuration is provided, the default below is used.

```
JS next.config.js

module.exports = {
  images: {
    minimumCacheTTL: 60, // 1 minute
  },
}
```

You can increase the TTL to reduce the number of revalidations and potentially lower cost:

```
JS next.config.js

module.exports = {
  images: {
    minimumCacheTTL: 2678400, // 31 days
  },
}
```

The expiration (or rather Max Age) of the optimized image is defined by either the `minimumCacheTTL` or the upstream image `Cache-Control` header, whichever is larger.

If you need to change the caching behavior per image, you can configure `headers` to set the `Cache-Control` header on the upstream image (e.g. `/some-asset.jpg`, not `/_next/image` itself).

There is no mechanism to invalidate the cache at this time, so its best to keep `minimumCacheTTL` low. Otherwise you may need to manually change the `src` prop or delete `<distDir>/cache/images`.

Disable Static Imports

The default behavior allows you to import static files such as `import icon from './icon.png'` and then pass that to the `src` property.

In some cases, you may wish to disable this feature if it conflicts with other plugins that expect the import to behave differently.

You can disable static image imports inside your `next.config.js`:

```
JS next.config.js ✖  
  
module.exports = {  
  images: {  
    disableStaticImages: true,  
  },  
}
```

Dangerously Allow SVG

The default `loader` does not optimize SVG images for a few reasons. First, SVG is a vector format meaning it can be resized losslessly. Second, SVG

has many of the same features as HTML/CSS, which can lead to vulnerabilities without proper [Content Security Policy \(CSP\) headers](#).

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`:

```
JS next.config.js
```

```
module.exports = {
  images: {
    dangerouslyAllowSVG: true,
    contentDispositionType: 'attachment',
    contentSecurityPolicy: "default-src 'self'
  },
}
```

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.



```
next.config.js
```

```
module.exports = {
  images: {
    contentDispositionType: 'inline',
  },
}
```

Animated Images

The default `loader` will automatically bypass Image Optimization for animated images and serve the image as-is.

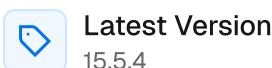
Auto-detection for animated files is best-effort and supports GIF, APNG, and WebP. If you want to explicitly bypass Image Optimization for a given animated image, use the `unoptimized` prop.

Version History

Version	Changes
---------	---------

v13.0.0	next/image renamed to next/legacy/image
---------	---

Was this helpful?    



ⓘ You are currently viewing the documentation for Pages Router.

Link

`<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:

```
TS pages/index.tsx TypeScript ▾   
  
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:

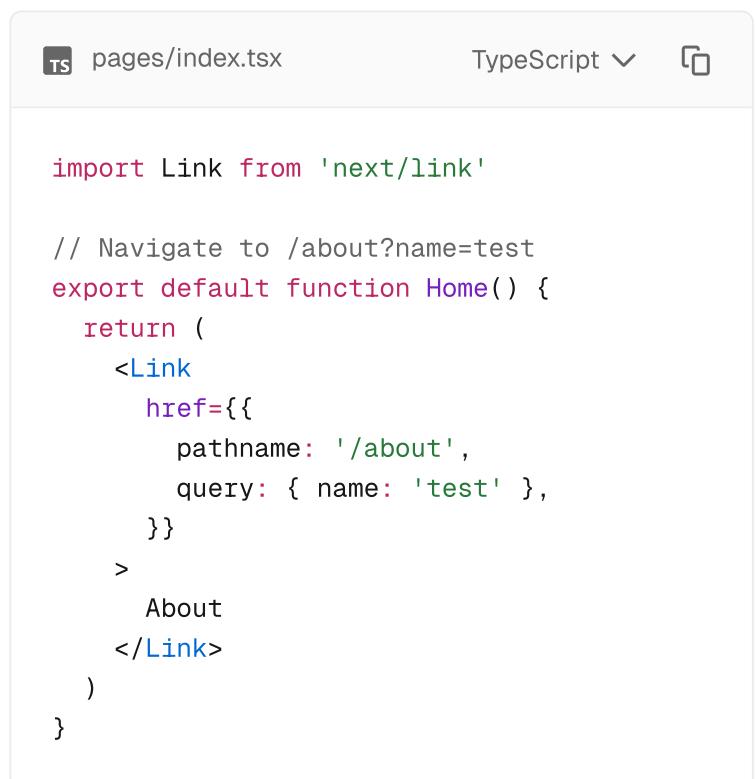
Prop	Example	Type	Req
<code>href</code>	<code>href="/dashboard"</code>	String or Object	Yes
<code>as</code>	<code>as="/post/abc"</code>	String or Object	-

Prop	Example	Type	Req
<code>replace</code>	<code>replace={false}</code>	Boolean	-
<code>scroll</code>	<code>scroll={false}</code>	Boolean	-
<code>prefetch</code>	<code>prefetch={false}</code>	Boolean	-
<code>legacyBehavior</code>	<code>legacyBehavior={true}</code>	Boolean	-
<code>passHref</code>	<code>passHref={true}</code>	Boolean	-
<code>shallow</code>	<code>shallow={false}</code>	Boolean	-
<code>locale</code>	<code>locale="fr"</code>	String or Boolean	-
<code>onNavigate</code>	<code>onNavigate={(e) => {}}</code>	Function	-

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.



pages/index.tsx

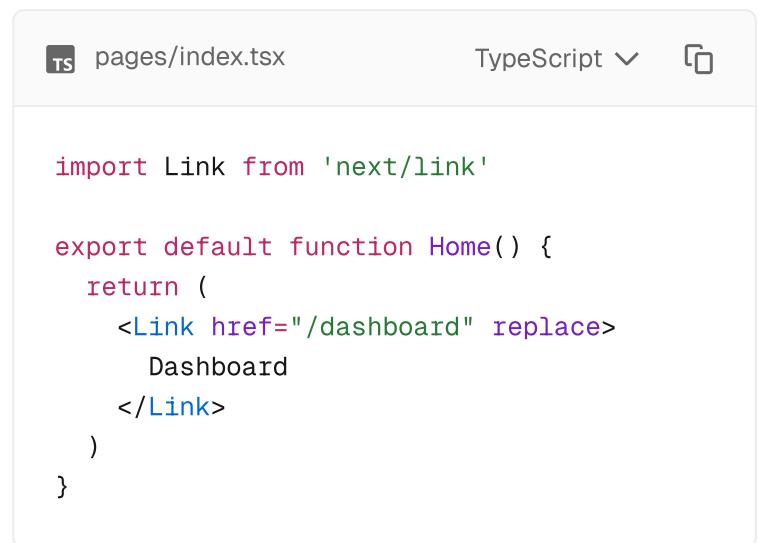
TypeScript

```
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.



```
TS pages/index.tsx TypeScript ▾
```

```
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 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 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>
  )
}
```

```
)  
}
```

legacyBehavior

Warning: The `legacyBehavior` prop will be removed in Next.js v16. To adopt the new `<Link>` behavior, remove any `<a>` tags used as children of `<Link>`. A [codemod is available](#) to help you automatically upgrade your codebase.

Since version 13, an `<a>` element is no longer required as a child of the `<Link>` component. If you still need the old behavior for compatibility reasons, you can add the `legacyBehavior` prop.

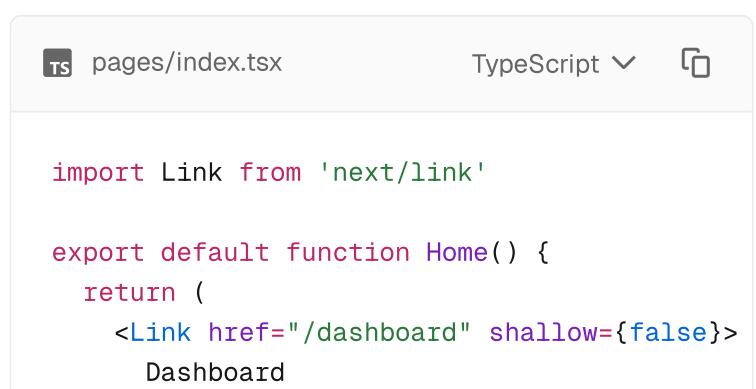
Good to know: when `legacyBehavior` is not set to `true`, all `anchor` tag properties can be passed to `next/link` as well such as, `className`, `onClick`, etc.

passHref

Forces `Link` to send the `href` property to its child. Defaults to `false`. See the [passing a functional component](#) example for more information.

shallow

Update the path of the current page without rerunning `getStaticProps`, `getServerSideProps` or `getInitialProps`. Defaults to `false`.



```
TS pages/index.tsx TypeScript ⓘ  
  
import Link from 'next/link'  
  
export default function Home() {  
  return (  
    <Link href="/dashboard" shallow={false}>  
      Dashboard  
    </Link>  
  )  
}
```

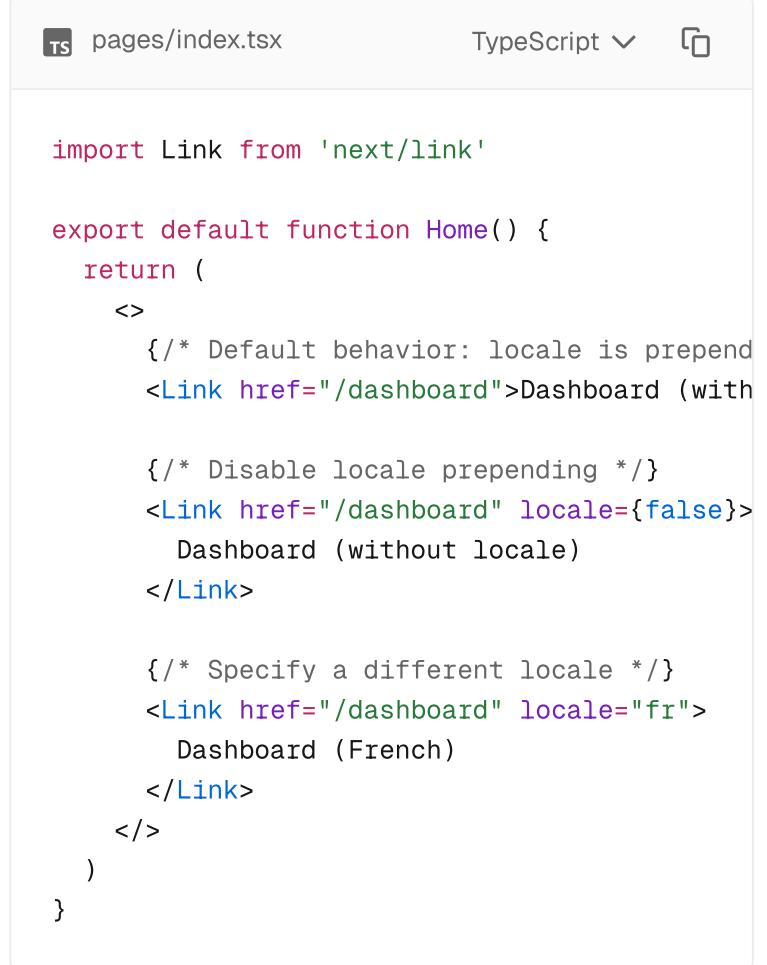
```
</Link>
)
}
```

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.



```
TS pages/index.tsx TypeScript ▾
```

```
import Link from 'next/link'

export default function Home() {
  return (
    <>
      /* Default behavior: locale is prepend
      <Link href="/dashboard">Dashboard (with

      /* Disable locale prepending */
      <Link href="/dashboard" locale={false}>
        Dashboard (without locale)
      </Link>

      /* Specify a different locale */
      <Link href="/dashboard" locale="fr">
        Dashboard (French)
      </Link>
    </>
  )
}
```

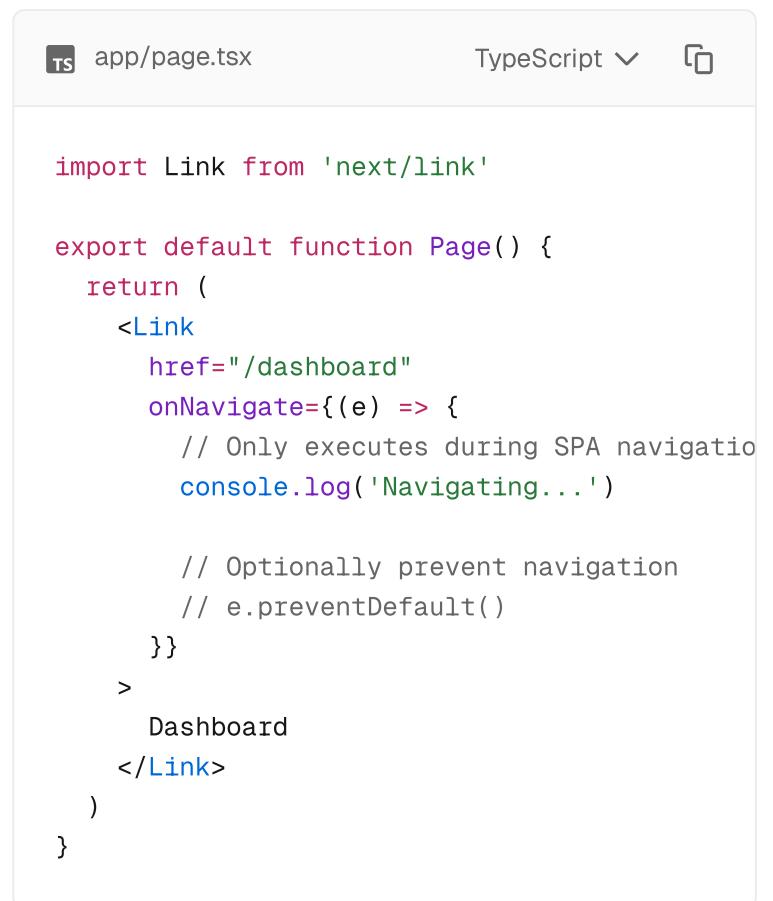
as

Optional decorator for the path that will be shown in the browser URL bar. Before Next.js 9.5.3 this was used for dynamic routes, check our [previous docs ↗](#) to see how it worked.

When this path differs from the one provided in `href` the previous `href / as` behavior is used as shown in the [previous docs ↗](#).

onNavigate

An event handler called during client-side navigation. The handler receives an event object that includes a `preventDefault()` method, allowing you to cancel the navigation if needed.



```
TS app/page.tsx TypeScript ▾
```

```
import Link from 'next/link'

export default function Page() {
  return (
    <Link
      href="/dashboard"
      onNavigate={(e) => {
        // Only executes during SPA navigation
        console.log('Navigating...')

        // Optionally prevent navigation
        // e.preventDefault()
      }}
    >
      Dashboard
    </Link>
  )
}
```

Good to know: While `onClick` and `onNavigate` may seem similar, they serve different purposes. `onClick` executes for all click events, while `onNavigate` only runs during client-side navigation. Some key differences:

- When using modifier keys (Ctrl / Cmd + Click), `onClick` executes but `onNavigate` doesn't since Next.js prevents default navigation for new tabs.
- External URLs won't trigger `onNavigate` since it's only for client-side and same-origin navigations.
- Links with the `download` attribute will work with `onClick` but not `onNavigate` since the browser will treat the linked URL as a download.

Examples

The following examples demonstrate how to use the `<Link>` component in different scenarios.

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`



A screenshot of a code editor window titled "pages/blog/index.tsx". The file contains the following TypeScript code:

```
import Link from 'next/link'

function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={`/blog/${post.slug}`}>
            </li>
      )))
    </ul>
  )
}
```

Scrolling to an `id`

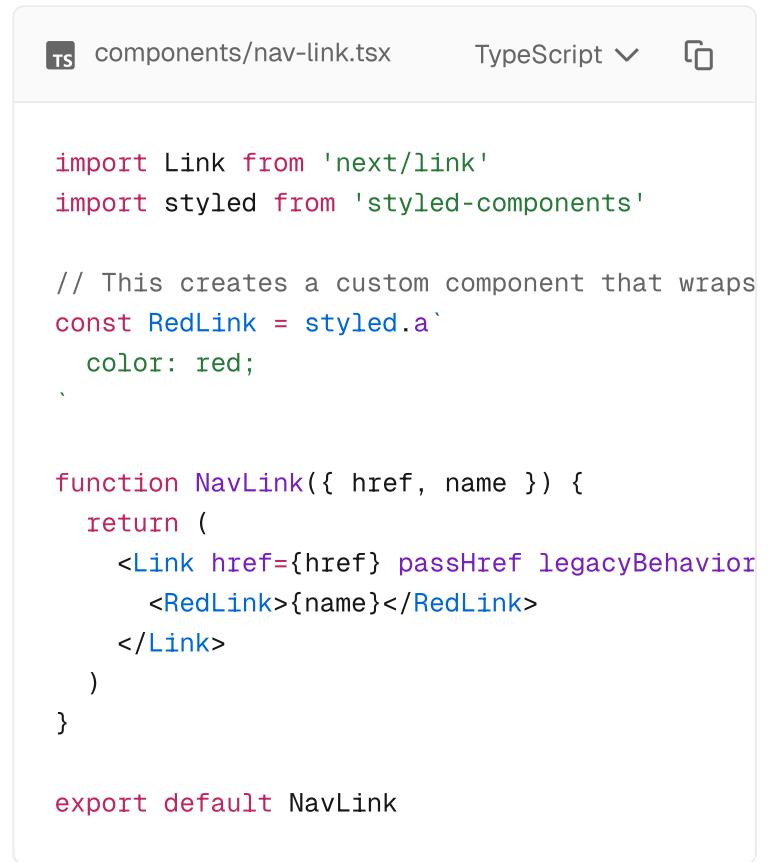
If you'd like to scroll to a specific `id` on navigation, you can append your URL with a `#` hash link or just pass a hash link to the `href` prop. This is possible since `<Link>` renders to an `<a>` element.

```
<Link href="/dashboard#settings">Settings</Link>

// Output
<a href="/dashboard#settings">Settings</a>
```

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`.



```
TS components/nav-link.tsx TypeScript ▾
```

```
import Link from 'next/link'
import styled from 'styled-components'

// This creates a custom component that wraps
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 prop
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>
  )
}

```

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
```

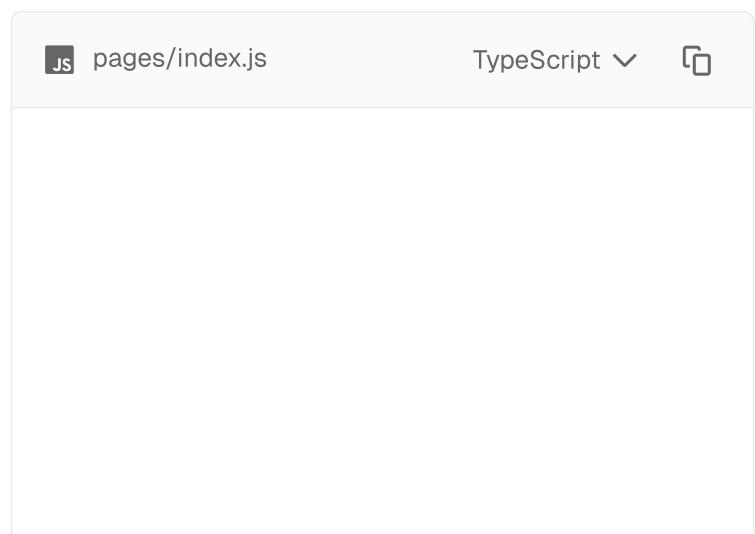
The above example has a link to:

- 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:



A screenshot of a code editor interface. At the top, there's a toolbar with a JS icon, the file name "pages/index.js", a TypeScript dropdown menu, and a close button. The main area of the editor is currently empty, showing only the header bar.

```
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 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`:



A screenshot of a code editor showing a file named "pages/index.tsx". The code is identical to the one above, but the last line has been modified:

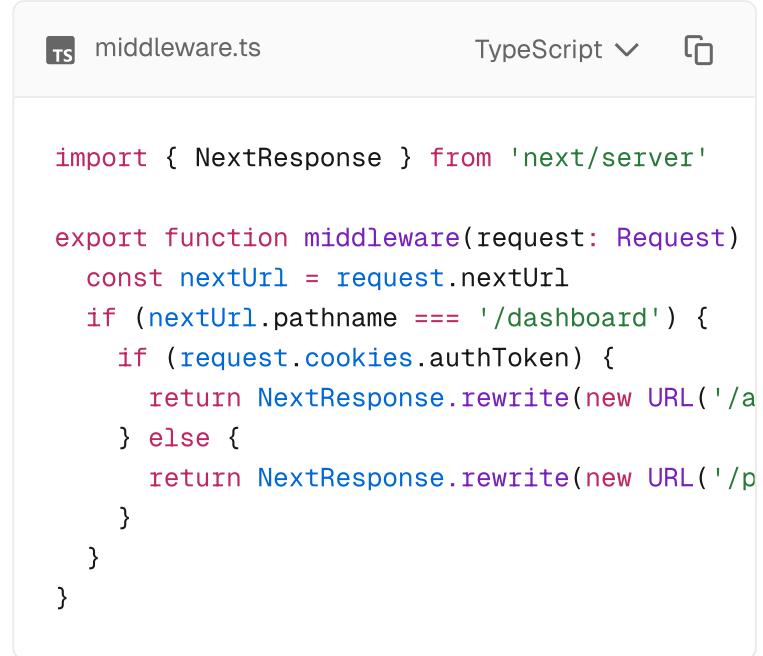
```
import Link from 'next/link'

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:

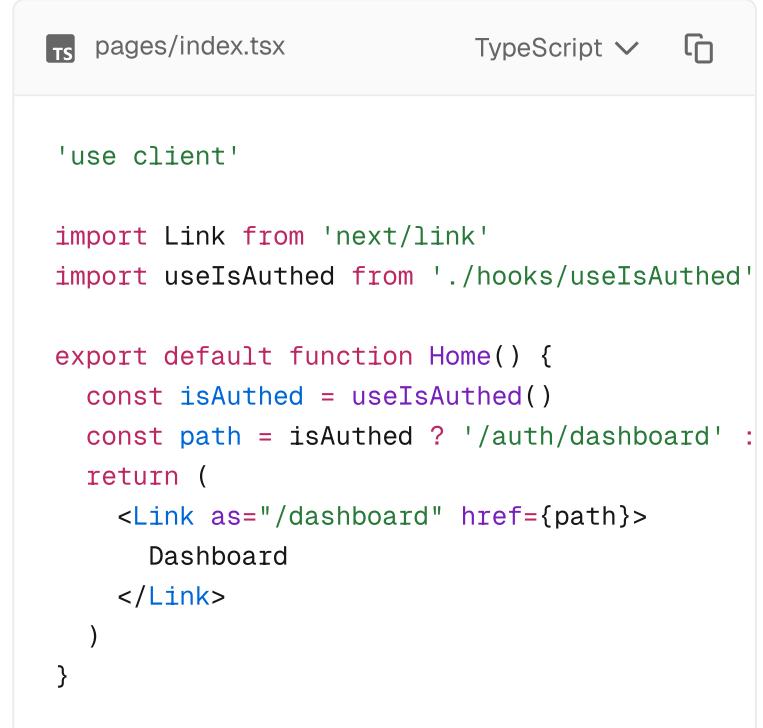


The screenshot shows a code editor window with the file name `middleware.ts` and a TypeScript icon. The code is a Next.js middleware function:

```
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', nextUrl))
    } else {
      return NextResponse.rewrite(new URL('/public/dashboard', nextUrl))
    }
  }
}
```

In this case, you would want to use the following code in your `<Link />` component:



The screenshot shows a code editor window with the file name `pages/index.tsx` and a TypeScript icon. The code defines a `Home` component:

```
'use client'

import Link from 'next/link'
import useIsAuthed from './hooks/useIsAuthed'

export default function Home() {
  const isAuthenticated = useIsAuthed()
  const path = isAuthenticated ? '/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
v15.4.0	Add <code>auto</code> as an alias to the default <code>prefetch</code> behavior.
v15.3.0	Add <code>onNavigate</code> API
v13.0.0	No longer requires a child <code><a></code> tag. A <code>codemod</code> is provided to automatically update your codebase.
v10.0.0	<code>href</code> props pointing to a dynamic route are automatically resolved and no longer require an <code>as</code> prop.
v8.0.0	Improved prefetching performance.
v1.0.0	<code>next/link</code> introduced.

Was this helpful?    



(i) You are currently viewing the documentation for Pages Router.

Script

This API reference will help you understand how to use `props` available for the Script Component. For features and usage, please see the [Optimizing Scripts](#) page.

```
TS app/dashboard/page.tsx TypeScript ▾ ⌂  
  
import Script from 'next/script'  
  
export default function Dashboard() {  
  return (  
    <>  
      <Script src="https://example.com/script"  
      </>  
    )  
}
```

Props

Here's a summary of the props available for the Script Component:

Prop	Example	Type
<code>src</code>	<code>src="http://example.com/script"</code>	String

Prop	Example	Type
------	---------	------

strategy	strategy="lazyOnload"	String
onLoad	onLoad={onLoadFunc}	Funct
onReady	onReady={onReadyFunc}	Funct
onError	onError={onErrorFunc}	Funct

Required Props

The `<Script />` component requires the following properties.

src

A path string specifying the URL of an external script. This can be either an absolute external URL or an internal path. The `src` property is required unless an inline script is used.

Optional Props

The `<Script />` component accepts a number of additional properties beyond those which are required.

strategy

The loading strategy of the script. There are four different strategies that can be used:

- `beforeInteractive` : Load before any Next.js code and before any page hydration occurs.
- `afterInteractive` : (**default**) Load early but after some hydration on the page occurs.
- `lazyOnload` : Load during browser idle time.
- `worker` : (experimental) Load in a web worker.

beforeInteractive

Scripts that load with the `beforeInteractive` strategy are injected into the initial HTML from the server, downloaded before any Next.js module, and executed in the order they are placed.

Scripts denoted with this strategy are preloaded and fetched before any first-party code, but their execution **does not block page hydration from occurring**.

`beforeInteractive` scripts must be placed inside the `Document` Component (`pages/_document.js`) and are designed to load scripts that are needed by the entire site (i.e. the script will load when any page in the application has been loaded server-side).

This strategy should only be used for critical scripts that need to be fetched as soon as possible.

```
JS pages/_document.js Copy  
  
import { Html, Head, Main, NextScript } from 'next/script'  
  
export default function Document() {  
  return (  
    <Html>  
      <Head />  
      <body>  
        <Main />  
        <NextScript />  
      </body>  
    </Html>  
  )  
}
```

```
<Script  
  src="https://example.com/script.js"  
  strategy="beforeInteractive"  
/>  
</body>  
</Html>  
)  
}
```

Good to know: Scripts with `beforeInteractive` will always be injected inside the `head` of the HTML document regardless of where it's placed in the component.

Some examples of scripts that should be fetched as soon as possible with `beforeInteractive` include:

- Bot detectors
- Cookie consent managers

afterInteractive

Scripts that use the `afterInteractive` strategy are injected into the HTML client-side and will load after some (or all) hydration occurs on the page.

This is the default strategy of the Script component and should be used for any script that needs to load as soon as possible but not before any first-party Next.js code.

`afterInteractive` scripts can be placed inside of any page or layout and will only load and execute when that page (or group of pages) is opened in the browser.

js app/page.js



```
import Script from 'next/script'  
  
export default function Page() {  
  return (  
    <>
```

```
<Script src="https://example.com/script
  </>
)
}
```

Some examples of scripts that are good candidates for `afterInteractive` include:

- Tag managers
- Analytics

lazyOnload

Scripts that use the `lazyOnload` strategy are injected into the HTML client-side during browser idle time and will load after all resources on the page have been fetched. This strategy should be used for any background or low priority scripts that do not need to load early.

`lazyOnload` scripts can be placed inside of any page or layout and will only load and execute when that page (or group of pages) is opened in the browser.

JS app/page.js



```
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script src="https://example.com/script
        </>
    )
}
```

Examples of scripts that do not need to load immediately and can be fetched with `lazyOnload` include:

- Chat support plugins

- Social media widgets

worker

Warning: The `worker` strategy is not yet stable and does not yet work with the App Router. Use with caution.

Scripts that use the `worker` strategy are off-loaded to a web worker in order to free up the main thread and ensure that only critical, first-party resources are processed on it. While this strategy can be used for any script, it is an advanced use case that is not guaranteed to support all third-party scripts.

To use `worker` as a strategy, the `nextScriptWorkers` flag must be enabled in `next.config.js`:

`next.config.js`

```
module.exports = {
  experimental: {
    nextScriptWorkers: true,
  },
}
```

`worker` scripts can only currently be used in the `pages/` directory:

`pages/home.tsx`

TypeScript ▾

```
import Script from 'next/script'

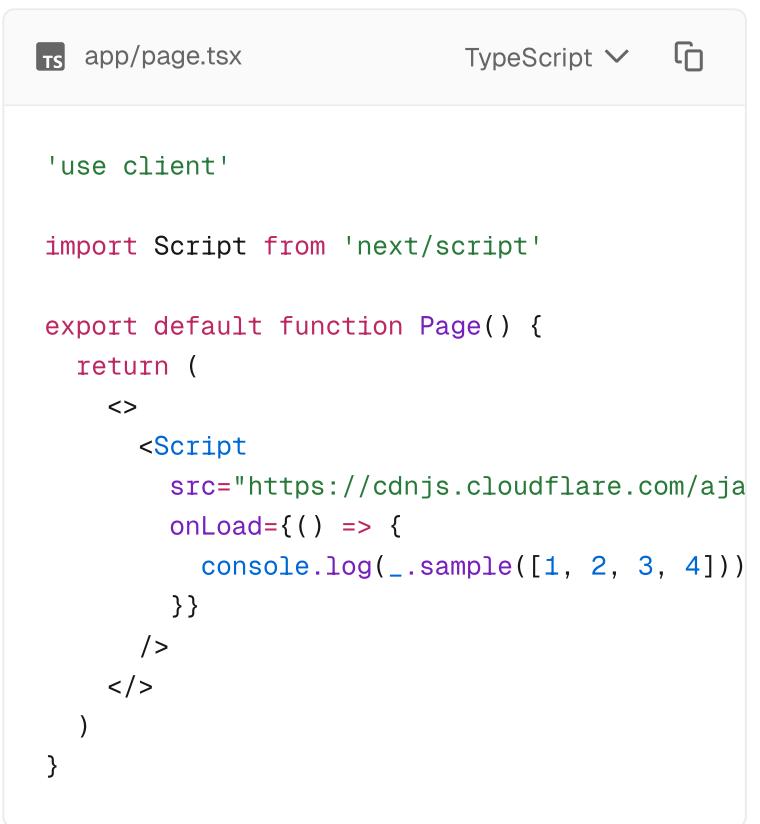
export default function Home() {
  return (
    <>
      <Script src="https://example.com/script" />
    </>
  )
}
```

onLoad

Warning: `onLoad` does not yet work with Server Components and can only be used in Client Components. Further, `onLoad` can't be used with `beforeInteractive` – consider using `onReady` instead.

Some third-party scripts require users to run JavaScript code once after the script has finished loading in order to instantiate content or call a function. If you are loading a script with either `afterInteractive` or `lazyOnload` as a loading strategy, you can execute code after it has loaded using the `onLoad` property.

Here's an example of executing a lodash method only after the library has been loaded.



```
TS app/page.tsx TypeScript ▾ ⌂

'use client'

import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://cdnjs.cloudflare.com/aj
        onLoad={() => {
          console.log(_.sample([1, 2, 3, 4]))
        }}
      />
    </>
  )
}
```

onReady

Warning: `onReady` does not yet work with Server Components and can only be used in Client Components.

Some third-party scripts require users to run JavaScript code after the script has finished loading and every time the component is mounted (after a route navigation for example). You can execute code after the script's load event when it first loads and then after every subsequent component re-mount using the `onReady` property.

Here's an example of how to re-instantiate a Google Maps JS embed every time the component is mounted:

```
import { useRef } from 'react'
import Script from 'next/script'

export default function Page() {
  const mapRef = useRef()

  return (
    <>
      <div ref={mapRef}></div>
      <Script
        id="google-maps"
        src="https://maps.googleapis.com/maps
        onReady={() => {
          new google.maps.Map(mapRef.current,
            center: { lat: -34.397, lng: 150.
              zoom: 8,
            })
        }}
      />
    </>
  )
}
```

onError

Warning: `onError` does not yet work with Server Components and can only be used in Client Components. `onError` cannot be used with the `beforeInteractive` loading strategy.

Sometimes it is helpful to catch when a script fails to load. These errors can be handled with the `onError` property:

```
import Script from 'next/script'

export default function Page() {
  return (
    <>
    <Script
      src="https://example.com/script.js"
      onError={(e: Error) => {
        console.error('Script failed to load')
      }}
    />
    </>
  )
}
```

Version History

Version	Changes
v13.0.0	beforeInteractive and afterInteractive is modified to support app.
v12.2.4	onReady prop added.
v12.2.2	Allow next/script with beforeInteractive to be placed in _document.
v11.0.0	next/script introduced.

Was this helpful?    

[Copy page](#)

Using Pages Router

Features available in /pages

Latest Version

15.5.4

ⓘ You are currently viewing the documentation for Pages Router.

File-system conventions

instrumentation

API reference for the instrumentation.j...

Middleware

Learn how to use Middleware to run code before a...

public

Next.js allows you to serve static files, like images,...

src Directory

Save pages under the `src` folder as an alternative to...

Was this helpful?



 Using Pages Router
Features available in /pages

 Latest Version
15.5.4

(i) You are currently viewing the documentation for Pages Router.

instrumentation.js

The `instrumentation.js|ts` file is used to integrate observability tools into your application, allowing you to track the performance and behavior, and to debug issues in production.

To use it, place the file in the **root** of your application or inside a `src` folder if using one.

Exports

`register` (optional)

The file exports a `register` function that is called **once** when a new Next.js server instance is initiated. `register` can be an async function.

TS instrumentation.ts TypeScript ▾

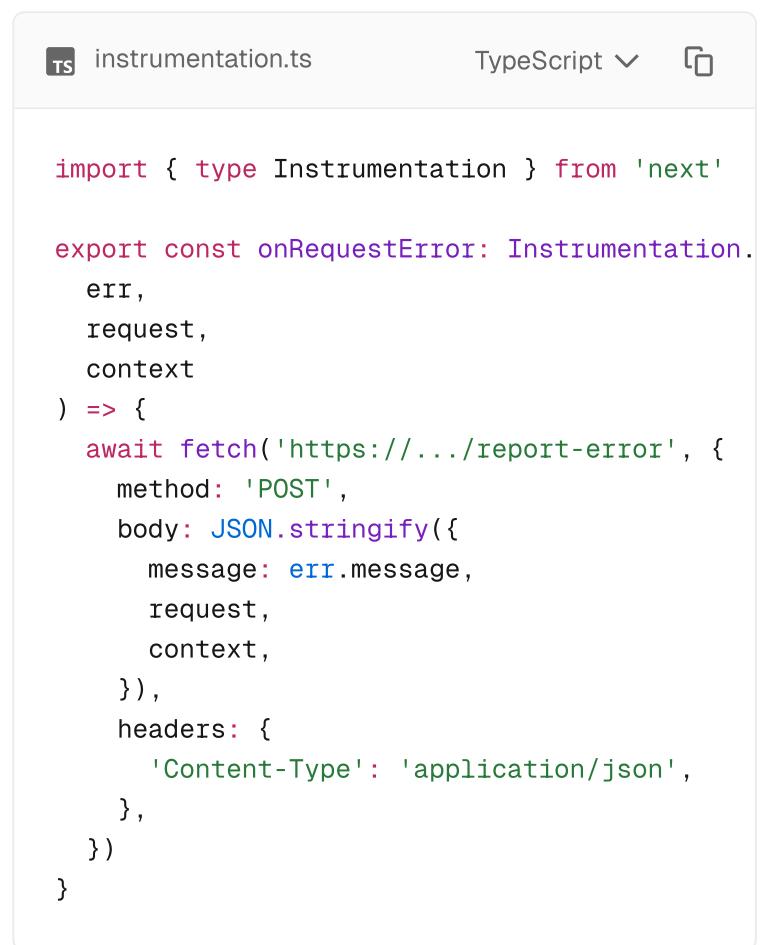
```
import { registerOTel } from '@vercel/otel'

export function register() {
  registerOTel('next-app')
}
```

`onRequestError` (optional)

You can optionally export an `onRequestError` function to track **server** errors to any custom observability provider.

- If you're running any async tasks in `onRequestError`, make sure they're awaited. `onRequestError` will be triggered when the Next.js server captures the error.
- The `error` instance might not be the original error instance thrown, as it may be processed by React if encountered during Server Components rendering. If this happens, you can use `digest` property on an error to identify the actual error type.



The screenshot shows a code editor window with the following details:

- File name: `instrumentation.ts`
- TypeScript version: `TypeScript v`
- Code content:

```
import { type Instrumentation } from 'next'

export const onRequestError: Instrumentation['onRequestError'] = (
  err,
  request,
  context
) => {
  await fetch('https://.../report-error', {
    method: 'POST',
    body: JSON.stringify({
      message: err.message,
      request,
      context,
    }),
    headers: {
      'Content-Type': 'application/json',
    },
  })
}
```

Parameters

The function accepts three parameters: `error`, `request`, and `context`.



The screenshot shows a code editor window with the following details:

- File name: `Types`
- TypeScript version: `v`
- Code content:

```
export function onRequestError(
```

```
error: { digest: string } & Error,
request: {
  path: string // resource path, e.g. /blog
  method: string // request method. e.g. GET
  headers: { [key: string]: string }
},
context: {
  routerKind: 'Pages Router' | 'App Router'
  routePath: string // the route file path,
  routeType: 'render' | 'route' | 'action'
  renderSource:
    | 'react-server-components'
    | 'react-server-components-payload'
    | 'server-rendering'
  revalidateReason: 'on-demand' | 'stale' |
  renderType: 'dynamic' | 'dynamic-resume'
}
): void | Promise<void>
```

- `error` : The caught error itself (type is always `Error`), and a `digest` property which is the unique ID of the error.
 - `request` : Read-only request information associated with the error.
 - `context` : The context in which the error occurred. This can be the type of router (App or Pages Router), and/or (Server Components (`'render'`)), Route Handlers (`'route'`), Server Actions (`'action'`), or Middleware (`'middleware'`)).

Specifying the runtime

The `instrumentation.js` file works in both the Node.js and Edge runtime, however, you can use `process.env.NEXT_RUNTIME` to target a specific runtime.

is instrumentation.js

```
export function register() {
  if (process.env.NEXT_RUNTIME === 'edge') {
    return require('./register.edge')
  } else {
    return require('/register.node')
  }
}
```

```
        }
    }

    export function onRequestError() {
        if (process.env.NEXT_RUNTIME === 'edge') {
            return require('./on-request-error.edge')
        } else {
            return require('./on-request-error.node')
        }
    }
}
```

Version History

Version	Changes
v15.0.0	onRequestError introduced, instrumentation stable
v14.0.4	Turbopack support for instrumentation
v13.2.0	instrumentation introduced as an experimental feature

v15.0.0	onRequestError introduced, instrumentation stable
v14.0.4	Turbopack support for instrumentation
v13.2.0	instrumentation introduced as an experimental feature

Was this helpful?     

 Using Pages Router
Features available in /pages

 Latest Version
15.5.4

 You are currently viewing the documentation for Pages Router.

Middleware

The `middleware.js|ts` file is used to write **Middleware** and run code on the server 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 executes before routes are rendered. It's particularly useful for implementing custom server-side logic like authentication, logging, or handling redirects.

Good to know:

Middleware is meant to be invoked separately of your render code and in optimized cases deployed to your CDN for fast redirect/rewrite handling, you should not attempt relying on shared modules or globals.

To pass information from Middleware to your application, use [headers](#), [cookies](#), [rewrites](#), [redirects](#), or the URL.

Create a `middleware.ts` (or `.js`) file in the project root, or inside `src` if applicable, so that it is located at the same level as `pages` or `app`.

 middleware.ts

TypeScript ▼



```
import { NextResponse, NextRequest } from 'next'

// This function can be marked `async` if using
export function middleware(request: NextRequest)
  return NextResponse.redirect(new URL('/home'),
```

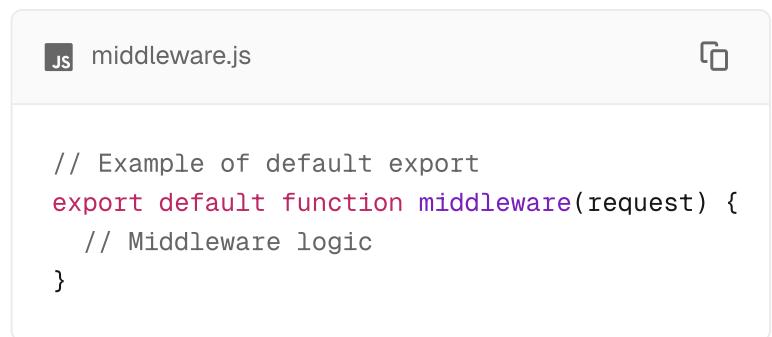
```
        }
      }

      export const config = {
        matcher: '/about/:path*',
      }
    }
```

Exports

Middleware function

The file must export a single function, either as a default export or named `middleware`. Note that multiple middleware from the same file are not supported.



A screenshot of a code editor showing a file named `middleware.js`. The file contains the following code:

```
// Example of default export
export default function middleware(request) {
  // Middleware logic
}
```

Config object (optional)

Optionally, a config object can be exported alongside the Middleware function. This object includes the `matcher` to specify paths where the Middleware applies.

Matcher

The `matcher` option allows you to target specific paths for the Middleware to run on. You can specify these paths in several ways:

- For a single path: Directly use a string to define the path, like `'/about'`.
- For multiple paths: Use an array to list multiple paths, such as `matcher: ['/about',`

'/contact'] , which applies the Middleware to both /about and /contact .

middleware.js

```
export const config = {  
  matcher: ['/about/:path*', '/dashboard/:path*']  
}
```

Additionally, the `matcher` option supports complex path specifications through regular expressions, such as

```
matcher:  
['/((?!api|_next/static|_next/image|.*\\.png$).*)']
```

, enabling precise control over which paths to include or exclude.

The `matcher` option accepts an array of objects with the following keys:

- `source` : The path or pattern used to match the request paths. It can be a string for direct path matching or a pattern for more complex matching.
- `regexp` (optional): A regular expression string that fine-tunes the matching based on the source. It provides additional control over which paths are included or excluded.
- `locale` (optional): A boolean that, when set to `false` , ignores locale-based routing in path matching.
- `has` (optional): Specifies conditions based on the presence of specific request elements such as headers, query parameters, or cookies.
- `missing` (optional): Focuses on conditions where certain request elements are absent, like missing headers or cookies.

middleware.js

```
export const config = {
  matcher: [
    {
      source: '/api/*',
      regexp: '^/api/(.*)',
      locale: false,
      has: [
        { type: 'header', key: 'Authorization' },
        { type: 'query', key: 'userId', value: '' },
      ],
      missing: [{ type: 'cookie', key: 'session' }],
    },
  ]
}
```

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:

- The `matcher` values need to be constants so they can be statically analyzed at build-time. Dynamic values such as variables will be ignored.
- For backward compatibility, Next.js always considers `/public` as `/public/index`. Therefore, a matcher of `/public/:path` will match.

Params

request

When defining Middleware, the default export function accepts a single parameter, `request`. This parameter is an instance of `NextRequest`, which represents the incoming HTTP request.



```
TS middleware.ts TypeScript ▾ ⌂

import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest
    // Middleware logic goes here
}
```

Good to know:

- `NextRequest` is a type that represents incoming HTTP requests in Next.js Middleware, whereas `NextResponse` is a class used to manipulate and send back HTTP responses.

NextResponse

The `NextResponse` API allows you to:

- `redirect` the incoming request to a different URL
- `rewrite` the response by displaying a given URL
- Set request headers for API Routes, `getServerSideProps`, and `rewrite` destinations
- Set response cookies
- Set response headers

To produce a response from Middleware, you can:

1. `rewrite` to a route ([Page](#) or [Edge API Route](#)) that produces a response
 2. return a `NextResponse` directly. See [Producing a Response](#)
-

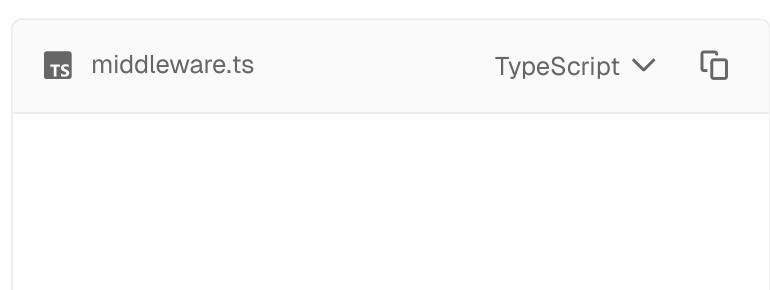
Execution order

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`
-

Runtime

Middleware defaults to using the Edge runtime. As of v15.5, we have support for using the Node.js runtime. To enable, in your middleware file, set the runtime to `nodejs` in the `config` object:



The screenshot shows a code editor interface with a file named `middleware.ts`. The file is associated with the TypeScript language (indicated by the `TS` icon). The editor has a toolbar at the top with options for "TypeScript" and a refresh symbol. The main workspace is currently empty, showing a light gray background.

```
export const config = {
  runtime: 'nodejs',
}
```

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.

next.config.js

```
module.exports = {
  skipTrailingSlashRedirect: true,
}
```

middleware.js

```
const legacyPrefixes = ['/docs', '/blog']

export default async function middleware(req) {
  const { pathname } = req.nextUrl

  if (legacyPrefixes.some((prefix) => pathname.startsWith(prefix)))
    return NextResponse.next()

  // apply trailing slash handling
  if (
    !pathname.endsWith('/') &&
    !pathname.match(/((?!\.well-known(?:\.\w+)?|index)\.)*\w+/))
    ) {
```

```
        return NextResponse.redirect(
          new URL(`#${req.nextUrl.pathname}/`, req.url)
        )
      }
    }
  }
}
```

`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.

JS next.config.js

```
module.exports = {
  skipMiddlewareUrlNormalize: true,
}
```

JS middleware.js

```
export default async function middleware(req) {
  const { pathname } = req.nextUrl

  // GET /_next/data/build-id/hello.json

  console.log(pathname)
  // with the flag this now /_next/data/build-id/
  // without the flag this would be normalized
}
```

Examples

Conditional Statements

TS middleware.ts

TypeScript ▾

```
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'))
```

```
}
```

```
if (request.nextUrl.pathname.startsWith('/das
    return NextResponse.rewrite(new URL('/dashb
}
}
```

Using Cookies

Cookies are regular headers. On a `Request`, they are stored in the `Cookie` header. On a `Response` they are in the `Set-Cookie` header. Next.js provides a convenient way to access and manipulate these cookies through the `cookies` extension on `NextRequest` and `NextResponse`.

1. For incoming requests, `cookies` comes with the following methods: `get`, `getAll`, `set`, and `delete`. You can check for the existence of a cookie with `has` or remove all cookies with `clear`.
2. For outgoing responses, `cookies` have the following methods `get`, `getAll`, `set`, and `delete`.

middleware.ts

TypeScript

```
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
    // Getting cookies from the request using the .get() method
    let cookie = request.cookies.get('nextjs')
    console.log(cookie) // => { name: 'nextjs', value: 'fast' }
    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

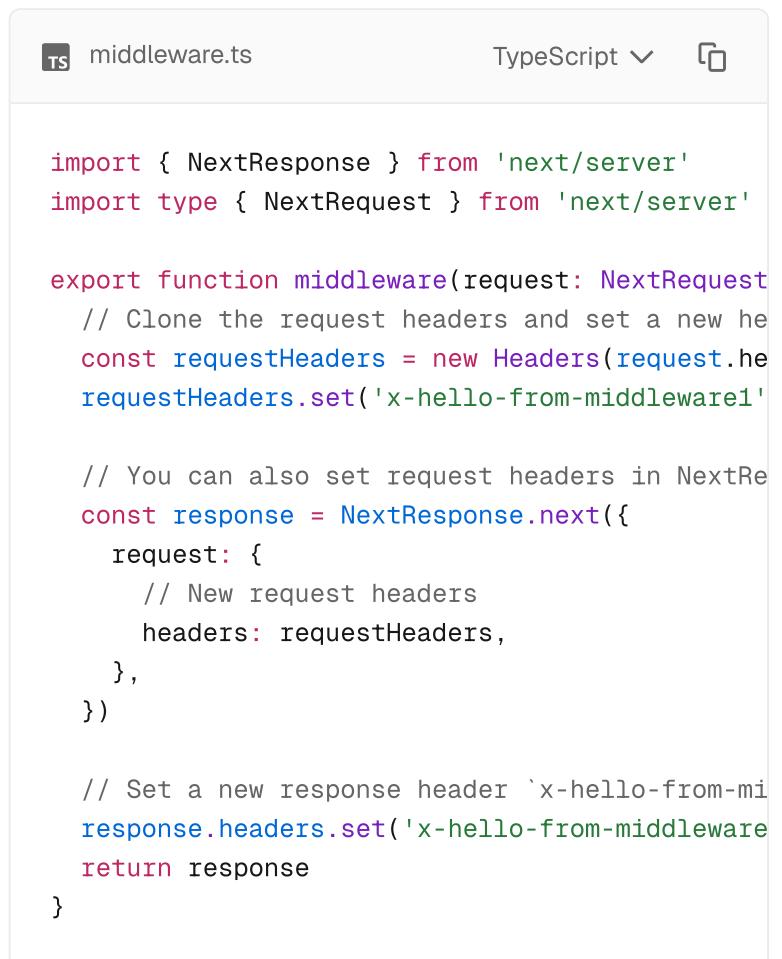
    // Setting cookies on the response using the .set() method
    const response = NextResponse.next()
    response.cookies.set('vercel', 'fast')
    response.cookies.set({
        name: 'vercel',
        value: 'fast',
        path: '/',
    })
}
```

```
})
cookie = response.cookies.get('vercel')
console.log(cookie) // => { name: 'vercel', v
// The outgoing response will have a `Set-Coo

return response
}
```

Setting Headers

You can set request and response headers using the `NextResponse` API (setting *request* headers is available since Next.js v13.0.0).



```
TS middleware.ts TypeScript ▾ ⌂

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
  const requestHeaders = new Headers(request.headers)
  requestHeaders.set('x-hello-from-middleware1', 'true')

  // You can also set request headers in NextResponse
  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', 'true')
  return response
}
```

Note that the snippet uses:

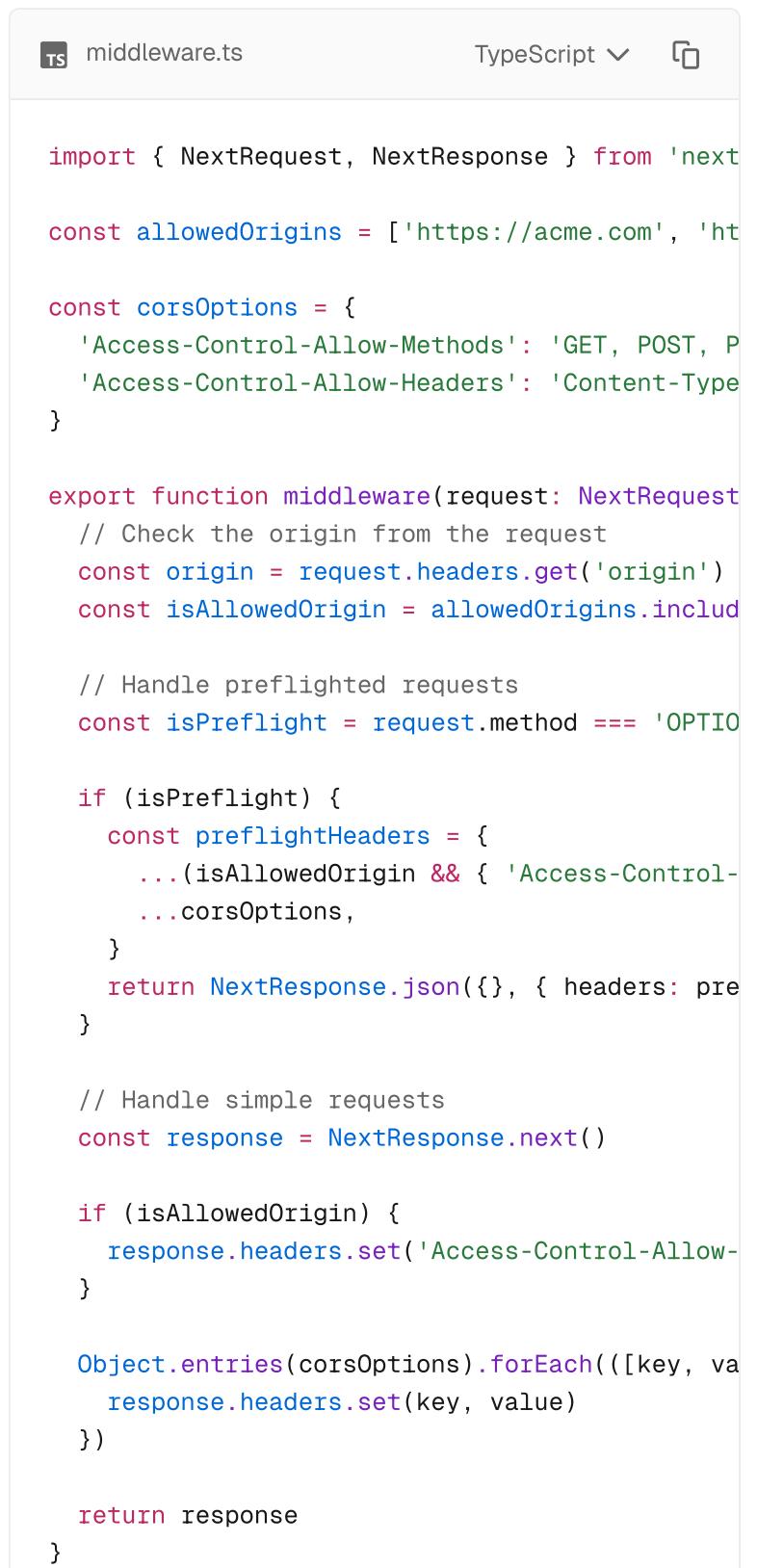
- `NextResponse.next({ request: { headers: requestHeaders } })` to make `requestHeaders` available upstream
- **NOT** `NextResponse.next({ headers: requestHeaders })` which makes `requestHeaders` available to clients

Learn more in [NextResponse headers in Middleware](#).

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.



The screenshot shows a code editor window with the file name "middleware.ts" and a TypeScript language icon. The code implements a middleware function to handle CORS requests. It defines allowed origins and sets CORS options for both preflighted and simple requests, including methods and headers.

```
import { NextRequest, NextResponse } from 'next'

const allowedOrigins = ['https://acme.com', 'ht']

const corsOptions = {
  'Access-Control-Allow-Methods': 'GET, POST, P',
  'Access-Control-Allow-Headers': 'Content-Type'
}

export function middleware(request: NextRequest) {
  // Check the origin from the request
  const origin = request.headers.get('origin')
  const isAllowedOrigin = allowedOrigins.includ

  // Handle preflighted requests
  const isPreflight = request.method === 'OPTIO

  if (isPreflight) {
    const preflightHeaders = {
      ...(isAllowedOrigin && { 'Access-Control-
      ...corsOptions,
    }
    return NextResponse.json({}, { headers: pre
  }

  // Handle simple requests
  const response = NextResponse.next()

  if (isAllowedOrigin) {
    response.headers.set('Access-Control-Allow-
  }

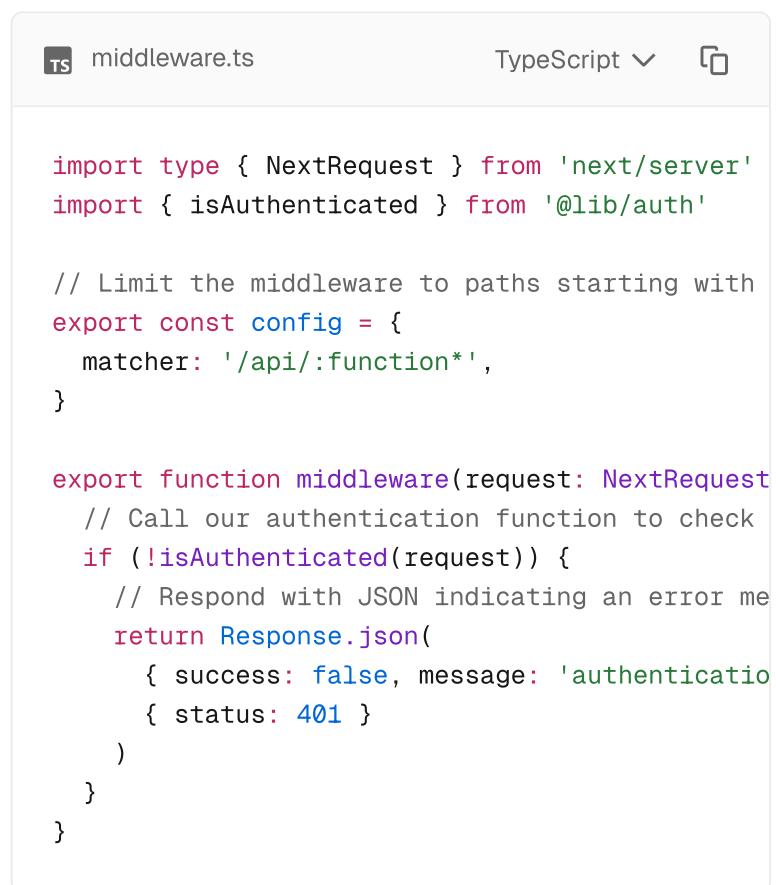
  Object.entries(corsOptions).forEach(([key, va
    response.headers.set(key, value)
  })

  return response
}
```

```
export const config = {
  matcher: '/api/:path*',
}
```

Producing a response

You can respond from Middleware directly by returning a `Response` or `NextResponse` instance. (This is available since [Next.js v13.1.0 ↗](#))



```
middleware.ts  TypeScript ▾
```

```
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 if (!isAuthenticated(request)) {
    // Respond with JSON indicating an error message
    return Response.json({
      success: false, message: 'authentication failed',
      status: 401
    })
  }
}
```

Negative matching

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:



```
middleware.js
```

```
export const config = {
  matcher: [
    /* 
      * Match all request paths except for the ones:
      * - api (API routes)
      * - _next/static (static files)
      * - _next/image (image optimization files)
    */
  ],
}
```

```
* - favicon.ico, sitemap.xml, robots.txt (
```

```
 */
'/(?!api|_next/static|_next/image|favicon.
```

```
],
}
```

You can also bypass Middleware for certain requests by using the `missing` or `has` arrays, or a combination of both:

JS middleware.js



```
export const config = {
  matcher: [
    /*
     * Match all request paths except for the ones
     * - api (API routes)
     * - _next/static (static files)
     * - _next/image (image optimization files)
     * - favicon.ico, sitemap.xml, robots.txt (
     */
  ],
  source:
    '/(?!api|_next/static|_next/image|favicon)',
  missing: [
    { type: 'header', key: 'next-router-preload' },
    { type: 'header', key: 'purpose', value: 'navigation' },
  ],
  has: [
    { type: 'header', key: 'next-router-preload' },
    { type: 'header', key: 'purpose', value: 'navigation' },
  ],
  source:
    '/(?!api|_next/static|_next/image|favicon)',
  has: [{ type: 'header', key: 'x-present' }],
  missing: [{ type: 'header', key: 'x-missing' }],
}
```

`waitFor` and `NextFetchEvent`

The `NextFetchEvent` object extends the native `FetchEvent` ↗ object, and includes the `waitFor()` ↗ method.

The `waitFor()` 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.

```
TS middleware.ts

import { NextResponse } from 'next/server'
import type { NextFetchEvent, NextRequest } from 'next'

export function middleware(req: NextRequest, event: NextFetchEvent) {
  event.waitFor(
    fetch('https://my-analytics-platform.com',
      { method: 'POST',
        body: JSON.stringify({ pathname: req.nextUrl.pathname })
      })
  )

  return NextResponse.next()
}
```

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',
  headers: {
    'x-test': 'true'
  }
}).toEqual(true))
```

```
})
).toEqual(false)
```

The entire middleware function can also be tested.

```
import { isRewrite, getRewrittenUrl } from 'next'

const request = new NextRequest('https://nextjs.org')
const response = await middleware(request)
expect(isRewrite(response)).toEqual(true)
expect(getRewrittenUrl(response)).toEqual('http://nextjs.org')
// getRedirectUrl could also be used if the response is a redirect
```

Platform support

Deployment Option	Supported
Node.js server	Yes
Docker container	Yes
Static export	No
Adapters	Platform-specific

Learn how to [configure Middleware](#) when self-hosting Next.js.

Version history

Version	Changes
v15.5.0	Middleware can now use the Node.js runtime (stable)
v15.2.0	Middleware can now use the Node.js runtime (experimental)

Version Changes

v13.1.0 Advanced Middleware flags added

v13.0.0 Middleware can modify request headers, response headers, and send responses

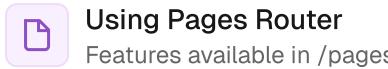
v12.2.0 Middleware is stable, please see the [upgrade guide](#)

v12.0.9 Enforce absolute URLs in Edge Runtime ([PR ↗](#))

v12.0.0 Middleware (Beta) added

Was this helpful?





(i) You are currently viewing the documentation for Pages Router.

public Folder

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:

avatar.js

```
import Image from 'next/image'

export function Avatar({ id, alt }) {
  return <Image src={`/avatars/${id}.png`} alt={alt} />
}

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:

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.](#)

Was this helpful?    

 **Using Pages Router**

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

 **Latest Version**

15.5.4

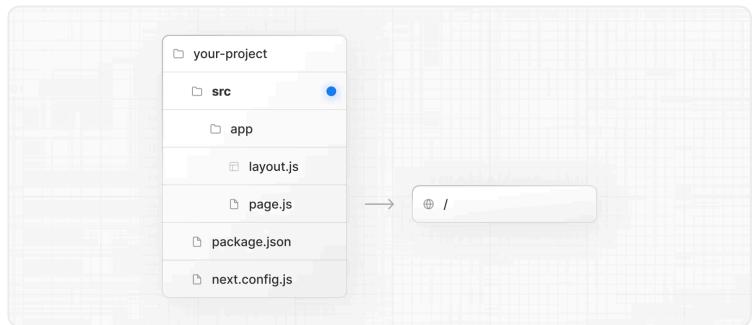


src Directory

As an alternative to having the special Next.js `app` or `pages` directories in the root of your project, Next.js also supports the common pattern of placing application code under the `src` folder.

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` folder, 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` folder.
- 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/`.

Was this helpful?    

Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4

Functions

getInitialProps

Fetch dynamic data on the server for your React...

getServerSideProps

API reference for `getServerSideProps`. Learn how to fetc...

getStaticPaths

API reference for `getStaticPaths`. Learn how to fetc...

getStaticProps

API reference for `getStaticProps`. Learn how to use...

NextRequest

API Reference for NextRequest.

NextResponse

API Reference for NextResponse.

useAmp

Enable AMP in a page, and control the way Next.js...

useReportWebVitals

useReportWebVital

useRouter

Learn more about the API of the Next.js Router, an...

userAgent

The userAgent helper extends the Web Request API...

Was this helpful?





Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



getInitialProps

Good to know: `getInitialProps` is a legacy API. We recommend using `getStaticProps` or `getServerSideProps` instead.

`getInitialProps` is an `async` function that can be added to the default exported React component for the page. It will run on both the server-side and again on the client-side during page transitions. The result of the function will be forwarded to the React component as `props`.

TS pages/index.tsx

TypeScript



```
import { NextPageContext } from 'next'

Page.getInitialProps = async (ctx: NextPageCo
const res = await fetch('https://api.github
const json = await res.json()
return { stars: json.stargazers_count }
}

export default function Page({ stars }: { sta
    return stars
}
```

Good to know:

- Data returned from `getInitialProps` is serialized when server rendering. Ensure the returned object from `getInitialProps` is a plain `Object`, and not using `Date`, `Map` or `Set`.
- For the initial page load, `getInitialProps` will run on the server only. `getInitialProps` will then

also run on the client when navigating to a different route with the `next/link` component or by using `next/router`.

- If `getInitialProps` is used in a custom `_app.js`, and the page being navigated to is using `getServerSideProps`, then `getInitialProps` will also run on the server.

Context Object

`getInitialProps` receives a single argument called `context`, which is an object with the following properties:

Name	Description
<code>pathname</code>	Current route, the path of the page in <code>/pages</code>
<code>query</code>	Query string of the URL, parsed as an object
<code>asPath</code>	<code>String</code> of the actual path (including the query) shown in the browser
<code>req</code>	HTTP request object ↗ (server only)
<code>res</code>	HTTP response object ↗ (server only)
<code>err</code>	Error object if any error is encountered during the rendering

Caveats

- `getInitialProps` can only be used in `pages/` top level files, and not in nested components. To have nested data fetching at the component level, consider exploring the [App Router](#).

- Regardless of whether your route is static or dynamic, any data returned from `getInitialProps` as `props` will be able to be examined on the client-side in the initial HTML. This is to allow the page to be [hydrated ↗](#) correctly. Make sure that you don't pass any sensitive information that shouldn't be available on the client in `props`.

Was this helpful?    



Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



getServerSideProps

When exporting a function called

`getServerSideProps` (Server-Side Rendering)

from a page, Next.js will pre-render this page on each request using the data returned by

`getServerSideProps`. This is useful if you want to fetch data that changes often, and have the page update to show the most current data.

TS pages/index.tsx

TypeScript



```
import type { InferGetServerSidePropsType, Ge

type Repo = {
  name: string
  stargazers_count: number
}

export const getServerSideProps = (async () =>
  // Fetch data from external API
  const res = await fetch('https://api.github.com/repos/reactjs/react')
  const repo: Repo = await res.json()
  // Pass data to the page via props
  return { props: { repo } }
) satisfies GetServerSideProps<{ repo: Repo }

export default function Page({
  repo,
}: InferGetServerSidePropsType<typeof getServerSideProps>) {
  return (
    <main>
      <p>{repo.stargazers_count}</p>
    </main>
  )
}
```

You can import modules in top-level scope for use in `getServerSideProps`. Imports used will **not be bundled for the client-side**. This means you can write **server-side code directly in `getServerSideProps`**, including fetching data from your database.

Context parameter

The `context` parameter is an object containing the following keys:

Name	Description
<code>params</code>	If this page uses a dynamic route , <code>params</code> contains the route parameters. If the page name is <code>[id].js</code> , then <code>params</code> will look like <code>{ id: ... }</code> .
<code>req</code>	The HTTP IncomingMessage object ↗ , with an additional <code>cookies</code> prop, which is an object with string keys mapping to string values of cookies.
<code>res</code>	The HTTP response object ↗ .
<code>query</code>	An object representing the query string, including dynamic route parameters.
<code>preview</code>	(Deprecated for <code>draftMode</code>) <code>preview</code> is <code>true</code> if the page is in the Preview Mode and <code>false</code> otherwise.
<code>previewData</code>	(Deprecated for <code>draftMode</code>) The <code>preview</code> data set by <code>setPreviewData</code> .
<code>draftMode</code>	<code>draftMode</code> is <code>true</code> if the page is in the Draft Mode and <code>false</code> otherwise.
<code>resolvedUrl</code>	A normalized version of the request URL that strips the <code>_next/data</code> prefix

Name	Description
	for client transitions and includes original query values.
locale	Contains the active locale (if enabled).
locales	Contains all supported locales (if enabled).
defaultLocale	Contains the configured default locale (if enabled).

getServerSideProps return values

The `getServerSideProps` function should return an object with **any one of the following** properties:

props

The `props` object is a key-value pair, where each value is received by the page component. It should be a [serializable object ↗](#) so that any props passed, could be serialized with [JSON.stringify ↗](#).

```
export async function getServerSideProps(context) {
  return {
    props: { message: `Next.js is awesome` },
  }
}
```

notFound

The `notFound` boolean allows the page to return a `404` status and [404 Page](#). With `notFound: true`, the page will return a `404` even if there was a

successfully generated page before. This is meant to support use cases like user-generated content getting removed by its author.

```
export async function getServerSideProps(context) {
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  if (!data) {
    return {
      notFound: true,
    }
  }

  return {
    props: { data }, // will be passed to the page
  }
}
```

redirect

The `redirect` object allows redirecting to internal and external resources. It should match the shape of

```
{ destination: string, permanent: boolean }
```

In some rare cases, you might need to assign a custom status code for older `HTTP` clients to properly redirect. In these cases, you can use the `statusCode` property instead of the `permanent` property, but not both.

```
export async function getServerSideProps(context) {
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  if (!data) {
    return {
      redirect: {
        destination: '/',
        permanent: false,
      },
    }
  }

  return {
    props: {}, // will be passed to the page
  }
}
```

```
    }  
}
```

Version History

Version	Changes
v13.4.0	App Router is now stable with simplified data fetching
v10.0.0	<code>locale</code> , <code>locales</code> , <code>defaultLocale</code> , and <code>notFound</code> options added.
v9.3.0	<code>getServerSideProps</code> introduced.

Version	Changes
v13.4.0	App Router is now stable with simplified data fetching
v10.0.0	<code>locale</code> , <code>locales</code> , <code>defaultLocale</code> , and <code>notFound</code> options added.
v9.3.0	<code>getServerSideProps</code> introduced.

Was this helpful?    



Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

getStaticPaths

When exporting a function called `getStaticPaths` from a page that uses [Dynamic Routes](#), Next.js will statically pre-render all the paths specified by `getStaticPaths`.

pages/repo/[name].tsx

TypeScript ▾



```
import type {
  InferGetStaticPropsType,
  GetStaticProps,
  GetStaticPaths,
} from 'next'

type Repo = {
  name: string
  stargazers_count: number
}

export const getStaticPaths = (async () => {
  return {
    paths: [
      {
        params: {
          name: 'next.js',
        },
      },
    ], // See the "paths" section below
    fallback: true, // false or "blocking"
  }
}) satisfies GetStaticPaths

export const getStaticProps = (async (context) => {
  const res = await fetch('https://api.github.com/repos/nextjs/next')
  const repo = await res.json()
  return { props: { repo } }
}) satisfies GetStaticProps<{
  repo: Repo
}>
```

```
>
```

```
export default function Page({  
  repo,  
}: InferGetStaticPropsType<typeof getStaticPr  
  return repo.stargazers_count  
}
```

getStaticPaths return values

The `getStaticPaths` function should return an object with the following **required** properties:

paths

The `paths` key determines which paths will be pre-rendered. For example, suppose that you have a page that uses [Dynamic Routes](#) named `pages/posts/[id].js`. If you export `getStaticPaths` from this page and return the following for `paths`:

```
return {  
  paths: [  
    { params: { id: '1' }},  
    {  
      params: { id: '2' },  
      // with i18n configured the locale for  
      locale: "en",  
    },  
    ],  
    fallback: ...  
  }  
}
```

Then, Next.js will statically generate `/posts/1` and `/posts/2` during `next build` using the page component in `pages/posts/[id].js`.

The value for each `params` object must match the parameters used in the page name:

- If the page name is `pages/posts/[postId]/[commentId]`, then `params` should contain `postId` and `commentId`.
- If the page name uses [catch-all routes](#) like `pages/[...slug]`, then `params` should contain `slug` (which is an array). If this array is `['hello', 'world']`, then Next.js will statically generate the page at `/hello/world`.
- If the page uses an [optional catch-all route](#), use `null`, `[]`, `undefined` or `false` to render the root-most route. For example, if you supply `slug: false` for `pages/[[...slug]]`, Next.js will statically generate the page `/`.

The `params` strings are **case-sensitive** and ideally should be normalized to ensure the paths are generated correctly. For example, if `WoRLD` is returned for a param it will only match if `WoRLD` is the actual path visited, not `world` or `World`.

Separate of the `params` object a `locale` field can be returned when [i18n is configured](#), which configures the locale for the path being generated.

`fallback: false`

If `fallback` is `false`, then any paths not returned by `getStaticPaths` will result in a **404 page**.

When `next build` is run, Next.js will check if `getStaticPaths` returned `fallback: false`, it will then build **only** the paths returned by `getStaticPaths`. This option is useful if you have a small number of paths to create, or new page data is not added often. If you find that you need to add more paths, and you have `fallback: false`, you will need to run `next build` again so that the new paths can be generated.

The following example pre-renders one blog post per page called `pages/posts/[id].js`. The list of blog posts will be fetched from a CMS and returned by `getStaticPaths`. Then, for each page, it fetches the post data from a CMS using `getStaticProps`.

`JS pages/posts/[id].js`



```
function Post({ post }) {
  // Render post...
}

// This function gets called at build time
export async function getStaticPaths() {
  // Call an external API endpoint to get pos
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to pre-render base
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

  // We'll pre-render only these paths at bui
  // { fallback: false } means other routes s
  return { paths, fallback: false }
}

// This also gets called at build time
export async function getStaticProps({ params
  // params contains the post `id`.
  // If the route is like /posts/1, then para
  const res = await fetch(`https://.../posts/
  const post = await res.json()

  // Pass post data to the page via props
  return { props: { post } }
}

export default Post
```

`fallback: true`

► Examples

If `fallback` is `true`, then the behavior of `getStaticProps` changes in the following ways:

- The paths returned from `getStaticPaths` will be rendered to `HTML` at build time by `getStaticProps`.
- The paths that have not been generated at build time will **not** result in a 404 page. Instead, Next.js will serve a “**fallback**” version of the page on the first request to such a path. Web crawlers, such as Google, won’t be served a fallback and instead the path will behave as in `fallback: 'blocking'`.
- When a page with `fallback: true` is navigated to through `next/link` or `next/router` (client-side) Next.js will *not* serve a fallback and instead the page will behave as `fallback: 'blocking'`.
- In the background, Next.js will statically generate the requested path `HTML` and `JSON`. This includes running `getStaticProps`.
- When complete, the browser receives the `JSON` for the generated path. This will be used to automatically render the page with the required props. From the user’s perspective, the page will be swapped from the fallback page to the full page.
- At the same time, Next.js adds this path to the list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, like other pages pre-rendered at build time.

Good to know: `fallback: true` is not supported when using `output: 'export'`.

When is `fallback: true` useful?

`fallback: true` is useful if your app has a very large number of static pages that depend on data (such as a very large e-commerce site). If you want

to pre-render all product pages, the builds would take a very long time.

Instead, you may statically generate a small subset of pages and use `fallback: true` for the rest. When someone requests a page that is not generated yet, the user will see the page with a loading indicator or skeleton component.

Shortly after, `getStaticProps` finishes and the page will be rendered with the requested data. From now on, everyone who requests the same page will get the statically pre-rendered page.

This ensures that users always have a fast experience while preserving fast builds and the benefits of Static Generation.

`fallback: true` will not *update* generated pages, for that take a look at [Incremental Static Regeneration](#).

`fallback: 'blocking'`

If `fallback` is `'blocking'`, new paths not returned by `getStaticPaths` will wait for the `HTML` to be generated, identical to SSR (hence why *blocking*), and then be cached for future requests so it only happens once per path.

`getStaticProps` will behave as follows:

- The paths returned from `getStaticPaths` will be rendered to `HTML` at build time by `getStaticProps`.
- The paths that have not been generated at build time will **not** result in a 404 page. Instead, Next.js will SSR on the first request and return the generated `HTML`.

- When complete, the browser receives the `HTML` for the generated path. From the user's perspective, it will transition from "the browser is requesting the page" to "the full page is loaded". There is no flash of loading/fallback state.
- At the same time, Next.js adds this path to the list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, like other pages pre-rendered at build time.

`fallback: 'blocking'` will not *update* generated pages by default. To update generated pages, use [Incremental Static Regeneration](#) in conjunction with `fallback: 'blocking'`.

Good to know: `fallback: 'blocking'` is not supported when using `output: 'export'`.

Fallback pages

In the “fallback” version of a page:

- The page's props will be empty.
- Using the `router`, you can detect if the fallback is being rendered, `router.isFallback` will be `true`.

The following example showcases using

`isFallback`:

```
JS pages/posts/[id].js Copy
import { useRouter } from 'next/router'

function Post({ post }) {
  const router = useRouter()

  // If the page is not yet generated, this will
  // initially until getStaticProps() finishes
```

```

    if (router.isFallback) {
      return <div>Loading...</div>
    }

    // Render post...
  }

  // This function gets called at build time
  export async function getStaticPaths() {
    return {
      // Only `/posts/1` and `/posts/2` are generated
      paths: [{ params: { id: '1' } }, { params
        // Enable statically generating additional
        // For example: `/posts/3`
        fallback: true,
      }]
    }
  }

  // This also gets called at build time
  export async function getStaticProps({ params
    // params contains the post `id`.
    // If the route is like /posts/1, then para
    const res = await fetch(`https://.../posts/
    const post = await res.json()

    // Pass post data to the page via props
    return {
      props: { post },
      // Re-generate the post at most once per
      // if a request comes in
      revalidate: 1,
    }
  }

  export default Post

```

Version History

Version	Changes
---------	---------

v13.4.0	App Router is now stable with simplified data fetching, including generateStaticParams()
---------	--

v12.2.0	On-Demand Incremental Static Regeneration is stable.
---------	--

Version	Changes
---------	---------

v12.1.0	On-Demand Incremental Static Regeneration added (beta).
---------	---

v9.5.0	Stable Incremental Static Regeneration
--------	--

v9.3.0	<code>getStaticPaths</code> introduced.
--------	---

Was this helpful?





Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

getStaticProps

Exporting a function called `getStaticProps` will pre-render a page at build time using the props returned from the function:

```
TS pages/index.tsx TypeScript 
```

```
import type { InferGetStaticPropsType, GetStaticProps } from 'next'

type Repo = {
  name: string
  stargazers_count: number
}

export const getStaticProps = (async (context) => {
  const res = await fetch('https://api.github.com/repos/vercel/next.js')
  const repo = await res.json()
  return { props: { repo } }
}) satisfies GetStaticProps<{
  repo: Repo
}>

export default function Page({
  repo,
}: InferGetStaticPropsType<typeof getStaticProps>) {
  return repo.stargazers_count
}
```

You can import modules in top-level scope for use in `getStaticProps`. Imports used will **not be bundled for the client-side**. This means you can write **server-side code directly in `getStaticProps`**, including fetching data from your database.

Context parameter

The `context` parameter is an object containing the following keys:

Name	Description
<code>params</code>	Contains the route parameters for pages using dynamic routes . For example, if the page name is <code>[id].js</code> , then <code>params</code> will look like <code>{ id: ... }</code> . You should use this together with <code>getStaticPaths</code> , which we'll explain later.
<code>preview</code>	(Deprecated for <code>draftMode</code>) <code>preview</code> is <code>true</code> if the page is in the Preview Mode and <code>false</code> otherwise.
<code>previewData</code>	(Deprecated for <code>draftMode</code>) The <code>preview</code> data set by <code>setPreviewData</code> .
<code>draftMode</code>	<code>draftMode</code> is <code>true</code> if the page is in the Draft Mode and <code>false</code> otherwise.
<code>locale</code>	Contains the active locale (if enabled).
<code>locales</code>	Contains all supported locales (if enabled).
<code>defaultLocale</code>	Contains the configured default locale (if enabled).
<code>revalidateReason</code>	Provides a reason for why the function was called. Can be one of: "build" (run at build time), "stale" (revalidate period expired, or running in development mode), "on-demand" (triggered via on-demand revalidation)

getStaticProps return values

The `getStaticProps` function should return an object containing either `props`, `redirect`, or `notFound` followed by an optional `revalidate` property.

props

The `props` object is a key-value pair, where each value is received by the page component. It should be a [serializable object ↗](#) so that any props passed, could be serialized with [JSON.stringify](#) ↗.

```
export async function getStaticProps(context) {
  return {
    props: { message: `Next.js is awesome` },
  }
}
```

revalidate

The `revalidate` property is the amount in seconds after which a page re-generation can occur (defaults to `false` or no revalidation).

```
// This function gets called at build time on
// It may be called again, on a serverless fu
// revalidation is enabled and a new request
export async function getStaticProps() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  return {
    props: {
      posts,
    },
    // Next.js will attempt to re-generate th
    // - When a request comes in
    // - At most once every 10 seconds
    revalidate: 10, // In seconds
  }
}
```

```
    }  
}
```

Learn more about [Incremental Static Regeneration](#).

The cache status of a page leveraging ISR can be determined by reading the value of the `x-nextjs-cache` response header. The possible values are the following:

- `MISS` - the path is not in the cache (occurs at most once, on the first visit)
- `STALE` - the path is in the cache but exceeded the revalidate time so it will be updated in the background
- `HIT` - the path is in the cache and has not exceeded the revalidate time

notFound

The `notFound` boolean allows the page to return a `404` status and [404 Page](#). With `notFound: true`, the page will return a `404` even if there was a successfully generated page before. This is meant to support use cases like user-generated content getting removed by its author. Note, `notFound` follows the same `revalidate` behavior [described here](#).

```
export async function getStaticProps(context) {
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  if (!data) {
    return {
      notFound: true,
    }
  }

  return {
    props: { data }, // will be passed to the
  }
}
```

Good to know: `notFound` is not needed for `fallback: false` mode as only paths returned from `getStaticPaths` will be pre-rendered.

redirect

The `redirect` object allows redirecting to internal or external resources. It should match the shape of `{ destination: string, permanent: boolean }`

In some rare cases, you might need to assign a custom status code for older `HTTP` clients to properly redirect. In these cases, you can use the `statusCode` property instead of the `permanent` property, **but not both**. You can also set `basePath: false` similar to redirects in `next.config.js`.

```
export async function getStaticProps(context) {
  const res = await fetch(`https://...`)
  const data = await res.json()

  if (!data) {
    return {
      redirect: {
        destination: '/',
        permanent: false,
        // statusCode: 301
      },
    }
  }
}
```

```
        }
    }

    return {
      props: { data }, // will be passed to the
    }
}
```

If the redirects are known at build-time, they should be added in `next.config.js` instead.

Reading files: Use `process.cwd()`

Files can be read directly from the filesystem in `getStaticProps`.

In order to do so you have to get the full path to a file.

Since Next.js compiles your code into a separate directory you can't use `__dirname` as the path it returns will be different from the Pages Router.

Instead you can use `process.cwd()` which gives you the directory where Next.js is being executed.

```
import { promises as fs } from 'fs'
import path from 'path'

// posts will be populated at build time by g
function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>
          <h3>{post.filename}</h3>
          <p>{post.content}</p>
        </li>
      )))
    </ul>
  )
}
```

```
// This function gets called at build time on
// It won't be called on client-side, so you
// direct database queries.
export async function getStaticProps() {
  const postsDirectory = path.join(process.cwd(), 'posts')
  const filenames = await fs.readdir(postsDirectory)

  const posts = filenames.map(async (filename) => {
    const filePath = path.join(postsDirectory, filename)
    const fileContents = await fs.readFile(filePath, 'utf8')

    // Generally you would parse/transform the
    // For example you can transform markdown

    return {
      filename,
      content: fileContents,
    }
  })
  // By returning { props: { posts } }, the Browser
  // will receive `posts` as a prop at build
  return {
    props: {
      posts: await Promise.all(posts),
    },
  }
}

export default Blog
```

Version History

Version	Changes
v13.4.0	App Router is now stable with simplified data fetching
v12.2.0	On-Demand Incremental Static Regeneration is stable.
v12.1.0	On-Demand Incremental Static Regeneration added (beta).
v10.0.0	<code>locale</code> , <code>locales</code> , <code>defaultLocale</code> , and <code>notFound</code> options added.

Version	Changes
v10.0.0	fallback: 'blocking' return option added.
v9.5.0	Stable Incremental Static Regeneration
v9.3.0	<code>getStaticProps</code> introduced.

Was this helpful?





Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



NextRequest

NextRequest extends the [WebRequest API ↗](#) with additional convenience methods.

cookies

Read or mutate the [Set-Cookie ↗](#) header of the request.

set(name, value)

Given a name, set a cookie with the given value on the request.

```
// Given incoming request /home
// Set a cookie to hide the banner
// request will have a 'Set-Cookie:show-banner'
request.cookies.set('show-banner', 'false')
```

get(name)

Given a cookie name, return the value of the cookie. If the cookie is not found, `undefined` is returned. If multiple cookies are found, the first one is returned.

```
// Given incoming request /home
// { name: 'show-banner', value: 'false', Pat
```

```
request.cookies.get('show-banner')
```

getAll()

Given a cookie name, return the values of the cookie. If no name is given, return all cookies on the request.

```
// Given incoming request /home
// [
//   { name: 'experiments', value: 'new-prici
//   { name: 'experiments', value: 'winter-la
// ]
request.cookies.getAll('experiments')
// Alternatively, get all cookies for the req
request.cookies.getAll()
```

delete(name)

Given a cookie name, delete the cookie from the request.

```
// Returns true for deleted, false is nothing
request.cookies.delete('experiments')
```

has(name)

Given a cookie name, return `true` if the cookie exists on the request.

```
// Returns true if cookie exists, false if it
request.cookies.has('experiments')
```

clear()

Remove the `Set-Cookie` header from the request.

```
request.cookies.clear()
```

nextUrl

Extends the native [URL](#) API with additional convenience methods, including Next.js specific properties.

```
// Given a request to /home, pathname is /hom  
request.nextUrl.pathname  
// Given a request to /home?name=lee, searchP  
request.nextUrl.searchParams
```

The following options are available:

Property	Type	Description
basePath	string	The base path of the URL.
buildId	string undefined	The build identifier of the Next.js application. Can be customized .
defaultLocale	string undefined	The default locale for internationalization .
domainLocale		
- defaultLocale	string	The default locale within a domain.
- domain	string	The domain associated with a specific locale.
- http	boolean undefined	Indicates if the domain is using HTTP.
locales	string[] undefined	An array of available locales.
locale	string undefined	The currently active locale.
url	URL	The URL object.

Version History

Version	Changes
v15.0.0	ip and geo removed.

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



NextResponse

NextResponse extends the [Web Response API](#) with additional convenience methods.

cookies

Read or mutate the [Set-Cookie](#) header of the response.

set(name, value)

Given a name, set a cookie with the given value on the response.

```
// Given incoming request /home
let response = NextResponse.next()
// Set a cookie to hide the banner
response.cookies.set('show-banner', 'false')
// Response will have a `Set-Cookie:show-banner=false`
return response
```

get(name)

Given a cookie name, return the value of the cookie. If the cookie is not found, [undefined](#) is returned. If multiple cookies are found, the first one is returned.

```
// Given incoming request /home
let response = NextResponse.next()
// { name: 'show-banner', value: 'false', Path
response.cookies.get('show-banner')
```

getAll()

Given a cookie name, return the values of the cookie. If no name is given, return all cookies on the response.

```
// Given incoming request /home
let response = NextResponse.next()
// [
//   { name: 'experiments', value: 'new-pricing' },
//   { name: 'experiments', value: 'winter-lights' }
// ]
response.cookies.getAll('experiments')
// Alternatively, get all cookies for the response
response.cookies.getAll()
```

delete(name)

Given a cookie name, delete the cookie from the response.

```
// Given incoming request /home
let response = NextResponse.next()
// Returns true for deleted, false is nothing
response.cookies.delete('experiments')
```

json()

Produce a response with the given JSON body.

TS app/api/route.ts

TypeScript ▾



```
import { NextResponse } from 'next/server'
```

```
export async function GET(request: Request) {  
  return NextResponse.json({ error: 'Internal  
  '}  
}
```

redirect()

Produce a response that redirects to a [URL ↗](#).

```
import { NextResponse } from 'next/server'  
  
return NextResponse.redirect(new URL('/new',
```

The [URL ↗](#) can be created and modified before being used in the `NextResponse.redirect()` method. For example, you can use the `request.nextUrl` property to get the current URL, and then modify it to redirect to a different URL.

```
import { NextResponse } from 'next/server'  
  
// Given an incoming request...  
const loginUrl = new URL('/login', request.url)  
// Add ?from=/incoming-url to the /login URL  
loginUrl.searchParams.set('from', request.nextUrl)  
// And redirect to the new URL  
return NextResponse.redirect(loginUrl)
```

rewrite()

Produce a response that rewrites (proxies) the given [URL ↗](#) while preserving the original URL.

```
import { NextResponse } from 'next/server'

// Incoming request: /about, browser shows /a
// Rewritten request: /proxy, browser shows /
return NextResponse.rewrite(new URL('/proxy',
```

next()

The `next()` method is useful for Middleware, as it allows you to return early and continue routing.

```
import { NextResponse } from 'next/server'

return NextResponse.next()
```

You can also forward `headers` upstream when producing the response, using

```
NextResponse.next({ request: { headers } })
:
```

```
import { NextResponse } from 'next/server'

// Given an incoming request...
const newHeaders = new Headers(request.headers)
// Add a new header
newHeaders.set('x-version', '123')
// Forward the modified request headers upstream
return NextResponse.next({
  request: {
    // New request headers
    headers: newHeaders,
  },
})
```

This forwards `newHeaders` upstream to the target page, route, or server action, and does not expose them to the client. While this pattern is useful for passing data upstream, it should be used with

caution because the headers containing this data may be forwarded to external services.

In contrast, `NextResponse.next({ headers })` is a shorthand for sending headers from middleware to the client. This is **NOT** good practice and should be avoided. Among other reasons because setting response headers like `Content-Type`, can override framework expectations (for example, the `Content-Type` used by Server Actions), leading to failed submissions or broken streaming responses.

```
import { type NextRequest, NextResponse } from 'next/server'

async function middleware(request: NextRequest) {
  const headers = await injectAuth(request.headers)
  // DO NOT forward headers like this
  return NextResponse.next({ headers })
}
```

In general, avoid copying all incoming request headers because doing so can leak sensitive data to clients or upstream services.

Prefer a defensive approach by creating a subset of incoming request headers using an allow-list. For example, you might discard custom `x-*` headers and only forward known-safe headers:

```
import { type NextRequest, NextResponse } from 'next/server'

function middleware(request: NextRequest) {
  const incoming = new Headers(request.headers)
  const forwarded = new Headers()

  for (const [name, value] of incoming) {
    const headerName = name.toLowerCase()
    // Keep only known-safe headers, discard others
    if (
      !headerName.startsWith('x-') &&
      headerName !== 'authorization' &&
      headerName !== 'cookie'
    ) {
      // Preserve original header name casing
      forwarded.set(name, value)
    }
  }
  return NextResponse.next({ headers: forwarded })
}
```

```
        }
    }

    return NextResponse.next({
        request: {
            headers: forwarded,
        },
    })
}
```

Was this helpful?     



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



useAmp

► Examples

Warning: Built-in AMP support will be removed in Next.js 16.

To enable AMP, add the following config to your page:

JS pages/index.js



```
export const config = { amp: true }
```

The `amp` config accepts the following values:

- `true` - The page will be AMP-only
- `'hybrid'` - The page will have two versions, one with AMP and another one with HTML

To learn more about the `amp` config, read the sections below.

AMP First Page

Take a look at the following example:

JS pages/about.js



```
export const config = { amp: true }

function About(props) {
  return <h3>My AMP About Page!</h3>
}

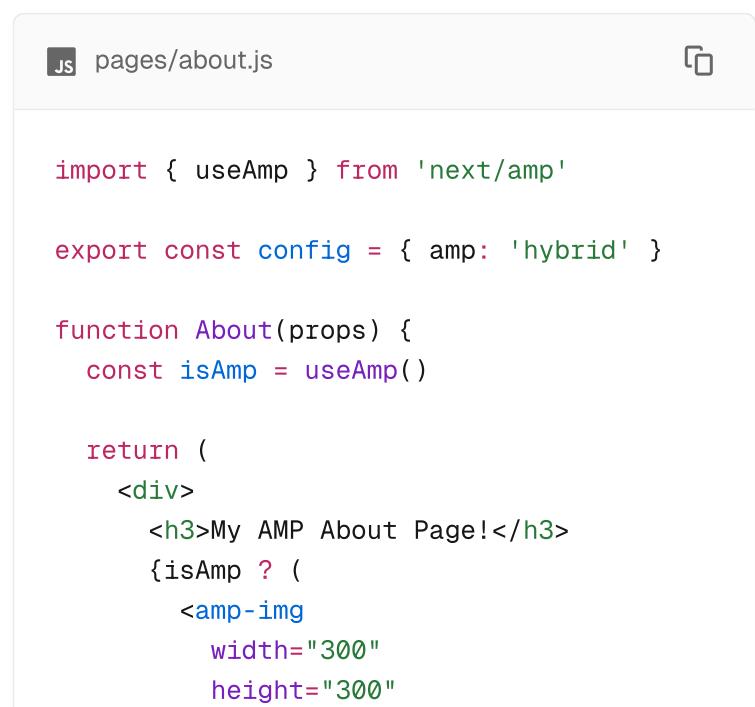
export default About
```

The page above is an AMP-only page, which means:

- The page has no Next.js or React client-side runtime
- The page is automatically optimized with [AMP Optimizer](#), an optimizer that applies the same transformations as AMP caches (improves performance by up to 42%)
- The page has a user-accessible (optimized) version of the page and a search-engine indexable (unoptimized) version of the page

Hybrid AMP Page

Take a look at the following example:



```
JS pages/about.js
```

```
import { useAmp } from 'next/amp'

export const config = { amp: 'hybrid' }

function About(props) {
  const isAmp = useAmp()

  return (
    <div>
      <h3>My AMP About Page!</h3>
      {isAmp ? (
        <amp-img
          width="300"
          height="300"
      ) : (
        <img
          alt="Placeholder image for the AMP page" style={style}
        />
      )}
    </div>
  )
}
```

```
        src="/my-img.jpg"
        alt="a cool image"
        layout="responsive"
      />
    ) : (
       {
  switch (metric.name) {
    case 'FCP':
      // handle FCP results
    }
    case 'LCP':
      // handle LCP results
    }
    // ...
  }
}

function MyApp({ Component, pageProps }) {
  useReportWebVitals(handleWebVitals)

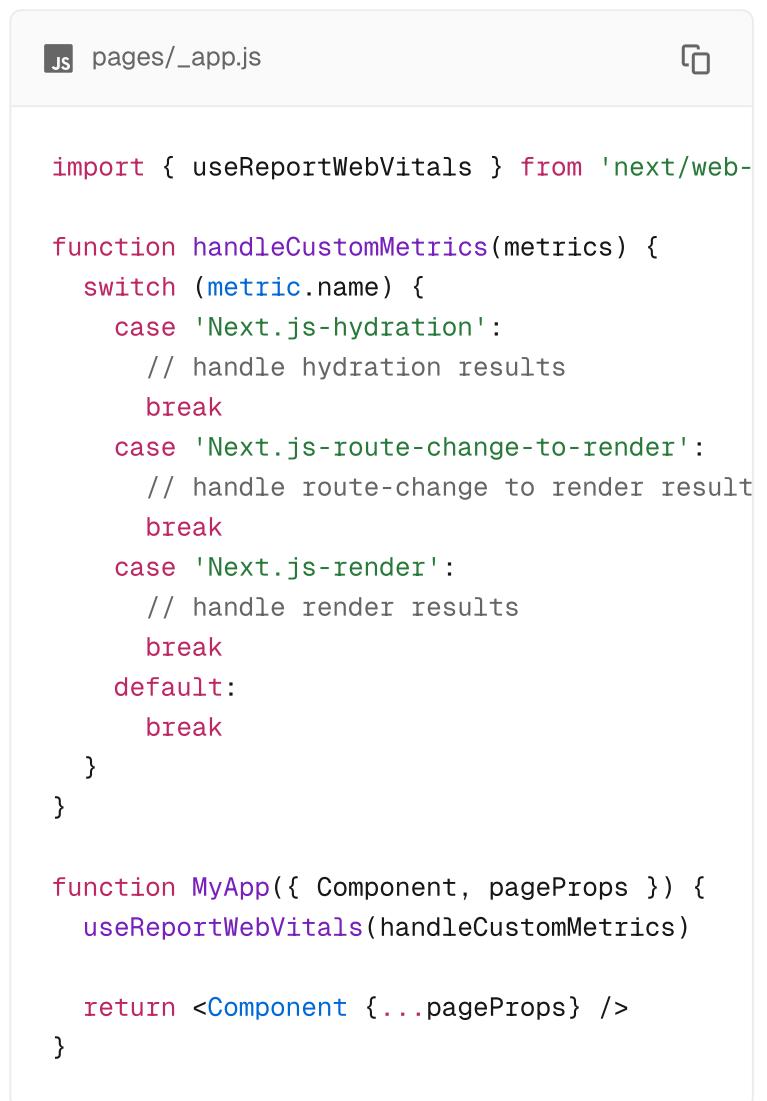
  return <Component {...pageProps} />
}
```

Custom Metrics

In addition to the core metrics listed above, there are some additional custom metrics that measure the time it takes for the page to hydrate and render:

- `Next.js-hydration` : Length of time it takes for the page to start and finish hydrating (in ms)
- `Next.js-route-change-to-render` : Length of time it takes for a page to start rendering after a route change (in ms)
- `Next.js-render` : Length of time it takes for a page to finish render after a route change (in ms)

You can handle all the results of these metrics separately:



The screenshot shows a code editor window with the file `pages/_app.js` open. The code uses the `useReportWebVitals` hook from the `'next/web-vitals'` package to handle custom metrics. It defines a `handleCustomMetrics` function that switches on the metric name. It handles three specific metrics: `Next.js-hydration`, `Next.js-route-change-to-render`, and `Next.js-render`. For each case, it includes a comment indicating the purpose and a `break` statement. If none of the cases match, it falls through to a `default` block, which also contains a `break` statement. Finally, the `useReportWebVitals` hook is called with the `handleCustomMetrics` function as its argument. The code then returns the `Component` wrapped in a component that receives the `pageProps`.

```
js pages/_app.js

import { useReportWebVitals } from 'next/web-vitals'

function handleCustomMetrics(metrics) {
  switch (metric.name) {
    case 'Next.js-hydration':
      // handle hydration results
      break
    case 'Next.js-route-change-to-render':
      // handle route-change to render result
      break
    case 'Next.js-render':
      // handle render results
      break
    default:
      break
  }
}

function MyApp({ Component, pageProps }) {
  useReportWebVitals(handleCustomMetrics)

  return <Component {...pageProps} />
}
```

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:

```
function postWebVitals(metrics) {  
  const body = JSON.stringify(metric)  
  const url = 'https://example.com/analytics'  
  
  // Use `navigator.sendBeacon()` if available  
  if (navigator.sendBeacon) {  
    navigator.sendBeacon(url, body)  
  } else {  
    fetch(url, { body, method: 'POST', keepalive: true })  
  }  
  
  useReportWebVitals(postWebVitals)  
}
```

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  
  // https://github.com/vercel/next.js/blob/canary/packages/react-next/docs/guides/tracking.md  
  window.gtag('event', metric.name, {  
    value: Math.round(metric.name === 'CLS' ? metric.value : metric.duration),  
    event_label: metric.id, // id unique to this metric  
    non_interaction: true, // avoids affecting other interactions  
  });  
})
```

Read more about [sending results to Google Analytics ↗](#).

Was this helpful?  





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



useRouter

If you want to access the `router` object inside any function component in your app, you can use the `useRouter` hook, take a look at the following example:

```
import { useRouter } from 'next/router'

function ActiveLink({ children, href }) {
  const router = useRouter()
  const style = {
    marginRight: 10,
    color: router.asPath === href ? 'red' : ''
  }

  const handleClick = (e) => {
    e.preventDefault()
    router.push(href)
  }

  return (
    <a href={href} onClick={handleClick} style={style}>
      {children}
    </a>
  )
}

export default ActiveLink
```

`useRouter` is a [React Hook ↗](#), meaning it cannot be used with classes. You can either use `withRouter` or wrap your class in a function component.

router object

The following is the definition of the `router` object returned by both `useRouter` and `withRouter`:

- `pathname : String` - The path for current route file that comes after `/pages`. Therefore, `basePath`, `locale` and trailing slash (`trailingSlash: true`) are not included.
- `query : Object` - The query string parsed to an object, including [dynamic route](#) parameters. It will be an empty object during prerendering if the page doesn't use [Server-side Rendering](#).
Defaults to `{}`
- `asPath : String` - The path as shown in the browser including the search params and respecting the `trailingSlash` configuration.
`basePath` and `locale` are not included.
- `isFallback : boolean` - Whether the current page is in [fallback mode](#).
- `basePath : String` - The active `basePath` (if enabled).
- `locale : String` - The active locale (if enabled).
- `locales : String[]` - All supported locales (if enabled).
- `defaultLocale : String` - The current default locale (if enabled).
- `domainLocales : Array<{domain, defaultLocale, locales}>` - Any configured domain locales.
- `isReady : boolean` - Whether the router fields are updated client-side and ready for use.
Should only be used inside of `useEffect`

methods and not for conditionally rendering on the server. See related docs for use case with [automatically statically optimized pages](#)

- `isPreview`: `boolean` - Whether the application is currently in [preview mode](#).

Using the `asPath` field may lead to a mismatch between client and server if the page is rendered using server-side rendering or [automatic static optimization](#). Avoid using `asPath` until the `isReady` field is `true`.

The following methods are included inside `router`:

router.push

Handles client-side transitions, this method is useful for cases where [next/link](#) is not enough.

```
router.push(url, as, options)
```

- `url`: `UrlObject | String` - The URL to navigate to (see [Node.js URL module documentation](#) ↗ for `UrlObject` properties).
- `as`: `UrlObject | String` - Optional decorator for the path that will be shown in the browser URL bar. Before Next.js 9.5.3 this was used for dynamic routes.
- `options` - Optional object with the following configuration options:
 - `scroll` - Optional boolean, controls scrolling to the top of the page after navigation. Defaults to `true`
 - `shallow` : Update the path of the current page without rerunning `getStaticProps`, `getServerSideProps` or `getInitialProps`. Defaults to `false`

- `locale` - Optional string, indicates locale of the new page

You don't need to use `router.push` for external URLs. [window.location](#) ↗ is better suited for those cases.

Navigating to `pages/about.js`, which is a predefined route:

```
import { useRouter } from 'next/router'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => rout
      Click me
    </button>
  )
}
```

Navigating `pages/post/[pid].js`, which is a dynamic route:

```
import { useRouter } from 'next/router'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => rout
      Click me
    </button>
  )
}
```

Redirecting the user to `pages/login.js`, useful for pages behind [authentication](#):

```
import { useEffect } from 'react'
import { useRouter } from 'next/router'

// Here you would fetch and return the user
const useUser = () => ({ user: null, loading:
```

```

export default function Page() {
  const { user, loading } = useUser()
  const router = useRouter()

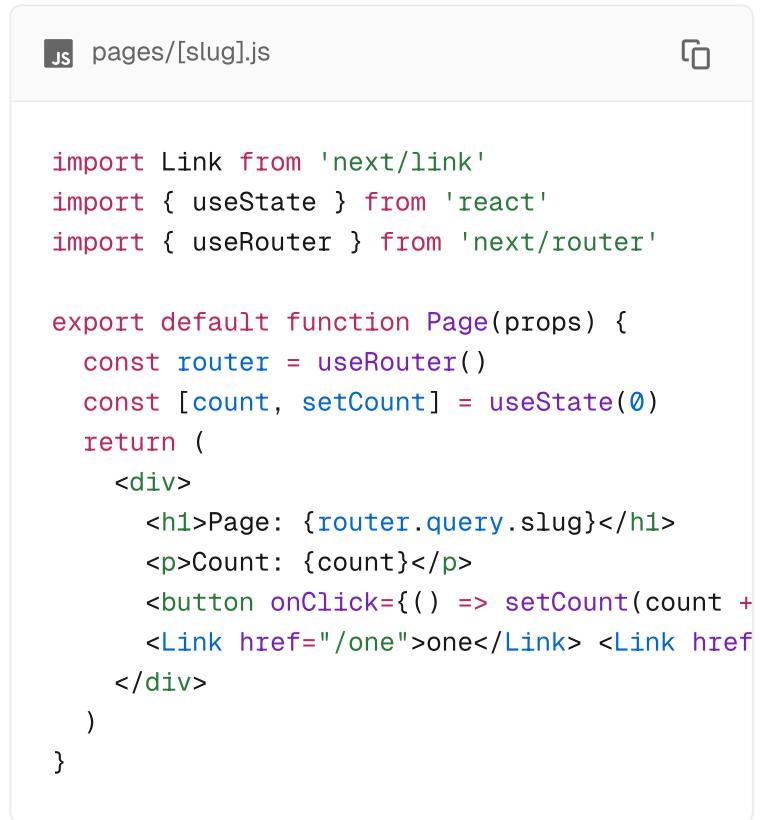
  useEffect(() => {
    if (!(user || loading)) {
      router.push('/login')
    }
  }, [user, loading])

  return <p>Redirecting...</p>
}

```

Resetting state after navigation

When navigating to the same page in Next.js, the page's state **will not** be reset by default as React does not unmount unless the parent component has changed.



```

JS pages/[slug].js

```

```

import Link from 'next/link'
import { useState } from 'react'
import { useRouter } from 'next/router'

export default function Page(props) {
  const router = useRouter()
  const [count, setCount] = useState(0)
  return (
    <div>
      <h1>Page: {router.query.slug}</h1>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
      <Link href="/one">one</Link> <Link href="/two">two</Link>
    </div>
  )
}

```

In the above example, navigating between /one and /two **will not** reset the count. The useState is maintained between renders because the top-level React component, Page, is the same.

If you do not want this behavior, you have a couple of options:

- Manually ensure each state is updated using `useEffect`. In the above example, that could look like:

```
useEffect(() => {
  setCount(0)
}, [router.query.slug])
```

- Use a React `key` to tell React to remount the component ↗. To do this for all pages, you can use a custom app:

JS pages/_app.js

```
import { useRouter } from 'next/router'

export default function MyApp({ Component,
  const router = useRouter()
  return <Component key={router.asPath} ...
})
```

With URL object

You can use a URL object in the same way you can use it for `next/link`. Works for both the `url` and `as` parameters:

```
import { useRouter } from 'next/router'

export default function ReadMore({ post }) {
  const router = useRouter()

  return (
    <button
      type="button"
      onClick={() => {
        router.push({
          pathname: '/post/[pid]',
          query: { pid: post.id },
        })
      }}
    >
      Click here to read more
    </button>
  )
}
```

router.replace

Similar to the `replace` prop in `next/link`, `router.replace` will prevent adding a new URL entry into the `history` stack.

```
router.replace(url, as, options)
```

- The API for `router.replace` is exactly the same as the API for `router.push`.

Take a look at the following example:

```
import { useRouter } from 'next/router'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => rout
      Click me
    </button>
  )
}
```

router.prefetch

Prefetch pages for faster client-side transitions.

This method is only useful for navigations without `next/link`, as `next/link` takes care of prefetching pages automatically.

This is a production only feature. Next.js doesn't prefetch pages in development.

```
router.prefetch(url, as, options)
```

- `url` - The URL to prefetch, including explicit routes (e.g. `/dashboard`) and dynamic routes (e.g. `/product/[id]`)
- `as` - Optional decorator for `url`. Before Next.js 9.5.3 this was used to prefetch dynamic routes.
- `options` - Optional object with the following allowed fields:
 - `locale` - allows providing a different locale from the active one. If `false`, `url` has to include the locale as the active locale won't be used.

Let's say you have a login page, and after a login, you redirect the user to the dashboard. For that case, we can prefetch the dashboard to make a faster transition, like in the following example:

```
import { useCallback, useEffect } from 'react'
import { useRouter } from 'next/router'

export default function Login() {
  const router = useRouter()
  const handleSubmit = useCallback((e) => {
    e.preventDefault()

    fetch('/api/login', {
      method: 'POST',
    })
    .then((res) => res.json())
    .then((data) => {
      if (data.error) {
        alert(data.error)
      } else {
        router.push('/dashboard')
      }
    })
  }, [router])
}

 handleSubmit
```

```

        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          /* Form data */
        }),
      }).then((res) => {
        // Do a fast client-side transition to
        if (res.ok) router.push('/dashboard')
      })
    }, []
  )

useEffect(() => {
  // Prefetch the dashboard page
  router.prefetch('/dashboard')
}, [router])

return (
  <form onSubmit={handleSubmit}>
    {/* Form fields */}
    <button type="submit">Login</button>
  </form>
)
}

```

router.beforePopState

In some cases (for example, if using a [Custom Server](#)), you may wish to listen to [popstate](#) and do something before the router acts on it.

```
router.beforePopState(cb)
```

- `cb` - The function to run on incoming `popstate` events. The function receives the state of the event as an object with the following props:
 - `url : String` - the route for the new state. This is usually the name of a `page`
 - `as : String` - the url that will be shown in the browser
 - `options : Object` - Additional options sent by `router.push`

If `cb` returns `false`, the Next.js router will not handle `popstate`, and you'll be responsible for handling it in that case. See [Disabling file-system routing](#).

You could use `beforePopState` to manipulate the request, or force a SSR refresh, as in the following example:

```
import { useEffect } from 'react'
import { useRouter } from 'next/router'

export default function Page() {
  const router = useRouter()

  useEffect(() => {
    router.beforePopState(({ url, as, options }) {
      // I only want to allow these two routes
      if (as !== '/' && as !== '/other') {
        // Have SSR render bad routes as a 404
        window.location.href = as
        return false
      }
    })
  }, [router])

  return <p>Welcome to the page</p>
}
```

router.back

Navigate back in history. Equivalent to clicking the browser's back button. It executes

```
window.history.back()
```

```
import { useRouter } from 'next/router'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.back()}>
      Click here to go back
    </button>
  )
}
```

```
}
```

router.reload

Reload the current URL. Equivalent to clicking the browser's refresh button. It executes

```
window.location.reload()
```

```
import { useRouter } from 'next/router'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.reload()}
      Click here to reload
    </button>
  )
}
```

router.events

You can listen to different events happening inside the Next.js Router. Here's a list of supported events:

- `routeChangeStart(url, { shallow })` - Fires when a route starts to change
- `routeChangeComplete(url, { shallow })` - Fires when a route changed completely
- `routeChangeError(err, url, { shallow })` - Fires when there's an error when changing routes, or a route load is cancelled
 - `err.cancelled` - Indicates if the navigation was cancelled
- `beforeHistoryChange(url, { shallow })` - Fires before changing the browser's history
- `hashChangeStart(url, { shallow })` - Fires when the hash will change but not the page

- `hashChangeComplete(url, { shallow })` -
Fires when the hash has changed but not the page

Good to know: Here `url` is the URL shown in the browser, including the `basePath`.

For example, to listen to the router event

`routeChangeStart`, open or create

`pages/_app.js` and subscribe to the event, like

so:

```
import { useEffect } from 'react'
import { useRouter } from 'next/router'

export default function MyApp({ Component, pageProps }) {
  const router = useRouter()

  useEffect(() => {
    const handleRouteChange = (url, { shallow }) => {
      console.log(`App is changing to ${url} ${shallow ? 'with' : 'without'} shallow routing`)
    }

    router.events.on('routeChangeStart', handleRouteChange)
    // If the component is unmounted, unsubscribe from the event with the `off` method:
    return () => {
      router.events.off('routeChangeStart', handleRouteChange)
    }
  }, [router])

  return <Component {...pageProps} />
}
```

We use a [Custom App](#) (`pages/_app.js`) for this example to subscribe to the event because it's not unmounted on page navigations, but you can subscribe to router events on any component in your application.

Router events should be registered when a component mounts ([useEffect](#) ↗ or [componentDidMount](#) ↗ / [componentWillUnmount](#) ↗) or imperatively when an event happens.

If a route load is cancelled (for example, by clicking two links rapidly in succession),

`routeChangeError` will fire. And the passed `err` will contain a `cancelled` property set to `true`, as in the following example:

```
import { useEffect } from 'react'
import { useRouter } from 'next/router'

export default function MyApp({ Component, pageProps }) {
  const router = useRouter()

  useEffect(() => {
    const handleRouteChangeError = (err, url) =>
      if (err.cancelled) {
        console.log(`Route to ${url} was cancelled`)
      }
  })

  router.events.on('routeChangeError', handleRouteChangeError)

  // If the component is unmounted, unsubscribe from the event with the `off` method:
  return () => {
    router.events.off('routeChangeError', handleRouteChangeError)
  }, [router]
}

return <Component {...pageProps} />
}
```

The `next/compat/router` export

This is the same `useRouter` hook, but can be used in both `app` and `pages` directories.

It differs from `next/router` in that it does not throw an error when the pages router is not mounted, and instead has a return type of `NextRouter | null`. This allows developers to convert components to support running in both `app` and `pages` as they transition to the `app` router.

A component that previously looked like this:

```
import { useRouter } from 'next/router'
const MyComponent = () => {
  const { isReady, query } = useRouter()
  // ...
}
```

Will error when converted over to `next/compat/router`, as `null` can not be destructured. Instead, developers will be able to take advantage of new hooks:

```
import { useEffect } from 'react'
import { useRouter } from 'next/compat/router'
import { useSearchParams } from 'next/navigation'
const MyComponent = () => {
  const router = useRouter() // may be null or undefined
  const searchParams = useSearchParams()
  useEffect(() => {
    if (router && !router.isReady) {
      return
    }
    // In `app/`, searchParams will be ready
    // `pages/` it will be available after the component renders
    const search = searchParams.get('search')
    // ...
  }, [router, searchParams])
  // ...
}
```

This component will now work in both `pages` and `app` directories. When the component is no longer used in `pages`, you can remove the references to the compat router:

```
import { useSearchParams } from 'next/navigation'
const MyComponent = () => {
  const searchParams = useSearchParams()
  // As this component is only used in `app/`
  const search = searchParams.get('search')
  // ...
}
```

Using `useRouter` outside of Next.js context in pages

Another specific use case is when rendering components outside of a Next.js application context, such as inside `getServerSideProps` on the `pages` directory. In this case, the `compat` router can be used to avoid errors:

```
import { renderToString } from 'react-dom/server'
import { useRouter } from 'next/compat/router'
const MyComponent = () => {
  const router = useRouter() // may be null or undefined
  // ...
}
export async function getServerSideProps() {
  const renderedComponent = renderToString(<MyComponent />)
  return {
    props: {
      renderedComponent,
    },
  }
}
```

Potential ESLint errors

Certain methods accessible on the `router` object return a Promise. If you have the ESLint rule, [no-floating-promises](#) enabled, consider disabling it either globally, or for the affected line.

If your application needs this rule, you should either `void` the promise – or use an `async`

function, `await` the Promise, then void the function call. **This is not applicable when the method is called from inside an `onClick` handler.**

The affected methods are:

- `router.push`
- `router.replace`
- `router.prefetch`

Potential solutions

```
import { useEffect } from 'react'
import { useRouter } from 'next/router'

// Here you would fetch and return the user
const useUser = () => ({ user: null, loading: false })

export default function Page() {
  const { user, loading } = useUser()
  const router = useRouter()

  useEffect(() => {
    // disable the linting on the next line -
    // eslint-disable-next-line no-floating-promises
    router.push('/login')

    // void the Promise returned by router.push
    if (!(user || loading)) {
      void router.push('/login')
    }
    // or use an async function, await the Promise
    async function handleRouteChange() {
      if (!(user || loading)) {
        await router.push('/login')
      }
    }
    void handleRouteChange()
  }, [user, loading])

  return <p>Redirecting...</p>
}
```

withRouter

If `useRouter` is not the best fit for you, `withRouter` can also add the same `router` object to any component.

Usage

```
import { withRouter } from 'next/router'

function Page({ router }) {
  return <p>{router.pathname}</p>
}

export default withRouter(Page)
```

TypeScript

To use class components with `withRouter`, the component needs to accept a router prop:

```
import React from 'react'
import { withRouter, NextRouter } from 'next/'

interface WithRouterProps {
  router: NextRouter
}

interface MyComponentProps extends WithRouterProps

class MyComponent extends React.Component<MyComponentProps> {
  render() {
    return <p>{this.props.router.pathname}</p>
  }
}

export default withRouter(MyComponent)
```

Was this helpful?    



Using Pages Router

Features available in /pages



Latest Version

15.5.4



You are currently viewing the documentation for Pages Router.

userAgent

The `userAgent` helper extends the [Web Request API](#) with additional properties and methods to interact with the user agent object from the request.

middleware.ts

TypeScript



```
import { NextRequest, NextResponse, userAgent }

export function middleware(request: NextRequest) {
  const url = request.nextUrl
  const { device } = userAgent(request)

  // device.type can be: 'mobile', 'tablet',
  // 'wearable', 'embedded', or undefined (for
  const viewport = device.type || 'desktop'

  url.searchParams.set('viewport', viewport)
  return NextResponse.rewrite(url)
}
```

isBot

A boolean indicating whether the request comes from a known bot.

browser

An object containing information about the browser used in the request.

- `name` : A string representing the browser's name, or `undefined` if not identifiable.
 - `version` : A string representing the browser's version, or `undefined`.
-

device

An object containing information about the device used in the request.

- `model` : A string representing the model of the device, or `undefined`.
 - `type` : A string representing the type of the device, such as `console`, `mobile`, `tablet`, `smarttv`, `wearable`, `embedded`, or `undefined`.
 - `vendor` : A string representing the vendor of the device, or `undefined`.
-

engine

An object containing information about the browser's engine.

- `name` : A string representing the engine's name. Possible values include: `Amaya`, `Blink`, `EdgeHTML`, `Flow`, `Gecko`, `Goanna`, `iCab`, `KHTML`, `Links`, `Lynx`, `NetFront`, `NetSurf`, `Presto`, `Tasman`, `Trident`, `w3m`, `WebKit` or `undefined`.

- `version` : A string representing the engine's version, or `undefined`.
-

OS

An object containing information about the operating system.

- `name` : A string representing the name of the OS, or `undefined`.
 - `version` : A string representing the version of the OS, or `undefined`.
-

CPU

An object containing information about the CPU architecture.

- `architecture` : A string representing the architecture of the CPU. Possible values include: `68k`, `amd64`, `arm`, `arm64`, `armhf`, `avr`, `ia32`, `ia64`, `irix`, `irix64`, `mips`, `mips64`, `pa-risc`, `ppc`, `sparc`, `sparc64` or `undefined`

Was this helpful?



[Copy page](#)

Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for
Pages Router.

Latest Version

15.5.4



Configuration

next.config.j...

Learn about the options available in next.config.js f...

TypeScript

Next.js provides a TypeScript-first development...

ESLint

Next.js reports ESLint errors and warnings during...

Was this helpful?    

 **Using Pages Router**
Features available in /pages

 **Latest Version**
15.5.4

 You are currently viewing the documentation for Pages Router.

next.config.js Options

Next.js can be configured through a

`next.config.js` file in the root of your project directory (for example, by `package.json`) with a default export.

 `next.config.js`



```
// @ts-check

/** @type {import('next').NextConfig} */
const nextConfig = {
  /* config options here */
}

module.exports = nextConfig
```

ECMAScript Modules

`next.config.js` is a regular Node.js module, not a JSON file. It gets used by the Next.js server and build phases, and it's not included in the browser build.

If you need [ECMAScript modules](#), you can use `next.config.mjs`:



```
// @ts-check

/**
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  /* config options here */
}

export default nextConfig
```

Good to know: `next.config` with the `.cjs`, `.cts`, or `.mts` extensions are currently **not** supported.

Configuration as a Function

You can also use a function:



```
// @ts-check

export default (phase, { defaultConfig }) =>
/*
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  /* config options here */
}
return nextConfig
}
```

Async Configuration

Since Next.js 12.1.0, you can use an async function:



```
// @ts-check
```

```
module.exports = async (phase, { defaultConfig }) => {
  /* @type {import('next').NextConfig} */
  const nextConfig = {
    /* config options here */
  }
  return nextConfig
}
```

Phase

`phase` is the current context in which the configuration is loaded. You can see the [available phases ↗](#). Phases can be imported from `next/constants`:

```
next.config.js
```

```
// @ts-check

const { PHASE_DEVELOPMENT_SERVER } = require(
  'next/constants'
)

module.exports = (phase, { defaultConfig }) => {
  if (phase === PHASE_DEVELOPMENT_SERVER) {
    return {
      /* development only config options here */
    }
  }

  return {
    /* config options for all phases except development */
  }
}
```

TypeScript

If you are using TypeScript in your project, you can use `next.config.ts` to use TypeScript in your configuration:

```
next.config.ts
```

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  /* config options here */
}

export default nextConfig
```

The commented lines are the place where you can put the configs allowed by `next.config.js`, which are [defined in this file ↗](#).

However, none of the configs are required, and it's not necessary to understand what each config does. Instead, search for the features you need to enable or modify in this section and they will show you what to do.

Avoid using new JavaScript features not available in your target Node.js version. `next.config.js` will not be parsed by Webpack or Babel.

This page documents all the available configuration options:

Unit Testing (experimental)

Starting in Next.js 15.1, the `next/experimental/testing/server` package contains utilities to help unit test `next.config.js` files.

The `unstable_getResponseFromNextConfig` function runs the `headers`, `redirects`, and `rewrites` functions from `next.config.js` with the provided request information and returns `NextResponse` with the results of the routing.

The response from

`unstable_getResponseFromNextConfig` only considers `next.config.js` fields and does not consider middleware or filesystem routes, so the result in production may be different than the unit test.

```
import {
  getRedirectUrl,
  unstable_getResponseFromNextConfig,
} from 'next/experimental/testing/server'

const response = await unstable_getResponseFr
url: 'https://nextjs.org/test',
nextConfig: {
  async redirects() {
    return [{ source: '/test', destination:
    },
    ],
  },
})
expect(response.status).toEqual(307)
expect(getRedirectUrl(response)).toEqual('htt
```

allowedDev...

Use
`'allowedDevOrigins'`
to configure...

assetPrefix

Learn how to use
the `assetPrefix`
config option to...

basePath

Use `'basePath'` to
deploy a Next.js
application under...

bundlePages...

Enable automatic
dependency
bundling for Page...

compress

crossOrigin

Next.js provides gzip compression to compress...

Use the `crossOrigin` option to add a...

devIndicators

Optimized pages include an indicator to let yo...

distDir

Set a custom build directory to use instead of the...

env

Learn to add and access environment...

eslint

Next.js reports ESLint errors and warnings during...

exportPathM...

Customize the pages that will be exported as HTM...

generateBuil...

Configure the build id, which is used to identify...

generateEta...

Next.js will generate etags for every page by...

headers

Add custom HTTP headers to your Next.js app.

httpAgentOp...

Next.js will automatically use HTTP Keep-Alive...

images

Custom configuration for the next/image...

onDemandE...

Configure how Next.js will dispose and keep...

optimizePac...

API Reference for optimizePackageIn Next.js Config...

output

Next.js automatically traces which files...

pageExtensi...

Extend the default page extensions used by Next.js...

poweredByH...

Next.js will add the `x-powered-by` header by default...

productionBr...

Enables browser source map generation during...

reactStrictM...

The complete Next.js runtime is now Strict Mode...

redirects

Add redirects to your Next.js app.

rewrites

Add rewrites to your Next.js app.

Runtime Con...

Add client and server runtime configuration to...

serverExtern...

trailingSlash

Opt-out specific dependencies from the...

Configure Next.js pages to resolve with or without a...

transpilePac...

Automatically transpile and bundle...

turbo

Configure Next.js with Turbopack-specific options

typescript

Next.js reports TypeScript errors by default. Learn...

urlImports

Configure Next.js to allow importing modules from...

useLightning...

Enable experimental support for...

webpack

Learn how to customize the webpack config...

webVitalsAtt...

Learn how to use the webVitalsAttributi...

Was this helpful?    

 **Using Pages Router**
Features available in /pages

 **Latest Version**
15.5.4

(i) You are currently viewing the documentation for Pages Router.

allowedDevOrigins

Next.js does not automatically block cross-origin requests during development, but will block by default in a future major version of Next.js to prevent unauthorized requesting of internal assets/endpoints that are available in development mode.

To configure a Next.js application to allow requests from origins other than the hostname the server was initialized with (`localhost` by default) you can use the `allowedDevOrigins` config option.

`allowedDevOrigins` allows you to set additional origins that can be used in development mode. For example, to use `local-origin.dev` instead of only `localhost`, open `next.config.js` and add the `allowedDevOrigins` config:

 next.config.js 

```
module.exports = {
  allowedDevOrigins: ['local-origin.dev', '*'],
}
```

Was this helpful?





(i) You are currently viewing the documentation for Pages Router.



assetPrefix

Attention: Deploying to Vercel automatically configures a global CDN for your Next.js project. You do not need to manually setup an Asset Prefix.

Good to know: Next.js 9.5+ added support for a customizable [Base Path](#), which is better suited for hosting your application on a sub-path like `/docs`. We do not suggest you use a custom Asset Prefix for this use case.

Set up a CDN

To set up a [CDN](#), you can set up an asset prefix and configure your CDN's origin to resolve to the domain that Next.js is hosted on.

Open `next.config.mjs` and add the `assetPrefix` config based on the [phase](#):

next.config.mjs

```
// @ts-check
import { PHASE_DEVELOPMENT_SERVER } from 'nex

export default (phase) => {
  const isDev = phase === PHASE_DEVELOPMENT_S
  /* 
   * @type {import('next').NextConfig}
  */
```

```
 */
const nextConfig = {
  assetPrefix: isDev ? undefined : 'https://'
}
return nextConfig
}
```

Next.js will automatically use your asset prefix for the JavaScript and CSS files it loads from the `/_next/` path (`.next/static/` folder). For example, with the above configuration, the following request for a JS chunk:

```
/_next/static/chunks/4b9b41aaa062cbbfeff4add70f2
```

Would instead become:

```
https://cdn.mydomain.com/_next/static/chunks/4b9b41aaa062cbbfeff4add70f2
```

The exact configuration for uploading your files to a given CDN will depend on your CDN of choice.

The only folder you need to host on your CDN is the contents of `.next/static/`, which should be uploaded as `_next/static/` as the above URL request indicates. **Do not upload the rest of your `.next/` folder**, as you should not expose your server code and other configuration to the public.

While `assetPrefix` covers requests to `_next/static`, it does not influence the following paths:

- Files in the `public` folder; if you want to serve those assets over a CDN, you'll have to introduce the prefix yourself
- `/_next/data/` requests for `getServerSideProps` pages. These requests will always be made against the main domain since they're not static.

- `/_next/data/` requests for `getStaticProps` pages. These requests will always be made against the main domain to support [Incremental Static Generation](#), even if you're not using it (for consistency).

Was this helpful?    

Using Pages Router

Features available in /pages

Latest Version

15.5.4

You are currently viewing the documentation for Pages Router.

basePath

To deploy a Next.js application under a sub-path of a domain you can use the `basePath` config option.

`basePath` allows you to set a path prefix for the application. For example, to use `/docs` instead of `''` (an empty string, the default), open `next.config.js` and add the `basePath` config:

next.config.js

```
module.exports = {  
  basePath: '/docs',  
}
```

Good to know: This value must be set at build time and cannot be changed without re-building as the value is inlined in the client-side bundles.

Links

When linking to other pages using `next/link` and `next/router` the `basePath` will be automatically applied.

For example, using `/about` will automatically become `/docs/about` when `basePath` is set to `/docs`.

```
export default function HomePage() {
  return (
    <>
      <Link href="/about">About Page</Link>
    </>
  )
}
```

Output html:

```
<a href="/docs/about">About Page</a>
```

This makes sure that you don't have to change all links in your application when changing the `basePath` value.

Images

When using the `next/image` component, you will need to add the `basePath` in front of `src`.

For example, using `/docs/me.png` will properly serve your image when `basePath` is set to `/docs`.

```
import Image from 'next/image'

function Home() {
  return (
    <>
      <h1>My Homepage</h1>
      <Image
        src="/docs/me.png"
        alt="Picture of the author"
        width={500}
        height={500}
      />
      <p>Welcome to my homepage!</p>
    </>
  )
}

export default Home
```


 **Using Pages Router**
Features available in /pages

 **Latest Version**
15.5.4

(i) You are currently viewing the documentation for Pages Router.

bundlePagesRouter-Dependencies

Enable automatic server-side dependency bundling for Pages Router applications. Matches the automatic dependency bundling in App Router.

 next.config.js 

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  bundlePagesRouterDependencies: true,
}

module.exports = nextConfig
```

Explicitly opt-out certain packages from being bundled using the `serverExternalPackages` option.

Version History

Version	Changes
v15.0.0	Moved from experimental to stable. Renamed from <code>bundlePagesExternals</code> to <code>bundlePagesRouterDependencies</code>

v15.0.0 Moved from experimental to stable. Renamed from `bundlePagesExternals` to `bundlePagesRouterDependencies`

Was this helpful?



 **Using Pages Router**
Features available in /pages

 **Latest Version**
15.5.4

(i) You are currently viewing the documentation for Pages Router.

compress

By default, Next.js uses `gzip` to compress rendered content and static files when using `next start` or a custom server. This is an optimization for applications that do not have compression configured. If compression is *already* configured in your application via a custom server, Next.js will not add compression.

You can check if compression is enabled and which algorithm is used by looking at the `Accept-Encoding` ↗ (browser accepted options) and `Content-Encoding` ↗ (currently used) headers in the response.

Disabling compression

To disable **compression**, set the `compress` config option to `false`:

 next.config.js



```
module.exports = {
  compress: false,
}
```

We do not recommend disabling compression unless you have compression configured on your server, as compression reduces bandwidth usage and improves the performance of your application. For example, you're using [nginx](#) and want to switch to `brotli`, set the `compress` option to `false` to allow nginx to handle compression.

Was this helpful?    

 Using Pages Router
Features available in /pages

 Latest Version
15.5.4

(i) You are currently viewing the documentation for Pages Router.

crossOrigin

Use the `crossOrigin` option to add a `crossOrigin` attribute [↗](#) in all `<script>` tags generated by the `next/script` and `next/head` components, and define how cross-origin requests should be handled.

 next.config.js



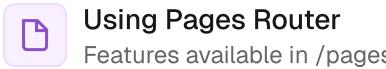
```
module.exports = {  
  crossOrigin: 'anonymous',  
}
```

Options

- `'anonymous'` : Adds `crossOrigin="anonymous"` [↗](#) attribute.
- `'use-credentials'` : Adds `crossOrigin="use-credentials"` [↗](#).

Was this helpful?





ⓘ You are currently viewing the documentation for Pages Router.

devIndicators

`devIndicators` allows you to configure the on-screen indicator that gives context about the current route you're viewing during development.

```
Types
```

```
devIndicators: false | {
  position?: 'bottom-right'
  | 'bottom-left'
  | 'top-right'
  | 'top-left', // defaults to 'bottom-left'
},
```

Setting `devIndicators` to `false` will hide the indicator, however Next.js will continue to surface any build or runtime errors that were encountered.

Troubleshooting

Indicator not marking a route as static

If you expect a route to be static and the indicator has marked it as dynamic, it's likely the route has opted out of static rendering.

You can confirm if a route is [static](#) or [dynamic](#) by building your application using

`next build --debug`, and checking the output in your terminal. Static (or prerendered) routes will display a `o` symbol, whereas dynamic routes will display a `f` symbol. For example:

Route (app)	Size
<code>o /_not-found</code>	<code>0 B</code>
<code>f /products/[id]</code>	<code>0 B</code>

`o` (Static) prerendered as static content
`f` (Dynamic) server-rendered on demand

When exporting `getServerSideProps` or `getInitialProps` from a page, it will be marked as dynamic.

Version History

Version	Changes
v15.2.0	Improved on-screen indicator with new <code>position</code> option. <code>appIsrStatus</code> , <code>buildActivity</code> , and <code>buildActivityPosition</code> options have been deprecated.
v15.0.0	Static on-screen indicator added with <code>appIsrStatus</code> option.

Was this helpful?



Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



distDir

You can specify a name to use for a custom build directory to use instead of `.next`.

Open `next.config.js` and add the `distDir` config:

JS next.config.js



```
module.exports = {  
  distDir: 'build',  
}
```

Now if you run `next build` Next.js will use `build` instead of the default `.next` folder.

`distDir` **should not** leave your project directory. For example, `../build` is an **invalid** directory.

Was this helpful?





Using Pages Router

Features available in /pages



ⓘ You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



env

Since the release of [Next.js 9.4](#) we now have a more intuitive and ergonomic experience for [adding environment variables](#). Give it a try!

Good to know: environment variables specified in this way will **always** be included in the JavaScript bundle, prefixing the environment variable name with `NEXT_PUBLIC_` only has an effect when specifying them [through the environment or .env files](#).

To add environment variables to the JavaScript bundle, open `next.config.js` and add the `env` config:

js next.config.js



```
module.exports = {
  env: {
    customKey: 'my-value',
  },
}
```

Now you can access `process.env.customKey` in your code. For example:

```
function Page() {  
    return <h1>The value of customKey is: {process.env.customKey}  
}  
  
export default Page
```

Next.js will replace `process.env.customKey` with `'my-value'` at build time. Trying to destructure `process.env` variables won't work due to the nature of webpack [DefinePlugin ↗](#).

For example, the following line:

```
return <h1>The value of customKey is: {process.env.customKey}
```

Will end up being:

```
return <h1>The value of customKey is: {'my-value'}
```

Was this helpful?    



(i) You are currently viewing the documentation for Pages Router.



eslint

When ESLint is detected in your project, Next.js fails your **production build** (`next build`) when errors are present.

If you'd like Next.js to produce production code even when your application has ESLint errors, you can disable the built-in linting step completely. This is not recommended unless you already have ESLint configured to run in a separate part of your workflow (for example, in CI or a pre-commit hook).

Open `next.config.js` and enable the `ignoreDuringBuilds` option in the `eslint` config:

```
JS next.config.js 
```

```
module.exports = {
  eslint: {
    // Warning: This allows production builds
    // your project has ESLint errors.
    ignoreDuringBuilds: true,
  },
}
```

Was this helpful?





Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



exportPathMap

This feature is exclusive to `next export` and currently **deprecated** in favor of `getStaticPaths` with `pages` or `generateStaticParams` with `app`.

`exportPathMap` allows you to specify a mapping of request paths to page destinations, to be used during export. Paths defined in `exportPathMap` will also be available when using `next dev`.

Let's start with an example, to create a custom `exportPathMap` for an app with the following pages:

- `pages/index.js`
- `pages/about.js`
- `pages/post.js`

Open `next.config.js` and add the following `exportPathMap` config:

`JS` next.config.js

```
module.exports = {
  exportPathMap: async function (
    defaultPathMap,
    { dev, dir, outDir, distDir, buildId }
  ) {
    return {
      '/': { page: '/' },
      '/about': { page: '/about' },
      '/p/hello-nextjs': { page: '/post', que
```

```
'/p/learn-nextjs': { page: '/post', query: {} },
'/p/deploy-nextjs': { page: '/post', query: {} },
},
}
```

Good to know: the `query` field in `exportPathMap` cannot be used with [automatically statically optimized pages](#) or [getStaticProps pages](#) as they are rendered to HTML files at build-time and additional query information cannot be provided during `next export`.

The pages will then be exported as HTML files, for example, `/about` will become `/about.html`.

`exportPathMap` is an `async` function that receives 2 arguments: the first one is `defaultPathMap`, which is the default map used by Next.js. The second argument is an object with:

- `dev` - `true` when `exportPathMap` is being called in development. `false` when running `next export`. In development `exportPathMap` is used to define routes.
- `dir` - Absolute path to the project directory
- `outDir` - Absolute path to the `out/` directory ([configurable with `-o`](#)). When `dev` is `true` the value of `outDir` will be `null`.
- `distDir` - Absolute path to the `.next/` directory ([configurable with the `distDir` config](#))
- `buildId` - The generated build id

The returned object is a map of pages where the `key` is the `pathname` and the `value` is an object that accepts the following fields:

- `page : String` - the page inside the `pages` directory to render

- `query`: Object - the `query` object passed to `getInitialProps` when prerendering. Defaults to `{}`

The exported `pathname` can also be a filename (for example, `/readme.md`), but you may need to set the `Content-Type` header to `text/html` when serving its content if it is different than `.html`.

Adding a trailing slash

It is possible to configure Next.js to export pages as `index.html` files and require trailing slashes, `/about` becomes `/about/index.html` and is routable via `/about/`. This was the default behavior prior to Next.js 9.

To switch back and add a trailing slash, open `next.config.js` and enable the `trailingSlash` config:

```
js next.config.js
```

```
module.exports = {
  trailingSlash: true,
}
```

Customizing the output directory

`next export` will use `out` as the default output directory, you can customize this using the `-o` argument, like so:

```
>_ Terminal
```

```
next export -o outdir
```

Warning: Using `exportPathMap` is deprecated and is overridden by `getStaticPaths` inside `pages`. We don't recommend using them together.

Was this helpful?





(i) You are currently viewing the documentation for Pages Router.



generateBuildId

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`:

`JS` next.config.js

```
module.exports = {
  generateBuildId: async () => {
    // This could be anything, using the late
    return process.env.GIT_HASH
  },
}
```

Was this helpful?



(i) You are currently viewing the documentation for Pages Router.



generateEtags

Next.js will generate [etags](#) for every page by default. You may want to disable etag generation for HTML pages depending on your cache strategy.

Open `next.config.js` and disable the `generateEtags` option:

`JS` `next.config.js`



```
module.exports = {  
  generateEtags: false,  
}
```

Was this helpful?



Using Pages Router
Features available in /pages

Latest Version
15.5.4

(i) You are currently viewing the documentation for Pages Router.

headers

Headers allow you to set custom HTTP headers on the response to an incoming request on a given path.

To set custom HTTP headers you can use the `headers` key in `next.config.js`:

next.config.js

```
module.exports = {
  async headers() {
    return [
      {
        source: '/about',
        headers: [
          {
            key: 'x-custom-header',
            value: 'my custom header value',
          },
          {
            key: 'x-another-custom-header',
            value: 'my other custom header va
          },
        ],
      },
    ]
  }
}
```

`headers` is an `async` function that expects an array to be returned holding objects with `source` and `headers` properties:

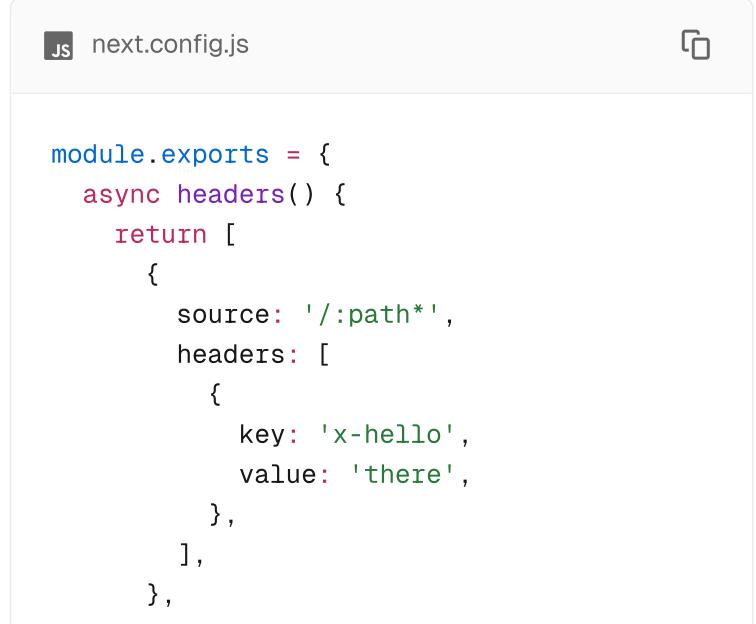
- `source` is the incoming request path pattern.
- `headers` is an array of response header objects, with `key` and `value` properties.
- `basePath: false` or `undefined` - if false the basePath won't be included when matching, can be used for external rewrites only.
- `locale: false` or `undefined` - whether the locale should not be included when matching.
- `has` is an array of `has objects` with the `type`, `key` and `value` properties.
- `missing` is an array of `missing objects` with the `type`, `key` and `value` properties.

Headers are checked before the filesystem which includes pages and `/public` files.

Header Overriding Behavior

If two headers match the same path and set the same header key, the last header key will override the first. Using the below headers, the path

`/hello` will result in the header `x-hello` being `world` due to the last header value set being `world`.



```
JS next.config.js

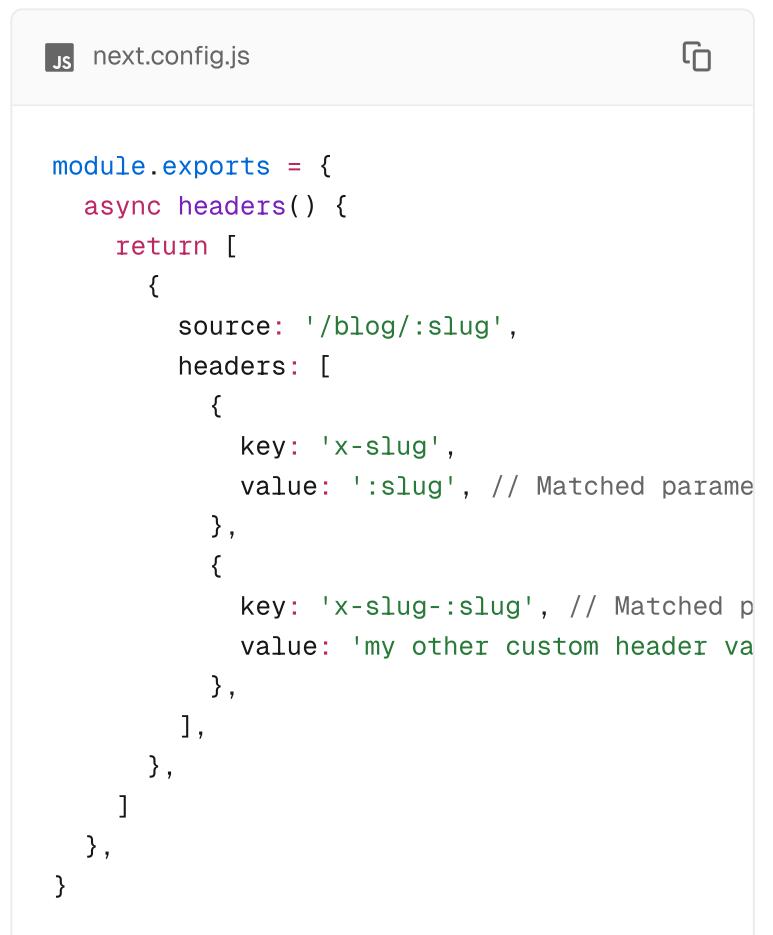
module.exports = {
  async headers() {
    return [
      {
        source: '/:path*',
        headers: [
          {
            key: 'x-hello',
            value: 'there',
          },
        ],
      },
    ],
  },
}
```

```
{  
  source: '/hello',  
  headers: [  
    {  
      key: 'x-hello',  
      value: 'world',  
    },  
  ],  
},  
}
```

Path Matching

Path matches are allowed, for example

`/blog/:slug` will match `/blog/hello-world` (no nested paths):



The screenshot shows a code editor window with a file named `next.config.js`. The code defines an `async headers()` function that returns an array of headers. One header is defined with a path matching pattern `/blog/:slug`, where `:slug` is a placeholder for a dynamic value. The `value` for this header is `:slug`, which is noted as being a matched parameter. Another header is defined with a path matching pattern `x-slug-:slug`, where `:slug` is another placeholder for a dynamic value. The `value` for this header is `'my other custom header value'`.

```
module.exports = {  
  async headers() {  
    return [  
      {  
        source: '/blog/:slug',  
        headers: [  
          {  
            key: 'x-slug',  
            value: ':slug', // Matched parameter  
          },  
          {  
            key: 'x-slug-:slug', // Matched parameter  
            value: 'my other custom header value',  
          },  
        ],  
      },  
    ],  
  },  
}
```

Wildcard Path Matching

To match a wildcard path you can use `*` after a parameter, for example `/blog/:slug*` will match `/blog/a/b/c/d/hello-world`:

```
JS next.config.js

module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:slug*', // Matched param
        headers: [
          {
            key: 'x-slug',
            value: ':slug*', // Matched param
          },
          {
            key: 'x-slug-:slug*', // Matched param
            value: 'my other custom header value',
          },
        ],
      },
    ],
  }
}
```

Regex Path Matching

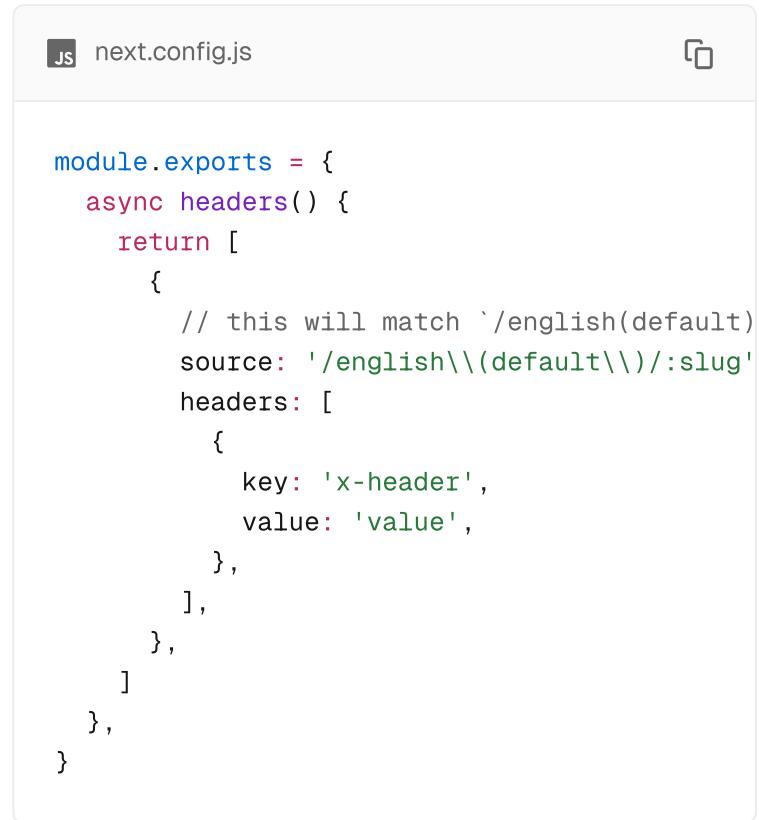
To match a regex path you can wrap the regex in parenthesis after a parameter, for example `/blog/:slug(\d{1,})` will match `/blog/123` but not `/blog/abc`:

```
JS next.config.js

module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:post(\d{1,})',
        headers: [
          {
            key: 'x-post',
            value: ':post',
          },
        ],
      },
    ],
  }
}
```

```
    ]  
},  
}
```

The following characters `(`, `)`, `{`, `}`, `:`, `*`, `+`, `?` are used for regex path matching, so when used in the `source` as non-special values they must be escaped by adding `\`` before them:



A screenshot of a code editor showing a file named `next.config.js`. The code defines an `async headers()` function that returns an array of objects. One object has a `source` field containing a regex pattern `'/english\\((default)\\)/:slug'`. The `headers` field contains an array of objects, each with a `key` of `'x-header'` and a `value` of `'value'`.

```
js next.config.js  
  
module.exports = {  
  async headers() {  
    return [  
      {  
        source: '/english\\((default)\\)/:slug'  
        headers: [  
          {  
            key: 'x-header',  
            value: 'value'  
          }  
        ]  
      }  
    ]  
  }  
}
```

Header, Cookie, and Query Matching

To only apply a header when header, cookie, or query values also match the `has` field or don't match the `missing` field can be used. Both the `source` and all `has` items must match and all `missing` items must not match for the header to be applied.

`has` and `missing` items can have the following fields:

- `type`: `String` - must be either `header`, `cookie`, `host`, or `query`.
- `key`: `String` - the key from the selected type to match against.
- `value`: `String` or `undefined` - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value `first-(?<paramName>.*)` is used for `first-second` then `second` will be usable in the destination with `:paramName`.

```
JS next.config.js

module.exports = {
  async headers() {
    return [
      // if the header `x-add-header` is present
      // the `x-another-header` header will be
      {
        source: '/:path*',
        has: [
          {
            type: 'header',
            key: 'x-add-header',
          },
        ],
        headers: [
          {
            key: 'x-another-header',
            value: 'hello',
          },
        ],
      },
      // if the header `x-no-header` is not present
      // the `x-another-header` header will be
      {
        source: '/:path*',
        missing: [
          {
            type: 'header',
            key: 'x-no-header',
          },
        ],
        headers: [
          {
            key: 'x-another-header',
            value: 'hello',
          },
        ],
      },
    ],
  }
}
```

```
        },
        ],
    },
    // if the source, query, and cookie are
    // the `x-authorized` header will be ap
{
    source: '/specific/:path*',
    has: [
        {
            type: 'query',
            key: 'page',
            // the page value will not be ava
            // header key/values since value
            // doesn't use a named capture gr
            value: 'home',
        },
        {
            type: 'cookie',
            key: 'authorized',
            value: 'true',
        },
    ],
    headers: [
        {
            key: 'x-authorized',
            value: ':authorized',
        },
    ],
},
// if the header `x-authorized` is pres
// contains a matching value, the `x-an
{
    source: '/:path*',
    has: [
        {
            type: 'header',
            key: 'x-authorized',
            value: '(?<authorized>yes|true)',
        },
    ],
    headers: [
        {
            key: 'x-another-header',
            value: ':authorized',
        },
    ],
},
// if the host is `example.com`,
// this header will be applied
{
    source: '/:path*',
    has: [
        {
            type: 'host',
            value: 'example.com',
        },
    ],
}
```

```
,  
],  
headers: [  
  {  
    key: 'x-another-header',  
    value: ':authorized',  
  },  
],  
},  
}  
}
```

Headers with basePath support

When leveraging `basePath` support with headers each `source` is automatically prefixed with the `basePath` unless you add `basePath: false` to the header:

```
js next.config.js □  
  
module.exports = {  
  basePath: '/docs',  
  
  async headers() {  
    return [  
      {  
        source: '/with-basePath', // becomes  
        headers: [  
          {  
            key: 'x-hello',  
            value: 'world',  
          },  
        ],  
      },  
      {  
        source: '/without-basePath', // is no  
        headers: [  
          {  
            key: 'x-hello',  
            value: 'world',  
          },  
        ],  
        basePath: false,  
      },  
    ]  
  },  
}
```

```
    },
  ],
},
}
```

Headers with i18n support

When leveraging `i18n support` with headers each `source` is automatically prefixed to handle the configured `locales` unless you add `locale: false` to the header. If `locale: false` is used you must prefix the `source` with a locale for it to be matched correctly.

```
JS next.config.js
```

```
module.exports = {
  i18n: {
    locales: ['en', 'fr', 'de'],
    defaultLocale: 'en',
  },
  async headers() {
    return [
      {
        source: '/with-locale', // automatica
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // does not handle locales automatica
        source: '/nl/with-locale-manual',
        locale: false,
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // this matches '' since 'en' is the
      },
    ],
  },
}
```

```
source: '/en',
locale: false,
headers: [
  {
    key: 'x-hello',
    value: 'world',
  },
],
{
  // this gets converted to /(en|fr|de)
  // `/` or `/fr` routes like /:path*
  source: '/(.*)',
  headers: [
    {
      key: 'x-hello',
      value: 'world',
    },
  ],
},
]
```

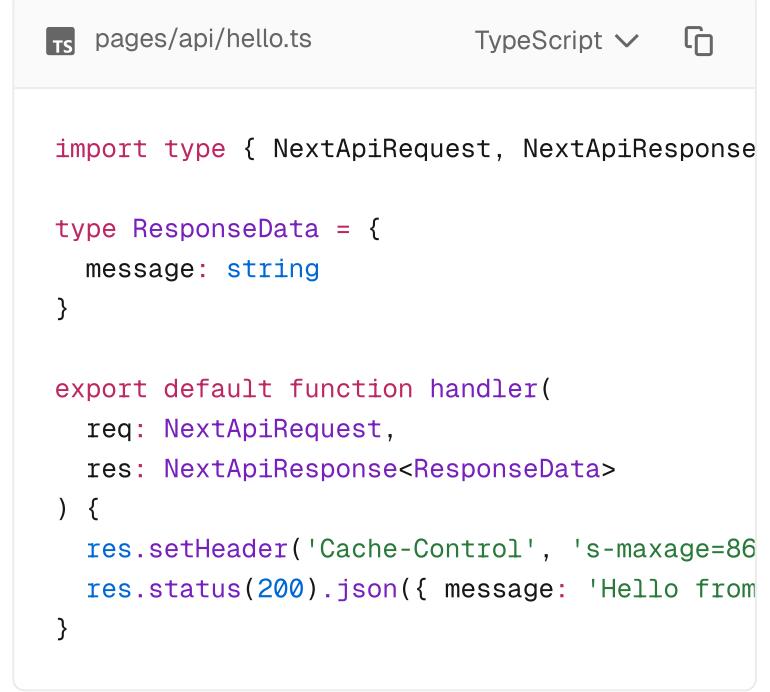
Cache-Control

Next.js sets the `Cache-Control` header of `public, max-age=31536000, immutable` for 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 cannot set `Cache-Control` headers in `next.config.js` for these assets.

However, you can set `Cache-Control` headers for other responses or data.

If you need to revalidate the cache of a page that has been [statically generated](#), you can do so by setting the `revalidate` prop in the page's `getStaticProps` function.

To cache the response from an [API Route](#), you can use `res.setHeader`:



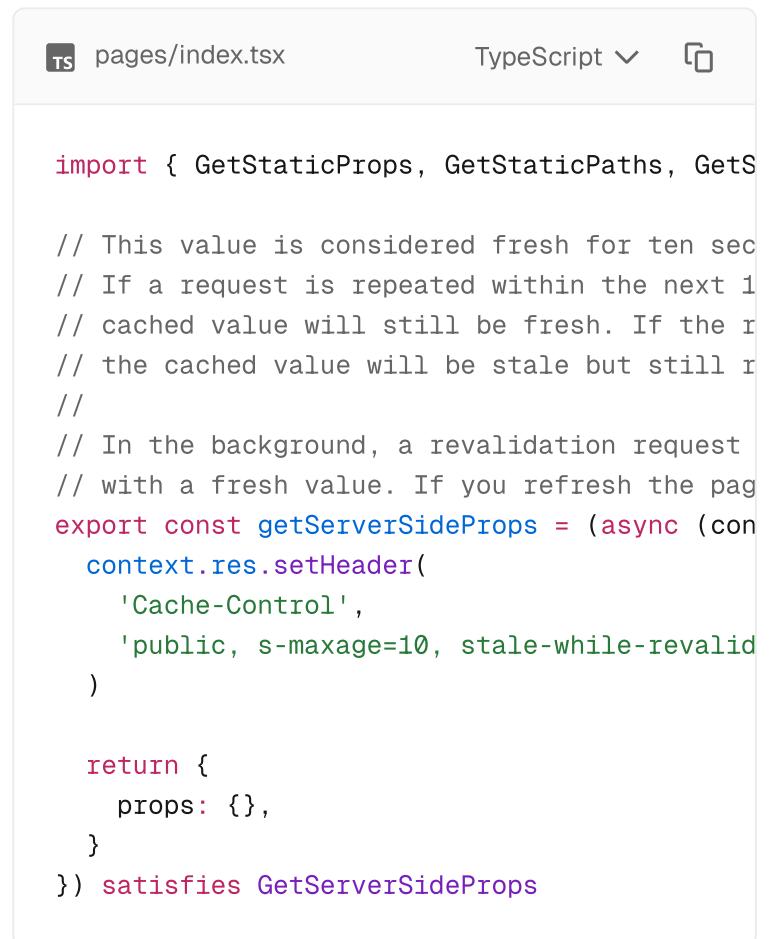
pages/api/hello.ts TypeScript

```
import type { NextApiRequest, NextApiResponse } from 'next'

type ResponseData = {
  message: string
}

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<ResponseData>
) {
  res.setHeader('Cache-Control', 's-maxage=86400')
  res.status(200).json({ message: 'Hello from Next.js' })
}
```

You can also use caching headers (`Cache-Control`) inside `getServerSideProps` to cache dynamic responses. For example, using `stale-while-revalidate`.



pages/index.tsx TypeScript

```
import { GetStaticProps, GetStaticPaths, GetServerSideProps } from 'next'

// This value is considered fresh for ten seconds
// If a request is repeated within the next 10 seconds
// cached value will still be fresh. If the request
// is made after 10 seconds
// the cached value will be stale but still returned
// In the background, a revalidation request
// will be triggered with a fresh value. If you refresh the page
// again, it will receive a fresh value
export const getServerSideProps = (async (context) => {
  context.res.setHeader(
    'Cache-Control',
    'public, s-maxage=10, stale-while-revalidate'
  )

  return {
    props: {},
  }
}) satisfies GetServerSideProps
```

Options

CORS

[Cross-Origin Resource Sharing \(CORS\)](#) ↗ is a security feature that allows you to control which sites can access your resources. You can set the `Access-Control-Allow-Origin` header to allow a specific origin to access your API Endpoints.

```
async headers() {
  return [
    {
      source: "/api/:path*",  

      headers: [
        {
          key: "Access-Control-Allow-Origin",
          value: "*", // Set your origin
        },
        {
          key: "Access-Control-Allow-Methods",
          value: "GET, POST, PUT, DELETE, OPTIONS"
        },
        {
          key: "Access-Control-Allow-Headers",
          value: "Content-Type, Authorization"
        },
      ],
    },
  ];
},
```

X-DNS-Prefetch-Control

This [header](#) ↗ controls DNS prefetching, allowing browsers to proactively perform domain name resolution on external links, images, CSS, JavaScript, and more. This prefetching is performed in the background, so the [DNS](#) ↗ is more likely to be resolved by the time the referenced items are needed. This reduces latency when the user clicks a link.

```
{  
  key: 'X-DNS-Prefetch-Control',  
  value: 'on'  
}
```

Strict-Transport-Security

This header [↗](#) informs browsers it should only be accessed using HTTPS, instead of using HTTP. Using the configuration below, all present and future subdomains will use HTTPS for a `max-age` of 2 years. This blocks access to pages or subdomains that can only be served over HTTP.

```
{  
  key: 'Strict-Transport-Security',  
  value: 'max-age=63072000; includeSubDomains'  
}
```

X-Frame-Options

This header [↗](#) indicates whether the site should be allowed to be displayed within an `iframe`. This can prevent against clickjacking attacks.

This header has been superseded by CSP's `frame-ancestors` option, which has better support in modern browsers (see [Content Security Policy](#) for configuration details).

```
{  
  key: 'X-Frame-Options',  
  value: 'SAMEORIGIN'  
}
```

Permissions-Policy

This header [↗](#) allows you to control which features and APIs can be used in the browser. It was previously named `Feature-Policy`.

```
{  
  key: 'Permissions-Policy',  
  value: 'camera=(), microphone=(), geolocation=()'}
```

X-Content-Type-Options

This header [↗](#) prevents the browser from attempting to guess the type of content if the `Content-Type` header is not explicitly set. This can prevent XSS exploits for websites that allow users to upload and share files.

For example, a user trying to download an image, but having it treated as a different `Content-Type` like an executable, which could be malicious. This header also applies to downloading browser extensions. The only valid value for this header is `nosniff`.

```
{  
  key: 'X-Content-Type-Options',  
  value: 'nosniff'}
```

Referrer-Policy

This header [↗](#) controls how much information the browser includes when navigating from the current website (origin) to another.

```
{  
  key: 'Referrer-Policy',  
  value: 'origin-when-cross-origin'}
```

Content-Security-Policy

Learn more about adding a [Content Security Policy](#) to your application.

Version History

Version	Changes
v13.3.0	missing added.
v10.2.0	has added.
v9.5.0	Headers added.

Was this helpful?    



(i) You are currently viewing the documentation for Pages Router.



httpAgentOptions

In Node.js versions prior to 18, Next.js automatically polyfills `fetch()` with [undici](#) and enables [HTTP Keep-Alive ↗](#) by default.

To disable HTTP Keep-Alive for all `fetch()` calls on the server-side, open `next.config.js` and add the `httpAgentOptions` config:

```
JS next.config.js 
```

```
module.exports = {
  httpAgentOptions: {
    keepAlive: false,
  },
}
```

Was this helpful?

Using Pages Router
Features available in /pages

Latest Version
15.5.4

You are currently viewing the documentation for Pages Router.

images

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure `next.config.js` with the following:

```
JS next.config.js   
  
module.exports = {  
  images: {  
    loader: 'custom',  
    loaderFile: './my/image/loader.js',  
  },  
}
```

This `loaderFile` must point to a file relative to the root of your Next.js application. The file must export a default function that returns a string, for example:

```
JS my/image/loader.js   
  
export default function myImageLoader({ src,  
  return `https://example.com/${src}?w=${width}`  
}
```

Alternatively, you can use the `loader` prop to pass the function to each instance of `next/image`

To learn more about configuring the behavior of the built-in [Image Optimization API](#) and the [Image Component](#), see [Image Configuration Options](#) for available options.

Example Loader Configuration

- [Akamai](#)
- [AWS CloudFront](#)
- [Cloudinary](#)
- [Cloudflare](#)
- [Contentful](#)
- [Fastly](#)
- [Gumlet](#)
- [ImageEngine](#)
- [Imgix](#)
- [PixelBin](#)
- [Sanity](#)
- [Sirv](#)
- [Supabase](#)
- [Thumbor](#)
- [Imagekit](#)
- [Nitrogen AIO](#)

Akamai

```
// Docs: https://techdocs.akamai.com/ivm/referencedata/loader.html#akamai
export default function akamaiLoader({ src, width }) {
  return `https://example.com/${src}?imwidth=${width}`;
}
```

AWS CloudFront

```
// Docs: https://aws.amazon.com/developer/applications/cloudfront-image-loading
export default function cloudfrontLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`);
  url.searchParams.set('format', 'auto');
  url.searchParams.set('width', width.toString());
  url.searchParams.set('quality', (quality || 75).toString());
  return url.href;
}
```

Cloudinary

```
// Demo: https://res.cloudinary.com/demo/images/f_auto,c_limit,w_500
export default function cloudinaryLoader({ src, width, quality }) {
  const params = ['f_auto', 'c_limit', `w_${width}`];
  if (quality) params.push(`q_${quality}`);
  return `https://example.com/${params.join('/')}`;
}
```

Cloudflare

```
// Docs: https://developers.cloudflare.com/image-delivery/javascript-image-loading
export default function cloudflareLoader({ src, width, quality }) {
  const params = [`width=${width}`, `quality=${quality}`];
  return `https://example.com/cdn-cgi/image/${src}?${params.join('&')}`;
}
```

Contentful

```
// Docs: https://www.contentful.com/developers/docs/references/javascript-image-loading
export default function contentfulLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`);
  url.searchParams.set('fm', 'webp');
  url.searchParams.set('w', width.toString());
  url.searchParams.set('q', (quality || 75).toString());
  return url.href;
}
```

Fastly

```
// Docs: https://developer.fastly.com/reference/javascript-image-loading
export default function fastlyLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`);
  url.searchParams.set('fm', 'auto');
  url.searchParams.set('w', width.toString());
  url.searchParams.set('q', (quality || 75).toString());
  return url.href;
}
```

```
urlSearchParams.set('auto', 'webp')
urlSearchParams.set('width', width.toString())
urlSearchParams.set('quality', (quality || 75).toString())
return url.href
}
```

Gumlet

```
// Docs: https://docs.gumlet.com/reference/images/gumlet-loader
export default function gumletLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  urlSearchParams.set('format', 'auto')
  urlSearchParams.set('w', width.toString())
  urlSearchParams.set('q', (quality || 75).toString())
  return url.href
}
```

ImageEngine

```
// Docs: https://support.imageengine.io/hc/en-us/articles/360000000000
export default function imageengineLoader({ src, width, quality }) {
  const compression = 100 - (quality || 50)
  const params = [ `w_${width}` , `cmpr_${compression}` ]
  return `https://example.com${src}?imgeng=/${params.join('&')}`
}
```

Imgix

```
// Demo: https://static.imgix.net/daisy.png?fit=auto&w=500
export default function imgixLoader({ src, width, height, quality }) {
  const url = new URL(`https://example.com${src}`)
  const params = urlSearchParams
  params.set('auto', params.getAll('auto').join(','))
  params.set('fit', params.get('fit') || 'max')
  params.set('w', params.get('w') || width.toString())
  params.set('q', (quality || 50).toString())
  return url.href
}
```

PixelBin

```
// Doc (Resize): https://www.pixelbin.io/docs
```

```
// Doc (Optimise): https://www.pixelbin.io/do
// Doc (Auto Format Delivery): https://www.pi
export default function pixelBinLoader({ src,
  const name = '<your-cloud-name>'
  const opt = `t.resize(w:${width})~t.compress
return `https://cdn.pixelbin.io/v2/${name}/
}
```

Sanity

```
// Docs: https://www.sanity.io/docs/image-url
export default function sanityLoader({ src, w
  const prj = 'zp7mbokg'
  const dataset = 'production'
  const url = new URL(`https://cdn.sanity.io/
urlSearchParams.set('auto', 'format')
urlSearchParams.set('fit', 'max')
urlSearchParams.set('w', width.toString())
if (quality) {
  urlSearchParams.set('q', quality.toStrin
}
return url.href
}
```

Sirv

```
// Docs: https://sirv.com/help/articles/dynam
export default function sirvLoader({ src, wid
  const url = new URL(`https://example.com${s
  const params = url.searchParams
  params.set('format', params.getAll('format')
  params.set('w', params.get('w') || width.to
  params.set('q', (quality || 85).toString())
  return url.href
}
```

Supabase

```
// Docs: https://supabase.com/docs/guides/sto
export default function supabaseLoader({ src,
  const url = new URL(`https://example.com${s
  url.searchParams.set('width', width.toString()
  url.searchParams.set('quality', (quality || 1
  return url.href
})
```

Thumbor

```
// Docs: https://thumbor.readthedocs.io/en/la
export default function thumborLoader({ src,
  const params = [`w-${width}x0`, `filters:qual
  return `https://example.com${params.join('/')
})
```

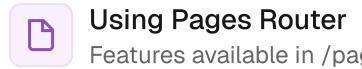
ImageKit.io

```
// Docs: https://imagekit.io/docs/image-trans
export default function imageKitLoader({ src,
  const params = [`w-${width}`, `q-${quality}
  return `https://ik.imagekit.io/your_imageki
})
```

Nitrogen AIO

```
// Docs: https://docs.n7.io/aio/integrations
export default function aioLoader({ src, width
  const url = new URL(src, window.location.href)
  const params = url.searchParams
  const aioParams = params.getAll('aio')
  aioParams.push(`w-${width}`)
  if (quality) {
    aioParams.push(`q-${quality.toString()}`)
  }
  params.set('aio', aioParams.join(';'))
  return url.href
})
```

Was this helpful?    



Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



onDemandEntries

Next.js exposes some options that give you some control over how the server will dispose or keep in memory built pages in development.

To change the defaults, open `next.config.js` and add the `onDemandEntries` config:

`JS` next.config.js

```
module.exports = {
  onDemandEntries: {
    // period (in ms) where the server will k
    maxInactiveAge: 25 * 1000,
    // number of pages that should be kept si
    pagesBufferLength: 2,
  },
}
```

Was this helpful?



Using Pages Router
Features available in /pages

Latest Version
15.5.4

You are currently viewing the documentation for Pages Router.

optimizePackageImports

Some packages can export hundreds or thousands of modules, which can cause performance issues in development and production.

Adding a package to

`experimental.optimizePackageImports` will only load the modules you are actually using, while still giving you the convenience of writing import statements with many named exports.

```
JS next.config.js 
```

```
module.exports = {
  experimental: {
    optimizePackageImports: ['package-name'],
  },
}
```

The following libraries are optimized by default:

- `lucide-react`
- `date-fns`
- `lodash-es`
- `ramda`
- `antd`

- react-bootstrap
- ahooks
- @ant-design/icons
- @headlessui/react
- @headlessui-float/react
- @heroicons/react/20/solid
- @heroicons/react/24/solid
- @heroicons/react/24/outline
- @visx/visx
- @tremor/react
- rxjs
- @mui/material
- @mui/icons-material
- recharts
- react-use
- @material-ui/core
- @material-ui/icons
- @tabler/icons-react
- mui-core
- react-icons/*
- effect
- @effect/*

Was this helpful?    



Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



output

During a build, Next.js will automatically trace each page and its dependencies to determine all of the files that are needed for deploying a production version of your application.

This feature helps reduce the size of deployments drastically. Previously, when deploying with Docker you would need to have all files from your package's `dependencies` installed to run `next start`. Starting with Next.js 12, you can leverage Output File Tracing in the `.next/` directory to only include the necessary files.

Furthermore, this removes the need for the deprecated `serverless` target which can cause various issues and also creates unnecessary duplication.

How it Works

During `next build`, Next.js will use `@vercel/nft` to statically analyze `import`, `require`, and `fs` usage to determine all files that a page might load.

Next.js' production server is also traced for its needed files and output at

`.next/next-server.js.nft.json` which can be leveraged in production.

To leverage the `.nft.json` files emitted to the `.next` output directory, you can read the list of files in each trace that are relative to the `.nft.json` file and then copy them to your deployment location.

Automatically Copying Traced Files

Next.js can automatically create a `standalone` folder that copies only the necessary files for a production deployment including select files in `node_modules`.

To leverage this automatic copying you can enable it in your `next.config.js`:



```
JS next.config.js

module.exports = {
  output: 'standalone',
}
```

This will create a folder at `.next/standalone` which can then be deployed on its own without installing `node_modules`.

Additionally, a minimal `server.js` file is also output which can be used instead of `next start`. This minimal server does not copy the `public` or `.next/static` folders by default as these should ideally be handled by a CDN instead, although these folders can be copied to the `standalone/public` and `standalone/.next/static` folders manually, after

which `server.js` file will serve these automatically.

To copy these manually, you can use the `cp` command-line tool after you `next build`:

```
>_ Terminal
```

```
cp -r public .next/standalone/ && cp -r .next
```

To start your minimal `server.js` file locally, run the following command:

```
>_ Terminal
```

```
node .next/standalone/server.js
```

Good to know:

- `next.config.js` is read during `next build` and serialized into the `server.js` output file. If the legacy `serverRuntimeConfig` or `publicRuntimeConfig` options are being used, the values will be specific to values at build time.
- If your project needs to listen to a specific port or hostname, you can define `PORT` or `HOSTNAME` environment variables before running `server.js`. For example, run `PORT=8080 HOSTNAME=0.0.0.0 node server.js` to start the server on `http://0.0.0.0:8080`.

Caveats

- While tracing in monorepo setups, the project directory is used for tracing by default. For `next build packages/web-app`, `packages/web-app` would be the tracing root and any files outside of that folder will not be included. To include files outside of this folder

you can set `outputFileTracingRoot` in your `next.config.js`.

`JS packages/web-app/next.config.js`

```
const path = require('path')

module.exports = {
  // this includes files from the monorepo ba
  outputFileTracingRoot: path.join(__dirname,
}
```

- There are some cases in which Next.js might fail to include required files, or might incorrectly include unused files. In those cases, you can leverage `outputFileTracingExcludes` and `outputFileTracingIncludes` respectively in `next.config.js`. Each option accepts an object whose keys are **route globss** (matched with [picomatch ↗](#) against the route path, e.g. `/api/hello`) and whose values are **glob patterns resolved from the project root** that specify files to include or exclude in the trace.

`JS next.config.js`

```
module.exports = {
  outputFileTracingExcludes: {
    '/api/hello': ['./un-necessary-folder/**/'],
  },
  outputFileTracingIncludes: {
    '/api/another': ['./necessary-folder/**/*'],
    '/api/login/\\"[\\"[\\\".\\\".\\\".slug\\\"]]\\"]': [
      './node_modules/aws-crt/dist/bin/**/*',
    ],
  },
}
```

Using a `src/` directory does not change how you write these options:

- **Keys** still match the route path (`'/api/hello'`, `'/products/[id]'`, etc.).

- **Values** can reference paths under `src/` since they are resolved relative to the project root.

```
JS next.config.js Copy  
  
module.exports = {  
  outputFileTracingIncludes: {  
    '/products/*': ['src/lib/payments/**/*'],  
    '/**': ['src/config/runtime/**/*.json'],  
  },  
  outputFileTracingExcludes: {  
    '/api/*': ['src/temp/**/*', 'public/large'],  
  },  
}
```

You can also target all routes using a global key like `'/*'`:

```
JS next.config.js Copy  
  
module.exports = {  
  outputFileTracingIncludes: {  
    '/**': ['src/i18n/locales/**/*.json'],  
  },  
}
```

These options are applied to server traces and do not affect routes that do not produce a server trace file:

- Edge Runtime routes are not affected.
- Fully static pages are not affected.

In monorepos or when you need to include files outside the app folder, combine `outputFileTracingRoot` with includes:

```
JS next.config.js Copy  
  
const path = require('path')  
  
module.exports = {  
  // Trace from the monorepo root
```

```
 outputFileTracingRoot: path.join(_dirname,  
 outputFileTracingIncludes: {  
   '/route1': ['./shared/assets/**/*'],  
 },  
 }
```

Good to know:

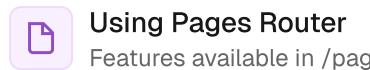
- Prefer forward slashes (/) in patterns for cross-platform compatibility.
- Keep patterns as narrow as possible to avoid oversized traces (avoid **/* at the repo root).

Common include patterns for native/runtime assets:

JS next.config.js

```
module.exports = {  
  outputFileTracingIncludes: {  
    '/**': ['node_modules/sharp/**/*', 'node_m  
  },  
}
```

Was this helpful?     



(i) You are currently viewing the documentation for Pages Router.

pageExtensions

You can extend the default Page extensions (`.tsx`, `.ts`, `.jsx`, `.js`) used by Next.js. Inside `next.config.js`, add the `pageExtensions` config:

`next.config.js`

```
module.exports = {  
  pageExtensions: ['mdx', 'md', 'jsx', 'js',  
} 
```

Changing these values affects *all* Next.js pages, including the following:

- `middleware.js`
- `instrumentation.js`
- `pages/_document.js`
- `pages/_app.js`
- `pages/api/`

For example, if you reconfigure `.ts` page extensions to `.page.ts`, you would need to rename pages like `middleware.page.ts`, `instrumentation.page.ts`, `_app.page.ts`.

Including non-page files in the pages directory

You can colocate test files or other files used by components in the `pages` directory. Inside `next.config.js`, add the `pageExtensions` config:

```
JS next.config.js Copy  
  
module.exports = {  
  pageExtensions: ['page.tsx', 'page.ts', 'pa  
}  
  
}
```

Then, rename your pages to have a file extension that includes `.page` (e.g. rename `MyPage.tsx` to `MyPage.page.tsx`). Ensure you rename *all* Next.js pages, including the files mentioned above.

Was this helpful? 😞 😢 😊 ★★

Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

Latest Version

15.5.4



poweredByHeader

By default Next.js will add the `x-powered-by` header. To opt-out of it, open `next.config.js` and disable the `poweredByHeader` config:

next.config.js



```
module.exports = {  
  poweredByHeader: false,  
}
```

Was this helpful?



 **Using Pages Router**
Features available in /pages

 **Latest Version**
15.5.4

(i) You are currently viewing the documentation for Pages Router.

productionBrowserSourceMaps

Source Maps are enabled by default during development. During production builds, they are disabled to prevent you leaking your source on the client, unless you specifically opt-in with the configuration flag.

Next.js provides a configuration flag you can use to enable browser source map generation during the production build:

 next.config.js

```
module.exports = {
  productionBrowserSourceMaps: true,
}
```

When the `productionBrowserSourceMaps` option is enabled, the source maps will be output in the same directory as the JavaScript files. Next.js will automatically serve these files when requested.

- Adding source maps can increase `next build` time
- Increases memory usage during `next build`

Was this helpful?





(i) You are currently viewing the documentation for Pages Router.

reactStrictMode

Good to know: Since Next.js 13.5.1, Strict Mode is `true` by default with `app` router, so the above configuration is only necessary for `pages`. You can still disable Strict Mode by setting `reactStrictMode: false`.

Suggested: We strongly suggest you enable Strict Mode in your Next.js application to better prepare your application for the future of React.

React's [Strict Mode](#) is a development mode only feature for highlighting potential problems in an application. It helps to identify unsafe lifecycles, legacy API usage, and a number of other features.

The Next.js runtime is Strict Mode-compliant. To opt-in to Strict Mode, configure the following option in your `next.config.js`:

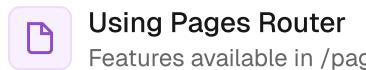
next.config.js

```
module.exports = {
  reactStrictMode: true,
}
```

If you or your team are not ready to use Strict Mode in your entire application, that's OK! You can incrementally migrate on a page-by-page basis using `<React.StrictMode>`.

Was this helpful?





(i) You are currently viewing the documentation for Pages Router.

redirects

Redirects allow you to redirect an incoming request path to a different destination path.

To use redirects you can use the `redirects` key in `next.config.js`:

`JS` next.config.js

```
module.exports = {
  async redirects() {
    return [
      {
        source: '/about',
        destination: '/',
        permanent: true,
      },
    ],
  },
}
```

`redirects` is an `async` function that expects an array to be returned holding objects with `source`, `destination`, and `permanent` properties:

- `source` is the incoming request path pattern.
- `destination` is the path you want to route to.
- `permanent` `true` or `false` - if `true` will use the 308 status code which instructs clients/search engines to cache the redirect

forever, if `false` will use the 307 status code which is temporary and is not cached.

Why does Next.js use 307 and 308? Traditionally a 302 was used for a temporary redirect, and a 301 for a permanent redirect, but many browsers changed the request method of the redirect to `GET`, regardless of the original method. For example, if the browser made a request to `POST /v1/users` which returned status code `302` with location `/v2/users`, the subsequent request might be `GET /v2/users` instead of the expected `POST /v2/users`. Next.js uses the 307 temporary redirect, and 308 permanent redirect status codes to explicitly preserve the request method used.

- `basePath`: `false` or `undefined` - if false the `basePath` won't be included when matching, can be used for external redirects only.
- `locale`: `false` or `undefined` - whether the locale should not be included when matching.
- `has` is an array of `has objects` with the `type`, `key` and `value` properties.
- `missing` is an array of `missing objects` with the `type`, `key` and `value` properties.

Redirects are checked before the filesystem which includes pages and `/public` files.

When using the Pages Router, redirects are not applied to client-side routing (`Link`, `router.push`) unless `Middleware` is present and matches the path.

When a redirect is applied, any query values provided in the request will be passed through to the redirect destination. For example, see the following redirect configuration:

```
{  
  source: '/old-blog/:path*',  
  destination: '/blog/:path*',  
}
```

```
permanent: false  
}
```

Good to know: Remember to include the forward slash / before the colon : in path parameters of the source and destination paths, otherwise the path will be treated as a literal string and you run the risk of causing infinite redirects.

When /old-blog/post-1?hello=world is requested, the client will be redirected to /blog/post-1?hello=world .

Path Matching

Path matches are allowed, for example

/old-blog/:slug will match

/old-blog/hello-world (no nested paths):

next.config.js

```
module.exports = {  
  async redirects() {  
    return [  
      {  
        source: '/old-blog/:slug',  
        destination: '/news/:slug', // Matches  
        permanent: true,  
      },  
    ]  
  },  
}
```

Wildcard Path Matching

To match a wildcard path you can use * after a parameter, for example /blog/:slug* will match /blog/a/b/c/d/hello-world :

next.config.js

```
module.exports = {
  async redirects() {
    return [
      {
        source: '/blog/:slug*',
        destination: '/news/:slug*', // Match
        permanent: true,
      },
    ],
  },
}
```

Regex Path Matching

To match a regex path you can wrap the regex in parentheses after a parameter, for example

/post/:slug(\d{1,}) will match /post/123

but not /post/abc :

next.config.js

```
module.exports = {
  async redirects() {
    return [
      {
        source: '/post/:slug(\d{1,})',
        destination: '/news/:slug', // Match
        permanent: false,
      },
    ],
  },
}
```

The following characters (,), {, }, :, *, +, ?, are used for regex path matching, so when used in the source as non-special values they must be escaped by adding \ before them:

next.config.js

```
module.exports = {
  async redirects() {
    return [
      {
        // this will match `/english(default)`
      },
    ],
  },
}
```

```
        source: '/english\\(default\\)/:slug',
        destination: '/en-us/:slug',
        permanent: false,
      },
    ],
  },
}
```

Header, Cookie, and Query Matching

To only match a redirect when header, cookie, or query values also match the `has` field or don't match the `missing` field can be used. Both the `source` and all `has` items must match and all `missing` items must not match for the redirect to be applied.

`has` and `missing` items can have the following fields:

- `type`: `String` - must be either `header`, `cookie`, `host`, or `query`.
- `key`: `String` - the key from the selected type to match against.
- `value`: `String` or `undefined` - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value `first-(?<paramName>.*)` is used for `first-second` then `second` will be usable in the destination with `:paramName`.

next.config.js

```
module.exports = {
  async redirects() {
    return [
      // if the header `x-redirect-me` is pre
```

```
// this redirect will be applied
{
  source: '/:path((?!another-page$).*)'
  has: [
    {
      type: 'header',
      key: 'x-redirect-me',
    },
  ],
  permanent: false,
  destination: '/another-page',
},
// if the header `x-dont-redirect` is present
// this redirect will NOT be applied
{
  source: '/:path((?!another-page$).*)'
  missing: [
    {
      type: 'header',
      key: 'x-do-not-redirect',
    },
  ],
  permanent: false,
  destination: '/another-page',
},
// if the source, query, and cookie are
// this redirect will be applied
{
  source: '/specific/:path*',
  has: [
    {
      type: 'query',
      key: 'page',
      // the page value will not be available
      // destination since value is provided
      // use a named capture group e.g.
      value: 'home',
    },
    {
      type: 'cookie',
      key: 'authorized',
      value: 'true',
    },
  ],
  permanent: false,
  destination: '/another/:path*',
},
// if the header `x-authorized` is present
// contains a matching value, this redirect
{
  source: '/',
  has: [
    {
      type: 'header',
      key: 'x-authorized',
    },
  ],
}
```

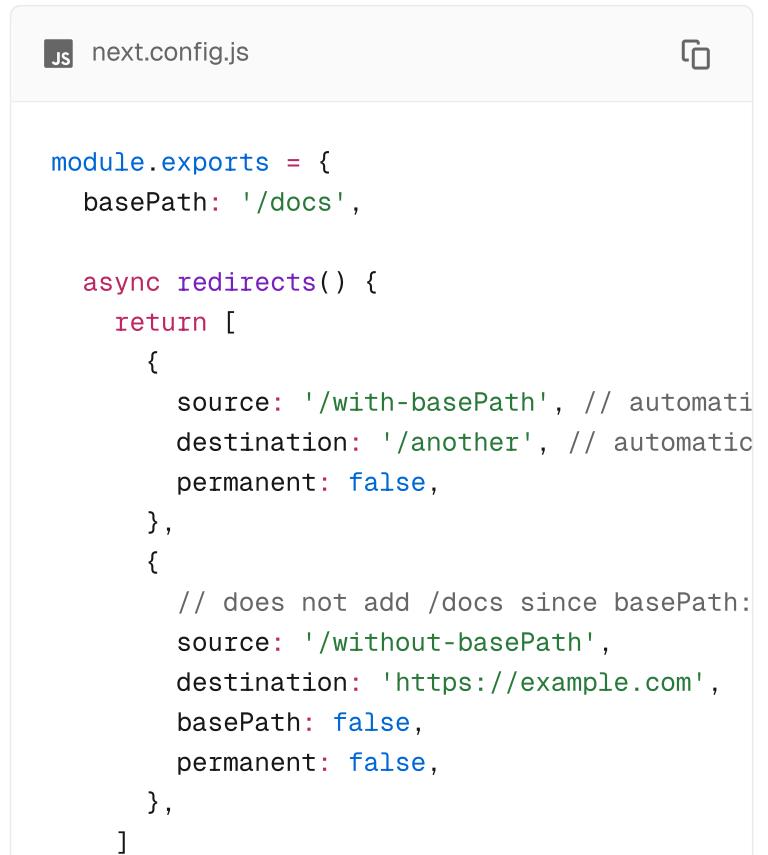
```

        value: '(<authorized>yes|true)',
    },
    ],
    permanent: false,
    destination: '/home?authorized=:autho
},
// if the host is `example.com`,
// this redirect will be applied
{
    source: '/:path((?!another-page$).*)'
    has: [
        {
            type: 'host',
            value: 'example.com',
        },
        ],
        permanent: false,
        destination: '/another-page',
    },
]
},
}

```

Redirects with basePath support

When leveraging `basePath` support with redirects each `source` and `destination` is automatically prefixed with the `basePath` unless you add `basePath: false` to the redirect:



The screenshot shows a code editor window with a file named `next.config.js`. The code defines a `basePath` and an `async redirects()` function.

```

JS next.config.js

module.exports = {
  basePath: '/docs',

  async redirects() {
    return [
      {
        source: '/with-basePath', // automatic
        destination: '/another', // automatic
        permanent: false,
      },
      {
        // does not add /docs since basePath:
        source: '/without-basePath',
        destination: 'https://example.com',
        basePath: false,
        permanent: false,
      },
    ]
  }
}

```

```
 },  
 }
```

Redirects with i18n support

When leveraging `i18n support` with redirects each `source` and `destination` is automatically prefixed to handle the configured `locales` unless you add `locale: false` to the redirect. If `locale: false` is used you must prefix the `source` and `destination` with a locale for it to be matched correctly.

```
js next.config.js
```

```
module.exports = {  
  i18n: {  
    locales: ['en', 'fr', 'de'],  
    defaultLocale: 'en',  
  },  
  
  async redirects() {  
    return [  
      {  
        source: '/with-locale', // automatica  
        destination: '/another', // automatic  
        permanent: false,  
      },  
      {  
        // does not handle locales automatica  
        source: '/nl/with-locale-manual',  
        destination: '/nl/another',  
        locale: false,  
        permanent: false,  
      },  
      {  
        // this matches '/' since 'en' is the  
        source: '/en',  
        destination: '/en/another',  
        locale: false,  
        permanent: false,  
      },  
      // it's possible to match all locales e  
      {  
        source: '/:locale/page',  
        destination: '/en/newpage',  
        permanent: false,  
        locale: false,  
      },  
    ]  
  }  
}
```

```
{  
    // this gets converted to /(en|fr|de)  
    // `/` or `/fr` routes like /:path* w  
    source: '/(.*)',  
    destination: '/another',  
    permanent: false,  
},  
]  
},  
}
```

In some rare cases, you might need to assign a custom status code for older HTTP Clients to properly redirect. In these cases, you can use the `statusCode` property instead of the `permanent` property, but not both. To ensure IE11 compatibility, a `Refresh` header is automatically added for the 308 status code.

Other Redirects

- Inside [API Routes](#) and [Route Handlers](#), you can redirect based on the incoming request.
- Inside `getStaticProps` and `getServerSideProps`, you can redirect specific pages at request-time.

Version History

Version	Changes
v13.3.0	<code>missing</code> added.
v10.2.0	<code>has</code> added.
v9.5.0	<code>redirects</code> added.

Was this helpful?  



Using Pages Router

Features available in /pages

Latest Version

15.5.4

(i) You are currently viewing the documentation for Pages Router.

rewrites

Rewrites allow you to map an incoming request path to a different destination path.

Rewrites act as a URL proxy and mask the destination path, making it appear the user hasn't changed their location on the site. In contrast, [redirects](#) will reroute to a new page and show the URL changes.

To use rewrites you can use the `rewrites` key in `next.config.js`:

next.config.js



```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/about',
        destination: '/',
      },
    ],
  },
}
```

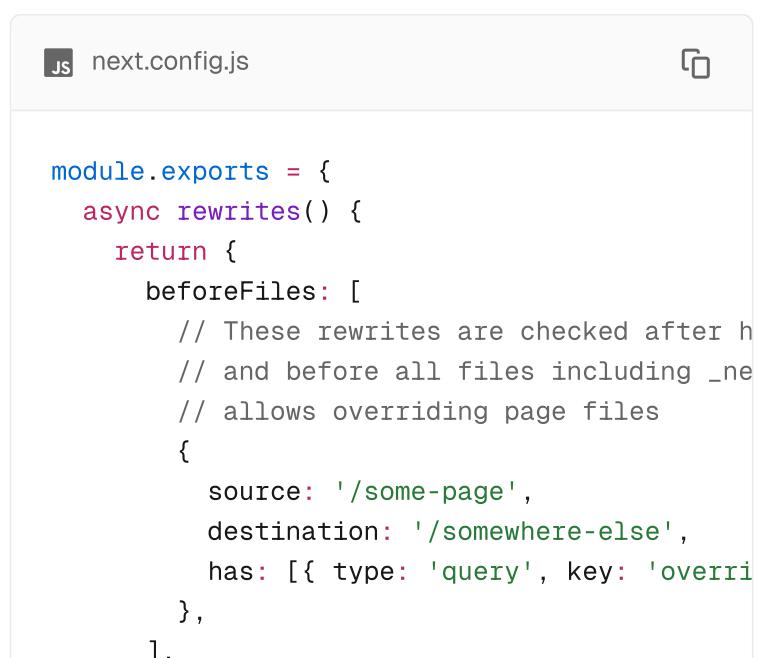
Rewrites are applied to client-side routing. In the example above, navigating to

`<Link href="/about">` will serve content from `/` while keeping the URL as `/about`.

`rewrites` is an async function that expects to return either an array or an object of arrays (see below) holding objects with `source` and `destination` properties:

- `source` : `String` - is the incoming request path pattern.
- `destination` : `String` is the path you want to route to.
- `basePath` : `false` or `undefined` - if false the basePath won't be included when matching, can be used for external rewrites only.
- `locale` : `false` or `undefined` - whether the locale should not be included when matching.
- `has` is an array of `has` objects with the `type`, `key` and `value` properties.
- `missing` is an array of `missing` objects with the `type`, `key` and `value` properties.

When the `rewrites` function returns an array, rewrites are applied after checking the filesystem (pages and `/public` files) and before dynamic routes. When the `rewrites` function returns an object of arrays with a specific shape, this behavior can be changed and more finely controlled, as of `v10.1` of Next.js:



```
JS next.config.js
```

```
module.exports = {
  async rewrites() {
    return {
      beforeFiles: [
        // These rewrites are checked after h
        // and before all files including _ne
        // allows overriding page files
        {
          source: '/some-page',
          destination: '/somewhere-else',
          has: [{ type: 'query', key: 'overri
        },
      ],
    }
  }
}
```

```
afterFiles: [
  // These rewrites are checked after p
  // are checked but before dynamic rou
  {
    source: '/non-existent',
    destination: '/somewhere-else',
  },
],
fallback: [
  // These rewrites are checked after b
  // and dynamic routes are checked
  {
    source: '/:path*',
    destination: `https://my-old-site.c
  },
],
},
}
```

Good to know: rewrites in `beforeFiles` do not check the filesystem/dynamic routes immediately after matching a source, they continue until all `beforeFiles` have been checked.

The order Next.js routes are checked is:

1. `headers` are checked/applied
2. `redirects` are checked/applied
3. `beforeFiles` rewrites are checked/applied
4. static files from the [public directory](#), `_next/static` files, and non-dynamic pages are checked/served
5. `afterFiles` rewrites are checked/applied, if one of these rewrites is matched we check dynamic routes/static files after each match
6. `fallback` rewrites are checked/applied, these are applied before rendering the 404 page and after dynamic routes/all static assets have been checked. If you use [fallback: true/'blocking'](#) in `getStaticPaths`, the fallback `rewrites` defined in your `next.config.js` will *not* be run.

Rewrite parameters

When using parameters in a rewrite the parameters will be passed in the query by default when none of the parameters are used in the destination.

```
JS next.config.js ✖  
  
module.exports = {  
  async rewrites() {  
    return [  
      {  
        source: '/old-about/:path*',  
        destination: '/about', // The :path p  
      },  
    ]  
  },  
}
```

If a parameter is used in the destination none of the parameters will be automatically passed in the query.

```
JS next.config.js ✖  
  
module.exports = {  
  async rewrites() {  
    return [  
      {  
        source: '/docs/:path*',  
        destination: '/:path*', // The :path  
      },  
    ]  
  },  
}
```

You can still pass the parameters manually in the query if one is already used in the destination by specifying the query in the destination.

```
JS next.config.js ✖
```

```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/:first/:second',
        destination: '/:first?second=:second'
        // Since the :first parameter is used
        // will not automatically be added in
        // as shown above
      },
    ]
  },
}
```

Good to know: Static pages from [Automatic Static Optimization](#) or [prerendering](#) params from rewrites will be parsed on the client after hydration and provided in the query.

Path Matching

Path matches are allowed, for example

/blog/:slug will match /blog/hello-world (no nested paths):

next.config.js



```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/blog/:slug',
        destination: '/news/:slug', // Matches
      },
    ],
  },
}
```

Wildcard Path Matching

To match a wildcard path you can use `*` after a parameter, for example `/blog/:slug*` will match `/blog/a/b/c/d/hello-world`:

```
JS next.config.js

module.exports = {
  async rewrites() {
    return [
      {
        source: '/blog/:slug*',
        destination: '/news/:slug*', // Match
      },
    ],
  },
}
```

Regex Path Matching

To match a regex path you can wrap the regex in parenthesis after a parameter, for example

`/blog/:slug(\d{1,})` will match `/blog/123`
but not `/blog/abc`:

```
JS next.config.js

module.exports = {
  async rewrites() {
    return [
      {
        source: '/old-blog/:post(\d{1,})',
        destination: '/blog/:post', // Matche
      },
    ],
  },
}
```

The following characters `(`, `)`, `{`, `}`, `[`, `]`, `|`, `\`, `^`, `.`, `:`, `*`, `+`, `-`, `?`, `$` are used for regex path matching, so when used in the `source` as non-special values they must be escaped by adding `\` before them:

```

module.exports = {
  async rewrites() {
    return [
      {
        // this will match `/english(default)`
        source: '/english\\((default\\))/:slug',
        destination: '/en-us/:slug',
      },
    ],
  },
}

```

Header, Cookie, and Query Matching

To only match a rewrite when header, cookie, or query values also match the `has` field or don't match the `missing` field can be used. Both the `source` and all `has` items must match and all `missing` items must not match for the rewrite to be applied.

`has` and `missing` items can have the following fields:

- `type : String` - must be either `header`, `cookie`, `host`, or `query`.
- `key : String` - the key from the selected type to match against.
- `value : String` or `undefined` - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value `first-(?<paramName>.*)` is used for `first-second` then `second` will be usable in the destination with `:paramName`.

```
module.exports = {
  async rewrites() {
    return [
      // if the header `x-rewrite-me` is present
      // this rewrite will be applied
      {
        source: '/:path*',
        has: [
          {
            type: 'header',
            key: 'x-rewrite-me',
          },
        ],
        destination: '/another-page',
      },
      // if the header `x-rewrite-me` is not present
      // this rewrite will be applied
      {
        source: '/:path*',
        missing: [
          {
            type: 'header',
            key: 'x-rewrite-me',
          },
        ],
        destination: '/another-page',
      },
      // if the source, query, and cookie are present
      // this rewrite will be applied
      {
        source: '/specific/:path*',
        has: [
          {
            type: 'query',
            key: 'page',
            // the page value will not be available
            // destination since value is provided
            // use a named capture group e.g.
            value: 'home',
          },
          {
            type: 'cookie',
            key: 'authorized',
            value: 'true',
          },
        ],
        destination: '/:path*/home',
      },
      // if the header `x-authorized` is present
      // contains a matching value, this rewrite will be applied
      {
        source: '/:path*',
        has: [
          {
            type: 'header',
            key: 'x-authorized',
            value: 'true',
          },
        ],
        destination: '/:path*/home',
      },
    ]
  }
}
```

```
{  
    type: 'header',  
    key: 'x-authorized',  
    value: '(?<authorized>yes|true)',  
},  
],  
destination: '/home?authorized=:autho  
},  
// if the host is `example.com`,  
// this rewrite will be applied  
{  
    source: '/:path*',  
    has: [  
        {  
            type: 'host',  
            value: 'example.com',  
        },  
        ],  
    destination: '/another-page',  
},  
]  
},  
}
```

Rewriting to an external URL

► Examples

Rewrites allow you to rewrite to an external URL. This is especially useful for incrementally adopting Next.js. The following is an example rewrite for redirecting the `/blog` route of your main app to an external site.

next.config.js

```
module.exports = {  
  async rewrites() {  
    return [  
      {  
        source: '/blog',  
        destination: 'https://example.com/blo  
      },  
      {  
        source: '/blog/:slug',  
        destination: 'https://example.com/blo  
      }  
    ]  
  }  
}
```

```
        },
      ],
    },
}
```

If you're using `trailingSlash: true`, you also need to insert a trailing slash in the `source` parameter. If the destination server is also expecting a trailing slash it should be included in the `destination` parameter as well.

next.config.js

```
module.exports = {
  trailingSlash: true,
  async rewrites() {
    return [
      {
        source: '/blog/',
        destination: 'https://example.com/blog/',
      },
      {
        source: '/blog/:path*/',
        destination: 'https://example.com/blog/:path*/',
      },
    ]
  },
}
```

Incremental adoption of Next.js

You can also have Next.js fall back to proxying to an existing website after checking all Next.js routes.

This way you don't have to change the rewrites configuration when migrating more pages to Next.js

next.config.js

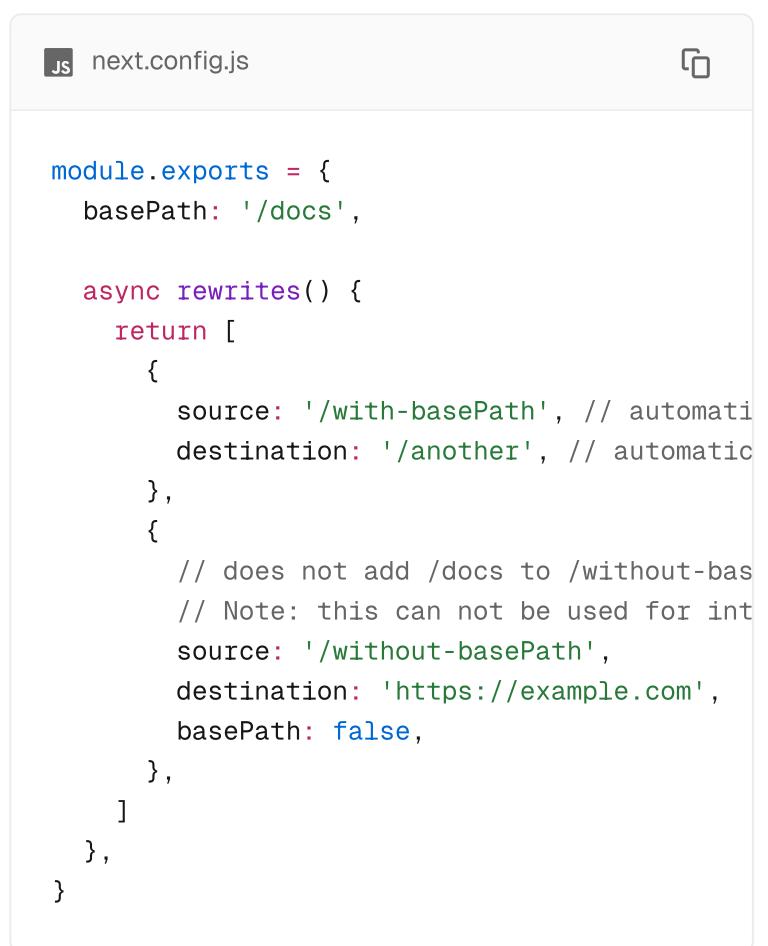
```
module.exports = {
  async rewrites() {
    return {
      fallback: [

```

```
{  
  source: '/:path*',  
  destination: `https://custom-routes  
  },  
],  
}  
},  
}
```

Rewrites with basePath support

When leveraging `basePath` support with rewrites each `source` and `destination` is automatically prefixed with the `basePath` unless you add `basePath: false` to the rewrite:



The screenshot shows a code editor window with a file named `next.config.js`. The code defines a `module.exports` object with a `basePath` property set to `'/docs'`. It then defines an `async rewrites()` function which returns an array of rewrite rules. The first rule has a `source` of `'/with-basePath'` and a `destination` of `'/another'`. The second rule has a `source` of `'/without-basePath'`, a `destination` of `'https://example.com'`, and a `basePath: false` option.

```
JS next.config.js Copy  
  
module.exports = {  
  basePath: '/docs',  
  
  async rewrites() {  
    return [  
      {  
        source: '/with-basePath', // automatic  
        destination: '/another', // automatic  
      },  
      {  
        // does not add /docs to /without-bas  
        // Note: this can not be used for int  
        source: '/without-basePath',  
        destination: 'https://example.com',  
        basePath: false,  
      },  
    ]  
  },  
}
```

Rewrites with i18n support

When leveraging `i18n` support with rewrites each `source` and `destination` is automatically prefixed to handle the configured `locales` unless you add `locale: false` to the rewrite. If `locale: false` is used you must prefix the

source and destination with a locale for it to be matched correctly.

next.config.js



```
module.exports = {
  i18n: {
    locales: ['en', 'fr', 'de'],
    defaultLocale: 'en',
  },
  async rewrites() {
    return [
      {
        source: '/with-locale', // automatica
        destination: '/another', // automatic
      },
      {
        // does not handle locales automatica
        source: '/nl/with-locale-manual',
        destination: '/nl/another',
        locale: false,
      },
      {
        // this matches '/' since `en` is the
        source: '/en',
        destination: '/en/another',
        locale: false,
      },
      {
        // it's possible to match all locales
        source: '/:locale/api-alias/:path*',
        destination: '/api/:path*',
        locale: false,
      },
      {
        // this gets converted to /(en|fr|de)
        // `/` or `/fr` routes like `/:path*` w
        source: '/(.*)',
        destination: '/another',
      },
    ]
  },
}
```

Version History

Version**Changes**

v13.3.0

missing added.

v10.2.0

has added.

v9.5.0

Headers added.

Was this helpful?



Using Pages Router

Features available in /pages

Latest Version

15.5.4

ⓘ You are currently viewing the documentation for Pages Router.

Runtime Config

Warning:

- This feature is deprecated. We recommend using [environment variables](#) instead, which also can support reading runtime values.
- You can run code on server startup using the [register](#) function.
- This feature does not work with [Automatic Static Optimization](#), [Output File Tracing](#), or [React Server Components](#).

To add runtime configuration to your app, open

`next.config.js` and add the
`publicRuntimeConfig` and
`serverRuntimeConfig` configs:

`next.config.js`

```
module.exports = {
  serverRuntimeConfig: {
    // Will only be available on the server
    mySecret: 'secret',
    secondSecret: process.env.SECOND_SECRET,
  },
  publicRuntimeConfig: {
    // Will be available on both server and client
    staticFolder: '/static',
  },
}
```

Place any server-only runtime config under
`serverRuntimeConfig`.

Anything accessible to both client and server-side code should be under `publicRuntimeConfig`.

A page that relies on `publicRuntimeConfig` **must** use `getInitialProps` or `getServerSideProps` or your application must have a [Custom App](#) with `getInitialProps` to opt-out of [Automatic Static Optimization](#). Runtime configuration won't be available to any page (or component in a page) without being server-side rendered.

To get access to the runtime configs in your app use `next/config`, like so:

```
import getConfig from 'next/config'
import Image from 'next/image'

// Only holds serverRuntimeConfig and publicRuntimeConfig
const { serverRuntimeConfig, publicRuntimeConfig } = getConfig()
// Will only be available on the server-side
console.log(serverRuntimeConfig.mySecret)
// Will be available on both server-side and client-side
console.log(publicRuntimeConfig.staticFolder)

function MyImage() {
  return (
    <div>
      <Image
        src={`${publicRuntimeConfig.staticFolder}/logo.png`}
        alt="logo"
        layout="fill"
      />
    </div>
  )
}

export default MyImage
```

Was this helpful?

Using Pages Router
Features available in /pages

Latest Version
15.5.4

You are currently viewing the documentation for Pages Router.

serverExternalPackages

Opt-out specific dependencies from being included in the automatic bundling of the `bundlePagesRouterDependencies` option.

These pages will then use native Node.js `require` to resolve the dependency.

```
JS next.config.js 
```

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  serverExternalPackages: ['@acme/ui'],
}

module.exports = nextConfig
```

Next.js includes a [short list of popular packages ↗](#) that currently are working on compatibility and automatically opt-ed out:

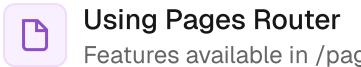
- `@appsignal/nodejs`
- `@aws-sdk/client-s3`
- `@aws-sdk/s3-presigned-post`
- `@blockfrost/blockfrost-js`
- `@highlight-run/node`
- `@huggingface/transformers`

- `@jpg-store/lucid-cardano`
- `@libsqli/client`
- `@mikro-orm/core`
- `@mikro-orm/knex`
- `@node-rs/argon2`
- `@node-rs/bcrypt`
- `@prisma/client`
- `@react-pdf/renderer`
- `@sentry/profiling-node`
- `@sparticuz/chromium`
- `@sparticuz/chromium-min`
- `@swc/core`
- `@xenova/transformers`
- `argon2`
- `autoprefixer`
- `aws-crt`
- `bcrypt`
- `better-sqlite3`
- `canvas`
- `chromadb-default-embed`
- `config`
- `cpu-features`
- `cypress`
- `dd-trace`
- `eslint`
- `express`
- `firebase-admin`
- `htmlrewriter`
- `import-in-the-middle`
- `isolated-vm`

- jest
- jsdom
- keyv
- libsql
- mdx-bundler
- mongodb
- mongoose
- newrelic
- next-mdx-remote
- next-seo
- node-cron
- node-pty
- node-web-audio-api
- onnxruntime-node
- oslo
- pg
- playwright
- playwright-core
- postcss
- prettier
- prisma
- puppeteer-core
- puppeteer
- ravendb
- require-in-the-middle
- rimraf
- sharp
- shiki
- sqlite3
- ts-node

- [ts-morph](#)
- [typescript](#)
- [vscode-oniguruma](#)
- [webpack](#)
- [websocket](#)
- [zeromq](#)

Was this helpful?    



(i) You are currently viewing the documentation for Pages Router.

trailingSlash

By default Next.js will redirect URLs with trailing slashes to their counterpart without a trailing slash. For example `/about/` will redirect to `/about`. You can configure this behavior to act the opposite way, where URLs without trailing slashes are redirected to their counterparts with trailing slashes.

Open `next.config.js` and add the `trailingSlash` config:

`JS` next.config.js

```
module.exports = {  
  trailingSlash: true,  
}
```

With this option set, URLs like `/about` will redirect to `/about/`.

When using `trailingSlash: true`, certain URLs are exceptions and will not have a trailing slash appended:

- Static file URLs, such as files with extensions.
- Any paths under `.well-known/`.

For example, the following URLs will remain unchanged: `/file.txt`,

`images/photos/picture.png`, and
`.well-known/subfolder/config.json`.

When used with `output: "export"` configuration,
the `/about` page will output `/about/index.html`
(instead of the default `/about.html`).

Version History

Version	Changes
v9.5.0	trailingSlash added.

Was this helpful?    

Using Pages Router
Features available in /pages

Latest Version
15.5.4

You are currently viewing the documentation for Pages Router.

transpilePackages

Next.js can automatically transpile and bundle dependencies from local packages (like monorepos) or from external dependencies (`node_modules`). This replaces the `next-transpile-modules` package.

next.config.js

```
/* @type {import('next').NextConfig} */
const nextConfig = {
  transpilePackages: ['package-name'],
}

module.exports = nextConfig
```

Version History

Version	Changes
v13.0.0	<code>transpilePackages</code> added.

Was this helpful?



ⓘ You are currently viewing the documentation for Pages Router.

turbo

⚠ This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

The `turbopack` option lets you customize [Turbopack](#) to transform different files and change how modules are resolved.

TS

next.config.ts

TypeScript ▾



```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  turbopack: {
    // ...
  },
}

export default nextConfig
```

Good to know:

- Turbopack for Next.js does not require loaders or loader configuration for built-in functionality. Turbopack has built-in support for CSS and compiling modern JavaScript, so there's no need for `css-loader`, `postcss-loader`, or `babel-loader` if you're using `@babel/preset-env`.

Reference

Options

The following options are available for the `turbo` configuration:

Option	Description
<code>root</code>	Sets the application root directory. Should be an absolute path.
<code>rules</code>	List of supported webpack loaders to apply when running with Turbopack.
<code>resolveAlias</code>	Map aliased imports to modules to load in their place.
<code>resolveExtensions</code>	List of extensions to resolve when importing files.

Supported loaders

The following loaders have been tested to work
with Turbopack's webpack loader implementation,
but many other webpack loaders should work as
well even if not listed here:

- [babel-loader](#) ↗
- [@svgr/webpack](#) ↗
- [svg-inline-loader](#) ↗
- [yaml-loader](#) ↗
- [string-replace-loader](#) ↗
- [raw-loader](#) ↗
- [sass-loader](#) ↗
- [graphql-tag/loader](#) ↗

Examples

Root directory

Turbopack uses the root directory to resolve modules. Files outside of the project root are not resolved.

Next.js automatically detects the root directory of your project. It does so by looking for one of these files:

- `pnpm-lock.yaml`
- `package-lock.json`
- `yarn.lock`
- `bun.lock`
- `bun.lockb`

If you have a different project structure, for example if you don't use workspaces, you can manually set the `root` option:

```
JS next.config.js ✖  
  
const path = require('path')  
module.exports = {  
  turbopack: {  
    root: path.join(__dirname, '..'),  
  },  
}
```

Configuring webpack loaders

If you need loader support beyond what's built in, many webpack loaders already work with Turbopack. There are currently some limitations:

- Only a core subset of the webpack loader API is implemented. Currently, there is enough

coverage for some popular loaders, and we'll expand our API support in the future.

- Only loaders that return JavaScript code are supported. Loaders that transform files like stylesheets or images are not currently supported.
- Options passed to webpack loaders must be plain JavaScript primitives, objects, and arrays. For example, it's not possible to pass `require()` plugin modules as option values.

To configure loaders, add the names of the loaders you've installed and any options in `next.config.js`, mapping file extensions to a list of loaders.

Here is an example below using the `@svgr/webpack` loader, which enables importing `.svg` files and rendering them as React components.

```
JS next.config.js

module.exports = {
  turbopack: {
    rules: {
      '*.svg': {
        loaders: ['@svgr/webpack'],
        as: '*.js',
      },
    },
  },
}
```

For loaders that require configuration options, you can use an object format instead of a string:

```
JS next.config.js

module.exports = {
  turbopack: {
    rules: {
```

```
'*.svg': {
  loaders: [
    {
      loader: '@svgr/webpack',
      options: {
        icon: true,
      },
    },
    ],
    as: '*.js',
  },
},
},
}
```

Good to know: Prior to Next.js version 13.4.4, `turbo.rules` was named `turbo.loaders` and only accepted file extensions like `.mdx` instead of `*.mdx`.

Resolving aliases

Turbopack can be configured to modify module resolution through aliases, similar to webpack's `resolve.alias` ↗ configuration.

To configure resolve aliases, map imported patterns to their new destination in `next.config.js`:

```
JS next.config.js
```

```
module.exports = {
  turbopack: {
    resolveAlias: {
      underscore: 'lodash',
      mocha: { browser: 'mocha/browser-entry' },
    },
  },
}
```

This aliases imports of the `underscore` package to the `lodash` package. In other words, `import underscore from 'underscore'` will load the `lodash` module instead of `underscore`.

Turbopack also supports conditional aliasing through this field, similar to Node.js' [conditional exports](#). At the moment only the `browser` condition is supported. In the case above, imports of the `mocha` module will be aliased to `mocha/browser-entry.js` when Turbopack targets browser environments.

Resolving custom extensions

Turbopack can be configured to resolve modules with custom extensions, similar to webpack's [resolve.extensions](#) configuration.

To configure resolve extensions, use the `resolveExtensions` field in `next.config.js`:

```
JS next.config.js Copy  
  
module.exports = {  
  turbopack: {  
    resolveExtensions: ['.mdx', '.tsx', '.ts']  
  },  
}
```

This overwrites the original resolve extensions with the provided list. Make sure to include the default extensions.

For more information and guidance for how to migrate your app to Turbopack from webpack, see [Turbopack's documentation on webpack compatibility](#).

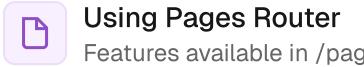
Version History

Version Changes

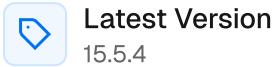
15.3.0 `experimental.turbo` is changed to `turbopack`.

13.0.0 `experimental.turbo` introduced.

Was this helpful?    



(i) You are currently viewing the documentation for Pages Router.



typescript

Next.js fails your **production build** (`next build`) when TypeScript errors are present in your project.

If you'd like Next.js to dangerously produce production code even when your application has errors, you can disable the built-in type checking step.

If disabled, be sure you are running type checks as part of your build or deploy process, otherwise this can be very dangerous.

Open `next.config.js` and enable the `ignoreBuildErrors` option in the `typescript` config:

```
JS next.config.js 
```

```
module.exports = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```


Using Pages Router

Features available in /pages

Latest Version

15.5.4

ⓘ You are currently viewing the documentation for Pages Router.

urlImports

⚠ This feature is currently experimental and subject to change, it's not recommended for production. Try it out and share your feedback on [GitHub](#).

URL imports are an experimental feature that allows you to import modules directly from external servers (instead of from the local disk).

Warning: Only use domains that you trust to download and execute on your machine. Please exercise discretion, and caution until the feature is flagged as stable.

To opt-in, add the allowed URL prefixes inside `next.config.js`:

next.config.js



```
module.exports = {
  experimental: {
    urlImports: ['https://example.com/assets/'],
  },
}
```

Then, you can import modules directly from URLs:

```
import { a, b, c } from 'https://example.com/
```

URL Imports can be used everywhere normal package imports can be used.

Security Model

This feature is being designed with **security as the top priority**. To start, we added an experimental flag forcing you to explicitly allow the domains you accept URL imports from. We're working to take this further by limiting URL imports to execute in the browser sandbox using the [Edge Runtime](#).

Lockfile

When using URL imports, Next.js will create a `next.lock` directory containing a lockfile and fetched assets. This directory **must be committed to Git**, not ignored by `.gitignore`.

- When running `next dev`, Next.js will download and add all newly discovered URL Imports to your lockfile.
- When running `next build`, Next.js will use only the lockfile to build the application for production.

Typically, no network requests are needed and any outdated lockfile will cause the build to fail. One exception is resources that respond with

`Cache-Control: no-cache`. These resources will have a `no-cache` entry in the lockfile and will always be fetched from the network on each build.

Examples

Skypack

```
import confetti from 'https://cdn.skypack.dev
import { useEffect } from 'react'

export default () => {
  useEffect(() => {
    confetti()
  })
  return <p>Hello</p>
}
```

Static Image Imports

```
import Image from 'next/image'
import logo from 'https://example.com/assets/'

export default () => (
  <div>
    <Image src={logo} placeholder="blur" />
  </div>
)
```

URLs in CSS

```
.className {
  background: url('https://example.com/assets')
```

Asset Imports

```
const logo = new URL('https://example.com/ass
console.log(logo.pathname)

// prints "/_next/static/media/file.a9727b5d.
```




You are currently viewing the documentation for Pages Router.



useLightningcss

This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

Experimental support for using [Lightning CSS](#), a fast CSS bundler and minifier, written in Rust.

next.config.ts

TypeScript ▾



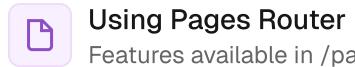
```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    useLightningcss: true,
  },
}

export default nextConfig
```

Was this helpful?





(i) You are currently viewing the documentation for Pages Router.

Custom Webpack Config

Good to know: changes to webpack config are not covered by semver so proceed at your own risk

Before continuing to add custom webpack configuration to your application make sure Next.js doesn't already support your use-case:

- [CSS imports](#)
- [CSS modules](#)
- [Sass/SCSS imports](#)
- [Sass/SCSS modules](#)
- [Customizing babel configuration](#)

Some commonly asked for features are available as plugins:

- [@next/mdx ↗](#)
- [@next/bundle-analyzer ↗](#)

In order to extend our usage of `webpack`, you can define a function that extends its config inside `next.config.js`, like so:

next.config.js



```
module.exports = {
```

```
webpack: (config, { buildId, dev, isServer, defaultLoaders }) => {
  // Important: return the modified config
  return config
},
```

The `webpack` function is executed three times, twice for the server (nodejs / edge runtime) and once for the client. This allows you to distinguish between client and server configuration using the `isServer` property.

The second argument to the `webpack` function is an object with the following properties:

- `buildId : String` - The build id, used as a unique identifier between builds.
- `dev : Boolean` - Indicates if the compilation will be done in development.
- `isServer : Boolean` - It's `true` for server-side compilation, and `false` for client-side compilation.
- `nextRuntime : String | undefined` - The target runtime for server-side compilation; either `"edge"` or `"nodejs"`, it's `undefined` for client-side compilation.
- `defaultLoaders : Object` - Default loaders used internally by Next.js:
 - `babel : Object` - Default `babel-loader` configuration.

Example usage of `defaultLoaders.babel`:

```
// Example config for adding a loader that de
// This source was taken from the @next/mdx p
// https://github.com/vercel/next.js/tree/can
module.exports = {
  webpack: (config, options) => {
```

```
config.module.rules.push({
  test: /\.mdx/,
  use: [
    options.defaultLoaders.babel,
    {
      loader: '@mdx-js/loader',
      options: pluginOptions.options,
    },
  ],
})
```

```
return config
},
```

```
}
```

nextRuntime

Notice that `isServer` is `true` when `nextRuntime` is `"edge"` or `"nodejs"`, `nextRuntime "edge"` is currently for middleware and Server Components in edge runtime only.

Was this helpful?    

Using Pages Router
Features available in /pages

Latest Version
15.5.4

(i) You are currently viewing the documentation for Pages Router.

webVitalsAttribution

When debugging issues related to Web Vitals, it is often helpful if we can pinpoint the source of the problem. For example, in the case of Cumulative Layout Shift (CLS), we might want to know the first element that shifted when the single largest layout shift occurred. Or, in the case of Largest Contentful Paint (LCP), we might want to identify the element corresponding to the LCP for the page. If the LCP element is an image, knowing the URL of the image resource can help us locate the asset we need to optimize.

Pinpointing the biggest contributor to the Web Vitals score, aka [attribution ↗](#), allows us to obtain more in-depth information like entries for [PerformanceEventTiming ↗](#), [PerformanceNavigationTiming ↗](#) and [PerformanceResourceTiming ↗](#).

Attribution is disabled by default in Next.js but can be enabled **per metric** by specifying the following in `next.config.js`.

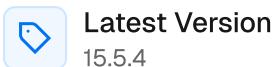
next.config.js



```
module.exports = {
  experimental: {
    webVitalsAttribution: ['CLS', 'LCP'],
  },
}
```

Valid attribution values are all `web-vitals` metrics specified in the [NextWebVitalsMetric](#) ↗ type.

Was this helpful?    



(i) You are currently viewing the documentation for Pages Router.

TypeScript

Next.js comes with built-in TypeScript, automatically installing the necessary packages and configuring the proper settings when you create a new project with `create-next-app`.

To add TypeScript to an existing project, rename 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.

Good to know: If you already have 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.

Examples

Type checking `next.config.ts`

You can use TypeScript and import types in your Next.js configuration by using `next.config.ts`.

next.config.ts



```
import type { NextConfig } from 'next'
```

```
const nextConfig: NextConfig = {
  /* config options here */
}

export default nextConfig
```

Good to know: 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`.

When using the `next.config.js` file, you can add some type checking in your IDE using JSDoc as below:

```
JS next.config.js
```

```
// @ts-check

/** @type {import('next').NextConfig} */
const nextConfig = {
  /* config options here */
}

module.exports = nextConfig
```

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.

Works in both the Pages and App Router for the `href` prop in `next/link`. In the App Router, it also types `next/navigation` methods like `push`, `replace`, and `prefetch`. It does not type `next/router` methods in Pages Router.

Literal `href` strings are validated, while non-literal `href`s may require a cast with `as Route`.

To opt-into this feature, `typedRoutes` need to be enabled and the project needs to be using TypeScript.

```
next.config.ts
```

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  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.

Good to know: If you set up your project without `create-next-app`, ensure the generated Next.js types are included by adding `.next/types/**/*.ts` to the `include` array in your `tsconfig.json`:

```
tsconfig.json
```

```
{
  "include": [
    "next-env.d.ts",
    ".next/types/**/*.ts",
    "**/*.ts",
    "**/*.tsx"
  ],
  "exclude": ["node_modules"]
}
```

Currently, support includes any string literal, including dynamic segments. For non-literal strings, you need to manually cast with `as Route`. The example below shows both `next/link` and `next/navigation` usage:

```
'use client'

import type { Route } from 'next'
import Link from 'next/link'
import { useRouter } from 'next/navigation'

export default function Example() {
  const router = useRouter()
  const slug = 'nextjs'

  return (
    <>
      /* Link: literal and dynamic */
      <Link href="/about" />
      <Link href={`/blog/${slug}`} />
      <Link href={{('/blog/' + slug) as Route}}
        /* TypeScript error if href is not a valid string */
        <Link href="/about" />

      /* Router: literal and dynamic strings */
      <button onClick={() => router.push('/about')}>
        Replace Blog
      </button>
      <button onClick={() => router.replace(`/
        Prefetch Contact
      </button>

      /* For non-literal strings, cast to Router */
      <button onClick={() => router.push((`/
        Push Non-literal Blog
      </button>
    </>
  )
}
```

The same applies for redirecting routes defined by middleware:

```
import { NextRequest, NextResponse } from 'next/server'

export function middleware(request: NextRequest) {
  if (request.nextUrl.pathname === '/middleware')
    return NextResponse.redirect(new URL('/', request.url))
}
```

```
        return NextResponse.next()
    }
```

ts app/some/page.tsx

```
import type { Route } from 'next'

export default function Page() {
    return <Link href={'/middleware-redirect' a
}
```

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 }: { h
    return (
        <Link href={href}>
            <div>My Card</div>
        </Link>
    )
}
```

You can also type a simple data structure and iterate to render links:

ts components/nav-items.ts

```
import type { Route } from 'next'

type NavItem<T extends string = string> = {
    href: T
    label: string
}

export const navItems: NavItem<Route>[] = [
    { href: '/', label: 'Home' },
    { href: '/about', label: 'About' },
    { href: '/blog', label: 'Blog' },
]
```

Then, map over the items to render `Link`s:

```
import Link from 'next/link'
import { navItems } from './nav-items'

export function Nav() {
  return (
    <nav>
      {navItems.map((item) => (
        <Link key={item.href} href={item.href}
              {item.label}>
        </Link>
      ))}
    </nav>
  )
}
```

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.

Type IntelliSense for Environment Variables

During development, Next.js generates a `.d.ts` file in `.next/types` that contains information about the loaded environment variables for your editor's IntelliSense. If the same environment variable key is defined in multiple files, it is deduplicated according to the [Environment Variable Load Order](#).

To opt-into this feature, `experimental.typedEnv` needs to be enabled and the project needs to be using TypeScript.

```
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  experimental: {
    typedEnv: true,
  },
}

export default nextConfig
```

Good to know: Types are generated based on the environment variables loaded at development runtime, which excludes variables from `.env.production*` files by default. To include production-specific variables, run `next dev` with `NODE_ENV=production`.

Static Generation and Server-side Rendering

For `getStaticProps`, `getStaticPaths`, and `getServerSideProps`, you can use the `GetStaticProps`, `GetStaticPaths`, and `GetServerSideProps` types respectively:

pages/blog/[slug].tsx



```
import type { GetStaticProps, GetStaticPaths, GetServerSideProps }

export const getStaticProps = (async (context
  // ...
) satisfies GetStaticProps)

export const getStaticPaths = (async () => {
  // ...
) satisfies GetStaticPaths

export const getServerSideProps = (async (con
  // ...
) satisfies GetServerSideProps
```

Good to know: `satisfies` was added to TypeScript in [4.9 ↗](#). We recommend upgrading to the latest version of TypeScript.

With API Routes

The following is an example of how to use the built-in types for API routes:

pages/api/hello.ts

```
import type { NextApiRequest, NextApiResponse }

export default function handler(req: NextApiR
  res.status(200).json({ name: 'John Doe' })
}
```

You can also type the response data:

pages/api/hello.ts

```
import type { NextApiRequest, NextApiResponse }

type Data = {
  name: string
}

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<Data>
) {
  res.status(200).json({ name: 'John Doe' })
}
```

With 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, pa
  return <Component {...pageProps} />
}
```

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.

Custom `tsconfig` path

In some cases, you might want to use a different TypeScript configuration for builds or tooling. To do that, set `typescript.tsconfigPath` in `next.config.ts` to point Next.js to another `tsconfig` file.

```
TS next.config.ts Copy

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  typescript: {
    tsconfigPath: 'tsconfig.build.json',
  },
}

export default nextConfig
```

For example, switch to a different config for production builds:

```
TS next.config.ts Copy

import type { NextConfig } from 'next'

const isProd = process.env.NODE_ENV === 'prod'

const nextConfig: NextConfig = {
  typescript: {
    tsconfigPath: isProd ? 'tsconfig.build.json' : 'tsconfig.development.json',
  },
}

export default nextConfig
```

► Why you might use a separate `tsconfig` for builds

Good to know:

- IDEs typically read `tsconfig.json` for diagnostics and IntelliSense, so you can still see IDE warnings while production builds use the alternate config. Mirror critical options if you want parity in the editor.
- In development, only `tsconfig.json` is watched for changes. If you edit a different file name via `typescript.tsconfigPath`, restart the dev server to apply changes.
- The configured file is used in `next dev`, `next build`, `next lint`, and `next typegen`.

Disabling TypeScript errors in production

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`:



```
TS next.config.ts

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```

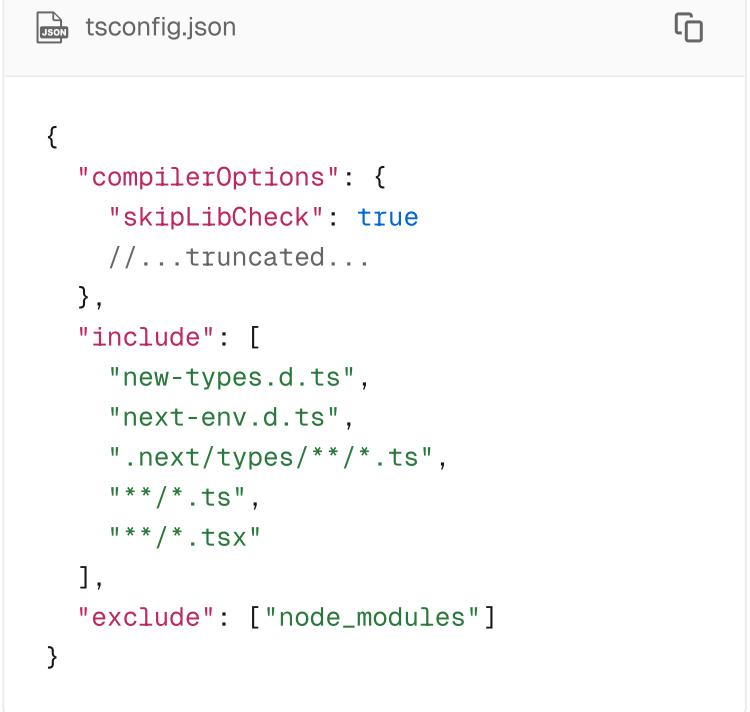
```
}
```

```
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.

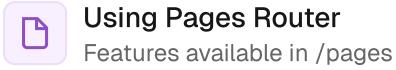
v13.2.0 Statically typed links are available in beta.

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`.

Was this helpful?





Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



ESLint

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.

Reference

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`.

Rules

The full set of rules is as follows:

Enabled in recommended config	Rule	Description
✓	@next/next/google-font-display	Enforce font-display behavior with Google Fonts.
✓	@next/next/google-font-preconnect	Ensure preconnect is used with Google Fonts.
✓	@next/next/inline-script-id	Enforce id attribute on next/script components with inline content.
✓	@next/next/next-script-for-ga	Prefer next/script component when using the inline script for Google Analytics.
✓	@next/next/no-assign-module-variable	Prevent assignment to the module variable.
✓	@next/next/no-async-client-component	Prevent Client Components from being async functions.
✓	@next/next/no-before-interactive-script-outside-document	Prevent usage of next/script's beforeInteractive strategy outside of pages/_document.
✓	@next/next/no-css-tags	Prevent manual stylesheet tags.
✓	@next/next/no-document-import-in-page	Prevent importing next/document outside of pages/_document.

**Enabled in
recommended**

config

Rule

Description



@next/next/no-duplicate-head

Prevent duplicate usage of <Head> pages/_document



@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



@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 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



@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.

Enabled in recommended config	Rule	Description
	@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.

We recommend using an appropriate [integration ↗](#) to view warnings and errors directly in your code editor during development.

Examples

Linting custom directories and files

By default, Next.js will run ESLint for all files in the `pages/`, `app/`, `components/`, `lib/`, and `src/` directories. However, you can specify which directories using the `dirs` option in the `eslint config` in `next.config.js` for production builds:

```
JS next.config.js   
  
module.exports = {  
  eslint: {  
    dirs: ['pages', 'utils'], // Only run ESL  
  },  
}
```

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.
```

Specifying a root directory within a monorepo

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`:



```
import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta dirname is available after N
  baseDirectory: import.meta.dirname,
})

const eslintConfig = [
  ...compat.config({
    extends: ['next'],
    settings: {
      next: {
        rootDir: 'packages/my-app/',
      },
    },
  }),
]

export default eslintConfig
```

`rootDir` can be a path (relative or absolute), a glob (i.e. `"packages/*/*"`), or an array of paths and/or globs.

Disabling the cache

To improve performance, information of files processed by ESLint are cached by default. This is

stored in `.next/cache` or in your defined [build directory](#). If you include any ESLint rules that depend on more than the contents of a single source file and need to disable the cache, use the `--no-cache` flag with `next lint`.

```
>_ Terminal   
next lint --no-cache
```

Disabling rules

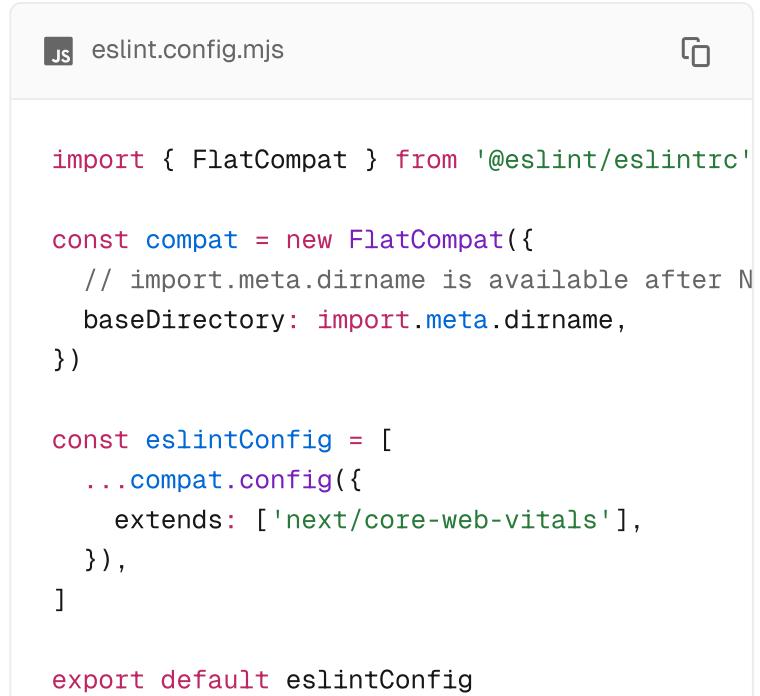
If you would like to modify or disable any rules provided by the supported plugins (`react`, `react-hooks`, `next`), you can directly change them using the `rules` property in your `.eslintrc`:

```
eslint.config.mjs   
  
import { FlatCompat } from '@eslint/eslintrc'  
  
const compat = new FlatCompat({  
  // import.meta.dirname is available after N  
  baseDirectory: import.meta.dirname,  
})  
  
const eslintConfig = [  
  ...compat.config({  
    extends: ['next'],  
    rules: {  
      'react/no-unescaped-entities': 'off',  
      '@next/next/no-page-custom-font': 'off'  
    },  
  }),  
]  
  
export default eslintConfig
```

With 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.



```
import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta.dirname is available after N
  baseDirectory: import.meta.dirname,
})

const eslintConfig = [
  ...compat.config({
    extends: ['next/core-web-vitals'],
  }),
]

export default eslintConfig
```

`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](#).

With 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:



```
import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta.dirname is available after N
  baseDirectory: import.meta.dirname,
})

const eslintConfig = [
```

```
    ...compat.config({  
      extends: ['next/core-web-vitals', 'next/t  
    }),  
  ]  
  
  export default eslintConfig
```

Those rules are based on

[plugin:@typescript-eslint/recommended](#) [↗]. See [typescript-eslint > Configs](#) [↗] for more details.

With Prettier

ESLint also contains code formatting rules, which can conflict with your existing [Prettier](#) [↗] setup. We recommend including [eslint-config-prettier](#) [↗] in your ESLint config to make ESLint and Prettier work together.

First, install the dependency:

```
>_ Terminal ✖  
  
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:

```
JS eslint.config.mjs ✖  
  
import { FlatCompat } from '@eslint/eslintrc'  
  
const compat = new FlatCompat({  
  // import.meta.dirname is available after N  
  baseDirectory: import.meta.dirname,  
})  
  
const eslintConfig = [  
  ...compat.config({
```

```
        extends: ['next', 'prettier'],
    },
]

export default eslintConfig
```

Running lint on staged files

If you would like to use `next lint` with `lint-staged` to run the linter on staged git files, you'll have to add the following to the `.lintstagedrc.js` file in the root of your project in order to specify usage of the `--file` flag.

```
JS .lintstagedrc.js

const path = require('path')

const buildEsLintCommand = (filenames) =>
  `next lint --fix --file ${filenames
    .map((f) => path.relative(process.cwd(),
      .join(' --file ')))}

module.exports = {
  '*.{js,jsx,ts,tsx}': [buildEsLintCommand],
}
```

Disabling linting during production builds

If you do not want ESLint to run during `next build`, you can set the `eslint.ignoreDuringBuilds` option in `next.config.js` to `true`:

```
TS next.config.ts TypeScript ▾

import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  eslint: {
```

```
// Warning: This allows production builds
// your project has ESLint errors.
ignoreDuringBuilds: true,
},
}

export default nextConfig
```

Migrating existing config

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.

Recommended plugin ruleset

If 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`:

```
>_ Terminal
```

```
npm install --save-dev @next/eslint-plugin-next

yarn add --dev @next/eslint-plugin-next

pnpm add --save-dev @next/eslint-plugin-next

bun add --dev @next/eslint-plugin-next
```

This eliminates the risk of collisions or errors that can occur due to importing the same plugin or parser across multiple configurations.

Additional configurations

If you already use a separate ESLint configuration and want to include `eslint-config-next`, ensure that it is extended last after other configurations.

For example:

```
eslint.config.mjs
```

```
import js from '@eslint/js'
import { FlatCompat } from '@eslint/eslintrc'

const compat = new FlatCompat({
  // import.meta.dirname is available after N
  baseDirectory: import.meta.dirname,
  recommendedConfig: js.configs.recommended,
})
```

```
const eslintConfig = [
  ...compat.config({
    extends: ['eslint:recommended', 'next'],
  }),
]

export default eslintConfig
```

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.

Was this helpful?    

Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.

Latest Version

15.5.4



CLI

Next.js comes with **two** Command Line Interface (CLI) tools:

- **create-next-app** : Quickly create a new Next.js application using the default template or an [example ↗](#) from a public GitHub repository.
- **next** : Run the Next.js development server, build your application, and more.

create-next...

Create Next.js apps using one command with th...

next CLI

Learn how to run and build your application with...

Was this helpful?





Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



create-next-app CLI

The `create-next-app` CLI allows you to create a new Next.js application using the default template or an [example ↗](#) from a public GitHub repository. It is the easiest way to get started with Next.js.

Basic usage:

>_ Terminal



```
npx create-next-app@latest [project-name] [op]
```

Reference

The following options are available:

Options	Description
<code>-h</code> or <code>--help</code>	Show all available options
<code>-v</code> or <code>--version</code>	Output the version number
<code>--no-*</code>	Negate default options. E.g. <code>--no-ts</code>
<code>--ts</code> or <code>--typescript</code>	Initialize as a TypeScript project (default)

Options	Description
--js or --javascript	Initialize as a JavaScript project
--tailwind	Initialize with Tailwind CSS config (default)
--eslint	Initialize with ESLint config
--biome	Initialize with Biome config
--no-linter	Skip linter configuration
--app	Initialize as an App Router project
--api	Initialize a project with only route handlers
--src-dir	Initialize inside a <code>src/</code> directory
--turbopack	Enable Turbopack by default for development
--import-alias <code><alias-to-configure></code>	Specify import alias to use (default "@/*")
--empty	Initialize an empty project
--use-npm	Explicitly tell the CLI to bootstrap the application using npm
--use-pnpm	Explicitly tell the CLI to bootstrap the application using pnpm
--use-yarn	Explicitly tell the CLI to bootstrap the application using Yarn
--use-bun	Explicitly tell the CLI to bootstrap the application using Bun
-e or --example [name] [github-url]	An example to bootstrap the app with

Options	Description
--example-path <path-to-example>	Specify the path to the example separately
--reset-preferences	Explicitly tell the CLI to reset any stored preferences
--skip-install	Explicitly tell the CLI to skip installing packages
--disable-git	Explicitly tell the CLI to disable git initialization
--yes	Use previous preferences or defaults for all options

Examples

With the default template

To create a new app using the default template, run the following command in your terminal:

```
>_ Terminal
npx create-next-app@latest
```

You will then be asked the following prompts:

```
>_ Terminal
What is your project named? my-app
Would you like to use TypeScript? No / Yes
Which linter would you like to use? ESLint /
Would you like to use Tailwind CSS? No / Yes
Would you like your code inside a `src/` dire
Would you like to use App Router? (recommende
Would you like to use Turbopack? (recommended
Would you like to customize the import alias
```

Linter Options

ESLint: The traditional and most popular JavaScript linter. Includes Next.js-specific rules from `eslint-plugin-next`.

Biome: A fast, modern linter and formatter that combines the functionality of ESLint and Prettier. Includes built-in Next.js and React domain support for optimal performance.

None: Skip linter configuration entirely. You can always add a linter later.

Note: `next lint` command will be deprecated in Next.js 16. New projects should use the chosen linter directly (e.g., `biome check` or `eslint`).

Once you've answered the prompts, a new project will be created with your chosen configuration.

With an official Next.js example

To create a new app using an official Next.js example, use the `--example` flag. For example:



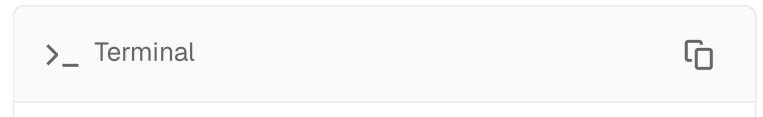
```
>_ Terminal
```

```
npx create-next-app@latest --example [example]
```

You can view a list of all available examples along with setup instructions in the [Next.js repository ↗](#).

With any public GitHub example

To create a new app using any public GitHub example, use the `--example` option with the GitHub repo's URL. For example:



```
>_ Terminal
```

```
npx create-next-app@latest --example "https:/
```

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



next CLI

The Next.js CLI allows you to develop, build, start your application, and more.

Basic usage:

>_ Terminal



```
npx next [command] [options]
```

Reference

The following options are available:

Options	Description
<code>-h</code> or <code>--help</code>	Shows all available options
<code>-v</code> or <code>--version</code>	Outputs the Next.js version number

Commands

The following commands are available:

Command	Description
<code>dev</code>	Starts Next.js in development mode with Hot Module Reloading, error reporting, and more.
<code>build</code>	Creates an optimized production build of your application. Displaying information about each route.
<code>start</code>	Starts Next.js in production mode. The application should be compiled with <code>next build</code> first.
<code>info</code>	Prints relevant details about the current system which can be used to report Next.js bugs.
<code>lint</code>	Runs ESLint for all files in the <code>/src</code> , <code>/app</code> , <code>/pages</code> , <code>/components</code> , and <code>/lib</code> directories. It also provides a guided setup to install any required dependencies if ESLint it is not already configured in your application.
<code>telemetry</code>	Allows you to enable or disable Next.js' completely anonymous telemetry collection.
<code>typegen</code>	Generates TypeScript definitions for routes, pages, layouts, and route handlers without running a full build.

Good to know: Running `next` without a command is an alias for `next dev`.

next dev options

`next dev` starts the application in development mode with Hot Module Reloading (HMR), error reporting, and more. The following options are available when running `next dev`:

Option	Description
<code>-h, --help</code>	Show all available options.
<code>[directory]</code>	A directory in which to build the application. If not provided, current directory is used.
<code>--turbopack</code>	Starts development mode using Turbopack . Also available as <code>--turbo</code> .
<code>-p or --port <port></code>	Specify a port number on which to start the application. Default: 3000, env: PORT
<code>-H or --hostname <hostname></code>	Specify a hostname on which to start the application. Useful for making the application available for other devices on the network. Default: 0.0.0.0
<code>--experimental- https</code>	Starts the server with HTTPS and generates a self-signed certificate.
<code>--experimental- https-key <path></code>	Path to a HTTPS key file.
<code>--experimental- https-cert <path></code>	Path to a HTTPS certificate file.
<code>--experimental- https-ca <path></code>	Path to a HTTPS certificate authority file.
<code>--experimental- upload-trace <traceUrl></code>	Reports a subset of the debugging trace to a remote HTTP URL.

next build options

`next build` creates an optimized production build of your application. The output displays information about each route. For example:

<code>Route</code> (app)	Size
└ <code>/_not-found</code>	0 B
└ <code>/products/[id]</code>	0 B

○ (Static) prerendered as static content
 ↗ (Dynamic) server-rendered on demand

- **Size:** The size of assets downloaded when navigating to the page client-side. The size for each route only includes its dependencies.
- **First Load JS:** The size of assets downloaded when visiting the page from the server. The amount of JS shared by all is shown as a separate metric.

Both of these values are [compressed with gzip](#). The first load is indicated by green, yellow, or red. Aim for green for performant applications.

The following options are available for the `next build` command:

Option	Description
<code>-h, --help</code>	Show all available options.
<code>[directory]</code>	A directory on which to build the application. If not provided, the current directory will be used.
<code>--turbopack</code>	Build using Turbopack (beta). Also available as <code>--turbo</code> .
<code>-d</code> or <code>--debug</code>	Enables a more verbose build output. With this flag enabled additional build output like rewrites, redirects, and headers will be shown.
<code>--profile</code>	Enables production profiling for React .

Option	Description
--no-lint	Disables linting. <i>Note: linting will be removed from <code>next build</code> in Next 16. If you're using Next 15.5+ with a linter other than <code>eslint</code>, linting during build will not occur.</i>
--no-mangling	Disables mangling [↗] . This may affect performance and should only be used for debugging purposes.
--experimental-app-only	Builds only App Router routes.
--experimental-build-mode [mode]	Uses an experimental build mode. (choices: "compile", "generate", default: "default")
--debug-prerender	Debug prerender errors in development.

next start options

`next start` starts the application in production mode. The application should be compiled with `next build` first.

The following options are available for the `next start` command:

Option	Description
-h or --help	Show all available options.
[directory]	A directory on which to start the application. If no directory is provided, the current directory will be used.
-p or --port <port>	Specify a port number on which to start the

Option Description

application. (default: 3000, env: PORT)
<code>-H</code> or <code>--hostname <hostname></code>
<code>--keepAliveTimeout <keepAliveTimeout></code>

next info options

`next info` prints relevant details about the current system which can be used to report Next.js bugs when opening a [GitHub issue ↗](#). This information includes Operating System platform/arch/version, Binaries (Node.js, npm, Yarn, pnpm), package versions (`next`, `react`, `react-dom`), and more.

The output should look like this:

```
>_ Terminal
```

```
Operating System:
  Platform: darwin
  Arch: arm64
  Version: Darwin Kernel Version 23.6.0
  Available memory (MB): 65536
  Available CPU cores: 10
Binaries:
  Node: 20.12.0
  npm: 10.5.0
  Yarn: 1.22.19
  pnpm: 9.6.0
Relevant Packages:
  next: 15.0.0-canary.115 // Latest available
  eslint-config-next: 14.2.5
  react: 19.0.0-rc
  react-dom: 19.0.0
  typescript: 5.5.4
Next.js Config:
```

output: N/A

The following options are available for the `next info` command:

Option	Description
<code>-h</code> or <code>--help</code>	Show all available options
<code>--verbose</code>	Collects additional information for debugging.

next lint options

Warning: This option is deprecated and will be removed in Next 16. A [codemod](#) is available to migrate to ESLint CLI.

`next lint` runs ESLint for all files in the `pages/`, `app/`, `components/`, `lib/`, and `src/` directories. It also provides a guided setup to install any required dependencies if ESLint is not already configured in your application.

The following options are available for the `next lint` command:

Option	Description
<code>[directory]</code>	A base directory on which to lint the application. If not provided, the current directory will be used.
<code>-d</code> , <code>--dir</code> , <code><dirs ...></code>	Include directory, or directories, to run ESLint.
<code>--file</code> , <code><files ...></code>	Include file, or files, to run ESLint.

Option	Description
--ext, [exts ...]	Specify JavaScript file extensions. (default: [".js", ".mjs", ".cjs", ".jsx", ".ts", ".mts", ".cts", ".tsx"])
-c, --config, <config>	Uses this configuration file, overriding all other configuration options.
--resolve-plugins-relative-to, <rprt>	Specify a directory where plugins should be resolved from.
--strict	Creates a <code>.eslintrc.json</code> file using the Next.js strict configuration.
--rulesdir, <rulesdir...>	Uses additional rules from this directory(s).
--fix	Automatically fix linting issues.
--fix-type <fixType>	Specify the types of fixes to apply (e.g., problem, suggestion, layout).
--ignore-path <path>	Specify a file to ignore.
--no-ignore <path>	Disables the <code>--ignore-path</code> option.
--quiet	Reports errors only.
--max-warnings [maxWarnings]	Specify the number of warnings before triggering a non-zero exit code. (default: -1)
-o, --output-file, <outputFile>	Specify a file to write report to.
-f, --format, <format>	Uses a specific output format.

Option	Description
--no-inline-config	Prevents comments from changing config or rules.
--report-unused-disable-directives-severity <level>	Specify severity level for unused eslint-disable directives. (choices: "error", "off", "warn")
--no-cache	Disables caching.
--cache-location, <cacheLocation>	Specify a location for cache.
--cache-strategy, [cacheStrategy]	Specify a strategy to use for detecting changed files in the cache. (default: "metadata")
--error-on-unmatched-pattern	Reports errors when any file patterns are unmatched.
-h, --help	Displays this message.

next telemetry options

Next.js collects **completely anonymous** telemetry data about general usage. Participation in this anonymous program is optional, and you can opt-out if you prefer not to share information.

The following options are available for the `next telemetry` command:

Option	Description
-h, --help	Show all available options.
--enable	Enables Next.js' telemetry collection.
--disable	Disables Next.js' telemetry collection.

next typegen Options

`next typegen` generates TypeScript definitions for your application's routes without performing a full build. This is useful for IDE autocomplete and CI type-checking of route usage.

Previously, route types were only generated during `next dev` or `next build`, which meant running `tsc --noEmit` directly wouldn't validate your route types. Now you can generate types independently and validate them externally:

```
# Generate route types first, then validate w
next typegen && tsc --noEmit

# Or in CI workflows for type checking withou
next typegen && npm run type-check
```

The following options are available for the `next typegen` command:

Option	Description
<code>-h, --help</code>	Show all available options.
<code>[directory]</code>	A directory on which to generate types. If not provided, the current directory will be used.

Output files are written to `<distDir>/types` (default: `.next/types`):

```
next typegen
```

```
# or for a specific app  
next typegen ./apps/web
```

Good to know: `next typegen` loads your Next.js config (`next.config.js`, `next.config.mjs`, or `next.config.ts`) using the production build phase. Ensure any required environment variables and dependencies are available so the config can load correctly.

Examples

Debugging prerender errors

If you encounter prerendering errors during `next build`, you can pass the `--debug-prerender` flag to get more detailed output:



A screenshot of a terminal window titled "Terminal". The command `next build --debug-prerender` is typed into the terminal. The terminal interface includes a close button in the top right corner.

```
next build --debug-prerender
```

This enables several experimental options to make debugging easier:

- Disables server code minification:
 - `experimental.serverMinification = false`
 - `experimental.turbopackMinify = false`
- Generates source maps for server bundles:
 - `experimental.serverSourceMaps = true`
- Enables source map consumption in child processes used for prerendering:
 - `experimental.enablePrerenderSourceMaps = true`

- Continues building even after the first prerender error, so you can see all issues at once:
 - `experimental.prerenderEarlyExit = false`

This helps surface more readable stack traces and code frames in the build output.

Warning: `--debug-prerender` is for debugging in development only. Do not deploy builds generated with `--debug-prerender` to production, as it may impact performance.

Changing the default port

By default, Next.js uses `http://localhost:3000` during development and with `next start`. The default port can be changed with the `-p` option, like so:

```
>_ Terminal
```

```
next dev -p 4000
```

Or using the `PORT` environment variable:

```
>_ Terminal
```

```
PORT=4000 next dev
```

Good to know: `PORT` cannot be set in `.env` as booting up the HTTP server happens before any other code is initialized.

Using HTTPS during development

For certain use cases like webhooks or authentication, you can use [HTTPS ↗](#) to have a

secure environment on `localhost`. Next.js can generate a self-signed certificate with `next dev` using the `--experimental-https` flag:

```
>_ Terminal   
next dev --experimental-https
```

With the generated certificate, the Next.js development server will exist at `https://localhost:3000`. The default port `3000` is used unless a port is specified with `-p`, `--port`, or `PORT`.

You can also provide a custom certificate and key with `--experimental-https-key` and `--experimental-https-cert`. Optionally, you can provide a custom CA certificate with `--experimental-https-ca` as well.

```
>_ Terminal   
next dev --experimental-https --experimental-
```

`next dev --experimental-https` is only intended for development and creates a locally trusted certificate with [mkcert](#) . In production, use properly issued certificates from trusted authorities.

Configuring a timeout for downstream proxies

When deploying Next.js behind a downstream proxy (e.g. a load-balancer like AWS ELB/ALB), it's important to configure Next's underlying HTTP server with [keep-alive timeouts](#)  that are *larger* than the downstream proxy's timeouts. Otherwise, once a keep-alive timeout is reached for a given

TCP connection, Node.js will immediately terminate that connection without notifying the downstream proxy. This results in a proxy error whenever it attempts to reuse a connection that Node.js has already terminated.

To configure the timeout values for the production Next.js server, pass `--keepAliveTimeout` (in milliseconds) to `next start`, like so:

```
>_ Terminal   
next start --keepAliveTimeout 70000
```

Passing Node.js arguments

You can pass any [node arguments](#) to `next` commands. For example:

```
>_ Terminal   
NODE_OPTIONS='--throw-deprecation' next  
NODE_OPTIONS='-r esm' next  
NODE_OPTIONS='--inspect' next
```

Version Changes

v15.5.0 Add the `next typegen` command

v15.4.0 Add `--debug-prerender` option for `next build` to help debug prerender errors.

Was this helpful?    

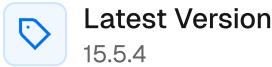


Using Pages Router

Features available in /pages



(i) You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



Edge Runtime

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 is used for rendering your application.
- The **Edge Runtime** which contains a more limited [set of APIs](#), used in [Middleware](#).

Caveats

- The Edge Runtime does not support all Node.js APIs. Some packages may not work as expected.
- The Edge Runtime does not support Incremental Static Regeneration (ISR).
- Both runtimes can support [streaming](#) depending on your deployment adapter.

Reference

The Edge Runtime supports the following APIs:

Network APIs

API	Description
Blob	Represents a blob
fetch	Fetches a resource
FetchEvent	Represents a fetch event
File	Represents a file
FormData	Represents form data
Headers	Represents HTTP headers
Request	Represents an HTTP request
Response	Represents an HTTP response
URLSearchParams	Represents URL search parameters
WebSocket	Represents a websocket connection

Encoding APIs

API	Description
atob	Decodes a base-64 encoded string
btoa	Encodes a string in base-64
TextDecoder	Decodes a Uint8Array into a string
TextDecoderStream	Chainable decoder for streams
TextEncoder	Encodes a string into a Uint8Array
TextEncoderStream	Chainable encoder for streams

Stream APIs

API	Description
ReadableStream ↗	Represents a readable stream
ReadableStreamBYOBReader ↗	Represents a reader of a ReadableStream
ReadableStreamDefaultReader ↗	Represents a reader of a ReadableStream
TransformStream ↗	Represents a transform stream
WritableStream ↗	Represents a writable stream
WritableStreamDefaultWriter ↗	Represents a writer of a WritableStream

Crypto APIs

API	Description
crypto ↗	Provides access to the cryptographic functionality of the platform
CryptoKey ↗	Represents a cryptographic key
SubtleCrypto ↗	Provides access to common cryptographic primitives, like hashing, signing, encryption or decryption

Web Standard APIs

API	Description
AbortController ↗	Allows you to abort one or more DOM requests as and when desired
Array ↗	Represents an array of values
ArrayBuffer ↗	Represents a generic, fixed-length raw binary data buffer
Atomics ↗	Provides atomic operations as static methods
BigInt ↗	Represents a whole number with arbitrary precision
BigInt64Array ↗	Represents a typed array of 64-bit signed integers
BigUint64Array ↗	Represents a typed array of 64-bit unsigned integers
Boolean ↗	Represents a logical entity and can have two values: <code>true</code> and <code>false</code>
clearInterval ↗	Cancels a timed, repeating action which was previously established by a call to <code>setInterval()</code>
clearTimeout ↗	Cancels a timed, repeating action which was previously established by a call to <code>setTimeout()</code>
console ↗	Provides access to the browser's debugging console
DataView ↗	Represents a generic view of an <code>ArrayBuffer</code>
Date ↗	Represents a single moment in time in a platform-independent format
decodeURI ↗	Decodes a Uniform Resource Identifier (URI) previously created

API

Description

by `encodeURI` or by a similar routine

`decodeURIComponent`



Decodes a Uniform Resource Identifier (URI) component previously created by `encodeURIComponent` or by a similar routine

`DOMException`



Represents an error that occurs in the DOM

`encodeURI`



Encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character

`encodeURIComponent`



Encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character

`Error`

Represents an error when trying to execute a statement or accessing a property

`EvalError`

Represents an error that occurs regarding the global function `eval()`

`Float32Array`



Represents a typed array of 32-bit floating point numbers

`Float64Array`



Represents a typed array of 64-bit floating point numbers

`Function`

Represents a function

`Infinity`

Represents the mathematical Infinity value

API	Description
Int8Array	Represents a typed array of 8-bit signed integers
Int16Array	Represents a typed array of 16-bit signed integers
Int32Array	Represents a typed array of 32-bit signed integers
Intl	Provides access to internationalization and localization functionality
isFinite	Determines whether a value is a finite number
isNaN	Determines whether a value is NaN or not
JSON	Provides functionality to convert JavaScript values to and from the JSON format
Map	Represents a collection of values, where each value may occur only once
Math	Provides access to mathematical functions and constants
Number	Represents a numeric value
Object	Represents the object that is the base of all JavaScript objects
parseFloat	Parses a string argument and returns a floating point number
parseInt	Parses a string argument and returns an integer of the specified radix
Promise	Represents the eventual completion (or failure) of an asynchronous operation, and its resulting value

API	Description
Proxy	Represents an object that is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc)
queueMicrotask	Queues a microtask to be executed
RangeError	Represents an error when a value is not in the set or range of allowed values
ReferenceError	Represents an error when a non-existent variable is referenced
Reflect	Provides methods for interceptable JavaScript operations
RegExp	Represents a regular expression, allowing you to match combinations of characters
Set	Represents a collection of values, where each value may occur only once
setInterval	Repeatedly calls a function, with a fixed time delay between each call
setTimeout	Calls a function or evaluates an expression after a specified number of milliseconds
SharedArrayBuffer	Represents a generic, fixed-length raw binary data buffer
String	Represents a sequence of characters
structuredClone	Creates a deep copy of a value
Symbol	Represents a unique and immutable data type that is used

API	Description
	as the key of an object property
SyntaxError ↗	Represents an error when trying to interpret syntactically invalid code
TypeError ↗	Represents an error when a value is not of the expected type
Uint8Array ↗	Represents a typed array of 8-bit unsigned integers
Uint8ClampedArray ↗	Represents a typed array of 8-bit unsigned integers clamped to 0–255
Uint32Array ↗	Represents a typed array of 32-bit unsigned integers
URIError ↗	Represents an error when a global URI handling function was used in a wrong way
URL ↗	Represents an object providing static methods used for creating object URLs
URLPattern ↗	Represents a URL pattern
URLSearchParams ↗	Represents a collection of key/value pairs
WeakMap ↗	Represents a collection of key/value pairs in which the keys are weakly referenced
WeakSet ↗	Represents a collection of objects in which each object may occur only once
WebAssembly ↗	Provides access to WebAssembly

Next.js Specific Polyfills

- [AsyncLocalStorage](#) ↗

Environment Variables

You can use `process.env` to access Environment Variables for both `next dev` and `next build`.

Unsupported APIs

The Edge Runtime has some restrictions including:

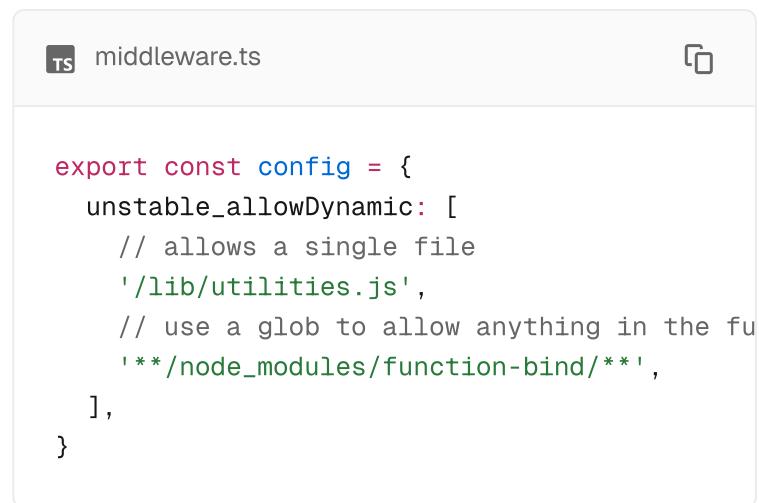
- Native Node.js APIs **are not supported**. For example, you can't read or write to the filesystem.
- `node_modules` *can* be used, as long as they implement ES Modules and do not use native Node.js APIs.
- Calling `require` directly is **not allowed**. Use ES Modules instead.

The following JavaScript language features are disabled, and **will not work**:

API	Description
<code>eval</code> ↗	Evaluates JavaScript code represented as a string
<code>new Function(evalString)</code> ↗	Creates a new function with the code provided as an argument
<code>WebAssembly.compile</code> ↗	Compiles a WebAssembly module from a buffer source
<code>WebAssembly.instantiate</code> ↗	Compiles and instantiates a WebAssembly module from a buffer source

In rare cases, your code could contain (or import) some dynamic code evaluation statements which *can not be reached at runtime* and which can not

be removed by treeshaking. You can relax the check to allow specific files with your Middleware configuration:



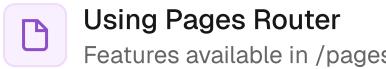
A screenshot of a code editor showing a file named "middleware.ts". The code contains a single export statement defining a constant "config" with a property "unstable_allowDynamic" set to an array of globs. The globs include a single file path and a glob pattern for node modules.

```
export const config = {
  unstable_allowDynamic: [
    // allows a single file
    '/lib/utilities.js',
    // use a glob to allow anything in the function-bind directory
    '**/node_modules/function-bind/**',
  ],
}
```

`unstable_allowDynamic` is a [glob ↗](#), or an array of globs, ignoring dynamic code evaluation for specific files. The globs are relative to your application root folder.

Be warned that if these statements are executed on the Edge, *they will throw and cause a runtime error.*

Was this helpful?    

**Using Pages Router**

Features available in /pages



(i) You are currently viewing the documentation for
Pages Router.

**Latest Version**

15.5.4



Turbopack

Turbopack is an **incremental bundler** optimized for JavaScript and TypeScript, written in Rust, and built into **Next.js**. You can use Turbopack with both the Pages and App Router for a **much faster** local development experience.

Why Turbopack?

We built Turbopack to push the performance of Next.js, including:

- **Unified Graph:** Next.js supports multiple output environments (e.g., client and server). Managing multiple compilers and stitching bundles together can be tedious. Turbopack uses a **single, unified graph** for all environments.
- **Bundling vs Native ESM:** Some tools skip bundling in development and rely on the browser's native ESM. This works well for small apps but can slow down large apps due to excessive network requests. Turbopack **bundles** in dev, but in an optimized way to keep large apps fast.

- **Incremental Computation:** Turbopack parallelizes work across cores and **caches** results down to the function level. Once a piece of work is done, Turbopack won't repeat it.
 - **Lazy Bundling:** Turbopack only bundles what is actually requested by the dev server. This lazy approach can reduce initial compile times and memory usage.
-

Getting started

To enable Turbopack in your Next.js project, add the `--turbopack` flag to the `dev` and `build` scripts in your `package.json` file:



```
JSON package.json

{
  "scripts": {
    "dev": "next dev --turbopack",
    "build": "next build --turbopack",
    "start": "next start"
  }
}
```

Currently, Turbopack for `dev` is stable, while `build` is in beta. We are actively working on production support as Turbopack moves closer to stability.

Supported features

Turbopack in Next.js has **zero-configuration** for the common use cases. Below is a summary of what is supported out of the box, plus some

references to how you can configure Turbopack further when needed.

Language features

Feature	Status	Notes
JavaScript & TypeScript	Supported	Uses SWC under the hood. Type-checking is not done by Turbopack (run <code>tsc --watch</code> or rely on your IDE for type checks).
ECMAScript (ESNext)	Supported	Turbopack supports the latest ECMAScript features, matching SWC's coverage.
CommonJS	Supported	<code>require()</code> syntax is handled out of the box.
ESM	Supported	Static and dynamic <code>import</code> are fully supported.
Babel	Partially Unsupported	Turbopack does not include Babel by default. However, you can configure babel-loader via the Turbopack config .

Framework and React features

Feature	Status	Notes
JSX / TSX	Supported	SWC handles JSX/TSX compilation.
Fast Refresh	Supported	No configuration needed.

Feature	Status	Notes
React Server Components (RSC)	Supported	For the Next.js App Router. Turbopack ensures correct server/client bundling.
Root layout creation	Unsupported	Automatic creation of a root layout in App Router is not supported. Turbopack will instruct you to create it manually.

CSS and styling

Feature	Status	Notes
Global CSS	Supported	Import <code>.css</code> files directly in your application.
CSS Modules	Supported	<code>.module.css</code> files work natively (Lightning CSS).
CSS Nesting	Supported	Lightning CSS supports modern CSS nesting .
@import syntax	Supported	Combine multiple CSS files.
PostCSS	Supported	Automatically processes <code>postcss.config.js</code> in a Node.js worker pool. Useful for Tailwind, Autoprefixer, etc.
Sass / SCSS (Next.js)	Supported	For Next.js, Sass is supported out of the box. In the future, Turbopack standalone usage will likely require a loader config.
Less	Planned via plugins	Not yet supported by default. Will likely require a loader config once custom loaders are stable.

Feature	Status	Notes
Lightning CSS	In Use	Handles CSS transformations. Some low-usage CSS Modules features (like <code>:local/:global</code>) as standalone pseudo-classes) are not yet supported. See below for more details.

Assets

Feature	Status	Notes
Static Assets (images, fonts)	Supported	Importing <code>import img from './img.png'</code> works out of the box. In Next.js, returns an object for the <code><Image /></code> component.
JSON Imports	Supported	Named or default imports from <code>.json</code> are supported.

Module resolution

Feature	Status	Notes
Path Aliases	Supported	Reads <code>tsconfig.json</code> 's <code>paths</code> and <code>baseUrl</code> , matching Next.js behavior.
Manual Aliases	Supported	Configure <code>resolveAlias</code> in <code>next.config.js</code> (similar to <code>webpack.resolve.alias</code>).
Custom Extensions	Supported	Configure <code>resolveExtensions</code> in <code>next.config.js</code> .
AMD	Partially Supported	Basic transforms work; advanced AMD usage is limited.

Performance and Fast Refresh

Feature	Status	Notes
Fast Refresh	Supported	Updates JavaScript, TypeScript, and CSS without a full refresh.
Incremental Bundling	Supported	Turbopack lazily builds only what's requested by the dev server, speeding up large apps.

Known gaps with webpack

There are a number of non-trivial behavior differences between webpack and Turbopack that are important to be aware of when migrating an application. Generally, these are less of a concern for new applications.

CSS Module Ordering

Turbopack will follow JS import order to order [CSS modules](#) which are not otherwise ordered. For example:



```
components/BlogPost.jsx

import utilStyles from './utils.module.css'
import buttonStyles from './button.module.css'
export default function BlogPost() {
  return (
    <div className={utilStyles.container}>
      <button className={buttonStyles.primary}>
    </div>
  )
}
```

In this example, Turbopack will ensure that

`utils.module.css` will appear before

`button.module.css` in the produced CSS chunk,

following the import order

Webpack generally does this as well, but there are cases where it will ignore JS inferred ordering, for example if it infers the JS file is side-effect-free.

This can lead to subtle rendering changes when adopting Turbopack, if applications have come to rely on an arbitrary ordering. Generally, the solution is easy, e.g. have `button.module.css` `@import utils.module.css` to force the ordering, or identify the conflicting rules and change them to not target the same properties.

Bundle Sizes

Turbopack does not yet have an equivalent to the [Inner Graph Optimization ↗](#) in webpack. This optimization is useful to tree shake large modules. For example:

```
import heavy from 'some-heavy-dependency.js'

export function usesHeavy() {
  return heavy.run()
}

export const CONSTANT_VALUE = 3
```

If an application only uses `CONSTANT_VALUE`

Turbopack will detect this and delete the

`usesHeavy` export but not the corresponding

`import`. However, with the

`optimization.innerGraph = true` option

enabled, webpack can delete the `import` too.

We are planning to offer an equivalent to the

`innerGraph` optimization in Turbopack but it is

still under development. If you are affected by this gap, consider manually splitting these modules.

Build Caching

Webpack supports [disk build caching](#) ↗ to speed up builds. We are planning to support an analogous feature in Turbopack but it is not ready yet. On the `next@canary` release you can experiment with our solution by enabling the `experimental.turbopackPersistentCaching` flag.

Good to know: For this reason, when comparing webpack and Turbopack performance, make sure to delete the `.next` folder between builds to see a fair comparison.

Webpack plugins

Turbopack does not support webpack plugins. This affects third-party tools that rely on webpack's plugin system for integration. We do support [webpack loaders](#). If you depend on webpack plugins, you'll need to find Turbopack-compatible alternatives or continue using webpack until equivalent functionality is available.

Unsupported and unplanned features

Some features are not yet implemented or not planned:

- **Legacy CSS Modules features**
 - Standalone `:local` and `:global` pseudo-classes (only the function variant)

:global(...) is supported).

- The `@value` rule (superseded by CSS variables).
- `:import` and `:export` ICSS rules.
- `composes` in `.module.css` composing a `.css` file. In webpack this would treat the `.css` file as a CSS Module, with Turbopack the `.css` file will always be global. This means that if you want to use `composes` in a CSS Module, you need to change the `.css` file to a `.module.css` file.
- `@import` in CSS Modules importing `.css` as a CSS Module. In webpack this would treat the `.css` file as a CSS Module, with Turbopack the `.css` file will always be global. This means that if you want to use `@import` in a CSS Module, you need to change the `.css` file to a `.module.css` file.
- **webpack()** configuration in `next.config.js` Turbopack replaces webpack, so `webpack()` configs are not recognized. Use the [turbopack config](#) instead.
- **AMP** Not planned for Turbopack support in Next.js.
- **Yarn PnP** Not planned for Turbopack support in Next.js.
- **experimental.urlImports** Not planned for Turbopack.
- **experimental.esmExternals** Not planned. Turbopack does not support the legacy `esmExternals` configuration in Next.js.
- **Some Next.js Experimental Flags**
 - `experimental.nextScriptWorkers`
 - `experimental.sri.algorithm`

- `experimental.fallbackNodePolyfills` We plan to implement these in the future.

For a full, detailed breakdown of each feature flag and its status, see the [Turbopack API Reference](#).

Configuration

Turbopack can be configured via `next.config.js` (or `next.config.ts`) under the `turbopack` key. Configuration options include:

- `rules` Define additional [webpack loaders](#) for file transformations.
- `resolveAlias` Create manual aliases (like `resolve.alias` in webpack).
- `resolveExtensions` Change or extend file extensions for module resolution.
- `moduleIds` Set how module IDs are generated (`'named'` vs `'deterministic'`).
- `memoryLimit` Set a memory limit (in bytes) for Turbopack.

```
JS next.config.js ⌂

module.exports = {
  turbopack: {
    // Example: adding an alias and custom fi
    resolveAlias: {
      underscore: 'lodash',
    },
    resolveExtensions: ['.mdx', '.tsx', '.ts'
  },
}
```

For more in-depth configuration examples, see the [Turbopack config documentation](#).

Generating trace files for performance debugging

If you encounter performance or memory issues and want to help the Next.js team diagnose them, you can generate a trace file by appending `NEXT_TURBOPACK_TRACING=1` to your dev command:

```
NEXT_TURBOPACK_TRACING=1 next dev --turbopack
```

This will produce a `.next/trace-turbopack` file. Include that file when creating a GitHub issue on the [Next.js repo ↗](#) to help us investigate.

Summary

Turbopack is a **Rust-based, incremental** bundler designed to make local development and builds fast—especially for large applications. It is integrated into Next.js, offering zero-config CSS, React, and TypeScript support.

Stay tuned for more updates as we continue to improve Turbopack and add production build support. In the meantime, give it a try with `next dev --turbopack` and let us know your feedback.

Version Changes

Version Changes

v15.5.0 Turbopack support for build beta

v15.3.0 Experimental support for build

v15.0.0 Turbopack for dev stable

Was this helpful?



**Using App Router**

Features available in /app



Architecture

**Latest Version**

15.5.4



Learn about the Next.js architecture and how it works under the hood.

Accessibility

The built-in accessibility features of Next.js.

Fast Refresh

Fast Refresh is a hot module reloading...

Next.js Com...

Next.js Compiler, written in Rust, which transforms...

Supported B...

Browser support and which JavaScript...

Was this helpful?





Using App Router
Features available in /app



Accessibility



Latest Version
15.5.4



The Next.js team is committed to making Next.js accessible to all developers (and their end-users). By adding accessibility features to Next.js by default, we aim to make the Web more inclusive for everyone.

Route Announcements

When transitioning between pages rendered on the server (e.g. using the `<a href>` tag) screen readers and other assistive technology announce the page title when the page loads so that users understand that the page has changed.

In addition to traditional page navigations, Next.js also supports client-side transitions for improved performance (using `next/link`). To ensure that client-side transitions are also announced to assistive technology, Next.js includes a route announcer by default.

The Next.js route announcer looks for the page name to announce by first inspecting `document.title`, then the `<h1>` element, and finally the URL pathname. For the most accessible user experience, ensure that each page in your application has a unique and descriptive title.

Linting

Next.js provides an [integrated ESLint experience](#) out of the box, including custom rules for Next.js. By default, Next.js includes

`eslint-plugin-jsx-a11y` to help catch accessibility issues early, including warning on:

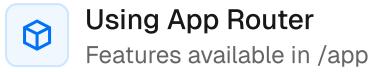
- [aria-props ↗](#)
- [aria-proptypes ↗](#)
- [aria-unsupported-elements ↗](#)
- [role-has-required-aria-props ↗](#)
- [role-supports-aria-props ↗](#)

For example, this plugin helps ensure you add alt text to `img` tags, use correct `aria-*` attributes, use correct `role` attributes, and more.

Accessibility Resources

- [WebAIM WCAG checklist ↗](#)
- [WCAG 2.2 Guidelines ↗](#)
- [The A11y Project ↗](#)
- Check [color contrast ratios ↗](#) between foreground and background elements
- Use `prefers-reduced-motion` ↗ when working with animations

Was this helpful?    



Fast Refresh

Fast refresh is a React feature integrated into Next.js that allows you live reload the browser page while maintaining temporary client-side state when you save changes to a file. It's enabled by default in all Next.js applications on **9.4 or newer**. With Fast Refresh enabled, most edits should be visible within a second.

How It Works

- If you edit a file that **only exports React component(s)**, Fast Refresh will update the code only for that file, and re-render your component. You can edit anything in that file, including styles, rendering logic, event handlers, or effects.
- If you edit a file with exports that *aren't* React components, Fast Refresh will re-run both that file, and the other files importing it. So if both `Button.js` and `Modal.js` import `theme.js`, editing `theme.js` will update both components.
- Finally, if you **edit a file that's imported by files outside of the React tree**, Fast Refresh **will fall back to doing a full reload**. You might have a file which renders a React component but also exports a value that is imported by a **non-React component**. For example, maybe your component also exports a constant, and a non-

React utility file imports it. In that case, consider migrating the constant to a separate file and importing it into both files. This will re-enable Fast Refresh to work. Other cases can usually be solved in a similar way.

Error Resilience

Syntax Errors

If you make a syntax error during development, you can fix it and save the file again. The error will disappear automatically, so you won't need to reload the app. **You will not lose component state.**

Runtime Errors

If you make a mistake that leads to a runtime error inside your component, you'll be greeted with a contextual overlay. Fixing the error will automatically dismiss the overlay, without reloading the app.

Component state will be retained if the error did not occur during rendering. If the error did occur during rendering, React will remount your application using the updated code.

If you have [error boundaries](#) in your app (which is a good idea for graceful failures in production), they will retry rendering on the next edit after a rendering error. This means having an error boundary can prevent you from always getting reset to the root app state. However, keep in mind that error boundaries shouldn't be *too* granular. They are used by React in production, and should always be designed intentionally.

Limitations

Fast Refresh tries to preserve local React state in the component you're editing, but only if it's safe to do so. Here's a few reasons why you might see local state being reset on every edit to a file:

- Local state is not preserved for class components (only function components and Hooks preserve state).
- The file you're editing might have *other* exports in addition to a React component.
- Sometimes, a file would export the result of calling a higher-order component like `HOC(WrappedComponent)`. If the returned component is a class, its state will be reset.
- Anonymous arrow functions like `export default () => <div />`; cause Fast Refresh to not preserve local component state. For large codebases you can use our [name-default-component codemod](#).

As more of your codebase moves to function components and Hooks, you can expect state to be preserved in more cases.

Tips

- Fast Refresh preserves React local state in function components (and Hooks) by default.
- Sometimes you might want to *force* the state to be reset, and a component to be remounted. For example, this can be handy if you're tweaking an animation that only happens on mount. To do this, you can add `// @refresh`

`reset` anywhere in the file you're editing. This directive is local to the file, and instructs Fast Refresh to remount components defined in that file on every edit.

- You can put `console.log` or `debugger;` into the components you edit during development.
 - Remember that imports are case sensitive. Both fast and full refresh can fail, when your import doesn't match the actual filename. For example, `'./header'` vs `'./Header'`.
-

Fast Refresh and Hooks

When possible, Fast Refresh attempts to preserve the state of your component between edits. In particular, `useState` and `useRef` preserve their previous values as long as you don't change their arguments or the order of the Hook calls.

Hooks with dependencies—such as `useEffect`, `useMemo`, and `useCallback`—will *always* update during Fast Refresh. Their list of dependencies will be ignored while Fast Refresh is happening.

For example, when you edit

`useMemo(() => x * 2, [x])` to
`useMemo(() => x * 10, [x])`, it will re-run even though `x` (the dependency) has not changed. If React didn't do that, your edit wouldn't reflect on the screen!

Sometimes, this can lead to unexpected results. For example, even a `useEffect` with an empty array of dependencies would still re-run once during Fast Refresh.

However, writing code resilient to occasional re-running of `useEffect` is a good practice even without Fast Refresh. It will make it easier for you to introduce new dependencies to it later on and it's enforced by [React Strict Mode](#), which we highly recommend enabling.

Was this helpful?    

 Using App Router
Features available in /app

 Latest Version
15.5.4



Supported Browsers



Next.js supports **modern browsers** with zero configuration.

- Chrome 64+
- Edge 79+
- Firefox 67+
- Opera 51+
- Safari 12+

Browserslist

If you would like to target specific browsers or features, Next.js supports [Browserslist](#) configuration in your `package.json` file. Next.js uses the following Browserslist configuration by default:

```
JSON package.json
```

```
{  
  "browserslist": [  
    "chrome 64",  
    "edge 79",  
    "firefox 67",  
    "opera 51",  
    "safari 12"  
  ]  
}
```

Polyfills

We inject [widely used polyfills](#), including:

- [fetch\(\)](#) — Replacing: `whatwg-fetch` and `unfetch`.
- [URL](#) — Replacing: the `url` package (Node.js API).
- [Object.assign\(\)](#) — Replacing: `object-assign`, `object.assign`, and `core-js/object/assign`.

If any of your dependencies include these polyfills, they'll be eliminated automatically from the production build to avoid duplication.

In addition, to reduce bundle size, Next.js will only load these polyfills for browsers that require them. The majority of the web traffic globally will not download these polyfills.

Custom Polyfills

If your own code or any external npm dependencies require features not supported by your target browsers (such as IE 11), you need to add polyfills yourself.

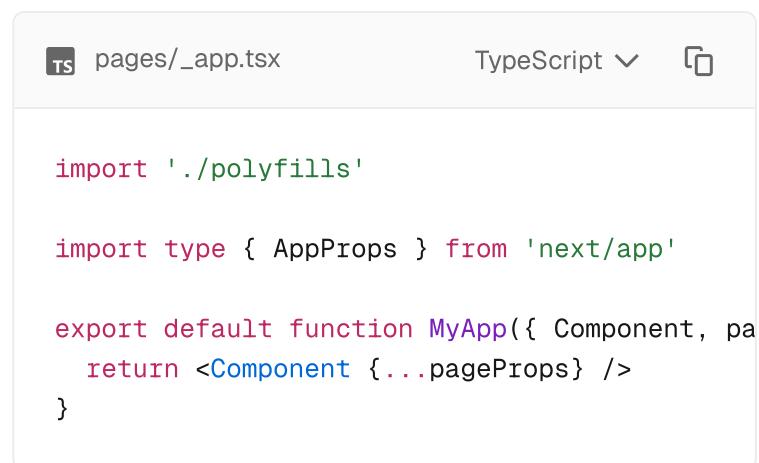
In App Router

To include polyfills, you can import them into the `instrumentation-client.js` file.

```
TS instrumentation-client.ts
import './polyfills'
```

In Pages Router

In this case, you should add a top-level import for the **specific polyfill** you need in your **Custom <App>** or the individual component.



```
TS pages/_app.tsx TypeScript ▾
```

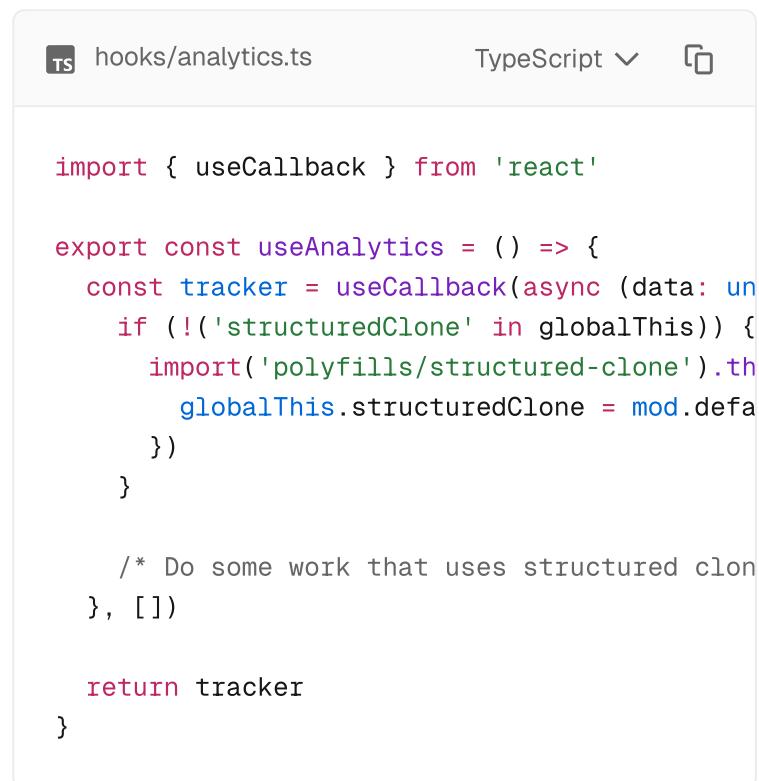
```
import './polyfills'

import type { AppProps } from 'next/app'

export default function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

Conditionally loading polyfills

The best approach is to isolate unsupported features to specific UI sections and conditionally load the polyfill if needed.



```
TS hooks/analytics.ts TypeScript ▾
```

```
import { useCallback } from 'react'

export const useAnalytics = () => {
  const tracker = useCallback(async (data: unknown) => {
    if (!('structuredClone' in globalThis)) {
      import('polyfills/structured-clone').then(mod =>
        globalThis.structuredClone = mod.default
      )
    }
  }, [])
  /* Do some work that uses structured clone */
  return tracker
}
```

JavaScript Language Features

Next.js allows you to use the latest JavaScript features out of the box. In addition to [ES6 features ↗](#), Next.js also supports:

- [Async/await ↗](#) (ES2017)
- [Object Rest/Spread Properties ↗](#) (ES2018)
- [Dynamic `import\(\)` ↗](#) (ES2020)
- [Optional Chaining ↗](#) (ES2020)
- [Nullish Coalescing ↗](#) (ES2020)
- [Class Fields ↗](#) and [Static Properties ↗](#) (ES2022)
- and more!

TypeScript Features

Next.js has built-in TypeScript support. [Learn more here.](#)

Customizing Babel Config (Advanced)

You can customize babel configuration. [Learn more here.](#)

Was this helpful?    



Using App Router

Features available in /app



Next.js Community



Latest Version

15.5.4



With over 5 million weekly downloads, Next.js has a large and active community of developers across the world. Here's how you can get involved in our community:

Contributing

There are a couple of ways you can contribute to the development of Next.js:

- [Documentation](#): Suggest improvements or even write new sections to help our users understand how to use Next.js.
- [Examples ↗](#): Help developers integrate Next.js with other tools and services by creating a new example or improving an existing one.
- [Codebase ↗](#): Learn more about the underlying architecture, contribute to bug fixes, errors, and suggest new features.

Discussions

If you have a question about Next.js, or want to help others, you're always welcome to join the conversation:

- [GitHub Discussions ↗](#)

- [Discord ↗](#)

- [Reddit ↗](#)

Social Media

Follow Next.js on [Twitter ↗](#) for the latest updates, and subscribe to the [Vercel YouTube channel ↗](#) for Next.js videos.

Code of Conduct

We believe in creating an inclusive, welcoming community. As such, we ask all members to adhere to our [Code of Conduct ↗](#). This document outlines our expectations for participant behavior. We invite you to read it and help us maintain a safe and respectful environment.

Contribution ...

Learn how to contribute to Next.js...

Rspack

Use the `next-rspack` plugin to bundle your...

Was this helpful?



 Using App Router
Features available in /app

 Latest Version
15.5.4



Docs Contribution Guide

Welcome to the Next.js Docs Contribution Guide!
We're excited to have you here.

This page provides instructions on how to edit the Next.js documentation. Our goal is to ensure that everyone in the community feels empowered to contribute and improve our docs.

Why Contribute?

Open-source work is never done, and neither is documentation. Contributing to docs is a good way for beginners to get involved in open-source and for experienced developers to clarify more complex topics while sharing their knowledge with the community.

By contributing to the Next.js docs, you're helping us build a more robust learning resource for all developers. Whether you've found a typo, a confusing section, or you've realized that a particular topic is missing, your contributions are welcomed and appreciated.

How to Contribute

The docs content can be found on the [Next.js repo](#)

↗ . To contribute, you can edit the files directly on GitHub or clone the repo and edit the files locally.

GitHub Workflow

If you're new to GitHub, we recommend reading the [GitHub Open Source Guide](#) ↗ to learn how to fork a repository, create a branch, and submit a pull request.

Good to know: The underlying docs code lives in a private codebase that is synced to the Next.js public repo. This means that you can't preview the docs locally. However, you'll see your changes on [nextjs.org](#) ↗ after merging a pull request.

Writing MDX

The docs are written in [MDX](#) ↗ , a markdown format that supports JSX syntax. This allows us to embed React components in the docs. See the [GitHub Markdown Guide](#) ↗ for a quick overview of markdown syntax.

VSCode

Previewing Changes Locally

VSCode has a built-in markdown previewer that you can use to see your edits locally. To enable the previewer for MDX files, you'll need to add a configuration option to your user settings.

Open the command palette (`⌘ + ⌘ + P` on Mac or `Ctrl + Shift + P` on Windows) and search from `Preferences: Open User Settings (JSON)`.

Then, add the following line to your `settings.json` file:



```
{  
  "files.associations": {  
    "*.mdx": "markdown"  
  }  
}
```

Next, open the command palette again, and search for `Markdown: Preview File` or `Markdown: Open Preview to the Side`. This will open a preview window where you can see your formatted changes.

Extensions

We also recommend the following extensions for VSCode users:

- [MDX](#): Intellisense and syntax highlighting for MDX.
- [Prettier](#): Format MDX files on save.

Review Process

Once you've submitted your contribution, the Next.js or Developer Experience teams will review your changes, provide feedback, and merge the pull request when it's ready.

Please let us know if you have any questions or need further assistance in your PR's comments. Thank you for contributing to the Next.js docs and being a part of our community!

Tip: Run `pnpm prettier-fix` to run Prettier before submitting your PR.

File Structure

The docs use **file-system routing**. Each folder and files inside [/docs](#) represent a route segment. These segments are used to generate the URL paths, navigation, and breadcrumbs.

The file structure reflects the navigation that you see on the site, and by default, navigation items are sorted alphabetically. However, we can change the order of the items by prepending a two-digit number (`00-`) to the folder or file name.

For example, in the [functions API Reference](#), the pages are sorted alphabetically because it makes it easier for developers to find a specific function:

```
04-functions
├── after.mdx
├── cacheLife.mdx
├── cacheTag.mdx
└── ...
```

But, in the [routing section](#), the files are prefixed with a two-digit number, sorted in the order developers should learn these concepts:

```
01-routing
├── 01-defining-routes.mdx
├── 02-pages.mdx
└── 03-layouts-and-templates.mdx
└── ...
```

To quickly find a page, you can use `⌘ + P` (Mac) or `Ctrl + P` (Windows) to open the search bar on VSCode. Then, type the slug of the page you're looking for. E.g. `defining-routes`

Why not use a manifest?

We considered using a manifest file (another popular way to generate the docs navigation), but we found that a manifest would quickly get out of sync with the

files. File-system routing forces us to think about the structure of the docs and feels more native to Next.js.

Metadata

Each page has a metadata block at the top of the file separated by three dashes.

Required Fields

The following fields are **required**:

Field	Description
<code>title</code>	The page's <code><h1></code> title, used for SEO and OG Images.
<code>description</code>	The page's description, used in the <code><meta name="description"></code> tag for SEO.



```
required-fields.mdx
```

```
---
```

```
title: Page Title
```

```
description: Page Description
```

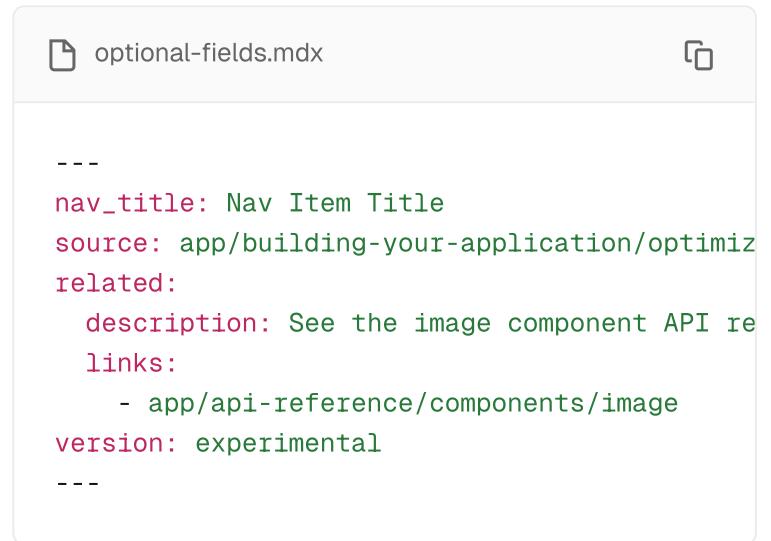
```
---
```

It's good practice to limit the page title to 2-3 words (e.g. Optimizing Images) and the description to 1-2 sentences (e.g. Learn how to optimize images in Next.js).

Optional Fields

The following fields are **optional**:

Field	Description
<code>nav_title</code>	Overrides the page's title in the navigation. This is useful when the page's title is too long to fit. If not provided, the <code>title</code> field is used.
<code>source</code>	Pulls content into a shared page. See Shared Pages .
<code>related</code>	A list of related pages at the bottom of the document. These will automatically be turned into cards. See Related Links .
<code>version</code>	A stage of development. e.g. <code>experimental</code> , <code>legacy</code> , <code>unstable</code> , <code>RC</code>



The screenshot shows a code editor window with a file named "optional-fields.mdx". The code contains several fields defined with their respective values:

```
---
```

```
nav_title: Nav Item Title
source: app/building-your-application/optimiz
related:
  description: See the image component API re
  links:
    - app/api-reference/components/image
version: experimental
---
```

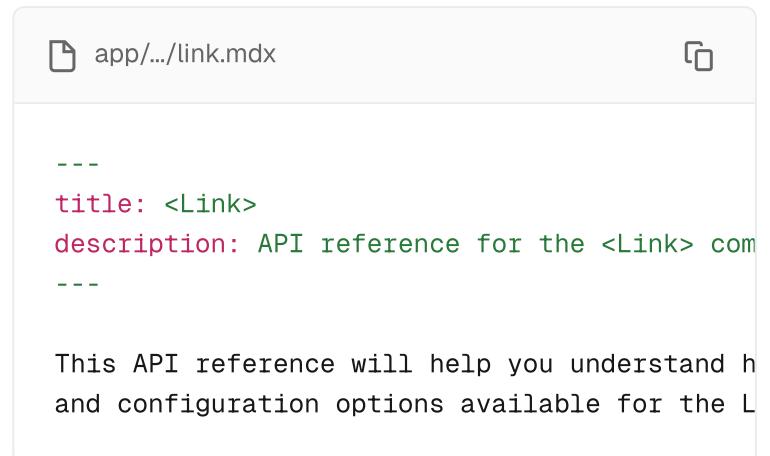
App and Pages Docs

Since most of the features in the **App Router** and **Pages Router** are completely different, their docs for each are kept in separate sections (`02-app` and `03-pages`). However, there are a few features that are shared between them.

Shared Pages

To avoid content duplication and risk the content becoming out of sync, we use the `source` field to pull content from one page into another. For example, the `<Link>` component behaves *mostly* the same in **App** and **Pages**. Instead of duplicating the content, we can pull the content from

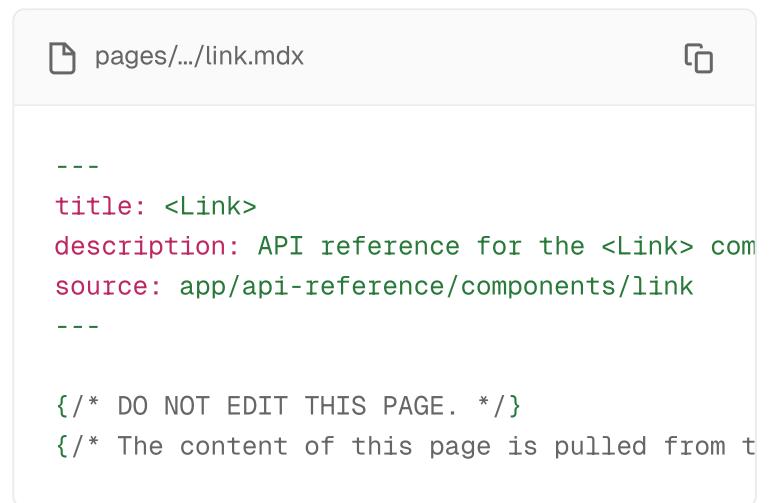
`app/ ... /link.mdx` into `pages/ ... /link.mdx`:



```
app/.../link.mdx

---
title: <Link>
description: API reference for the <Link> com
---


This API reference will help you understand h
and configuration options available for the L
```



```
pages/.../link.mdx

---
title: <Link>
description: API reference for the <Link> com
source: app/api-reference/components/link
---


{/* DO NOT EDIT THIS PAGE. */}
{/* The content of this page is pulled from t
```

We can therefore edit the content in one place and have it reflected in both sections.

Shared Content

In shared pages, sometimes there might be content that is **App Router** or **Pages Router** specific. For example, the `<Link>` component has a `shallow` prop that is only available in **Pages** but not in **App**.

To make sure the content only shows in the correct router, we can wrap content blocks in an

<AppOnly> or <PagesOnly> components:

The screenshot shows a code editor window with a file named "app/.../link.mdx". The content is divided into three sections: 1) A top section labeled "This content is shared between App and Pages." 2) A middle section labeled "<PagesOnly>" containing the text "This content will only be shown on the Pages". 3) A bottom section labeled "</PagesOnly>" containing the text "This content is shared between App and Pages.". There are icons for a file, copy, and close at the top right.

```
This content is shared between App and Pages.

<PagesOnly>

This content will only be shown on the Pages

</PagesOnly>

This content is shared between App and Pages.
```

You'll likely use these components for examples and code blocks.

Code Blocks

Code blocks should contain a minimum working example that can be copied and pasted. This means that the code should be able to run without any additional configuration.

For example, if you're showing how to use the <Link> component, you should include the `import` statement and the <Link> component itself.

The screenshot shows a code editor window with a file named "app/page.tsx". It contains a single export default function named "Page" which returns a component with a Link element. The code is highlighted in green and pink. There are icons for a file, copy, and close at the top right.

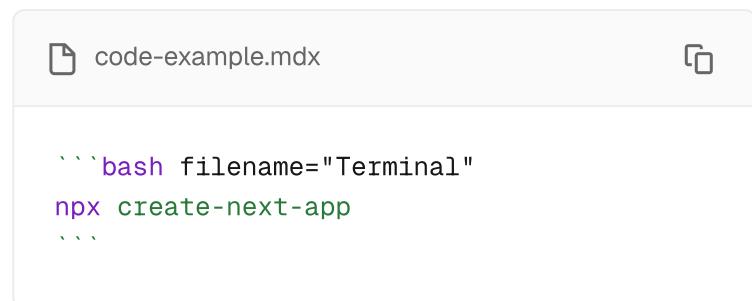
```
import Link from 'next/link'

export default function Page() {
  return <Link href="/about">About</Link>
}
```

Always run examples locally before committing them. This will ensure that the code is up-to-date and working.

Language and Filename

Code blocks should have a header that includes the language and the `filename`. Add a `filename` prop to render a special Terminal icon that helps orientate users where to input the command. For example:



Most examples in the docs are written in `tsx` and `jsx`, and a few in `bash`. However, you can use any supported language, here's the [full list ↗](#).

When writing JavaScript code blocks, we use the following language and extension combinations.

	Language	Extension
JavaScript files with JSX code	<code>```jsx</code>	<code>.js</code>
JavaScript files without JSX	<code>```js</code>	<code>.js</code>
TypeScript files with JSX	<code>```tsx</code>	<code>.tsx</code>
TypeScript files without JSX	<code>```ts</code>	<code>.ts</code>

Good to know:

- Make sure to use `.js` extension with **JSX** code at JavaScript files.
- For example, ````jsx filename="app/layout.js"`

TS and JS Switcher

Add a language switcher to toggle between TypeScript and JavaScript. Code blocks should be

TypeScript first with a JavaScript version to accommodate users.

Currently, we write TS and JS examples one after the other, and link them with `switcher` prop:

Good to know: We plan to automatically compile TypeScript snippets to JavaScript in the future. In the meantime, you can use [transform.tools](#) ↗.

Line Highlighting

Code lines can be highlighted. This is useful when you want to draw attention to a specific part of the code. You can highlight lines by passing a number to the `highlight` prop.

Single Line: `highlight={1}`

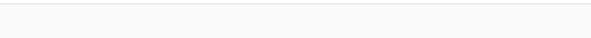


The image shows a screenshot of the Visual Studio Code (VS Code) code editor. The title bar says "app/page.tsx". The editor displays the following TypeScript code:

```
import Link from 'next/link'

export default function Page() {
  return <Link href="/about">About</Link>
}
```

Multiple Lines: highlight={1,3}



```
import Link from 'next/link'
```

```
export default function Page() {
  return <Link href="/about">About</Link>
}
```

Range of Lines: `highlight={1-5}`

TS app/page.tsx

```
import Link from 'next/link'

export default function Page() {
  return <Link href="/about">About</Link>
}
```

Icons

The following icons are available for use in the docs:

mdx-icon.mdx

```
<Check size={18} />
<Cross size={18} />
```

Output:



We do not use emojis in the docs.

Notes

For information that is important but not critical, use notes. Notes are a good way to add

information without distracting the user from the main content.

```
notes.mdx
```

```
> **Good to know**: This is a single line note.  
> **Good to know**:  
>  
> - We also use this format for multi-line notes.  
> - There are sometimes multiple items worth knowing or keeping in mind.
```

Output:

Good to know: This is a single line note.

Good to know:

- We also use this format for multi-line notes.
- There are sometimes multiple items worth knowing or keeping in mind.

Related Links

Related Links guide the user's learning journey by adding links to logical next steps.

- Links will be displayed in cards under the main content of the page.
- Links will be automatically generated for pages that have child pages.

Create related links using the `related` field in the page's metadata.

```
example.mdx
```

```
---  
related:
```

`description`: Learn how to quickly get start
links:

- [app/building-your-application/routing/d](#)
- [app/building-your-application/data-fetc](#)
- [app/api-reference/file-conventions/page](#)

Nested Fields

Field	Required?	Description
<code>title</code>	Optional	The title of the card list. Defaults to Next Steps .
<code>description</code>	Optional	The description of the card list.
<code>links</code>	Required	A list of links to other doc pages. Each list item should be a relative URL path (without a leading slash) e.g. app/api-reference/file-conventions/page

Diagrams

Diagrams are a great way to explain complex concepts. We use [Figma ↗](#) to create diagrams, following Vercel's design guide.

The diagrams currently live in the `/public` folder in our private Next.js site. If you'd like to update or add a diagram, please open a [GitHub issue ↗](#) with your ideas.

Custom Components and HTML

These are the React Components available for the docs: `<Image />` (next/image), `<PagesOnly />`, `<AppOnly />`, `<Cross />`, and `<Check />`. We do not allow raw HTML in the docs besides the `<details>` tag.

If you have ideas for new components, please open a [GitHub issue ↗](#).

Style Guide

This section contains guidelines for writing docs for those who are new to technical writing.

Page Templates

While we don't have a strict template for pages, there are page sections you'll see repeated across the docs:

- **Overview:** The first paragraph of a page should tell the user what the feature is and what it's used for. Followed by a minimum working example or its API reference.
- **Convention:** If the feature has a convention, it should be explained here.
- **Examples:** Show how the feature can be used with different use cases.
- **API Tables:** API Pages should have an overview table at the top of the page with jump-to-section links (when possible).
- **Next Steps (Related Links):** Add links to related pages to guide the user's learning journey.

Feel free to add these sections as needed.

Page Types

Docs pages are also split into two categories: Conceptual and Reference.

- **Conceptual** pages are used to explain a concept or feature. They are usually longer and contain more information than reference pages. In the Next.js docs, conceptual pages are found in the **Building Your Application** section.
- **Reference** pages are used to explain a specific API. They are usually shorter and more focused. In the Next.js docs, reference pages are found in the **API Reference** section.

Good to know: Depending on the page you're contributing to, you may need to follow a different voice and style. For example, conceptual pages are more instructional and use the word *you* to address the user. Reference pages are more technical, they use more imperative words like "create, update, accept" and tend to omit the word *you*.

Voice

Here are some guidelines to maintain a consistent style and voice across the docs:

- Write clear, concise sentences. Avoid tangents.
 - If you find yourself using a lot of commas, consider breaking the sentence into multiple sentences or use a list.
 - Swap out complex words for simpler ones. For example, *use* instead of *utilize*.
- Be mindful with the word *this*. It can be ambiguous and confusing, don't be afraid to repeat the subject of the sentence if unclear.
 - For example, *Next.js uses React* instead of *Next.js uses this*.

- Use an active voice instead of passive. An active sentence is easier to read.
 - For example, *Next.js uses React* instead of *React is used by Next.js*. If you find yourself using words like *was* and *by* you may be using a passive voice.
- Avoid using words like *easy*, *quick*, *simple*, *just*, etc. This is subjective and can be discouraging to users.
- Avoid negative words like *don't*, *can't*, *won't*, etc. This can be discouraging to readers.
 - For example, "*You can use the* `Link` *component to create links between pages*" instead of "*Don't use the* `<a>` *tag to create links between pages*".
- Write in second person (you/your). This is more personal and engaging.
- Use gender-neutral language. Use *developers*, *users*, or *readers*, when referring to the audience.
- If adding code examples, ensure they are properly formatted and working.

While these guidelines are not exhaustive, they should help you get started. If you'd like to dive deeper into technical writing, check out the [Google Technical Writing Course ↗](#).

Thank you for contributing to the docs and being part of the Next.js community!

Was this helpful?





Using App Router
Features available in /app



Rspack Integration



Latest Version
15.5.4



This feature is currently experimental and subject to change, it's not recommended for production.
Try it out and share your feedback on [GitHub](#).

The Rspack team has created a community plugin for Next.js, which is part of a [partnering effort ↗](#) with the Rspack team.

This plugin is currently experimental. Please use this [discussion thread ↗](#) to give feedback on any issues you encounter.

Learn more on the [Rspack docs ↗](#) and try out [this example ↗](#).

Was this helpful?





Using App Router

Features available in /app



App Router



Latest Version

15.5.4



The **App Router** is a file-system based router that uses React's latest features such as [Server Components ↗](#), [Suspense ↗](#), and [Server Functions ↗](#).

Next Steps

Learn the fundamentals of building an App Router project, from installation to layouts, navigation, server and client components.

Installation

Learn how to create a new Next.js applicatio...

Project Struc...

Learn the folder and file conventions in...

Layouts and ...

Learn how to create your first pages and layout...

Linking and ...

Learn how the built-in navigation optimizations...

Server and C...

Learn how you can
use React Server
and Client...

Was this helpful?    



Using Pages Router

Features available in /pages



You are currently viewing the documentation for Pages Router.



Latest Version

15.5.4



Pages Router

You're viewing the documentation for the Pages Router. See the [App Router](#) documentation for the latest features.

The **Pages Router** uses an intuitive file-system router to map each file to a route.

Before Next.js 13, the Pages Router was the main way to create routes in Next.js. It's still supported in newer versions of Next.js, but we recommend migrating to the new [App Router](#) to leverage React's latest features.

Getting Start...

Learn how to create full-stack web applications...

Guides

Learn how to implement common UI...

Building Your...

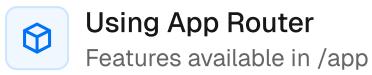
Learn how to use Next.js features to build your...

API Reference

Next.js API Reference for the Pages Router.

Was this helpful?





⚠ You are currently viewing documentation for version 14 of Next.js.



Docs Contribution Guide

Welcome to the Next.js Docs Contribution Guide!

We're excited to have you here.

This page provides instructions on how to edit the Next.js documentation. Our goal is to ensure that everyone in the community feels empowered to contribute and improve our docs.

Why Contribute?

Open-source work is never done, and neither is documentation. Contributing to docs is a good way for beginners to get involved in open-source and for experienced developers to clarify more complex topics while sharing their knowledge with the community.

By contributing to the Next.js docs, you're helping us build a more robust learning resource for all developers. Whether you've found a typo, a confusing section, or you've realized that a particular topic is missing, your contributions are welcomed and appreciated.

How to Contribute

The docs content can be found on the [Next.js repo ↗](#). To contribute, you can edit the files directly on GitHub or clone the repo and edit the files locally.

GitHub Workflow

If you're new to GitHub, we recommend reading the [GitHub Open Source Guide ↗](#) to learn how to fork a repository, create a branch, and submit a pull request.

Good to know: The underlying docs code lives in a private codebase that is synced to the Next.js public repo. This means that you can't preview the docs locally. However, you'll see your changes on [nextjs.org ↗](#) after merging a pull request.

Writing MDX

The docs are written in [MDX ↗](#), a markdown format that supports JSX syntax. This allows us to embed React components in the docs. See the [GitHub Markdown Guide ↗](#) for a quick overview of markdown syntax.

VSCode

Previewing Changes Locally

VSCode has a built-in markdown previewer that you can use to see your edits locally. To enable the previewer for MDX files, you'll need to add a configuration option to your user settings.

Open the command palette (`⌘ + ⌘ + P` on Mac or `Ctrl + Shift + P` on Windows) and search from `Preferences: Open User Settings (JSON)`.

Then, add the following line to your `settings.json` file:



```
JSON settings.json
```

```
{  
  "files.associations": {  
    "*.mdx": "markdown"  
  }  
}
```

Next, open the command palette again, and search for `Markdown: Preview File` or `Markdown: Open Preview to the Side`. This will open a preview window where you can see your formatted changes.

Extensions

We also recommend the following extensions for VSCode users:

- [MDX](#) : Intellisense and syntax highlighting for MDX.
- [Grammarly](#) : Grammar and spell checker.
- [Prettier](#) : Format MDX files on save.

Review Process

Once you've submitted your contribution, the Next.js or Developer Experience teams will review your changes, provide feedback, and merge the pull request when it's ready.

Please let us know if you have any questions or need further assistance in your PR's comments. Thank you for contributing to the Next.js docs and being a part of our community!

Tip: Run `pnpm prettier-fix` to run Prettier before submitting your PR.

File Structure

The docs use **file-system routing**. Each folder and files inside [/docs](#) represent a route segment. These segments are used to generate the URL paths, navigation, and breadcrumbs.

The file structure reflects the navigation that you see on the site, and by default, navigation items are sorted alphabetically. However, we can change the order of the items by prepending a two-digit number (`00-`) to the folder or file name.

For example, in the [functions API Reference](#), the pages are sorted alphabetically because it makes it easier for developers to find a specific function:

```
03-functions
├── cookies.mdx
├── draft-mode.mdx
├── fetch.mdx
└── ...
```

But, in the [routing section](#), the files are prefixed with a two-digit number, sorted in the order developers should learn these concepts:

```
02-routing
├── 01-defining-routes.mdx
├── 02-pages-and-layouts.mdx
├── 03-linking-and-navigating.mdx
└── ...
```

To quickly find a page, you can use `⌘ + P` (Mac) or `Ctrl + P` (Windows) to open the search bar on VSCode. Then, type the slug of the page you're looking for. E.g. `defining-routes`

Why not use a manifest?

We considered using a manifest file (another popular way to generate the docs navigation), but we found that a manifest would quickly get out of sync with the files. File-system routing forces us to think about the structure of the docs and feels more native to Next.js.

Metadata

Each page has a metadata block at the top of the file separated by three dashes.

Required Fields

The following fields are **required**:

Field	Description
<code>title</code>	The page's <code><h1></code> title, used for SEO and OG Images.
<code>description</code>	The page's description, used in the <code><meta name="description"></code> tag for SEO.



```
📁 required-fields.mdx
```

```
---
```

```
title: Page Title
```

```
description: Page Description
```

```
---
```

It's good practice to limit the page title to 2-3 words (e.g. Optimizing Images) and the description to 1-2 sentences (e.g. Learn how to optimize images in Next.js).

Optional Fields

The following fields are **optional**:

Field	Description
<code>nav_title</code>	Overrides the page's title in the navigation. This is useful when the page's title is too long to fit. If not provided, the <code>title</code> field is used.
<code>source</code>	Pulls content into a shared page. See Shared Pages .
<code>related</code>	A list of related pages at the bottom of the document. These will automatically be turned into cards. See Related Links .

```
optional-fields.mdx
```

```
---
```

```
nav_title: Nav Item Title
source: app/building-your-application/optimiz
related:
  description: See the image component API re
  links:
    - app/api-reference/components/image
---
```

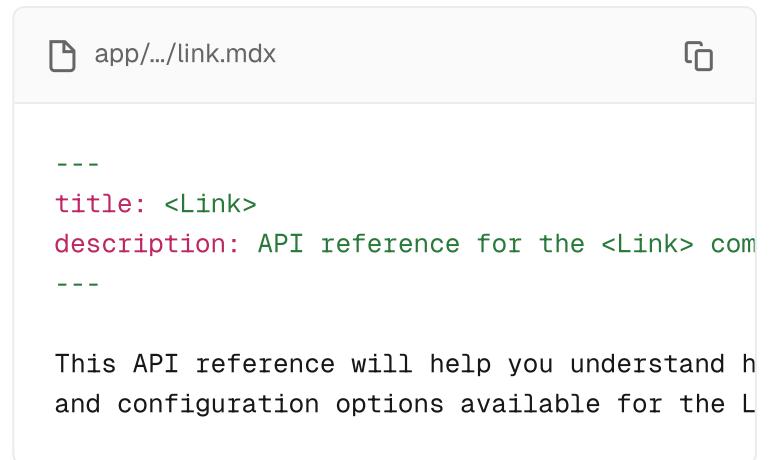
App and Pages Docs

Since most of the features in the [App Router](#) and [Pages Router](#) are completely different, their docs for each are kept in separate sections (`02-app` and `03-pages`). However, there are a few features that are shared between them.

Shared Pages

To avoid content duplication and risk the content becoming out of sync, we use the `source` field to

pull content from one page into another. For example, the `<Link>` component behaves *mostly* the same in **App** and **Pages**. Instead of duplicating the content, we can pull the content from `app/ ... /link.mdx` into `pages/ ... /link.mdx`:



```
app/.../link.mdx

---
title: <Link>
description: API reference for the <Link> com
---

This API reference will help you understand h
and configuration options available for the L
```



```
pages/.../link.mdx

---
title: <Link>
description: API reference for the <Link> com
source: app/api-reference/components/link
---

/* DO NOT EDIT THIS PAGE. */
/* The content of this page is pulled from t
```

We can therefore edit the content in one place and have it reflected in both sections.

Shared Content

In shared pages, sometimes there might be content that is **App Router** or **Pages Router** specific. For example, the `<Link>` component has a `shallow` prop that is only available in **Pages** but not in **App**.

To make sure the content only shows in the correct router, we can wrap content blocks in an `<AppOnly>` or `<PagesOnly>` components:



This content is shared between App and Pages.

<PagesOnly>

This content will only be shown on the Pages

</PagesOnly>

This content is shared between App and Pages.

You'll likely use these components for examples and code blocks.

Code Blocks

Code blocks should contain a minimum working example that can be copied and pasted. This means that the code should be able to run without any additional configuration.

For example, if you're showing how to use the `<Link>` component, you should include the `import` statement and the `<Link>` component itself.



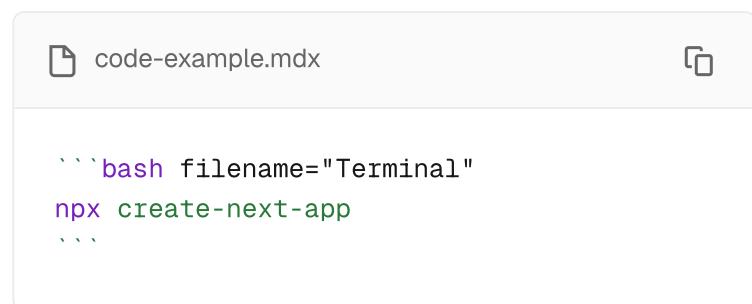
```
import Link from 'next/link'

export default function Page() {
  return <Link href="/about">About</Link>
}
```

Always run examples locally before committing them. This will ensure that the code is up-to-date and working.

Language and Filename

Code blocks should have a header that includes the language and the `filename`. Add a `filename` prop to render a special Terminal icon that helps orientate users where to input the command. For example:



Most examples in the docs are written in `tsx` and `jsx`, and a few in `bash`. However, you can use any supported language, here's the [full list ↗](#).

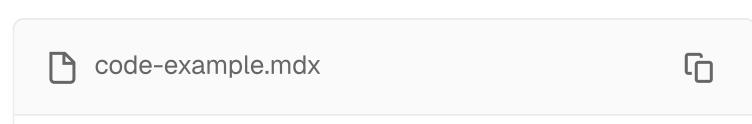
When writing JavaScript code blocks, we use the following language and extension combinations.

	Language	Extension
JavaScript files with JSX code	<code>```jsx</code>	<code>.js</code>
JavaScript files without JSX	<code>```js</code>	<code>.js</code>
TypeScript files with JSX	<code>```tsx</code>	<code>.tsx</code>
TypeScript files without JSX	<code>```ts</code>	<code>.ts</code>

TS and JS Switcher

Add a language switcher to toggle between TypeScript and JavaScript. Code blocks should be TypeScript first with a JavaScript version to accommodate users.

Currently, we write TS and JS examples one after the other, and link them with `switcher` prop:



```
```tsx filename="app/page.tsx" switcher
```

```
...
```

```
```jsx filename="app/page.js" switcher
```

```
...
```

Good to know: We plan to automatically compile TypeScript snippets to JavaScript in the future. In the meantime, you can use [transform.tools ↗](#).

Line Highlighting

Code lines can be highlighted. This is useful when you want to draw attention to a specific part of the code. You can highlight lines by passing a number to the `highlight` prop.

Single Line: `highlight={1}`



The screenshot shows a code editor window with a file named "app/page.tsx". A single line of code, `import Link from 'next/link'`, is highlighted with a blue background. The code block contains the following:

```
TS app/page.tsx
import Link from 'next/link'

export default function Page() {
  return <Link href="/about">About</Link>
}
```

Multiple Lines: `highlight={1,3}`

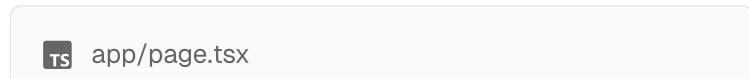


The screenshot shows a code editor window with a file named "app/page.tsx". Multiple lines of code are highlighted with a blue background. The code block contains the following:

```
TS app/page.tsx
import Link from 'next/link'

export default function Page() {
  return <Link href="/about">About</Link>
}
```

Range of Lines: `highlight={1-5}`



The screenshot shows a code editor window with a file named "app/page.tsx". A range of lines, specifically lines 1 through 5, are highlighted with a blue background. The code block contains the following:

```
TS app/page.tsx
import Link from 'next/link'

export default function Page() {
  return <Link href="/about">About</Link>
}
```

```
import Link from 'next/link'

export default function Page() {
  return <Link href="/about">About</Link>
}
```

Icons

The following icons are available for use in the docs:

```
mdx-icon.mdx
```

```
<Check size={18} />
<Cross size={18} />
```

Output:



We do not use emojis in the docs.

Notes

For information that is important but not critical, use notes. Notes are a good way to add information without distracting the user from the main content.

```
notes.mdx
```

```
> **Good to know**: This is a single line note
> **Good to know**:
>
> - We also use this format for multi-line no
```

> - There are sometimes multiple items worth

Output:

Good to know: This is a single line note.

Good to know:

- We also use this format for multi-line notes.
- There are sometimes multiple item worths knowing or keeping in mind.

Related Links

Related Links guide the user's learning journey by adding links to logical next steps.

- Links will be displayed in cards under the main content of the page.
- Links will be automatically generated for pages that have child pages. For example, the [Optimizing](#) section has links to all of its child pages.

Create related links using the `related` field in the page's metadata.

```
example.mdx
```

```
---
```

```
related:
  description: Learn how to quickly get start
  links:
    - app/building-your-application/routing/d
    - app/building-your-application/data-fetc
    - app/api-reference/file-conventions/page
---
```

Nested Fields

Field	Required?	Description
<code>title</code>	Optional	The title of the card list. Defaults to Next Steps .
<code>description</code>	Optional	The description of the card list.
<code>links</code>	Required	A list of links to other doc pages. Each list item should be a relative URL path (without a leading slash) e.g. <code>app/api-reference/file-conventions/page</code>

Diagrams

Diagrams are a great way to explain complex concepts. We use [Figma](#) ↗ to create diagrams, following Vercel's design guide.

The diagrams currently live in the `/public` folder in our private Next.js site. If you'd like to update or add a diagram, please open a [GitHub issue](#) ↗ with your ideas.

Custom Components and HTML

These are the React Components available for the docs: `<Image />` (`next/image`), `<PagesOnly />`, `<AppOnly />`, `<Cross />`, and `<Check />`. We do not allow raw HTML in the docs besides the `<details>` tag.

If you have ideas for new components, please open a [GitHub issue](#).

Style Guide

This section contains guidelines for writing docs for those who are new to technical writing.

Page Templates

While we don't have a strict template for pages, there are page sections you'll see repeated across the docs:

- **Overview:** The first paragraph of a page should tell the user what the feature is and what it's used for. Followed by a minimum working example or its API reference.
- **Convention:** If the feature has a convention, it should be explained here.
- **Examples:** Show how the feature can be used with different use cases.
- **API Tables:** API Pages should have an overview table at the top of the page with jump-to-section links (when possible).
- **Next Steps (Related Links):** Add links to related pages to guide the user's learning journey.

Feel free to add these sections as needed.

Page Types

Docs pages are also split into two categories: Conceptual and Reference.

- **Conceptual** pages are used to explain a concept or feature. They are usually longer and contain more information than reference pages. In the Next.js docs, conceptual pages are found in the **Building Your Application** section.
- **Reference** pages are used to explain a specific API. They are usually shorter and more focused. In the Next.js docs, reference pages are found in the **API Reference** section.

Good to know: Depending on the page you're contributing to, you may need to follow a different voice and style. For example, conceptual pages are more instructional and use the word *you* to address the user. Reference pages are more technical, they use more imperative words like "create, update, accept" and tend to omit the word *you*.

Voice

Here are some guidelines to maintain a consistent style and voice across the docs:

- Write clear, concise sentences. Avoid tangents.
 - If you find yourself using a lot of commas, consider breaking the sentence into multiple sentences or use a list.
 - Swap out complex words for simpler ones. For example, *use* instead of *utilize*.
- Be mindful with the word *this*. It can be ambiguous and confusing, don't be afraid to repeat the subject of the sentence if unclear.
 - For example, *Next.js uses React* instead of *Next.js uses this*.
- Use an active voice instead of passive. An active sentence is easier to read.
 - For example, *Next.js uses React* instead of *React is used by Next.js*. If you find yourself

using words like *was* and *by you* may be using a passive voice.

- Avoid using words like *easy*, *quick*, *simple*, *just*, etc. This is subjective and can be discouraging to users.
- Avoid negative words like *don't*, *can't*, *won't*, etc. This can be discouraging to readers.
- For example, "*You can use the `Link` component to create links between pages*" instead of "*Don't use the `<a>` tag to create links between pages*".
- Write in second person (you/your). This is more personal and engaging.
- Use gender-neutral language. Use *developers*, *users*, or *readers*, when referring to the audience.
- If adding code examples, ensure they are properly formatted and working.

While these guidelines are not exhaustive, they should help you get started. If you'd like to dive deeper into technical writing, check out the [Google Technical Writing Course ↗](#).

Thank you for contributing to the docs and being part of the Next.js community!

Was this helpful?    