Using App Router
Features available in /app

Using Latest Version
15.3.3

Layouts and Pages
Linking and Navigating
Server and Client Components
Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

Guides
Analytics
Authentication

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

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

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

```ts
export default async function Page() {
  const data = await fetch('https://...', { next: { revalidate: 3600 } })
}
```

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:

```ts
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

```ts
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 key
  )
}
```

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

```ts
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 a 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:

app/lib/data.ts — TypeScript

```ts
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:

app/lib/data.ts — TypeScript

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

Then, call `revalidateTag` in a Route Handler or Server Action:

app/lib/actions.ts — TypeScript

```ts
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:

app/lib/actions.ts — TypeScript

```ts
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_cache

API Reference for the unstable_cache function.

revalidatePath

API Reference for the revalidatePath function.

revalidateTag

API Reference for the revalidateTag function.

Previous
Error Handling

Next
CSS

Was this helpful?

Subscribe to our newsletter

Stay updated on new releases and features, guides, and case studies.

you@domain.com Subscribe

Using App Router
Features available in /app

Using Latest Version
15.3.3

Layouts and Pages
Linking and Navigating
Server and Client Components
Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

Guides
Analytics
Authentication

App Router > Getting Started > Error Handling

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.

app/actions.ts — TypeScript

```ts
'use server'

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

  const res = await fetch('https://api.vercel.app/posts', {
    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.

app/ui/form.tsx — TypeScript

```tsx
'use client'

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

const initialState = {
  message: '',
}

export function Form() {
  const [state, formAction, pending] = useActionState(createPost, initialSt

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

Server Components

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

app/page.tsx — TypeScript

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

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

  return '...'
}
```


Edit this page on GitHub

Not found

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

app/blog/[slug]/page.tsx — TypeScript

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

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

  if (!post) {
    notFound()
  }

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

app/blog/[slug]/not-found.tsx — TypeScript

```tsx
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

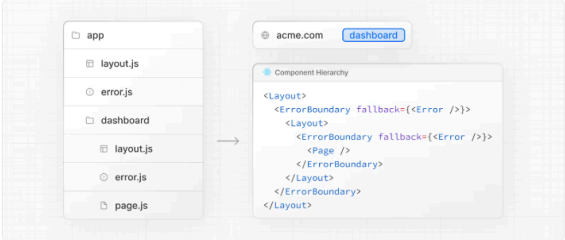
Next.js uses error boundaries to handle uncaught exceptions. Error boundaries catch errors in their child components and display a fallback UI instead of the component tree that crashed.

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

app/dashboard/error.tsx — TypeScript

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

import { useEffect } from 'react'

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

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

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.



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.

app/global-error.tsx TypeScript

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

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

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.

Previous
Updating data

Next
Caching and Revalidating

Using App Router
Features available in /app

Using Latest Version
15.3.3

Layouts and Pages
Linking and Navigating
Server and Client Components
Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

Guides

Analytics
Authentication

Fetching Data and Streaming

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:

app/blog/page.tsx TypeScript

```tsx
export default async function Page() {
  const data = await fetch('https://api.vercel.app/blog')
  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:

app/blog/page.tsx TypeScript

```tsx
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:

app/blog/page.tsx TypeScript

```tsx
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:

app/ui/posts.tsx — TypeScript

```tsx
'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:

app/blog/page.tsx — TypeScript

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

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

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>
  )
}
```

Deduplicating requests with `React.cache`

Deduplication is the process of *preventing duplicate requests* for the same resource during a render pass. It allows you to fetch the same data in different components while preventing multiple network requests to your data source.

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

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

app/lib/data.ts — TypeScript

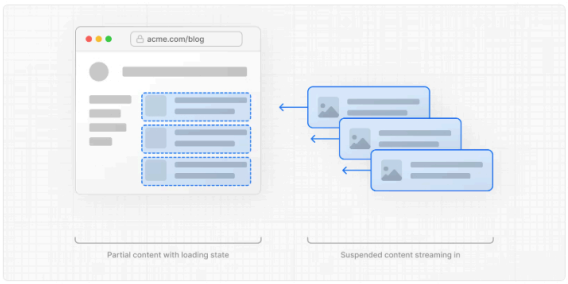
```tsx
import { cache } from 'react'
import { db, posts, eq } from '@/lib/db'

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 `dynamicIO` 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.

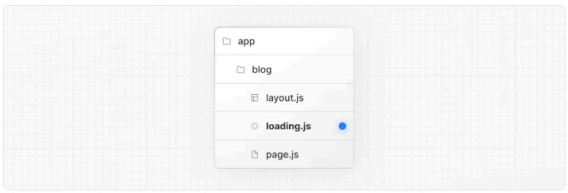


There are two ways you can implement streaming in your application:

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

With `loading.js`

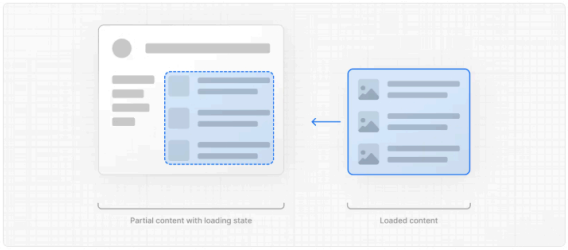
You can create a `loading.js` file in the same folder as your page to stream the **entire page** while the data is being fetched. For example, to stream `app/blog/page.js`, add the file inside the `app/blog` folder.



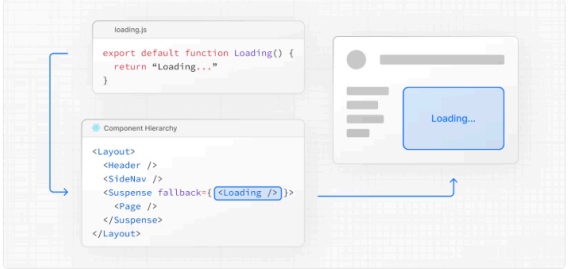
app/blog/loading.tsx TypeScript

```tsx
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.

app/blog/page.tsx — TypeScript

```tsx
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> boundary will be streamed *
        <Suspense fallback={<BlogListSkeleton />}>
          <BlogList />
        </Suspense>
      </main>
    </div>
  )
}
```

Creating meaningful loading states

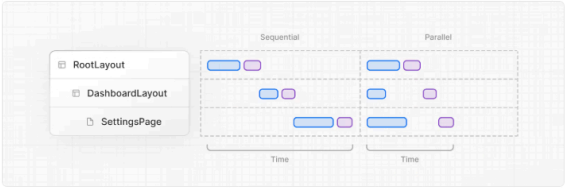
An instant loading state is fallback UI that is shown immediately to the user after navigation. For the best user experience, we recommend designing loading states that are meaningful and help users understand the app is responding. For example, you can use skeletons and spinners, or a small but meaningful 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:

app/artist/[username]/page.tsx — TypeScript

```tsx
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 Playlists component is loading */}
      <Suspense fallback={<div>Loading...</div>}>
        {/* Pass the artist ID to the Playlists component */}
        <Playlists artistID={artist.id} />
      </Suspense>
    </>
  )
}

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

```

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

app/artist/[username]/page.tsx — TypeScript

```tsx
import { getArtist, getAlbums } from '@/app/lib/data'

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` ↗:

app/artist/[username]/page.tsx — TypeScript

```tsx
import Albums from './albums'

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

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

app/item/[id]/page.tsx — TypeScript

```tsx
import { getItem } from '@/lib/data'

export default async function Page({
  params,
}: {
  params: Promise<{ id: string }>
}) {
  const { id } = await params
  // starting loading item data
  preload(id)
```

```tsx
  // 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 returns undefined
  // https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/
  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.

utils/get-item.ts — TypeScript

```ts
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.js and learn best practices for protecting your application's data.

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 console when running Next.js in development mode.

taint

Enable tainting Objects and Values.

Previous
Partial Prerendering

Next
Updating data

Using App Router
Features available in /app
Using Latest Version
15.3.3

Getting Started
Installation
Project Structure
Layouts and Pages
Linking and Navigating
Server and Client Components
Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

Layouts and Pages

Edit this page on GitHub

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 (`/`):

app/page.tsx TypeScript

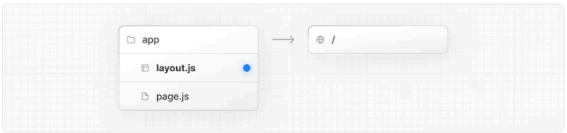
```tsx
export default function Page() {
  return <h1>Hello Next.js!</h1>
}
```

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:



app/layout.tsx TypeScript

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

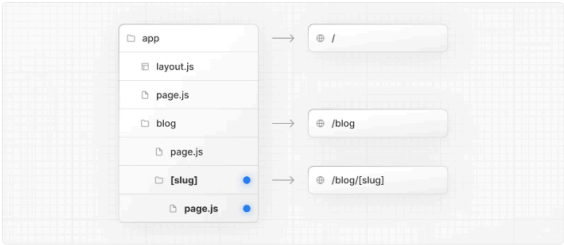
- **Folders** are used to define the route segments that map to URL segments.
- **Files** (like `page` and `layout`) are used to create UI that is shown for a segment.

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:

app/blog/page.tsx — TypeScript

```tsx
// 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:



app/blog/[slug]/page.tsx — TypeScript

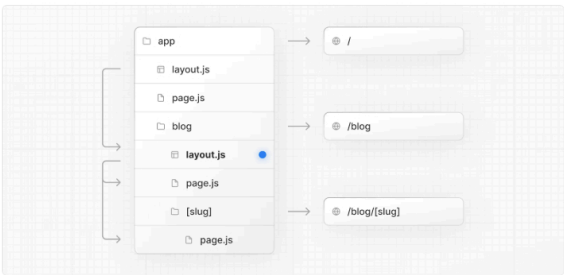
```tsx
function generateStaticParams() {}

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.



app/blog/layout.tsx — TypeScript

```tsx
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.

app/blog/[slug]/page.tsx — TypeScript

```tsx
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.

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:

app/ui/post.tsx — TypeScript

```tsx
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}`}>{post.title}</Link>
        </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.

API Reference

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

Linking and Navigating

Learn how the built-in navigation optimizations work, including prefetching, prerendering, and client-...

layout.js

API reference for the layout.js file.

page.js

Link Component

API reference for the page.js file.

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

Dynamic Segments

Dynamic Route Segments can be used to programmatically generate route segments from dynamic data.

Previous
Project Structure

Next
Linking and Navigating

Using App Router
Features available in /app

Using Latest Version
15.3.3

Getting Started
Installation
Project Structure
Layouts and Pages
Linking and Navigating
Server and Client Components
Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

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 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.

app/layout.tsx — TypeScript

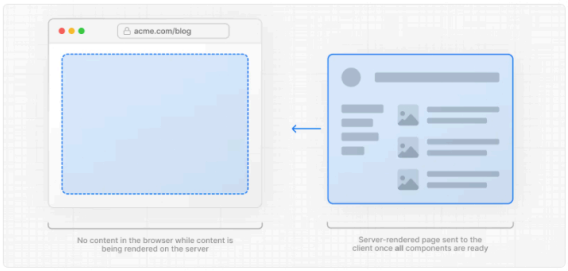
```tsx
import Link from 'next/link'

export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <html>
      <body>
        <nav>
          {/* Prefetched when the link is hovered or enters the viewport */}
          <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.

Edit this page on GitHub

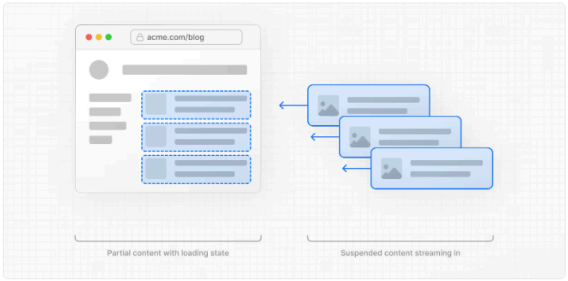


To improve the navigation experience to dynamic routes, you can use streaming.

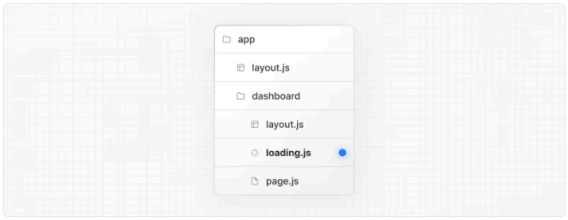
Streaming

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:



app/dashboard/loading.tsx — TypeScript

```tsx
export default function Loading() {
  // Add fallback UI that will be shown while the route is loading.
  return <LoadingSkeleton />
}
```

Behind the scenes, Next.js will automatically wrap the `page.tsx` contents and 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.

app/blog/[slug]/loading.tsx — TypeScript

```tsx
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`:

app/blog/[slug]/page.tsx — TypeScript

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

  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.

```tsx
'use client'

import { useLinkStatus } from 'next/link'

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


```tsx
'use client'

import { useLinkStatus } from 'next/link'

export default function LoadingIndicator() {
  const { pending } = useLinkStatus()
  return pending ? (
    <div role="status" aria-label="Loading" className="spinner" />
  ) : 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.

```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);
  }
}
```

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).

```jsx
<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 resouce usage without fully disabling prefetch, you can prefetch only on hover. This limits prefetching to routes the user is more *likely* to visit, rather all links in the viewport.

app/ui/hover-prefetch-link.tsx — TypeScript

```tsx
'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:

```tsx
'use client'

import { useSearchParams } from 'next/navigation'

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

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

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

`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:

```tsx
'use client'

import { usePathname } from 'next/navigation'

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

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

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

Next Steps

Link Component

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

loading.js

API reference for the loading.js file.

Prefetching

Learn how to configure prefetching in Next.js

Previous
Layouts and Pages

Next
Server and Client Components

Next.js Conf

Previews

Using App Router
Features available in /app

Using Latest Version
15.3.3

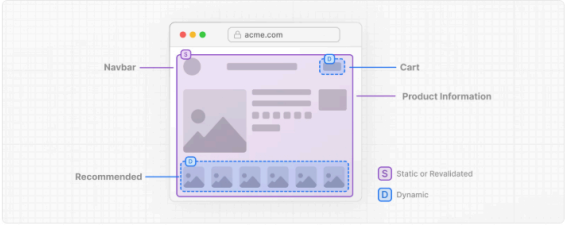
Layouts and Pages
Linking and Navigating
Server and Client Components
Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

Guides
Analytics
Authentication

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

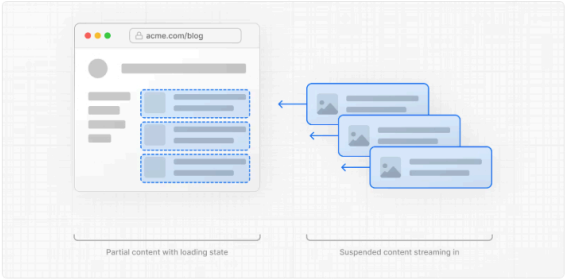
content and enable streaming

app/page.js

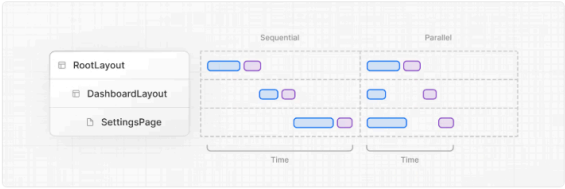
```jsx
import { Suspense } from 'react'
import StaticComponent from './StaticComponent'
import DynamicComponent from './DynamicComponent'
import Fallback from './Fallback'

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:

next.config.ts TypeScript

```ts
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:

/app/dashboard/layout.tsx

```ts
export const experimental_ppr = true

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

/app/dashboard/layout.js

```jsx
export const experimental_ppr = true
```

```tsx
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:

app/user.tsx — TypeScript

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

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

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

app/page.tsx — TypeScript

```tsx
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:

app/page.tsx — TypeScript

```tsx
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.

app/table.tsx — TypeScript

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

Next Steps

Learn more about the config option for Partial Prerendering.

ppr

Learn how to enable Partial Prerendering in Next.js.

Previous
Server and Client Components

Next
Fetching Data and Streaming

Using App Router
Features available in /app

Using Latest Version
15.3.3

Getting Started
Installation
Project Structure
Layouts and Pages
Linking and Navigating
Server and Client Components
Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

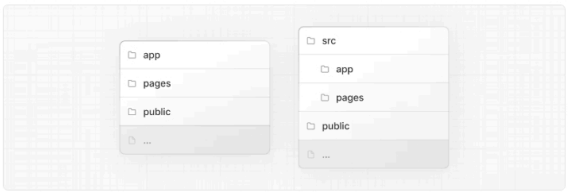
Project structure and organization

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

Routing Files

`layout`	`.js` `.jsx` `.tsx`	Layout
`page`	`.js` `.jsx` `.tsx`	Page
`loading`	`.js` `.jsx` `.tsx`	Loading UI
`not-found`	`.js` `.jsx` `.tsx`	Not found UI
`error`	`.js` `.jsx` `.tsx`	Error UI
`global-error`	`.js` `.jsx` `.tsx`	Global error UI
`route`	`.js` `.ts`	API endpoint
`template`	`.js` `.jsx` `.tsx`	Re-rendered layout
`default`	`.js` `.jsx` `.tsx`	Parallel route fallback page

Edit this page on GitHub

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

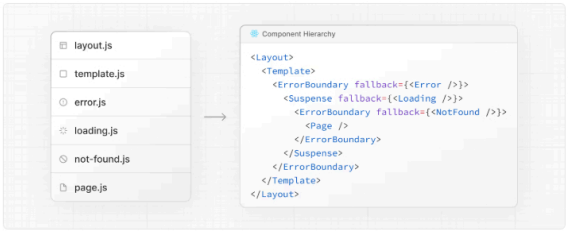
Next.js is **unopinionated** about how you organize and colocate your project files. But it does provide several features to help you organize your project.

Component hierarchy

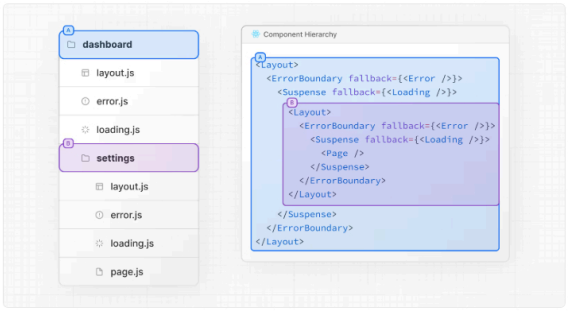
Component hierarchy

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.

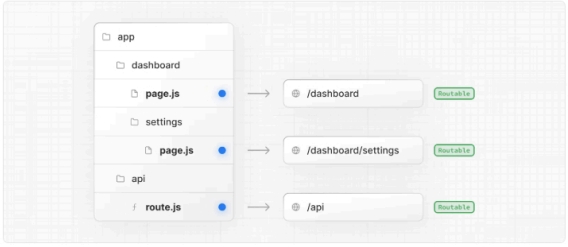


Colocation

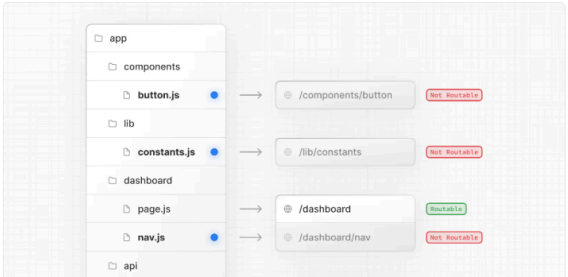
In the `app` directory, nested folders define route structure. Each folder represents a route segment that is mapped to a corresponding segment in a URL path.

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.

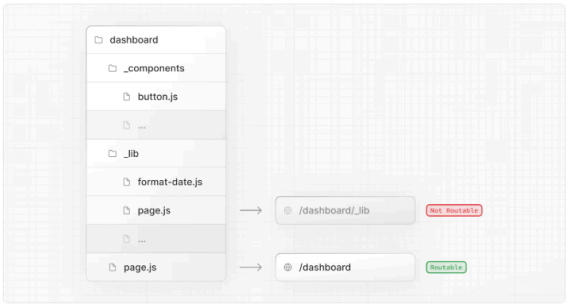


Good to know: While you **can** colocate your project files in `app` you don't **have** to. If you prefer, you can keep them outside the `app` directory.

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.

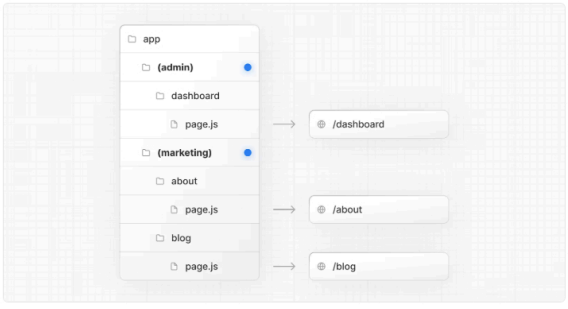
Good to know:

- 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.

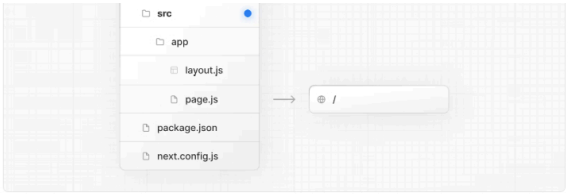


Route groups are useful for:

- 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.



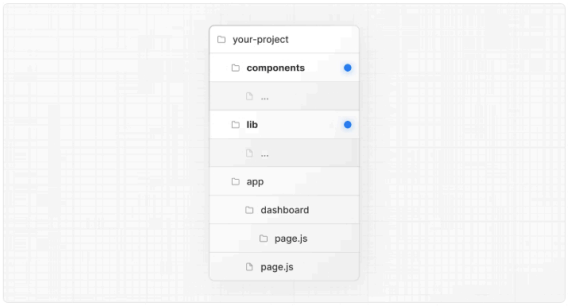
Examples

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.

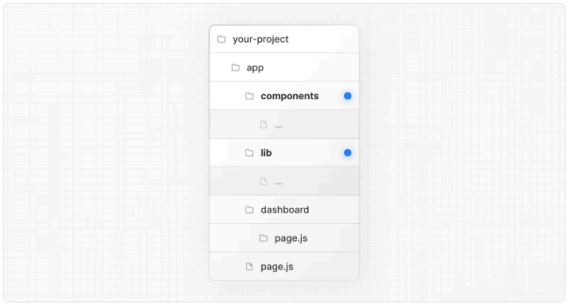
Store project files outside of `app`

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



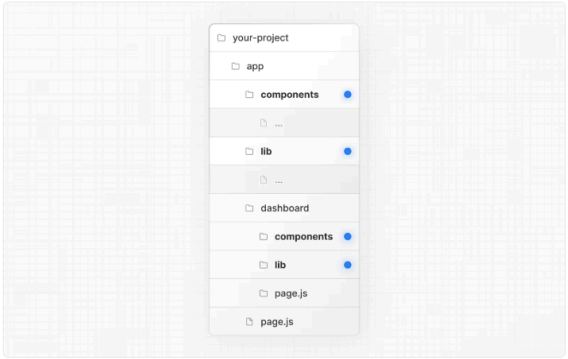
Store project files in top-level folders inside of `app`

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



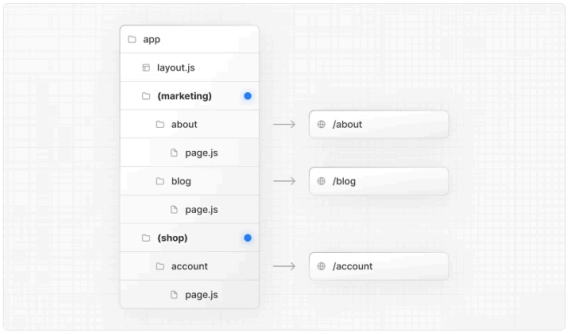
Split project files by feature or route

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.

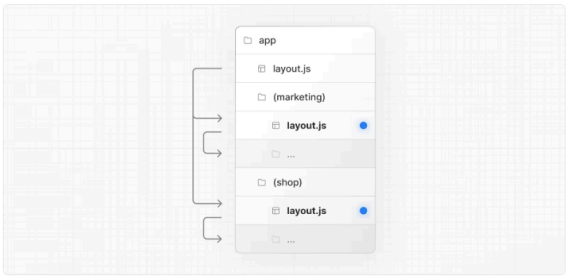


Organize routes without affecting the URL path

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)`).

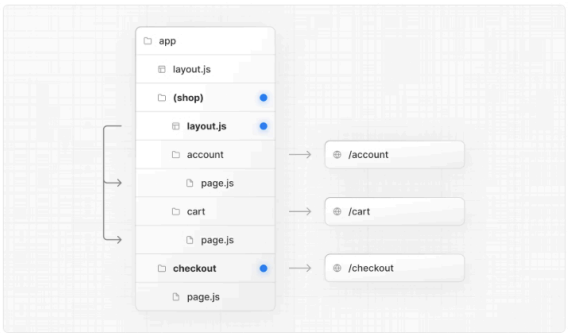


Even though routes inside `(marketing)` and `(shop)` share the same URL hierarchy, you can create a different layout for each group by adding a `layout.js` file inside their folders.



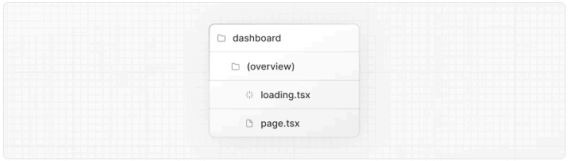
Opting specific segments into a layout

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

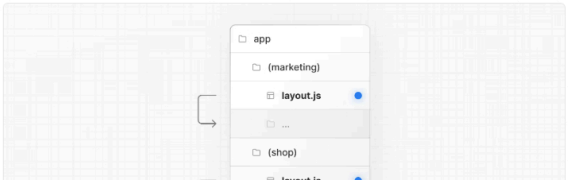
To apply a loading skeleton via a `loading.js` file to a specific route, create a new route group (e.g., `/(overview)`) and then move your `loading.tsx` inside that route group.



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.

Previous
Installation

Next
Layouts and Pages

Using App Router
Features available in /app

Using Latest Version
15.3.3

Getting Started
Installation
Project Structure
Layouts and Pages
Linking and Navigating
Server and Client Components
Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

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.

app/[id]/page.tsx — TypeScript

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

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

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

app/ui/like-button.tsx — TypeScript

```tsx
'use client'

import { useState } from 'react'

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

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.

Edit this page on GitHub

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.

app/ui/counter.tsx — TypeScript

```tsx
'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)}>Click me</button>
    </div>
  )
}
```

`"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.

app/layout.tsx — TypeScript

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

```
}
```

app/ui/search.tsx — TypeScript

```tsx
'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.

app/[id]/page.tsx — TypeScript

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

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

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

app/ui/like-button.tsx — TypeScript

```tsx
'use client'

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

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.

app/ui/modal.tsx — TypeScript

```tsx
'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>`:

app/page.tsx — TypeScript

```tsx
import Modal from './ui/modal'
import Cart from './ui/cart'

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

app/theme-provider.tsx — TypeScript

```tsx
'use client'

import { createContext } from 'react'
```

```tsx
export const ThemeContext = createContext({})

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

Then, import it into a Server Component (e.g. `layout`):

app/layout.tsx — TypeScript

```tsx
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:

app/gallery.tsx — TypeScript

```tsx
'use client'

import { useState } from 'react'
import { Carousel } from 'acme-carousel'

export default function Gallery() {
  const [isOpen, setIsOpen] = useState(false)

  return (
    <div>
      <button onClick={() => setIsOpen(true)}>View pictures</button>
      {/* Works, since Carousel is used within a Client Component */}
      {isOpen && <Carousel />}
    </div>
  )
}
```

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

app/carousel.tsx — TypeScript

```tsx
'use client'

import { Carousel } from 'acme-carousel'

export default Carousel
```

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

app/page.tsx — TypeScript

```tsx
import Carousel from './carousel'

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

lib/data.ts — TypeScript

```ts
export async function getData() {
  const res = await fetch('https://external-service.com/data', {
    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:

lib/data.js

```js
import 'server-only'

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

  return res.json()
}
```

Now, 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 render a component on the client.

Previous
Linking and Navigating

Next
Partial Prerendering

Using App Router
Features available in /app

Using Latest Version
15.3.3

Layouts and Pages
Linking and Navigating
Server and Client Components
Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

Guides

Analytics
Authentication

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.

app/lib/actions.ts — TypeScript

```ts
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:

app/page.tsx — TypeScript

```tsx
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:

Edit this page on GitHub

app/actions.ts TypeScript

```ts
'use server'

export async function createPost() {}
```

app/ui/button.tsx TypeScript

```ts
'use client'

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

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

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

app/client-component.tsx TypeScript

```ts
'use client'

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

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

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:

app/ui/form.tsx TypeScript

```ts
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>
  )
}
```

app/actions.ts TypeScript

```ts
'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`.

app/like-button.tsx TypeScript

```ts
'use client'
```

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

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

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

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:

app/ui/button.tsx — TypeScript

```tsx
'use client'

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

export function Button() {
  const [state, action, pending] = useActionState(createPost, false)

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

app/lib/actions.ts — TypeScript

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

export async function createPost(formData: FormData) {
  '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:

app/lib/actions.ts — TypeScript

```tsx
'use server'

import { redirect } from 'next/navigation'

export async function createPost(formData: FormData) {
  // Update data
  // ...

  redirect('/posts')
}
```

Cookies

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

app/actions.ts — TypeScript

```tsx
'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:

app/view-count.tsx — TypeScript

```tsx
'use client'

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

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

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

  // You can use `isPending` to give users feedback
  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.

Previous
Fetching Data and Streaming

Next
Error Handling

Using App Router
Features available in /app
Using Latest Version
15.3.3

Partial Prerendering
Fetching Data and Streaming
Updating data
Error Handling
Caching and Revalidating
CSS
Image Optimization
Font Optimization
Metadata and OG images
Deploying
Upgrading

Guides
Analytics
Authentication
CI Build Caching
Content Security Policy
CSS-in-JS

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@latest eslint-config-next@latest
```

Good to know:

- If you see a peer dependencies warning, you may need to update `react` and `react-dom` to the suggested versions, or you use the `--force` or `--legacy-peer-deps` flag to ignore the warning. This won't be necessary once both Next.js 15 and React 19 are stable.

React 19

- The minimum versions of `react` and `react-dom` is now 19.
- `useFormState` has been replaced by `useActionState`. The `useFormState` hook is still available in React 19, but it is deprecated and will be removed in a future release. `useActionState` is recommended and includes additional properties like reading the `pending` state directly. Learn more.
- `useFormStatus` now includes additional keys like `data`, `method`, and `action`. If you are not using React 19, only the `pending` key is available. Learn more.
- Read more in the React 19 upgrade guide.

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
```


Edit this page on GitHub

```ts
import { cookies, type UnsafeUnwrappedCookies } from 'next/headers'

// Before
const cookieStore = cookies()
const token = cookieStore.get('token')

// After
const cookieStore = cookies() as unknown as UnsafeUnwrappedCookies
// will log a warning in dev
const token = cookieStore.get('token')
```

`headers`

Recommended Async Usage

```ts
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

```ts
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

```ts
import { draftMode } from 'next/headers'

// Before
const { isEnabled } = draftMode()

// After
const { isEnabled } = await draftMode()
```

Temporary Synchronous Usage

app/page.tsx TypeScript

```ts
import { draftMode, type UnsafeUnwrappedDraftMode } from 'next/headers'

// Before
const { isEnabled } = draftMode()

// After
// will log a warning in dev
const { isEnabled } = draftMode() as unknown as UnsafeUnwrappedDraftMode
```

`params` & `searchParams`

Asynchronous Layout

app/layout.tsx TypeScript

```ts
// Before
type Params = { slug: string }

export function generateMetadata({ params }: { params: Params }) {
  const { slug } = params
}

export default async function Layout({
  children,
  params,
}: {
  children: React.ReactNode
  params: Params
}) {
  const { slug } = params
}

// After
type Params = Promise<{ slug: string }>

export async function generateMetadata({ params }: { params: Params }) {
  const { slug } = await params
}

export default async function Layout({
  children,
  params,
}: {
  children: React.ReactNode
```

```
  params: Params
}) {
  const { slug } = await params
}
```

Synchronous Layout

app/layout.tsx — TypeScript

```tsx
// 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

app/page.tsx — TypeScript

```tsx
// Before
type Params = { slug: string }
type SearchParams = { [key: string]: string | string[] | undefined }

export function generateMetadata({
  params,
  searchParams,
}: {
  params: Params
  searchParams: SearchParams
}) {
  const { slug } = params
  const { query } = searchParams
}

export default async function Page({
  params,
  searchParams,
}: {
  params: Params
  searchParams: SearchParams
}) {
  const { slug } = params
  const { query } = searchParams
}

// After
type Params = Promise<{ slug: string }>
type SearchParams = Promise<{ [key: string]: string | string[] | undefined

export async function generateMetadata(props: {
  params: Params
  searchParams: SearchParams
}) {
  const params = await props.params
  const searchParams = await props.searchParams
  const slug = params.slug
  const query = searchParams.query
}

export default async function Page(props: {
  params: Params
  searchParams: SearchParams
}) {
  const params = await props.params
  const searchParams = await props.searchParams
  const slug = params.slug
  const query = searchParams.query
}
```

Synchronous Page

```tsx
'use client'

// Before
type Params = { slug: string }
type SearchParams = { [key: string]: string | string[] | undefined }

export default function Page({
  params,
  searchParams,
}: {
  params: Params
  searchParams: SearchParams
}) {
  const { slug } = params
  const { query } = searchParams
}
```

```
17
18  // After
19  import { use } from 'react'
20
21  type Params = Promise<{ slug: string }>
22  type SearchParams = Promise<{ [key: string]: string | string[] | undefined
23
24  export default function Page(props: {
25    params: Params
26    searchParams: SearchParams
27  }) {
28    const params = use(props.params)
29    const searchParams = use(props.searchParams)
30    const slug = params.slug
31    const query = searchParams.query
32  }
```

```
1   // Before
2   export default function Page({ params, searchParams }) {
3     const { slug } = params
4     const { query } = searchParams
5   }
6
7   // After
8   import { use } from "react"
9
10  export default function Page(props) {
11    const params = use(props.params)
12    const searchParams = use(props.searchParams)
13    const slug = params.slug
14    const query = searchParams.query
15  }
16
```

Route Handlers

app/api/route.ts

```
1   // Before
2   type Params = { slug: string }
3
4   export async function GET(request: Request, segmentData: { params: Params }
5     const params = segmentData.params
6     const slug = params.slug
7   }
8
9   // After
10  type Params = Promise<{ slug: string }>
11
12  export async function GET(request: Request, segmentData: { params: Params }
13    const params = await segmentData.params
14    const slug = params.slug
15  }
```

app/api/route.js

```
1   // Before
2   export async function GET(request, segmentData) {
3     const params = segmentData.params
4     const slug = params.slug
5   }
6
7   // After
8   export async function GET(request, segmentData) {
9     const params = await segmentData.params
10    const slug = params.slug
11  }
```

`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.

app/layout.js

```
1   export default async function RootLayout() {
2     const a = await fetch('https://...') // Not Cached
3     const b = await fetch('https://...', { cache: 'force-cache' }) // Cached
4
5     // ...
6   }
```

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.

app/layout.js

```
// Since this is the root layout, all fetch requests in the app
// that don't set their own cache option will be cached.
export const fetchCache = 'default-cache'

export default async function RootLayout() {
  const a = await fetch('https://...') // Cached
  const b = await fetch('https://...', { cache: 'no-store' }) // Not cached

  // ...
}
```

Route Handlers

`GET` functions in Route Handlers are no longer cached by default. To opt `GET` methods into caching, you can use a route config option such as `export const dynamic = 'force-static'` in your Route Handler file.

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:

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.

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`.

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`.

next.config.js

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  // Before
  experimental: {
    serverComponentsExternalPackages: ['package-name'],
  },

  // After
  serverExternalPackages: ['package-name'],
}

module.exports = nextConfig
```

Speed Insights

Auto instrumentation for Speed Insights was removed in Next.js 15.

To continue using Speed Insights, follow the Vercel Speed Insights Quickstart ↗ guide.

`NextRequest` Geolocation

The `geo` and `ip` properties on `NextRequest` have been removed as these values are provided by your hosting provider. A codemod is available to automate this migration.

If you are using Vercel, you can alternatively use the `geolocation` and `ipAddress` functions from `@vercel/functions` ↗ instead:

middleware.ts

```
import { geolocation } from '@vercel/functions'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  const { city } = geolocation(request)

  // ...
}
```

middleware.ts

```
import { ipAddress } from '@vercel/functions'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  const ip = ipAddress(request)

  // ...
}
```

Previous
Version 14

Next
Videos