IntroductionWelcome to the Next.js documentation!
What is Next.js?

Next.js is a framework for building web applications.
With Next.js, you can build user interfaces using React components. Then, Next.js provides additional structure, features, and optimizations for your application.
Under the hood, Next.js also abstracts and automatically configures tooling for you, like bundling, compiling, and more. This allows you to focus on building your application instead of spending time setting up tooling.
Whether you're an individual developer or part of a larger team, Next.js can help you build interactive, dynamic, and fast web applications.
Main Features

Some of the main Next.js features include:
FeatureDescriptionRoutingA file-system based router built on top of Server Components that supports layouts, nested routing, loading states, error handling, and more.RenderingClient-side and Server-side Rendering with Client and Server Components. Further optimized with Static and Dynamic Rendering on the server with Next.js. Streaming on Edge and Node.js runtimes.Data FetchingSimplified data fetching with async/await support in React Components and the fetch()s API that aligns with React and the Web Platform.StylingSupport for your preferred styling methods, including CSS Modules, Tailwind CSS, and CSS-in-JSOptimizationsImage, Fonts, and Script Optimizations to improve your application's Core Web Vitals and User Experience.TypeScriptImproved support for TypeScript, with better type checking and more efficient compilation, as well as custom TypeScript Plugin and type checker.API ReferenceUpdates to the API design throughout Next.js. Please refer to the API Reference Section for new APIs.
How to Use These Docs

The sections and pages of the docs are organized sequentially, from basic to advanced, so you can follow them step-by-step when building your Next.js application. However, you can read them in any order or skip to the pages that apply to your use case.
At the top of the sidebar, you'll notice a dropdown menu that allows you to switch

between the App Router and the Pages Router features. Since there are features that are unique to each directory, it's important to keep track of which tab is selected.
On the right side of the page, you'll see a table of contents that makes it easier to navigate between sections of a page. The breadcrumbs at the top of the page will also indicate whether you're viewing App Router docs or Pages Router docs.
To get started, checkout the Installation. If you're new to React or Server Components, we recommend reading the React Essentials page.
Pre-Requisite Knowledge

Although our docs are designed to be beginner-friendly, we need to establish a baseline so that the docs can stay focused on Next.js functionality. We'll make sure to provide links to relevant documentation whenever we introduce a new concept.
To get the most out of our docs, it's recommended that you have a basic understanding of HTML, CSS, and React. If you need to brush up on your React skills, check out these resources:

React: Official React Documentation
React Essentials

Accessibility

For optimal accessibility when using a screen reader while reading the docs, we recommend using Firefox and NVDA, or Safari and VoiceOver.
Join our Community

If you have questions about anything related to Next.js, you're always welcome to ask our community on GitHub Discussions, Discord, Twitter, and Reddit.
InstallationSystem Requirements:

Node.js 16.8 or later.
macOS, Windows (including WSL), and Linux are supported.

Automatic Installation

We recommend creating a new Next.js app using create-next-app, which sets up everything automatically for you. To create a project, run:
Terminal npx create-next-app@latest
On installation, you'll see the following prompts:
Terminal What is your project named? my-app
Would you like to use TypeScript? No / Yes
Would you like to use ESLint? No / Yes
Would you like to use Tailwind CSS? No / Yes
Would you like to use `src/` directory? No / Yes
Would you like to use App Router? (recommended) No / Yes
Would you like to customize the default import alias? No / Yes
Next.js now ships with TypeScript, ESLint, and Tailwind CSS configuration by default. You can also choose to use the src directory for your application code.
After the prompts, create-next-app will create a folder with your project name and install the required dependencies.

Good to know: While you can use the Pages Router in your new project. We recommend starting new applications with the App Router to leverage React's latest features.

Manual Installation

To manually create a new Next.js app, install the required packages:
Terminal npm install next@latest react@latest react-dom@latest
Open package.json and add the following scripts:
package.json {
```
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  }
}
```
These scripts refer to the different stages of developing an application:

dev: runs next dev to start Next.js in development mode.
build: runs next build to build the application for production usage.
start: runs next start to start a Next.js production server.

lint: runs next lint to set up Next.js' built-in ESLint configuration.

Creating directories

Next.js uses file-system routing, which means how you structure your files determines the routes in your application.
The app directory

For new applications, we recommend using the App Router. This router allows you to use React's latest features and is an evolution of the Pages Router based on community feedback.
To use the app router, create an app/ folder, then add a layout.tsx and page.tsx file. These will be rendered when the user visits the root of your application (/).

Create a root layout inside app/layout.tsx with the required <html> and <body> tags:
app/layout.tsxTypeScript export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
)
}

Finally, create a home page app/page.tsx with some initial content:
app/page.tsxTypeScript export default function Page() {
  return <h1>Hello, Next.js!</h1>
}

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

Learn more about using the App Router.
The pages directory (optional)

If you prefer to use the Pages Router instead of the App Router, you can create a pages/ directory at the root of your project.
Then, add an index.tsx file inside your pages folder. This will be your home page (/):
pages/index.tsxTypeScript export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
Next, add an _app.tsx file inside pages/ to define the global layout. Learn more about the custom App file).
pages/_app.tsxTypeScript import type { AppProps } from 'next/app'

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

Finally, add a _document.tsx file inside pages/ to control the initial response from the server. Learn more about the custom Document file.
pages/_document.tsx import { Html, Head, Main, NextScript } from 'next/document'

export default function Document() {
  return (
    <Html>
      <Head />
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
)
}
Learn more about using the Pages Router.

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

The public folder (optional)

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

Run npm run dev to start the development server.
Visit http://localhost:3000 to view your application.
Edit app/layout.tsx (or pages/index.tsx) file and save it to see the updated result in your browser.
Next.js Project StructureThis page provides an overview of the file and folder structure of a Next.js project. It covers top-level files and folders, configuration files, and routing conventions within the app and pages directories.
Top-level files

Next.jsnext.config.jsConfiguration file for Next.jsmiddleware.tsNext.js request middlewareinstrumentation.tsOpenTelemetry and Instrumentation.envEnvironment variables.env.localLocal environment variables.env.productionProduction environment variables.env.developmentDevelopment environment variables.next-env.d.tsTypeScript declaration file for Next.jsEcosystempackage.jsonProject dependencies and scripts.gitignoreGit files and folders to ignoretsconfig.jsonConfiguration file for TypeScriptjsconfig.jsonConfiguration file for JavaScript.eslintrc.jsonConfiguration file for ESLint
Top-level folders

appApp RouterpagesPages RouterpublicStatic assets to be servedsrcOptional application source folder
app Routing Conventions

Routing Files

layout.js .jsx .tsxLayoutpage.js .jsx .tsxPageloading.js .jsx .tsxLoading UInot-found.js .jsx .tsxNot found UIerror.js .jsx .tsxError UIglobal-error.js .jsx .tsxGlobal error UIroute.js .tsAPI endpointtemplate.js .jsx .tsxRe-rendered layoutdefault.js .jsx .tsxParallel route fallback page
Nested Routes

folderRoute segmentfolder/folderNested route segment
Dynamic Routes

[folder]Dynamic route segment[...folder]Catch-all segments[[...folder]]Optional catch-all segments
Route Groups and Private Folders

(folder)Group routes without affecting routing_folderOpt folder and all child segments out of routing
Parallel and Intercepted Routes

@folderNamed slot(.)folderIntercept same level(..)folderIntercept one level above(..)(..)folderIntercept two levels above(...)folderIntercept from root
Metadata File Conventions

App Icons

favicon.icoFavicon fileicon.ico .jpg .jpeg .png .svgApp Icon fileicon.js .ts .tsxGenerated App Iconapple-icon.jpg .jpeg, .pngApple App Icon fileapple-icon.js .ts .tsxGenerated Apple App Icon
Open Graph and Twitter Images

opengraph-image.jpg .jpeg .png .gifOpen Graph image fileopengraph-image.js .ts .tsxGenerated Open Graph imagetwitter-image.jpg .jpeg .png .gifTwitter image filetwitter-image.js .ts .tsxGenerated Twitter image
SEO

sitemap.xmlSitemap filesitemap.js .tsGenerated Sitemaprobots.txtRobots filerobots.js .tsGenerated Robots file
pages Routing Conventions

Special Files

_app.js .jsx .tsxCustom App_document.js .jsx .tsxCustom Document_error.js .jsx .tsxCustom Error Page404.js .jsx .tsx404 Error Page500.js .jsx .tsx500 Error Page
Routes

Folder conventionindex.js .jsx .tsxHome pagefolder/index.js .jsx .tsxNested pageFile conventionindex.js .jsx .tsxHome pagefile.js .jsx .tsxNested page
Dynamic Routes

Folder convention[folder]/index.js .jsx .tsxDynamic route segment[...folder]/ index.js .jsx .tsxCatch-all segments[[...folder]]/index.js .jsx .tsxOptional catch-all segmentsFile convention[file].js .jsx .tsxDynamic route segment[...file].js .jsx .tsxCatch-all segments[[...file]].js .jsx .tsxOptional catch-all segments
React EssentialsTo build applications with Next.js, it helps to be familiar with React's newer features such as Server Components. This page will go through the differences between Server and Client Components, when to use them, and recommended patterns.
If you're new to React, we also recommend referring to the React Docs. Here are some great resources for learning:

React Tutorial
Thinking in React
Learn React

Server Components

Server and Client Components allow developers to build applications that span the server and client, combining the rich interactivity of client-side apps with the improved performance of traditional server rendering.
Thinking in Server Components

Similar to how React changed the way we think about building UIs, React Server Components introduce a new mental model for building hybrid applications that leverage the server and the client.
Instead of React rendering your whole application client-side (such as in the case of Single-Page Applications), React now gives you the flexibility to choose where to render your components based on their purpose.
For example, consider a page in your application:

If we were to split the page into smaller components, you'll notice that the majority of components are non-interactive and can be rendered on the server as Server Components. For smaller pieces of interactive UI, we can sprinkle in Client Components. This aligns with Next.js server-first approach.

Why Server Components?

So, you may be thinking, why Server Components? What are the advantages of using them over Client Components?

Server Components allow developers to better leverage server infrastructure. For example, you can move data fetching to the server, closer to your database, and keep large dependencies that previously would impact the client JavaScript bundle size on the server, leading to improved performance. Server Components make writing a React application feel similar to PHP or Ruby on Rails, but with the power and flexibility of React and the components model for templating UI.

With Server Components, the initial page load is faster, and the client-side JavaScript bundle size is reduced. The base client-side runtime is cacheable and predictable in size, and does not increase as your application grows. Additional JavaScript is only added as client-side interactivity is used in your application through Client Components.

When a route is loaded with Next.js, the initial HTML is rendered on the server. This HTML is then progressively enhanced in the browser, allowing the client to take over the application and add interactivity, by asynchronously loading the Next.js and React client-side runtime.

To make the transition to Server Components easier, all components inside the App Router are Server Components by default, including special files and colocated components. This allows you to automatically adopt them with no extra work, and achieve great performance out of the box. You can also optionally opt-in to Client Components using the 'use client' directive.

Client Components

Client Components enable you to add client-side interactivity to your application. In Next.js, they are pre-rendered on the server and hydrated on the client. You can think of Client Components as how components in the Pages Router have always worked.

The "use client" directive

The "use client" directive is a convention to declare a boundary between a Server and Client Component module graph.
app/counter.tsxTypeScript 'use client'

```
import { useState } from 'react'

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

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

"use client" sits between server-only and client code. It's placed at the top of a file, above imports, to define the cut-off point where it crosses the boundary from the server-only to the client part. Once "use client" is defined in a file, all other modules imported into it, including child components, are considered part of the client bundle.
Since Server Components are the default, all components are part of the Server Component module graph unless defined or imported in a module that starts with the "use client" directive.

Good to know:

Components in the Server Component module graph are guaranteed to be only rendered on the server.
Components in the Client Component module graph are primarily rendered on the client, but with Next.js, they can also be pre-rendered on the server and hydrated on the client.
The "use client" directive must be defined at the top of a file before any imports.
"use client" does not need to be defined in every file. The Client module boundary only needs to be defined once, at the "entry point", for all modules imported into it to be considered a Client Component.

When to use Server and Client Components?

To simplify the decision between Server and Client Components, we recommend using Server Components (default in the app directory) until you have a use case for a Client Component.
This table summarizes the different use cases for Server and Client Components:
What do you need to do?Server ComponentClient ComponentFetch data.Access backend resources (directly)Keep sensitive information on the server (access tokens, API keys, etc)Keep large dependencies on the server / Reduce client-side JavaScriptAdd interactivity and event listeners (onClick(), onChange(), etc)Use State and Lifecycle Effects (useState(), useReducer(), useEffect(), etc)Use browser-only APIsUse custom hooks that depend on state, effects, or browser-only APIsUse React Class components
Patterns

Moving Client Components to the Leaves

To improve the performance of your application, we recommend moving Client Components to the leaves of your component tree where possible.
For example, you may have a Layout that has static elements (e.g. logo, links, etc) and an interactive search bar that uses state.
Instead of making the whole layout a Client Component, move the interactive logic to a Client Component (e.g. <SearchBar />) and keep your layout as a Server Component. This means you don't have to send all the component Javascript of the layout to the client.
app/layout.tsxTypeScript // SearchBar is a Client Component
import SearchBar from './searchbar'
// Logo is a Server Component
import Logo from './logo'

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

```
  )
}
```

Composing Client and Server Components

Server and Client Components can be combined in the same component tree.
Behind the scenes, React handles rendering as follows:

On the server, React renders all Server Components before sending the result to the client.

This includes Server Components nested inside Client Components.
Client Components encountered during this stage are skipped.

On the client, React renders Client Components and slots in the rendered result of Server Components, merging the work done on the server and client.

If any Server Components are nested inside a Client Component, their rendered content will be placed correctly within the Client Component.

Good to know: In Next.js, during the initial page load, both the rendered result of Server Components from the above step and Client Components are pre-rendered on the server as HTML to produce a faster initial page load.

Nesting Server Components inside Client Components

Given the rendering flow outlined above, there is a restriction around importing a Server Component into a Client Component, as this approach would require an additional server round trip.
Unsupported Pattern: Importing Server Components into Client Components

The following pattern is not supported. You cannot import a Server Component into a Client Component:
app/example-client-component.tsxTypeScript 'use client'

// This pattern will **not** work!
// You cannot import a Server Component into a Client Component.
import ExampleServerComponent from './example-server-component'

export default function ExampleClientComponent({
  children,
}: {
  children: React.ReactNode
}) {
  const [count, setCount] = useState(0)

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button>

      <ExampleServerComponent />
    </>
)
}

Recommended Pattern: Passing Server Components to Client Components as Props

Instead, when designing Client Components you can use React props to mark "slots" for Server Components.
The Server Component will be rendered on the server, and when the Client Component is rendered on the client, the "slot" will be filled in with the rendered result of the Server Component.
A common pattern is to use the React children prop to create the "slot". We can refactor <ExampleClientComponent> to accept a generic children prop and move the import and explicit nesting of <ExampleClientComponent> up to a parent component.
app/example-client-component.tsxTypeScript 'use client'

import { useState } from 'react'

export default function ExampleClientComponent({
  children,
}: {

```
  children: React.ReactNode
}) {
  const [count, setCount] = useState(0)

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button>

      {children}
    </>
  )
}
```

Now, <ExampleClientComponent> has no knowledge of what children is. It doesn't know that children will eventually be filled in by the result of a Server Component. The only responsibility ExampleClientComponent has is to decide where whatever children will eventually be placed.
In a parent Server Component, you can import both the <ExampleClientComponent> and <ExampleServerComponent> and pass <ExampleServerComponent> as a child of <ExampleClientComponent>:
app/page.tsxTypeScript // This pattern works:

```
// You can pass a Server Component as a child or prop of a
// Client Component.
import ExampleClientComponent from './example-client-component'
import ExampleServerComponent from './example-server-component'

// Pages in Next.js are Server Components by default
export default function Page() {
  return (
    <ExampleClientComponent>
      <ExampleServerComponent />
    </ExampleClientComponent>
  )
}
```

With this approach, the rendering of <ExampleClientComponent> and <ExampleServerComponent> are decoupled and can be rendered independently - aligning with Server Components, which are rendered on the server before Client Components.

Good to know

This pattern is already applied in layouts and pages with the children prop so you don't have to create an additional wrapper component.
Passing React components (JSX) to other components is not a new concept and has always been part of the React composition model.

This composition strategy works across Server and Client Components because the component that receives the prop has no knowledge of what the prop is. It is only responsible for where the thing that it is passed should be placed.

This allows the passed prop to be rendered independently, in this case, on the server, well before the Client Component is rendered on the client.
The very same strategy of "lifting content up" has been used to avoid state changes in a parent component re-rendering an imported nested child component.

You're not limited to the children prop. You can use any prop to pass JSX.

Passing props from Server to Client Components (Serialization)

Props passed from the Server to Client Components need to be serializable. This means that values such as functions, Dates, etc, cannot be passed directly to Client Components.

Where is the Network Boundary?
In the App Router, the network boundary is between Server Components and Client Components. This is different from the Pages where the boundary is between getStaticProps/getServerSideProps and Page Components. Data fetched inside Server Components do not need to be serialized as it doesn't cross the network boundary unless it is passed to a Client Component. Learn more about data fetching with Server Components.

Keeping Server-Only Code out of Client Components (Poisoning)

Since JavaScript modules can be shared between both Server and Client Components, it's possible for code that was only ever intended to be run on the server to sneak its way into the client.
For example, take the following data-fetching function:
lib/data.tsTypeScript export async function getData() {
  const res = await fetch('https://external-service.com/data', {
    headers: {
      authorization: process.env.API_KEY,
    },

```
  })

  return res.json()
}
```

At first glance, it appears that getData works on both the server and the client. But because the environment variable API_KEY is not prefixed with NEXT_PUBLIC, it's a private variable that can only be accessed on the server. Next.js replaces private environment variables with the empty string in client code to prevent leaking secure information.
As a result, even though getData() can be imported and executed on the client, it won't work as expected. And while making the variable public would make the function work on the client, it would leak sensitive information.
So, this function was written with the intention that it would only ever be executed on the server.
The "server only" package

To prevent this sort of unintended client usage of server code, we can use the server-only package to give other developers a build-time error if they ever accidentally import one of these modules into a Client Component.
To use server-only, first install the package:
Terminal npm install server-only
Then import the package into any module that contains server-only code:
lib/data.js import 'server-only'

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

  return res.json()
}
```

Now, any Client Component that imports getData() will receive a build-time error explaining that this module can only be used on the server.
The corresponding package client-only can be used to mark modules that contain client-only code – for example, code that accesses the window object.
Data Fetching

Although it's possible to fetch data in Client Components, we recommend fetching data in Server Components unless you have a specific reason for fetching data on the client. Moving data fetching to the server leads to better performance and user experience.
Learn more about data fetching.
Third-party packages

Since Server Components are new, third-party packages in the ecosystem are just beginning to add the "use client" directive to components that use client-only features like useState, useEffect, and createContext.
Today, many components from npm packages that use client-only features do not yet have the directive. These third-party components will work as expected within your own Client Components since they have the "use client" directive, but they won't work within Server Components.
For example, let's say you've installed the hypothetical acme-carousel package which has a <Carousel /> component. This component uses useState, but it doesn't yet have the "use client" directive.
If you use <Carousel /> within a Client Component, it will work as expected:
app/gallery.tsxTypeScript 'use client'

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

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

  return (
    <div>
      <button onClick={() => setIsOpen(true)}>View pictures</button>

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

However, if you try to use it directly within a Server Component, you'll see an error:
app/page.tsxTypeScript import { Carousel } from 'acme-carousel'

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

```
    <div>
      <p>View pictures</p>

      {/* Error: `useState` can not be used within Server Components */}
      <Carousel />
    </div>
  )
}
```

This is because Next.js doesn't know <Carousel /> is using client-only features.
To fix this, you can wrap third-party components that rely on client-only features in your own Client Components:
app/carousel.tsxTypeScript 'use client'

```
import { Carousel } from 'acme-carousel'

export default Carousel
```

Now, you can use <Carousel /> directly within a Server Component:
app/page.tsxTypeScript import Carousel from './carousel'

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

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

We don't expect you to need to wrap most third-party components since it's likely you'll be using them within Client Components. However, one exception is provider components, since they rely on React state and context, and are typically needed at the root of an application. Learn more about third-party context providers below.
Library Authors

In a similar fashion, library authors creating packages to be consumed by other developers can use the "use client" directive to mark client entry points of their package. This allows users of the package to import package components directly into

their Server Components without having to create a wrapping boundary.
You can optimize your package by using 'use client' deeper in the tree, allowing the imported modules to be part of the Server Component module graph.
It's worth noting some bundlers might strip out "use client" directives. You can find an example of how to configure esbuild to include the "use client" directive in the React Wrap Balancer and Vercel Analytics repositories.

Context

Most React applications rely on context to share data between components, either directly via createContext, or indirectly via provider components imported from third-party libraries.
In Next.js 13, context is fully supported within Client Components, but it cannot be created or consumed directly within Server Components. This is because Server Components have no React state (since they're not interactive), and context is primarily used for rerendering interactive components deep in the tree after some React state has been updated.
We'll discuss alternatives for sharing data between Server Components, but first, let's take a look at how to use context within Client Components.
Using context in Client Components

All of the context APIs are fully supported within Client Components:
app/sidebar.tsxTypeScript 'use client'

```
import { createContext, useContext, useState } from 'react'

const SidebarContext = createContext()

export function Sidebar() {
  const [isOpen, setIsOpen] = useState()

  return (
    <SidebarContext.Provider value={{ isOpen }}>
      <SidebarNav />
    </SidebarContext.Provider>
  )
}
```

```
function SidebarNav() {
  let { isOpen } = useContext(SidebarContext)

  return (
    <div>
      <p>Home</p>

      {isOpen && <Subnav />}
    </div>
  )
}
```

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

app/layout.tsxTypeScript
```
import { createContext } from 'react'

//  createContext is not supported in Server Components
export const ThemeContext = createContext({})

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <ThemeContext.Provider value="dark">{children}</ThemeContext.Provider>
      </body>
    </html>
  )
}
```

To fix this, create your context and render its provider inside of a Client Component:

app/theme-provider.tsxTypeScript
```
'use client'

import { createContext } from 'react'

export const ThemeContext = createContext({})

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

Your Server Component will now be able to directly render your provider since it's been marked as a Client Component:

app/layout.tsxTypeScript
```
import ThemeProvider from './theme-provider'
```

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

With the provider rendered at the root, all other Client Components throughout your app will be able to consume this context.

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

Rendering third-party context providers in Server Components

Third-party npm packages often include Providers that need to be rendered near the root of your application. If these providers include the "use client" directive, they can be rendered directly inside of your Server Components. However, since Server Components are so new, many third-party providers won't have added the directive yet. If you try to render a third-party provider that doesn't have "use client", it will cause an error:
app/layout.tsxTypeScript import { ThemeProvider } from 'acme-theme'

```
export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        {/*  Error: `createContext` can't be used in Server Components */}
        <ThemeProvider>{children}</ThemeProvider>
      </body>
    </html>
  )
}
```

To fix this, wrap third-party providers in your own Client Component:
app/providers.js 'use client'

```
import { ThemeProvider } from 'acme-theme'
import { AuthProvider } from 'acme-auth'

export function Providers({ children }) {
  return (
    <ThemeProvider>
      <AuthProvider>{children}</AuthProvider>
    </ThemeProvider>
  )
}
```

Now, you can import and render <Providers /> directly within your root layout.
app/layout.js import { Providers } from './providers'

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

With the providers rendered at the root, all the components and hooks from these libraries will work as expected within your own Client Components.
Once a third-party library has added "use client" to its client code, you'll be able to remove the wrapper Client Component.
Sharing data between Server Components

Since Server Components are not interactive and therefore do not read from React state, you don't need React context to share data. Instead, you can use native JavaScript patterns for common data that multiple Server Components need to access. For example, a module can be used to share a database connection across multiple components:
utils/database.tsTypeScript export const db = new DatabaseConnection()

app/users/layout.tsxTypeScript import { db } from '@utils/database'

```
export async function UsersLayout() {
  let users = await db.query()
```

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

app/users/[id]/page.tsxTypeScript import { db } from '@utils/database'

```
export async function DashboardPage() {
  let user = await db.query()
  // ...
}
```

In the above example, both the layout and page need to make database queries. Each of these components shares access to the database by importing the @utils/database module. This JavaScript pattern is called global singletons.
Sharing fetch requests between Server Components

When fetching data, you may want to share the result of a fetch between a page or layout and some of its children components. This is an unnecessary coupling between the components and can lead to passing props back and forth between components. Instead, we recommend colocating data fetching alongside the component that consumes the data. fetch requests are automatically deduped in Server Components, so each route segment can request exactly the data it needs without worrying about duplicate requests. Next.js will read the same value from the fetch cache.
App RouterThe App Router is a new paradigm for building applications using React's latest features. If you're already familiar with Next.js, you'll find that the App Router is a natural evolution of the existing file-system based router in the Pages Router.
For new applications, we recommend using the App Router. For existing applications, you can incrementally migrate to the App Router.
This section of the documentation includes the features available in the App Router:
Building Your Application
Next.js provides the building blocks to create flexible, full-stack web applications. The guides in Building Your Application explain how to use these features and how to customize your application's behavior.
The sections and pages are organized sequentially, from basic to advanced, so you can follow them step-by-step when building your Next.js application. However, you can read them in any order or skip to the pages that apply to your use case.
If you're new to Next.js, we recommend starting with the Routing, Rendering, Data Fetching and Styling sections, as they introduce the fundamental Next.js and web concepts to help you get started. Then, you can dive deeper into the other sections such as Optimizing and Configuring. Finally, once you're ready, checkout the Deploying and Upgrading sections.
Routing FundamentalsThe skeleton of every application is routing. This page will introduce you to the fundamental concepts of routing for the web and how to handle

routing in Next.js.
Terminology

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

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

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

The app Router

In version 13, Next.js introduced a new App Router built on React Server Components, which supports shared layouts, nested routing, loading states, error handling, and more. The App Router works in a new directory named app. The app directory works alongside the pages directory to allow for incremental adoption. This allows you to opt some routes of your application into the new behavior while keeping other routes in the pages directory for previous behavior. If your application uses the pages directory, please also see the Pages Router documentation.

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

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

Recommendation: Check out the Server and Client Components page if you're new to

Server Components.

Roles of Folders and Files

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

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

Route Segments

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

Nested Routes

To create a nested route, you can nest folders inside each other. For example, you can add a new /dashboard/settings route by nesting two new folders in the app directory. The /dashboard/settings route is composed of three segments:

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

File Conventions

Next.js provides a set of special files to create UI with specific behavior in nested routes: layoutShared UI for a segment and its childrenpageUnique UI of a route and make

routes publicly accessibleloadingLoading UI for a segment and its childrennot-foundNot found UI for a segment and its childrenerrorError UI for a segment and its childrenglobal-errorGlobal Error UIrouteServer-side API endpointtemplateSpecialized re-rendered Layout UIdefaultFallback UI for Parallel Routes

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

Component Hierarchy

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

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

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

Colocation

In addition to special files, you have the option to colocate your own files (e.g. components, styles, tests, etc) inside folders in the app directory.
This is because while folders define routes, only the contents returned by page.js or route.js are publically addressable.

Learn more about Project Organization and Colocation.
Server-Centric Routing with Client-side Navigation

Unlike the pages directory which uses client-side routing, the App Router uses server-

centric routing to align with Server Components and data fetching on the server. With server-centric routing, the client does not have to download a route map and the same request for Server Components can be used to look up routes. This optimization is useful for all applications, but has a larger impact on applications with many routes.
Although routing is server-centric, the router uses client-side navigation with the Link Component - resembling the behavior of a Single-Page Application. This means when a user navigates to a new route, the browser will not reload the page. Instead, the URL will be updated and Next.js will only render the segments that change.
Additionally, as users navigate around the app, the router will store the result of the React Server Component payload in an in-memory client-side cache. The cache is split by route segments which allows invalidation at any level and ensures consistency across React's concurrent renders. This means that for certain cases, the cache of a previously fetched segment can be re-used, further improving performance.
Learn more about Linking and Navigating.
Partial Rendering

When navigating between sibling routes (e.g. /dashboard/settings and /dashboard/analytics below), Next.js will only fetch and render the layouts and pages in routes that change. It will not re-fetch or re-render anything above the segments in the subtree. This means that in routes that share a layout, the layout will be preserved when a user navigates between sibling pages.

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

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

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

These patterns allow you to build richer and more complex UIs, democratizing features

that were historically complex for small teams and individual developers to implement.
Next Steps

Now that you understand the fundamentals of routing in Next.js, follow the links below to create your first routes:
Defining Routes
We recommend reading the Routing Fundamentals page before continuing.

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

Next.js uses a file-system based router where folders are used to define routes. Each folder represents a route segment that maps to a URL segment. To create a nested route, you can nest folders inside each other.

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

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

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

Creating UI

Special file conventions are used to create UI for each route segment. The most common are pages to show UI unique to a route, and layouts to show UI that is shared across multiple routes.
For example, to create your first page, add a page.js file inside the app directory and export a React component:
app/page.tsxTypeScript export default function Page() {
  return <h1>Hello, Next.js!</h1>
}

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

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

A page is UI that is unique to a route. You can define pages by exporting a component from a page.js file. Use nested folders to define a route and a page.js file to make the route publicly accessible.
Create your first page by adding a page.js file inside the app directory:

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

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

Good to know:

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

Layouts

A layout is UI that is shared between multiple pages. On navigation, layouts preserve

state, remain interactive, and do not re-render. Layouts can also be nested.
You can define a layout by default exporting a React component from a layout.js file. The component should accept a children prop that will be populated with a child layout (if it exists) or a child page during rendering.

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

      {children}
    </section>
)
}

Good to know:

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

Root Layout (Required)

The root layout is defined at the top level of the app directory and applies to all routes. This layout enables you to modify the initial HTML returned from the server.
app/layout.tsxTypeScript export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
)
}

Good to know:

The app directory must include a root layout.
The root layout must define <html> and <body> tags since Next.js does not automatically create them.
You can use the built-in SEO support to manage <head> HTML elements, for example, the <title> element.
You can use route groups to create multiple root layouts. See an example here.
The root layout is a Server Component by default and can not be set to a Client Component.

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

Nesting Layouts

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

app/dashboard/layout.tsxTypeScript export default function DashboardLayout({

```
  children,
}: {
  children: React.ReactNode
}) {
  return <section>{children}</section>
}
```

Good to know:

Only the root layout can contain <html> and <body> tags.

If you were to combine the two layouts above, the root layout (app/layout.js) would wrap the dashboard layout (app/dashboard/layout.js), which would wrap route segments inside app/dashboard/*.
The two layouts would be nested as such:

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

Templates are similar to layouts in that they wrap each child layout or page. Unlike layouts that persist across routes and maintain state, templates create a new instance for each of their children on navigation. This means that when a user navigates between routes that share a template, a new instance of the component is mounted, DOM elements are recreated, state is not preserved, and effects are re-synchronized. There may be cases where you need those specific behaviors, and templates would be a more suitable option than layouts. For example:

Enter/exit animations using CSS or animation libraries.
Features that rely on useEffect (e.g logging page views) and useState (e.g a per-page feedback form).
To change the default framework behavior. For example, Suspense Boundaries inside layouts only show the fallback the first time the Layout is loaded and not when switching pages. For templates, the fallback is shown on each navigation.

Recommendation: We recommend using Layouts unless you have a specific reason to use Template.

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

app/template.tsxTypeScript export default function Template({ children }: { children: React.ReactNode }) {
  return {children}
}

The rendered output of a route segment with a layout and a template will be as such:
Output <Layout>
  {/* Note that the template is given a unique key. */}
  <Template key={routeParam}>{children}</Template>
</Layout>
Modifying <head>

In the app directory, you can modify the <head> HTML elements such as title and meta using the built-in SEO support.
Metadata can be defined by exporting a metadata object or generateMetadata function in a layout.js or page.js file.
app/page.tsxTypeScript import { Metadata } from 'next'

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

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

Good to know: You should not manually add <head> tags such as <title> and <meta> to root layouts. Instead, you should use the Metadata API which automatically handles advanced requirements such as streaming and de-duplicating <head> elements.

Learn more about available metadata options in the API reference.
Linking and NavigatingThe Next.js router uses server-centric routing with client-side navigation. It supports instant loading states and concurrent rendering. This means navigation maintains client-side state, avoids expensive re-renders, is interruptible, and doesn't cause race conditions.
There are two ways to navigate between routes:

<Link> Component
useRouter Hook

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

<Link> is a React component that extends the HTML <a> element to provide prefetching and client-side navigation between routes. It is the primary way to navigate between routes in Next.js.
To use <Link>, import it from next/link, and pass a href prop to the component:
app/page.tsxTypeScript import Link from 'next/link'

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

There are optional props you can pass to <Link>. See the API reference for more information.
Examples

Linking to Dynamic Segments

When linking to dynamic segments, you can use template literals and interpolation to generate a list of links. For example, to generate a list of blog posts:
app/blog/PostList.js import Link from 'next/link'

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

```
}
```

Checking Active Links

You can use usePathname() to determine if a link is active. For example, to add a class to the active link, you can check if the current pathname matches the href of the link:
app/ui/Navigation.js 'use client'

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

export function Navigation({ navLinks }) {
  const pathname = usePathname()

  return (
    <>
      {navLinks.map((link) => {
        const isActive = pathname.startsWith(link.href)

        return (
          <Link
            className={isActive ? 'text-blue' : 'text-black'}
            href={link.href}
            key={link.name}
          >
            {link.name}
          </Link>
        )
      })}
    </>
  )
}
```

Scrolling to an id

The default behavior of <Link> is to scroll to the top of the route segment that has changed. When there is an id defined in href, it will scroll to the specific id, similarly to a normal <a> tag.

useRouter() Hook

The useRouter hook allows you to programmatically change routes inside Client Components.
To use useRouter, import it from next/navigation, and call the hook inside your Client Component:
app/page.js 'use client'

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

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

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

The useRouter provides methods such as push(), refresh(), and more. See the API reference for more information.

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

How Navigation Works

A route transition is initiated using <Link> or calling router.push().
The router updates the URL in the browser's address bar.
The router avoids unnecessary work by re-using segments that haven't changed (e.g. shared layouts) from the client-side cache. This is also referred to as partial rendering.
If the conditions of soft navigation are met, the router fetches the new segment from the cache rather than the server. If not, the router performs a hard navigation and fetches the Server Component payload from the server.
If created, loading UI is shown from the server while the payload is being fetched.
The router uses the cached or fresh payload to render the new segments on the client.

Client-side Caching of Rendered Server Components

Good to know: This client-side cache is different from the server-side Next.js HTTP cache.

The new router has an in-memory client-side cache that stores the rendered result of Server Components (payload). The cache is split by route segments which allows invalidation at any level and ensures consistency across concurrent renders.
As users navigate around the app, the router will store the payload of previously fetched segments and prefetched segments in the cache.
This means, for certain cases, the router can re-use the cache instead of making a new request to the server. This improves performance by avoiding re-fetching data and re-rendering components unnecessarily.
Invalidating the Cache

Server Actions can be used to revalidate data on-demand by path (revalidatePath) or by cache tag (revalidateTag).
Prefetching

Prefetching is a way to preload a route in the background before it's visited. The rendered result of prefetched routes is added to the router's client-side cache. This makes navigating to a prefetched route near-instant.
By default, routes are prefetched as they become visible in the viewport when using the <Link> component. This can happen when the page first loads or through scrolling. Routes can also be programmatically prefetched using the prefetch method of the useRouter() hook.
Static and Dynamic Routes:

If the route is static, all the Server Component payloads for the route segments will be prefetched.
If the route is dynamic, the payload from the first shared layout down until the first loading.js file is prefetched. This reduces the cost of prefetching the whole route dynamically and allows instant loading states for dynamic routes.

Good to know:

Prefetching is only enabled in production.
Prefetching can be disabled by passing prefetch={false} to <Link>.

Soft Navigation

On navigation, the cache for changed segments is reused (if it exists), and no new requests are made to the server for data.
Conditions for Soft Navigation

On navigation, Next.js will use soft navigation if the route you are navigating to has been prefetched, and either doesn't include dynamic segments or has the same dynamic parameters as the current route.
For example, consider the following route that includes a dynamic [team] segment: /dashboard/[team]/*. The cached segments below /dashboard/[team]/* will only be invalidated when the [team] parameter changes.

Navigating from /dashboard/team-red/* to /dashboard/team-red/* will be a soft navigation.
Navigating from /dashboard/team-red/* to /dashboard/team-blue/* will be a hard navigation.

Hard Navigation

On navigation, the cache is invalidated and the server refetches data and re-renders the changed segments.
Back/Forward Navigation

Back and forward navigation (popstate event) has a soft navigation behavior. This means, the client-side cache is re-used and navigation is near-instant.
Focus and Scroll Management

By default, Next.js will set focus and scroll into view the segment that's changed on navigation.
Route GroupsIn the app directory, nested folders are normally mapped to URL paths. However, you can mark a folder as a Route Group to prevent the folder from being included in the route's URL path.
This allows you to organize your route segments and project files into logical groups without affecting the URL path structure.
Route groups are useful for:

Organizing routes into groups e.g. by site section, intent, or team.
Enabling nested layouts in the same route segment level:

Creating multiple nested layouts in the same segment, including multiple root layouts
Adding a layout to a subset of routes in a common segment

Convention

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

Organize routes without affecting the URL path

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

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

Opting specific segments into a layout

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

Creating multiple root layouts

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

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

Good to know:

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

Dynamic RoutesWhen you don't know the exact segment names ahead of time and want to create routes from dynamic data, you can use Dynamic Segments that are filled in at request time or prerendered at build time.
Convention

A Dynamic Segment can be created by wrapping a folder's name in square brackets: [folderName]. For example, [id] or [slug].
Dynamic Segments are passed as the params prop to layout, page, route, and generateMetadata functions.
Example

For example, a blog could include the following route app/blog/[slug]/page.js where [slug] is the Dynamic Segment for blog posts.
app/blog/[slug]/page.tsxTypeScript export default function Page({ params }: { params: { slug: string } }) {
  return My Post: {params.slug}
}

RouteExample URLparamsapp/blog/[slug]/page.js/blog/a{ slug: 'a' }app/blog/[slug]/page.js/blog/b{ slug: 'b' }app/blog/[slug]/page.js/blog/c{ slug: 'c' }
See the generateStaticParams() page to learn how to generate the params for the segment.

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

Generating Static Params

The generateStaticParams function can be used in combination with dynamic route segments to statically generate routes at build time instead of on-demand at request time.
app/blog/[slug]/page.tsxTypeScript export async function generateStaticParams() {
  const posts = await fetch('https://.../posts').then((res) => res.json())

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

The primary benefit of the generateStaticParams function is its smart retrieval of data. If content is fetched within the generateStaticParams function using a fetch request, the requests are automatically deduplicated. This means a fetch request with the same arguments across multiple generateStaticParams, Layouts, and Pages will only be made once, which decreases build times.
Use the migration guide if you are migrating from the pages directory.
See generateStaticParams server function documentation for more information and advanced use cases.
Catch-all Segments

Dynamic Segments can be extended to catch-all subsequent segments by adding an ellipsis inside the brackets [...folderName].
For example, app/shop/[...slug]/page.js will match /shop/clothes, but also /shop/clothes/tops, /shop/clothes/tops/t-shirts, and so on.
RouteExample URLparamsapp/shop/[...slug]/page.js/shop/a{ slug: ['a'] }app/shop/[...slug]/page.js/shop/a/b{ slug: ['a', 'b'] }app/shop/[...slug]/page.js/shop/a/b/c{ slug: ['a', 'b', 'c'] }
Optional Catch-all Segments

Catch-all Segments can be made optional by including the parameter in double square brackets: [[...folderName]].
For example, app/shop/[[...slug]]/page.js will also match /shop, in addition to /shop/clothes, /shop/clothes/tops, /shop/clothes/tops/t-shirts.
The difference between catch-all and optional catch-all segments is that with optional, the route without the parameter is also matched (/shop in the example above).
RouteExample URLparamsapp/shop/[[...slug]]/page.js/shop{}app/shop/[[...slug]]/page.js/shop/a{ slug: ['a'] }app/shop/[[...slug]]/page.js/shop/a/b{ slug: ['a', 'b'] }app/shop/[[...slug]]/page.js/shop/a/b/c{ slug: ['a', 'b', 'c'] }
TypeScript

When using TypeScript, you can add types for params depending on your configured route segment.
app/blog/[slug]/page.tsxTypeScript export default function Page({ params }: { params:

{ slug: string } }) {
  return <h1>My Page</h1>
}

Routeparams Type Definitionapp/blog/[slug]/page.js{ slug: string }app/shop/[...slug]/page.js{ slug: string[] }app/[categoryId]/[itemId]/page.js{ categoryId: string, itemId: string }

Good to know: This may be done automatically by the TypeScript plugin in the future.
Loading UI and StreamingThe special file loading.js helps you create meaningful Loading UI with React Suspense. With this convention, you can show an instant loading state from the server while the content of a route segment loads. The new content is automatically swapped in once rendering is complete.

Instant Loading States

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

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

In the same folder, loading.js will be nested inside layout.js. It will automatically wrap the page.js file and any children below in a <Suspense> boundary.

Good to know:

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

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

Streaming with Suspense

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

What is Streaming?

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

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

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

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

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

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

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

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

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

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

<Suspense> works by wrapping a component that performs an asynchronous action (e.g. fetch data), showing fallback UI (e.g. skeleton, spinner) while it's happening, and then swapping in your component once the action completes.
app/dashboard/page.tsxTypeScript import { Suspense } from 'react'
import { PostFeed, Weather } from './Components'

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

By using Suspense, you get the benefits of:

Streaming Server Rendering - Progressively rendering HTML from the server to the client.
Selective Hydration - React prioritizes what components to make interactive first based on user interaction.

For more Suspense examples and use cases, please see the React Documentation.
SEO

Next.js will wait for data fetching inside generateMetadata to complete before streaming

UI to the client. This guarantees the first part of a streamed response includes <head> tags.
Since streaming is server-rendered, it does not impact SEO. You can use the Mobile Friendly Test tool from Google to see how your page appears to Google's web crawlers and view the serialized HTML (source).
Error HandlingThe error.js file convention allows you to gracefully handle unexpected runtime errors in nested routes.

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

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

app/dashboard/error.tsxTypeScript 'use client' // Error components must be Client Components

```
import { useEffect } from 'react'

export default function Error({
  error,
  reset,
}: {
  error: Error
  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>
```

```
  )
}
```

How error.js Works

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

Recovering From Errors

The cause of an error can sometimes be temporary. In these cases, simply trying again might resolve the issue.
An error component can use the reset() function to prompt the user to attempt to recover from the error. When executed, the function will try to re-render the Error boundary's contents. If successful, the fallback error component is replaced with the result of the re-render.
app/dashboard/error.tsxTypeScript 'use client'

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

```
  )
}
```

Nested Routes

React components created through special files are rendered in a specific nested hierarchy.
For example, a nested route with two segments that both include layout.js and error.js files are rendered in the following simplified component hierarchy:

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

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

Handling Errors in Layouts

error.js boundaries do not catch errors thrown in layout.js or template.js components of the same segment. This intentional hierarchy keeps important UI that is shared between sibling routes (such as navigation) visible and functional when an error occurs.
To handle errors within a specific layout or template, place an error.js file in the layouts parent segment.
To handle errors within the root layout or template, use a variation of error.js called global-error.js.
Handling Errors in Root Layouts

The root app/error.js boundary does not catch errors thrown in the root app/layout.js or app/template.js component.
To specifically handle errors in these root components, use a variation of error.js called app/global-error.js located in the root app directory.

Unlike the root error.js, the global-error.js error boundary wraps the entire application, and its fallback component replaces the root layout when active. Because of this, it is important to note that global-error.js must define its own <html> and <body> tags. global-error.js is the least granular error UI and can be considered "catch-all" error handling for the whole application. It is unlikely to be triggered often as root components are typically less dynamic, and other error.js boundaries will catch most errors.
Even if a global-error.js is defined, it is still recommended to define a root error.js whose fallback component will be rendered within the root layout, which includes globally shared UI and branding.
app/global-error.tsxTypeScript 'use client'

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

Handling Server Errors

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

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

This is a security precaution to avoid leaking potentially sensitive details included in the error to the client.
The message property contains a generic message about the error and the digest property contains an automatically generated hash of the error that can be used to match the corresponding error in server-side logs.
During development, the Error object forwarded to the client will be serialized and include the message of the original error for easier debugging.
Parallel RoutesParallel Routing allows you to simultaneously or conditionally render one or more pages in the same layout. For highly dynamic sections of an app, such as dashboards and feeds on social sites, Parallel Routing can be used to implement complex routing patterns.
For example, you can simultaneously render the team and analytics pages.

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

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

Convention

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

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

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

The folder structure above means that the component in app/layout.js now accepts the @analytics and @team slots props, and can render them in parallel alongside the children prop:
app/layout.tsxTypeScript export default function Layout(props: {
  children: React.ReactNode
  analytics: React.ReactNode
  team: React.ReactNode
}) {
  return (
    <>
      {props.children}

```
      {props.team}
      {props.analytics}
    </>
  )
}
```

Good to know: The children prop is an implicit slot that does not need to be mapped to a folder. This means app/page.js is equivalent to app/@children/page.js.

Unmatched Routes

By default, the content rendered within a slot will match the current URL.
In the case of an unmatched slot, the content that Next.js renders differs based on the routing technique and folder structure.
default.js

You can define a default.js file to render as a fallback when Next.js cannot recover a slot's active state based on the current URL.
Consider the following folder structure. The @team slot has a settings directory, but @analytics does not.

If you were to navigate from the root / to /settings, the content that gets rendered is different based on the type of navigation and the availability of the default.js file.
With @analytics/default.jsWithout @analytics/default.jsSoft Navigation@team/settings/page.js and @analytics/page.js@team/settings/page.js and @analytics/page.jsHard Navigation@team/settings/page.js and @analytics/default.js404
Soft Navigation

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

On a hard navigation - a navigation that requires a full page reload - Next.js will first try to render the unmatched slot's default.js file. If that's not available, a 404 gets rendered.

The 404 for unmatched routes helps ensure that you don't accidentally render a route that shouldn't be parallel rendered.

useSelectedLayoutSegment(s)

Both useSelectedLayoutSegment and useSelectedLayoutSegments accept a parallelRoutesKey, which allows you read the active route segment within that slot.
app/layout.tsxTypeScript 'use client'

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

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

When a user navigates to @authModal/login, or /login in the URL bar, loginSegments will be equal to the string "login".
Examples

Modals

Parallel Routing can be used to render modals.

The @authModal slot renders a<Modal> component that can be shown by navigating

to a matching route, for example /login.
app/layout.tsxTypeScript export default async function Layout(props: {
  // ...
  authModal: React.ReactNode
}) {
  return (
    <>
      {/* ... */}
      {props.authModal}
    </>
)
}

app/@authModal/login/page.tsxTypeScript import { Modal } from 'components/modal'

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

To ensure that the contents of the modal don't get rendered when it's not active, you can create a default.js file that returns null.
app/@authModal/default.tsxTypeScript export default function Default() {
  return null
}

Dismissing a modal

If a modal was initiated through client navigation, e.g. by using <Link href="/login">, you can dismiss the modal by calling router.back() or by using a Link component.
app/@authModal/login/page.tsxTypeScript 'use client'
import { useRouter } from 'next/navigation'
import { Modal } from 'components/modal'

export default async function Login() {
  const router = useRouter()
  return (
    <Modal>

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

More information on modals is covered in the Intercepting Routes section.

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

app/@authModal/[...catchAll]/page.tsxTypeScript export default function CatchAll() {
  return null
}

Catch-all routes take precedence over default.js.

Conditional Routes

Parallel Routes can be used to implement conditional routing. For example, you can render a @dashboard or @login route depending on the authentication state.
app/layout.tsxTypeScript import { getUser } from '@/lib/auth'

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

Intercepting RoutesIntercepting routes allows you to load a route within the current layout while keeping the context for the current page. This routing paradigm can be useful when you want to "intercept" a certain route to show a different route.
For example, when clicking on a photo from within a feed, a modal overlaying the feed should show up with the photo. In this case, Next.js intercepts the /feed route and

"masks" this URL to show /photo/123 instead.

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

Convention

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

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

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

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

Examples

Modals

Intercepting Routes can be used together with Parallel Routes to create modals.
Using this pattern to create modals overcomes some common challenges when working with modals, by allowing you to:

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

Reopen the modal on forwards navigation

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

Other examples could include opening a login modal in a top navbar while also having a dedicated /login page, or opening a shopping cart in a side modal.
View an example of modals with Intercepted and Parallel Routes.
Route HandlersRoute Handlers allow you to create custom request handlers for a given route using the Web Request and Response APIs.

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

Convention

Route Handlers are defined in a route.js|ts file inside the app directory:
app/api/route.tsTypeScript export async function GET(request: Request) {}

Route Handlers can be nested inside the app directory, similar to page.js and layout.js. But there cannot be a route.js file at the same route segment level as page.js.
Supported HTTP Methods

The following HTTP methods are supported: GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS. If an unsupported method is called, Next.js will return a 405 Method Not Allowed response.
Extended NextRequest and NextResponse APIs

In addition to supporting native Request and Response. Next.js extends them with

NextRequest and NextResponse to provide convenient helpers for advanced use cases.
Behavior

Static Route Handlers

Route Handlers are statically evaluated by default when using the GET method with the Response object.
app/items/route.tsTypeScript import { NextResponse } from 'next/server'

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

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

TypeScript Warning: Although Response.json() is valid, native TypeScript types currently shows an error, you can use NextResponse.json() for typed responses instead.

Dynamic Route Handlers

Route handlers are evaluated dynamically when:

Using the Request object with the GET method.
Using any of the other HTTP methods.
Using Dynamic Functions like cookies and headers.
The Segment Config Options manually specifies dynamic mode.

For example:
app/products/api/route.tsTypeScript import { NextResponse } from 'next/server'

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

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

Similarly, the POST method will cause the Route Handler to be evaluated dynamically.
app/items/route.tsTypeScript import { NextResponse } from 'next/server'

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

  const data = await res.json()

  return NextResponse.json(data)
}
```

Good to know: Previously, API Routes could have been used for use cases like handling form submissions. Route Handlers are likely not the solution for these use cases. We will be recommending the use of mutations for this when ready.

Route Resolution

You can consider a route the lowest level routing primitive.

They do not participate in layouts or client-side navigations like page.
There cannot be a route.js file at the same route as page.js.

PageRouteResultapp/page.jsapp/route.js Conflictapp/page.jsapp/api/route.js Validapp/[user]/page.jsapp/api/route.js Valid
Each route.js or page.js file takes over all HTTP verbs for that route.
app/page.js export default function Page() {
  return <h1>Hello, Next.js!</h1>
}

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

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

You can revalidate static data fetches using the next.revalidate option:
app/items/route.tsTypeScript import { NextResponse } from 'next/server'

export async function GET() {
  const res = await fetch('https://data.mongodb-api.com/...', {
    next: { revalidate: 60 }, // Revalidate every 60 seconds
  })
  const data = await res.json()

  return NextResponse.json(data)
}

Alternatively, you can use the revalidate segment config option:
 export const revalidate = 60
Dynamic Functions

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

You can read cookies with cookies from next/headers. This server function can be called directly in a Route Handler, or nested inside of another function.
This cookies instance is read-only. To set cookies, you need to return a new Response using the Set-Cookie header.
app/api/route.tsTypeScript import { cookies } from 'next/headers'

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

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

Alternatively, you can use abstractions on top of the underlying Web APIs to read cookies (NextRequest):
app/api/route.tsTypeScript import { type NextRequest } from 'next/server'

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

Headers

You can read headers with headers from next/headers. This server function can be called directly in a Route Handler, or nested inside of another function.
This headers instance is read-only. To set headers, you need to return a new Response with new headers.
app/api/route.tsTypeScript import { headers } from 'next/headers'

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

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

Alternatively, you can use abstractions on top of the underlying Web APIs to read headers (NextRequest):

```
app/api/route.tsTypeScript import { type NextRequest } from 'next/server'

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

Redirects

```
app/api/route.tsTypeScript import { redirect } from 'next/navigation'

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

Dynamic Route Segments

We recommend reading the Defining Routes page before continuing.

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

```
app/items/[slug]/route.tsTypeScript export async function GET(
  request: Request,
  { params }: { params: { slug: string } }
) {
```

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

RouteExample URLparamsapp/items/[slug]/route.js/items/a{ slug: 'a' }app/items/[slug]/route.js/items/b{ slug: 'b' }app/items/[slug]/route.js/items/c{ slug: 'c' }
Streaming

Streaming is commonly used in combination with Large Language Models (LLMs), such an OpenAI, for AI-generated content. Learn more about the AI SDK.
app/api/completion/route.tsTypeScript import { Configuration, OpenAIApi } from 'openai-edge'
import { OpenAIStream, StreamingTextResponse } from 'ai'

```
const config = new Configuration({
  apiKey: process.env.OPENAI_API_KEY,
})
const openai = new OpenAIApi(config)

export const runtime = 'edge'

export async function POST(req: Request) {
  const { prompt } = await req.json()
  const response = await openai.createCompletion({
    model: 'text-davinci-003',
    stream: true,
    temperature: 0.6,
    prompt: 'What is Next.js?',
  })

  const stream = OpenAIStream(response)
  return new StreamingTextResponse(stream)
}
```

These abstractions use the Web APIs to create a stream. You can also use the underlying Web APIs directly.
app/api/route.tsTypeScript // https://developer.mozilla.org/en-US/docs/Web/API/ReadableStream#convert_async_iterator_to_stream
function iteratorToStream(iterator: any) {
  return new ReadableStream({
    async pull(controller) {
      const { value, done } = await iterator.next()

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

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

const encoder = new TextEncoder()

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

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

  return new Response(stream)
}
```

Request Body

You can read the Request body using the standard Web API methods:
app/items/route.tsTypeScript import { NextResponse } from 'next/server'

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

Request Body FormData

You can read the FormData using the the request.formData() function:
app/items/route.tsTypeScript import { NextResponse } from 'next/server'

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

CORS

You can set CORS headers on a Response using the standard Web API methods:
app/api/route.tsTypeScript export async function GET(request: Request) {

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

Edge and Node.js Runtimes

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

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

Non-UI Responses

You can use Route Handlers to return non-UI content. Note that sitemap.xml, robots.txt, app icons, and open graph images all have built-in support.
app/rss.xml/route.tsTypeScript export async function GET() {
  return new Response(`<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">

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

</rss>`)
}

Segment Config Options

Route Handlers use the same route segment configuration as pages and layouts.
app/items/route.tsTypeScript export const dynamic = 'auto'
export const dynamicParams = true
export const revalidate = false
export const fetchCache = 'auto'
export const runtime = 'nodejs'
export const preferredRegion = 'auto'

See the API reference for more details.
Middleware
Middleware allows you to run code before a request is completed. Then, based on the incoming request, you can modify the response by rewriting, redirecting, modifying the request or response headers, or responding directly.
Middleware runs before cached content and routes are matched. See Matching Paths for more details.
Convention

Use the file middleware.ts (or .js) in the root of your project to define Middleware. For example, at the same level as pages or app, or inside src if applicable.
Example

middleware.ts import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

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

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

Middleware will be invoked for every route in your project. The following is the execution order:

headers from next.config.js
redirects from next.config.js
Middleware (rewrites, redirects, etc.)
beforeFiles (rewrites) from next.config.js
Filesystem routes (public/, _next/static/, pages/, app/, etc.)
afterFiles (rewrites) from next.config.js
Dynamic Routes (/blog/[slug])
fallback (rewrites) from next.config.js

There are two ways to define which paths Middleware will run on:

Custom matcher config
Conditional statements

Matcher

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

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

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

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

The matcher config allows full regex so matching like negative lookaheads or character matching is supported. An example of a negative lookahead to match all except specific paths can be seen here:

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

Good to know: The matcher values need to be constants so they can be statically analyzed at build-time. Dynamic values such as variables will be ignored.

Configured matchers:

MUST start with /
Can include named parameters: /about/:path matches /about/a and /about/b but not /about/a/c
Can have modifiers on named parameters (starting with :): /about/:path* matches /about/a/b/c because * is zero or more. ? is zero or one and + one or more
Can use regular expression enclosed in parenthesis: /about/(.*) is the same as /about/:path*

Read more details on path-to-regexp documentation.

Good to know: For backward compatibility, Next.js always considers /public as /public/index. Therefore, a matcher of /public/:path will match.

Conditional Statements

middleware.ts import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

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

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

NextResponse

The NextResponse API allows you to:

redirect the incoming request to a different URL
rewrite the response by displaying a given URL
Set request headers for API Routes, getServerSideProps, and rewrite destinations
Set response cookies
Set response headers

To produce a response from Middleware, you can:
rewrite to a route (Page or Route Handler) that produces a response
return a NextResponse directly. See Producing a Response

Using Cookies

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

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

middleware.ts import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

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

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

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

  return response
}
```

Setting Headers

You can set request and response headers using the NextResponse API (setting request headers is available since Next.js v13.0.0).
middleware.tsTypeScript import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

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

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

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

Good to know: Avoid setting large headers as it might cause 431 Request Header Fields Too Large error depending on your backend web server configuration.

Producing a Response

You can respond from Middleware directly by returning a Response or NextResponse instance. (This is available since Next.js v13.1.0)
middleware.tsTypeScript import { NextRequest, NextResponse } from 'next/server'
import { isAuthenticated } from '@lib/auth'

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

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

```
  }
}
```

Advanced Middleware Flags

In v13.1 of Next.js two additional flags were introduced for middleware, skipMiddlewareUrlNormalize and skipTrailingSlashRedirect to handle advanced use cases.
skipTrailingSlashRedirect allows disabling Next.js default redirects for adding or removing trailing slashes allowing custom handling inside middleware which can allow maintaining the trailing slash for some paths but not others allowing easier incremental migrations.
next.config.js

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

middleware.js

```
const legacyPrefixes = ['/docs', '/blog']

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

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

  // apply trailing slash handling
  if (
    !pathname.endsWith('/') &&
    !pathname.match(/((?!\.well-known(?:\/.*)?)(?:[^/]+\/)*[^/]+\.\w+)/)
  ) {
    req.nextUrl.pathname += '/'
    return NextResponse.redirect(req.nextUrl)
  }
}
```

skipMiddlewareUrlNormalize allows disabling the URL normalizing Next.js does to make handling direct visits and client-transitions the same. There are some advanced cases where you need full control using the original URL which this unlocks.
next.config.js

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

middleware.js

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

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

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

Version History

Version	Changes
v13.1.0	Advanced Middleware flags added
v13.0.0	Middleware can modify request headers, response headers, and send responses
v12.2.0	Middleware is stable, please see the upgrade guide
v12.0.9	Enforce absolute URLs in Edge Runtime (PR)
v12.0.0	Middleware (Beta) added

Project Organization and File Colocation

Apart from routing folder and file conventions, Next.js is unopinionated about how you organize and colocate your project files.
This page shares default behavior and features you can use to organize your project.

Safe colocation by default
Project organization features
Project organization strategies

Safe colocation by default

In the app directory, nested folder hierarchy defines route structure.
Each folder represents a route segment that is mapped to a corresponding segment in a URL path.
However, even though route structure is defined through folders, a route is not publically accessible until a page.js or route.js file is added to a route segment.

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

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

Good to know:

This is different from the pages directory, where any file in pages is considered a route.

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

Project organization features

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

Private Folders

Private folders can be created by prefixing a folder with an underscore: _folderName
This indicates the folder is a private implementation detail and should not be considered by the routing system, thereby opting the folder and all its subfolders out of routing.

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

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

Good to know

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

Route Groups

Route groups can be created by wrapping a folder in parenthesis: (folderName)
This indicates the folder is for organizational purposes and should not be included in the route's URL path.

Route groups are useful for:

Organizing routes into groups e.g. by site section, intent, or team.
Enabling nested layouts in the same route segment level:

Creating multiple nested layouts in the same segment, including multiple root layouts
Adding a layout to a subset of routes in a common segment

src Directory

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

Module Path Aliases

Next.js supports Module Path Aliases which make it easier to read and maintain imports across deeply nested project files.
app/dashboard/settings/analytics/page.js // before
import { Button } from '../../../components/button'

// after
import { Button } from '@/components/button'
Project organization strategies

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

The following section lists a very high-level overview of common strategies. The simplest takeaway is to choose a strategy that works for you and your team and be consistent across the project.

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

Store project files outside of app

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

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.
InternationalizationNext.js enables you to configure the routing and rendering of content to support multiple languages. Making your site adaptive to different locales includes translated content (localization) and internationalized routes.
Terminology

Locale: An identifier for a set of language and formatting preferences. This usually includes the preferred language of the user and possibly their geographic region.

en-US: English as spoken in the United States
nl-NL: Dutch as spoken in the Netherlands
nl: Dutch, no specific region

Routing Overview

It’s recommended to use the user’s language preferences in the browser to select which locale to use. Changing your preferred language will modify the incoming Accept-Language header to your application.
For example, using the following libraries, you can look at an incoming Request to determine which locale to select, based on the Headers, locales you plan to support, and the default locale.
middleware.js import { match } from '@formatjs/intl-localematcher'
import Negotiator from 'negotiator'

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

match(languages, locales, defaultLocale) // -> 'en-US'
```

Routing can be internationalized by either the sub-path (/fr/products) or domain (my-site.fr/products). With this information, you can now redirect the user based on the locale inside Middleware.
middleware.js import { NextResponse } from 'next/server'

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

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

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

  // Redirect if there is no locale
  if (pathnameIsMissingLocale) {
```

```
    const locale = getLocale(request)

    // e.g. incoming request is /products
    // The new URL is now /en-US/products
    return NextResponse.redirect(
      new URL(`/${locale}/${pathname}`, request.url)
    )
  }
}

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

Finally, ensure all special files inside app/ are nested under app/[lang]. This enables the Next.js router to dynamically handle different locales in the route, and forward the lang parameter to every layout and page. For example:

```
app/[lang]/page.js // You now have access to the current locale
// e.g. /en-US/products -> `lang` is "en-US"
export default async function Page({ params: { lang } }) {
  return ...
}
```

The root layout can also be nested in the new folder (e.g. app/[lang]/layout.js).

Localization

Changing displayed content based on the user’s preferred locale, or localization, is not something specific to Next.js. The patterns described below would work the same with any web application.

Let’s assume we want to support both English and Dutch content inside our application. We might maintain two different “dictionaries”, which are objects that give us a mapping from some key to a localized string. For example:

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

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

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

app/[lang]/dictionaries.js

```
import 'server-only'

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

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

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

app/[lang]/page.js

```
import { getDictionary } from './dictionaries'

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

Because all layouts and pages in the app/ directory default to Server Components, we do not need to worry about the size of the translation files affecting our client-side JavaScript bundle size. This code will only run on the server, and only the resulting HTML will be sent to the browser.

Static Generation

To generate static routes for a given set of locales, we can use generateStaticParams with any page or layout. This can be global, for example, in the root layout:

app/[lang]/layout.js

```
export async function generateStaticParams() {
  return [{ lang: 'en-US' }, { lang: 'de' }]
}

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

Examples

Minimal i18n routing and translations
next-intl
RenderingRendering converts the code you write into user interfaces.
React 18 and Next.js 13 introduced new ways to render your application. This page will help you understand the differences between rendering environments, strategies, runtimes, and how to opt into them.
Rendering Environments

There are two environments where your application code can be rendered: the client and the server.

The client refers to the browser on a user's device that sends a request to a server for your application code. It then turns the response from the server into an interface the user can interact with.
The server refers to the computer in a data center that stores your application code, receives requests from a client, does some computation, and sends back an appropriate response.

Good to know: Server can refer to computers in regions where your application is deployed to, the Edge Network where your application code is distributed, or Content Delivery Networks (CDNs) where the result of the rendering work can be cached.

Component-level Client and Server Rendering

Before React 18, the primary way to render your application using React was entirely on the client.
Next.js provided an easier way to break down your application into pages and prerender on the server by generating HTML and sending it to the client to be hydrated by React. However, this led to additional JavaScript needed on the client to make the initial HTML interactive.
Now, with Server and Client Components, React can render on the client and the

server meaning you can choose the rendering environment at the component level. By default, the app router uses Server Components, allowing you to easily render components on the server and reducing the amount of JavaScript sent to the client.
Static and Dynamic Rendering on the Server

In addition to client-side and server-side rendering with React components, Next.js gives you the option to optimize rendering on the server with Static and Dynamic Rendering.
Static Rendering

With Static Rendering, both Server and Client Components can be prerendered on the server at build time. The result of the work is cached and reused on subsequent requests. The cached result can also be revalidated.

Good to know: This is equivalent to Static Site Generation (SSG) and Incremental Static Regeneration (ISR) in the Pages Router.

Server and Client Components are rendered differently during Static Rendering:

Client Components have their HTML and JSON prerendered and cached on the server. The cached result is then sent to the client for hydration.
Server Components are rendered on the server by React, and their payload is used to generate HTML. The same rendered payload is also used to hydrate the components on the client, resulting in no JavaScript needed on the client.

Dynamic Rendering

With Dynamic Rendering, both Server and Client Components are rendered on the server at request time. The result of the work is not cached.

Good to know: This is equivalent to Server-Side Rendering (getServerSideProps()) in the Pages Router.

To learn more about static and dynamic behavior, see the Static and Dynamic

Rendering page. To learn more about caching, see the Caching sections.
Edge and Node.js Runtimes

On the server, there are two runtimes where your pages can be rendered:

The Node.js Runtime (default) has access to all Node.js APIs and compatible packages from the ecosystem.
The Edge Runtime is based on Web APIs.

Both runtimes support streaming from the server, depending on your deployment infrastructure.
To learn how to switch between runtimes, see the Edge and Node.js Runtimes page.
Next Steps

Now that you understand the fundamentals of rendering, you can learn more about implementing the different rendering strategies and runtimes:
Static and Dynamic RenderingIn Next.js, a route can be statically or dynamically rendered.

In a static route, components are rendered on the server at build time. The result of the work is cached and reused on subsequent requests.
In a dynamic route, components are rendered on the server at request time.

Static Rendering (Default)

By default, Next.js statically renders routes to improve performance. This means all the rendering work is done ahead of time and can be served from a Content Delivery Network (CDN) geographically closer to the user.
Static Data Fetching (Default)

By default, Next.js will cache the result of fetch() requests that do not specifically opt out of caching behavior. This means that fetch requests that do not set a cache option will use the force-cache option.
If any fetch requests in the route use the revalidate option, the route will be re-rendered statically during revalidation.
To learn more about caching data fetching requests, see the Caching and Revalidating page.
Dynamic Rendering

During static rendering, if a dynamic function or a dynamic fetch() request (no caching) is discovered, Next.js will switch to dynamically rendering the whole route at request time. Any cached data requests can still be re-used during dynamic rendering.
This table summarizes how dynamic functions and caching affect the rendering behavior of a route:
Data FetchingDynamic FunctionsRenderingStatic (Cached)NoStaticStatic (Cached)YesDynamicNot CachedNoDynamicNot CachedYesDynamic
Note how dynamic functions always opt the route into dynamic rendering, regardless of whether the data fetching is cached or not. In other words, static rendering is dependent not only on the data fetching behavior, but also on the dynamic functions used in the route.

Good to know: In the future, Next.js will introduce hybrid server-side rendering where layouts and pages in a route can be independently statically or dynamically rendered, instead of the whole route.

Dynamic Functions

Dynamic functions rely on information that can only be known at request time such as a user's cookies, current requests headers, or the URL's search params. In Next.js, these dynamic functions are:

Using cookies() or headers() in a Server Component will opt the whole route into dynamic rendering at request time.
Using useSearchParams() in Client Components will skip static rendering and instead render all Client Components up to the nearest parent Suspense boundary on the client.

We recommend wrapping the Client Component that uses useSearchParams() in a <Suspense/> boundary. This will allow any Client Components above it to be statically

rendered. Example.

Using the searchParams Pages prop will opt the page into dynamic rendering at request time.

Dynamic Data Fetching

Dynamic data fetches are fetch() requests that specifically opt out of caching behavior by setting the cache option to 'no-store' or revalidate to 0.
The caching options for all fetch requests in a layout or page can also be set using the segment config object.
To learn more about Dynamic Data Fetching, see the Data Fetching page.
Edge and Node.js Runtimes
In the context of Next.js, "runtime" refers to the set of libraries, APIs, and general functionality available to your code during execution.
Next.js has two server runtimes where you can render parts of your application code:

Node.js Runtime
Edge Runtime

Each runtime has its own set of APIs. Please refer to the Node.js Docs and Edge Docs for the full list of available APIs. Both runtimes can also support streaming depending on your deployment infrastructure.
By default, the app directory uses the Node.js runtime. However, you can opt into different runtimes (e.g. Edge) on a per-route basis.
Runtime Differences

There are many considerations to make when choosing a runtime. This table shows the major differences at a glance. If you want a more in-depth analysis of the differences, check out the sections below.
NodeServerlessEdgeCold Boot/~250msInstantHTTP StreamingYesYesYesIOAllAllfetchScalability/ HighHighestSecurityNormalHighHighLatencyNormalLowLowestnpm PackagesAllAllA smaller subset
Edge Runtime

In Next.js, the lightweight Edge Runtime is a subset of available Node.js APIs.
The Edge Runtime is ideal if you need to deliver dynamic, personalized content at low latency with small, simple functions. The Edge Runtime's speed comes from its minimal use of resources, but that can be limiting in many scenarios.
For example, code executed in the Edge Runtime on Vercel cannot exceed between 1 MB and 4 MB, this limit includes imported packages, fonts and files, and will vary depending on your deployment infrastructure.
Node.js Runtime

Using the Node.js runtime gives you access to all Node.js APIs, and all npm packages that rely on them. However, it's not as fast to start up as routes using the Edge runtime. Deploying your Next.js application to a Node.js server will require managing, scaling, and configuring your infrastructure. Alternatively, you can consider deploying your Next.js application to a serverless platform like Vercel, which will handle this for you.
Serverless Node.js

Serverless is ideal if you need a scalable solution that can handle more complex computational loads than the Edge Runtime. With Serverless Functions on Vercel, for example, your overall code size is 50MB including imported packages, fonts, and files. The downside compared to routes using the Edge is that it can take hundreds of milliseconds for Serverless Functions to boot up before they begin processing requests. Depending on the amount of traffic your site receives, this could be a frequent occurrence as the functions are not frequently "warm".
Examples

Segment Runtime Option

You can specify a runtime for individual route segments in your Next.js application. To do so, declare a variable called runtime and export it. The variable must be a string, and

must have a value of either 'nodejs' or 'edge' runtime.The following example demonstrates a page route segment that exports a runtime with a value of 'edge':app/page.tsxTypeScript export const runtime = 'edge' // 'nodejs' (default) | 'edge'If the segment runtime is not set, the default nodejs runtime will be used. You do not need to use the runtime option if you do not plan to change from the Node.js runtime.
Data FetchingThe Next.js App Router introduces a new, simplified data fetching system built on React and the Web platform. This page will go through the fundamental concepts and patterns to help you manage your data's lifecycle.
Here's a quick overview of the recommendations on this page:

Fetch data on the server using Server Components.
Fetch data in parallel to minimize waterfalls and reduce loading times.
For Layouts and Pages, fetch data where it's used. Next.js will automatically dedupe requests in a tree.
Use Loading UI, Streaming and Suspense to progressively render a page and show a result to the user while the rest of the content loads.

The fetch() API

The new data fetching system is built on top of the native fetch() Web API and makes use of async and await in Server Components.

React extends fetch to provide automatic request deduping.
Next.js extends the fetch options object to allow each request to set its own caching and revalidating rules.

Learn how to use fetch in Next.js.
Fetching Data on the Server

Whenever possible, we recommend fetching data in Server Components. Server Components always fetch data on the server. This allows you to:

Have direct access to backend data resources (e.g. databases).
Keep your application more secure by preventing sensitive information, such as access tokens and API keys, from being exposed to the client.
Fetch data and render in the same environment. This reduces both the back-and-forth communication between client and server, as well as the work on the main thread on the client.

Perform multiple data fetches with single round-trip instead of multiple individual requests on the client.
Reduce client-server waterfalls.
Depending on your region, data fetching can also happen closer to your data source, reducing latency and improving performance.

Good to know: It's still possible to fetch data client-side. We recommend using a third-party library such as SWR or React Query with Client Components. In the future, it'll also be possible to fetch data in Client Components using React's use() hook.

Fetching Data at the Component Level

In the App Router, you can fetch data inside layouts, pages, and components. Data fetching is also compatible with Streaming and Suspense.

Good to know: For layouts, it's not possible to pass data between a parent layout and its children components. We recommend fetching data directly inside the layout that needs it, even if you're requesting the same data multiple times in a route. Behind the scenes, React and Next.js will cache and dedupe requests to avoid the same data being fetched more than once.

Parallel and Sequential Data Fetching

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

With parallel data fetching, requests in a route are eagerly initiated and will load data at the same time. This reduces client-server waterfalls and the total time it takes to load data.
With sequential data fetching, requests in a route are dependent on each other and create waterfalls. There may be cases where you want this pattern because one fetch depends on the result of the other, or you want a condition to be satisfied before the next fetch to save resources. However, this behavior can also be unintentional and lead to longer loading times.

Learn how to implement parallel and sequential data fetching.

Automatic fetch() Request Deduping

If you need to fetch the same data (e.g. current user) in multiple components in a tree, Next.js will automatically cache fetch requests (GET) that have the same input in a temporary cache. This optimization prevents the same data from being fetched more than once during a rendering pass.

On the server, the cache lasts the lifetime of a server request until the rendering process completes.

This optimization applies to fetch requests made in Layouts, Pages, Server Components, generateMetadata and generateStaticParams.
This optimization also applies during static generation.

On the client, the cache lasts the duration of a session (which could include multiple client-side re-renders) before a full page reload.

Good to know:

fetch requests are automatically deduplicated under the following conditions.
If you're unable to use fetch, React provides a cache function to allow you to manually cache data for the duration of the request.

Static and Dynamic Data Fetching

There are two types of data: Static and Dynamic.

Static Data is data that doesn't change often. For example, a blog post.
Dynamic Data is data that changes often or can be specific to users. For example, a shopping cart list.

By default, Next.js automatically does static fetches. This means that the data will be fetched at build time, cached, and reused on each request. As a developer, you have

control over how the static data is cached and revalidated.
There are two benefits to using static data:

It reduces the load on your database by minimizing the number of requests made.
The data is automatically cached for improved loading performance.

However, if your data is personalized to the user or you want to always fetch the latest data, you can mark requests as dynamic and fetch data on each request without caching.
Learn how to do Static and Dynamic data fetching.
Caching Data

Caching is the process of storing data in a location (e.g. Content Delivery Network) so it doesn't need to be re-fetched from the original source on each request.

The Next.js Cache is a persistent HTTP cache that can be globally distributed. This means the cache can scale automatically and be shared across multiple regions depending on your platform (e.g. Vercel).
Next.js extends the options object of the fetch() function to allow each request on the server to set its own persistent caching behavior. Together with component-level data fetching, this allows you to configure caching within your application code directly where the data is being used.
During server rendering, when Next.js comes across a fetch, it will check the cache to see if the data is already available. If it is, it will return the cached data. If not, it will fetch and store data for future requests.

Good to know: If you're unable to use fetch, React provides a cache function to allow you to manually cache data for the duration of the request.

Learn more about caching in Next.js.
Revalidating Data

Revalidation is the process of purging the cache and re-fetching the latest data. This is useful when your data changes and you want to ensure your application shows the latest version without having to rebuild your entire application.
Next.js provides two types of revalidation:

Background: Revalidates the data at a specific time interval.

On-demand: Revalidates the data whenever there is an update.

Learn how to revalidate data.
Streaming and Suspense

Streaming and Suspense are new React features that allow you to progressively render and incrementally stream rendered units of the UI to the client.
With Server Components and nested layouts, you're able to instantly render parts of the page that do not specifically require data, and show a loading state for parts of the page that are fetching data. This means the user does not have to wait for the entire page to load before they can start interacting with it.

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

Previous Next.js data fetching methods such as getServerSideProps, getStaticProps, and getInitialProps are not supported in the new App Router. However, you can still use them in the Pages Router.
Next Steps

Now that you understand the fundamentals of data fetching in Next.js, you can learn more about managing data in your application:
Data FetchingThe Next.js App Router allows you to fetch data directly in your React components by marking the function as async and using await for the Promise.
Data fetching is built on top of the fetch() Web API and React Server Components.
When using fetch(), requests are automatically deduped by default.
Next.js extends the fetch options object to allow each request to set its own caching and revalidating.
async and await in Server Components

You can use async and await to fetch data in Server Components.
app/page.tsxTypeScript async function getData() {
  const res = await fetch('https://api.example.com/...')
  // The return value is *not* serialized
  // You can return Date, Map, Set, etc.

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

  return res.json()
}

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

  return <main></main>
}

Good to know:
To use an async Server Component with TypeScript, ensure you are using TypeScript 5.1.3 or higher and @types/react 18.2.8 or higher.

Server Component Functions

Next.js provides helpful server functions you may need when fetching data in Server Components:

cookies()
headers()

use in Client Components

use is a new React function that accepts a promise conceptually similar to await. use

handles the promise returned by a function in a way that is compatible with components, hooks, and Suspense. Learn more about use in the React RFC.
Wrapping fetch in use is currently not recommended in Client Components and may trigger multiple re-renders. For now, if you need to fetch data in a Client Component, we recommend using a third-party library such as SWR or React Query.

Good to know: We'll be adding more examples once fetch and use work in Client Components.

Static Data Fetching

By default, fetch will automatically fetch and cache data indefinitely.
fetch('https://...') // cache: 'force-cache' is the default
Revalidating Data

To revalidate cached data at a timed interval, you can use the next.revalidate option in fetch() to set the cache lifetime of a resource (in seconds).
fetch('https://...', { next: { revalidate: 10 } })
See Revalidating Data for more information.

Good to know:
Caching at the fetch level with revalidate or cache: 'force-cache' stores the data across requests in a shared cache. You should avoid using it for user-specific data (i.e. requests that derive data from cookies() or headers())

Dynamic Data Fetching

To fetch fresh data on every fetch request, use the cache: 'no-store' option.
fetch('https://...', { cache: 'no-store' })
Data Fetching Patterns

Parallel Data Fetching

To minimize client-server waterfalls, we recommend this pattern to fetch data in parallel:
app/artist/[username]/page.tsxTypeScript import Albums from './albums'

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

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

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

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

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

By starting the fetch prior to calling await in the Server Component, each request can eagerly start to fetch requests at the same time. This sets the components up so you can avoid waterfalls.
We can save time by initiating both requests in parallel, however, the user won't see the rendered result until both promises are resolved.
To improve the user experience, you can add a suspense boundary to break up the

rendering work and show part of the result as soon as possible:
artist/[username]/page.tsxTypeScript import { getArtist, getArtistAlbums, type Album } from './api'

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

  // Wait for the artist's promise to resolve first
  const artist = await artistData

  return (
    <>
      <h1>{artist.name}</h1>
      {/* Send the artist information first,
          and wrap albums in a suspense boundary */}
      <Suspense fallback={<div>Loading...</div>}>
        <Albums promise={albumData} />
      </Suspense>
    </>
  )
}

// Albums Component
async function Albums({ promise }: { promise: Promise<Album[]> }) {
  // Wait for the albums promise to resolve
  const albums = await promise

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

Take a look at the preloading pattern for more information on improving components structure.
Sequential Data Fetching

To fetch data sequentially, you can fetch directly inside the component that needs it, or you can await the result of fetch inside the component that needs it:
app/artist/page.tsxTypeScript // ...

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

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

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

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

By fetching data inside the component, each fetch request and nested segment in the route cannot start fetching data and rendering until the previous request or segment has completed.
Blocking Rendering in a Route

By fetching data in a layout, rendering for all route segments beneath it can only start once the data has finished loading.
In the pages directory, pages using server-rendering would show the browser loading spinner until getServerSideProps had finished, then render the React component for that page. This can be described as "all or nothing" data fetching. Either you had the entire data for your page, or none.
In the app directory, you have additional options to explore:

First, you can use loading.js to show an instant loading state from the server while streaming in the result from your data fetching function.
Second, you can move data fetching lower in the component tree to only block rendering for the parts of the page that need it. For example, moving data fetching to a specific component rather than fetching it at the root layout.

Whenever possible, it's best to fetch data in the segment that uses it. This also allows you to show a loading state for only the part of the page that is loading, and not the entire page.
Data Fetching without fetch()

You might not always have the ability to use and configure fetch requests directly if you're using a third-party library such as an ORM or database client.
In cases where you cannot use fetch but still want to control the caching or revalidating behavior of a layout or page, you can rely on the default caching behavior of the segment or use the segment cache configuration.
Default Caching Behavior

Any data fetching libraries that do not use fetch directly will not affect caching of a route, and will be static or dynamic depending on the route segment.
If the segment is static (default), the output of the request will be cached and revalidated (if configured) alongside the rest of the segment. If the segment is dynamic, the output of the request will not be cached and will be re-fetched on every request when the segment is rendered.

Good to know: Dynamic functions like cookies() and headers() will make the route segment dynamic.

Segment Cache Configuration

As a temporary solution, until the caching behavior of third-party queries can be configured, you can use segment configuration to customize the cache behavior of the entire segment.
app/page.tsxTypeScript import prisma from './lib/prisma'

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

async function getPosts() {
  const posts = await prisma.post.findMany()
  return posts
}

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

Caching DataNext.js has built-in support for caching data, both on a per-request basis (recommended) or for an entire route segment.

Per-request Caching

fetch()

By default, all fetch() requests are cached and deduplicated automatically. This means that if you make the same request twice, the second request will reuse the result from the first request.
app/page.tsxTypeScript async function getComments() {
  const res = await fetch('https://...') // The result is cached
  return res.json()
}

// This function is called twice, but the result is only fetched once

```
const comments = await getComments() // cache MISS

// The second call could be anywhere in your application
const comments = await getComments() // cache HIT
```

Requests are not cached if:

Dynamic methods (next/headers, export const POST, or similar) are used and the fetch is a POST request (or uses Authorization or cookie headers)
fetchCache is configured to skip cache by default
revalidate: 0 or cache: 'no-store' is configured on individual fetch

Requests made using fetch can specify a revalidate option to control the revalidation frequency of the request.
app/page.tsxTypeScript
```
export default async function Page() {
  // revalidate this data every 10 seconds at most
  const res = await fetch('https://...', { next: { revalidate: 10 } })
  const data = res.json()
  // ...
}
```

React cache()

React allows you to cache() and deduplicate requests, memoizing the result of the wrapped function call. The same function called with the same arguments will reuse a cached value instead of re-running the function.
utils/getUser.tsTypeScript
```
import { cache } from 'react'

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

app/user/[id]/layout.tsxTypeScript
```
import { getUser } from '@utils/getUser'

export default async function UserLayout({ params: { id } }) {
  const user = await getUser(id)
  // ...
}
```

app/user/[id]/page.tsxTypeScript
```
import { getUser } from '@utils/getUser'
```

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

Although the getUser() function is called twice in the example above, only one query will be made to the database. This is because getUser() is wrapped in cache(), so the second request can reuse the result from the first request.

Good to know:

fetch() caches requests automatically, so you don't need to wrap functions that use fetch() with cache(). See automatic request deduping for more information.
In this new model, we recommend fetching data directly in the component that needs it, even if you're requesting the same data in multiple components, rather than passing the data between components as props.
We recommend using the server-only package to make sure server data fetching functions are never used on the client.

POST requests and cache()

POST requests are automatically deduplicated when using fetch – unless they are inside of POST Route Handler or come after reading headers()/cookies(). For example, if you are using GraphQL and POST requests in the above cases, you can use cache to deduplicate requests. The cache arguments must be flat and only include primitives. Deep objects won't match for deduplication.
utils/getUser.tsTypeScript import { cache } from 'react'

```
export const getUser = cache(async (id: string) => {
  const res = await fetch('...', { method: 'POST', body: '...' })
  // ...
})
```

Preload pattern with cache()

As a pattern, we suggest optionally exposing a preload() export in utilities or components that do data fetching.
components/User.tsxTypeScript import { getUser } from '@utils/getUser'

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

By calling preload, you can eagerly start fetching data you're likely going to need.
app/user/[id]/page.tsxTypeScript import User, { preload } from '@components/User'

```
export default async function Page({
  params: { id },
}: {
  params: { id: string }
}) {
  preload(id) // starting loading the user data now
  const condition = await fetchCondition()
  return condition ? <User id={id} /> : null
}
```

Good to know:

The preload() function can have any name. It's a pattern, not an API.
This pattern is completely optional and something you can use to optimize on a case-by-case basis.
This pattern is a further optimization on top of parallel data fetching. Now you don't have to pass promises down as props and can instead rely on the preload pattern.

Combining cache, preload, and server-only

You can combine the cache function, the preload pattern, and the server-only package

to create a data fetching utility that can be used throughout your app.
utils/getUser.tsTypeScript import { cache } from 'react'
import 'server-only'

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

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

With this approach, you can eagerly fetch data, cache responses, and guarantee that this data fetching only happens on the server.
The getUser.ts exports can be used by layouts, pages, or components to give them control over when a user's data is fetched.
Segment-level Caching

Good to know: We recommend using per-request caching for improved granularity and control over caching.

Segment-level caching allows you to cache and revalidate data used in route segments. This mechanism allows different segments of a path to control the cache lifetime of the entire route. Each page.tsx and layout.tsx in the route hierarchy can export a revalidate value that sets the revalidation time for the route.
app/page.tsxTypeScript export const revalidate = 60 // revalidate this segment every 60 seconds

Good to know:

If a page, layout, and fetch request inside components all specify a revalidate frequency, the lowest value of the three will be used.
Advanced: You can set fetchCache to 'only-cache' or 'force-cache' to ensure that all fetch requests opt into caching but the revalidation frequency might still be lowered by individual fetch requests. See fetchCache for more information.

Revalidating DataNext.js allows you to update specific static routes without needing to rebuild your entire site. Revalidation (also known as Incremental Static Regeneration) allows you to retain the benefits of static while scaling to millions of pages.
There are two types of revalidation in Next.js:

Background: Revalidates the data at a specific time interval.
On-demand: Revalidates the data based on an event such as an update.

Background Revalidation

To revalidate cached data at a specific interval, you can use the next.revalidate option in fetch() to set the cache lifetime of a resource (in seconds).
fetch('https://...', { next: { revalidate: 60 } })
If you want to revalidate data that does not use fetch (i.e. using an external package or query builder), you can use the route segment config.
app/page.tsxTypeScript export const revalidate = 60 // revalidate this page every 60 seconds

In addition to fetch, you can also revalidate data using cache.
How it works

When a request is made to the route that was statically rendered at build time, it will initially show the cached data.
Any requests to the route after the initial request and before 60 seconds are also cached and instantaneous.
After the 60-second window, the next request will still show the cached (stale) data.
Next.js will trigger a regeneration of the data in the background.
Once the route generates successfully, Next.js will invalidate the cache and show the updated route. If the background regeneration fails, the old data would still be unaltered.

When a request is made to a route segment that hasn’t been generated, Next.js will dynamically render the route on the first request. Future requests will serve the static route segments from the cache.

Good to know: Check if your upstream data provider has caching enabled by default. You might need to disable (e.g. useCdn: false), otherwise a revalidation won't be able to pull fresh data to update the ISR cache. Caching can occur at a CDN (for an endpoint being requested) when it returns the Cache-Control header. ISR on Vercel persists the cache globally and handles rollbacks.

On-demand Revalidation

If you set a revalidate time of 60, all visitors will see the same generated version of your site for one minute. The only way to invalidate the cache is if someone visits the page after the minute has passed.
The Next.js App Router supports revalidating content on-demand based on a route or cache tag. This allows you to manually purge the Next.js cache for specific fetches, making it easier to update your site when:

Content from your headless CMS is created or updated.
Ecommerce metadata changes (price, description, category, reviews, etc).

Using On-Demand Revalidation

Data can be revalidated on-demand by path (revalidatePath) or by cache tag (revalidateTag).
For example, the following fetch adds the cache tag collection:
app/page.tsxTypeScript export default async function Page() {
  const res = await fetch('https://...', { next: { tags: ['collection'] } })
  const data = await res.json()
  // ...
}

This cached data can then be revalidated on-demand by calling revalidateTag in a Route Handler.
app/api/revalidate/route.tsTypeScript import { NextRequest, NextResponse } from 'next/server'
import { revalidateTag } from 'next/cache'

export async function GET(request: NextRequest) {
  const tag = request.nextUrl.searchParams.get('tag')
  revalidateTag(tag)
  return NextResponse.json({ revalidated: true, now: Date.now() })
}

Error Handling and Revalidation

If an error is thrown while attempting to revalidate data, the last successfully generated data will continue to be served from the cache. On the next subsequent request, Next.js will retry revalidating the data.
Server ActionsServer Actions are an alpha feature in Next.js, built on top of React Actions. They enable server-side data mutations, reduced client-side JavaScript, and progressively enhanced forms. They can be defined inside Server Components and/or called from Client Components:
With Server Components:
app/add-to-cart.js import { cookies } from 'next/headers'

```
// Server action defined inside a Server Component
export default function AddToCart({ productId }) {
  async function addItem(data) {
    'use server'

    const cartId = cookies().get('cartId')?.value
    await saveToDb({ cartId, data })
  }

  return (
    <form action={addItem}>
      <button type="submit">Add to Cart</button>
    </form>
  )
}
```

With Client Components:
app/actions.js 'use server'

```
export async function addItem(data) {
  const cartId = cookies().get('cartId')?.value
  await saveToDb({ cartId, data })
}
```

app/add-to-cart.js 'use client'

```
import { addItem } from './actions.js'

// Server Action being called inside a Client Component
export default function AddToCart({ productId }) {
  return (
    <form action={addItem}>
      <button type="submit">Add to Cart</button>
    </form>
  )
}
```

Good to know:

Using Server Actions will opt into running the React experimental channel.
React Actions, useOptimistic, and useFormStatus are not a Next.js or React Server Components specific features.
Next.js integrates React Actions into the Next.js router, bundler, and caching system, including adding data mutation APIs like revalidateTag and revalidatePath.

Convention

You can enable Server Actions in your Next.js project by enabling the experimental serverActions flag.
next.config.js module.exports = {
  experimental: {
    serverActions: true,
  },
}
Creation

Server Actions can be defined in two places:

Inside the component that uses it (Server Components only)
In a separate file (Client and Server Components), for reusability. You can define multiple Server Actions in a single file.

With Server Components

Create a Server Action by defining an asynchronous function with the "use server" directive at the top of the function body. This function should have serializable arguments and a serializable return value based on the React Server Components protocol.
app/server-component.js export default function ServerComponent() {

```
  async function myAction() {
    'use server'
    // ...
  }
}
```
With Client Components

If you're using a Server Action inside a Client Component, create your action in a separate file with the "use server" directive at the top of the file. Then, import the Server Action into your Client Component:
app/actions.js 'use server'

```
export async function myAction() {
  // ...
}
```
app/client-component.js 'use client'

```
import { myAction } from './actions'

export default function ClientComponent() {
  return (
    <form action={myAction}>
      <button type="submit">Add to Cart</button>
    </form>
  )
}
```

Good to know: When using a top-level "use server" directive, all exports below will be considered Server Actions. You can have multiple Server Actions in a single file.

Invocation

You can invoke Server Actions using the following methods:

Using action: React's action prop allows invoking a Server Action on a <form> element.
Using formAction: React's formAction prop allows handling <button>, <input type="submit">, and <input type="image"> elements in a <form>.
Custom Invocation with startTransition: Invoke Server Actions without using action or

formAction by using startTransition. This method disables Progressive Enhancement.

action

You can use React's action prop to invoke a Server Action on a form element. Server Actions passed with the action prop act as asynchronous side effects in response to user interaction.

app/add-to-cart.js

```
export default function AddToCart({ productId }) {
  async function addItem(data) {
    'use server'

    const cartId = cookies().get('cartId')?.value
    await saveToDb({ cartId, data })
  }

  return (
    <form action={addItem}>
      <button type="submit">Add to Cart</button>
    </form>
  )
}
```

Good to know: An action is similar to the HTML primitive action

formAction

You can use formAction prop to handle Form Actions on elements such as button, input type="submit", and input type="image". The formAction prop takes precedence over the form's action.

app/form.js

```
export default function Form() {
  async function handleSubmit() {
    'use server'
    // ...
  }

  async function submitImage() {
    'use server'
    // ...
```

```
  }

  return (
    <form action={handleSubmit}>
      <input type="text" name="name" />
      <input type="image" formAction={submitImage} />
      <button type="submit">Submit</button>
    </form>
  )
}
```

Good to know: A formAction is the HTML primitive formaction. React now allows you to pass functions to this attribute.

Custom invocation using startTransition

You can also invoke Server Actions without using action or formAction. You can achieve this by using startTransition provided by the useTransition hook, which can be useful if you want to use Server Actions outside of forms, buttons, or inputs.

Good to know: Using startTransition disables the out-of-the-box Progressive Enhancement.

app/components/example-client-component.js 'use client'

```
import { useTransition } from 'react'
import { addItem } from '../actions'

function ExampleClientComponent({ id }) {
  let [isPending, startTransition] = useTransition()

  return (
    <button onClick={() => startTransition(() => addItem(id))}>
      Add To Cart
    </button>
  )
}
```

app/actions.js 'use server'

```
export async function addItem(id) {
  await addItemToDb(id)
  // Marks all product pages for revalidating
```

```
  revalidatePath('/product/[id]')
}
```
Custom invocation without startTransition

If you aren't doing Server Mutations, you can directly pass the function as a prop like any other function.

app/posts/[id]/page.tsxTypeScript
```
import kv from '@vercel/kv'
import LikeButton from './like-button'

export default function Page({ params }: { params: { id: string } }) {
  async function increment() {
    'use server'
    await kv.incr(`post:id:${params.id}`)
  }

  return <LikeButton increment={increment} />
}
```

app/post/[id]/like-button.tsxTypeScript
```
'use client'

export default function LikeButton({
  increment,
}: {
  increment: () => Promise<void>
}) {
  return (
    <button
      onClick={async () => {
        await increment()
      }}
    >
      Like
    </button>
  )
}
```

Enhancements

Experimental useOptimistic

The experimental useOptimistic hook provides a way to implement optimistic updates in your application. Optimistic updates are a technique that enhances user experience by making the app appear more responsive.
When a Server Action is invoked, the UI is updated immediately to reflect the expected outcome, instead of waiting for the Server Action's response.
app/thread.js 'use client'

```
import { experimental_useOptimistic as useOptimistic } from 'react'
import { send } from './actions.js'

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

  return (
    <div>
      {optimisticMessages.map((m) => (
        <div>
          {m.message}
          {m.sending ? 'Sending...' : ''}
        </div>
      ))}
      <form
        action={async (formData) => {
          const message = formData.get('message')
          formRef.current.reset()
          addOptimisticMessage(message)
          await send(message)
        }}
        ref={formRef}
      >
        <input type="text" name="message" />
      </form>
    </div>
  )
}
```

Experimental useFormStatus

The experimental useFormStatus hook can be used within Form Actions, and provides the pending property.
app/form.js 'use client'

```
import { experimental_useFormStatus as useFormStatus } from 'react-dom'

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

  return (
    <input
      type="submit"
      className={pending ? 'button-pending' : 'button-normal'}
      disabled={pending}
    >
      Submit
    </input>
  )
}
```

Progressive Enhancement

Progressive Enhancement allows a <form> to function properly without JavaScript, or with JavaScript disabled. This allows users to interact with the form and submit data even if the JavaScript for the form hasn't been loaded yet or if it fails to load.
Both Server Form Actions and Client Form Actions support Progressive Enhancement, using one of two strategies:

If a Server Action is passed directly to a <form>, the form is interactive even if JavaScript is disabled.
If a Client Action is passed to a <form>, the form is still interactive, but the action will be placed in a queue until the form has hydrated. The <form> is prioritized with Selective Hydration, so it happens quickly.

app/components/example-client-component.js 'use client'

```
import { useState } from 'react'
import { handleSubmit } from './actions.js'
```

```
export default function ExampleClientComponent({ myAction }) {
  const [input, setInput] = useState()

  return (
    <form action={handleSubmit} onChange={(e) => setInput(e.target.value)}>
      {/* ... */}
    </form>
  )
}
```

In both cases, the form is interactive before hydration occurs. Although Server Actions have the additional benefit of not relying on client JavaScript, you can still compose additional behavior with Client Actions where desired without sacrificing interactivity.

Size Limitation

By default, the maximum size of the request body sent to a Server Action is 1MB. This prevents large amounts of data being sent to the server, which consumes a lot of server resource to parse.

However, you can configure this limit using the experimental serverActionsBodySizeLimit option. It can take the number of bytes or any string format supported by bytes, for example 1000, '500kb' or '3mb'.

next.config.js

```
module.exports = {
  experimental: {
    serverActions: true,
    serverActionsBodySizeLimit: '2mb',
  },
}
```

Examples

Usage with Client Components

Import

Server Actions cannot be defined within Client Components, but they can be imported. To use Server Actions in Client Components, you can import the action from a file containing a top-level "use server" directive.
app/actions.js 'use server'

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

app/components/example-client-component.js 'use client'

```
import { useTransition } from 'react'
import { addItem } from '../actions'

function ExampleClientComponent({ id }) {
  let [isPending, startTransition] = useTransition()

  return (
    <button onClick={() => startTransition(() => addItem(id))}>
      Add To Cart
    </button>
  )
}
```

Props

Although importing Server Actions is recommended, in some cases you might want to pass down a Server Action to a Client Component as a prop.
For example, you might want to use a dynamically generated value within the action. In that case, passing a Server Action down as a prop might be a viable solution.
app/components/example-server-component.js import { ExampleClientComponent } from './components/example-client-component.js'

```
function ExampleServerComponent({ id }) {
  async function updateItem(data) {
    'use server'
    modifyItem({ id, data })
  }

  return <ExampleClientComponent updateItem={updateItem} />
}
```

app/components/example-client-component.js 'use client'

```
function ExampleClientComponent({ updateItem }) {
  async function action(formData: FormData) {
    await updateItem(formData)
  }

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

On-demand Revalidation

Server Actions can be used to revalidate data on-demand by path (revalidatePath) or by cache tag (revalidateTag).

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

async function revalidate() {
  'use server'
  revalidateTag('blog-posts')
}
```

Validation

The data passed to a Server Action can be validated or sanitized before invoking the action. For example, you can create a wrapper function that receives the action as its argument, and returns a function that invokes the action if it's valid.

app/actions.js 'use server'

```
import { withValidate } from 'lib/form-validation'

export const action = withValidate((data) => {
  // ...
})
```

lib/form-validation.js

```
export function withValidate(action) {
  return async (formData: FormData) => {
```

```
    'use server'

    const isValidData = verifyData(formData)

    if (!isValidData) {
      throw new Error('Invalid input.')
    }

    const data = process(formData)
    return action(data)
  }
}
```

Using headers

You can read incoming request headers such as cookies and headers within a Server Action.

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

async function addItem(data) {
  'use server'

  const cartId = cookies().get('cartId')?.value

  await saveToDb({ cartId, data })
}
```

Additionally, you can modify cookies within a Server Action.

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

async function create(data) {
  'use server';

  const cart = await createCart():
  cookies().set('cartId', cart.id)
  // or
  cookies().set({
    name: 'cartId',
    value: cart.id,
    httpOnly: true,
    path: '/'
  })
}
```

Glossary

Actions

Actions are an experimental feature in React, allowing you to run async code in response to a user interaction.
Actions are not Next.js or React Server Components specific, however, they are not yet available in the stable version of React. When using Actions through Next.js, you are opting into using the React experimental channel.
Actions are defined through the action prop on an element. Typically when building HTML forms, you pass a URL to the action prop. With Actions, React now allows you to pass a function directly.
React also provides built-in solutions for optimistic updates with Actions. It's important to note new patterns are still being developed and new APIs may still be added.
Form Actions

Actions integrated into the web standard <form> API, and enable out-of-the-box progressive enhancement and loading states. Similar to the HTML primitive formaction.
Server Functions

Functions that run on the server, but can be called on the client.
Server Actions

Server Functions called as an action.
Server Actions can be progressively enhanced by passing them to a form element's action prop. The form is interactive before any client-side JavaScript has loaded. This means React hydration is not required for the form to submit.

Server Mutations

Server Actions that mutates your data and calls redirect, revalidatePath, or revalidateTag.
Styling
Next.js supports different ways of styling your application, including:

Global CSS: Simple to use and familiar for those experienced with traditional CSS, but can lead to larger CSS bundles and difficulty managing styles as the application grows.
CSS Modules: Create locally scoped CSS classes to avoid naming conflicts and improve maintainability.
Tailwind CSS: A utility-first CSS framework that allows for rapid custom designs by composing utility classes.
Sass: A popular CSS preprocessor that extends CSS with features like variables, nested rules, and mixins.
CSS-in-JS: Embed CSS directly in your JavaScript components, enabling dynamic and scoped styling.

Learn more about each approach by exploring their respective documentation:
CSS Modules

Next.js has built-in support for CSS Modules using the .module.css extension.
CSS Modules locally scope CSS by automatically creating a unique class name. This allows you to use the same class name in different files without worrying about collisions. This behavior makes CSS Modules the ideal way to include component-level CSS.
Example

CSS Modules can be imported into any file inside the app directory:app/dashboard/ layout.tsxTypeScript import styles from './styles.module.css'

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

```
  padding: 24px;
}
```

CSS Modules are an optional feature and are only enabled for files with the .module.css extension.
Regular <link> stylesheets and global CSS files are still supported.
In production, all CSS Module files will be automatically concatenated into many minified and code-split .css files.
These .css files represent hot execution paths in your application, ensuring the minimal amount of CSS is loaded for your application to paint.
Global Styles

Global styles can be imported into any layout, page, or component inside the app directory.
Good to know: This is different from the pages directory, where you can only import global styles inside the _app.js file.
For example, consider a stylesheet named app/global.css: body {
  padding: 20px 20px 60px;
  max-width: 680px;
  margin: 0 auto;
}Inside the root layout (app/layout.js), import the global.css stylesheet to apply the styles to every route in your application:app/layout.tsxTypeScript // These styles apply to every route in the application
import './global.css'

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

External Stylesheets

Stylesheets published by external packages can be imported anywhere in the app directory, including colocated components:app/layout.tsxTypeScript import 'bootstrap/dist/css/bootstrap.css'

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

Good to know: External stylesheets must be directly imported from an npm package or downloaded and colocated with your codebase. You cannot use <link rel="stylesheet" />.

Additional Features

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

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

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

Install the Tailwind CSS packages and run the init command to generate both the

tailwind.config.js and postcss.config.js files:
Terminal npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
Configuring Tailwind

Inside tailwind.config.js, add paths to the files that will use Tailwind CSS class names:
tailwind.config.js /** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    './app/**/*.{js,ts,jsx,tsx,mdx}', // Note the addition of the `app` directory.
    './pages/**/*.{js,ts,jsx,tsx,mdx}',
    './components/**/*.{js,ts,jsx,tsx,mdx}',

    // Or if using `src` directory:
    './src/**/*.{js,ts,jsx,tsx,mdx}',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
You do not need to modify postcss.config.js.
Importing Styles

Add the Tailwind CSS directives that Tailwind will use to inject its generated styles to a Global Stylesheet in your application, for example:app/globals.css @tailwind base;
@tailwind components;
@tailwind utilities;Inside the root layout (app/layout.tsx), import the globals.css stylesheet to apply the styles to every route in your application.app/layout.tsxTypeScript
import type { Metadata } from 'next'

// These styles apply to every route in the application
import './globals.css'

export const metadata: Metadata = {
  title: 'Create Next App',
  description: 'Generated by create next app',
}

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

After installing Tailwind CSS and adding the global styles, you can use Tailwind's utility classes in your application.app/page.tsxTypeScript
```
export default function Page() {
  return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>
}
```

Usage with Turbopack

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

Warning: CSS-in-JS libraries which require runtime JavaScript are not currently supported in Server Components. Using CSS-in-JS with newer React features like Server Components and Streaming requires library authors to support the latest version of React, including concurrent rendering.
We're working with the React team on upstream APIs to handle CSS and JavaScript assets with support for React Server Components and streaming architecture.
The following libraries are supported in Client Components in the app directory (alphabetical):
kuma-ui
pandacss
styled-jsx
styled-components
style9
tamagui
vanilla-extract
The following are currently working on support:
emotion

Material UI

Good to know: We're testing out different CSS-in-JS libraries and we'll be adding more examples for libraries that support React 18 features and/or the app directory.
If you want to style Server Components, we recommend using CSS Modules or other solutions that output CSS files, like PostCSS or Tailwind CSS.Configuring CSS-in-JS in app

Configuring CSS-in-JS is a three-step opt-in process that involves:
A style registry to collect all CSS rules in a render.
The new useServerInsertedHTML hook to inject rules before any content that might use them.
A Client Component that wraps your app with the style registry during initial server-side rendering.
styled-jsx

Using styled-jsx in Client Components requires using v5.1.0. First, create a new registry:app/registry.tsxTypeScript 'use client'

```
import React, { useState } from 'react'
import { useServerInsertedHTML } from 'next/navigation'
import { StyleRegistry, createStyleRegistry } from 'styled-jsx'

export default function StyledJsxRegistry({
  children,
}: {
  children: React.ReactNode
}) {
  // Only create stylesheet once with lazy initial state
  // x-ref: https://reactjs.org/docs/hooks-reference.html#lazy-initial-state
  const [jsxStyleRegistry] = useState(() => createStyleRegistry())

  useServerInsertedHTML(() => {
    const styles = jsxStyleRegistry.styles()
    jsxStyleRegistry.flush()
    return <>{styles}</>
  })

  return <StyleRegistry registry={jsxStyleRegistry}>{children}</StyleRegistry>
}
```

Then, wrap your root layout with the registry:app/layout.tsxTypeScript import

```
StyledJsxRegistry from './registry'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <body>
        <StyledJsxRegistry>{children}</StyledJsxRegistry>
      </body>
    </html>
  )
}
```
View an example here.Styled Components

Below is an example of how to configure styled-components@v6.0.0-rc.1 or greater:First, use the styled-components API to create a global registry component to collect all CSS style rules generated during a render, and a function to return those rules. Then use the useServerInsertedHTML hook to inject the styles collected in the registry into the <head> HTML tag in the root layout.lib/registry.tsxTypeScript 'use client'

```
import React, { useState } from 'react'
import { useServerInsertedHTML } from 'next/navigation'
import { ServerStyleSheet, StyleSheetManager } from 'styled-components'

export default function StyledComponentsRegistry({
  children,
}: {
  children: React.ReactNode
}) {
  // Only create stylesheet once with lazy initial state
  // x-ref: https://reactjs.org/docs/hooks-reference.html#lazy-initial-state
  const [styledComponentsStyleSheet] = useState(() => new ServerStyleSheet())

  useServerInsertedHTML(() => {
    const styles = styledComponentsStyleSheet.getStyleElement()
    styledComponentsStyleSheet.instance.clearTag()
    return <>{styles}</>
  })

  if (typeof window !== 'undefined') return <>{children}</>
```

```
  return (
    <StyleSheetManager sheet={styledComponentsStyleSheet.instance}>
      {children}
    </StyleSheetManager>
  )
}Wrap the children of the root layout with the style registry component:app/
layout.tsxTypeScript import StyledComponentsRegistry from './lib/registry'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <body>
        <StyledComponentsRegistry>{children}</StyledComponentsRegistry>
      </body>
    </html>
  )
}View an example here.
```
Good to know:

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

Sass
Next.js has built-in support for Sass using both the .scss and .sass extensions. You can use component-level Sass via CSS Modules and the .module.scssor .module.sass extension.
First, install sass:
Terminal npm install --save-dev sass

Good to know:
Sass supports two different syntax, each with their own extension.
The .scss extension requires you use the SCSS syntax,

while the .sass extension requires you use the Indented Syntax ("Sass").
If you're not sure which to choose, start with the .scss extension which is a superset of CSS, and doesn't require you learn the
Indented Syntax ("Sass").

Customizing Sass Options

If you want to configure the Sass compiler, use sassOptions in next.config.js.
next.config.js const path = require('path')

```
module.exports = {
  sassOptions: {
    includePaths: [path.join(__dirname, 'styles')],
  },
}
```

Sass Variables

Next.js supports Sass variables exported from CSS Module files.
For example, using the exported primaryColor Sass variable:
app/variables.module.scss $primary-color: #64ff00;

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

app/page.js // maps to root `/` URL

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

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

Optimizations
Next.js comes with a variety of built-in optimizations designed to improve your application's speed and Core Web Vitals. This guide will cover the optimizations you can leverage to enhance your user experience.
Built-in Components

Built-in components abstract away the complexity of implementing common UI optimizations. These components are:

Images: Built on the native <img> element. The Image Component optimizes images for performance by lazy loading and automatically resizing images based on device size.
Link: Built on the native <a> tags. The Link Component prefetches pages in the background, for faster and smoother page transitions.
Scripts: Built on the native <script> tags. The Script Component gives you control over loading and execution of third-party scripts.

Metadata

Metadata helps search engines understand your content better (which can result in better SEO), and allows you to customize how your content is presented on social media, helping you create a more engaging and consistent user experience across various platforms.
The Metadata API in Next.js allows you to modify the <head> element of a page. You can configure metadata in two ways:
Config-based Metadata: Export a static metadata object or a dynamic generateMetadata function in a layout.js or page.js file.
File-based Metadata: Add static or dynamically generated special files to route segments.
Additionally, you can create dynamic Open Graph Images using JSX and CSS with imageResponse constructor.

Static Assets

Next.js /public folder can be used to serve static assets like images, fonts, and other files. Files inside /public can also be cached by CDN providers so that they are delivered efficiently.
Analytics and Monitoring

For large applications, Next.js integrates with popular analytics and monitoring tools to help you understand how your application is performing. Learn more in the OpenTelemetry and Instrumentation guides.
Image Optimization
Examples
Image Component

According to Web Almanac, images account for a huge portion of the typical website’s page weight and can have a sizable impact on your website's LCP performance.
The Next.js Image component extends the HTML <img> element with features for automatic image optimization:

Size Optimization: Automatically serve correctly sized images for each device, using modern image formats like WebP and AVIF.
Visual Stability: Prevent layout shift automatically when images are loading.
Faster Page Loads: Images are only loaded when they enter the viewport using native browser lazy loading, with optional blur-up placeholders.
Asset Flexibility: On-demand image resizing, even for images stored on remote servers

Ø<ß¥ Watch: Learn more about how to use next/image !’ YouTube (9 minutes).

Usage

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

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

To use a local image, import your .jpg, .png, or .webp image files.
Next.js will automatically determine the width and height of your image based on the imported file. These values are used to prevent Cumulative Layout Shift while your image is loading.
app/page.js import Image from 'next/image'
import profilePic from './me.png'

export default function Page() {

```
  return (
    <Image
      src={profilePic}
      alt="Picture of the author"
      // width={500} automatically provided
      // height={500} automatically provided
      // blurDataURL="data:..." automatically provided
      // placeholder="blur" // Optional blur-up while loading
    />
  )
}
```

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

Remote Images

To use a remote image, the src property should be a URL string.
Since Next.js does not have access to remote files during the build process, you'll need to provide the width, height and optional blurDataURL props manually.
The width and height attributes are used to infer the correct aspect ratio of image and avoid layout shift from the image loading in. The width and height do not determine the rendered size of the image file. Learn more about Image Sizing.
app/page.js import Image from 'next/image'

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

To safely allow optimizing images, define a list of supported URL patterns in next.config.js. Be as specific as possible to prevent malicious usage. For example, the following configuration will only allow images from a specific AWS S3 bucket:
next.config.js module.exports = {

```
  images: {
    remotePatterns: [
```

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

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

Domains

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

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

Learn more about remotePatterns configuration.

Loaders

Note that in the example earlier, a partial URL ("/me.png") is provided for a remote image. This is possible because of the loader architecture.

A loader is a function that generates the URLs for your image. It modifies the provided src, and generates multiple URLs to request the image at different sizes. These multiple URLs are used in the automatic srcset generation, so that visitors to your site will be served an image that is the right size for their viewport.

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

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

Priority

You should add the priority property to the image that will be the Largest Contentful Paint (LCP) element for each page. Doing so allows Next.js to specially prioritize the image for loading (e.g. through preload tags or priority hints), leading to a meaningful boost in LCP.
The LCP element is typically the largest image or text block visible within the viewport of the page. When you run next dev, you'll see a console warning if the LCP element is an <Image> without the priority property.
Once you've identified the LCP image, you can add the property like this:

app/page.js import Image from 'next/image'
import profilePic from '../public/me.png'

export default function Page() {
  return <Image src={profilePic} alt="Picture of the author" priority />
}
See more about priority in the next/image component documentation.
Image Sizing

One of the ways that images most commonly hurt performance is through layout shift, where the image pushes other elements around on the page as it loads in. This performance problem is so annoying to users that it has its own Core Web Vital, called Cumulative Layout Shift. The way to avoid image-based layout shifts is to always size your images. This allows the browser to reserve precisely enough space for the image before it loads.
Because next/image is designed to guarantee good performance results, it cannot be used in a way that will contribute to layout shift, and must be sized in one of three ways:

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

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

object-position to define how the image should occupy that space.
Normalize your images
If you're serving images from a source that you control, consider modifying your image pipeline to normalize the images to a specific size.
Modify your API calls
If your application is retrieving image URLs using an API call (such as to a CMS), you may be able to modify the API call to return the image dimensions along with the URL.

If none of the suggested methods works for sizing your images, the next/image component is designed to work well on a page alongside standard <img> elements.
Styling

Styling the Image component is similar to styling a normal <img> element, but there are a few guidelines to keep in mind:

Use className or style, not styled-jsx.

In most cases, we recommend using the className prop. This can be an imported CSS Module, a global stylesheet, etc.
You can also use the style prop to assign inline styles.
You cannot use styled-jsx because it's scoped to the current component (unless you mark the style as global).

When using fill, the parent element must have position: relative

This is necessary for the proper rendering of the image element in that layout mode.

When using fill, the parent element must have display: block

This is the default for elements but should be specified otherwise.

Examples

Responsive

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

export default function Responsive() {
  return (
    <div style={{ display: 'flex', flexDirection: 'column' }}>
      <Image
        alt="Mountains"
        // Importing an image will
        // automatically set the width and height
        src={mountains}
        sizes="100vw"
        // Make the image display full width
        style={{
          width: '100%',
          height: 'auto',
        }}
      />
    </div>
  )
}
```

Fill Container

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

export default function Fill() {
  return (
    <div
      style={{
        display: 'grid',
        gridGap: '8px',
        gridTemplateColumns: 'repeat(auto-fit, minmax(400px, auto))',
      }}
    >
```

```
      <div style={{ position: 'relative', height: '400px' }}>
        <Image
          alt="Mountains"
          src={mountains}
          fill
          sizes="(min-width: 808px) 50vw, 100vw"
          style={{
            objectFit: 'cover', // cover, contain, none
          }}
        />
      </div>
      {/* And more images in the grid... */}
    </div>
  )
}
```

Background Image

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

export default function Background() {
  return (
    <Image
      alt="Mountains"
      src={mountains}
      placeholder="blur"
      quality={100}
      fill
      sizes="100vw"
      style={{
        objectFit: 'cover',
      }}
    />
  )
}
```

For examples of the Image component used with the various styles, see the Image Component Demo.

Other Properties

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

The next/image component and Next.js Image Optimization API can be configured in the next.config.js file. These configurations allow you to enable remote images, define custom image breakpoints, change caching behavior and more.
Read the full image configuration documentation for more information.
Font Optimization
next/font will automatically optimize your fonts (including custom fonts) and remove external network requests for improved privacy and performance.

Ø<ß¥ Watch: Learn more about how to use next/font !’ YouTube (6 minutes).

next/font includes built-in automatic self-hosting for any font file. This means you can optimally load web fonts with zero layout shift, thanks to the underlying CSS size-adjust property used.
This new font system also allows you to conveniently use all Google Fonts with performance and privacy in mind. CSS and font files are downloaded at build time and self-hosted with the rest of your static assets. No requests are sent to Google by the browser.
Google Fonts

Automatically self-host any Google Font. Fonts are included in the deployment and served from the same domain as your deployment. No requests are sent to Google by the browser.
Get started by importing the font you would like to use from next/font/google as a function. We recommend using variable fonts for the best performance and flexibility.
app/layout.tsxTypeScript import { Inter } from 'next/font/google'

```
// If loading a variable font, you don't need to specify the font weight
const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
})

export default function RootLayout({
```

```
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={inter.className}>
      <body>{children}</body>
    </html>
  )
}
```
If you can't use a variable font, you will need to specify a weight:app/layout.tsxTypeScript import { Roboto } from 'next/font/google'

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

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

You can specify multiple weights and/or styles by using an array:
app/layout.js const roboto = Roboto({
```
  weight: ['400', '700'],
  style: ['normal', 'italic'],
  subsets: ['latin'],
  display: 'swap',
})
```

Good to know: Use an underscore (_) for font names with multiple words. E.g. Roboto Mono should be imported as Roboto_Mono.

Specifying a subset

Google Fonts are automatically subset. This reduces the size of the font file and improves performance. You'll need to define which of these subsets you want to preload. Failing to specify any subsets while preload is true will result in a warning. This can be done by adding it to the function call:
app/layout.tsxTypeScript const inter = Inter({ subsets: ['latin'] })

View the Font API Reference for more information.
Using Multiple Fonts

You can import and use multiple fonts in your application. There are two approaches you can take.
The first approach is to create a utility function that exports a font, imports it, and applies its className where needed. This ensures the font is preloaded only when it's rendered:
app/fonts.tsTypeScript import { Inter, Roboto_Mono } from 'next/font/google'

```
export const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
})

export const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  display: 'swap',
})
```

app/layout.tsxTypeScript import { inter } from './fonts'

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

app/page.tsxTypeScript import { roboto_mono } from './fonts'

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

```
    <>
      <h1 className={roboto_mono.className}>My page</h1>
    </>
  )
}
```
In the example above, Inter will be applied globally, and Roboto Mono can be imported and applied as needed.
Alternatively, you can create a CSS variable and use it with your preferred CSS solution:
app/layout.tsxTypeScript import { Inter, Roboto_Mono } from 'next/font/google'
import styles from './global.css'

```
const inter = Inter({
  subsets: ['latin'],
  variable: '--font-inter',
  display: 'swap',
})

const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  variable: '--font-roboto-mono',
  display: 'swap',
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={`${inter.variable} ${roboto_mono.variable}`}>
      <body>
        <h1>My App</h1>
        <div>{children}</div>
      </body>
    </html>
  )
}
```
app/global.css html {
```
  font-family: var(--font-inter);
}

h1 {
  font-family: var(--font-roboto-mono);
}
```
In the example above, Inter will be applied globally, and any <h1> tags will be styled with Roboto Mono.

Recommendation: Use multiple fonts conservatively since each new font is an additional resource the client has to download.

Local Fonts

Import next/font/local and specify the src of your local font file. We recommend using variable fonts for the best performance and flexibility.
app/layout.tsxTypeScript import localFont from 'next/font/local'

```
// Font files can be colocated inside of `app`
const myFont = localFont({
  src: './my-font.woff2',
  display: 'swap',
})

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

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

```
 const roboto = localFont({
  src: [
    {
      path: './Roboto-Regular.woff2',
      weight: '400',
      style: 'normal',
    },
    {
      path: './Roboto-Italic.woff2',
      weight: '400',
      style: 'italic',
    },
    {
```

```
      path: './Roboto-Bold.woff2',
      weight: '700',
      style: 'normal',
    },
    {
      path: './Roboto-BoldItalic.woff2',
      weight: '700',
      style: 'italic',
    },
  ],
})
```

View the Font API Reference for more information.

With Tailwind CSS

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

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

app/layout.tsxTypeScript
```
import { Inter, Roboto_Mono } from 'next/font/google'

const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
  variable: '--font-inter',
})

const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  display: 'swap',
  variable: '--font-roboto-mono',
})

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

```
  )
}
```

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

```
tailwind.config.js /** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    './pages/**/*.{js,ts,jsx,tsx}',
    './components/**/*.{js,ts,jsx,tsx}',
    './app/**/*.{js,ts,jsx,tsx}',
  ],
  theme: {
    extend: {
      fontFamily: {
        sans: ['var(--font-inter)'],
        mono: ['var(--font-roboto-mono)'],
      },
    },
  },
  plugins: [],
}
```

You can now use the font-sans and font-mono utility classes to apply the font to your elements.

Preloading

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

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

Reusing fonts

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

to do the following:

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

To load a third-party script for multiple routes, import next/script and include the script directly in your layout component:app/dashboard/layout.tsxTypeScript import Script from 'next/script'

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

The third-party script is fetched when the folder route (e.g. dashboard/page.js) or any nested route (e.g. dashboard/settings/page.js) is accessed by the user. Next.js will ensure the script will only load once, even if a user navigates between multiple routes in the same layout.
Application Scripts

To load a third-party script for all routes, import next/script and include the script directly in your root layout:app/layout.tsxTypeScript import Script from 'next/script'

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

```
      <body>{children}</body>
      <Script src="https://example.com/script.js" />
    </html>
  )
}
```

This script will load and execute when any route in your application is accessed. Next.js will ensure the script will only load once, even if a user navigates between multiple pages.

Recommendation: We recommend only including third-party scripts in specific pages or layouts in order to minimize any unnecessary impact to performance.

Strategy

Although the default behavior of next/script allows you load third-party scripts in any page or layout, you can fine-tune its loading behavior by using the strategy property:

beforeInteractive: Load the script before any Next.js code and before any page hydration occurs.
afterInteractive: (default) Load the script early but after some hydration on the page occurs.
lazyOnload: Load the script later during browser idle time.
worker: (experimental) Load the script in a web worker.

Refer to the next/script API reference documentation to learn more about each strategy and their use cases.
Offloading Scripts To A Web Worker (Experimental)

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

Scripts that use the worker strategy are offloaded and executed in a web worker with Partytown. This can improve the performance of your site by dedicating the main thread to the rest of your application code.
This strategy is still experimental and can only be used if the nextScriptWorkers flag is enabled in next.config.js:

```
next.config.js module.exports = {
  experimental: {
    nextScriptWorkers: true,
  },
}
```
Then, run next (normally npm run dev or yarn dev) and Next.js will guide you through the installation of the required packages to finish the setup:
Terminal npm run dev
You'll see instructions like these: Please install Partytown by running npm install @builder.io/partytown
Once setup is complete, defining strategy="worker" will automatically instantiate Partytown in your application and offload the script to a web worker.
pages/home.tsxTypeScript import Script from 'next/script'

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

There are a number of trade-offs that need to be considered when loading a third-party script in a web worker. Please see Partytown's tradeoffs documentation for more information.
Inline Scripts

Inline scripts, or scripts not loaded from an external file, are also supported by the Script component. They can be written by placing the JavaScript within curly braces:
```
 <Script id="show-banner">
  {`document.getElementById('banner').classList.remove('hidden')`}
</Script>
```
Or by using the dangerouslySetInnerHTML property:
```
 <Script
  id="show-banner"
  dangerouslySetInnerHTML={{
    __html: `document.getElementById('banner').classList.remove('hidden')`,
  }}
/>
```

Warning: An id property must be assigned for inline scripts in order for Next.js to track and optimize the script.

Executing Additional Code

Event handlers can be used with the Script component to execute additional code after a certain event occurs:

onLoad: Execute code after the script has finished loading.
onReady: Execute code after the script has finished loading and every time the component is mounted.
onError: Execute code if the script fails to load.

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

```
import Script from 'next/script'

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

Refer to the next/script API reference to learn more about each event handler and view examples.

Additional Attributes

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

```
export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        id="example-script"
        nonce="XUENAJFW"
        data-test="script"
      />
    </>
  )
}
```

MetadataNext.js has a Metadata API that can be used to define your application metadata (e.g. meta and link tags inside your HTML head element) for improved SEO and web shareability.
There are two ways you can add metadata to your application:

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

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

To define static metadata, export a Metadata object from a layout.js or static page.js file.
layout.tsx / page.tsxTypeScript import { Metadata } from 'next'

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

export default function Page() {}
```

For all the available options, see the API Reference.
Dynamic Metadata

You can use generateMetadata function to fetch metadata that requires dynamic values.
app/products/[id]/page.tsxTypeScript import { Metadata, ResolvingMetadata } from 'next'

```
type Props = {
  params: { id: string }
  searchParams: { [key: string]: string | string[] | undefined }
}

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

  // fetch data
  const product = await fetch(`https://.../${id}`).then((res) => res.json())

  // optionally access and extend (rather than replace) parent metadata
  const previousImages = (await parent).openGraph?.images || []

  return {
    title: product.title,
    openGraph: {
      images: ['/some-specific-page-image.jpg', ...previousImages],
    },
  }
}

export default function Page({ params, searchParams }: Props) {}
```

For all the available params, see the API Reference.

Good to know:

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

File-based metadata

These special files are available for metadata:

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

You can use these for static metadata, or you can programatically generate these files with code.
For implementation and examples, see the Metadata Files API Reference and Dynamic Image Generation.
Behavior

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

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

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

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

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

Ordering

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

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

Merging

Following the evaluation order, Metadata objects exported from multiple segments in the same route are shallowly merged together to form the final metadata output of a route. Duplicate keys are replaced based on their ordering.
This means metadata with nested fields such as openGraph and robots that are defined in an earlier segment are overwritten by the last segment to define them.
Overwriting fields

```
app/layout.js export const metadata = {
  title: 'Acme',
  openGraph: {
    title: 'Acme',
    description: 'Acme is a...',
  },
}
app/blog/page.js export const metadata = {
  title: 'Blog',
  openGraph: {
    title: 'Blog',
  },
}

// Output:
// <title>Blog</title>
// <meta property="og:title" content="Blog" />
```

In the example above:

title from app/layout.js is replaced by title in app/blog/page.js.
All openGraph fields from app/layout.js are replaced in app/blog/page.js because app/blog/page.js sets openGraph metadata. Note the absence of openGraph.description.

If you'd like to share some nested fields between segments while overwriting others, you can pull them out into a separate variable:
app/shared-metadata.js export const openGraphImage = { images: ['http://...'] }
app/page.js import { openGraphImage } from './shared-metadata'

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

app/about/page.js import { openGraphImage } from '../shared-metadata'

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

In the example above, the OG image is shared between app/layout.js and app/about/page.js while the titles are different.
Inheriting fields

app/layout.js

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

app/about/page.js

```
export const metadata = {
  title: 'About',
}

// Output:
// <title>About</title>
```

// <meta property="og:title" content="Acme" />
// <meta property="og:description" content="Acme is a..." />
Notes

title from app/layout.js is replaced by title in app/about/page.js.
All openGraph fields from app/layout.js are inherited in app/about/page.js because app/about/page.js doesn't set openGraph metadata.

Dynamic Image Generation

The ImageResponse constructor allows you to generate dynamic images using JSX and CSS. This is useful for creating social media images such as Open Graph images, Twitter cards, and more.
ImageResponse uses the Edge Runtime, and Next.js automatically adds the correct headers to cached images at the edge, helping improve performance and reducing recomputation.
To use it, you can import ImageResponse from next/server:
app/about/route.js import { ImageResponse } from 'next/server'

```
export const runtime = 'edge'

export async function GET() {
  return new ImageResponse(
    (
      <div
        style={{
          fontSize: 128,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          textAlign: 'center',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        Hello world!
      </div>
    ),
    {
      width: 1200,
      height: 600,
```

```
    }
  )
}
```

ImageResponse integrates well with other Next.js APIs, including Route Handlers and file-based Metadata. For example, you can use ImageResponse in a opengraph-image.tsx file to generate Open Graph images at build time or dynamically at request time.

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

Good to know:

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

JSON-LD

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

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

app/products/[id]/page.tsxTypeScript
```
export default async function Page({ params }) {
  const product = await getProduct(params.id)

  const jsonLd = {
    '@context': 'https://schema.org',
    '@type': 'Product',
    name: product.name,
    image: product.image,
    description: product.description,
```

```
  }

  return (
    <section>
      {/* Add JSON-LD to your page */}
      <script
        type="application/ld+json"
        dangerouslySetInnerHTML={{ __html: JSON.stringify(jsonLd) }}
      />
      {/* ... */}
    </section>
  )
}
```

You can validate and test your structured data with the Rich Results Test for Google or the generic Schema Markup Validator.
You can type your JSON-LD with TypeScript using community packages like schema-dts:

```
 import { Product, WithContext } from 'schema-dts'

const jsonLd: WithContext<Product> = {
  '@context': 'https://schema.org',
  '@type': 'Product',
  name: 'Next.js Sticker',
  image: 'https://nextjs.org/imgs/sticker.png',
  description: 'Dynamic at the speed of static.',
}
```

Static Assets
Next.js can serve static files, like images, under a folder called public in the root directory. Files inside public can then be referenced by your code starting from the base URL (/).
For example, if you add me.png inside public, the following code will access the image:
Avatar.js import Image from 'next/image'

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

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

Good to know:

The directory must be named public. The name cannot be changed and it's the only directory used to serve static assets.
Only assets that are in the public directory at build time will be served by Next.js. Files

added at runtime won't be available. We recommend using a third-party service like AWS S3 for persistent file storage.

Lazy Loading
Lazy loading in Next.js helps improve the initial loading performance of an application by decreasing the amount of JavaScript needed to render a route.
It allows you to defer loading of Client Components and imported libraries, and only include them in the client bundle when they're needed. For example, you might want to defer loading a modal until a user clicks to open it.
There are two ways you can implement lazy loading in Next.js:

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

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

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

Importing Client Components

app/page.js 'use client'

```
import { useState } from 'react'
import dynamic from 'next/dynamic'

// Client Components:
const ComponentA = dynamic(() => import('../components/A'))
const ComponentB = dynamic(() => import('../components/B'))
const ComponentC = dynamic(() => import('../components/C'), { ssr: false })
```

```
export default function ClientComponentExample() {
  const [showMore, setShowMore] = useState(false)

  return (
    <div>
      {/* Load immediately, but in a separate client bundle */}
      <ComponentA />

      {/* Load on demand, only when/if the condition is met */}
      {showMore && <ComponentB />}
      <button onClick={() => setShowMore(!showMore)}>Toggle</button>

      {/* Load only on the client side */}
      <ComponentC />
    </div>
  )
}Skipping SSR
```

When using React.lazy() and Suspense, Client Components will be pre-rendered (SSR) by default.If you want to disable pre-rendering for a Client Component, you can use the ssr option set to false: const ComponentC = dynamic(() => import('../components/C'), { ssr: false })Importing Server Components

If you dynamically import a Server Component, only the Client Components that are children of the Server Component will be lazy-loaded - not the Server Component itself.app/page.js import dynamic from 'next/dynamic'

```
// Server Component:
const ServerComponent = dynamic(() => import('../components/ServerComponent'))

export default function ServerComponentExample() {
  return (
    <div>
      <ServerComponent />
    </div>
  )
}Loading External Libraries
```

External libraries can be loaded on demand using the import() function. This example uses the external library fuse.js for fuzzy search. The module is only loaded on the client after the user types in the search input.app/page.js 'use client'

```
import { useState } from 'react'

const names = ['Tim', 'Joe', 'Bel', 'Lee']

export default function Page() {
  const [results, setResults] = useState()

  return (
    <div>
      <input
        type="text"
        placeholder="Search"
        onChange={async (e) => {
          const { value } = e.currentTarget
          // Dynamically load fuse.js
          const Fuse = (await import('fuse.js')).default
          const fuse = new Fuse(names)

          setResults(fuse.search(value))
        }}
      />
      <pre>Results: {JSON.stringify(results, null, 2)}</pre>
    </div>
  )
}Adding a custom loading component
```

```
app/page.js import dynamic from 'next/dynamic'

const WithCustomLoading = dynamic(
  () => import('../components/WithCustomLoading'),
  {
    loading: () => <p>Loading...</p>,
  }
)

export default function Page() {
  return (
    <div>
```

```
      {/* The loading component will be rendered while  <WithCustomLoading/> is
loading */}
      <WithCustomLoading />
    </div>
  )
}
```
Importing Named Exports

To dynamically import a named export, you can return it from the Promise returned by import() function:components/hello.js 'use client'

```
export function Hello() {
  return <p>Hello!</p>
}
```
app/page.js import dynamic from 'next/dynamic'

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

Next.js Speed Insights allows you to analyze and measure the performance of pages using different metrics.

You can start collecting your Real Experience Score with zero-configuration on Vercel deployments.

The rest of this documentation describes the built-in relayer Next.js Speed Insights uses.

Web Vitals

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

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

OpenTelemetry

Good to know: This feature is experimental, you need to explicitly opt-in by providing experimental.instrumentationHook = true; in your next.config.js.

Observability is crucial for understanding and optimizing the behavior and performance of your Next.js app.
As applications become more complex, it becomes increasingly difficult to identify and diagnose issues that may arise. By leveraging observability tools, such as logging and metrics, developers can gain insights into their application's behavior and identify areas for optimization. With observability, developers can proactively address issues before they become major problems and provide a better user experience. Therefore, it is highly recommended to use observability in your Next.js applications to improve performance, optimize resources, and enhance user experience.
We recommend using OpenTelemetry for instrumenting your apps.
It's a platform-agnostic way to instrument apps that allows you to change your observability provider without changing your code.
Read Official OpenTelemetry docs for more information about OpenTelemetry and how it works.
This documentation uses terms like Span, Trace or Exporter throughout this doc, all of which can be found in the OpenTelemetry Observability Primer.
Next.js supports OpenTelemetry instrumentation out of the box, which means that we already instrumented Next.js itself.
When you enable OpenTelemetry we will automatically wrap all your code like getStaticProps in spans with helpful attributes.

Good to know: We currently support OpenTelemetry bindings only in serverless functions.
We don't provide any for edge or client side code.

Getting Started

OpenTelemetry is extensible but setting it up properly can be quite verbose.
That's why we prepared a package @vercel/otel that helps you get started quickly.
It's not extensible and you should configure OpenTelemetry manually if you need to customize your setup.
Using @vercel/otel

To get started, you must install @vercel/otel:
Terminal npm install @vercel/otel

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

your-project/instrumentation.tsTypeScript import { registerOTel } from '@vercel/otel'

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

Good to know

The instrumentation file should be in the root of your project and not inside the app or pages directory. If you're using the src folder, then place the file inside src alongside pages and app.
If you use the pageExtensions config option to add a suffix, you will also need to update the instrumentation filename to match.
We have created a basic with-opentelemetry example that you can use.

Manual OpenTelemetry configuration

If our wrapper @vercel/otel doesn't suit your needs, you can configure OpenTelemetry manually.
Firstly you need to install OpenTelemetry packages:
Terminal npm install @opentelemetry/sdk-node @opentelemetry/resources @opentelemetry/semantic-conventions @opentelemetry/sdk-trace-base @opentelemetry/exporter-trace-otlp-http
Now you can initialize NodeSDK in your instrumentation.ts.
OpenTelemetry APIs are not compatible with edge runtime, so you need to make sure that you are importing them only when process.env.NEXT_RUNTIME === 'nodejs'. We recommend creating a new file instrumentation.node.ts which you conditionally import only when using node:
instrumentation.tsTypeScript export async function register() {

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

instrumentation.node.tsTypeScript import { trace, context } from '@opentelemetry/api' import { NodeSDK } from '@opentelemetry/sdk-node'

```
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-http'
import { Resource } from '@opentelemetry/resources'
import { SemanticResourceAttributes } from '@opentelemetry/semantic-conventions'
import { SimpleSpanProcessor } from '@opentelemetry/sdk-trace-node'

const sdk = new NodeSDK({
  resource: new Resource({
    [SemanticResourceAttributes.SERVICE_NAME]: 'next-app',
  }),
  spanProcessor: new SimpleSpanProcessor(new OTLPTraceExporter()),
})
sdk.start()
```

Doing this is equivalent to using @vercel/otel, but it's possible to modify and extend. For example, you could use @opentelemetry/exporter-trace-otlp-grpc instead of @opentelemetry/exporter-trace-otlp-http or you can specify more resource attributes.
Testing your instrumentation

You need an OpenTelemetry collector with a compatible backend to test OpenTelemetry traces locally.
We recommend using our OpenTelemetry dev environment.
If everything works well you should be able to see the root server span labeled as GET /requested/pathname.
All other spans from that particular trace will be nested under it.
Next.js traces more spans than are emitted by default.
To see more spans, you must set NEXT_OTEL_VERBOSE=1.
Deployment

Using OpenTelemetry Collector

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

We made sure that OpenTelemetry works out of the box on Vercel.
Follow Vercel documentation to connect your project to an observability provider.
Self-hosting

Deploying to other platforms is also straightforward. You will need to spin up your own OpenTelemetry Collector to receive and process the telemetry data from your Next.js app.
To do this, follow the OpenTelemetry Collector Getting Started guide, which will walk you through setting up the collector and configuring it to receive data from your Next.js app.
Once you have your collector up and running, you can deploy your Next.js app to your chosen platform following their respective deployment guides.
Custom Exporters

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

You can add a custom span with OpenTelemetry APIs.
Terminal npm install @opentelemetry/api
The following example demonstrates a function that fetches GitHub stars and adds a custom fetchGithubStars span to track the fetch request's result:

```
 import { trace } from '@opentelemetry/api'

export async function fetchGithubStars() {
  return await trace
    .getTracer('nextjs-example')
    .startActiveSpan('fetchGithubStars', async (span) => {
      try {
```

```
      return await getValue()
    } finally {
      span.end()
    }
  })
}
```

The register function will execute before your code runs in a new environment.
You can start creating new spans, and they should be correctly added to the exported trace.
Default Spans in Next.js

Next.js automatically instruments several spans for you to provide useful insights into your application's performance.
Attributes on spans follow OpenTelemetry semantic conventions. We also add some custom attributes under the next namespace:

next.span_name - duplicates span name
next.span_type - each span type has a unique identifier
next.route - The route pattern of the request (e.g., /[param]/user).
next.page

This is an internal value used by an app router.
You can think about it as a route to a special file (like page.ts, layout.ts, loading.ts and others)
It can be used as a unique identifier only when paired with next.route because /layout can be used to identify both /(groupA)/layout.ts and /(groupB)/layout.ts

[http.method] [next.route]

next.span_type: BaseServer.handleRequest

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

Common HTTP attributes

http.method
http.status_code

Server HTTP attributes

http.route
http.target

next.span_name
next.span_type
next.route

render route (app) [next.route]

next.span_type: AppRender.getBodyResult.

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

next.span_name
next.span_type
next.route

fetch [http.method] [http.url]

next.span_type: AppRender.fetch

This span represents the fetch request executed in your code.
Attributes:

Common HTTP attributes

http.method

Client HTTP attributes

http.url
net.peer.name
net.peer.port (only if specified)

next.span_name
next.span_type

executing api route (app) [next.route]

next.span_type: AppRouteRouteHandlers.runHandler.

This span represents the execution of an API route handler in the app router.
Attributes:

next.span_name
next.span_type
next.route

getServerSideProps [next.route]

next.span_type: Render.getServerSideProps.

This span represents the execution of getServerSideProps for a specific route.
Attributes:

next.span_name
next.span_type
next.route

getStaticProps [next.route]

next.span_type: Render.getStaticProps.

This span represents the execution of getStaticProps for a specific route.
Attributes:

next.span_name
next.span_type
next.route

render route (pages) [next.route]

next.span_type: Render.renderDocument.

This span represents the process of rendering the document for a specific route.
Attributes:

next.span_name
next.span_type
next.route

generateMetadata [next.page]

next.span_type: ResolveMetadata.generateMetadata.

This span represents the process of generating metadata for a specific page (a single route can have multiple of these spans).
Attributes:

next.span_name
next.span_type

next.page
Instrumentation
If you export a function named register from a instrumentation.ts (or .js) file in the root directory of your project (or inside the src folder if using one), we will call that function whenever a new Next.js server instance is bootstrapped.

Good to know

This feature is experimental. To use it, you must explicitly opt in by defining experimental.instrumentationHook = true; in your next.config.js.
The instrumentation file should be in the root of your project and not inside the app or pages directory. If you're using the src folder, then place the file inside src alongside pages and app.
If you use the pageExtensions config option to add a suffix, you will also need to update the instrumentation filename to match.
We have created a basic with-opentelemetry example that you can use.

When your register function is deployed, it will be called on each cold boot (but exactly once in each environment).
Sometimes, it may be useful to import a file in your code because of the side effects it will cause. For example, you might import a file that defines a set of global variables, but never explicitly use the imported file in your code. You would still have access to the global variables the package has declared.
You can import files with side effects in instrumentation.ts, which you might want to use in your register function as demonstrated in the following example:
your-project/instrumentation.tsTypeScript import { init } from 'package-init'

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

However, we recommend importing files with side effects using import from within your register function instead. The following example demonstrates a basic usage of import in a register function:
your-project/instrumentation.tsTypeScript export async function register() {
  await import('package-with-side-effect')
}

By doing this, you can colocate all of your side effects in one place in your code, and avoid any unintended consequences from importing files.
We call register in all environments, so it's necessary to conditionally import any code that doesn't support both edge and nodejs. You can use the environment variable NEXT_RUNTIME to get the current environment. Importing an environment-specific code would look like this:

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

  if (process.env.NEXT_RUNTIME === 'edge') {
    await import('./instrumentation-edge')
  }
}
Configuring
Next.js allows you to customize your project to meet specific requirements. This includes integrations with TypeScript, ESlint, and more, as well as internal configuration options such as Absolute Imports and Environment Variables.
TypeScript
Next.js provides a TypeScript-first development experience for building your React application.
It comes with built-in TypeScript support for automatically installing the necessary packages and configuring the proper settings.
As well as a TypeScript Plugin for your editor.
Ø<ß¥ Watch: Learn about the built-in TypeScript plugin !’ YouTube (3 minutes)

New Projects

create-next-app now ships with TypeScript by default.
Terminal npx create-next-app@latest
Existing Projects

Add TypeScript to your project by renaming a file to .ts / .tsx. Run next dev and next build to automatically install the necessary dependencies and add a tsconfig.json file with the recommended config options.
TypeScript Plugin

Next.js includes a custom TypeScript plugin and type checker, which VSCode and other code editors can use for advanced type-checking and auto-completion.You can enable the plugin in VS Code by:

Opening the command palette (Ctrl/# + Shift + P)
Searching for "TypeScript: Select TypeScript Version"
Selecting "Use Workspace Version"
Now, when editing files, the custom plugin will be enabled. When running next build, the custom type checker will be used.Plugin Features

The TypeScript plugin can help with:
Warning if the invalid values for segment config options are passed.
Showing available options and in-context documentation.
Ensuring the use client directive is used correctly.
Ensuring client hooks (like useState) are only used in Client Components.

Good to know: More features will be added in the future.

Minimum TypeScript Version

It is highly recommended to be on at least v4.5.2 of TypeScript to get syntax features such as type modifiers on import names and performance improvements.
Statically Typed Links

Next.js can statically type links to prevent typos and other errors when using next/link, improving type safety when navigating between pages.To opt-into this feature, experimental.typedRoutes need to be enabled and the project needs to be using TypeScript.next.config.js /** @type {import('next').NextConfig} */
```
const nextConfig = {
  experimental: {
    typedRoutes: true,
  },
}
```

module.exports = nextConfigNext.js will generate a link definition in .next/types that contains information about all existing routes in your application, which TypeScript can then use to provide feedback in your editor about invalid links.Currently, experimental support includes any string literal, including dynamic segments. For non-literal strings, you currently need to manually cast the href with as Route: import type { Route } from 'next';

```
import Link from 'next/link'

// No TypeScript errors if href is a valid route
<Link href="/about" />
<Link href="/blog/nextjs" />
<Link href={`/blog/${slug}`} />
<Link href={('/blog' + slug) as Route} />

// TypeScript errors if href is not a valid route
<Link href="/aboot" />
```
To accept href in a custom component wrapping next/link, use a generic:
```
import type { Route } from 'next'
import Link from 'next/link'

function Card<T extends string>({ href }: { href: Route<T> | URL }) {
  return (
    <Link href={href}>
      <div>My Card</div>
    </Link>
  )
}
```
How does it work?

When running next dev or next build, Next.js generates a hidden .d.ts file inside .next that contains information about all existing routes in your application (all valid routes as the href type of Link). This .d.ts file is included in tsconfig.json and the TypeScript compiler will check that .d.ts and provide feedback in your editor about invalid links.

End-to-End Type Safety

Next.js 13 has enhanced type safety. This includes:

No serialization of data between fetching function and page: You can fetch directly in components, layouts, and pages on the server. This data does not need to be serialized (converted to a string) to be passed to the client side for consumption in React. Instead, since app uses Server Components by default, we can use values like Date, Map, Set, and more without any extra steps. Previously, you needed to manually type the boundary between server and client with Next.js-specific types.

Streamlined data flow between components: With the removal of _app in favor of root layouts, it is now easier to visualize the data flow between components and pages. Previously, data flowing between individual pages and _app were difficult to type and could introduce confusing bugs. With colocated data fetching in Next.js 13, this is no longer an issue.

Data Fetching in Next.js now provides as close to end-to-end type safety as possible without being prescriptive about your database or content provider selection.We're able to type the response data as you would expect with normal TypeScript. For example:app/page.tsx async function getData() {

```
  const res = await fetch('https://api.example.com/...')
  // The return value is *not* serialized
  // You can return Date, Map, Set, etc.
  return res.json()
}

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

  return '...'
}
```For complete end-to-end type safety, this also requires your database or content provider to support TypeScript. This could be through using an ORM or type-safe query builder.Async Server Component TypeScript Error

To use an async Server Component with TypeScript, ensure you are using TypeScript 5.1.3 or higher and @types/react 18.2.8 or higher.If you are using an older version of TypeScript, you may see a 'Promise<Element>' is not a valid JSX element type error. Updating to the latest version of TypeScript and @types/react should resolve this issue.Passing Data Between Server & Client Components

When passing data between a Server and Client Component through props, the data is still serialized (converted to a string) for use in the browser. However, it does not need a special type. It’s typed the same as passing any other props between components.Further, there is less code to be serialized, as un-rendered data does not cross between the server and client (it remains on the server). This is only now possible through support for Server Components.

Path aliases and baseUrl

Next.js automatically supports the tsconfig.json "paths" and "baseUrl" options.
You can learn more about this feature on the Module Path aliases documentation.

Type checking next.config.js

The next.config.js file must be a JavaScript file as it does not get parsed by Babel or TypeScript, however you can add some type checking in your IDE using JSDoc as below:

```
 // @ts-check

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

module.exports = nextConfig
```

Incremental type checking

Since v10.2.1 Next.js supports incremental type checking when enabled in your tsconfig.json, this can help speed up type checking in larger applications.

Ignoring TypeScript Errors

Next.js fails your production build (next build) when TypeScript errors are present in your project.

If you'd like Next.js to dangerously produce production code even when your application has errors, you can disable the built-in type checking step.

If disabled, be sure you are running type checks as part of your build or deploy process, otherwise this can be very dangerous.

Open next.config.js and enable the ignoreBuildErrors option in the typescript config:

next.config.js module.exports = {

```
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```

Version Changes

VersionChangesv13.2.0Statically typed links are available in beta.v12.0.0SWC is now used by default to compile TypeScript and TSX for faster builds.v10.2.1Incremental type checking support added when enabled in your tsconfig.json.

ESLint

Next.js provides an integrated ESLint experience out of the box. Add next lint as a script to package.json:

package.json {
  "scripts": {
    "lint": "next lint"
  }
}

Then run npm run lint or yarn lint:

Terminal yarn lint

If you don't already have ESLint configured in your application, you will be guided through the installation and configuration process.

Terminal yarn lint

You'll see a prompt like this:

? How would you like to configure ESLint?
'o Strict (recommended)
Base
Cancel

One of the following three options can be selected:

Strict: Includes Next.js' base ESLint configuration along with a stricter Core Web Vitals rule-set. This is the recommended configuration for developers setting up ESLint for the first time.

.eslintrc.json {
  "extends": "next/core-web-vitals"
}

Base: Includes Next.js' base ESLint configuration.

.eslintrc.json {
  "extends": "next"
}

Cancel: Does not include any ESLint configuration. Only select this option if you plan on

setting up your own custom ESLint configuration.

If either of the two configuration options are selected, Next.js will automatically install eslint and eslint-config-next as development dependencies in your application and create an .eslintrc.json file in the root of your project that includes your selected configuration.
You can now run next lint every time you want to run ESLint to catch errors. Once ESLint has been set up, it will also automatically run during every build (next build). Errors will fail the build, while warnings will not.

If you do not want ESLint to run during next build, refer to the documentation for Ignoring ESLint.

We recommend using an appropriate integration to view warnings and errors directly in your code editor during development.
ESLint Config

The default configuration (eslint-config-next) includes everything you need to have an optimal out-of-the-box linting experience in Next.js. If you do not have ESLint already configured in your application, we recommend using next lint to set up ESLint along with this configuration.

If you would like to use eslint-config-next along with other ESLint configurations, refer to the Additional Configurations section to learn how to do so without causing any conflicts.

Recommended rule-sets from the following ESLint plugins are all used within eslint-config-next:

eslint-plugin-react
eslint-plugin-react-hooks
eslint-plugin-next

This will take precedence over the configuration from next.config.js.
ESLint Plugin

Next.js provides an ESLint plugin, eslint-plugin-next, already bundled within the base

configuration that makes it possible to catch common issues and problems in a Next.js application. The full set of rules is as follows:
Enabled in the recommended configuration
RuleDescription@next/next/google-font-displayEnforce font-display behavior with Google Fonts.@next/next/google-font-preconnectEnsure preconnect is used with Google Fonts.@next/next/inline-script-idEnforce id attribute on next/script components with inline content.@next/next/next-script-for-gaPrefer next/script component when using the inline script for Google Analytics.@next/next/no-assign-module-variablePrevent assignment to the module variable.@next/next/no-async-client-componentPrevent client components from being async functions.@next/next/no-before-interactive-script-outside-documentPrevent usage of next/script's beforeInteractive strategy outside of pages/_document.js.@next/next/no-css-tagsPrevent manual stylesheet tags.@next/next/no-document-import-in-pagePrevent importing next/document outside of pages/_document.js.@next/next/no-duplicate-headPrevent duplicate usage of <Head> in pages/_document.js.@next/next/no-head-elementPrevent usage of <head> element.@next/next/no-head-import-in-documentPrevent usage of next/head in pages/_document.js.@next/next/no-html-link-for-pagesPrevent usage of <a> elements to navigate to internal Next.js pages.@next/next/no-img-elementPrevent usage of <img> element due to slower LCP and higher bandwidth.@next/next/no-page-custom-fontPrevent page-only custom fonts.@next/next/no-script-component-in-headPrevent usage of next/script in next/head component.@next/next/no-styled-jsx-in-documentPrevent usage of styled-jsx in pages/_document.js.@next/next/no-sync-scriptsPrevent synchronous scripts.@next/next/no-title-in-document-headPrevent usage of <title> with Head component from next/document.@next/next/no-typosPrevent common typos in Next.js's data fetching functions@next/next/no-unwanted-polyfillioPrevent duplicate polyfills from Polyfill.io.
If you already have ESLint configured in your application, we recommend extending from this plugin directly instead of including eslint-config-next unless a few conditions are met. Refer to the Recommended Plugin Ruleset to learn more.
Custom Settings

rootDir

If you're using eslint-plugin-next in a project where Next.js isn't installed in your root directory (such as a monorepo), you can tell eslint-plugin-next where to find your Next.js application using the settings property in your .eslintrc:
.eslintrc.json {
  "extends": "next",

```
  "settings": {
    "next": {
      "rootDir": "packages/my-app/"
    }
  }
}
```

rootDir can be a path (relative or absolute), a glob (i.e. "packages/*/"), or an array of paths and/or globs.

Linting Custom Directories and Files

By default, Next.js will run ESLint for all files in the pages/, app/, components/, lib/, and src/ directories. However, you can specify which directories using the dirs option in the eslint config in next.config.js for production builds:

```
next.config.js module.exports = {
  eslint: {
    dirs: ['pages', 'utils'], // Only run ESLint on the 'pages' and 'utils' directories during production builds (next build)
  },
}
```

Similarly, the --dir and --file flags can be used for next lint to lint specific directories and files:

```
Terminal next lint --dir pages --dir utils --file bar.js
```

Caching

To improve performance, information of files processed by ESLint are cached by default. This is stored in .next/cache or in your defined build directory. If you include any ESLint rules that depend on more than the contents of a single source file and need to disable the cache, use the --no-cache flag with next lint.

```
Terminal next lint --no-cache
```

Disabling Rules

If you would like to modify or disable any rules provided by the supported plugins (react, react-hooks, next), you can directly change them using the rules property in your .eslintrc:

.eslintrc.json {
  "extends": "next",
  "rules": {
    "react/no-unescaped-entities": "off",
    "@next/next/no-page-custom-font": "off"
  }
}
Core Web Vitals

The next/core-web-vitals rule set is enabled when next lint is run for the first time and the strict option is selected.
.eslintrc.json {
  "extends": "next/core-web-vitals"
}
next/core-web-vitals updates eslint-plugin-next to error on a number of rules that are warnings by default if they affect Core Web Vitals.

The next/core-web-vitals entry point is automatically included for new applications built with Create Next App.

Usage With Other Tools

Prettier

ESLint also contains code formatting rules, which can conflict with your existing Prettier setup. We recommend including eslint-config-prettier in your ESLint config to make ESLint and Prettier work together.
First, install the dependency:
Terminal npm install --save-dev eslint-config-prettier

yarn add --dev eslint-config-prettier
Then, add prettier to your existing ESLint config:
.eslintrc.json {
  "extends": ["next", "prettier"]

}
lint-staged

If you would like to use next lint with lint-staged to run the linter on staged git files, you'll have to add the following to the .lintstagedrc.js file in the root of your project in order to specify usage of the --file flag.
.lintstagedrc.js const path = require('path')

```
const buildEslintCommand = (filenames) =>
  `next lint --fix --file ${filenames
    .map((f) => path.relative(process.cwd(), f))
    .join(' --file ')}`

module.exports = {
  '*.{js,jsx,ts,tsx}': [buildEslintCommand],
}
```

Migrating Existing Config

Recommended Plugin Ruleset

If you already have ESLint configured in your application and any of the following conditions are true:

You have one or more of the following plugins already installed (either separately or through a different config such as airbnb or react-app):

react
react-hooks
jsx-a11y
import

You've defined specific parserOptions that are different from how Babel is configured within Next.js (this is not recommended unless you have customized your Babel configuration)

You have eslint-plugin-import installed with Node.js and/or TypeScript resolvers defined to handle imports

Then we recommend either removing these settings if you prefer how these properties have been configured within eslint-config-next or extending directly from the Next.js ESLint plugin instead:

```
 module.exports = {
  extends: [
    //...
    'plugin:@next/next/recommended',
  ],
}
```

The plugin can be installed normally in your project without needing to run next lint:
Terminal npm install --save-dev @next/eslint-plugin-next

yarn add --dev @next/eslint-plugin-next
This eliminates the risk of collisions or errors that can occur due to importing the same plugin or parser across multiple configurations.
Additional Configurations

If you already use a separate ESLint configuration and want to include eslint-config-next, ensure that it is extended last after other configurations. For example:

```
.eslintrc.json {
  "extends": ["eslint:recommended", "next"]
}
```

The next configuration already handles setting default values for the parser, plugins and settings properties. There is no need to manually re-declare any of these properties unless you need a different configuration for your use case.
If you include any other shareable configurations, you will need to make sure that these properties are not overwritten or modified. Otherwise, we recommend removing any configurations that share behavior with the next configuration or extending directly from the Next.js ESLint plugin as mentioned above.
Environment Variables
Examples
Environment Variables

Next.js comes with built-in support for environment variables, which allows you to do the following:

Use .env.local to load environment variables
Bundle environment variables for the browser by prefixing with NEXT_PUBLIC_

Loading Environment Variables

Next.js has built-in support for loading environment variables from .env.local into process.env.
.env.local DB_HOST=localhost
DB_USER=myuser
DB_PASS=mypassword

This loads process.env.DB_HOST, process.env.DB_USER, and process.env.DB_PASS into the Node.js environment automatically allowing you to use them in Route Handlers.For example:app/api/route.js export async function GET() {

```
  const db = await myDB.connect({
    host: process.env.DB_HOST,
    username: process.env.DB_USER,
    password: process.env.DB_PASS,
  })
  // ...
}
```

Referencing Other Variables

Next.js will automatically expand variables that use $ to reference other variables e.g. $VARIABLE inside of your .env* files. This allows you to reference other secrets. For example:
.env TWITTER_USER=nextjs
TWITTER_URL=https://twitter.com/$TWITTER_USER
In the above example, process.env.TWITTER_URL would be set to https://twitter.com/nextjs.

Good to know: If you need to use variable with a $ in the actual value, it needs to be escaped e.g. \$.

Bundling Environment Variables for the Browser

Non-NEXT_PUBLIC_ environment variables are only available in the Node.js

environment, meaning they aren't accessible to the browser (the client runs in a different environment).
In order to make the value of an environment variable accessible in the browser, Next.js can "inline" a value, at build time, into the js bundle that is delivered to the client, replacing all references to process.env.[variable] with a hard-coded value. To tell it to do this, you just have to prefix the variable with NEXT_PUBLIC_. For example:
Terminal NEXT_PUBLIC_ANALYTICS_ID=abcdefghijk
This will tell Next.js to replace all references to process.env.NEXT_PUBLIC_ANALYTICS_ID in the Node.js environment with the value from the environment in which you run next build, allowing you to use it anywhere in your code. It will be inlined into any JavaScript sent to the browser.

Note: After being built, your app will no longer respond to changes to these environment variables. For instance, if you use a Heroku pipeline to promote slugs built in one environment to another environment, or if you build and deploy a single Docker image to multiple environments, all NEXT_PUBLIC_ variables will be frozen with the value evaluated at build time, so these values need to be set appropriately when the project is built. If you need access to runtime environment values, you'll have to setup your own API to provide them to the client (either on demand or during initialization).

pages/index.js import setupAnalyticsService from '../lib/my-analytics-service'

// 'NEXT_PUBLIC_ANALYTICS_ID' can be used here as it's prefixed by 'NEXT_PUBLIC_'.
// It will be transformed at build time to `setupAnalyticsService('abcdefghijk')`.
setupAnalyticsService(process.env.NEXT_PUBLIC_ANALYTICS_ID)

function HomePage() {
  return <h1>Hello World</h1>
}

export default HomePage
Note that dynamic lookups will not be inlined, such as:
 // This will NOT be inlined, because it uses a variable
const varName = 'NEXT_PUBLIC_ANALYTICS_ID'
setupAnalyticsService(process.env[varName])

// This will NOT be inlined, because it uses a variable
const env = process.env
setupAnalyticsService(env.NEXT_PUBLIC_ANALYTICS_ID)
Default Environment Variables

In general only one .env.local file is needed. However, sometimes you might want to add some defaults for the development (next dev) or production (next start) environment.
Next.js allows you to set defaults in .env (all environments), .env.development (development environment), and .env.production (production environment).
.env.local always overrides the defaults set.

Good to know: .env, .env.development, and .env.production files should be included in your repository as they define defaults. .env*.local should be added to .gitignore, as those files are intended to be ignored. .env.local is where secrets can be stored.

Environment Variables on Vercel

When deploying your Next.js application to Vercel, Environment Variables can be configured in the Project Settings.
All types of Environment Variables should be configured there. Even Environment Variables used in Development – which can be downloaded onto your local device afterwards.
If you've configured Development Environment Variables you can pull them into a .env.local for usage on your local machine using the following command:
Terminal vercel env pull .env.local
Test Environment Variables

Apart from development and production environments, there is a 3rd option available: test. In the same way you can set defaults for development or production environments, you can do the same with a .env.test file for the testing environment (though this one is not as common as the previous two). Next.js will not load environment variables from .env.development or .env.production in the testing environment.
This one is useful when running tests with tools like jest or cypress where you need to set specific environment vars only for testing purposes. Test default values will be loaded if NODE_ENV is set to test, though you usually don't need to do this manually as testing tools will address it for you.
There is a small difference between test environment, and both development and production that you need to bear in mind: .env.local won't be loaded, as you expect tests to produce the same results for everyone. This way every test execution will use the same env defaults across different executions by ignoring your .env.local (which is intended to override the default set).

Good to know: similar to Default Environment Variables, .env.test file should be included in your repository, but .env.test.local shouldn't, as .env*.local are intended to be ignored through .gitignore.

While running unit tests you can make sure to load your environment variables the same way Next.js does by leveraging the loadEnvConfig function from the @next/env package.

```
 // The below can be used in a Jest global setup file or similar for your testing set-up
import { loadEnvConfig } from '@next/env'

export default async () => {
  const projectDir = process.cwd()
  loadEnvConfig(projectDir)
}
```

Environment Variable Load Order

Environment variables are looked up in the following places, in order, stopping once the variable is found.

process.env
.env.$(NODE_ENV).local
.env.local (Not checked when NODE_ENV is test.)
.env.$(NODE_ENV)
.env

For example, if NODE_ENV is development and you define a variable in both .env.development.local and .env, the value in .env.development.local will be used.

Good to know: The allowed values for NODE_ENV are production, development and test.

Good to know

If you are using a /src directory, .env.* files should remain in the root of your project.
If the environment variable NODE_ENV is unassigned, Next.js automatically assigns development when running the next dev command, or production for all other commands.

Absolute Imports and Module Path Aliases
Examples
Absolute Imports and Aliases

Next.js has in-built support for the "paths" and "baseUrl" options of tsconfig.json and jsconfig.json files.
These options allow you to alias project directories to absolute paths, making it easier to import modules. For example:

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

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

Good to know: create-next-app will prompt to configure these options for you.

Absolute Imports

The baseUrl configuration option allows you to import directly from the root of the project.
An example of this configuration:
tsconfig.json or jsconfig.json {

```
  "compilerOptions": {
    "baseUrl": "."
  }
}
```

components/button.tsxTypeScript export default function Button() {

```
  return <button>Click me</button>
}
```

app/page.tsxTypeScript import Button from 'components/button'

```
export default function HomePage() {
  return (
    <>
      <h1>Hello World</h1>
      <Button />
    </>
  )
}
```

Module Aliases

In addition to configuring the baseUrl path, you can use the "paths" option to "alias" module paths.
For example, the following configuration maps @/components/* to components/*:
tsconfig.json or jsconfig.json {
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/components/*": ["components/*"]
    }
  }
}
components/button.tsxTypeScript export default function Button() {
  return <button>Click me</button>
}

app/page.tsxTypeScript import Button from '@/components/button'

export default function HomePage() {
  return (
    <>
      <h1>Hello World</h1>
      <Button />
    </>
)
}

Each of the "paths" are relative to the baseUrl location. For example:
 // tsconfig.json or jsconfig.json
{
  "compilerOptions": {
    "baseUrl": "src/",
    "paths": {
      "@/styles/*": ["styles/*"],
      "@/components/*": ["components/*"]
    }
  }
}
 // pages/index.js
import Button from '@/components/button'
import '@/styles/styles.css'
import Helper from 'utils/helper'

```
export default function HomePage() {
  return (
    <Helper>
      <h1>Hello World</h1>
      <Button />
    </Helper>
  )
}
```

MDX

Markdown is a lightweight markup language used to format text. It allows you to write using plain text syntax and convert it to structurally valid HTML. It's commonly used for writing content on websites and blogs.

You write...

```
 I **love** using [Next.js](https://nextjs.org/)
```

Output:

```
 <p>I <strong>love</strong> using <a href="https://nextjs.org/">Next.js</a></p>
```

MDX is a superset of markdown that lets you write JSX directly in your markdown files. It is a powerful way to add dynamic interactivity and embed React components within your content.

Next.js can support both local MDX content inside your application, as well as remote MDX files fetched dynamically on the server. The Next.js plugin handles transforming Markdown and React components into HTML, including support for usage in Server Components (default in app).

@next/mdx

The @next/mdx package is configured in the next.config.js file at your projects root. It sources data from local files, allowing you to create pages with a .mdx extension, directly in your /pages or /app directory.

Getting Started

Install the @next/mdx package:Terminal npm install @next/mdx @mdx-js/loader @mdx-js/react @types/mdxCreate mdx-components.tsx in the root of your application (the parent folder of app/ or src/):mdx-components.tsxTypeScript import type { MDXComponents } from 'mdx/types'

// This file allows you to provide custom React components
// to be used in MDX files. You can import and use any

```
// React component you want, including components from
// other libraries.

// This file is required to use MDX in `app` directory.
export function useMDXComponents(components: MDXComponents):
MDXComponents {
  return {
    // Allows customizing built-in components, e.g. to add styling.
    // h1: ({ children }) => <h1 style={{ fontSize: "100px" }}>{children}</h1>,
    ...components,
  }
}Update next.config.js to use mdxRs:next.config.js /** @type {import('next').NextConfig}
*/
const nextConfig = {
  experimental: {
    mdxRs: true,
  },
}

const withMDX = require('@next/mdx')()
module.exports = withMDX(nextConfig)Add a new file with MDX content to your app
directory:app/hello.mdx Hello, Next.js!

You can import and use React components in MDX files.Import the MDX file inside a
page to display the content:app/page.tsxTypeScript import HelloWorld from './hello.mdx'

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

Remote MDX

If your Markdown or MDX files do not live inside your application, you can fetch them dynamically on the server. This is useful for fetching content from a CMS or other data source.
There are two popular community packages for fetching MDX content: next-mdx-remote and contentlayer. For example, the following example uses next-mdx-remote:

Good to know: Please proceed with caution. MDX compiles to JavaScript and is executed on the server. You should only fetch MDX content from a trusted source, otherwise this can lead to remote code execution (RCE).

app/page.tsxTypeScript import { MDXRemote } from 'next-mdx-remote/rsc'

```
export default async function Home() {
  const res = await fetch('https://...')
  const markdown = await res.text()
  return <MDXRemote source={markdown} />
}
```

Layouts

To share a layout around MDX content, you can use the built-in layouts support with the App Router.

Remark and Rehype Plugins

You can optionally provide remark and rehype plugins to transform the MDX content. For example, you can use remark-gfm to support GitHub Flavored Markdown.
Since the remark and rehype ecosystem is ESM only, you'll need to use next.config.mjs as the configuration file.
next.config.js

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

const withMDX = require('@next/mdx')({
  options: {
    remarkPlugins: [],
    rehypePlugins: [],
    // If you use `MDXProvider`, uncomment the following line.
    // providerImportSource: "@mdx-js/react",
  },
})
module.exports = withMDX(nextConfig)
```

Frontmatter

Frontmatter is a YAML like key/value pairing that can be used to store data about a page. @next/mdx does not support frontmatter by default, though there are many solutions for adding frontmatter to your MDX content, such as gray-matter.
To access page metadata with @next/mdx, you can export a meta object from within the .mdx file:
```
 export const meta = {
  author: 'Rich Haines',
}

# My MDX page
```
Custom Elements

One of the pleasant aspects of using markdown, is that it maps to native HTML elements, making writing fast, and intuitive:
```
 This is a list in markdown:

- One
- Two
- Three
```
The above generates the following HTML:
```
 <p>This is a list in markdown:</p>

<ul>
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
</ul>
```
When you want to style your own elements to give a custom feel to your website or application, you can pass in shortcodes. These are your own custom components that map to HTML elements. To do this you use the MDXProvider and pass a components object as a prop. Each object key in the components object maps to a HTML element name.
To enable you need to specify providerImportSource: "@mdx-js/react" in next.config.js.
```
next.config.js const withMDX = require('@next/mdx')({
  // ...
  options: {
    providerImportSource: '@mdx-js/react',
  },
})
```

Then setup the provider in your page
pages/index.js import { MDXProvider } from '@mdx-js/react'
import Image from 'next/image'
import { Heading, InlineCode, Pre, Table, Text } from 'my-components'

```
const ResponsiveImage = (props) => (
  <Image
    alt={props.alt}
    sizes="100vw"
    style={{ width: '100%', height: 'auto' }}
    {...props}
  />
)

const components = {
  img: ResponsiveImage,
  h1: Heading.H1,
  h2: Heading.H2,
  p: Text,
  pre: Pre,
  code: InlineCode,
}

export default function Post(props) {
  return (
    <MDXProvider components={components}>
      <main {...props} />
    </MDXProvider>
  )
}
```

If you use it across the site you may want to add the provider to _app.js so all MDX pages pick up the custom element config.
Deep Dive: How do you transform markdown into HTML?

React does not natively understand Markdown. The markdown plaintext needs to first be transformed into HTML. This can be accomplished with remark and rehype. remark is an ecosystem of tools around markdown. rehype is the same, but for HTML. For example, the following code snippet transforms markdown into HTML:

```
 import { unified } from 'unified'
import remarkParse from 'remark-parse'
import remarkRehype from 'remark-rehype'
import rehypeSanitize from 'rehype-sanitize'
```

```
import rehypeStringify from 'rehype-stringify'

main()

async function main() {
  const file = await unified()
    .use(remarkParse) // Convert into markdown AST
    .use(remarkRehype) // Transform to HTML AST
    .use(rehypeSanitize) // Sanitize HTML input
    .use(rehypeStringify) // Convert AST into serialized HTML
    .process('Hello, Next.js!')

  console.log(String(file)) // <p>Hello, Next.js!</p>
}
```

The remark and rehype ecosystem contains plugins for syntax highlighting, linking headings, generating a table of contents, and more.

When using @next/mdx as shown below, you do not need to use remark or rehype directly, as it is handled for you.

Using the Rust-based MDX compiler (Experimental)

Next.js supports a new MDX compiler written in Rust. This compiler is still experimental and is not recommended for production use. To use the new compiler, you need to configure next.config.js when you pass it to withMDX:

```
next.config.js module.exports = withMDX({
  experimental: {
    mdxRs: true,
  },
})
```

Helpful Links

MDX

@next/mdx

remark

rehype

src Directory

As an alternative to having the special Next.js app or pages directories in the root of your project, Next.js also supports the common pattern of placing application code

under the src directory.
This separates application code from project configuration files which mostly live in the root of a project. Which is preferred by some individuals and teams.
To use the src directory, move the app Router folder or pages Router folder to src/app or src/pages respectively.

Good to know

The /public directory should remain in the root of your project.
Config files like package.json, next.config.js and tsconfig.json should remain in the root of your project.
.env.* files should remain in the root of your project.
src/app or src/pages will be ignored if app or pages are present in the root directory.
If you're using src, you'll probably also move other application folders such as /components or /lib.
If you're using Tailwind CSS, you'll need to add the /src prefix to the tailwind.config.js file in the content section.

Draft ModeStatic rendering is useful when your pages fetch data from a headless CMS. However, it’s not ideal when you’re writing a draft on your headless CMS and want to view the draft immediately on your page. You’d want Next.js to render these pages at request time instead of build time and fetch the draft content instead of the published content. You’d want Next.js to switch to dynamic rendering only for this specific case.
Next.js has a feature called Draft Mode which solves this problem. Here are instructions on how to use it.
Step 1: Create and access the Route Handler

First, create a Route Handler. It can have any name - e.g. app/api/draft/route.ts
Then, import draftMode from next/headers and call the enable() method.
app/api/draft/route.tsTypeScript // route handler enabling draft mode
```
import { draftMode } from 'next/headers'

export async function GET(request: Request) {
  draftMode().enable()
  return new Response('Draft mode is enabled')
}
```

This will set a cookie to enable draft mode. Subsequent requests containing this cookie will trigger Draft Mode changing the behavior for statically generated pages (more on this later).
You can test this manually by visiting /api/draft and looking at your browser’s developer

tools. Notice the Set-Cookie response header with a cookie named __prerender_bypass.
Securely accessing it from your Headless CMS

In practice, you'd want to call this Route Handler securely from your headless CMS. The specific steps will vary depending on which headless CMS you're using, but here are some common steps you could take.
These steps assume that the headless CMS you're using supports setting custom draft URLs. If it doesn't, you can still use this method to secure your draft URLs, but you'll need to construct and access the draft URL manually.
First, you should create a secret token string using a token generator of your choice. This secret will only be known by your Next.js app and your headless CMS. This secret prevents people who don't have access to your CMS from accessing draft URLs.
Second, if your headless CMS supports setting custom draft URLs, specify the following as the draft URL. This assumes that your Route Handler is located at app/api/draft/route.ts
Terminal https://<your-site>/api/draft?secret=<token>&slug=<path>

<your-site> should be your deployment domain.
<token> should be replaced with the secret token you generated.
<path> should be the path for the page that you want to view. If you want to view /posts/foo, then you should use &slug=/posts/foo.

Your headless CMS might allow you to include a variable in the draft URL so that <path> can be set dynamically based on the CMS's data like so: &slug=/posts/{entry.fields.slug}
Finally, in the Route Handler:

Check that the secret matches and that the slug parameter exists (if not, the request should fail).
Call draftMode.enable() to set the cookie.
Then redirect the browser to the path specified by slug.

app/api/draft/route.tsTypeScript // route handler with secret and slug
```
import { draftMode } from 'next/headers'
import { redirect } from 'next/navigation'

export async function GET(request: Request) {
  // Parse query string parameters
  const { searchParams } = new URL(request.url)
  const secret = searchParams.get('secret')
  const slug = searchParams.get('slug')
```

```
  // Check the secret and next parameters
  // This secret should only be known to this route handler and the CMS
  if (secret !== 'MY_SECRET_TOKEN' || !slug) {
    return new Response('Invalid token', { status: 401 })
  }

  // Fetch the headless CMS to check if the provided `slug` exists
  // getPostBySlug would implement the required fetching logic to the headless CMS
  const post = await getPostBySlug(slug)

  // If the slug doesn't exist prevent draft mode from being enabled
  if (!post) {
    return new Response('Invalid slug', { status: 401 })
  }

  // Enable Draft Mode by setting the cookie
  draftMode().enable()

  // Redirect to the path from the fetched post
  // We don't redirect to searchParams.slug as that might lead to open redirect
vulnerabilities
  redirect(post.slug)
}
```

If it succeeds, then the browser will be redirected to the path you want to view with the draft mode cookie.
Step 2: Update page

The next step is to update your page to check the value of draftMode().isEnabled.
If you request a page which has the cookie set, then data will be fetched at request time (instead of at build time).
Furthermore, the value of isEnabled will be true.
app/page.tsxTypeScript // page that fetches data

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

async function getData() {
  const { isEnabled } = draftMode()

  const url = isEnabled
    ? 'https://draft.example.com'
    : 'https://production.example.com'
```

```
  const res = await fetch(url)

  return res.json()
}

export default async function Page() {
  const { title, desc } = await getData()

  return (
    <main>
      <h1>{title}</h1>
      <p>{desc}</p>
    </main>
  )
}
```

That's it! If you access the draft Route Handler (with secret and slug) from your headless CMS or manually, you should now be able to see the draft content. And if you update your draft without publishing, you should be able to view the draft.
Set this as the draft URL on your headless CMS or access manually, and you should be able to see the draft.
Terminal https://<your-site>/api/draft?secret=<token>&slug=<path>
More Details

Clear the Draft Mode cookie

By default, the Draft Mode session ends when the browser is closed.
To clear the Draft Mode cookie manually, create a Route Handler that calls draftMode().disable():
app/api/disable-draft/route.tsTypeScript

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

export async function GET(request: Request) {
  draftMode().disable()
  return new Response('Draft mode is disabled')
}
```

Then, send a request to /api/disable-draft to invoke the Route Handler. If calling this

route using next/link, you must pass prefetch={false} to prevent accidentally deleting the cookie on prefetch.
Unique per next build

A new bypass cookie value will be generated each time you run next build.
This ensures that the bypass cookie can’t be guessed.

Good to know: To test Draft Mode locally over HTTP, your browser will need to allow third-party cookies and local storage access.
Deploying
Congratulations! You're here because you are ready to deploy your Next.js application. This page will show how to deploy either managed or self-hosted using the Next.js Build API.
Next.js Build API

next build generates an optimized version of your application for production. This standard output includes:

HTML files for pages using getStaticProps or Automatic Static Optimization
CSS files for global styles or for individually scoped styles
JavaScript for pre-rendering dynamic content from the Next.js server
JavaScript for interactivity on the client-side through React

This output is generated inside the .next folder:

.next/static/chunks/pages – Each JavaScript file inside this folder relates to the route with the same name. For example, .next/static/chunks/pages/about.js would be the JavaScript file loaded when viewing the /about route in your application
.next/static/media – Statically imported images from next/image are hashed and copied here
.next/static/css – Global CSS files for all pages in your application
.next/server/pages – The HTML and JavaScript entry points prerendered from the server. The .nft.json files are created when Output File Tracing is enabled and contain all the file paths that depend on a given page.
.next/server/chunks – Shared JavaScript chunks used in multiple places throughout your application
.next/cache – Output for the build cache and cached images, responses, and pages from the Next.js server. Using a cache helps decrease build times and improve

performance of loading images

All JavaScript code inside .next has been compiled and browser bundles have been minified to help achieve the best performance and support all modern browsers.
Managed Next.js with Vercel

Vercel is the fastest way to deploy your Next.js application with zero configuration. When deploying to Vercel, the platform automatically detects Next.js, runs next build, and optimizes the build output for you, including:

Persisting cached assets across deployments if unchanged
Immutable deployments with a unique URL for every commit
Pages are automatically statically optimized, if possible
Assets (JavaScript, CSS, images, fonts) are compressed and served from a Global Edge Network
API Routes are automatically optimized as isolated Serverless Functions that can scale infinitely
Middleware is automatically optimized as Edge Functions that have zero cold starts and boot instantly

In addition, Vercel provides features like:

Automatic performance monitoring with Next.js Speed Insights
Automatic HTTPS and SSL certificates
Automatic CI/CD (through GitHub, GitLab, Bitbucket, etc.)
Support for Environment Variables
Support for Custom Domains
Support for Image Optimization with next/image
Instant global deployments via git push

Deploy a Next.js application to Vercel for free to try it out.
Self-Hosting

You can self-host Next.js with support for all features using Node.js or Docker. You can also do a Static HTML Export, which has some limitations.
Node.js Server

Next.js can be deployed to any hosting provider that supports Node.js. For example, AWS EC2 or a DigitalOcean Droplet.
First, ensure your package.json has the "build" and "start" scripts:
package.json {
```
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start"
  }
}
```
Then, run npm run build to build your application. Finally, run npm run start to start the Node.js server. This server supports all features of Next.js.

If you are using next/image, consider adding sharp for more performant Image Optimization in your production environment by running npm install sharp in your project directory. On Linux platforms, sharp may require additional configuration to prevent excessive memory usage.

Docker Image

Next.js can be deployed to any hosting provider that supports Docker containers. You can use this approach when deploying to container orchestrators such as Kubernetes or HashiCorp Nomad, or when running inside a single node in any cloud provider.

Install Docker on your machine
Clone the with-docker example
Build your container: docker build -t nextjs-docker .
Run your container: docker run -p 3000:3000 nextjs-docker

If you need to use different Environment Variables across multiple environments, check out our with-docker-multi-env example.
Static HTML Export

If you'd like to do a static HTML export of your Next.js app, follow the directions on our Static HTML Export documentation.

Other Services

The following services support Next.js v12+. Below, you'll find examples or guides to deploy Next.js to each service.

Managed Server

AWS Copilot
Digital Ocean App Platform
Google Cloud Run
Heroku
Railway
Render

Good to know: There are also managed platforms that allow you to use a Dockerfile as shown in the example above.

Static Only

The following services only support deploying Next.js using output: 'export'.

GitHub Pages

You can also manually deploy the output from output: 'export' to any static hosting provider, often through your CI/CD pipeline like GitHub Actions, Jenkins, AWS CodeBuild, Circle CI, Azure Pipelines, and more.

Serverless

AWS Amplify
Azure Static Web Apps
Cloudflare Pages
Firebase
Netlify
Terraform
SST

Good to know: Not all serverless providers implement the Next.js Build API from next start. Please check with the provider to see what features are supported.

Automatic Updates

When you deploy your Next.js application, you want to see the latest version without needing to reload.
Next.js will automatically load the latest version of your application in the background when routing. For client-side navigations, next/link will temporarily function as a normal <a> tag.

Good to know: If a new page (with an old version) has already been prefetched by next/link, Next.js will use the old version. Navigating to a page that has not been prefetched (and is not cached at the CDN level) will load the latest version.

Manual Graceful shutdowns

When self-hosting, you might want to run code when the server shuts down on SIGTERM or SIGINT signals.
You can set the env variable NEXT_MANUAL_SIG_HANDLE to true and then register a handler for that signal inside your _document.js file. You will need to register the env variable directly in the package.json script, not in the .env file.

Good to know: Manual signal handling is not available in next dev.

```
package.json {
  "scripts": {
    "dev": "next dev",
    "build": "next build",
```

```
    "start": "NEXT_MANUAL_SIG_HANDLE=true next start"
  }
}
```

pages/_document.js

```
if (process.env.NEXT_MANUAL_SIG_HANDLE) {
  // this should be added in your custom _document
  process.on('SIGTERM', () => {
    console.log('Received SIGTERM: ', 'cleaning up')
    process.exit(0)
  })

  process.on('SIGINT', () => {
    console.log('Received SIGINT: ', 'cleaning up')
    process.exit(0)
  })
}
```

Static Exports

Next.js enables starting as a static site or Single-Page Application (SPA), then later optionally upgrading to use features that require a server.

When running next build, Next.js generates an HTML file per route. By breaking a strict SPA into individual HTML files, Next.js can avoid loading unnecessary JavaScript code on the client-side, reducing the bundle size and enabling faster page loads.

Since Next.js supports this static export, it can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets.

Configuration

To enable a static export, change the output mode inside next.config.js:

next.config.js

```
/**
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  output: 'export',
  // Optional: Add a trailing slash to all paths `/about` -> `/about/`
  // trailingSlash: true,
  // Optional: Change the output directory `out` -> `dist`
  // distDir: 'dist',
}

module.exports = nextConfig
```

After running next build, Next.js will produce an out folder which contains the HTML/CSS/JS assets for your application.

Supported Features

The core of Next.js has been designed to support static exports.Server Components

When you run next build to generate a static export, Server Components consumed inside the app directory will run during the build, similar to traditional static-site generation.The resulting component will be rendered into static HTML for the initial page load and a static payload for client navigation between routes. No changes are required for your Server Components when using the static export, unless they consume dynamic server functions.app/page.tsxTypeScript export default async function Page() {
  // This fetch will run on the server during `next build`
  const res = await fetch('https://api.example.com/...')
  const data = await res.json()

  return <main>...</main>
}Client Components

If you want to perform data fetching on the client, you can use a Client Component with SWR to deduplicate requests.app/other/page.tsxTypeScript 'use client'

import useSWR from 'swr'

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

export default function Page() {
  const { data, error } = useSWR(
    `https://jsonplaceholder.typicode.com/posts/1`,
    fetcher
)
  if (error) return 'Failed to load'
  if (!data) return 'Loading...'

  return data.title
}Since route transitions happen client-side, this behaves like a traditional SPA. For example, the following index route allows you to navigate to different posts on the client:app/page.tsxTypeScript import Link from 'next/link'

```
export default function Page() {
  return (
    <>
      <h1>Index Page</h1>
      <hr />
      <ul>
        <li>
          <Link href="/post/1">Post 1</Link>
        </li>
        <li>
          <Link href="/post/2">Post 2</Link>
        </li>
      </ul>
    </>
  )
}
```

Image Optimization

Image Optimization through next/image can be used with a static export by defining a custom image loader in next.config.js. For example, you can optimize images with a service like Cloudinary:

next.config.js
```
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export',
  images: {
    loader: 'custom',
    loaderFile: './app/image.ts',
  },
}

module.exports = nextConfig
```

This custom loader will define how to fetch images from a remote source. For example, the following loader will construct the URL for Cloudinary:

app/image.tsTypeScript
```
export default function cloudinaryLoader({
  src,
  width,
  quality,
}: {
  src: string
  width: number
```

```
  quality?: number
}) {
  const params = ['f_auto', 'c_limit', `w_${width}`, `q_${quality || 'auto'}`]
  return `https://res.cloudinary.com/demo/image/upload/${params.join(
    ','
  )}${src}`
}
```

You can then use next/image in your application, defining relative paths to the image in Cloudinary:
app/page.tsxTypeScript import Image from 'next/image'

```
export default function Page() {
  return <Image alt="turtles" src="/turtles.jpg" width={300} height={300} />
}
```

Route Handlers

Route Handlers will render a static response when running next build. Only the GET HTTP verb is supported. This can be used to generate static HTML, JSON, TXT, or other files from dynamic or static data. For example:app/data.json/route.tsTypeScript import { NextResponse } from 'next/server'

```
export async function GET() {
  return NextResponse.json({ name: 'Lee' })
}
```

The above file app/data.json/route.ts will render to a static file during next build, producing data.json containing { name: 'Lee' }.If you need to read dynamic values from the incoming request, you cannot use a static export.Browser APIs

Client Components are pre-rendered to HTML during next build. Because Web APIs like window, localStorage, and navigator are not available on the server, you need to safely access these APIs only when running in the browser. For example: 'use client';

```
import { useEffect } from 'react';

export default function ClientComponent() {
  useEffect(() => {
    // You now have access to `window`
    console.log(window.innerHeight);
  }, [])
```

```
  return ...;
}
```

Unsupported Features

After enabling the static export output mode, all routes inside app are opted-into the following Route Segment Config: export const dynamic = 'error'With this configuration, your application will produce an error if you try to use server functions like headers or cookies, since there is no runtime server. This ensures local development matches the same behavior as a static export. If you need to use server functions, you cannot use a static export.The following additional dynamic features are not supported with a static export:

rewrites in next.config.js
redirects in next.config.js
headers in next.config.js
Middleware
Incremental Static Regeneration

Deploying

With a static export, Next.js can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets.
When running next build, Next.js generates the static export into the out folder. Using next export is no longer needed. For example, let's say you have the following routes:

/
/blog/[id]

After running next build, Next.js will generate the following files:

/out/index.html
/out/404.html
/out/blog/post-1.html
/out/blog/post-2.html

If you are using a static host like Nginx, you can configure rewrites from incoming requests to the correct files:

nginx.conf server {
  listen 80;
  server_name acme.com;

  root /var/www;

  location / {
      try_files /out/index.html =404;
  }

  location /blog/ {
      rewrite ^/blog/(.*)$ /out/blog/$1.html break;
  }

  error_page 404 /out/404.html;
  location = /404.html {
      internal;
  }
}
Version History

VersionChangesv13.4.0App Router (Stable) adds enhanced static export support, including using React Server Components and Route Handlers.v13.3.0next export is deprecated and replaced with "output": "export"
Upgrade GuideUpgrade your application to newer versions of Next.js or migrate from the Pages Router to the App Router.
CodemodsCodemods are transformations that run on your codebase programmatically. This allows a large number of changes to be programmatically applied without having to manually go through every file.
Next.js provides Codemod transformations to help upgrade your Next.js codebase when an API is updated or deprecated.
Usage

In your terminal, navigate (cd) into your project's folder, then run:
Terminal npx @next/codemod <transform> <path>
Replacing <transform> and <path> with appropriate values.

transform - name of transform

path - files or directory to transform
--dry Do a dry-run, no code will be edited
--print Prints the changed output for comparison

Next.js Codemods

13.2

Use Built-in Font

built-in-next-font

Terminal npx @next/codemod@latest built-in-next-font .
This codemod uninstalls the @next/font package and transforms @next/font imports into the built-in next/font.
For example:
 import { Inter } from '@next/font/google'
Transforms into:
 import { Inter } from 'next/font/google'
13.0

Rename Next Image Imports

next-image-to-legacy-image

Terminal npx @next/codemod@latest next-image-to-legacy-image .
Safely renames next/image imports in existing Next.js 10, 11, or 12 applications to next/legacy/image in Next.js 13. Also renames next/future/image to next/image.
For example:
pages/index.js import Image1 from 'next/image'
import Image2 from 'next/future/image'

```
export default function Home() {
  return (
    <div>
      <Image1 src="/test.jpg" width="200" height="300" />
      <Image2 src="/test.png" width="500" height="400" />
    </div>
  )
}
```

Transforms into:
pages/index.js // 'next/image' becomes 'next/legacy/image'
import Image1 from 'next/legacy/image'
// 'next/future/image' becomes 'next/image'
import Image2 from 'next/image'

```
export default function Home() {
  return (
    <div>
      <Image1 src="/test.jpg" width="200" height="300" />
      <Image2 src="/test.png" width="500" height="400" />
    </div>
  )
}
```

Migrate to the New Image Component

next-image-experimental

Terminal npx @next/codemod@latest next-image-experimental .
Dangerously migrates from next/legacy/image to the new next/image by adding inline styles and removing unused props.

Removes layout prop and adds style.
Removes objectFit prop and adds style.
Removes objectPosition prop and adds style.
Removes lazyBoundary prop.
Removes lazyRoot prop.

Remove <a> Tags From Link Components

new-link

Terminal npx @next/codemod@latest new-link .
Remove <a> tags inside Link Components, or add a legacyBehavior prop to Links that cannot be auto-fixed.

For example:

```
 <Link href="/about">
  <a>About</a>
</Link>
// transforms into
<Link href="/about">
  About
</Link>

<Link href="/about">
  <a onClick={() => console.log('clicked')}>About</a>
</Link>
// transforms into
<Link href="/about" onClick={() => console.log('clicked')}>
  About
</Link>
```

In cases where auto-fixing can't be applied, the legacyBehavior prop is added. This allows your app to keep functioning using the old behavior for that particular link.

```
 const Component = () => <a>About</a>

<Link href="/about">
  <Component />
</Link>
// becomes
<Link href="/about" legacyBehavior>
  <Component />
</Link>
```
11

Migrate from CRA

cra-to-next

Terminal npx @next/codemod cra-to-next
Migrates a Create React App project to Next.js; creating a Pages Router and necessary config to match behavior. Client-side only rendering is leveraged initially to prevent breaking compatibility due to window usage during SSR and can be enabled seamlessly to allow the gradual adoption of Next.js specific features.
Please share any feedback related to this transform in this discussion.
10

Add React imports

add-missing-react-import

Terminal npx @next/codemod add-missing-react-import
Transforms files that do not import React to include the import in order for the new React JSX transform to work.
For example:
my-component.js export default class Home extends React.Component {
  render() {
    return Hello World
  }
}
Transforms into:
my-component.js import React from 'react'
export default class Home extends React.Component {
  render() {
    return Hello World
  }
}
9

Transform Anonymous Components into Named Components

name-default-component

Terminal npx @next/codemod name-default-component
Versions 9 and above.
Transforms anonymous components into named components to make sure they work with Fast Refresh.
For example:
my-component.js export default function () {
  return Hello World

}
Transforms into:
my-component.js export default function MyComponent() {
  return Hello World
}
The component will have a camel-cased name based on the name of the file, and it also works with arrow functions.
8

Transform AMP HOC into page config

withamp-to-config

Terminal npx @next/codemod withamp-to-config
Transforms the withAmp HOC into Next.js 9 page configuration.
For example:
 // Before
import { withAmp } from 'next/amp'

function Home() {
  return <h1>My AMP Page</h1>
}

export default withAmp(Home)
 // After
export default function Home() {
  return <h1>My AMP Page</h1>
}

export const config = {
  amp: true,
}
6

Use withRouter

url-to-withrouter

Terminal npx @next/codemod url-to-withrouter
Transforms the deprecated automatically injected url property on top level pages to using withRouter and the router property it injects. Read more here: https://nextjs.org/docs/messages/url-deprecated
For example:
From import React from 'react'

```
export default class extends React.Component {
  render() {
    const { pathname } = this.props.url
    return <div>Current pathname: {pathname}</div>
  }
}
```

To import React from 'react'

```
import { withRouter } from 'next/router'
export default withRouter(
  class extends React.Component {
    render() {
      const { pathname } = this.props.router
      return <div>Current pathname: {pathname}</div>
    }
  }
)
```

This is one case. All the cases that are transformed (and tested) can be found in the __testfixtures__ directory.
App Router Incremental Adoption GuideThis guide will help you:

Update your Next.js application from version 12 to version 13
Upgrade features that work in both the pages and the app directories
Incrementally migrate your existing application from pages to app

Upgrading

Node.js Version

The minimum Node.js version is now v16.8. See the Node.js documentation for more information.

Next.js Version

To update to Next.js version 13, run the following command using your preferred package manager:

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

ESLint Version

If you're using ESLint, you need to upgrade your ESLint version:

```
Terminal npm install -D eslint-config-next@latest
```

Good to know: You may need to restart the ESLint server in VS Code for the ESLint changes to take effect. Open the Command Palette (cmd+shift+p on Mac; ctrl+shift+p on Windows) and search for ESLint: Restart ESLint Server.

Next Steps

After you've updated, see the following sections for next steps:

Upgrade new features: A guide to help you upgrade to new features such as the improved Image and Link Components.

Migrate from the pages to app directory: A step-by-step guide to help you incrementally migrate from the pages to the app directory.

Upgrading New Features

Next.js 13 introduced the new App Router with new features and conventions. The new Router is available in the app directory and co-exists with the pages directory.
Upgrading to Next.js 13 does not require using the new App Router. You can continue using pages with new features that work in both directories, such as the updated Image component, Link component, Script component, and Font optimization.
<Image/> Component

Next.js 12 introduced new improvements to the Image Component with a temporary import: next/future/image. These improvements included less client-side JavaScript, easier ways to extend and style images, better accessibility, and native browser lazy loading.
In version 13, this new behavior is now the default for next/image.
There are two codemods to help you migrate to the new Image Component:

next-image-to-legacy-image codemod: Safely and automatically renames next/image imports to next/legacy/image. Existing components will maintain the same behavior.
next-image-experimental codemod: Dangerously adds inline styles and removes unused props. This will change the behavior of existing components to match the new defaults. To use this codemod, you need to run the next-image-to-legacy-image codemod first.

<Link> Component

The <Link> Component no longer requires manually adding an <a> tag as a child. This behavior was added as an experimental option in version 12.2 and is now the default. In Next.js 13, <Link> always renders <a> and allows you to forward props to the underlying tag.
For example:

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

```
// Next.js 12: `<a>` has to be nested otherwise it's excluded
<Link href="/about">
  <a>About</a>
</Link>

// Next.js 13: `<Link>` always renders `<a>` under the hood
<Link href="/about">
  About
</Link>
```

To upgrade your links to Next.js 13, you can use the new-link codemod.

<Script> Component

The behavior of next/script has been updated to support both pages and app, but some changes need to be made to ensure a smooth migration:

Move any beforeInteractive scripts you previously included in _document.js to the root layout file (app/layout.tsx).
The experimental worker strategy does not yet work in app and scripts denoted with this strategy will either have to be removed or modified to use a different strategy (e.g. lazyOnload).
onLoad, onReady, and onError handlers will not work in Server Components so make sure to move them to a Client Component or remove them altogether.

Font Optimization

Previously, Next.js helped you optimize fonts by inlining font CSS. Version 13 introduces the new next/font module which gives you the ability to customize your font loading experience while still ensuring great performance and privacy. next/font is supported in both the pages and app directories.
While inlining CSS still works in pages, it does not work in app. You should use next/font instead.
See the Font Optimization page to learn how to use next/font.
Migrating from pages to app

Ø<ß¥ Watch: Learn how to incrementally adopt the App Router !’ YouTube (16 minutes).

Moving to the App Router may be the first time using React features that Next.js builds on top of such as Server Components, Suspense, and more. When combined with new Next.js features such as special files and layouts, migration means new concepts, mental models, and behavioral changes to learn.
We recommend reducing the combined complexity of these updates by breaking down your migration into smaller steps. The app directory is intentionally designed to work simultaneously with the pages directory to allow for incremental page-by-page migration.

The app directory supports nested routes and layouts. Learn more.
Use nested folders to define routes and a special page.js file to make a route segment publicly accessible. Learn more.
Special file conventions are used to create UI for each route segment. The most common special files are page.js and layout.js.

Use page.js to define UI unique to a route.
Use layout.js to define UI that is shared across multiple routes.
.js, .jsx, or .tsx file extensions can be used for special files.

You can colocate other files inside the app directory such as components, styles, tests, and more. Learn more.
Data fetching functions like getServerSideProps and getStaticProps have been replaced with a new API inside app. getStaticPaths has been replaced with generateStaticParams.
pages/_app.js and pages/_document.js have been replaced with a single app/layout.js root layout. Learn more.
pages/_error.js has been replaced with more granular error.js special files. Learn more.
pages/404.js has been replaced with the not-found.js file.
pages/api/* currently remain inside the pages directory.

Step 1: Creating the app directory

Update to the latest Next.js version (requires 13.4 or greater):
npm install next@latest
Then, create a new app directory at the root of your project (or src/ directory).
Step 2: Creating a Root Layout

Create a new app/layout.tsx file inside the app directory. This is a root layout that will apply to all routes inside app.
app/layout.tsxTypeScript export default function RootLayout({
  // Layouts must accept a children prop.
  // This will be populated with nested layouts or pages
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
)
}

The app directory must include a root layout.
The root layout must define <html>, and <body> tags since Next.js does not automatically create them
The root layout replaces the pages/_app.tsx and pages/_document.tsx files.
.js, .jsx, or .tsx extensions can be used for layout files.

To manage <head> HTML elements, you can use the built-in SEO support:
app/layout.tsxTypeScript import { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'Home',
  description: 'Welcome to Next.js',
}

Migrating _document.js and _app.js

If you have an existing _app or _document file, you can copy the contents (e.g. global styles) to the root layout (app/layout.tsx). Styles in app/layout.tsx will not apply to pages/*. You should keep _app/_document while migrating to prevent your pages/* routes from breaking. Once fully migrated, you can then safely delete them.
If you are using any React Context providers, they will need to be moved to a Client Component.

Migrating the getLayout() pattern to Layouts (Optional)

Next.js recommended adding a property to Page components to achieve per-page layouts in the pages directory. This pattern can be replaced with native support for nested layouts in the app directory.
See before and after exampleBeforecomponents/DashboardLayout.js export default function DashboardLayout({ children }) {
  return (

      <h2>My Dashboard</h2>
      {children}

)
}pages/dashboard/index.js import DashboardLayout from '../components/DashboardLayout'

export default function Page() {
  return My Page
}

Page.getLayout = function getLayout(page) {
  return <DashboardLayout>{page}</DashboardLayout>
}After

Remove the Page.getLayout property from pages/dashboard/index.js and follow the steps for migrating pages to the app directory.
app/dashboard/page.js export default function Page() {
  return My Page
}

Move the contents of DashboardLayout into a new Client Component to retain pages directory behavior.
app/dashboard/DashboardLayout.js 'use client' // this directive should be at top of the file, before any imports.

// This is a Client Component
export default function DashboardLayout({ children }) {
  return (

      <h2>My Dashboard</h2>
      {children}

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

Import the DashboardLayout into a new layout.js file inside the app directory.
app/dashboard/layout.js import DashboardLayout from './DashboardLayout'

```
// This is a Server Component
export default function Layout({ children }) {
  return <DashboardLayout>{children}</DashboardLayout>
}
```

You can incrementally move non-interactive parts of DashboardLayout.js (Client Component) into layout.js (Server Component) to reduce the amount of component JavaScript you send to the client.

Step 3: Migrating next/head

In the pages directory, the next/head React component is used to manage <head> HTML elements such as title and meta . In the app directory, next/head is replaced with the new built-in SEO support.
Before:
pages/index.tsxTypeScript import Head from 'next/head'

```
export default function Page() {
  return (
    <>
      <Head>
        <title>My page title</title>
      </Head>
    </>
  )
}
```

After:
app/page.tsxTypeScript import { Metadata } from 'next'

```
export const metadata: Metadata = {
  title: 'My Page Title',
```

}

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

See all metadata options.
Step 4: Migrating Pages

Pages in the app directory are Server Components by default. This is different from the pages directory where pages are Client Components.
Data fetching has changed in app. getServerSideProps, getStaticProps and getInitialProps have been replaced for a simpler API.
The app directory uses nested folders to define routes and a special page.js file to make a route segment publicly accessible.

pages Directoryapp DirectoryRouteindex.jspage.js/about.jsabout/page.js/aboutblog/[slug].jsblog/[slug]/page.js/blog/post-1

We recommend breaking down the migration of a page into two main steps:

Step 1: Move the default exported Page Component into a new Client Component.
Step 2: Import the new Client Component into a new page.js file inside the app directory.

Good to know: This is the easiest migration path because it has the most comparable behavior to the pages directory.

Step 1: Create a new Client Component

Create a new separate file inside the app directory (i.e. app/home-page.tsx or similar) that exports a Client Component. To define Client Components, add the 'use client' directive to the top of the file (before any imports).
Move the default exported page component from pages/index.js to app/home-page.tsx.

app/home-page.tsxTypeScript 'use client'

// This is a Client Component. It receives data as props and
// has access to state and effects just like Page components
// in the `pages` directory.

```
export default function HomePage({ recentPosts }) {
  return (
    <div>
      {recentPosts.map((post) => (
        <div key={post.id}>{post.title}</div>
      ))}
    </div>
  )
}
```

Step 2: Create a new page

Create a new app/page.tsx file inside the app directory. This is a Server Component by default.

Import the home-page.tsx Client Component into the page.

If you were fetching data in pages/index.js, move the data fetching logic directly into the Server Component using the new data fetching APIs. See the data fetching upgrade guide for more details.
app/page.tsxTypeScript // Import your Client Component

```
import HomePage from './home-page'

async function getPosts() {
  const res = await fetch('https://...')
  const posts = await res.json()
  return posts
}

export default async function Page() {
  // Fetch data directly in a Server Component
  const recentPosts = await getPosts()
  // Forward fetched data to your Client Component
  return <HomePage recentPosts={recentPosts} />
}
```

If your previous page used useRouter, you'll need to update to the new routing hooks. Learn more.

Start your development server and visit http://localhost:3000. You should see your

existing index route, now served through the app directory.

Step 5: Migrating Routing Hooks

A new router has been added to support the new behavior in the app directory.
In app, you should use the three new hooks imported from next/navigation: useRouter(), usePathname(), and useSearchParams().

The new useRouter hook is imported from next/navigation and has different behavior to the useRouter hook in pages which is imported from next/router.

The useRouter hook imported from next/router is not supported in the app directory but can continue to be used in the pages directory.

The new useRouter does not return the pathname string. Use the separate usePathname hook instead.
The new useRouter does not return the query object. Use the separate useSearchParams hook instead.
You can use useSearchParams and usePathname together to listen to page changes. See the Router Events section for more details.
These new hooks are only supported in Client Components. They cannot be used in Server Components.

app/example-client-component.tsxTypeScript 'use client'

```
import { useRouter, usePathname, useSearchParams } from 'next/navigation'

export default function ExampleClientComponent() {
  const router = useRouter()
  const pathname = usePathname()
  const searchParams = useSearchParams()

  // ...
}
```

In addition, the new useRouter hook has the following changes:

isFallback has been removed because fallback has been replaced.
The locale, locales, defaultLocales, domainLocales values have been removed because built-in i18n Next.js features are no longer necessary in the app directory.

Learn more about i18n.
basePath has been removed. The alternative will not be part of useRouter. It has not yet been implemented.
asPath has been removed because the concept of as has been removed from the new router.
isReady has been removed because it is no longer necessary. During static rendering, any component that uses the useSearchParams() hook will skip the prerendering step and instead be rendered on the client at runtime.

View the useRouter() API reference.
Step 6: Migrating Data Fetching Methods

The pages directory uses getServerSideProps and getStaticProps to fetch data for pages. Inside the app directory, these previous data fetching functions are replaced with a simpler API built on top of fetch() and async React Server Components.
app/page.tsxTypeScript export default async function Page() {
```
  // This request should be cached until manually invalidated.
  // Similar to `getStaticProps`.
  // `force-cache` is the default and can be omitted.
  const staticData = await fetch(`https://...`, { cache: 'force-cache' })

  // This request should be refetched on every request.
  // Similar to `getServerSideProps`.
  const dynamicData = await fetch(`https://...`, { cache: 'no-store' })

  // This request should be cached with a lifetime of 10 seconds.
  // Similar to `getStaticProps` with the `revalidate` option.
  const revalidatedData = await fetch(`https://...`, {
    next: { revalidate: 10 },
  })

  return <div>...</div>
}
```

Server-side Rendering (getServerSideProps)

In the pages directory, getServerSideProps is used to fetch data on the server and forward props to the default exported React component in the file. The initial HTML for

the page is prerendered from the server, followed by "hydrating" the page in the browser (making it interactive).
pages/dashboard.js // `pages` directory

```
export async function getServerSideProps() {
  const res = await fetch(`https://...`)
  const projects = await res.json()

  return { props: { projects } }
}

export default function Dashboard({ projects }) {
  return (
    <ul>
      {projects.map((project) => (
        <li key={project.id}>{project.name}</li>
      ))}
    </ul>
  )
}
```

In the app directory, we can colocate our data fetching inside our React components using Server Components. This allows us to send less JavaScript to the client, while maintaining the rendered HTML from the server.
By setting the cache option to no-store, we can indicate that the fetched data should never be cached. This is similar to getServerSideProps in the pages directory.
app/dashboard/page.tsxTypeScript // `app` directory

```
// This function can be named anything
async function getProjects() {
  const res = await fetch(`https://...`, { cache: 'no-store' })
  const projects = await res.json()

  return projects
}

export default async function Dashboard() {
  const projects = await getProjects()

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

Accessing Request Object

In the pages directory, you can retrieve request-based data based on the Node.js HTTP API.
For example, you can retrieve the req object from getServerSideProps and use it to retrieve the request's cookies and headers.
pages/index.js // `pages` directory

```
export async function getServerSideProps({ req, query }) {
  const authHeader = req.getHeaders()['authorization'];
  const theme = req.cookies['theme'];

  return { props: { ... }}
}

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

The app directory exposes new read-only functions to retrieve request data:

headers(): Based on the Web Headers API, and can be used inside Server Components to retrieve request headers.
cookies(): Based on the Web Cookies API, and can be used inside Server Components to retrieve cookies.

app/page.tsxTypeScript // `app` directory

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

async function getData() {
  const authHeader = headers().get('authorization')

  return '...'
}

export default async function Page() {
  // You can use `cookies()` or `headers()` inside Server Components
  // directly or in your data fetching function
  const theme = cookies().get('theme')
  const data = await getData()
  return '...'
}
```

Static Site Generation (getStaticProps)

In the pages directory, the getStaticProps function is used to pre-render a page at build time. This function can be used to fetch data from an external API or directly from a database, and pass this data down to the entire page as it's being generated during the build.

pages/index.js // `pages` directory

```
export async function getStaticProps() {
  const res = await fetch(`https://...`)
  const projects = await res.json()

  return { props: { projects } }
}

export default function Index({ projects }) {
  return projects.map((project) => <div>{project.name}</div>)
}
```

In the app directory, data fetching with fetch() will default to cache: 'force-cache', which will cache the request data until manually invalidated. This is similar to getStaticProps in the pages directory.

app/page.js // `app` directory

```
// This function can be named anything
async function getProjects() {
  const res = await fetch(`https://...`)
  const projects = await res.json()

  return projects
}

export default async function Index() {
  const projects = await getProjects()

  return projects.map((project) => <div>{project.name}</div>)
}
```

Dynamic paths (getStaticPaths)

In the pages directory, the getStaticPaths function is used to define the dynamic paths that should be pre-rendered at build time.

```
pages/posts/[id].js // `pages` directory
import PostLayout from '@/components/post-layout'

export async function getStaticPaths() {
  return {
    paths: [{ params: { id: '1' } }, { params: { id: '2' } }],
  }
}

export async function getStaticProps({ params }) {
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  return { props: { post } }
}

export default function Post({ post }) {
  return <PostLayout post={post} />
}
```

In the app directory, getStaticPaths is replaced with generateStaticParams. generateStaticParams behaves similarly to getStaticPaths, but has a simplified API for returning route parameters and can be used inside layouts. The return shape of generateStaticParams is an array of segments instead of an array of nested param objects or a string of resolved paths.

```
app/posts/[id]/page.js // `app` directory
import PostLayout from '@/components/post-layout'

export async function generateStaticParams() {
  return [{ id: '1' }, { id: '2' }]
}

async function getPost(params) {
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  return post
}

export default async function Post({ params }) {
  const post = await getPost(params)

  return <PostLayout post={post} />
}
```

Using the name generateStaticParams is more appropriate than getStaticPaths for the new model in the app directory. The get prefix is replaced with a more descriptive generate, which sits better alone now that getStaticProps and getServerSideProps are no longer necessary. The Paths suffix is replaced by Params, which is more appropriate for nested routing with multiple dynamic segments.

Replacing fallback

In the pages directory, the fallback property returned from getStaticPaths is used to define the behavior of a page that isn't pre-rendered at build time. This property can be set to true to show a fallback page while the page is being generated, false to show a 404 page, or blocking to generate the page at request time.

```
pages/posts/[id].js // `pages` directory

export async function getStaticPaths() {
  return {
    paths: [],
    fallback: 'blocking'
  };
}

export async function getStaticProps({ params }) {
  ...
}

export default function Post({ post }) {
  return ...
}
```

In the app directory the config.dynamicParams property controls how params outside of generateStaticParams are handled:

true: (default) Dynamic segments not included in generateStaticParams are generated on demand.
false: Dynamic segments not included in generateStaticParams will return a 404.

This replaces the fallback: true | false | 'blocking' option of getStaticPaths in the pages directory. The fallback: 'blocking' option is not included in dynamicParams because the difference between 'blocking' and true is negligible with streaming.

```
app/posts/[id]/page.js // `app` directory

export const dynamicParams = true;
```

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

async function getPost(params) {
  ...
}

export default async function Post({ params }) {
  const post = await getPost(params);

  return ...
}
```

With dynamicParams set to true (the default), when a route segment is requested that hasn't been generated, it will be server-rendered and cached as static data on success.

Incremental Static Regeneration (getStaticProps with revalidate)

In the pages directory, the getStaticProps function allows you to add a revalidate field to automatically regenerate a page after a certain amount of time. This is called Incremental Static Regeneration (ISR) and helps you update static content without redeploying.

```
pages/index.js // `pages` directory

export async function getStaticProps() {
  const res = await fetch(`https://.../posts`)
  const posts = await res.json()

  return {
    props: { posts },
    revalidate: 60,
  }
}

export default function Index({ posts }) {
  return (
    <Layout>
      <PostList posts={posts} />
    </Layout>
  )
}
```

In the app directory, data fetching with fetch() can use revalidate, which will cache the request for the specified amount of seconds.

```
app/page.js // `app` directory

async function getPosts() {
  const res = await fetch(`https://.../posts`, { next: { revalidate: 60 } })
  const data = await res.json()

  return data.posts
}

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

  return posts.map((post) => <div>{post.name}</div>)
}
```

API Routes

API Routes continue to work in the pages/api directory without any changes. However, they have been replaced by Route Handlers in the app directory.
Route Handlers allow you to create custom request handlers for a given route using the Web Request and Response APIs.
app/api/route.tsTypeScript export async function GET(request: Request) {}

Good to know: If you previously used API routes to call an external API from the client, you can now use Server Components instead to securely fetch data. Learn more about data fetching.

Step 7: Styling

In the pages directory, global stylesheets are restricted to only pages/_app.js. With the app directory, this restriction has been lifted. Global styles can be added to any layout, page, or component.

CSS Modules
Tailwind CSS
Global Styles
CSS-in-JS
External Stylesheets

Sass

Tailwind CSS

If you're using Tailwind CSS, you'll need to add the app directory to your tailwind.config.js file:
tailwind.config.js module.exports = {
  content: [
    './app/**/*.{js,ts,jsx,tsx,mdx}', // <-- Add this line
    './pages/**/*.{js,ts,jsx,tsx,mdx}',
    './components/**/*.{js,ts,jsx,tsx,mdx}',
  ],
}
You'll also need to import your global styles in your app/layout.js file:
app/layout.js import '../styles/globals.css'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
)
}
Learn more about styling with Tailwind CSS
Codemods

Next.js provides Codemod transformations to help upgrade your codebase when a feature is deprecated. See Codemods for more information.
API ReferenceThe Next.js API reference is divided into the following sections:
Components
Font Module
This API reference will help you understand how to use next/font/google and next/font/local. For features and usage, please see the Optimizing Fonts page.
Font Function Arguments

For usage, review Google Fonts and Local Fonts.
Keyfont/googlefont/localTypeRequiredsrcString or Array of ObjectsYesweightString or ArrayRequired/OptionalstyleString or Array-subsetsArray of Strings-axesArray of Strings-displayString-preloadBoolean-fallbackArray of Strings-adjustFontFallbackBoolean or String-variableString-declarationsArray of Objects-src

The path of the font file as a string or an array of objects (with type Array<{path: string, weight?: string, style?: string}>) relative to the directory where the font loader function is called.
Used in next/font/local

Required

Examples:

src:'./fonts/my-font.woff2' where my-font.woff2 is placed in a directory named fonts inside the app directory
src:[{path: './inter/Inter-Thin.ttf', weight: '100',},{path: './inter/Inter-Regular.ttf',weight: '400',},{path: './inter/Inter-Bold-Italic.ttf', weight: '700',style: 'italic',},]
if the font loader function is called in app/page.tsx using src:'../styles/fonts/my-font.ttf', then my-font.ttf is placed in styles/fonts at the root of the project

weight

The font weight with the following possibilities:

A string with possible values of the weights available for the specific font or a range of values if it's a variable font
An array of weight values if the font is not a variable google font. It applies to next/font/google only.

Used in next/font/google and next/font/local

Required if the font being used is not variable

Examples:

weight: '400': A string for a single weight value - for the font Inter, the possible values are '100', '200', '300', '400', '500', '600', '700', '800', '900' or 'variable' where 'variable' is the default)
weight: '100 900': A string for the range between 100 and 900 for a variable font
weight: ['100','400','900']: An array of 3 possible values for a non variable font

style

The font style with the following possibilities:

A string value with default value of 'normal'
An array of style values if the font is not a variable google font. It applies to next/font/google only.

Used in next/font/google and next/font/local

Optional

Examples:

style: 'italic': A string - it can be normal or italic for next/font/google
style: 'oblique': A string - it can take any value for next/font/local but is expected to come from standard font styles
style: ['italic','normal']: An array of 2 values for next/font/google - the values are from normal and italic

subsets

The font subsets defined by an array of string values with the names of each subset you would like to be preloaded. Fonts specified via subsets will have a link preload tag injected into the head when the preload option is true, which is the default.
Used in next/font/google

Optional

Examples:

subsets: ['latin']: An array with the subset latin

You can find a list of all subsets on the Google Fonts page for your font.
axes

Some variable fonts have extra axes that can be included. By default, only the font weight is included to keep the file size down. The possible values of axes depend on the specific font.
Used in next/font/google

Optional

Examples:

axes: ['slnt']: An array with value slnt for the Inter variable font which has slnt as additional axes as shown here. You can find the possible axes values for your font by using the filter on the Google variable fonts page and looking for axes other than wght

display

The font display with possible string values of 'auto', 'block', 'swap', 'fallback' or 'optional' with default value of 'swap'.
Used in next/font/google and next/font/local

Optional

Examples:

display: 'optional': A string assigned to the optional value

preload

A boolean value that specifies whether the font should be preloaded or not. The default is true.

Used in next/font/google and next/font/local

Optional

Examples:

preload: false

fallback

The fallback font to use if the font cannot be loaded. An array of strings of fallback fonts with no default.

Optional

Used in next/font/google and next/font/local
Examples:

fallback: ['system-ui', 'arial']: An array setting the fallback fonts to system-ui or arial

adjustFontFallback

For next/font/google: A boolean value that sets whether an automatic fallback font should be used to reduce Cumulative Layout Shift. The default is true.
For next/font/local: A string or boolean false value that sets whether an automatic fallback font should be used to reduce Cumulative Layout Shift. The possible values are 'Arial', 'Times New Roman' or false. The default is 'Arial'.

Used in next/font/google and next/font/local

Optional

Examples:

adjustFontFallback: false: for ``next/font/google`
adjustFontFallback: 'Times New Roman': for next/font/local

variable

A string value to define the CSS variable name to be used if the style is applied with the CSS variable method.
Used in next/font/google and next/font/local

Optional

Examples:

variable: '--my-font': The CSS variable --my-font is declared

declarations

An array of font face descriptor key-value pairs that define the generated @font-face further.
Used in next/font/local

Optional

Examples:

declarations: [{ prop: 'ascent-override', value: '90%' }]

Applying Styles

You can apply the font styles in three ways:

className
style
CSS Variables

className

Returns a read-only CSS className for the loaded font to be passed to an HTML element.
Hello, Next.js!
style

Returns a read-only CSS style object for the loaded font to be passed to an HTML element, including style.fontFamily to access the font family name and fallback fonts.
Hello World
CSS Variables

If you would like to set your styles in an external style sheet and specify additional options there, use the CSS variable method.
In addition to importing the font, also import the CSS file where the CSS variable is defined and set the variable option of the font loader object as follows:
app/page.tsxTypeScript import { Inter } from 'next/font/google'
import styles from '../styles/component.module.css'

```
const inter = Inter({
  variable: '--font-inter',
})
```

To use the font, set the className of the parent container of the text you would like to style to the font loader's variable value and the className of the text to the styles property from the external CSS file.
app/page.tsxTypeScript <main className={inter.variable}>
  Hello World
</main>

Define the text selector class in the component.module.css CSS file as follows:
styles/component.module.css .text {
  font-family: var(--font-inter);
  font-weight: 200;
  font-style: italic;
}

In the example above, the text Hello World is styled using the Inter font and the generated font fallback with font-weight: 200 and font-style: italic.
Using a font definitions file

Every time you call the localFont or Google font function, that font will be hosted as one instance in your application. Therefore, if you need to use the same font in multiple places, you should load it in one place and import the related font object where you need it. This is done using a font definitions file.
For example, create a fonts.ts file in a styles folder at the root of your app directory. Then, specify your font definitions as follows:
styles/fonts.tsTypeScript import { Inter, Lora, Source_Sans_Pro } from 'next/font/google'
import localFont from 'next/font/local'

```
// define your variable fonts
const inter = Inter()
const lora = Lora()
// define 2 weights of a non-variable font
const sourceCodePro400 = Source_Sans_Pro({ weight: '400' })
const sourceCodePro700 = Source_Sans_Pro({ weight: '700' })
// define a custom local font where GreatVibes-Regular.ttf is stored in the styles folder
const greatVibes = localFont({ src: './GreatVibes-Regular.ttf' })

export { inter, lora, sourceCodePro400, sourceCodePro700, greatVibes }
```

You can now use these definitions in your code as follows:
app/page.tsxTypeScript import { inter, lora, sourceCodePro700, greatVibes } from '../
styles/fonts'

```
export default function Page() {
  return (
    <div>
      <p className={inter.className}>Hello world using Inter font</p>
      <p style={lora.style}>Hello world using Lora font</p>
      <p className={sourceCodePro700.className}>
        Hello world using Source_Sans_Pro font with weight 700
      </p>
      <p className={greatVibes.className}>My title in Great Vibes font</p>
    </div>
  )
}
```

To make it easier to access the font definitions in your code, you can define a path alias

in your tsconfig.json or jsconfig.json files as follows:
tsconfig.json {
  "compilerOptions": {
    "paths": {
      "@/fonts": ["./styles/fonts"]
    }
  }
}
You can now import any font definition as follows:
app/about/page.tsxTypeScript import { greatVibes, sourceCodePro400 } from '@/fonts'

Version Changes

VersionChangesv13.2.0@next/font renamed to next/font. Installation no longer required.v13.0.0@next/font was added.
<Image>
Examples
Image Component

This API reference will help you understand how to use props and configuration options available for the Image Component. For features and usage, please see the Image Component page.
app/page.js import Image from 'next/image'

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

Props

Here's a summary of the props available for the Image Component:

PropExampleTypeRequiredsrcsrc="/profile.png"StringYeswidthwidth={500}Integer (px)Yesheightheight={500}Integer (px)Yesaltalt="Picture of the author"StringYesloaderloader={imageLoader}Function-fillfill={true}Boolean-sizessizes="(max-width: 768px) 100vw"String-qualityquality={80}Integer (1-100)-prioritypriority={true}Boolean-placeholderplaceholder="blur"String-stylestyle={{objectFit: "contain"}}Object-onLoadingCompleteonLoadingComplete={img => done())}Function-onLoadonLoad={event => done())}Function-onErroronError(event => fail()}Function-loadingloading="lazy"String-blurDataURLblurDataURL="data:image/jpeg..."String-Required Props

The Image Component requires the following properties: src, width, height, and alt.
app/page.js import Image from 'next/image'

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

src

Must be one of the following:

A statically imported image file
A path string. This can be either an absolute external URL, or an internal path depending on the loader prop.

When using an external URL, you must add it to remotePatterns in next.config.js.
width

The width property represents the rendered width in pixels, so it will affect how large the image appears.
Required, except for statically imported images or images with the fill property.
height

The height property represents the rendered height in pixels, so it will affect how large the image appears.
Required, except for statically imported images or images with the fill property.
alt

The alt property is used to describe the image for screen readers and search engines. It is also the fallback text if images have been disabled or an error occurs while loading the image.
It should contain text that could replace the image without changing the meaning of the page. It is not meant to supplement the image and should not repeat information that is already provided in the captions above or below the image.
If the image is purely decorative or not intended for the user, the alt property should be an empty string (alt="").
Learn more
Optional Props

The <Image /> component accepts a number of additional properties beyond those which are required. This section describes the most commonly-used properties of the Image component. Find details about more rarely-used properties in the Advanced Props section.
loader

A custom function used to resolve image URLs.

A loader is a function returning a URL string for the image, given the following parameters:

src
width
quality

Here is an example of using a custom loader:

```
 'use client'

import Image from 'next/image'

const imageLoader = ({ src, width, quality }) => {
  return `https://example.com/${src}?w=${width}&q=${quality || 75}`
}

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

Good to know: Using props like loader, which accept a function, require using Client Components to serialize the provided function.

Alternatively, you can use the loaderFile configuration in next.config.js to configure every instance of next/image in your application, without passing a prop.
fill

```
 fill={true} // {true} | {false}
```

A boolean that causes the image to fill the parent element instead of setting width and height.
The parent element must assign position: "relative", position: "fixed", or position: "absolute" style.
By default, the img element will automatically be assigned the position: "absolute" style.
The default image fit behavior will stretch the image to fit the container. You may prefer

to set object-fit: "contain" for an image which is letterboxed to fit the container and preserve aspect ratio.
Alternatively, object-fit: "cover" will cause the image to fill the entire container and be cropped to preserve aspect ratio. For this to look correct, the overflow: "hidden" style should be assigned to the parent element.
For more information, see also:

position
object-fit
object-position

sizes

A string that provides information about how wide the image will be at different breakpoints. The value of sizes will greatly affect performance for images using fill or which are styled to have a responsive size.
The sizes property serves two important purposes related to image performance:

First, the value of sizes is used by the browser to determine which size of the image to download, from next/image's automatically-generated source set. When the browser chooses, it does not yet know the size of the image on the page, so it selects an image that is the same size or larger than the viewport. The sizes property allows you to tell the browser that the image will actually be smaller than full screen. If you don't specify a sizes value in an image with the fill property, a default value of 100vw (full screen width) is used.
Second, the sizes property configures how next/image automatically generates an image source set. If no sizes value is present, a small source set is generated, suitable for a fixed-size image. If sizes is defined, a large source set is generated, suitable for a responsive image. If the sizes property includes sizes such as 50vw, which represent a percentage of the viewport width, then the source set is trimmed to not include any values which are too small to ever be necessary.

For example, if you know your styling will cause an image to be full-width on mobile devices, in a 2-column layout on tablets, and a 3-column layout on desktop displays, you should include a sizes property such as the following:

```
 import Image from 'next/image'

export default function Page() {
  return (
    <div className="grid-element">
      <Image
        fill
```

```
        src="/example.png"
        sizes="(max-width: 768px) 100vw, (max-width: 1200px) 50vw, 33vw"
      />
    </div>
  )
}
```

This example sizes could have a dramatic effect on performance metrics. Without the 33vw sizes, the image selected from the server would be 3 times as wide as it needs to be. Because file size is proportional to the square of the width, without sizes the user would download an image that's 9 times larger than necessary.
Learn more about srcset and sizes:

web.dev
mdn

quality

```
 quality={75} // {number 1-100}
```

The quality of the optimized image, an integer between 1 and 100, where 100 is the best quality and therefore largest file size. Defaults to 75.
priority

```
 priority={false} // {false} | {true}
```

When true, the image will be considered high priority and
preload. Lazy loading is automatically disabled for images using priority.
You should use the priority property on any image detected as the Largest Contentful Paint (LCP) element. It may be appropriate to have multiple priority images, as different images may be the LCP element for different viewport sizes.
Should only be used when the image is visible above the fold. Defaults to false.
placeholder

```
 placeholder = 'empty' // {empty} | {blur}
```

A placeholder to use while the image is loading. Possible values are blur or empty. Defaults to empty.

When blur, the blurDataURL property will be used as the placeholder. If src is an object from a static import and the imported image is .jpg, .png, .webp, or .avif, then blurDataURL will be automatically populated.
For dynamic images, you must provide the blurDataURL property. Solutions such as Plaiceholder can help with base64 generation.
When empty, there will be no placeholder while the image is loading, only empty space.
Try it out:

Demo the blur placeholder
Demo the shimmer effect with blurDataURL prop
Demo the color effect with blurDataURL prop

Advanced Props

In some cases, you may need more advanced usage. The <Image /> component optionally accepts the following advanced properties.
style

Allows passing CSS styles to the underlying image element.
components/ProfileImage.js const imageStyle = {
  borderRadius: '50%',
  border: '1px solid #fff',
}

export default function ProfileImage() {
  return <Image src="..." style={imageStyle} />
}
Remember that the required width and height props can interact with your styling. If you use styling to modify an image's width, you should also style its height to auto to preserve its intrinsic aspect ratio, or your image will be distorted.
onLoadingComplete

'use client'

<Image onLoadingComplete={(img) => console.log(img.naturalWidth)} />
A callback function that is invoked once the image is completely loaded and the placeholder has been removed.
The callback function will be called with one argument, a reference to the underlying <img> element.

Good to know: Using props like onLoadingComplete, which accept a function, require using Client Components to serialize the provided function.

onLoad

<Image onLoad={(e) => console.log(e.target.naturalWidth)} />
A callback function that is invoked when the image is loaded.
The load event might occur before the image placeholder is removed and the image is fully decoded. If you want to wait until the image has fully loaded, use onLoadingComplete instead.

Good to know: Using props like onLoad, which accept a function, require using Client Components to serialize the provided function.

onError

<Image onError={(e) => console.error(e.target.id)} />
A callback function that is invoked if the image fails to load.

Good to know: Using props like onError, which accept a function, require using Client Components to serialize the provided function.

loading

Recommendation: This property is only meant for advanced use cases. Switching an image to load with eager will normally hurt performance. We recommend using the priority property instead, which will eagerly preload the image.

loading = 'lazy' // {lazy} | {eager}
The loading behavior of the image. Defaults to lazy.
When lazy, defer loading the image until it reaches a calculated distance from the viewport.
When eager, load the image immediately.
Learn more about the loading attribute.
blurDataURL

A Data URL to be used as a placeholder image before the src image successfully loads. Only takes effect when combined with placeholder="blur".
Must be a base64-encoded image. It will be enlarged and blurred, so a very small image (10px or less) is recommended. Including larger images as placeholders may harm your application performance.
Try it out:

Demo the default blurDataURL prop
Demo the shimmer effect with blurDataURL prop
Demo the color effect with blurDataURL prop

You can also generate a solid color Data URL to match the image.
unoptimized

unoptimized = {false} // {false} | {true}
When true, the source image will be served as-is instead of changing quality, size, or format. Defaults to false.

```
import Image from 'next/image'

const UnoptimizedImage = (props) => {
  return <Image {...props} unoptimized />
}
```

Since Next.js 12.3.0, this prop can be assigned to all images by updating next.config.js with the following configuration:

```
next.config.js module.exports = {
  images: {
```

```
    unoptimized: true,
  },
}
```

Other Props

Other properties on the <Image /> component will be passed to the underlying img element with the exception of the following:

srcSet. Use Device Sizes instead.
decoding. It is always "async".

Configuration Options

In addition to props, you can configure the Image Component in next.config.js. The following options are available:
remotePatterns

To protect your application from malicious users, configuration is required in order to use external images. This ensures that only external images from your account can be served from the Next.js Image Optimization API. These external images can be configured with the remotePatterns property in your next.config.js file, as shown below:
next.config.js

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'example.com',
        port: '',
        pathname: '/account123/**',
      },
    ],
  },
}
```

Good to know: The example above will ensure the src property of next/image must start with https://example.com/account123/. Any other protocol, hostname, port, or unmatched path will respond with 400 Bad Request.

Below is another example of the remotePatterns property in the next.config.js file:

```
next.config.js module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
      },
    ],
  },
}
```

Good to know: The example above will ensure the src property of next/image must start with https://img1.example.com or https://me.avatar.example.com or any number of subdomains. Any other protocol or unmatched hostname will respond with 400 Bad Request.

Wildcard patterns can be used for both pathname and hostname and have the following syntax:

* match a single path segment or subdomain
** match any number of path segments at the end or subdomains at the beginning

The ** syntax does not work in the middle of the pattern.
domains

Warning: We recommend configuring strict remotePatterns instead of domains in order to protect your application from malicious users. Only use domains if you own all the content served from the domain.

Similar to remotePatterns, the domains configuration can be used to provide a list of allowed hostnames for external images.
However, the domains configuration does not support wildcard pattern matching and it cannot restrict protocol, port, or pathname.
Below is an example of the domains property in the next.config.js file:

```
next.config.js module.exports = {
  images: {
```

```
    domains: ['assets.acme.com'],
  },
}
```

loaderFile

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure the loaderFile in your next.config.js like the following:

```
next.config.js module.exports = {
  images: {
    loader: 'custom',
    loaderFile: './my/image/loader.js',
  },
}
```

This must point to a file relative to the root of your Next.js application. The file must export a default function that returns a string, for example:

```
 'use client'

export default function myImageLoader({ src, width, quality }) {
  return `https://example.com/${src}?w=${width}&q=${quality || 75}`
}
```

Alternatively, you can use the loader prop to configure each instance of next/image.
Examples:

Custom Image Loader Configuration

Good to know: Customizing the image loader file, which accepts a function, require using Client Components to serialize the provided function.

Advanced

The following configuration is for advanced use cases and is usually not necessary. If you choose to configure the properties below, you will override any changes to the Next.js defaults in future updates.

deviceSizes

If you know the expected device widths of your users, you can specify a list of device width breakpoints using the deviceSizes property in next.config.js. These widths are used when the next/image component uses sizes prop to ensure the correct image is served for user's device.
If no configuration is provided, the default below is used.

```
next.config.js module.exports = {
  images: {
    deviceSizes: [640, 750, 828, 1080, 1200, 1920, 2048, 3840],
  },
}
```

imageSizes

You can specify a list of image widths using the images.imageSizes property in your next.config.js file. These widths are concatenated with the array of device sizes to form the full array of sizes used to generate image srcsets.
The reason there are two separate lists is that imageSizes is only used for images which provide a sizes prop, which indicates that the image is less than the full width of the screen. Therefore, the sizes in imageSizes should all be smaller than the smallest size in deviceSizes.
If no configuration is provided, the default below is used.

```
next.config.js module.exports = {
  images: {
    imageSizes: [16, 32, 48, 64, 96, 128, 256, 384],
  },
}
```

formats

The default Image Optimization API will automatically detect the browser's supported image formats via the request's Accept header.
If the Accept head matches more than one of the configured formats, the first match in the array is used. Therefore, the array order matters. If there is no match (or the source image is animated), the Image Optimization API will fallback to the original image's format.
If no configuration is provided, the default below is used.

```
next.config.js module.exports = {
```

```
  images: {
    formats: ['image/webp'],
  },
}
```

You can enable AVIF support with the following configuration.

```
next.config.js module.exports = {
  images: {
    formats: ['image/avif', 'image/webp'],
  },
}
```

Good to know:

AVIF generally takes 20% longer to encode but it compresses 20% smaller compared to WebP. This means that the first time an image is requested, it will typically be slower and then subsequent requests that are cached will be faster.
If you self-host with a Proxy/CDN in front of Next.js, you must configure the Proxy to forward the Accept header.

Caching Behavior

The following describes the caching algorithm for the default loader. For all other loaders, please refer to your cloud provider's documentation.
Images are optimized dynamically upon request and stored in the <distDir>/cache/ images directory. The optimized image file will be served for subsequent requests until the expiration is reached. When a request is made that matches a cached but expired file, the expired image is served stale immediately. Then the image is optimized again in the background (also called revalidation) and saved to the cache with the new expiration date.
The cache status of an image can be determined by reading the value of the x-nextjs-cache response header. The possible values are the following:

MISS - the path is not in the cache (occurs at most once, on the first visit)
STALE - the path is in the cache but exceeded the revalidate time so it will be updated in the background
HIT - the path is in the cache and has not exceeded the revalidate time

The expiration (or rather Max Age) is defined by either the minimumCacheTTL configuration or the upstream image Cache-Control header, whichever is larger. Specifically, the max-age value of the Cache-Control header is used. If both s-maxage and max-age are found, then s-maxage is preferred. The max-age is also passed-

through to any downstream clients including CDNs and browsers.

You can configure minimumCacheTTL to increase the cache duration when the upstream image does not include Cache-Control header or the value is very low.
You can configure deviceSizes and imageSizes to reduce the total number of possible generated images.
You can configure formats to disable multiple formats in favor of a single image format.

minimumCacheTTL

You can configure the Time to Live (TTL) in seconds for cached optimized images. In many cases, it's better to use a Static Image Import which will automatically hash the file contents and cache the image forever with a Cache-Control header of immutable.
next.config.js module.exports = {
  images: {
    minimumCacheTTL: 60,
  },
}
The expiration (or rather Max Age) of the optimized image is defined by either the minimumCacheTTL or the upstream image Cache-Control header, whichever is larger.
If you need to change the caching behavior per image, you can configure headers to set the Cache-Control header on the upstream image (e.g. /some-asset.jpg, not /_next/image itself).
There is no mechanism to invalidate the cache at this time, so its best to keep minimumCacheTTL low. Otherwise you may need to manually change the src prop or delete <distDir>/cache/images.
disableStaticImages

The default behavior allows you to import static files such as import icon from './icon.png and then pass that to the src property.
In some cases, you may wish to disable this feature if it conflicts with other plugins that expect the import to behave differently.
You can disable static image imports inside your next.config.js:
next.config.js module.exports = {
  images: {
    disableStaticImages: true,
  },
}

dangerouslyAllowSVG

The default loader does not optimize SVG images for a few reasons. First, SVG is a vector format meaning it can be resized losslessly. Second, SVG has many of the same features as HTML/CSS, which can lead to vulnerabilities without proper Content Security Policy (CSP) headers.
If you need to serve SVG images with the default Image Optimization API, you can set dangerouslyAllowSVG inside your next.config.js:
next.config.js module.exports = {
```
  images: {
    dangerouslyAllowSVG: true,
    contentDispositionType: 'attachment',
    contentSecurityPolicy: "default-src 'self'; script-src 'none'; sandbox;",
  },
}
```
In addition, it is strongly recommended to also set contentDispositionType to force the browser to download the image, as well as contentSecurityPolicy to prevent scripts embedded in the image from executing.
Animated Images

The default loader will automatically bypass Image Optimization for animated images and serve the image as-is.
Auto-detection for animated files is best-effort and supports GIF, APNG, and WebP. If you want to explicitly bypass Image Optimization for a given animated image, use the unoptimized prop.
Known Browser Bugs

This next/image component uses browser native lazy loading, which may fallback to eager loading for older browsers before Safari 15.4. When using the blur-up placeholder, older browsers before Safari 12 will fallback to empty placeholder. When using styles with width/height of auto, it is possible to cause Layout Shift on older browsers before Safari 15 that don't preserve the aspect ratio. For more details, see this MDN video.

Safari 15 and 16 display a gray border while loading. Safari 16.4 fixed this issue. Possible solutions:

Use CSS @supports (font: -apple-system-body) and (-webkit-appearance: none) { img[loading="lazy"] { clip-path: inset(0.6px) } }
Use priority if the image is above the fold

Firefox 67+ displays a white background while loading. Possible solutions:

Enable AVIF formats
Use placeholder="blur"

Version History

VersionChangesv13.2.0contentDispositionType configuration added.v13.0.6ref prop added.v13.0.0The next/image import was renamed to next/legacy/image. The next/future/image import was renamed to next/image. A codemod is available to safely and automatically rename your imports. wrapper removed. layout, objectFit, objectPosition, lazyBoundary, lazyRoot props removed. alt is required. onLoadingComplete receives reference to img element. Built-in loader config removed.v12.3.0remotePatterns and unoptimized configuration is stable.v12.2.0Experimental remotePatterns and experimental unoptimized configuration added. layout="raw" removed.v12.1.1style prop added. Experimental support for layout="raw" added.v12.1.0dangerouslyAllowSVG and contentSecurityPolicy configuration added.v12.0.9lazyRoot prop added.v12.0.0formats configuration added.AVIF support added.Wrapper changed to .v11.1.0onLoadingComplete and lazyBoundary props added.v11.0.0src prop support for static import.placeholder prop added.blurDataURL prop added.v10.0.5loader prop added.v10.0.1layout prop added.v10.0.0next/image introduced.
<Link>
Examples
Hello World
Active className on Link

<Link> is a React component that extends the HTML <a> element to provide prefetching and client-side navigation between routes. It is the primary way to navigate between routes in Next.js.
app/page.tsxTypeScript import Link from 'next/link'

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

Props

Here's a summary of the props available for the Link Component:
PropExampleTypeRequiredhrefhref="/dashboard"String or ObjectYesreplacereplace={false}Boolean-prefetchprefetch={false}Boolean-

Good to know: <a> tag attributes such as className or target="_blank" can be added to <Link> as props and will be passed to the underlying <a> element.

href (required)

The path or URL to navigate to.
```
 <Link href="/dashboard">Dashboard</Link>
```
href can also accept an object, for example:
```
 // Navigate to /about?name=test
<Link
  href={{
    pathname: '/about',
    query: { name: 'test' },
  }}
>
  About
</Link>
```
replace

Defaults to false. When true, next/link will replace the current history state instead of adding a new URL into the browser's history stack.
app/page.tsxTypeScript import Link from 'next/link'

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

prefetch

Defaults to true. When true, next/link will prefetch the page (denoted by the href) in the background. This is useful for improving the performance of client-side navigations. Any <Link /> in the viewport (initially or through scroll) will be preloaded.
Prefetch can be disabled by passing prefetch={false}. Prefetching is only enabled in production.
app/page.tsxTypeScript import Link from 'next/link'

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

Examples

Linking to Dynamic Routes

For dynamic routes, it can be handy to use template literals to create the link's path.

For example, you can generate a list of links to the dynamic route app/blog/[slug]/page.js:app/blog/page.js import Link from 'next/link'

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

Middleware

It's common to use Middleware for authentication or other purposes that involve rewriting the user to a different page. In order for the <Link /> component to properly prefetch links with rewrites via Middleware, you need to tell Next.js both the URL to display and the URL to prefetch. This is required to avoid un-necessary fetches to middleware to know the correct route to prefetch.
For example, if you want to serve a /dashboard route that has authenticated and visitor views, you may add something similar to the following in your Middleware to redirect the user to the correct page:
middleware.js export function middleware(req) {

```
  const nextUrl = req.nextUrl
  if (nextUrl.pathname === '/dashboard') {
    if (req.cookies.authToken) {
      return NextResponse.rewrite(new URL('/auth/dashboard', req.url))
    } else {
      return NextResponse.rewrite(new URL('/public/dashboard', req.url))
    }
  }
}
```

In this case, you would want to use the following code in your <Link /> component:

```
 import Link from 'next/link'
import useIsAuthed from './hooks/useIsAuthed'

export default function Page() {
  const isAuthed = useIsAuthed()
  const path = isAuthed ? '/auth/dashboard' : '/dashboard'
  return (
    <Link as="/dashboard" href={path}>
```

```
      Dashboard
    </Link>
  )
}
```

Version History

VersionChangesv13.0.0No longer requires a child <a> tag. A codemod is provided to automatically update your codebase.v10.0.0href props pointing to a dynamic route are automatically resolved and no longer require an as prop.v8.0.0Improved prefetching performance.v1.0.0next/link introduced.
<Script>
This API reference will help you understand how to use props available for the Script Component. For features and usage, please see the Optimizing Scripts page.
app/dashboard/page.tsxTypeScript import Script from 'next/script'

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

Props

Here's a summary of the props available for the Script Component:
PropExampleTypeRequiredsrcsrc="http://example.com/script"StringRequired unless inline script is usedstrategystrategy="lazyOnload"String-onLoadonLoad={onLoadFunc} Function-onReadyonReady={onReadyFunc}Function-onErroronError={onErrorFunc} Function-
Required Props

The <Script /> component requires the following properties.

src

A path string specifying the URL of an external script. This can be either an absolute external URL or an internal path. The src property is required unless an inline script is used.
Optional Props

The <Script /> component accepts a number of additional properties beyond those which are required.
strategy

The loading strategy of the script. There are four different strategies that can be used:

beforeInteractive: Load before any Next.js code and before any page hydration occurs.
afterInteractive: (default) Load early but after some hydration on the page occurs.
lazyOnload: Load during browser idle time.
worker: (experimental) Load in a web worker.

beforeInteractive

Scripts that load with the beforeInteractive strategy are injected into the initial HTML from the server, downloaded before any Next.js module, and executed in the order they are placed before any hydration occurs on the page.
Scripts denoted with this strategy are preloaded and fetched before any first-party code, but their execution does not block page hydration from occurring.
beforeInteractive scripts must be placed inside the root layout (app/layout.tsx) and are designed to load scripts that are needed by the entire site (i.e. the script will load when any page in the application has been loaded server-side).
This strategy should only be used for critical scripts that need to be fetched before any part of the page becomes interactive.

app/layout.tsxTypeScript import Script from 'next/script'

```
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
      <Script
        src="https://example.com/script.js"
        strategy="beforeInteractive"
      />
    </html>
  )
}
```

Good to know: Scripts with beforeInteractive will always be injected inside the head of the HTML document regardless of where it's placed in the component.

Some examples of scripts that should be loaded as soon as possible with beforeInteractive include:

Bot detectors
Cookie consent managers

afterInteractive

Scripts that use the afterInteractive strategy are injected into the HTML client-side and will load after some (or all) hydration occurs on the page. This is the default strategy of the Script component and should be used for any script that needs to load as soon as possible but not before any first-party Next.js code.
afterInteractive scripts can be placed inside of any page or layout and will only load and execute when that page (or group of pages) is opened in the browser.
app/page.js import Script from 'next/script'

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

```
    </>
  )
}
```
Some examples of scripts that are good candidates for afterInteractive include:

Tag managers
Analytics

lazyOnload

Scripts that use the lazyOnload strategy are injected into the HTML client-side during browser idle time and will load after all resources on the page have been fetched. This strategy should be used for any background or low priority scripts that do not need to load early.
lazyOnload scripts can be placed inside of any page or layout and will only load and execute when that page (or group of pages) is opened in the browser.
app/page.js import Script from 'next/script'

```
export default function Page() {
  return (
    <>
      <Script src="https://example.com/script.js" strategy="lazyOnload" />
    </>
  )
}
```
Examples of scripts that do not need to load immediately and can be fetched with lazyOnload include:

Chat support plugins
Social media widgets

worker

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

Scripts that use the worker strategy are off-loaded to a web worker in order to free up

the main thread and ensure that only critical, first-party resources are processed on it. While this strategy can be used for any script, it is an advanced use case that is not guaranteed to support all third-party scripts.
To use worker as a strategy, the nextScriptWorkers flag must be enabled in next.config.js:
next.config.js module.exports = {
  experimental: {
    nextScriptWorkers: true,
  },
}
worker scripts can only currently be used in the pages/ directory:
pages/home.tsxTypeScript import Script from 'next/script'

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

onLoad

Warning: onLoad does not yet work with Server Components and can only be used in Client Components. Further, onLoad can't be used with beforeInteractive – consider using onReady instead.

Some third-party scripts require users to run JavaScript code once after the script has finished loading in order to instantiate content or call a function. If you are loading a script with either afterInteractive or lazyOnload as a loading strategy, you can execute code after it has loaded using the onLoad property.
Here's an example of executing a lodash method only after the library has been loaded.
app/page.tsxTypeScript 'use client'

import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.20/lodash.min.js"

```
        onLoad={() => {
          console.log(_.sample([1, 2, 3, 4]))
        }}
      />
    </>
  )
}
```

onReady

Warning: onReady does not yet work with Server Components and can only be used in Client Components.

Some third-party scripts require users to run JavaScript code after the script has finished loading and every time the component is mounted (after a route navigation for example). You can execute code after the script's load event when it first loads and then after every subsequent component re-mount using the onReady property.
Here's an example of how to re-instantiate a Google Maps JS embed every time the component is mounted:
app/page.tsxTypeScript 'use client'

```
import { useRef } from 'react'
import Script from 'next/script'

export default function Page() {
  const mapRef = useRef()

  return (
    <>
      <div ref={mapRef}></div>
      <Script
        id="google-maps"
        src="https://maps.googleapis.com/maps/api/js"
        onReady={() => {
          new google.maps.Map(mapRef.current, {
            center: { lat: -34.397, lng: 150.644 },
            zoom: 8,
          })
        }}
      />
    </>
```

```
  )
}
```

onError

Warning: onError does not yet work with Server Components and can only be used in Client Components. onError cannot be used with the beforeInteractive loading strategy.

Sometimes it is helpful to catch when a script fails to load. These errors can be handled with the onError property:
app/page.tsxTypeScript 'use client'

```
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        onError={(e: Error) => {
          console.error('Script failed to load', e)
        }}
      />
    </>
  )
}
```

Version History

VersionChangesv13.0.0beforeInteractive and afterInteractive is modified to support app.v12.2.4onReady prop added.v12.2.2Allow next/script with beforeInteractive to be placed in _document.v11.0.0next/script introduced.
File Conventions
default.jsThis documentation is still being written. Please check back later.
error.jsAn error file defines an error UI boundary for a route segment.
It is useful for catching unexpected errors that occur in Server Components and Client Components and displaying a fallback UI.

app/dashboard/error.tsxTypeScript 'use client' // Error components must be Client Components

```
import { useEffect } from 'react'

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

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

Props

error

An instance of an Error object forwarded to the error.js Client Component.
error.message

The error message.

For errors forwarded from Client Components, this will be the original Error's message. For errors forwarded from Server Components, this will be a generic error message to avoid leaking sensitive details. errors.digest can be used to match the corresponding error in server-side logs.

error.digest

An automatically generated hash of the error thrown in a Server Component. It can be used to match the corresponding error in server-side logs.
reset

A function to reset the error boundary. When executed, the function will try to re-render the Error boundary's contents. If successful, the fallback error component is replaced with the result of the re-render.
Can be used to prompt the user to attempt to recover from the error.

Good to know:

error.js boundaries must be Client Components.
In Production builds, errors forwarded from Server Components will be stripped of specific error details to avoid leaking sensitive information.
An error.js boundary will not handle errors thrown in a layout.js component in the same segment because the error boundary is nested inside that layouts component.

To handle errors for a specific layout, place an error.js file in the layouts parent segment.
To handle errors within the root layout or template, use a variation of error.js called app/global-error.js.

global-error.js

To specifically handle errors in root layout.js, use a variation of error.js called app/global-error.js located in the root app directory.
app/global-error.tsxTypeScript 'use client'

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

Good to know:

global-error.js replaces the root layout.js when active and so must define its own <html> and <body> tags.
While designing error UI, you may find it helpful to use the React Developer Tools to manually toggle Error boundaries.

Version History

VersionChangesv13.1.0global-error introduced.v13.0.0error introduced.
layout.jsA layout is UI that is shared between routes.
app/dashboard/layout.tsxTypeScript export default function DashboardLayout({
  children,

```
}: {
  children: React.ReactNode
}) {
  return <section>{children}</section>
}
```

A root layout is the top-most layout in the root app directory. It is used to define the <html> and <body> tags and other globally shared UI.
app/layout.tsxTypeScript export default function RootLayout({

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

Props

children (required)

Layout components should accept and use a children prop. During rendering, children will be populated with the route segments the layout is wrapping. These will primarily be the component of a child Layout (if it exists) or Page, but could also be other special files like Loading or Error when applicable.
params (optional)

The dynamic route parameters object from the root segment down to that layout. ExampleURLparamsapp/dashboard/[team]/layout.js/dashboard/1{ team: '1' }app/shop/[tag]/[item]/layout.js/shop/1/2{ tag: '1', item: '2' }app/blog/[...slug]/layout.js/blog/1/2{ slug: ['1', '2'] }

For example:
app/shop/[tag]/[item]/layout.tsxTypeScript export default function ShopLayout({
  children,
  params,
}: {
  children: React.ReactNode
  params: {
    tag: string
    item: string
  }
}) {
  // URL -> /shop/shoes/nike-air-max-97
  // `params` -> { tag: 'shoes', item: 'nike-air-max-97' }
  return <section>{children}</section>
}

Good to know

Layout's do not receive searchParams

Unlike Pages, Layout components do not receive the searchParams prop. This is because a shared layout is not re-rendered during navigation which could lead to stale searchParams between navigations.
When using client-side navigation, Next.js automatically only renders the part of the page below the common layout between two routes.
For example, in the following directory structure, dashboard/layout.tsx is the common layout for both /dashboard/settings and /dashboard/analytics:
 app
% % % dashboard
    % % % layout.tsx
    % % % settings
    %   % % % page.tsx
    % % % analytics
        % % % page.js
When navigating from /dashboard/settings to /dashboard/analytics, page.tsx in /dashboard/analytics will be rendered on the server because it is UI that changed, while dashboard/layout.tsx will not be re-rendered because it is a common UI between the two routes.

This performance optimization allows navigation between pages that share a layout to be quicker as only the data fetching and rendering for the page has to run, instead of the entire route that could include shared layouts that fetch their own data.
Because dashboard/layout.tsx doesn't re-render, the searchParams prop in the layout Server Component might become stale after navigation.

Instead, use the Page searchParams prop or the useSearchParams hook in a Client Component, which is re-rendered on the client with the latest searchParams.

Root Layouts

The app directory must include a root app/layout.js.
The root layout must define <html> and <body> tags.

You should not manually add <head> tags such as <title> and <meta> to root layouts. Instead, you should use the Metadata API which automatically handles advanced requirements such as streaming and de-duplicating <head> elements.

You can use route groups to create multiple root layouts.

Navigating across multiple root layouts will cause a full page load (as opposed to a client-side navigation). For example, navigating from /cart that uses app/(shop)/layout.js to /blog that uses app/(marketing)/layout.js will cause a full page load. This only applies to multiple root layouts.

Version History

VersionChangesv13.0.0layout introduced.
loading.jsA loading file can create instant loading states built on Suspense.
By default, this file is a Server Component - but can also be used as a Client Component through the "use client" directive.
app/feed/loading.tsxTypeScript export default function Loading() {
  // Or a custom loading skeleton component
  return 'Loading...'

}

Loading UI components do not accept any parameters.

Good to know

While designing loading UI, you may find it helpful to use the React Developer Tools to manually toggle Suspense boundaries.

Version History

VersionChangesv13.0.0loading introduced.
not-found.jsThe not-found file is used to render UI when the notFound function is thrown within a route segment. Along with serving a custom UI, Next.js will also return a 404 HTTP status code.
app/blog/not-found.tsxTypeScript import Link from 'next/link'

```
export default function NotFound() {
  return (
    <div>
      <h2>Not Found</h2>
      <p>Could not find requested resource</p>
      <p>
        View <Link href="/blog">all posts</Link>
      </p>
    </div>
  )
}
```

Good to know: In addition to catching expected notFound() errors, the root app/not-found.js file also handles any unmatched URLs for your whole application. This means users that visit a URL that is not handled by your app will be shown the UI exported by the app/not-found.js file.

Props

not-found.js components do not accept any props.
Version History

VersionChangesv13.3.0Root app/not-found handles global unmatched URLs.v13.0.0not-found introduced.
page.jsA page is UI that is unique to a route.
app/blog/[slug]/page.tsxTypeScript export default function Page({
  params,
  searchParams,
}: {
  params: { slug: string }
  searchParams: { [key: string]: string | string[] | undefined }
}) {
  return <h1>My Page</h1>
}

Props

params (optional)

An object containing the dynamic route parameters from the root segment down to that page. For example:
ExampleURLparamsapp/shop/[slug]/page.js/shop/1{ slug: '1' }app/shop/[category]/[item]/page.js/shop/1/2{ category: '1', item: '2' }app/shop/[...slug]/page.js/shop/1/2{ slug: ['1', '2'] }
searchParams (optional)

An object containing the search parameters of the current URL. For example:
URLsearchParams/shop?a=1{ a: '1' }/shop?a=1&b=2{ a: '1', b: '2' }/shop?a=1&a=2{ a: ['1', '2'] }

Good to know:

searchParams is a Dynamic API whose values cannot be known ahead of time. Using it will opt the page into dynamic rendering at request time.
searchParams returns a plain JavaScript object and not a URLSearchParams instance.

Version History

VersionChangesv13.0.0page introduced.
route.jsRoute Handlers allow you to create custom request handlers for a given route using the Web Request and Response APIs.
HTTP Methods

A route file allows you to create custom request handlers for a given route. The following HTTP methods are supported: GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS.
route.tsTypeScript export async function GET(request: Request) {}

export async function HEAD(request: Request) {}

export async function POST(request: Request) {}

export async function PUT(request: Request) {}

export async function DELETE(request: Request) {}

export async function PATCH(request: Request) {}

// If `OPTIONS` is not defined, Next.js will automatically implement `OPTIONS` and set the appropriate Response `Allow` header depending on the other methods defined in the route handler.
export async function OPTIONS(request: Request) {}

Good to know: Route Handlers are only available inside the app directory. You do not need to use API Routes (pages) and Route Handlers (app) together, as Route

Handlers should be able to handle all use cases.

Parameters

request (optional)

The request object is a NextRequest object, which is an extension of the Web Request API. NextRequest gives you further control over the incoming request, including easily accessing cookies and an extended, parsed, URL object nextUrl.
context (optional)

app/dashboard/[team]/route.js export async function GET(request, context: { params }) {
  const team = params.team // '1'
}
Currently, the only value of context is params, which is an object containing the dynamic route parameters for the current route.
ExampleURLparamsapp/dashboard/[team]/route.js/dashboard/1{ team: '1' }app/shop/[tag]/[item]/route.js/shop/1/2{ tag: '1', item: '2' }app/blog/[...slug]/route.js/blog/1/2{ slug: ['1', '2'] }
NextResponse

Route Handlers can extend the Web Response API by returning a NextResponse object. This allows you to easily set cookies, headers, redirect, and rewrite. View the API reference.
Version History

VersionChangesv13.2.0Route handlers are introduced.
Route Segment ConfigThe Route Segment options allows you configure the behavior of a Page, Layout, or Route Handler by directly exporting the following variables:
OptionTypeDefaultdynamic'auto' | 'force-dynamic' | 'error' | 'force-static''auto'dynamicParamsbooleantruerevalidatefalse | 'force-cache' | 0 | numberfalsefetchCache'auto' | 'default-cache' | 'only-cache' | 'force-cache' | 'force-no-store' | 'default-no-store' | 'only-no-store''auto'runtime'nodejs' | 'edge''nodejs'preferredRegion'auto' | 'global' | 'home' | string | string[]'auto'
layout.tsx / page.tsx / route.tsTypeScript export const dynamic = 'auto'
export const dynamicParams = true
export const revalidate = false
export const fetchCache = 'auto'
export const runtime = 'nodejs'
export const preferredRegion = 'auto'

export default function MyComponent() {}

Good to know:

The values of the config options currently need be statically analyzable. For example revalidate = 600 is valid, but revalidate = 60 * 10 is not.

Options

dynamic

Change the dynamic behavior of a layout or page to fully static or fully dynamic.
layout.tsx / page.tsx / route.tsTypeScript export const dynamic = 'auto'
// 'auto' | 'force-dynamic' | 'error' | 'force-static'

Good to know: The new model in the app directory favors granular caching control at the fetch request level over the binary all-or-nothing model of getServerSideProps and getStaticProps at the page-level in the pages directory. The dynamic option is a way to opt back in to the previous model as a convenience and provides a simpler migration path.

'auto' (default): The default option to cache as much as possible without preventing any components from opting into dynamic behavior.

'force-dynamic': Force dynamic rendering and dynamic data fetching of a layout or page by disabling all caching of fetch requests and always revalidating. This option is equivalent to:

getServerSideProps() in the pages directory.
Setting the option of every fetch() request in a layout or page to { cache: 'no-store', next: { revalidate: 0 } }.
Setting the segment config to export const fetchCache = 'force-no-store'

'error': Force static rendering and static data fetching of a layout or page by causing an error if any components use dynamic functions or dynamic fetches. This option is equivalent to:

getStaticProps() in the pages directory.
Setting the option of every fetch() request in a layout or page to { cache: 'force-cache' }.
Setting the segment config to fetchCache = 'only-cache', dynamicParams = false.
dynamic = 'error' changes the default of dynamicParams from true to false. You can opt back into dynamically rendering pages for dynamic params not generated by generateStaticParams by manually setting dynamicParams = true.

'force-static': Force static rendering and static data fetching of a layout or page by forcing cookies(), headers() and useSearchParams() to return empty values.

Good to know:

Instructions on how to migrate from getServerSideProps and getStaticProps to dynamic: 'force-dynamic' and dynamic: 'error' can be found in the upgrade guide.

dynamicParams

Control what happens when a dynamic segment is visited that was not generated with generateStaticParams.
layout.tsx / page.tsxTypeScript export const dynamicParams = true // true | false,

true (default): Dynamic segments not included in generateStaticParams are generated on demand.
false: Dynamic segments not included in generateStaticParams will return a 404.

Good to know:

This option replaces the fallback: true | false | blocking option of getStaticPaths in the pages directory.
When dynamicParams = true, the segment uses Streaming Server Rendering.
If the dynamic = 'error' and dynamic = 'force-static' are used, it'll change the default of dynamicParams to false.

revalidate

Set the default revalidation time for a layout or page. This option does not override the revalidate value set by individual fetch requests.
layout.tsx / page.tsx / route.tsTypeScript export const revalidate = false
// false | 'force-cache' | 0 | number

false: (default) The default heuristic to cache any fetch requests that set their cache option to 'force-cache' or are discovered before a dynamic function is used.
Semantically equivalent to revalidate: Infinity which effectively means the resource should be cached indefinitely. It is still possible for individual fetch requests to use cache: 'no-store' or revalidate: 0 to avoid being cached and make the route dynamically rendered. Or set revalidate to a positive number lower than the route default to increase the revalidation frequency of a route.
0: Ensure a layout or page is always dynamically rendered even if no dynamic functions or dynamic data fetches are discovered. This option changes the default of fetch requests that do not set a cache option to 'no-store' but leaves fetch requests that opt into 'force-cache' or use a positive revalidate as is.
number: (in seconds) Set the default revalidation frequency of a layout or page to n seconds.

Revalidation Frequency

The lowest revalidate across each layout and page of a single route will determine the revalidation frequency of the entire route. This ensures that child pages are revalidated as frequently as their parent layouts.
Individual fetch requests can set a lower revalidate than the route's default revalidate to increase the revalidation frequency of the entire route. This allows you to dynamically opt-in to more frequent revalidation for certain routes based on some criteria.

fetchCache

This is an advanced option that should only be used if you specifically need to override the default behavior.By default, Next.js will cache any fetch() requests that are reachable before any dynamic functions are used and will not cache fetch requests that are discovered after dynamic functions are used.fetchCache allows you to override the default cache option of all fetch requests in a layout or page.layout.tsx / page.tsx / route.tsTypeScript export const fetchCache = 'auto'
// 'auto' | 'default-cache' | 'only-cache'
// 'force-cache' | 'force-no-store' | 'default-no-store' | 'only-no-store'
'auto' (default)- The default option to cache fetch requests before dynamic functions with the cache option they provide and not cache fetch requests after dynamic functions.
'default-cache': Allow any cache option to be passed to fetch but if no option is provided then set the cache option to 'force-cache'. This means that even fetch requests after dynamic functions are considered static.
'only-cache': Ensure all fetch requests opt into caching by changing the default to cache: 'force-cache' if no option is provided and causing an error if any fetch requests use cache: 'no-store'.
'force-cache': Ensure all fetch requests opt into caching by setting the cache option of all fetch requests to 'force-cache'.
'default-no-store': Allow any cache option to be passed to fetch but if no option is provided then set the cache option to 'no-store'. This means that even fetch requests before dynamic functions are considered dynamic.
'only-no-store': Ensure all fetch requests opt out of caching by changing the default to cache: 'no-store' if no option is provided and causing an error if any fetch requests use cache: 'force-cache'
'force-no-store': Ensure all fetch requests opt out of caching by setting the cache option

of all fetch requests to 'no-store'. This forces all fetch requests to be re-fetched every request even if they provide a 'force-cache' option.
Cross-route segment behavior

Any options set across each layout and page of a single route need to be compatible with each other.

If both the 'only-cache' and 'force-cache' are provided, then 'force-cache' wins. If both 'only-no-store' and 'force-no-store' are provided, then 'force-no-store' wins. The force option changes the behavior across the route so a single segment with 'force-*' would prevent any errors caused by 'only-*'.
The intention of the 'only-*' and force-*' options is to guarantee the whole route is either fully static or fully dynamic. This means:

A combination of 'only-cache' and 'only-no-store' in a single route is not allowed.
A combination of 'force-cache' and 'force-no-store' in a single route is not allowed.

A parent cannot provide 'default-no-store' if a child provides 'auto' or '*-cache' since that could make the same fetch have different behavior.

It is generally recommended to leave shared parent layouts as 'auto' and customize the options where child segments diverge.

runtime

layout.tsx / page.tsx / route.tsTypeScript export const runtime = 'nodejs'
// 'edge' | 'nodejs'

nodejs (default)
edge

Learn more about the Edge and Node.js runtimes.
preferredRegion

layout.tsx / page.tsx / route.tsTypeScript export const preferredRegion = 'auto'
// 'auto' | 'global' | 'home' | ['iad1', 'sfo1']

Support for preferredRegion, and regions supported, is dependent on your deployment platform.

Good to know:

If a preferredRegion is not specified, it will inherit the option of the nearest parent layout. The root layout defaults to all regions.

generateStaticParams

The generateStaticParams function can be used in combination with dynamic route segments to define the list of route segment parameters that will be statically generated at build time instead of on-demand at request time.
See the API reference for more details.
template.jsThis documentation is still being written. Please check back later.
Metadata Files API ReferenceThis section of the docs covers Metadata file conventions. File-based metadata can be defined by adding special metadata files to route segments.
Each file convention can be defined using a static file (e.g. opengraph-image.jpg), or a dynamic variant that uses code to generate the file (e.g. opengraph-image.js).
Once a file is defined, Next.js will automatically serve the file (with hashes in production for caching) and update the relevant head elements with the correct metadata, such as the asset's URL, file type, and image size.
favicon, icon, and apple-iconThe favicon, icon, or apple-icon file conventions allow you to set icons for your application.
They are useful for adding app icons that appear in places like web browser tabs, phone home screens, and search engine results.
There are two ways to set app icons:

Using image files (.ico, .jpg, .png)
Using code to generate an icon (.js, .ts, .tsx)

Image files (.ico, .jpg, .png)

Use an image file to set an app icon by placing a favicon, icon, or apple-icon image file within your /app directory.
The favicon image can only be located in the top level of app/.
Next.js will evaluate the file and automatically add the appropriate tags to your app's <head> element.
File conventionSupported file typesValid locationsfavicon.icoapp/
icon.ico, .jpg, .jpeg, .png, .svgapp/**/*apple-icon.jpg, .jpeg, .pngapp/**/*
favicon

Add a favicon.ico image file to the root /app route segment.
<head> output <link rel="icon" href="/favicon.ico" sizes="any" />
icon

Add an icon.(ico|jpg|jpeg|png|svg) image file.
<head> output <link
  rel="icon"
  href="/icon?<generated>"
  type="image/<generated>"
  sizes="<generated>"
/>
apple-icon

Add an apple-icon.(jpg|jpeg|png) image file.
<head> output <link
  rel="apple-touch-icon"
  href="/apple-icon?<generated>"
  type="image/<generated>"
  sizes="<generated>"
/>

Good to know

You can set multiple icons by adding a number suffix to the file name. For example, icon1.png, icon2.png, etc. Numbered files will sort lexically.
Favicons can only be set in the root /app segment. If you need more granularity, you can use icon.
The appropriate <link> tags and attributes such as rel, href, type, and sizes are determined by the icon type and metadata of the evaluated file.

For example, a 32 by 32px .png file will have type="image/png" and sizes="32x32" attributes.

sizes="any" is added to favicon.ico output to avoid a browser bug where an .ico icon is favored over .svg.

Generate icons using code (.js, .ts, .tsx)

In addition to using literal image files, you can programmatically generate icons using code.
Generate an app icon by creating an icon or apple-icon route that default exports a function.
File conventionSupported file typesicon.js, .ts, .tsxapple-icon.js, .ts, .tsx
The easiest way to generate an icon is to use the ImageResponse API from next/server.
app/icon.tsxTypeScript import { ImageResponse } from 'next/server'

```
// Route segment config
export const runtime = 'edge'

// Image metadata
export const size = {
  width: 32,
  height: 32,
}
export const contentType = 'image/png'

// Image generation
export default function Icon() {
  return new ImageResponse(
    (
      // ImageResponse JSX element
      <div
```

```
        style={{
          fontSize: 24,
          background: 'black',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
          color: 'white',
        }}
      >
        A
      </div>
    ),
    // ImageResponse options
    {
      // For convenience, we can re-use the exported icons size metadata
      // config to also set the ImageResponse's width and height.
      ...size,
    }
  )
}
```

<head> output <link rel="icon" href="/icon?<generated>" type="image/png" sizes="32x32" />

Good to know

By default, generated icons are statically optimized (generated at build time and cached) unless they use dynamic functions or dynamic data fetching.
You can generate multiple icons in the same file using generateImageMetadata.
You cannot generate a favicon icon. Use icon or a favicon.ico file instead.

Props

The default export function receives the following props:
params (optional)

An object containing the dynamic route parameters object from the root segment down to the segment icon or apple-icon is colocated in.
app/shop/[slug]/icon.tsxTypeScript export default function Icon({ params }: { params: { slug: string } }) {
  // ...
}

RouteURLparamsapp/shop/icon.js/shopundefinedapp/shop/[slug]/icon.js/shop/1{ slug: '1' }app/shop/[tag]/[item]/icon.js/shop/1/2{ tag: '1', item: '2' }app/shop/[...slug]/icon.js/shop/1/2{ slug: ['1', '2'] }
Returns

The default export function should return a Blob | ArrayBuffer | TypedArray | DataView | ReadableStream | Response.

Good to know: ImageResponse satisfies this return type.

Config exports

You can optionally configure the icon's metadata by exporting size and contentType variables from the icon or apple-icon route.
OptionTypesize{ width: number; height: number }contentTypestring - image MIME type size

icon.tsx / apple-icon.tsxTypeScript export const size = { width: 32, height: 32 }

export default function Icon() {}

<head> output <link rel="icon" sizes="32x32" />
contentType

icon.tsx / apple-icon.tsxTypeScript export const contentType = 'image/png'

export default function Icon() {}

<head> output <link rel="icon" type="image/png" />
Route Segment Config

icon and apple-icon are specialized Route Handlers that can use the same route segment configuration options as Pages and Layouts.
OptionTypeDefaultdynamic'auto' | 'force-dynamic' | 'error' | 'force-static''auto'revalidatefalse | 'force-cache' | 0 | numberfalseruntime'nodejs' | 'edge''nodejs'preferredRegion'auto' | 'global' | 'home' | string | string[]'auto'
app/icon.tsxTypeScript export const runtime = 'edge'

export default function Icon() {}

Version History

VersionChangesv13.3.0favicon icon and apple-icon introduced
opengraph-image and twitter-imageThe opengraph-image and twitter-image file conventions allow you to set Open Graph and Twitter images for a route segment. They are useful for setting the images that appear on social networks and messaging apps when a user shares a link to your site.
There are two ways to set Open Graph and Twitter images:

Using image files (.jpg, .png, .gif)
Using code to generate images (.js, .ts, .tsx)

Image files (.jpg, .png, .gif)

Use an image file to set a route segment's shared image by placing an opengraph-image or twitter-image image file in the segment.

Next.js will evaluate the file and automatically add the appropriate tags to your app's <head> element.
File conventionSupported file typesopengraph-image.jpg, .jpeg, .png, .giftwitter-image.jpg, .jpeg, .png, .gifopengraph-image.alt.txttwitter-image.alt.txt
opengraph-image

Add an opengraph-image.(jpg|jpeg|png|gif) image file to any route segment.
<head> output <meta property="og:image" content="<generated>" />
<meta property="og:image:type" content="<generated>" />
<meta property="og:image:width" content="<generated>" />
<meta property="og:image:height" content="<generated>" />
twitter-image

Add a twitter-image.(jpg|jpeg|png|gif) image file to any route segment.
<head> output <meta name="twitter:image" content="<generated>" />
<meta name="twitter:image:type" content="<generated>" />
<meta name="twitter:image:width" content="<generated>" />
<meta name="twitter:image:height" content="<generated>" />
opengraph-image.alt.txt

Add an accompanying opengraph-image.alt.txt file in the same route segment as the opengraph-image.(jpg|jpeg|png|gif) image it's alt text.
opengraph-image.alt.txt About Acme
<head> output <meta property="og:image:alt" content="About Acme" />
twitter-image.alt.txt

Add an accompanying twitter-image.alt.txt file in the same route segment as the twitter-image.(jpg|jpeg|png|gif) image it's alt text.
twitter-image.alt.txt About Acme
<head> output <meta property="og:image:alt" content="About Acme" />

Generate images using code (.js, .ts, .tsx)

In addition to using literal image files, you can programmatically generate images using code.
Generate a route segment's shared image by creating an opengraph-image or twitter-image route that default exports a function.
File conventionSupported file typesopengraph-image.js, .ts, .tsxtwitter-image.js, .ts, .tsx

Good to know

By default, generated images are statically optimized (generated at build time and cached) unless they use dynamic functions or dynamic data fetching.
You can generate multiple Images in the same file using generateImageMetadata.

The easiest way to generate an image is to use the ImageResponse API from next/server.
app/about/opengraph-image.tsxTypeScript import { ImageResponse } from 'next/server'

```
// Route segment config
export const runtime = 'edge'

// Image metadata
export const alt = 'About Acme'
export const size = {
  width: 1200,
  height: 630,
}

export const contentType = 'image/png'

// Font
const interSemiBold = fetch(
  new URL('./Inter-SemiBold.ttf', import.meta.url)
).then((res) => res.arrayBuffer())

// Image generation
export default async function Image() {
  return new ImageResponse(
    (
      // ImageResponse JSX element
      <div
```

```
        style={{
          fontSize: 128,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        About Acme
      </div>
    ),
    // ImageResponse options
    {
      // For convenience, we can re-use the exported opengraph-image
      // size config to also set the ImageResponse's width and height.
      ...size,
      fonts: [
        {
          name: 'Inter',
          data: await interSemiBold,
          style: 'normal',
          weight: 400,
        },
      ],
    }
  )
}
```

```
<head> output <meta property="og:image" content="<generated>" />
<meta property="og:image:alt" content="About Acme" />
<meta property="og:image:type" content="image/png" />
<meta property="og:image:width" content="1200" />
<meta property="og:image:height" content="630" />
```

Props

The default export function receives the following props:

params (optional)

An object containing the dynamic route parameters object from the root segment down to the segment opengraph-image or twitter-image is colocated in.
app/shop/[slug]/opengraph-image.tsxTypeScript export default function Image({ params }: { params: { slug: string } }) {
  // ...
}

RouteURLparamsapp/shop/opengraph-image.js/shopundefinedapp/shop/[slug]/opengraph-image.js/shop/1{ slug: '1' }app/shop/[tag]/[item]/opengraph-image.js/shop/1/2{ tag: '1', item: '2' }app/shop/[...slug]/opengraph-image.js/shop/1/2{ slug: ['1', '2'] }
Returns

The default export function should return a Blob | ArrayBuffer | TypedArray | DataView | ReadableStream | Response.

Good to know: ImageResponse satisfies this return type.

Config exports

You can optionally configure the image's metadata by exporting alt, size, and contentType variables from opengraph-image or twitter-image route.
OptionTypealtstringsize{ width: number; height: number }contentTypestring - image MIME type
alt

opengraph-image.tsx / twitter-image.tsxTypeScript export const alt = 'My images alt text'

export default function Image() {}

<head> output <meta property="og:image:alt" content="My images alt text" />
size

opengraph-image.tsx / twitter-image.tsxTypeScript export const size = { width: 1200, height: 630 }

export default function Image() {}

<head> output <meta property="og:image:width" content="1200" />
<meta property="og:image:height" content="630" />
contentType

opengraph-image.tsx / twitter-image.tsxTypeScript export const contentType = 'image/png'

export default function Image() {}

<head> output <meta property="og:image:type" content="image/png" />
Route Segment Config

opengraph-image and twitter-image are specialized Route Handlers that can use the same route segment configuration options as Pages and Layouts.
OptionTypeDefaultdynamic'auto' | 'force-dynamic' | 'error' | 'force-static''auto'revalidatefalse | 'force-cache' | 0 | numberfalseruntime'nodejs' | 'edge''nodejs'preferredRegion'auto' | 'global' | 'home' | string | string[]'auto'
app/opengraph-image.tsxTypeScript export const runtime = 'edge'

export default function Image() {}

Examples

Using external data

This example uses the params object and external data to generate the image.

Good to know:
By default, this generated image will be statically optimized. You can configure the individual fetch options or route segments options to change this behavior.

app/posts/[slug]/opengraph-image.tsxTypeScript import { ImageResponse } from 'next/server'

```
export const runtime = 'edge'

export const alt = 'About Acme'
export const size = {
  width: 1200,
  height: 630,
}
export const contentType = 'image/png'

export default async function Image({ params }: { params: { slug: string } }) {
  const post = await fetch(`https://.../posts/${params.slug}`).then((res) =>
    res.json()
  )

  return new ImageResponse(
    (
      <div
        style={{
          fontSize: 48,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        {post.title}
      </div>
    ),
    {
      ...size,
```

```
    }
  )
}
```

Version History

VersionChangesv13.3.0opengraph-image and twitter-image introduced.
robots.txtAdd or generate a robots.txt file that matches the Robots Exclusion Standard in the root of app directory to tell search engine crawlers which URLs they can access on your site.
Static robots.txt

app/robots.txt User-Agent: *
Allow: /
Disallow: /private/

Sitemap: https://acme.com/sitemap.xml
Generate a Robots file

Add a robots.js or robots.ts file that returns a Robots object.
app/robots.tsTypeScript import { MetadataRoute } from 'next'

```
export default function robots(): MetadataRoute.Robots {
  return {
    rules: {
      userAgent: '*',
      allow: '/',
      disallow: '/private/',
    },
    sitemap: 'https://acme.com/sitemap.xml',
  }
}
```

Output:

```
 User-Agent: *
Allow: /
Disallow: /private/

Sitemap: https://acme.com/sitemap.xml
```
Robots object

```
 type Robots = {
  rules:
    | {
        userAgent?: string | string[]
        allow?: string | string[]
        disallow?: string | string[]
        crawlDelay?: number
      }
    | Array<{
        userAgent: string | string[]
        allow?: string | string[]
        disallow?: string | string[]
        crawlDelay?: number
      }>
  sitemap?: string | string[]
  host?: string
}
```
Version History

VersionChangesv13.3.0robots introduced.
sitemap.xmlAdd or generate a sitemap.xml file that matches the Sitemaps XML format in the root of app directory to help search engine crawlers crawl your site more efficiently.
Static sitemap.xml

app/sitemap.xml
```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
```

```
    <loc>https://acme.com</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  </url>
  <url>
    <loc>https://acme.com/about</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  </url>
  <url>
    <loc>https://acme.com/blog</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  </url>
</urlset>
```

Generate a Sitemap

Add a sitemap.js or sitemap.ts file that returns Sitemap.

app/sitemap.tsTypeScript

```
import { MetadataRoute } from 'next'

export default function sitemap(): MetadataRoute.Sitemap {
  return [
    {
      url: 'https://acme.com',
      lastModified: new Date(),
    },
    {
      url: 'https://acme.com/about',
      lastModified: new Date(),
    },
    {
      url: 'https://acme.com/blog',
      lastModified: new Date(),
    },
  ]
}
```

Output:

acme.com/sitemap.xml

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://acme.com</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  </url>
  <url>
    <loc>https://acme.com/about</loc>
```

```
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  </url>
  <url>
    <loc>https://acme.com/blog</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
  </url>
</urlset>
```

Sitemap Return Type

```
 type Sitemap = Array<{
  url: string
  lastModified?: string | Date
}>
```

Good to know

In the future, we will support multiple sitemaps and sitemap indexes.

Version History

VersionChangesv13.3.0sitemap introduced.
Functions
cookiesThe cookies function allows you to read the HTTP incoming request cookies from a Server Component or write outgoing request cookies in a Server Action or Route Handler.

Good to know: cookies() is a Dynamic Function whose returned values cannot be known ahead of time. Using it in a layout or page will opt a route into dynamic rendering at request time.

cookies().get(name)

A method that takes a cookie name and returns an object with name and value. If a

cookie with name isn't found, it returns undefined. If multiple cookies match, it will only return the first match.
app/page.js import { cookies } from 'next/headers'

```
export default function Page() {
  const cookieStore = cookies()
  const theme = cookieStore.get('theme')
  return '...'
}
```

cookies().getAll()

A method that is similar to get, but returns a list of all the cookies with a matching name. If name is unspecified, it returns all the available cookies.
app/page.js import { cookies } from 'next/headers'

```
export default function Page() {
  const cookieStore = cookies()
  return cookieStore.getAll().map((cookie) => (
    <div key={cookie.name}>
      <p>Name: {cookie.name}</p>
      <p>Value: {cookie.value}</p>
    </div>
  ))
}
```

cookies().has(name)

A method that takes a cookie name and returns a boolean based on if the cookie exists (true) or not (false).
app/page.js import { cookies } from 'next/headers'

```
export default function Page() {
  const cookiesList = cookies()
  const hasCookie = cookiesList.has('theme')
  return '...'
}
```

cookies().set(name, value, options)

A method that takes a cookie name, value, and options and sets the outgoing request cookie.

Good to know: .set() is only available in a Server Action or Route Handler.

app/actions.js 'use server'

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

async function create(data) {
  cookies().set('name', 'lee')
  // or
  cookies().set('name', 'lee', { secure: true })
  // or
  cookies().set({
    name: 'name',
    value: 'lee',
    httpOnly: true,
    path: '/',
  })
}
```

Deleting cookies

To "delete" a cookie, you must set a new cookie with the same name and an empty value. You can also set the maxAge to 0 to expire the cookie immediately.

Good to know: .set() is only available in a Server Action or Route Handler.

app/actions.js 'use server'

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

async function create(data) {
  cookies().set({
    name: 'name',
    value: '',
    expires: new Date('2016-10-05'),
    path: '/', // For all paths
  })
```

}
You can only set cookies that belong to the same domain from which .set() is called. Additionally, the code must be executed on the same protocol (HTTP or HTTPS) as the cookie you want to update.
Version History

VersionChangesv13.0.0cookies introduced.
draftModeThe draftMode function allows you to detect Draft Mode inside a Server Component.
app/page.js import { draftMode } from 'next/headers'

```
export default function Page() {
  const { isEnabled } = draftMode()
  return (
    <main>
      <h1>My Blog Post</h1>
      <p>Draft Mode is currently {isEnabled ? 'Enabled' : 'Disabled'}</p>
    </main>
  )
}
```

Version History

VersionChangesv13.4.0draftMode introduced.
fetchNext.js extends the native Web fetch() API to allow each request on the server to set its own persistent caching semantics.
In the browser, the cache option indicates how a fetch request will interact with the browser's HTTP cache. With this extension, cache indicates how a server-side fetch request will interact with the framework's persistent HTTP cache.
You can call fetch with async and await directly within Server Components.
app/page.tsxTypeScript export default async function Page() {

```
  // This request should be cached until manually invalidated.
  // Similar to `getStaticProps`.
  // `force-cache` is the default and can be omitted.
  const staticData = await fetch(`https://...`, { cache: 'force-cache' })

  // This request should be refetched on every request.
  // Similar to `getServerSideProps`.
  const dynamicData = await fetch(`https://...`, { cache: 'no-store' })
```

```
  // This request should be cached with a lifetime of 10 seconds.
  // Similar to `getStaticProps` with the `revalidate` option.
  const revalidatedData = await fetch(`https://...`, {
    next: { revalidate: 10 },
  })

  return <div>...</div>
}

fetch(url, options)
```

Since Next.js extends the Web fetch() API, you can use any of the native options available.
Further, Next.js polyfills fetch on both the client and the server, so you can use fetch in both Server and Client Components.
options.cache

Configure how the request should interact with Next.js HTTP cache.

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

force-cache (default) - Next.js looks for a matching request in its HTTP cache.

If there is a match and it is fresh, it will be returned from the cache.
If there is no match or a stale match, Next.js will fetch the resource from the remote server and update the cache with the downloaded resource.

no-store - Next.js fetches the resource from the remote server on every request without looking in the cache, and it will not update the cache with the downloaded resource.

Good to know:

If you don't provide a cache option, Next.js will default to force-cache, unless a dynamic function such as cookies() is used, in which case it will default to no-store.
The no-cache option behaves the same way as no-store in Next.js.

options.next.revalidate

fetch(`https://...`, { next: { revalidate: false | 0 | number } })
Set the cache lifetime of a resource (in seconds).

false - Cache the resource indefinitely. Semantically equivalent to revalidate: Infinity. The HTTP cache may evict older resources over time.
0 - Prevent the resource from being cached.
number - (in seconds) Specify the resource should have a cache lifetime of at most n seconds.

Good to know:

If an individual fetch() request sets a revalidate number lower than the default revalidate of a route, the whole route revalidation interval will be decreased.
If two fetch requests with the same URL in the same route have different revalidate values, the lower value will be used.
As a convenience, it is not necessary to set the cache option if revalidate is set to a number since 0 implies cache: 'no-store' and a positive value implies cache: 'force-cache'.
Conflicting options such as { revalidate: 0, cache: 'force-cache' } or { revalidate: 10, cache: 'no-store' } will cause an error.

Version History

VersionChangesv13.0.0fetch introduced.
generateImageMetadataYou can use generateImageMetadata to generate different versions of one image or return multiple images for one route segment. This is useful for when you want to avoid hard-coding metadata values, such as for icons.
Parameters

generateImageMetadata function accepts the following parameters:
params (optional)

An object containing the dynamic route parameters object from the root segment down to the segment generateImageMetadata is called from.
icon.tsxTypeScript export function generateImageMetadata({
  params,
}: {
  params: { slug: string }
}) {
  // ...
}

RouteURLparamsapp/shop/icon.js/shopundefinedapp/shop/[slug]/icon.js/shop/1{ slug: '1' }app/shop/[tag]/[item]/icon.js/shop/1/2{ tag: '1', item: '2' }app/shop/[...slug]/icon.js/shop/1/2{ slug: ['1', '2'] }
Returns

The generateImageMetadata function should return an array of objects containing the image's metadata such as alt and size. In addition, each item must include an id value will be passed to the props of the image generating function.
Image Metadata ObjectTypeidstring (required)altstringsize{ width: number; height: number }contentTypestring
icon.tsxTypeScript import { ImageResponse } from 'next/server'

```
export function generateImageMetadata() {
  return [
    {
      contentType: 'image/png',
      size: { width: 48, height: 48 },
      id: 'small',
    },
    {
      contentType: 'image/png',
      size: { width: 72, height: 72 },
      id: 'medium',
    },
  ]
```

```
}

export default function Icon({ id }: { id: string }) {
  return new ImageResponse(
    (
      <div
        style={{
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
          fontSize: 88,
          background: '#000',
          color: '#fafafa',
        }}
      >
        Icon {id}
      </div>
    )
  )
}
```

Examples

Using external data

This example uses the params object and external data to generate multiple Open Graph images for a route segment.

app/products/[id]/opengraph-image.tsxTypeScript
```
import { ImageResponse } from 'next/server'
import { getCaptionForImage, getOGImages } from '@/app/utils/images'

export async function generateImageMetadata({
  params,
}: {
  params: { id: string }
}) {
```

```
  const images = await getOGImages(params.id)

  return images.map((image, idx) => ({
    id: idx,
    size: { width: 1200, height: 600 },
    alt: image.text,
    contentType: 'image/png',
  }))
}

export default async function Image({
  params,
  id,
}: {
  params: { id: string }
  id: number
}) {
  const productId = params.id
  const imageId = id
  const text = await getCaptionForImage(productId, imageId)

  return new ImageResponse(
    (
      <div
        style={
          {
            // ...
          }
        }
      >
        {text}
      </div>
    )
  )
}
```

Version History

VersionChangesv13.3.0generateImageMetadata introduced.
Metadata Object and generateMetadata OptionsThis page covers all Config-based Metadata options with generateMetadata and the static metadata object.
layout.tsx / page.tsxTypeScript import { Metadata } from 'next'

```
// either Static metadata
export const metadata: Metadata = {
  title: '...',
}

// or Dynamic metadata
export async function generateMetadata({ params }) {
  return {
    title: '...',
  }
}
```

Good to know:

The metadata object and generateMetadata function exports are only supported in Server Components.
You cannot export both the metadata object and generateMetadata function from the same route segment.

The metadata object

To define static metadata, export a Metadata object from a layout.js or page.js file.
layout.tsx / page.tsxTypeScript import { Metadata } from 'next'

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

export default function Page() {}
```

See the Metadata Fields for a complete list of supported options.
generateMetadata function

Dynamic metadata depends on dynamic information, such as the current route

parameters, external data, or metadata in parent segments, can be set by exporting a generateMetadata function that returns a Metadata object.
app/products/[id]/page.tsxTypeScript import { Metadata, ResolvingMetadata } from 'next'

```
type Props = {
  params: { id: string }
  searchParams: { [key: string]: string | string[] | undefined }
}

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

  // fetch data
  const product = await fetch(`https://.../${id}`).then((res) => res.json())

  // optionally access and extend (rather than replace) parent metadata
  const previousImages = (await parent).openGraph?.images || []

  return {
    title: product.title,
    openGraph: {
      images: ['/some-specific-page-image.jpg', ...previousImages],
    },
  }
}

export default function Page({ params, searchParams }: Props) {}
```

Parameters

generateMetadata function accepts the following parameters:

props - An object containing the parameters of the current route:

params - An object containing the dynamic route parameters object from the root segment down to the segment generateMetadata is called from. Examples: RouteURLparamsapp/shop/[slug]/page.js/shop/1{ slug: '1' }app/shop/[tag]/[item]/page.js/

shop/1/2{ tag: '1', item: '2' }app/shop/[...slug]/page.js/shop/1/2{ slug: ['1', '2'] }

searchParams - An object containing the current URL's search params. Examples: URLsearchParams/shop?a=1{ a: '1' }/shop?a=1&b=2{ a: '1', b: '2' }/shop?a=1&a=2{ a: ['1', '2'] }

parent - A promise of the resolved metadata from parent route segments.

Returns

generateMetadata should return a Metadata object containing one or more metadata fields.

Good to know:

If metadata doesn't depend on runtime information, it should be defined using the static metadata object rather than generateMetadata.
When rendering a route, Next.js will automatically deduplicate fetch requests for the same data across generateMetadata, generateStaticParams, Layouts, Pages, and Server Components. React cache can be used if fetch is unavailable.
searchParams are only available in page.js segments.
The redirect() and notFound() Next.js methods can also be used inside generateMetadata.

Metadata Fields

title

The title attribute is used to set the title of the document. It can be defined as a simple string or an optional template object.
String

layout.js / page.js export const metadata = {
  title: 'Next.js',
}
<head> output <title>Next.js</title>
Template object

app/layout.tsxTypeScript import { Metadata } from 'next'

export const metadata: Metadata = {
  title: {
    template: '...',
    default: '...',
    absolute: '...',
  },
}

Default

title.default can be used to provide a fallback title to child route segments that don't define a title.
app/layout.tsx import type { Metadata } from 'next'

export const metadata: Metadata = {
  title: {
    default: 'Acme',
  },
}
app/about/page.tsx import type { Metadata } from 'next'

export const metadata: Metadata = {}

// Output: <title>Acme</title>
Template

title.template can be used to add a prefix or a suffix to titles defined in child route segments.
app/layout.tsxTypeScript import { Metadata } from 'next'

```
export const metadata: Metadata = {
  title: {
    template: '%s | Acme',
    default: 'Acme', // a default is required when creating a template
  },
}
```

app/about/page.tsxTypeScript import { Metadata } from 'next'

```
export const metadata: Metadata = {
  title: 'About',
}
```

// Output: <title>About | Acme</title>

Good to know:

title.template applies to child route segments and not the segment it's defined in. This means:

title.default is required when you add a title.template.
title.template defined in layout.js will not apply to a title defined in a page.js of the same route segment.
title.template defined in page.js has no effect because a page is always the terminating segment (it doesn't have any children route segments).

title.template has no effect if a route has not defined a title or title.default.

Absolute

title.absolute can be used to provide a title that ignores title.template set in parent segments.
app/layout.tsxTypeScript import { Metadata } from 'next'

```
export const metadata: Metadata = {
  title: {
    template: '%s | Acme',
  },
}
```

app/about/page.tsxTypeScript import { Metadata } from 'next'

```
export const metadata: Metadata = {
  title: {
    absolute: 'About',
  },
}

// Output: <title>About</title>
```

Good to know:

layout.js

title (string) and title.default define the default title for child segments (that do not define their own title). It will augment title.template from the closest parent segment if it exists.
title.absolute defines the default title for child segments. It ignores title.template from parent segments.
title.template defines a new title template for child segments.

page.js

If a page does not define its own title the closest parents resolved title will be used.
title (string) defines the routes title. It will augment title.template from the closest parent segment if it exists.
title.absolute defines the route title. It ignores title.template from parent segments.

title.template has no effect in page.js because a page is always the terminating segment of a route.

description

layout.js / page.js export const metadata = {
  description: 'The React Framework for the Web',
}
<head> output <meta name="description" content="The React Framework for the Web" />
Basic Fields

layout.js / page.js export const metadata = {
  generator: 'Next.js',
  applicationName: 'Next.js',
  referrer: 'origin-when-cross-origin',
  keywords: ['Next.js', 'React', 'JavaScript'],
  authors: [{ name: 'Seb' }, { name: 'Josh', url: 'https://nextjs.org' }],
  colorScheme: 'dark',
  creator: 'Jiachi Liu',
  publisher: 'Sebastian Markbåge',
  formatDetection: {
    email: false,
    address: false,
    telephone: false,
  },
}
<head> output <meta name="application-name" content="Next.js" />
<meta name="author" content="Seb" />
<link rel="author" href="https://nextjs.org" />
<meta name="author" content="Josh" />
<meta name="generator" content="Next.js" />
<meta name="keywords" content="Next.js,React,JavaScript" />
<meta name="referrer" content="origin-when-cross-origin" />
<meta name="color-scheme" content="dark" />

<meta name="creator" content="Jiachi Liu" />
<meta name="publisher" content="Sebastian Markbåge" />
<meta name="format-detection" content="telephone=no, address=no, email=no" />
metadataBase

metadataBase is a convenience option to set a base URL prefix for metadata fields that require a fully qualified URL.

metadataBase allows URL-based metadata fields defined in the current route segment and below to use a relative path instead of an otherwise required absolute URL.
The field's relative path will be composed with metadataBase to form a fully qualified URL.
If not configured, metadataBase is automatically populated with a default value.

layout.js / page.js export const metadata = {
  metadataBase: new URL('https://acme.com'),
  alternates: {
    canonical: '/',
    languages: {
      'en-US': '/en-US',
      'de-DE': '/de-DE',
    },
  },
  openGraph: {
    images: '/og-image.png',
  },
}
<head> output <link rel="canonical" href="https://acme.com" />
<link rel="alternate" hreflang="en-US" href="https://acme.com/en-US" />
<link rel="alternate" hreflang="de-DE" href="https://acme.com/de-DE" />
<meta property="og:image" content="https://acme.com/og-image.png" />

Good to know:

metadataBase is typically set in root app/layout.js to apply to URL-based metadata fields across all routes.
All URL-based metadata fields that require absolute URLs can be configured with a metadataBase option.
metadataBase can contain a subdomain e.g. https://app.acme.com or base path e.g. https://acme.com/start/from/here
If a metadata field provides an absolute URL, metadataBase will be ignored.
Using a relative path in a URL-based metadata field without configuring a

metadataBase will cause a build error.
Next.js will normalize duplicate slashes between metadataBase (e.g. https://acme.com/) and a relative field (e.g. /path) to a single slash (e.g. https://acme.com/path)

Default value

If not configured, metadataBase has a default value

When VERCEL_URL is detected: https://${process.env.VERCEL_URL} otherwise it falls back to http://localhost:${process.env.PORT || 3000}.
When overriding the default, we recommend using environment variables to compute the URL. This allows configuring a URL for local development, staging, and production environments.

URL Composition

URL composition favors developer intent over default directory traversal semantics.

Trailing slashes between metadataBase and metadata fields are normalized.
An "absolute" path in a metadata field (that typically would replace the whole URL path) is treated as a "relative" path (starting from the end of metadataBase).

For example, given the following metadataBase:
app/layout.tsxTypeScript import { Metadata } from 'next'

```
export const metadata: Metadata = {
  metadataBase: new URL('https://acme.com'),
}
```

Any metadata fields that inherit the above metadataBase and set their own value will be resolved as follows:
metadata fieldResolved URL/https://acme.com./https://acme.compaymentshttps://acme.com/payments/paymentshttps://acme.com/payments./paymentshttps://acme.com/payments../paymentshttps://acme.com/paymentshttps://beta.acme.com/paymentshttps://beta.acme.com/payments
openGraph

layout.js / page.js export const metadata = {
  openGraph: {
    title: 'Next.js',
    description: 'The React Framework for the Web',
    url: 'https://nextjs.org',
    siteName: 'Next.js',
    images: [
      {
        url: 'https://nextjs.org/og.png',
        width: 800,
        height: 600,
      },
      {
        url: 'https://nextjs.org/og-alt.png',
        width: 1800,
        height: 1600,
        alt: 'My custom alt',
      },
    ],
    locale: 'en_US',
    type: 'website',
  },
}
<head> output <meta property="og:title" content="Next.js" />
<meta property="og:description" content="The React Framework for the Web" />
<meta property="og:url" content="https://nextjs.org/" />
<meta property="og:site_name" content="Next.js" />
<meta property="og:locale" content="en_US" />
<meta property="og:image:url" content="https://nextjs.org/og.png" />
<meta property="og:image:width" content="800" />
<meta property="og:image:height" content="600" />
<meta property="og:image:url" content="https://nextjs.org/og-alt.png" />
<meta property="og:image:width" content="1800" />
<meta property="og:image:height" content="1600" />
<meta property="og:image:alt" content="My custom alt" />
<meta property="og:type" content="website" />
layout.js / page.js export const metadata = {
  openGraph: {
    title: 'Next.js',
    description: 'The React Framework for the Web',
    type: 'article',
    publishedTime: '2023-01-01T00:00:00.000Z',

```
    authors: ['Seb', 'Josh'],
  },
}
<head> output <meta property="og:title" content="Next.js" />
<meta property="og:description" content="The React Framework for the Web" />
<meta property="og:type" content="article" />
<meta property="article:published_time" content="2023-01-01T00:00:00.000Z" />
<meta property="article:author" content="Seb" />
<meta property="article:author" content="Josh" />
```

Good to know:

It may be more convenient to use the file-based Metadata API for Open Graph images. Rather than having to sync the config export with actual files, the file-based API will automatically generate the correct metadata for you.

robots

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

export const metadata: Metadata = {
  robots: {
    index: false,
    follow: true,
    nocache: true,
    googleBot: {
      index: true,
      follow: false,
      noimageindex: true,
      'max-video-preview': -1,
      'max-image-preview': 'large',
      'max-snippet': -1,
    },
  },
}
<head> output <meta name="robots" content="noindex, follow, nocache" />
<meta
  name="googlebot"
  content="index, nofollow, noimageindex, max-video-preview:-1, max-image-
preview:large, max-snippet:-1"
/>
```

icons

Good to know: We recommend using the file-based Metadata API for icons where possible. Rather than having to sync the config export with actual files, the file-based API will automatically generate the correct metadata for you.

```
layout.js / page.js export const metadata = {
  icons: {
    icon: '/icon.png',
    shortcut: '/shortcut-icon.png',
    apple: '/apple-icon.png',
    other: {
      rel: 'apple-touch-icon-precomposed',
      url: '/apple-touch-icon-precomposed.png',
    },
  },
}
<head> output <link rel="shortcut icon" href="/shortcut-icon.png" />
<link rel="icon" href="/icon.png" />
<link rel="apple-touch-icon" href="/apple-icon.png" />
<link
  rel="apple-touch-icon-precomposed"
  href="/apple-touch-icon-precomposed.png"
/>
layout.js / page.js export const metadata = {
  icons: {
    icon: [{ url: '/icon.png' }, new URL('/icon.png', 'https://example.com')],
    shortcut: ['/shortcut-icon.png'],
    apple: [
      { url: '/apple-icon.png' },
      { url: '/apple-icon-x3.png', sizes: '180x180', type: 'image/png' },
    ],
    other: [
      {
        rel: 'apple-touch-icon-precomposed',
        url: '/apple-touch-icon-precomposed.png',
      },
    ],
  },
}
<head> output <link rel="shortcut icon" href="/shortcut-icon.png" />
```

```
<link rel="icon" href="/icon.png" />
<link rel="apple-touch-icon" href="/apple-icon.png" />
<link
  rel="apple-touch-icon-precomposed"
  href="/apple-touch-icon-precomposed.png"
/>
<link rel="icon" href="https://example.com/icon.png" />
<link
  rel="apple-touch-icon"
  href="/apple-icon-x3.png"
  sizes="180x180"
  type="image/png"
/>
```

Good to know: The msapplication-* meta tags are no longer supported in Chromium builds of Microsoft Edge, and thus no longer needed.

themeColor

Learn more about theme-color.
Simple theme color
layout.js / page.js
```
export const metadata = {
  themeColor: 'black',
}
```
<head> output
```
<meta name="theme-color" content="black" />
```
With media attribute
layout.js / page.js
```
export const metadata = {
  themeColor: [
    { media: '(prefers-color-scheme: light)', color: 'cyan' },
    { media: '(prefers-color-scheme: dark)', color: 'black' },
  ],
}
```
<head> output
```
<meta name="theme-color" media="(prefers-color-scheme: light)" content="cyan" />
<meta name="theme-color" media="(prefers-color-scheme: dark)" content="black" />
```
manifest

A web application manifest, as defined in the Web Application Manifest specification.

layout.js / page.js export const metadata = {
  manifest: 'https://nextjs.org/manifest.json',
}
<head> output <link rel="manifest" href="https://nextjs.org/manifest.json" />
twitter

Learn more about the Twitter Card markup reference.
layout.js / page.js export const metadata = {
  twitter: {
    card: 'summary_large_image',
    title: 'Next.js',
    description: 'The React Framework for the Web',
    siteId: '1467726470533754880',
    creator: '@nextjs',
    creatorId: '1467726470533754880',
    images: ['https://nextjs.org/og.png'],
  },
}
<head> output <meta name="twitter:card" content="summary_large_image" />
<meta name="twitter:site:id" content="1467726470533754880" />
<meta name="twitter:creator" content="@nextjs" />
<meta name="twitter:creator:id" content="1467726470533754880" />
<meta name="twitter:title" content="Next.js" />
<meta name="twitter:description" content="The React Framework for the Web" />
<meta name="twitter:image" content="https://nextjs.org/og.png" />
layout.js / page.js export const metadata = {
  twitter: {
    card: 'app',
    title: 'Next.js',
    description: 'The React Framework for the Web',
    siteId: '1467726470533754880',
    creator: '@nextjs',
    creatorId: '1467726470533754880',
    images: {
      url: 'https://nextjs.org/og.png',
      alt: 'Next.js Logo',
    },
    app: {
      name: 'twitter_app',
      id: {
        iphone: 'twitter_app://iphone',
        ipad: 'twitter_app://ipad',

```
      googleplay: 'twitter_app://googleplay',
    },
    url: {
      iphone: 'https://iphone_url',
      ipad: 'https://ipad_url',
    },
  },
 },
}
<head> output <meta name="twitter:site:id" content="1467726470533754880" />
<meta name="twitter:creator" content="@nextjs" />
<meta name="twitter:creator:id" content="1467726470533754880" />
<meta name="twitter:title" content="Next.js" />
<meta name="twitter:description" content="The React Framework for the Web" />
<meta name="twitter:card" content="app" />
<meta name="twitter:image" content="https://nextjs.org/og.png" />
<meta name="twitter:image:alt" content="Next.js Logo" />
<meta name="twitter:app:name:iphone" content="twitter_app" />
<meta name="twitter:app:id:iphone" content="twitter_app://iphone" />
<meta name="twitter:app:id:ipad" content="twitter_app://ipad" />
<meta name="twitter:app:id:googleplay" content="twitter_app://googleplay" />
<meta name="twitter:app:url:iphone" content="https://iphone_url" />
<meta name="twitter:app:url:ipad" content="https://ipad_url" />
<meta name="twitter:app:name:ipad" content="twitter_app" />
<meta name="twitter:app:name:googleplay" content="twitter_app" />
viewport
```

Good to know: The viewport meta tag is automatically set with the following default values. Usually, manual configuration is unnecessary as the default is sufficient. However, the information is provided for completeness.

```
layout.js / page.js export const metadata = {
  viewport: {
    width: 'device-width',
    initialScale: 1,
    maximumScale: 1,
  },
}
<head> output <meta
  name="viewport"
  content="width=device-width, initial-scale=1, maximum-scale=1"
```

/>
verification

layout.js / page.js export const metadata = {
  verification: {
    google: 'google',
    yandex: 'yandex',
    yahoo: 'yahoo',
    other: {
      me: ['my-email', 'my-link'],
    },
  },
}
<head> output <meta name="google-site-verification" content="google" />
<meta name="y_key" content="yahoo" />
<meta name="yandex-verification" content="yandex" />
<meta name="me" content="my-email" />
<meta name="me" content="my-link" />
appleWebApp

layout.js / page.js export const metadata = {
  itunes: {
    appId: 'myAppStoreID',
    appArgument: 'myAppArgument',
  },
  appleWebApp: {
    title: 'Apple Web App',
    statusBarStyle: 'black-translucent',
    startupImage: [
      '/assets/startup/apple-touch-startup-image-768x1004.png',
      {
        url: '/assets/startup/apple-touch-startup-image-1536x2008.png',
        media: '(device-width: 768px) and (device-height: 1024px)',
      },
    ],
  },
}
<head> output <meta

```
  name="apple-itunes-app"
  content="app-id=myAppStoreID, app-argument=myAppArgument"
/>
<meta name="apple-mobile-web-app-capable" content="yes" />
<meta name="apple-mobile-web-app-title" content="Apple Web App" />
<link
  href="/assets/startup/apple-touch-startup-image-768x1004.png"
  rel="apple-touch-startup-image"
/>
<link
  href="/assets/startup/apple-touch-startup-image-1536x2008.png"
  media="(device-width: 768px) and (device-height: 1024px)"
  rel="apple-touch-startup-image"
/>
<meta
  name="apple-mobile-web-app-status-bar-style"
  content="black-translucent"
/>
alternates
```

```
layout.js / page.js export const metadata = {
  alternates: {
    canonical: 'https://nextjs.org',
    languages: {
      'en-US': 'https://nextjs.org/en-US',
      'de-DE': 'https://nextjs.org/de-DE',
    },
    media: {
      'only screen and (max-width: 600px)': 'https://nextjs.org/mobile',
    },
    types: {
      'application/rss+xml': 'https://nextjs.org/rss',
    },
  },
}
<head> output <link rel="canonical" href="https://nextjs.org" />
<link rel="alternate" hreflang="en-US" href="https://nextjs.org/en-US" />
<link rel="alternate" hreflang="de-DE" href="https://nextjs.org/de-DE" />
<link
  rel="alternate"
  media="only screen and (max-width: 600px)"
  href="https://nextjs.org/mobile"
```

/>
<link
  rel="alternate"
  type="application/rss+xml"
  href="https://nextjs.org/rss"
/>
appLinks

layout.js / page.js export const metadata = {
  appLinks: {
    ios: {
      url: 'https://nextjs.org/ios',
      app_store_id: 'app_store_id',
    },
    android: {
      package: 'com.example.android/package',
      app_name: 'app_name_android',
    },
    web: {
      url: 'https://nextjs.org/web',
      should_fallback: true,
    },
  },
}
<head> output <meta property="al:ios:url" content="https://nextjs.org/ios" />
<meta property="al:ios:app_store_id" content="app_store_id" />
<meta property="al:android:package" content="com.example.android/package" />
<meta property="al:android:app_name" content="app_name_android" />
<meta property="al:web:url" content="https://nextjs.org/web" />
<meta property="al:web:should_fallback" content="true" />
archives

Describes a collection of records, documents, or other materials of historical interest (source).
layout.js / page.js export const metadata = {
  archives: ['https://nextjs.org/13'],
}
<head> output <link rel="archives" href="https://nextjs.org/13" />

assets

layout.js / page.js export const metadata = {
  assets: ['https://nextjs.org/assets'],
}
<head> output <link rel="assets" href="https://nextjs.org/assets" />
bookmarks

layout.js / page.js export const metadata = {
  bookmarks: ['https://nextjs.org/13'],
}
<head> output <link rel="bookmarks" href="https://nextjs.org/13" />
category

layout.js / page.js export const metadata = {
  category: 'technology',
}
<head> output <meta name="category" content="technology" />
other

All metadata options should be covered using the built-in support. However, there may be custom metadata tags specific to your site, or brand new metadata tags just released. You can use the other option to render any custom metadata tag.
layout.js / page.js export const metadata = {
  other: {
    custom: 'meta',
  },
}
<head> output <meta name="custom" content="meta" />
Unsupported Metadata

The following metadata types do not currently have built-in support. However, they can still be rendered in the layout or page itself.
MetadataRecommendation<meta http-equiv="...">Use appropriate HTTP Headers via redirect(), Middleware, Security Headers<base>Render the tag in the layout or page itself.<noscript>Render the tag in the layout or page itself.<style>Learn more about styling in Next.js.<script>Learn more about using scripts.<link rel="stylesheet" />import stylesheets directly in the layout or page itself.<link rel="preload />Use ReactDOM preload method<link rel="preconnect" />Use ReactDOM preconnect method<link rel="dns-prefetch" />Use ReactDOM prefetchDNS method
Resource hints

The <link> element has a number of rel keywords that can be used to hint to the browser that a external resource is likely to be needed. The browser uses this information to apply preloading optimizations depending on the keyword.
While the Metadata API doesn't directly support these hints, you can use new ReactDOM methods to safely insert them into the <head> of the document.
app/preload-resources.tsxTypeScript 'use client'

```
import ReactDOM from 'react-dom'

export function PreloadResources() {
  ReactDOM.preload('...', { as: '...' })
  ReactDOM.preconnect('...', { crossOrigin: '...' })
  ReactDOM.prefetchDNS('...')

  return null
}
```

<link rel="preload">

Start loading a resource early in the page rendering (browser) lifecycle. MDN Docs.
ReactDOM.preload(href: string, options: { as: string })
<head> output <link rel="preload" href="..." as="..." />

<link rel="preconnect">

Preemptively initiate a connection to an origin. MDN Docs.
ReactDOM.preconnect(href: string, options?: { crossOrigin?: string })
<head> output <link rel="preconnect" href="..." crossorigin />
<link rel="dns-prefetch">

Attempt to resolve a domain name before resources get requested. MDN Docs.
ReactDOM.prefetchDNS(href: string)
<head> output <link rel="dns-prefetch" href="..." />

Good to know:

These methods are currently only supported in Client Components, which are still Server Side Rendered on initial page load.
Next.js in-built features such as next/font, next/image and next/script automatically handle relevant resource hints.
React 18.3 does not yet include type definitions for ReactDOM.preload, ReactDOM.preconnect, and ReactDOM.preconnectDNS. You can use // @ts-ignore as a temporary solution to avoid type errors.

Types

You can add type safety to your metadata by using the Metadata type. If you are using the built-in TypeScript plugin in your IDE, you do not need to manually add the type, but you can still explicitly add it if you want.
metadata object

import type { Metadata } from 'next'

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

generateMetadata function

Regular function

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

export function generateMetadata(): Metadata {
  return {
    title: 'Next.js',
  }
}
```

Async function

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

export async function generateMetadata(): Promise<Metadata> {
  return {
    title: 'Next.js',
  }
}
```

With segment props

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

type Props = {
  params: { id: string }
```

```
  searchParams: { [key: string]: string | string[] | undefined }
}

export function generateMetadata({ params, searchParams }: Props): Metadata {
  return {
    title: 'Next.js',
  }
}

export default function Page({ params, searchParams }: Props) {}
```

With parent metadata

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

export async function generateMetadata(
  { params, searchParams }: Props,
  parent: ResolvingMetadata
): Promise<Metadata> {
  return {
    title: 'Next.js',
  }
}
```

JavaScript Projects

For JavaScript projects, you can use JSDoc to add type safety.

```
 /** @type {import("next").Metadata} */
export const metadata = {
  title: 'Next.js',
}
```

Version History

VersionChangesv13.2.0metadata and generateMetadata introduced.
generateStaticParamsThe generateStaticParams function can be used in combination with dynamic route segments to statically generate routes at build time instead of on-

demand at request time.

```
app/blog/[slug]/page.js // Return a list of `params` to populate the [slug] dynamic segment
export async function generateStaticParams() {
  const posts = await fetch('https://.../posts').then((res) => res.json())

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

// Multiple versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
export default function Page({ params }) {
  const { slug } = params
  // ...
}
```

Good to know

You can use the dynamicParams segment config option to control what happens when a dynamic segment is visited that was not generated with generateStaticParams.
During next dev, generateStaticParams will be called when you navigate to a route.
During next build, generateStaticParams runs before the corresponding Layouts or Pages are generated.
During revalidation (ISR), generateStaticParams will not be called again.
generateStaticParams replaces the getStaticPaths function in the Pages Router.

Parameters

options.params (optional)
If multiple dynamic segments in a route use generateStaticParams, the child generateStaticParams function is executed once for each set of params the parent generates.
The params object contains the populated params from the parent generateStaticParams, which can be used to generate the params in a child segment.
Returns

generateStaticParams should return an array of objects where each object represents the populated dynamic segments of a single route.

Each property in the object is a dynamic segment to be filled in for the route.
The properties name is the segment's name, and the properties value is what that segment should be filled in with.

Example RoutegenerateStaticParams Return Type/product/[id]{ id: string }[]/products/[category]/[product]{ category: string, product: string }[]/products/[...slug]{ slug: string[] }[]
Single Dynamic Segment

app/product/[id]/page.tsxTypeScript export function generateStaticParams() {
  return [{ id: '1' }, { id: '2' }, { id: '3' }]
}

// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /product/1
// - /product/2
// - /product/3
export default function Page({ params }: { params: { id: string } }) {
  const { id } = params
  // ...
}

Multiple Dynamic Segments

app/products/[category]/[product]/page.tsxTypeScript export function generateStaticParams() {
  return [
    { category: 'a', product: '1' },
    { category: 'b', product: '2' },
    { category: 'c', product: '3' },
  ]
}

// Three versions of this page will be statically generated

```
// using the `params` returned by `generateStaticParams`
// - /product/a/1
// - /product/b/2
// - /product/c/3
export default function Page({
  params,
}: {
  params: { category: string; product: string }
}) {
  const { category, product } = params
  // ...
}
```

Catch-all Dynamic Segment

app/product/[...slug]/page.tsxTypeScript export function generateStaticParams() {
  return [{ slug: ['a', '1'] }, { slug: ['b', '2'] }, { slug: ['c', '3'] }]
}

```
// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /product/a/1
// - /product/b/2
// - /product/c/3
export default function Page({ params }: { params: { slug: string[] } }) {
  const { slug } = params
  // ...
}
```

Examples

Multiple Dynamic Segments in a Route

You can generate params for dynamic segments above the current layout or page, but

not below. For example, given the app/products/[category]/[product] route:

app/products/[category]/[product]/page.js can generate params for both [category] and [product].
app/products/[category]/layout.js can only generate params for [category].

There are two approaches to generating params for a route with multiple dynamic segments:
Generate params from the bottom up

Generate multiple dynamic segments from the child route segment.
app/products/[category]/[product]/page.tsxTypeScript // Generate segments for both [category] and [product]

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

  return products.map((product) => ({
    category: product.category.slug,
    product: product.id,
  }))
}

export default function Page({
  params,
}: {
  params: { category: string; product: string }
}) {
  // ...
}
```

Generate params from the top down

Generate the parent segments first and use the result to generate the child segments.
app/products/[category]/layout.tsxTypeScript // Generate segments for [category]

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

  return products.map((product) => ({
```

```
    category: product.category.slug,
  }))
}

export default function Layout({ params }: { params: { category: string } }) {
  // ...
}
```

A child route segment's generateStaticParams function is executed once for each segment a parent generateStaticParams generates.
The child generateStaticParams function can use the params returned from the parent generateStaticParams function to dynamically generate its own segments.
app/products/[category]/[product]/page.tsxTypeScript // Generate segments for [product] using the `params` passed from
// the parent segment's `generateStaticParams` function

```
export async function generateStaticParams({
  params: { category },
}: {
  params: { category: string }
}) {
  const products = await fetch(
    `https://.../products?category=${category}`
  ).then((res) => res.json())

  return products.map((product) => ({
    product: product.id,
  }))
}

export default function Page({
  params,
}: {
  params: { category: string; product: string }
}) {
  // ...
}
```

Good to know: When rendering a route, Next.js will automatically deduplicate fetch requests for the same data across generateMetadata, generateStaticParams, Layouts, Pages, and Server Components. React cache can be used if fetch is unavailable.

Version History

VersionChangesv13.0.0generateStaticParams introduced.
headersThe headers function allows you to read the HTTP incoming request headers from a Server Component.
headers()

This API extends the Web Headers API. It is read-only, meaning you cannot set / delete the outgoing request headers.
app/page.tsxTypeScript import { headers } from 'next/headers'

```
export default function Page() {
  const headersList = headers()
  const referer = headersList.get('referer')

  return <div>Referer: {referer}</div>
}
```

Good to know:

headers() is a Dynamic Function whose returned values cannot be known ahead of time. Using it in a layout or page will opt a route into dynamic rendering at request time.

API Reference

```
 const headersList = headers()
```

Parameters

headers does not take any parameters.
Returns

headers returns a read-only Web Headers object.

Headers.entries(): Returns an iterator allowing to go through all key/value pairs contained in this object.
Headers.forEach(): Executes a provided function once for each key/value pair in this Headers object.
Headers.get(): Returns a String sequence of all the values of a header within a Headers object with a given name.
Headers.has(): Returns a boolean stating whether a Headers object contains a certain header.
Headers.keys(): Returns an iterator allowing you to go through all keys of the key/value pairs contained in this object.
Headers.values(): Returns an iterator allowing you to go through all values of the key/value pairs contained in this object.

Examples

Usage with Data Fetching

headers() can be used in combination with Suspense for Data Fetching.
app/page.js import { headers } from 'next/headers'

```
async function getUser() {
  const headersInstance = headers()
  const authorization = headersInstance.get('authorization')
  // Forward the authorization header
  const res = await fetch('...', {
    headers: { authorization },
  })
  return res.json()
}

export default async function UserPage() {
  const user = await getUser()
  return <h1>{user.name}</h1>
```

}
Version History

VersionChangesv13.0.0headers introduced.
ImageResponseThe ImageResponse constructor allows you to generate dynamic images using JSX and CSS. This is useful for generating social media images such as Open Graph images, Twitter cards, and more.
The following options are available for ImageResponse:

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

new ImageResponse(
  element: ReactElement,
  options: {
    width?: number = 1200
    height?: number = 630
    emoji?: 'twemoji' | 'blobmoji' | 'noto' | 'openmoji' = 'twemoji',
    fonts?: {
      name: string,
      data: ArrayBuffer,
      weight: number,
      style: 'normal' | 'italic'
    }[]
    debug?: boolean = false

    // Options that will be passed to the HTTP response
    status?: number = 200
    statusText?: string
    headers?: Record<string, string>
  },
)
```

Supported CSS Properties

Please refer to Satori’s documentation for a list of supported HTML and CSS features.
Version History

VersionChangesv13.3.0ImageReponse can be imported from next/server.v13.0.0ImageReponse introduced via @vercel/og package.
NextRequestNextRequest extends the Web Request API with additional convenience methods.
cookies

Read or mutate the Set-Cookie header of the request.
set(name, value)

Given a name, set a cookie with the given value on the request.

```
 // Given incoming request /home
// Set a cookie to hide the banner
// request will have a `Set-Cookie:show-banner=false;path=/home` header
request.cookies.set('show-banner', 'false')
```

get(name)

Given a cookie name, return the value of the cookie. If the cookie is not found, undefined is returned. If multiple cookies are found, the first one is returned.

```
 // Given incoming request /home
// { name: 'show-banner', value: 'false', Path: '/home' }
request.cookies.get('show-banner')
```

getAll()

Given a cookie name, return the values of the cookie. If no name is given, return all cookies on the request.

```
 // Given incoming request /home
// [
//   { name: 'experiments', value: 'new-pricing-page', Path: '/home' },
//   { name: 'experiments', value: 'winter-launch', Path: '/home' },
```

//]
request.cookies.getAll('experiments')
// Alternatively, get all cookies for the request
request.cookies.getAll()
delete(name)

Given a cookie name, delete the cookie from the request.
// Returns true for deleted, false is nothing is deleted
request.cookies.delete('experiments')
has(name)

Given a cookie name, return true if the cookie exists on the request.
// Returns true if cookie exists, false if it does not
request.cookies.has('experiments')
clear()

Remove the Set-Cookie header from the request.
request.cookies.clear()
nextUrl

Extends the native URL API with additional convenience methods, including Next.js specific properties.
// Given a request to /home, pathname is /home
request.nextUrl.pathname
// Given a request to /home?name=lee, searchParams is { 'name': 'lee' }
request.nextUrl.searchParams
Version History

VersionChangesv13.0.0useSearchParams introduced.
NextResponseNextResponse extends the Web Response API with additional convenience methods.
cookies

Read or mutate the Set-Cookie header of the response.
set(name, value)

Given a name, set a cookie with the given value on the response.

```
 // Given incoming request /home
let response = NextResponse.next()
// Set a cookie to hide the banner
response.cookies.set('show-banner', 'false')
// Response will have a `Set-Cookie:show-banner=false;path=/home` header
return response
```

get(name)

Given a cookie name, return the value of the cookie. If the cookie is not found, undefined is returned. If multiple cookies are found, the first one is returned.

```
 // Given incoming request /home
let response = NextResponse.next()
// { name: 'show-banner', value: 'false', Path: '/home' }
response.cookies.get('show-banner')
```

getAll()

Given a cookie name, return the values of the cookie. If no name is given, return all cookies on the response.

```
 // Given incoming request /home
```

```
let response = NextResponse.next()
// [
//   { name: 'experiments', value: 'new-pricing-page', Path: '/home' },
//   { name: 'experiments', value: 'winter-launch', Path: '/home' },
// ]
response.cookies.getAll('experiments')
// Alternatively, get all cookies for the response
response.cookies.getAll()
delete(name)
```

Given a cookie name, delete the cookie from the response.

```
 // Given incoming request /home
let response = NextResponse.next()
// Returns true for deleted, false is nothing is deleted
response.cookies.delete('experiments')
json()
```

Produce a response with the given JSON body.

app/api/route.tsTypeScript import { NextResponse } from 'next/server'

```
export async function GET(request: Request) {
  return NextResponse.json({ error: 'Internal Server Error' }, { status: 500 })
}
```

redirect()

Produce a response that redirects to a URL.

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

return NextResponse.redirect(new URL('/new', request.url))
```

The URL can be created and modified before being used in the NextResponse.redirect() method. For example, you can use the request.nextUrl property to get the current URL, and then modify it to redirect to a different URL.

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

```
// Given an incoming request...
const loginUrl = new URL('/login', request.url)
// Add ?from=/incoming-url to the /login URL
loginUrl.searchParams.set('from', request.nextUrl.pathname)
// And redirect to the new URL
return NextResponse.redirect(loginUrl)
```

rewrite()

Produce a response that rewrites (proxies) the given URL while preserving the original URL.

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

// Incoming request: /about, browser shows /about
// Rewritten request: /proxy, browser shows /about
return NextResponse.rewrite(new URL('/proxy', request.url))
```

next()

The next() method is useful for Middleware, as it allows you to return early and continue routing.

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

return NextResponse.next()
```

You can also forward headers when producing the response:

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

// Given an incoming request...
const newHeaders = new Headers(request.headers)
// Add a new header
newHeaders.set('x-version', '123')
// And produce a response with the new headers
return NextResponse.next({
  request: {
    // New request headers
    headers: newHeaders,
  },
})
```

notFoundThe notFound function allows you to render the not-found file within a route

segment as well as inject a <meta name="robots" content="noindex" /> tag.
notFound()

Invoking the notFound() function throws a NEXT_NOT_FOUND error and terminates rendering of the route segment in which it was thrown. Specifying a not-found file allows you to gracefully handle such errors by rendering a Not Found UI within the segment.
app/user/[id]/page.js import { notFound } from 'next/navigation'

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

export default async function Profile({ params }) {
  const user = await fetchUser(params.id)

  if (!user) {
    notFound()
  }

  // ...
}
```

Good to know: notFound() does not require you to use return notFound() due to using the TypeScript never type.

Version History

VersionChangesv13.0.0notFound introduced.
redirectThe redirect function allows you to redirect the user to another URL. redirect can be used in Server Components, Client Components, Route Handlers, and Server Actions.
If you need to redirect to a 404, use the notFound function instead.
Parameters

The redirect function accepts two arguments:
redirect(path, type)

ParameterTypeDescriptionpathstringThe URL to redirect to. Can be a relative or absolute path.type'replace' (default) or 'push' (default in Server Actions)The type of redirect to perform.
By default, redirect will use push (adding a new entry to the browser history stack) in Server Actions) and replace (replacing the current URL in the browser history stack) everywhere else. You can override this behavior by specifying the type parameter.
The type parameter has no effect when used in Server Components.
Returns

redirect does not return any value.
Example

Invoking the redirect() function throws a NEXT_REDIRECT error and terminates rendering of the route segment in which it was thrown.
app/team/[id]/page.js import { redirect } from 'next/navigation'

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

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

  // ...
}
```

Good to know: redirect does not require you to use return redirect() as it uses the TypeScript never type.

VersionChangesv13.0.0redirect introduced.
revalidatePathrevalidatePath allows you to revalidate data associated with a specific path. This is useful for scenarios where you want to update your cached data without waiting for a revalidation period to expire.
app/api/revalidate/route.tsTypeScript import { NextRequest, NextResponse } from 'next/server'
import { revalidatePath } from 'next/cache'

```
export async function GET(request: NextRequest) {
  const path = request.nextUrl.searchParams.get('path') || '/'
  revalidatePath(path)
  return NextResponse.json({ revalidated: true, now: Date.now() })
}
```

Good to know:

revalidatePath is available in both Node.js and Edge runtimes.
revalidatePath will revalidate all segments under a dynamic route segment. For example, if you have a dynamic segment /product/[id] and you call revalidatePath('/product/[id]'), then all segments under /product/[id] will be revalidated as requested.
revalidatePath only invalidates the cache when the path is next visited. This means calling revalidatePath with a dynamic route segment will not immediately trigger many revalidations at once. The invalidation only happens when the path is next visited.

Parameters

```
revalidatePath(path: string): void;
```

path: A string representing the filesystem path associated with the data you want to revalidate. This is not the literal route segment (e.g. /product/123) but instead the path on the filesystem (e.g. /product/[id]).

Returns

revalidatePath does not return any value.
Examples

Node.js Runtime

app/api/revalidate/route.tsTypeScript import { NextRequest, NextResponse } from 'next/server'
import { revalidatePath } from 'next/cache'

```
export async function GET(request: NextRequest) {
  const path = request.nextUrl.searchParams.get('path') || '/'
  revalidatePath(path)
  return NextResponse.json({ revalidated: true, now: Date.now() })
}
```

Edge Runtime

app/api/revalidate/route.tsTypeScript import { NextRequest, NextResponse } from 'next/server'
import { revalidatePath } from 'next/cache'

```
export const runtime = 'edge'

export async function GET(request: NextRequest) {
  const path = request.nextUrl.searchParams.get('path') || '/'
  revalidatePath(path)
  return NextResponse.json({ revalidated: true, now: Date.now() })
}
```

revalidateTagrevalidateTag allows you to revalidate data associated with a specific cache tag. This is useful for scenarios where you want to update your cached data without waiting for a revalidation period to expire.
app/api/revalidate/route.tsTypeScript import { NextRequest, NextResponse } from 'next/server'
import { revalidateTag } from 'next/cache'

```
export async function GET(request: NextRequest) {
  const tag = request.nextUrl.searchParams.get('tag')
  revalidateTag(tag)
  return NextResponse.json({ revalidated: true, now: Date.now() })
}
```

Good to know:

revalidateTag is available in both Node.js and Edge runtimes.

Parameters

```
 revalidateTag(tag: string): void;
```

tag: A string representing the cache tag associated with the data you want to revalidate.

You can add tags to fetch as follows:

```
 fetch(url, { next: { tags: [...] } });
```

Returns

revalidateTag does not return any value.

Examples

Node.js Runtime

app/api/revalidate/route.tsTypeScript import { NextRequest, NextResponse } from 'next/server'

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

export async function GET(request: NextRequest) {
  const tag = request.nextUrl.searchParams.get('tag')
  revalidateTag(tag)
  return NextResponse.json({ revalidated: true, now: Date.now() })
}
```

Edge Runtime

app/api/revalidate/route.tsTypeScript import { NextRequest, NextResponse } from 'next/server'

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

export const runtime = 'edge'

export async function GET(request: NextRequest) {
  const tag = request.nextUrl.searchParams.get('tag')
  revalidateTag(tag)
  return NextResponse.json({ revalidated: true, now: Date.now() })
}
```

useParamsuseParams is a Client Component hook that lets you read a route's dynamic params filled in by the current URL.

app/example-client-component.tsxTypeScript 'use client'

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

export default function ExampleClientComponent() {
  const params = useParams()

  // Route -> /shop/[tag]/[item]
  // URL -> /shop/shoes/nike-air-max-97
  // `params` -> { tag: 'shoes', item: 'nike-air-max-97' }
  console.log(params)

  return <></>
}
```

Parameters

```
 const params = useParams()
```
useParams does not take any parameters.
Returns

useParams returns an object containing the current route's filled in dynamic parameters.

Each property in the object is an active dynamic segment.
The properties name is the segment's name, and the properties value is what the segment is filled in with.
The properties value will either be a string or array of string's depending on the type of dynamic segment.
If the route contains no dynamic parameters, useParams returns an empty object.
If used in pages, useParams will return null.

For example:
RouteURLuseParams()app/shop/page.js/shopnullapp/shop/[slug]/page.js/shop/1{ slug: '1' }app/shop/[tag]/[item]/page.js/shop/1/2{ tag: '1', item: '2' }app/shop/[...slug]/page.js/shop/1/2{ slug: ['1', '2'] }
Version History

VersionChangesv13.3.0useParams introduced.
usePathnameusePathname is a Client Component hook that lets you read the current URL's pathname.
app/example-client-component.tsxTypeScript 'use client'

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

export default function ExampleClientComponent() {
  const pathname = usePathname()
  return <p>Current pathname: {pathname}</p>
}
```

usePathname intentionally requires using a Client Component. It's important to note Client Components are not a de-optimization. They are an integral part of the Server Components architecture.
For example, a Client Component with usePathname will be rendered into HTML on the

initial page load. When navigating to a new route, this component does not need to be re-fetched. Instead, the component is downloaded once (in the client JavaScript bundle), and re-renders based on the current state.

Good to know:

Reading the current URL from a Server Component is not supported. This design is intentional to support layout state being preserved across page navigations.
Compatibility mode:

usePathname can return null when a fallback route is being rendered or when a pages directory page has been automatically statically optimized by Next.js and the router is not ready.
Next.js will automatically update your types if it detects both an app and pages directory in your project.

Parameters

```
const pathname = usePathname()
```

usePathname does not take any parameters.
Returns

usePathname returns a string of the current URL's pathname. For example:
URLReturned value'/'/dashboard'/dashboard'/dashboard?v=2'/dashboard'/blog/hello-world'/blog/hello-world'
Examples

Do something in response to a route change

app/example-client-component.tsxTypeScript 'use client'

```
import { usePathname, useSearchParams } from 'next/navigation'

function ExampleClientComponent() {
  const pathname = usePathname()
  const searchParams = useSearchParams()
  useEffect(() => {
    // Do something here...
  }, [pathname, searchParams])
}
```

VersionChangesv13.0.0usePathname introduced.
useReportWebVitals
The useReportWebVitals hook allows you to report Core Web Vitals, and can be used in combination with your analytics service.

app/_components/web-vitals.js 'use client'

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

export function WebVitals() {
  useReportWebVitals((metric) => {
    console.log(metric)
  })
}app/layout.js import { WebVitals } from './_components/web-vitals'

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

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

useReportWebVitals

The metric object passed as the hook's argument consists of a number of properties:

id: Unique identifier for the metric in the context of the current page load
name: The name of the performance metric. Possible values include names of Web Vitals metrics (TTFB, FCP, LCP, FID, CLS) specific to a web application.
delta: The difference between the current value and the previous value of the metric. The value is typically in milliseconds and represents the change in the metric's value over time.
entries: An array of Performance Entries associated with the metric. These entries provide detailed information about the performance events related to the metric.
navigationType: Indicates the type of navigation that triggered the metric collection. Possible values include "navigate", "reload", "back_forward", and "prerender".
rating: A qualitative rating of the metric value, providing an assessment of the performance. Possible values are "good", "needs-improvement", and "poor". The rating is typically determined by comparing the metric value against predefined thresholds that indicate acceptable or suboptimal performance.
value: The actual value or duration of the performance entry, typically in milliseconds. The value provides a quantitative measure of the performance aspect being tracked by the metric. The source of the value depends on the specific metric being measured and can come from various Performance APIs.

Web Vitals

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

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

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

app/components/web-vitals.tsxTypeScript 'use client'

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

```
export function WebVitals() {
  useReportWebVitals((metric) => {
    switch (metric.name) {
      case 'FCP': {
        // handle FCP results
      }
      case 'LCP': {
        // handle LCP results
      }
      // ...
    }
  })
}
```

Usage on Vercel

Vercel Speed Insights are automatically configured on Vercel deployments, and don't require the use of useReportWebVitals. This hook is useful in local development, or if you're using a different analytics service.
Sending results to external systems

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

```
 useReportWebVitals((metric) => {
  const body = JSON.stringify(metric)
  const url = 'https://example.com/analytics'

  // Use `navigator.sendBeacon()` if available, falling back to `fetch()`.
  if (navigator.sendBeacon) {
    navigator.sendBeacon(url, body)
  } else {
    fetch(url, { body, method: 'POST', keepalive: true })
  }
})
```

Good to know: If you use Google Analytics, using the
id value can allow you to construct metric distributions manually (to calculate percentiles,

etc.)

```
 useReportWebVitals(metric => {
  // Use `window.gtag` if you initialized Google Analytics as this example:
  // https://github.com/vercel/next.js/blob/canary/examples/with-google-analytics/pages/
_app.js
  window.gtag('event', metric.name, {
    value: Math.round(metric.name === 'CLS' ? metric.value * 1000 : metric.value), //
values must be integers
    event_label: metric.id, // id unique to current page load
    non_interaction: true, // avoids affecting bounce rate.
  });
}
```

Read more about sending results to Google Analytics.
useRouterThe useRouter hook allows you to programmatically change routes inside Client Components.

Recommendation: Use the <Link> component for navigation unless you have a specific requirement for using useRouter.

app/example-client-component.tsxTypeScript 'use client'

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

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

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

useRouter()

router.push(href: string): Perform a client-side navigation to the provided route. Adds a new entry into the browser’s history stack.
router.replace(href: string): Perform a client-side navigation to the provided route without adding a new entry into the browser’s history stack.

router.refresh(): Refresh the current route. Making a new request to the server, re-fetching data requests, and re-rendering Server Components. The client will merge the updated React Server Component payload without losing unaffected client-side React (e.g. useState) or browser state (e.g. scroll position).
router.prefetch(href: string): Prefetch the provided route for faster client-side transitions.
router.back(): Navigate back to the previous route in the browser’s history stack using soft navigation.
router.forward(): Navigate forwards to the next page in the browser’s history stack using soft navigation.

Good to know:

The push() and replace() methods will perform a soft navigation if the new route has been prefetched, and either, doesn't include dynamic segments or has the same dynamic parameters as the current route.
next/link automatically prefetch routes as they become visible in the viewport.
refresh() could re-produce the same result if fetch requests are cached. Other dynamic functions like cookies and headers could also change the response.

Migrating from the pages directory:

The new useRouter hook should be imported from next/navigation and not next/router
The pathname string has been removed and is replaced by usePathname()
The query object has been removed and is replaced by useSearchParams()
router.events is not currently supported. See below.

View the full migration guide.

Examples

Router Events

You can listen for page changes by composing other Client Component hooks like usePathname and useSearchParams.
app/components/navigation-events.js 'use client'

```
import { useEffect } from 'react'
import { usePathname, useSearchParams } from 'next/navigation'

export function NavigationEvents() {
  const pathname = usePathname()
  const searchParams = useSearchParams()

  useEffect(() => {
    const url = `${pathname}?${searchParams}`
    console.log(url)
    // You can now use the current URL
    // ...
  }, [pathname, searchParams])

  return null
}
```

Which can be imported into a layout.

app/layout.js import { Suspense } from 'react'

```
import { NavigationEvents } from './components/navigation-events'

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

        <Suspense fallback={null}>
          <NavigationEvents />
        </Suspense>
      </body>
    </html>
  )
}
```

Good to know: <NavigationEvents> is wrapped in a Suspense boundary becauseuseSearchParams() causes client-side rendering up to the closest Suspense boundary during static rendering. Learn more.

VersionChangesv13.0.0useRouter from next/navigation introduced.

useSearchParamsuseSearchParams is a Client Component hook that lets you read the current URL's query string.

useSearchParams returns a read-only version of the URLSearchParams interface.

app/dashboard/search-bar.tsxTypeScript 'use client'

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

```
export default function SearchBar() {
  const searchParams = useSearchParams()

  const search = searchParams.get('search')

  // URL -> `/dashboard?search=my-project`
  // `search` -> 'my-project'
  return <>Search: {search}</>
}
```

Parameters

```
 const searchParams = useSearchParams()
```

useSearchParams does not take any parameters.

Returns

useSearchParams returns a read-only version of the URLSearchParams interface, which includes utility methods for reading the URL's query string:

URLSearchParams.get(): Returns the first value associated with the search parameter. For example:
URLsearchParams.get("a")/dashboard?a=1'1'/dashboard?a=''/dashboard?b=3null/ dashboard?a=1&a=2'1' - use getAll() to get all values

URLSearchParams.has(): Returns a boolean value indicating if the given parameter exists. For example:
URLsearchParams.has("a")/dashboard?a=1true/dashboard?b=3false

Learn more about other read-only methods of URLSearchParams, including the getAll(), keys(), values(), entries(), forEach(), and toString().

Good to know:

useSearchParams is a Client Component hook and is not supported in Server Components to prevent stale values during partial rendering.
If an application includes the /pages directory, useSearchParams will return ReadonlyURLSearchParams | null. The null value is for compatibility during migration since search params cannot be known during pre-rendering of a page that doesn't use getServerSideProps

Behavior

Static Rendering

If a route is statically rendered, calling useSearchParams() will cause the tree up to the closest Suspense boundary to be client-side rendered.
This allows a part of the page to be statically rendered while the dynamic part that uses searchParams is client-side rendered.
You can reduce the portion of the route that is client-side rendered by wrapping the component that uses useSearchParams in a Suspense boundary. For example:
app/dashboard/search-bar.tsxTypeScript 'use client'

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

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

  const search = searchParams.get('search')

  // This will not be logged on the server when using static rendering
  console.log(search)

  return <>Search: {search}</>
}
```

app/dashboard/page.tsxTypeScript import { Suspense } from 'react'
import SearchBar from './search-bar'

// This component passed as a fallback to the Suspense boundary

```
// will be rendered in place of the search bar in the initial HTML.
// When the value is available during React hydration the fallback
// will be replaced with the `<SearchBar>` component.
function SearchBarFallback() {
  return <>placeholder</>
}

export default function Page() {
  return (
    <>
      <nav>
        <Suspense fallback={<SearchBarFallback />}>
          <SearchBar />
        </Suspense>
      </nav>
      <h1>Dashboard</h1>
    </>
  )
}
```

Dynamic Rendering

If a route is dynamically rendered, useSearchParams will be available on the server during the initial server render of the Client Component.

Good to know: Setting the dynamic route segment config option to force-dynamic can be used to force dynamic rendering.

For example:
app/dashboard/search-bar.tsxTypeScript 'use client'

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

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

  const search = searchParams.get('search')

  // This will be logged on the server during the initial render
  // and on the client on subsequent navigations.
  console.log(search)
```

```
  return <>Search: {search}</>
}
```

app/dashboard/page.tsxTypeScript import SearchBar from './search-bar'

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

export default function Page() {
  return (
    <>
      <nav>
        <SearchBar />
      </nav>
      <h1>Dashboard</h1>
    </>
  )
}
```

Server Components

Pages

To access search params in Pages (Server Components), use the searchParams prop.
Layouts

Unlike Pages, Layouts (Server Components) do not receive the searchParams prop. This is because a shared layout is not re-rendered during navigation which could lead to stale searchParams between navigations. View detailed explanation.
Instead, use the Page searchParams prop or the useSearchParams hook in a Client Component, which is re-rendered on the client with the latest searchParams.
Examples

Updating searchParams

You can use useRouter or Link to set new searchParams. After a navigation is performed, the current page.js will receive an updated searchParams prop.
app/example-client-component.tsxTypeScript export default function ExampleClientComponent() {
```
  const router = useRouter()
  const pathname = usePathname()
  const searchParams = useSearchParams()!

  // Get a new searchParams string by merging the current
  // searchParams with a provided key/value pair
  const createQueryString = useCallback(
    (name: string, value: string) => {
      const params = new URLSearchParams(searchParams)
      params.set(name, value)

      return params.toString()
    },
    [searchParams]
  )

  return (
    <>
      <p>Sort By</p>

      {/* using useRouter */}
      <button
        onClick={() => {
          // <pathname>?sort=asc
          router.push(pathname + '?' + createQueryString('sort', 'asc'))
        }}
      >
        ASC
      </button>

      {/* using <Link> */}
      <Link
        href={
          // <pathname>?sort=desc
```

```
          pathname + '?' + createQueryString('sort', 'desc')
        }
      >
        DESC
      </Link>
    </>
  )
}
```

Version History

Version	Changes
v13.0.0	useSearchParams introduced.

useSelectedLayoutSegmentuseSelectedLayoutSegment is a Client Component hook that lets you read the active route segment one level below the Layout it is called from. It is useful for navigation UI, such as tabs inside a parent layout that change style depending on the active child segment.
app/example-client-component.tsxTypeScript 'use client'

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

export default function ExampleClientComponent() {
  const segment = useSelectedLayoutSegment()

  return <p>Active segment: {segment}</p>
}
```

Good to know:

Since useSelectedLayoutSegment is a Client Component hook, and Layouts are Server Components by default, useSelectedLayoutSegment is usually called via a Client Component that is imported into a Layout.
useSelectedLayoutSegment only returns the segment one level down. To return all active segments, see useSelectedLayoutSegments

Parameters

```
 const segment = useSelectedLayoutSegment()
```

useSelectedLayoutSegment does not take any parameters.

Returns

useSelectedLayoutSegment returns a string of the active segment or null if one doesn't exist.

For example, given the Layouts and URLs below, the returned segment would be:

Layout	Visited URL	Returned Segment
app/layout.js	/	null
app/dashboard/layout.js	/dashboard	'dashboard'
app/dashboard/layout.js	/dashboard	null
app/dashboard/layout.js	/dashboard/settings	'settings'
app/dashboard/layout.js	/dashboard/analytics	'analytics'
app/dashboard/layout.js	/dashboard/analytics/monthly	'analytics'

Examples

Creating an active link component

You can use useSelectedLayoutSegment to create an active link component that changes style depending on the active segment. For example, a featured posts list in the sidebar of a blog:

app/blog/blog-nav-link.tsxTypeScript 'use client'

```
import Link from 'next/link'
import { useSelectedLayoutSegment } from 'next/navigation'

// This *client* component will be imported into a blog layout
export default function BlogNavLink({
  slug,
  children,
}: {
  slug: string
  children: React.ReactNode
}) {
  // Navigating to `/blog/hello-world` will return 'hello-world'
  // for the selected layout segment
  const segment = useSelectedLayoutSegment()
```

```
  const isActive = slug === segment

  return (
    <Link
      href={`/blog/${slug}`}
      // Change style depending on whether the link is active
      style={{ fontWeight: isActive ? 'bold' : 'normal' }}
    >
      {children}
    </Link>
  )
}
```

app/blog/layout.tsxTypeScript // Import the Client Component into a parent Layout (Server Component)

```
import { BlogNavLink } from './blog-nav-link'
import getFeaturedPosts from './get-featured-posts'

export default async function Layout({
  children,
}: {
  children: React.ReactNode
}) {
  const featuredPosts = await getFeaturedPosts()
  return (
    <div>
      {featuredPosts.map((post) => (
        <div key={post.id}>
          <BlogNavLink slug={post.slug}>{post.title}</BlogNavLink>
        </div>
      ))}
      <div>{children}</div>
    </div>
  )
}
```

Version History

VersionChangesv13.0.0useSelectedLayoutSegment introduced.
useSelectedLayoutSegmentsuseSelectedLayoutSegments is a Client Component hook that lets you read the active route segments below the Layout it is called from.
It is useful for creating UI in parent Layouts that need knowledge of active child

segments such as breadcrumbs.
app/example-client-component.tsxTypeScript 'use client'

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

export default function ExampleClientComponent() {
  const segments = useSelectedLayoutSegments()

  return (
    <ul>
      {segments.map((segment, index) => (
        <li key={index}>{segment}</li>
      ))}
    </ul>
  )
}
```

Good to know:

Since useSelectedLayoutSegments is a Client Component hook, and Layouts are Server Components by default, useSelectedLayoutSegments is usually called via a Client Component that is imported into a Layout.
The returned segments include Route Groups, which you might not want to be included in your UI. You can use the filter() array method to remove items that start with a bracket.

Parameters

```
 const segments = useSelectedLayoutSegments()
```

useSelectedLayoutSegments does not take any parameters.
Returns

useSelectedLayoutSegments returns an array of strings containing the active segments one level down from the layout the hook was called from. Or an empty array if none exist.
For example, given the Layouts and URLs below, the returned segments would be:

LayoutVisited URLReturned Segmentsapp/layout.js/[]app/layout.js/dashboard['dashboard']app/layout.js/dashboard/settings['dashboard', 'settings']app/dashboard/layout.js/dashboard[]app/dashboard/layout.js/dashboard/settings['settings']
Version History

VersionChangesv13.0.0useSelectedLayoutSegments introduced.
next.config.js Options
Next.js can be configured through a next.config.js file in the root of your project directory.
next.config.js /** @type {import('next').NextConfig} */
const nextConfig = {
  /* config options here */
}

module.exports = nextConfig
This page documents all the available configuration options:
appDir
Good to know: This option is no longer needed as of Next.js 13.4. The App Router is now stable.

The App Router (app directory) enables support for layouts, Server Components, streaming, and colocated data fetching.
Using the app directory will automatically enable React Strict Mode. Learn how to incrementally adopt app.
assetPrefix

Attention: Deploying to Vercel automatically configures a global CDN for your Next.js project.
You do not need to manually setup an Asset Prefix.

Good to know: Next.js 9.5+ added support for a customizable Base Path, which is better suited for hosting your application on a sub-path like /docs.
We do not suggest you use a custom Asset Prefix for this use case.

To set up a CDN, you can set up an asset prefix and configure your CDN's origin to resolve to the domain that Next.js is hosted on.
Open next.config.js and add the assetPrefix config:
next.config.js const isProd = process.env.NODE_ENV === 'production'

module.exports = {

```
  // Use the CDN in production and localhost for development.
  assetPrefix: isProd ? 'https://cdn.mydomain.com' : undefined,
}
```

Next.js will automatically use your asset prefix for the JavaScript and CSS files it loads from the /_next/ path (.next/static/ folder). For example, with the above configuration, the following request for a JS chunk:

/_next/static/chunks/4b9b41aaa062cbbfeff4add70f256968c51ece5d.4d708494b3aed70c04f0.js

Would instead become:

https://cdn.mydomain.com/_next/static/chunks/4b9b41aaa062cbbfeff4add70f256968c51ece5d.4d708494b3aed70c04f0.js

The exact configuration for uploading your files to a given CDN will depend on your CDN of choice. The only folder you need to host on your CDN is the contents of .next/static/, which should be uploaded as _next/static/ as the above URL request indicates. Do not upload the rest of your .next/ folder, as you should not expose your server code and other configuration to the public.

While assetPrefix covers requests to _next/static, it does not influence the following paths:

Files in the public folder; if you want to serve those assets over a CDN, you'll have to introduce the prefix yourself

basePath

To deploy a Next.js application under a sub-path of a domain you can use the basePath config option.

basePath allows you to set a path prefix for the application. For example, to use /docs instead of '' (an empty string, the default), open next.config.js and add the basePath config:

next.config.js

```
module.exports = {
  basePath: '/docs',
}
```

Good to know: This value must be set at build time and cannot be changed without re-building as the value is inlined in the client-side bundles.

Links

When linking to other pages using next/link and next/router the basePath will be automatically applied.

For example, using /about will automatically become /docs/about when basePath is set

to /docs.

```
 export default function HomePage() {
  return (
    <>
      <Link href="/about">About Page</Link>
    </>
  )
}
```

Output html:

```
 <a href="/docs/about">About Page</a>
```

This makes sure that you don't have to change all links in your application when changing the basePath value.

Images

When using the next/image component, you will need to add the basePath in front of src.

For example, using /docs/me.png will properly serve your image when basePath is set to /docs.

```
 import Image from 'next/image'

function Home() {
  return (
    <>
      <h1>My Homepage</h1>
      <Image
        src="/docs/me.png"
        alt="Picture of the author"
        width={500}
        height={500}
      />
      <p>Welcome to my homepage!</p>
    </>
  )
}

export default Home
```

compress

Next.js provides gzip compression to compress rendered content and static files. In general you will want to enable compression on a HTTP proxy like nginx, to offload load from the Node.js process.

To disable compression, open next.config.js and disable the compress config:

next.config.js module.exports = {
  compress: false,
}
devIndicators
When you edit your code, and Next.js is compiling the application, a compilation indicator appears in the bottom right corner of the page.
Good to know: This indicator is only present in development mode and will not appear when building and running the app in production mode.
In some cases this indicator can be misplaced on your page, for example, when conflicting with a chat launcher. To change its position, open next.config.js and set the buildActivityPosition in the devIndicators object to bottom-right (default), bottom-left, top-right or top-left:next.config.js module.exports = {
  devIndicators: {
    buildActivityPosition: 'bottom-right',
  },
}In some cases this indicator might not be useful for you. To remove it, open next.config.js and disable the buildActivity config in devIndicators object:next.config.js module.exports = {
  devIndicators: {
    buildActivity: false,
  },
}
distDir
You can specify a name to use for a custom build directory to use instead of .next.
Open next.config.js and add the distDir config:
next.config.js module.exports = {
  distDir: 'build',
}
Now if you run next build Next.js will use build instead of the default .next folder.

distDir should not leave your project directory. For example, ../build is an invalid directory.
env

Since the release of Next.js 9.4 we now have a more intuitive and ergonomic experience for adding environment variables. Give it a try!

Examples
With env

Good to know: environment variables specified in this way will always be included in the JavaScript bundle, prefixing the environment variable name with NEXT_PUBLIC_ only has an effect when specifying them through the environment or .env files.

To add environment variables to the JavaScript bundle, open next.config.js and add the env config:

```
next.config.js module.exports = {
  env: {
    customKey: 'my-value',
  },
}
```

Now you can access process.env.customKey in your code. For example:

```
 function Page() {
  return <h1>The value of customKey is: {process.env.customKey}</h1>
}

export default Page
```

Next.js will replace process.env.customKey with 'my-value' at build time. Trying to destructure process.env variables won't work due to the nature of webpack DefinePlugin.

For example, the following line:

```
 return <h1>The value of customKey is: {process.env.customKey}</h1>
```

Will end up being:

```
 return <h1>The value of customKey is: {'my-value'}</h1>
```

eslint

When ESLint is detected in your project, Next.js fails your production build (next build) when errors are present.

If you'd like Next.js to produce production code even when your application has ESLint errors, you can disable the built-in linting step completely. This is not recommended unless you already have ESLint configured to run in a separate part of your workflow (for example, in CI or a pre-commit hook).

Open next.config.js and enable the ignoreDuringBuilds option in the eslint config:

```
next.config.js module.exports = {
  eslint: {
    // Warning: This allows production builds to successfully complete even if
    // your project has ESLint errors.
    ignoreDuringBuilds: true,
  },
}
```

exportPathMap (Deprecated)

This feature is exclusive to next export and currently deprecated in favor of getStaticPaths with pages or generateStaticParams with app.

Examples

Static Export

exportPathMap allows you to specify a mapping of request paths to page destinations, to be used during export. Paths defined in exportPathMap will also be available when

using next dev.
Let's start with an example, to create a custom exportPathMap for an app with the following pages:

pages/index.js
pages/about.js
pages/post.js

Open next.config.js and add the following exportPathMap config:

```
next.config.js module.exports = {
  exportPathMap: async function (
    defaultPathMap,
    { dev, dir, outDir, distDir, buildId }
  ) {
    return {
      '/': { page: '/' },
      '/about': { page: '/about' },
      '/p/hello-nextjs': { page: '/post', query: { title: 'hello-nextjs' } },
      '/p/learn-nextjs': { page: '/post', query: { title: 'learn-nextjs' } },
      '/p/deploy-nextjs': { page: '/post', query: { title: 'deploy-nextjs' } },
    }
  },
}
```

Good to know: the query field in exportPathMap cannot be used with automatically statically optimized pages or getStaticProps pages as they are rendered to HTML files at build-time and additional query information cannot be provided during next export.

The pages will then be exported as HTML files, for example, /about will become /about.html.
exportPathMap is an async function that receives 2 arguments: the first one is defaultPathMap, which is the default map used by Next.js. The second argument is an object with:

dev - true when exportPathMap is being called in development. false when running next export. In development exportPathMap is used to define routes.
dir - Absolute path to the project directory
outDir - Absolute path to the out/ directory (configurable with -o). When dev is true the value of outDir will be null.
distDir - Absolute path to the .next/ directory (configurable with the distDir config)
buildId - The generated build id

The returned object is a map of pages where the key is the pathname and the value is an object that accepts the following fields:

page: String - the page inside the pages directory to render

query: Object - the query object passed to getInitialProps when prerendering. Defaults to {}

The exported pathname can also be a filename (for example, /readme.md), but you may need to set the Content-Type header to text/html when serving its content if it is different than .html.

Adding a trailing slash

It is possible to configure Next.js to export pages as index.html files and require trailing slashes, /about becomes /about/index.html and is routable via /about/. This was the default behavior prior to Next.js 9.
To switch back and add a trailing slash, open next.config.js and enable the trailingSlash config:
next.config.js module.exports = {
  trailingSlash: true,
}
Customizing the output directory

next export will use out as the default output directory, you can customize this using the -o argument, like so:

Terminal next export -o outdir

Warning: Using exportPathMap is deprecated and is overridden by getStaticPaths inside pages. We don't recommend using them together.
generateBuildId
Next.js uses a constant id generated at build time to identify which version of your application is being served. This can cause problems in multi-server deployments when next build is run on every server. In order to keep a consistent build id between builds you can provide your own build id.
Open next.config.js and add the generateBuildId function:
next.config.js module.exports = {
  generateBuildId: async () => {
    // You can, for example, get the latest git commit hash here
    return 'my-build-id'
  },

```
}
```
generateEtags
Next.js will generate etags for every page by default. You may want to disable etag generation for HTML pages depending on your cache strategy.
Open next.config.js and disable the generateEtags option:
next.config.js
```
module.exports = {
  generateEtags: false,
}
```
headers
Headers allow you to set custom HTTP headers on the response to an incoming request on a given path.
To set custom HTTP headers you can use the headers key in next.config.js:
next.config.js
```
module.exports = {
  async headers() {
    return [
      {
        source: '/about',
        headers: [
          {
            key: 'x-custom-header',
            value: 'my custom header value',
          },
          {
            key: 'x-another-custom-header',
            value: 'my other custom header value',
          },
        ],
      },
    ]
  },
}
```
headers is an async function that expects an array to be returned holding objects with source and headers properties:

source is the incoming request path pattern.
headers is an array of response header objects, with key and value properties.
basePath: false or undefined - if false the basePath won't be included when matching, can be used for external rewrites only.
locale: false or undefined - whether the locale should not be included when matching.
has is an array of has objects with the type, key and value properties.
missing is an array of missing objects with the type, key and value properties.

Headers are checked before the filesystem which includes pages and /public files.
Header Overriding Behavior

If two headers match the same path and set the same header key, the last header key will override the first. Using the below headers, the path /hello will result in the header x-hello being world due to the last header value set being world.

next.config.js
```
module.exports = {
  async headers() {
    return [
      {
        source: '/:path*',
        headers: [
          {
            key: 'x-hello',
            value: 'there',
          },
        ],
      },
      {
        source: '/hello',
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
    ]
  },
}
```

Path Matching

Path matches are allowed, for example /blog/:slug will match /blog/hello-world (no nested paths):

next.config.js
```
module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:slug',
        headers: [
          {
            key: 'x-slug',
```

```
          value: ':slug', // Matched parameters can be used in the value
        },
        {
          key: 'x-slug-:slug', // Matched parameters can be used in the key
          value: 'my other custom header value',
        },
      ],
    },
  ]
 },
}
```

Wildcard Path Matching

To match a wildcard path you can use * after a parameter, for example /blog/:slug* will match /blog/a/b/c/d/hello-world:

```
next.config.js module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:slug*',
        headers: [
          {
            key: 'x-slug',
            value: ':slug*', // Matched parameters can be used in the value
          },
          {
            key: 'x-slug-:slug*', // Matched parameters can be used in the key
            value: 'my other custom header value',
          },
        ],
      },
    ]
  },
}
```

Regex Path Matching

To match a regex path you can wrap the regex in parenthesis after a parameter, for example /blog/:slug(\\d{1,}) will match /blog/123 but not /blog/abc:

```
next.config.js module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:post(\\d{1,})',
        headers: [
          {
            key: 'x-post',
            value: ':post',
          },
        ],
      },
    ]
  },
}
```

The following characters (,), {, }, :, *, +, ? are used for regex path matching, so when used in the source as non-special values they must be escaped by adding \\ before them:

```
next.config.js module.exports = {
  async headers() {
    return [
      {
        // this will match `/english(default)/something` being requested
        source: '/english\\(default\\)/:slug',
        headers: [
          {
            key: 'x-header',
            value: 'value',
          },
        ],
      },
    ]
  },
}
```

Header, Cookie, and Query Matching

To only apply a header when header, cookie, or query values also match the has field or don't match the missing field can be used. Both the source and all has items must match and all missing items must not match for the header to be applied.
has and missing items can have the following fields:

type: String - must be either header, cookie, host, or query.

key: String - the key from the selected type to match against.
value: String or undefined - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value first-(?<paramName>.*) is used for first-second then second will be usable in the destination with :paramName.

```
next.config.js module.exports = {
  async headers() {
    return [
      // if the header `x-add-header` is present,
      // the `x-another-header` header will be applied
      {
        source: '/:path*',
        has: [
          {
            type: 'header',
            key: 'x-add-header',
          },
        ],
        headers: [
          {
            key: 'x-another-header',
            value: 'hello',
          },
        ],
      },
      // if the header `x-no-header` is not present,
      // the `x-another-header` header will be applied
      {
        source: '/:path*',
        missing: [
          {
            type: 'header',
            key: 'x-no-header',
          },
        ],
        headers: [
          {
            key: 'x-another-header',
            value: 'hello',
          },
        ],
      },
      // if the source, query, and cookie are matched,
      // the `x-authorized` header will be applied
      {
```

```
  source: '/specific/:path*',
  has: [
    {
      type: 'query',
      key: 'page',
      // the page value will not be available in the
      // header key/values since value is provided and
      // doesn't use a named capture group e.g. (?<page>home)
      value: 'home',
    },
    {
      type: 'cookie',
      key: 'authorized',
      value: 'true',
    },
  ],
  headers: [
    {
      key: 'x-authorized',
      value: ':authorized',
    },
  ],
},
// if the header `x-authorized` is present and
// contains a matching value, the `x-another-header` will be applied
{
  source: '/:path*',
  has: [
    {
      type: 'header',
      key: 'x-authorized',
      value: '(?<authorized>yes|true)',
    },
  ],
  headers: [
    {
      key: 'x-another-header',
      value: ':authorized',
    },
  ],
},
// if the host is `example.com`,
// this header will be applied
{
  source: '/:path*',
  has: [
```

```
        {
          type: 'host',
          value: 'example.com',
        },
      ],
      headers: [
        {
          key: 'x-another-header',
          value: ':authorized',
        },
      ],
    },
  ]
 },
}
```

Headers with basePath support

When leveraging basePath support with headers each source is automatically prefixed with the basePath unless you add basePath: false to the header:

```
next.config.js module.exports = {
  basePath: '/docs',

  async headers() {
    return [
      {
        source: '/with-basePath', // becomes /docs/with-basePath
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        source: '/without-basePath', // is not modified since basePath: false is set
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
        basePath: false,
```

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

Headers with i18n support

When leveraging i18n support with headers each source is automatically prefixed to handle the configured locales unless you add locale: false to the header. If locale: false is used you must prefix the source with a locale for it to be matched correctly.

```
next.config.js module.exports = {
  i18n: {
    locales: ['en', 'fr', 'de'],
    defaultLocale: 'en',
  },

  async headers() {
    return [
      {
        source: '/with-locale', // automatically handles all locales
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // does not handle locales automatically since locale: false is set
        source: '/nl/with-locale-manual',
        locale: false,
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // this matches '/' since `en` is the defaultLocale
        source: '/en',
        locale: false,
```

```
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // this gets converted to /(en|fr|de)/(.*) so will not match the top-level
        // `/` or `/fr` routes like /:path* would
        source: '/(.*)',
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
    ]
  },
}
```

Cache-Control

You can set the Cache-Control header in your Next.js API Routes by using the res.setHeader method:

```
pages/api/user.js export default function handler(req, res) {
  res.setHeader('Cache-Control', 's-maxage=86400')
  res.status(200).json({ name: 'John Doe' })
}
```

You cannot set Cache-Control headers in next.config.js file as these will be overwritten in production to ensure that API Routes and static assets are cached effectively.

If you need to revalidate the cache of a page that has been statically generated, you can do so by setting the revalidate prop in the page's getStaticProps function.

Options

X-DNS-Prefetch-Control

This header controls DNS prefetching, allowing browsers to proactively perform domain name resolution on external links, images, CSS, JavaScript, and more. This prefetching is performed in the background, so the DNS is more likely to be resolved by the time the referenced items are needed. This reduces latency when the user clicks a link.

```
 {
  key: 'X-DNS-Prefetch-Control',
  value: 'on'
}
```

Strict-Transport-Security

This header informs browsers it should only be accessed using HTTPS, instead of using HTTP. Using the configuration below, all present and future subdomains will use HTTPS for a max-age of 2 years. This blocks access to pages or subdomains that can only be served over HTTP.
If you're deploying to Vercel, this header is not necessary as it's automatically added to all deployments unless you declare headers in your next.config.js.

```
 {
  key: 'Strict-Transport-Security',
  value: 'max-age=63072000; includeSubDomains; preload'
}
```

X-XSS-Protection

This header stops pages from loading when they detect reflected cross-site scripting (XSS) attacks. Although this protection is not necessary when sites implement a strong Content-Security-Policy disabling the use of inline JavaScript ('unsafe-inline'), it can still provide protection for older web browsers that don't support CSP.

```
 {
  key: 'X-XSS-Protection',
  value: '1; mode=block'
}
```

X-Frame-Options

This header indicates whether the site should be allowed to be displayed within an iframe. This can prevent against clickjacking attacks. This header has been superseded by CSP's frame-ancestors option, which has better support in modern browsers.

```
 {
  key: 'X-Frame-Options',
  value: 'SAMEORIGIN'
}
```

Permissions-Policy

This header allows you to control which features and APIs can be used in the browser. It was previously named Feature-Policy. You can view the full list of permission options here.

```
 {
  key: 'Permissions-Policy',
  value: 'camera=(), microphone=(), geolocation=(), browsing-topics=()'
}
```

X-Content-Type-Options

This header prevents the browser from attempting to guess the type of content if the Content-Type header is not explicitly set. This can prevent XSS exploits for websites that allow users to upload and share files. For example, a user trying to download an image, but having it treated as a different Content-Type like an executable, which could be malicious. This header also applies to downloading browser extensions. The only valid value for this header is nosniff.

```
 {
  key: 'X-Content-Type-Options',
  value: 'nosniff'
}
```

Referrer-Policy

This header controls how much information the browser includes when navigating from the current website (origin) to another. You can read about the different options here.

```
 {
  key: 'Referrer-Policy',
  value: 'origin-when-cross-origin'
}
```

Content-Security-Policy

This header helps prevent cross-site scripting (XSS), clickjacking and other code injection attacks. Content Security Policy (CSP) can specify allowed origins for content including scripts, stylesheets, images, fonts, objects, media (audio, video), iframes, and more.

You can read about the many different CSP options here.

You can add Content Security Policy directives using a template string.

```
 // Before defining your Security Headers
// add Content Security Policy directives using a template string.

const ContentSecurityPolicy = `
  default-src 'self';
  script-src 'self';
  child-src example.com;
  style-src 'self' example.com;
  font-src 'self';
`
```

When a directive uses a keyword such as self, wrap it in single quotes ''.

In the header's value, replace the new line with a space.

```
 {
  key: 'Content-Security-Policy',
  value: ContentSecurityPolicy.replace(/\s{2,}/g, ' ').trim()
}
```

Version History

VersionChangesv13.3.0missing added.v10.2.0has added.v9.5.0Headers added.

httpAgentOptions

In Node.js versions prior to 18, Next.js automatically polyfills fetch() with node-fetch and enables HTTP Keep-Alive by default.

To disable HTTP Keep-Alive for all fetch() calls on the server-side, open next.config.js and add the httpAgentOptions config:

next.config.js module.exports = {

```
  httpAgentOptions: {
```

```
    keepAlive: false,
  },
}
```

images

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure next.config.js with the following:

```
next.config.js module.exports = {
  images: {
    loader: 'custom',
    loaderFile: './my/image/loader.js',
  },
}
```

This loaderFile must point to a file relative to the root of your Next.js application. The file must export a default function that returns a string, for example:

```
 export default function myImageLoader({ src, width, quality }) {
  return `https://example.com/${src}?w=${width}&q=${quality || 75}`
}
```

Alternatively, you can use the loader prop to pass the function to each instance of next/image.

Example Loader Configuration

Akamai
Cloudinary
Cloudflare
Contentful
Fastly
Gumlet
ImageEngine
Imgix
Thumbor
Supabase

Akamai

```
 // Docs: https://techdocs.akamai.com/ivm/reference/test-images-on-demand
export default function akamaiLoader({ src, width, quality }) {
```

```
  return `https://example.com/${src}?imwidth=${width}`
}
```
Cloudinary

```
 // Demo: https://res.cloudinary.com/demo/image/upload/w_300,c_limit,q_auto/turtles.jpg
export default function cloudinaryLoader({ src, width, quality }) {
  const params = ['f_auto', 'c_limit', `w_${width}`, `q_${quality || 'auto'}`]
  return `https://example.com/${params.join(',')}${src}`
}
```
Cloudflare

```
 // Docs: https://developers.cloudflare.com/images/url-format
export default function cloudflareLoader({ src, width, quality }) {
  const params = [`width=${width}`, `quality=${quality || 75}`, 'format=auto']
  return `https://example.com/cdn-cgi/image/${params.join(',')}/${src}`
}
```
Contentful

```
 // Docs: https://www.contentful.com/developers/docs/references/images-api/
export default function contentfulLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  url.searchParams.set('fm', 'webp')
  url.searchParams.set('w', width.toString())
  url.searchParams.set('q', (quality || 75).toString())
  return url.href
}
```
Fastly

```
 // Docs: https://developer.fastly.com/reference/io/
export default function fastlyLoader({ src, width, quality }) {
```

```
  const url = new URL(`https://example.com${src}`)
  url.searchParams.set('auto', 'webp')
  url.searchParams.set('width', width.toString())
  url.searchParams.set('quality', (quality || 75).toString())
  return url.href
}
```

Gumlet

```
 // Docs: https://docs.gumlet.com/reference/image-transform-size
export default function gumletLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  url.searchParams.set('format', 'auto')
  url.searchParams.set('w', width.toString())
  url.searchParams.set('q', (quality || 75).toString())
  return url.href
}
```

ImageEngine

```
 // Docs: https://support.imageengine.io/hc/en-us/articles/360058880672-Directives
export default function imageengineLoader({ src, width, quality }) {
  const compression = 100 - (quality || 50)
  const params = [`w_${width}`, `cmpr_${compression}`)]
  return `https://example.com${src}?imgeng=/${params.join('/')`
}
```

Imgix

```
 // Demo: https://static.imgix.net/daisy.png?format=auto&fit=max&w=300
export default function imgixLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  const params = url.searchParams
  params.set('auto', params.getAll('auto').join(',') || 'format')
  params.set('fit', params.get('fit') || 'max')
  params.set('w', params.get('w') || width.toString())
  params.set('q', (quality || 50).toString())
```

```
  return url.href
}
```
Thumbor

```
 // Docs: https://thumbor.readthedocs.io/en/latest/
export default function thumborLoader({ src, width, quality }) {
  const params = [`${width}x0`, `filters:quality(${quality || 75})`]
  return `https://example.com${params.join('/')}${src}`
}
```
Supabase

```
 // Docs: https://supabase.com/docs/guides/storage/image-transformations#nextjs-loader
export default function supabaseLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  url.searchParams.set('width', width.toString())
  url.searchParams.set('quality', (quality || 75).toString())
  return url.href
}
```
incrementalCacheHandlerPathIn Next.js, the default cache handler uses the filesystem cache. This requires no configuration, however, you can customize the cache handler by using the incrementalCacheHandlerPath field in next.config.js.

```
next.config.js module.exports = {
  experimental: {
    incrementalCacheHandlerPath: require.resolve('./cache-handler.js'),
  },
}
```
Here's an example of a custom cache handler:

```
cache-handler.js const cache = new Map()

module.exports = class CacheHandler {
  constructor(options) {
    this.options = options
    this.cache = {}
  }

  async get(key) {
    return cache.get(key)
```

```
  }

  async set(key, data) {
    cache.set(key, {
      value: data,
      lastModified: Date.now(),
    })
  }
}
```

API Reference

The cache handler can implement the following methods: get, set, and revalidateTag.

get()

ParameterTypeDescriptionkeystringThe key to the cached value.

Returns the cached value or null if not found.

set()

ParameterTypeDescriptionkeystringThe key to store the data under.dataData or nullThe data to be cached.

Returns Promise<void>.

revalidateTag()

ParameterTypeDescriptiontagstringThe cache tag to revalidate.

Returns Promise<void>. Learn more about revalidating data or the revalidateTag() function.

mdxRsFor use with @next/mdx. Compile MDX files using the new Rust compiler.

next.config.js const withMDX = require('@next/mdx')()

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

```
const nextConfig = {
  pageExtensions: ['ts', 'tsx', 'mdx'],
  experimental: {
    mdxRs: true,
  },
}

module.exports = withMDX(nextConfig)
```

onDemandEntries

Next.js exposes some options that give you some control over how the server will dispose or keep in memory built pages in development.

To change the defaults, open next.config.js and add the onDemandEntries config:

next.config.js module.exports = {

```
  onDemandEntries: {
    // period (in ms) where the server will keep pages in the buffer
    maxInactiveAge: 25 * 1000,
    // number of pages that should be kept simultaneously without being disposed
    pagesBufferLength: 2,
  },
}
```

output

During a build, Next.js will automatically trace each page and its dependencies to determine all of the files that are needed for deploying a production version of your application.

This feature helps reduce the size of deployments drastically. Previously, when deploying with Docker you would need to have all files from your package's dependencies installed to run next start. Starting with Next.js 12, you can leverage Output File Tracing in the .next/ directory to only include the necessary files. Furthermore, this removes the need for the deprecated serverless target which can cause various issues and also creates unnecessary duplication.

How it Works

During next build, Next.js will use @vercel/nft to statically analyze import, require, and fs usage to determine all files that a page might load.

Next.js' production server is also traced for its needed files and output at .next/next-server.js.nft.json which can be leveraged in production.

To leverage the .nft.json files emitted to the .next output directory, you can read the list of files in each trace that are relative to the .nft.json file and then copy them to your deployment location.

Automatically Copying Traced Files

Next.js can automatically create a standalone folder that copies only the necessary files for a production deployment including select files in node_modules.
To leverage this automatic copying you can enable it in your next.config.js:
next.config.js module.exports = {
  output: 'standalone',
}
This will create a folder at .next/standalone which can then be deployed on its own without installing node_modules.
Additionally, a minimal server.js file is also output which can be used instead of next start. This minimal server does not copy the public or .next/static folders by default as these should ideally be handled by a CDN instead, although these folders can be copied to the standalone/public and standalone/.next/static folders manually, after which server.js file will serve these automatically.

Good to know:

next.config.js is read during next build and serialized into the server.js output file. If the legacy serverRuntimeConfig or publicRuntimeConfig options are being used, the values will be specific to values at build time.
If your project uses Image Optimization with the default loader, you must install sharp as a dependency:

Terminal npm i sharp
Terminal yarn add sharp
Terminal pnpm add sharp
Caveats

While tracing in monorepo setups, the project directory is used for tracing by default. For next build packages/web-app, packages/web-app would be the tracing root and any files outside of that folder will not be included. To include files outside of this folder you can set experimental.outputFileTracingRoot in your next.config.js.

packages/web-app/next.config.js module.exports = {
  experimental: {
    // this includes files from the monorepo base two directories up
    outputFileTracingRoot: path.join(__dirname, '../../'),

```
  },
}
```

There are some cases in which Next.js might fail to include required files, or might incorrectly include unused files. In those cases, you can leverage experimental.outputFileTracingExcludes and experimental.outputFileTracingIncludes respectively in next.config.js. Each config accepts an object with minimatch globs for the key to match specific pages and a value of an array with globs relative to the project's root to either include or exclude in the trace.

```
next.config.js module.exports = {
  experimental: {
    outputFileTracingExcludes: {
      '/api/hello': ['./un-necessary-folder/**/*'],
    },
    outputFileTracingIncludes: {
      '/api/another': ['./necessary-folder/**/*'],
    },
  },
}
```

Currently, Next.js does not do anything with the emitted .nft.json files. The files must be read by your deployment platform, for example Vercel, to create a minimal deployment. In a future release, a new command is planned to utilize these .nft.json files.

Experimental turbotrace

Tracing dependencies can be slow because it requires very complex computations and analysis. We created turbotrace in Rust as a faster and smarter alternative to the JavaScript implementation.
To enable it, you can add the following configuration to your next.config.js:

```
next.config.js module.exports = {
  experimental: {
    turbotrace: {
      // control the log level of the turbotrace, default is `error`
      logLevel?:
      | 'bug'
      | 'fatal'
      | 'error'
      | 'warning'
      | 'hint'
      | 'note'
```

```
      | 'suggestions'
      | 'info',
      // control if the log of turbotrace should contain the details of the analysis, default is
`false`
      logDetail?: boolean
      // show all log messages without limit
      // turbotrace only show 1 log message for each categories by default
      logAll?: boolean
      // control the context directory of the turbotrace
      // files outside of the context directory will not be traced
      // set the `experimental.outputFileTracingRoot` has the same effect
      // if the `experimental.outputFileTracingRoot` and this option are both set, the
`experimental.turbotrace.contextDirectory` will be used
      contextDirectory?: string
      // if there is `process.cwd()` expression in your code, you can set this option to tell
`turbotrace` the value of `process.cwd()` while tracing.
      // for example the require(process.cwd() + '/package.json') will be traced as require('/
path/to/cwd/package.json')
      processCwd?: string
      // control the maximum memory usage of the `turbotrace`, in `MB`, default is `6000`.
      memoryLimit?: number
    },
  },
}
```

pageExtensions

By default, Next.js accepts files with the following extensions: .tsx, .ts, .jsx, .js. This can be modified to allow other extensions like markdown (.md, .mdx).next.config.js const withMDX = require('@next/mdx')()

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  pageExtensions: ['ts', 'tsx', 'mdx'],
  experimental: {
    mdxRs: true,
  },
}

module.exports = withMDX(nextConfig)
```

For custom advanced configuration of Next.js, you can create a next.config.js or next.config.mjs file in the root of your project directory (next to package.json).next.config.js is a regular Node.js module, not a JSON file. It gets used by the Next.js server and build phases, and it's not included in the browser build.Take a look at the following next.config.js example:next.config.js

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

```
}

module.exports = nextConfigIf you need ECMAScript modules, you can use next.config.mjs:next.config.mjs /**
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  /* config options here */
}

export default nextConfigYou can also use a function:next.config.mjs module.exports = (phase, { defaultConfig }) => {
  /**
   * @type {import('next').NextConfig}
   */
  const nextConfig = {
    /* config options here */
  }
  return nextConfig
}Since Next.js 12.1.0, you can use an async function:next.config.js module.exports = async (phase, { defaultConfig }) => {
  /**
   * @type {import('next').NextConfig}
   */
  const nextConfig = {
    /* config options here */
  }
  return nextConfig
}phase is the current context in which the configuration is loaded. You can see the available phases. Phases can be imported from next/constants: const { PHASE_DEVELOPMENT_SERVER } = require('next/constants')

module.exports = (phase, { defaultConfig }) => {
  if (phase === PHASE_DEVELOPMENT_SERVER) {
    return {
      /* development only config options here */
    }
  }

  return {
    /* config options for all phases except development here */
  }
}The commented lines are the place where you can put the configs allowed by next.config.js, which are defined in this file.However, none of the configs are required, and it's not necessary to understand what each config does. Instead, search for the features you need to enable or modify in this section and they will show you what to do.
```

Avoid using new JavaScript features not available in your target Node.js version. next.config.js will not be parsed by Webpack, Babel or TypeScript.

poweredByHeader
By default Next.js will add the x-powered-by header. To opt-out of it, open next.config.js and disable the poweredByHeader config:
next.config.js module.exports = {
  poweredByHeader: false,
}
productionBrowserSourceMaps
Source Maps are enabled by default during development. During production builds, they are disabled to prevent you leaking your source on the client, unless you specifically opt-in with the configuration flag.
Next.js provides a configuration flag you can use to enable browser source map generation during the production build:
next.config.js module.exports = {
  productionBrowserSourceMaps: true,
}
When the productionBrowserSourceMaps option is enabled, the source maps will be output in the same directory as the JavaScript files. Next.js will automatically serve these files when requested.

Adding source maps can increase next build time
Increases memory usage during next build
reactStrictMode

Good to know: Since Next.js 13.4, Strict Mode is true by default with app router, so the above configuration is only necessary for pages. You can still disable Strict Mode by setting reactStrictMode: false.

Suggested: We strongly suggest you enable Strict Mode in your Next.js application to better prepare your application for the future of React.

React's Strict Mode is a development mode only feature for highlighting potential problems in an application. It helps to identify unsafe lifecycles, legacy API usage, and a number of other features.
The Next.js runtime is Strict Mode-compliant. To opt-in to Strict Mode, configure the following option in your next.config.js:
next.config.js module.exports = {
  reactStrictMode: true,
}
If you or your team are not ready to use Strict Mode in your entire application, that's OK! You can incrementally migrate on a page-by-page basis using <React.StrictMode>.
redirects
Redirects allow you to redirect an incoming request path to a different destination path.

To use redirects you can use the redirects key in next.config.js:

```
next.config.js module.exports = {
  async redirects() {
    return [
      {
        source: '/about',
        destination: '/',
        permanent: true,
      },
    ]
  },
}
```

redirects is an async function that expects an array to be returned holding objects with source, destination, and permanent properties:

source is the incoming request path pattern.
destination is the path you want to route to.
permanent true or false - if true will use the 308 status code which instructs clients/search engines to cache the redirect forever, if false will use the 307 status code which is temporary and is not cached.

Why does Next.js use 307 and 308? Traditionally a 302 was used for a temporary redirect, and a 301 for a permanent redirect, but many browsers changed the request method of the redirect to GET, regardless of the original method. For example, if the browser made a request to POST /v1/users which returned status code 302 with location /v2/users, the subsequent request might be GET /v2/users instead of the expected POST /v2/users. Next.js uses the 307 temporary redirect, and 308 permanent redirect status codes to explicitly preserve the request method used.

basePath: false or undefined - if false the basePath won't be included when matching, can be used for external redirects only.
locale: false or undefined - whether the locale should not be included when matching.
has is an array of has objects with the type, key and value properties.
missing is an array of missing objects with the type, key and value properties.

Redirects are checked before the filesystem which includes pages and /public files.
Redirects are not applied to client-side routing (Link, router.push), unless Middleware is present and matches the path.
When a redirect is applied, any query values provided in the request will be passed through to the redirect destination. For example, see the following redirect configuration:

```
 {
  source: '/old-blog/:path*',
  destination: '/blog/:path*',
  permanent: false
```

}
When /old-blog/post-1?hello=world is requested, the client will be redirected to /blog/post-1?hello=world.
Path Matching

Path matches are allowed, for example /old-blog/:slug will match /old-blog/hello-world (no nested paths):
next.config.js module.exports = {
  async redirects() {
    return [
      {
        source: '/old-blog/:slug',
        destination: '/news/:slug', // Matched parameters can be used in the destination
        permanent: true,
      },
    ]
  },
}
Wildcard Path Matching

To match a wildcard path you can use * after a parameter, for example /blog/:slug* will match /blog/a/b/c/d/hello-world:
next.config.js module.exports = {
  async redirects() {
    return [
      {
        source: '/blog/:slug*',
        destination: '/news/:slug*', // Matched parameters can be used in the destination
        permanent: true,
      },
    ]
  },
}
Regex Path Matching

To match a regex path you can wrap the regex in parentheses after a parameter, for example /post/:slug(\\d{1,}) will match /post/123 but not /post/abc:

```
next.config.js module.exports = {
  async redirects() {
    return [
      {
        source: '/post/:slug(\\d{1,})',
        destination: '/news/:slug', // Matched parameters can be used in the destination
        permanent: false,
      },
    ]
  },
}
```

The following characters (,), {, }, :, *, +, ? are used for regex path matching, so when used in the source as non-special values they must be escaped by adding \\ before them:

```
next.config.js module.exports = {
  async redirects() {
    return [
      {
        // this will match `/english(default)/something` being requested
        source: '/english\\(default\\)/:slug',
        destination: '/en-us/:slug',
        permanent: false,
      },
    ]
  },
}
```

Header, Cookie, and Query Matching

To only match a redirect when header, cookie, or query values also match the has field or don't match the missing field can be used. Both the source and all has items must match and all missing items must not match for the redirect to be applied.
has and missing items can have the following fields:

type: String - must be either header, cookie, host, or query.
key: String - the key from the selected type to match against.
value: String or undefined - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value first-(?<paramName>.*) is used for first-second then second will be usable in the destination with :paramName.

```
next.config.js module.exports = {
  async redirects() {
    return [
      // if the header `x-redirect-me` is present,
      // this redirect will be applied
      {
        source: '/:path((?!another-page$).*)',
        has: [
          {
            type: 'header',
            key: 'x-redirect-me',
          },
        ],
        permanent: false,
        destination: '/another-page',
      },
      // if the header `x-dont-redirect` is present,
      // this redirect will NOT be applied
      {
        source: '/:path((?!another-page$).*)',
        missing: [
          {
            type: 'header',
            key: 'x-do-not-redirect',
          },
        ],
        permanent: false,
        destination: '/another-page',
      },
      // if the source, query, and cookie are matched,
      // this redirect will be applied
      {
        source: '/specific/:path*',
        has: [
          {
            type: 'query',
            key: 'page',
            // the page value will not be available in the
            // destination since value is provided and doesn't
            // use a named capture group e.g. (?<page>home)
            value: 'home',
          },
          {
            type: 'cookie',
            key: 'authorized',
```

```
            value: 'true',
          },
        ],
        permanent: false,
        destination: '/another/:path*',
      },
      // if the header `x-authorized` is present and
      // contains a matching value, this redirect will be applied
      {
        source: '/',
        has: [
          {
            type: 'header',
            key: 'x-authorized',
            value: '(?<authorized>yes|true)',
          },
        ],
        permanent: false,
        destination: '/home?authorized=:authorized',
      },
      // if the host is `example.com`,
      // this redirect will be applied
      {
        source: '/:path((?!another-page$).*)',
        has: [
          {
            type: 'host',
            value: 'example.com',
          },
        ],
        permanent: false,
        destination: '/another-page',
      },
    ]
  },
}
```

Redirects with basePath support

When leveraging basePath support with redirects each source and destination is automatically prefixed with the basePath unless you add basePath: false to the redirect:

```
next.config.js module.exports = {
  basePath: '/docs',
```

```
  async redirects() {
    return [
      {
        source: '/with-basePath', // automatically becomes /docs/with-basePath
        destination: '/another', // automatically becomes /docs/another
        permanent: false,
      },
      {
        // does not add /docs since basePath: false is set
        source: '/without-basePath',
        destination: 'https://example.com',
        basePath: false,
        permanent: false,
      },
    ]
  },
}
```

Redirects with i18n support

When leveraging i18n support with redirects each source and destination is automatically prefixed to handle the configured locales unless you add locale: false to the redirect. If locale: false is used you must prefix the source and destination with a locale for it to be matched correctly.

```
next.config.js module.exports = {
  i18n: {
    locales: ['en', 'fr', 'de'],
    defaultLocale: 'en',
  },

  async redirects() {
    return [
      {
        source: '/with-locale', // automatically handles all locales
        destination: '/another', // automatically passes the locale on
        permanent: false,
      },
      {
        // does not handle locales automatically since locale: false is set
        source: '/nl/with-locale-manual',
        destination: '/nl/another',
```

```
      locale: false,
      permanent: false,
    },
    {
      // this matches '/' since `en` is the defaultLocale
      source: '/en',
      destination: '/en/another',
      locale: false,
      permanent: false,
    },
    // it's possible to match all locales even when locale: false is set
    {
      source: '/:locale/page',
      destination: '/en/newpage',
      permanent: false,
      locale: false,
    }
    {
      // this gets converted to /(en|fr|de)/(.*) so will not match the top-level
      // `/` or `/fr` routes like /:path* would
      source: '/(.*)',
      destination: '/another',
      permanent: false,
    },
  ]
 },
}
```

In some rare cases, you might need to assign a custom status code for older HTTP Clients to properly redirect. In these cases, you can use the statusCode property instead of the permanent property, but not both. To to ensure IE11 compatibility, a Refresh header is automatically added for the 308 status code.

Other Redirects

Inside API Routes, you can use res.redirect().

Inside getStaticProps and getServerSideProps, you can redirect specific pages at request-time.

Version History

VersionChangesv13.3.0missing added.v10.2.0has added.v9.5.0redirects added.
rewrites
Rewrites allow you to map an incoming request path to a different destination path. Rewrites act as a URL proxy and mask the destination path, making it appear the user hasn't changed their location on the site. In contrast, redirects will reroute to a new page and show the URL changes.

To use rewrites you can use the rewrites key in next.config.js:

```
next.config.js module.exports = {
  async rewrites() {
    return [
      {
        source: '/about',
        destination: '/',
      },
    ]
  },
}
```

Rewrites are applied to client-side routing, a <Link href="/about"> will have the rewrite applied in the above example.
rewrites is an async function that expects to return either an array or an object of arrays (see below) holding objects with source and destination properties:

source: String - is the incoming request path pattern.
destination: String is the path you want to route to.
basePath: false or undefined - if false the basePath won't be included when matching, can be used for external rewrites only.
locale: false or undefined - whether the locale should not be included when matching.
has is an array of has objects with the type, key and value properties.
missing is an array of missing objects with the type, key and value properties.

When the rewrites function returns an array, rewrites are applied after checking the filesystem (pages and /public files) and before dynamic routes. When the rewrites function returns an object of arrays with a specific shape, this behavior can be changed and more finely controlled, as of v10.1 of Next.js:

```
next.config.js module.exports = {
  async rewrites() {
    return {
      beforeFiles: [
        // These rewrites are checked after headers/redirects
        // and before all files including _next/public files which
        // allows overriding page files
        {
          source: '/some-page',
```

```
          destination: '/somewhere-else',
          has: [{ type: 'query', key: 'overrideMe' }],
        },
      ],
      afterFiles: [
        // These rewrites are checked after pages/public files
        // are checked but before dynamic routes
        {
          source: '/non-existent',
          destination: '/somewhere-else',
        },
      ],
      fallback: [
        // These rewrites are checked after both pages/public files
        // and dynamic routes are checked
        {
          source: '/:path*',
          destination: `https://my-old-site.com/:path*`,
        },
      ],
    }
  },
}
```

Good to know: rewrites in beforeFiles do not check the filesystem/dynamic routes immediately after matching a source, they continue until all beforeFiles have been checked.

The order Next.js routes are checked is:

headers are checked/applied
redirects are checked/applied
beforeFiles rewrites are checked/applied
static files from the public directory, _next/static files, and non-dynamic pages are checked/served
afterFiles rewrites are checked/applied, if one of these rewrites is matched we check dynamic routes/static files after each match
fallback rewrites are checked/applied, these are applied before rendering the 404 page and after dynamic routes/all static assets have been checked. If you use fallback: true/'blocking' in getStaticPaths, the fallback rewrites defined in your next.config.js will not be run.

Rewrite parameters

When using parameters in a rewrite the parameters will be passed in the query by default when none of the parameters are used in the destination.

next.config.js
```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/old-about/:path*',
        destination: '/about', // The :path parameter isn't used here so will be automatically passed in the query
      },
    ]
  },
}
```

If a parameter is used in the destination none of the parameters will be automatically passed in the query.

next.config.js
```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/docs/:path*',
        destination: '/:path*', // The :path parameter is used here so will not be automatically passed in the query
      },
    ]
  },
}
```

You can still pass the parameters manually in the query if one is already used in the destination by specifying the query in the destination.

next.config.js
```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/:first/:second',
        destination: '/:first?second=:second',
        // Since the :first parameter is used in the destination the :second parameter
        // will not automatically be added in the query although we can manually add it
        // as shown above
      },
    ]
  },
}
```

Good to know: Static pages from Automatic Static Optimization or prerendering params

from rewrites will be parsed on the client after hydration and provided in the query.

Path Matching

Path matches are allowed, for example /blog/:slug will match /blog/hello-world (no nested paths):

```
next.config.js module.exports = {
  async rewrites() {
    return [
      {
        source: '/blog/:slug',
        destination: '/news/:slug', // Matched parameters can be used in the destination
      },
    ]
  },
}
```

Wildcard Path Matching

To match a wildcard path you can use * after a parameter, for example /blog/:slug* will match /blog/a/b/c/d/hello-world:

```
next.config.js module.exports = {
  async rewrites() {
    return [
      {
        source: '/blog/:slug*',
        destination: '/news/:slug*', // Matched parameters can be used in the destination
      },
    ]
  },
}
```

Regex Path Matching

To match a regex path you can wrap the regex in parenthesis after a parameter, for example /blog/:slug(\\d{1,}) will match /blog/123 but not /blog/abc:

```
next.config.js module.exports = {
  async rewrites() {
    return [
      {
        source: '/old-blog/:post(\\d{1,})',
        destination: '/blog/:post', // Matched parameters can be used in the destination
      },
    ]
  },
}
```

The following characters (,), {, }, :, *, +, ? are used for regex path matching, so when used in the source as non-special values they must be escaped by adding \\ before them:

```
next.config.js module.exports = {
  async rewrites() {
    return [
      {
        // this will match `/english(default)/something` being requested
        source: '/english\\(default\\)/:slug',
        destination: '/en-us/:slug',
      },
    ]
  },
}
```

Header, Cookie, and Query Matching

To only match a rewrite when header, cookie, or query values also match the has field or don't match the missing field can be used. Both the source and all has items must match and all missing items must not match for the rewrite to be applied.
has and missing items can have the following fields:

type: String - must be either header, cookie, host, or query.
key: String - the key from the selected type to match against.
value: String or undefined - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value first-(?<paramName>.*) is used for first-second then second will be usable in the destination with :paramName.

```
next.config.js module.exports = {
  async rewrites() {
    return [
      // if the header `x-rewrite-me` is present,
```

```
// this rewrite will be applied
{
  source: '/:path*',
  has: [
    {
      type: 'header',
      key: 'x-rewrite-me',
    },
  ],
  destination: '/another-page',
},
// if the header `x-rewrite-me` is not present,
// this rewrite will be applied
{
  source: '/:path*',
  missing: [
    {
      type: 'header',
      key: 'x-rewrite-me',
    },
  ],
  destination: '/another-page',
},
// if the source, query, and cookie are matched,
// this rewrite will be applied
{
  source: '/specific/:path*',
  has: [
    {
      type: 'query',
      key: 'page',
      // the page value will not be available in the
      // destination since value is provided and doesn't
      // use a named capture group e.g. (?<page>home)
      value: 'home',
    },
    {
      type: 'cookie',
      key: 'authorized',
      value: 'true',
    },
  ],
  destination: '/:path*/home',
},
// if the header `x-authorized` is present and
// contains a matching value, this rewrite will be applied
```

```
      {
        source: '/:path*',
        has: [
          {
            type: 'header',
            key: 'x-authorized',
            value: '(?<authorized>yes|true)',
          },
        ],
        destination: '/home?authorized=:authorized',
      },
      // if the host is `example.com`,
      // this rewrite will be applied
      {
        source: '/:path*',
        has: [
          {
            type: 'host',
            value: 'example.com',
          },
        ],
        destination: '/another-page',
      },
    ]
  },
}
```

Rewriting to an external URL

Examples
Incremental adoption of Next.js
Using Multiple Zones

Rewrites allow you to rewrite to an external url. This is especially useful for incrementally adopting Next.js. The following is an example rewrite for redirecting the /blog route of your main app to an external site.

next.config.js

```
module.exports = {
  async rewrites() {
    return [
      {
        source: '/blog',
        destination: 'https://example.com/blog',
      },
```

```
      {
        source: '/blog/:slug',
        destination: 'https://example.com/blog/:slug', // Matched parameters can be used in
the destination
      },
    ]
  },
}
```

If you're using trailingSlash: true, you also need to insert a trailing slash in the source parameter. If the destination server is also expecting a trailing slash it should be included in the destination parameter as well.

next.config.js

```
module.exports = {
  trailingSlash: true,
  async rewrites() {
    return [
      {
        source: '/blog/',
        destination: 'https://example.com/blog/',
      },
      {
        source: '/blog/:path*/',
        destination: 'https://example.com/blog/:path*/',
      },
    ]
  },
}
```

Incremental adoption of Next.js

You can also have Next.js fall back to proxying to an existing website after checking all Next.js routes.

This way you don't have to change the rewrites configuration when migrating more pages to Next.js

next.config.js

```
module.exports = {
  async rewrites() {
    return {
      fallback: [
        {
          source: '/:path*',
          destination: `https://custom-routes-proxying-endpoint.vercel.app/:path*`,
        },
      ],
    }
```

```
  },
}
```

Rewrites with basePath support

When leveraging basePath support with rewrites each source and destination is automatically prefixed with the basePath unless you add basePath: false to the rewrite:

```
next.config.js module.exports = {
  basePath: '/docs',

  async rewrites() {
    return [
      {
        source: '/with-basePath', // automatically becomes /docs/with-basePath
        destination: '/another', // automatically becomes /docs/another
      },
      {
        // does not add /docs to /without-basePath since basePath: false is set
        // Note: this can not be used for internal rewrites e.g. `destination: '/another'`
        source: '/without-basePath',
        destination: 'https://example.com',
        basePath: false,
      },
    ]
  },
}
```

Rewrites with i18n support

When leveraging i18n support with rewrites each source and destination is automatically prefixed to handle the configured locales unless you add locale: false to the rewrite. If locale: false is used you must prefix the source and destination with a locale for it to be matched correctly.

```
next.config.js module.exports = {
  i18n: {
    locales: ['en', 'fr', 'de'],
    defaultLocale: 'en',
  },
```

```
  async rewrites() {
    return [
      {
        source: '/with-locale', // automatically handles all locales
        destination: '/another', // automatically passes the locale on
      },
      {
        // does not handle locales automatically since locale: false is set
        source: '/nl/with-locale-manual',
        destination: '/nl/another',
        locale: false,
      },
      {
        // this matches '/' since `en` is the defaultLocale
        source: '/en',
        destination: '/en/another',
        locale: false,
      },
      {
        // it's possible to match all locales even when locale: false is set
        source: '/:locale/api-alias/:path*',
        destination: '/api/:path*',
        locale: false,
      },
      {
        // this gets converted to /(en|fr|de)/(.*) so will not match the top-level
        // `/` or `/fr` routes like /:path* would
        source: '/(.*)',
        destination: '/another',
      },
    ]
  },
}
```

Version History

VersionChangesv13.3.0missing added.v10.2.0has added.v9.5.0Headers added.

Runtime Config

Good to know: This feature is considered legacy and does not work with Automatic Static Optimization, Output File Tracing, or React Server Components. Please use environment variables instead to avoid initialization overhead.

To add runtime configuration to your app, open next.config.js and add the publicRuntimeConfig and serverRuntimeConfig configs:
next.config.js module.exports = {
  serverRuntimeConfig: {
    // Will only be available on the server side
    mySecret: 'secret',
    secondSecret: process.env.SECOND_SECRET, // Pass through env variables
  },
  publicRuntimeConfig: {
    // Will be available on both server and client
    staticFolder: '/static',
  },
}
Place any server-only runtime config under serverRuntimeConfig.
Anything accessible to both client and server-side code should be under publicRuntimeConfig.

A page that relies on publicRuntimeConfig must use getInitialProps or getServerSideProps or your application must have a Custom App with getInitialProps to opt-out of Automatic Static Optimization. Runtime configuration won't be available to any page (or component in a page) without being server-side rendered.

To get access to the runtime configs in your app use next/config, like so:
 import getConfig from 'next/config'
import Image from 'next/image'

// Only holds serverRuntimeConfig and publicRuntimeConfig
const { serverRuntimeConfig, publicRuntimeConfig } = getConfig()
// Will only be available on the server-side
console.log(serverRuntimeConfig.mySecret)
// Will be available on both server-side and client-side
console.log(publicRuntimeConfig.staticFolder)

function MyImage() {
  return (

      <Image
        src={`${publicRuntimeConfig.staticFolder}/logo.png`}
        alt="logo"
        layout="fill"
      />

)
}

export default MyImage

serverComponentsExternalPackagesDependencies used inside Server Components and Route Handlers will automatically be bundled by Next.js.
If a dependency is using Node.js specific features, you can choose to opt-out specific dependencies from the Server Components bundling and use native Node.js require.
next.config.js /** @type {import('next').NextConfig} */
```
const nextConfig = {
  experimental: {
    serverComponentsExternalPackages: ['@acme/ui'],
  },
}

module.exports = nextConfig
```
Next.js includes a short list of popular packages that currently are working on compatibility and automatically opt-ed out:

@aws-sdk/client-s3
@aws-sdk/s3-presigned-post
@blockfrost/blockfrost-js
@jpg-store/lucid-cardano
@mikro-orm/core
@mikro-orm/knex
@prisma/client
@sentry/nextjs
@sentry/node
@swc/core
argon2
autoprefixer
aws-crt
bcrypt
better-sqlite3
canvas
cpu-features
cypress
eslint
express
firebase-admin
jest
jsdom
lodash
mdx-bundler
mongodb
mongoose
next-mdx-remote
next-seo
payload
pg

playwright
postcss
prettier
prisma
puppeteer
rimraf
sharp
shiki
sqlite3
tailwindcss
ts-node
typescript
vscode-oniguruma
webpack
trailingSlash
By default Next.js will redirect urls with trailing slashes to their counterpart without a trailing slash. For example /about/ will redirect to /about. You can configure this behavior to act the opposite way, where urls without trailing slashes are redirected to their counterparts with trailing slashes.
Open next.config.js and add the trailingSlash config:
next.config.js module.exports = {
  trailingSlash: true,
}
With this option set, urls like /about will redirect to /about/.
Version History

VersionChangesv9.5.0trailingSlash added.
transpilePackages
Next.js can automatically transpile and bundle dependencies from local packages (like monorepos) or from external dependencies (node_modules). This replaces the next-transpile-modules package.
next.config.js /** @type {import('next').NextConfig} */
const nextConfig = {
  transpilePackages: ['@acme/ui', 'lodash-es'],
}

module.exports = nextConfig
Version History

VersionChangesv13.0.0transpilePackages added.
turbo (Experimental)

Warning: These features are experimental and will only work with next --turbo.

webpack loaders

Currently, Turbopack supports a subset of webpack's loader API, allowing you to use some webpack loaders to transform code in Turbopack.
To configure loaders, add the names of the loaders you've installed and any options in next.config.js, mapping file extensions to a list of loaders:
next.config.js module.exports = {

```
  experimental: {
    turbo: {
      loaders: {
        // Option format
        '.md': [
          {
            loader: '@mdx-js/loader',
            options: {
              format: 'md',
            },
          },
        ],
        // Option-less format
        '.mdx': ['@mdx-js/loader'],
      },
    },
  },
}
```

Then, given the above configuration, you can use transformed code from your app:

```
 import MyDoc from './my-doc.mdx'

export default function Home() {
  return <MyDoc />
}
```

Resolve Alias

Through next.config.js, Turbopack can be configured to modify module resolution through aliases, similar to webpack's resolve.alias configuration.
To configure resolve aliases, map imported patterns to their new destination in next.config.js:
next.config.js module.exports = {
  experimental: {
    turbo: {
      resolveAlias: {
        underscore: 'lodash',
        mocha: { browser: 'mocha/browser-entry.js' },
      },
    },
  },
}
This aliases imports of the underscore package to the lodash package. In other words, import underscore from 'underscore' will load the lodash module instead of underscore.
Turbopack also supports conditional aliasing through this field, similar to Node.js's conditional exports. At the moment only the browser condition is supported. In the case above, imports of the mocha module will be aliased to mocha/browser-entry.js when Turbopack targets browser environments.
For more information and guidance for how to migrate your app to Turbopack from webpack, see Turbopack's documentation on webpack compatibility.
typedRoutes (experimental)Experimental support for statically typed links. This feature requires using the App Router as well as TypeScript in your project.
next.config.js /** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    typedRoutes: true,
  },
}

module.exports = nextConfig
typescript
Next.js fails your production build (next build) when TypeScript errors are present in your project.
If you'd like Next.js to dangerously produce production code even when your application has errors, you can disable the built-in type checking step.
If disabled, be sure you are running type checks as part of your build or deploy process, otherwise this can be very dangerous.
Open next.config.js and enable the ignoreBuildErrors option in the typescript config:
next.config.js module.exports = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.

```
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```

urlImports

URL imports are an experimental feature that allows you to import modules directly from external servers (instead of from the local disk).

Warning: This feature is experimental. Only use domains that you trust to download and execute on your machine. Please exercise discretion, and caution until the feature is flagged as stable.

To opt-in, add the allowed URL prefixes inside next.config.js:

```
next.config.js module.exports = {
  experimental: {
    urlImports: ['https://example.com/assets/', 'https://cdn.skypack.dev'],
  },
}
```

Then, you can import modules directly from URLs:

```
 import { a, b, c } from 'https://example.com/assets/some/module.js'
```

URL Imports can be used everywhere normal package imports can be used.

Security Model

This feature is being designed with security as the top priority. To start, we added an experimental flag forcing you to explicitly allow the domains you accept URL imports from. We're working to take this further by limiting URL imports to execute in the browser sandbox using the Edge Runtime.

Lockfile

When using URL imports, Next.js will create a next.lock directory containing a lockfile and fetched assets.

This directory must be committed to Git, not ignored by .gitignore.

When running next dev, Next.js will download and add all newly discovered URL Imports to your lockfile

When running next build, Next.js will use only the lockfile to build the application for production

Typically, no network requests are needed and any outdated lockfile will cause the build to fail.
One exception is resources that respond with Cache-Control: no-cache.
These resources will have a no-cache entry in the lockfile and will always be fetched from the network on each build.
Examples

Skypack

```
 import confetti from 'https://cdn.skypack.dev/canvas-confetti'
import { useEffect } from 'react'

export default () => {
  useEffect(() => {
    confetti()
  })
  return <p>Hello</p>
}
```

Static Image Imports

```
 import Image from 'next/image'
import logo from 'https://example.com/assets/logo.png'

export default () => (
  <div>
    <Image src={logo} placeholder="blur" />
  </div>
)
```

URLs in CSS

```
 .className {
  background: url('https://example.com/assets/hero.jpg');
}
```

Asset Imports

```
 const logo = new URL('https://example.com/assets/file.txt', import.meta.url)

console.log(logo.pathname)

// prints "/_next/static/media/file.a9727b5d.txt"
```

Custom Webpack Config

Good to know: changes to webpack config are not covered by semver so proceed at your own risk

Before continuing to add custom webpack configuration to your application make sure Next.js doesn't already support your use-case:

CSS imports
CSS modules
Sass/SCSS imports
Sass/SCSS modules
preact

Some commonly asked for features are available as plugins:

@next/mdx
@next/bundle-analyzer

In order to extend our usage of webpack, you can define a function that extends its config inside next.config.js, like so:

```
next.config.js module.exports = {
  webpack: (
    config,
    { buildId, dev, isServer, defaultLoaders, nextRuntime, webpack }
  ) => {
    // Important: return the modified config
    return config
  },
}
```

The webpack function is executed twice, once for the server and once for the client. This allows you to distinguish between client and server configuration using the isServer property.

The second argument to the webpack function is an object with the following properties:

buildId: String - The build id, used as a unique identifier between builds
dev: Boolean - Indicates if the compilation will be done in development
isServer: Boolean - It's true for server-side compilation, and false for client-side compilation
nextRuntime: String | undefined - The target runtime for server-side compilation; either "edge" or "nodejs", it's undefined for client-side compilation.
defaultLoaders: Object - Default loaders used internally by Next.js:

babel: Object - Default babel-loader configuration

Example usage of defaultLoaders.babel:

```
 // Example config for adding a loader that depends on babel-loader
// This source was taken from the @next/mdx plugin source:
// https://github.com/vercel/next.js/tree/canary/packages/next-mdx
module.exports = {
  webpack: (config, options) => {
    config.module.rules.push({
      test: /\.mdx/,
      use: [
        options.defaultLoaders.babel,
        {
          loader: '@mdx-js/loader',
          options: pluginOptions.options,
        },
      ],
    })

    return config
  },
}
```

nextRuntime

Notice that isServer is true when nextRuntime is "edge" or "nodejs", nextRuntime "edge" is currently for middleware and Server Components in edge runtime only.

webVitalsAttribution
When debugging issues related to Web Vitals, it is often helpful if we can pinpoint the source of the problem.
For example, in the case of Cumulative Layout Shift (CLS), we might want to know the first element that shifted when the single largest layout shift occurred.
Or, in the case of Largest Contentful Paint (LCP), we might want to identify the element corresponding to the LCP for the page.
If the LCP element is an image, knowing the URL of the image resource can help us locate the asset we need to optimize.
Pinpointing the biggest contributor to the Web Vitals score, aka attribution,
allows us to obtain more in-depth information like entries for PerformanceEventTiming, PerformanceNavigationTiming and PerformanceResourceTiming.
Attribution is disabled by default in Next.js but can be enabled per metric by specifying the following in next.config.js.
next.config.js experimental: {
  webVitalsAttribution: ['CLS', 'LCP']
}
Valid attribution values are all web-vitals metrics specified in the NextWebVitalsMetric type.
create-next-app
The easiest way to get started with Next.js is by using create-next-app. This CLI tool enables you to quickly start building a new Next.js application, with everything set up for you.
You can create a new app using the default Next.js template, or by using one of the official Next.js examples.
Interactive

You can create a new project interactively by running:
Terminal npx create-next-app@latest
Terminal yarn create next-app
Terminal pnpm create next-app
You will then be asked the following prompts:
Terminal What is your project named? my-app
Would you like to use TypeScript? No / Yes
Would you like to use ESLint? No / Yes
Would you like to use Tailwind CSS? No / Yes
Would you like to use `src/` directory? No / Yes
Would you like to use App Router? (recommended) No / Yes
Would you like to customize the default import alias? No / Yes
Once you've answered the prompts, a new project will be created with the correct configuration depending on your answers.
Non-interactive

You can also pass command line arguments to set up a new project non-interactively.
Further, you can negate default options by prefixing them with --no- (e.g. --no-eslint).
See create-next-app --help:
Terminal Usage: create-next-app <project-directory> [options]

Options:
  -V, --version   output the version number
  --ts, --typescript

    Initialize as a TypeScript project. (default)

  --js, --javascript

    Initialize as a JavaScript project.

  --tailwind

    Initialize with Tailwind CSS config. (default)

  --eslint

    Initialize with ESLint config.

  --app

    Initialize as an App Router project.

  --src-dir

    Initialize inside a `src/` directory.

  --import-alias <alias-to-configure>

    Specify import alias to use (default "@/*").

  --use-npm

    Explicitly tell the CLI to bootstrap the app using npm

  --use-pnpm

Explicitly tell the CLI to bootstrap the app using pnpm

--use-yarn

Explicitly tell the CLI to bootstrap the app using Yarn

-e, --example [name]|[github-url]

An example to bootstrap the app with. You can use an example name
from the official Next.js repo or a public GitHub URL. The URL can use
any branch and/or subdirectory

--example-path <path-to-example>

In a rare case, your GitHub URL might contain a branch name with
a slash (e.g. bug/fix-1) and the path to the example (e.g. foo/bar).
In this case, you must specify the path to the example separately:
--example-path foo/bar

--reset-preferences

Explicitly tell the CLI to reset any stored preferences

-h, --help output usage information

Why use Create Next App?

create-next-app allows you to create a new Next.js app within seconds. It is officially maintained by the creators of Next.js, and includes a number of benefits:

Interactive Experience: Running npx create-next-app@latest (with no arguments) launches an interactive experience that guides you through setting up a project.
Zero Dependencies: Initializing a project is as quick as one second. Create Next App has zero dependencies.
Offline Support: Create Next App will automatically detect if you're offline and bootstrap your project using your local package cache.
Support for Examples: Create Next App can bootstrap your application using an example from the Next.js examples collection (e.g. npx create-next-app --example api-routes) or any public GitHub repository.
Tested: The package is part of the Next.js monorepo and tested using the same integration test suite as Next.js itself, ensuring it works as expected with every release.
Edge Runtime
The Next.js Edge Runtime is based on standard Web APIs, it supports the following APIs:

Network APIs

APIDescriptionBlobRepresents a blobfetchFetches a resourceFetchEventRepresents a fetch eventFileRepresents a fileFormDataRepresents form dataHeadersRepresents HTTP headersRequestRepresents an HTTP requestResponseRepresents an HTTP responseURLSearchParamsRepresents URL search parametersWebSocketRepresents a websocket connection
Encoding APIs

APIDescriptionatobDecodes a base-64 encoded stringbtoaEncodes a string in base-64TextDecoderDecodes a Uint8Array into a stringTextDecoderStreamChainable decoder for streamsTextEncoderEncodes a string into a Uint8ArrayTextEncoderStreamChainable encoder for streams
Stream APIs

APIDescriptionReadableStreamRepresents a readable streamReadableStreamBYOBReaderRepresents a reader of a ReadableStreamReadableStreamDefaultReaderRepresents a reader of a ReadableStreamTransformStreamRepresents a transform streamWritableStreamRepresents a writable streamWritableStreamDefaultWriterRepresents a writer of a WritableStream
Crypto APIs

APIDescriptioncryptoProvides access to the cryptographic functionality of the platformCryptoKeyRepresents a cryptographic keySubtleCryptoProvides access to common cryptographic primitives, like hashing, signing, encryption or decryption
Web Standard APIs

APIDescriptionAbortControllerAllows you to abort one or more DOM requests as and when desiredArrayRepresents an array of valuesArrayBufferRepresents a generic, fixed-length raw binary data bufferAtomicsProvides atomic operations as static methodsBigIntRepresents a whole number with arbitrary precisionBigInt64ArrayRepresents a typed array of 64-bit signed integersBigUint64ArrayRepresents a typed array of 64-bit unsigned integersBooleanRepresents a logical entity and can have two values: true and falseclearIntervalCancels a timed, repeating action which was previously established by a call to setInterval()clearTimeoutCancels a timed, repeating action which was previously established by a call to setTimeout()consoleProvides access to the browser's debugging consoleDataViewRepresents a generic view of an ArrayBufferDateRepresents a single moment in time in a platform-independent formatdecodeURIDecodes a Uniform Resource Identifier (URI) previously created by encodeURI or by a similar routinedecodeURIComponentDecodes a Uniform Resource Identifier (URI) component previously created by encodeURIComponent or by a similar routineDOMExceptionRepresents an error that occurs in the DOMencodeURIEncodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the characterencodeURIComponentEncodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the characterErrorRepresents an error when trying to execute a statement or accessing a propertyEvalErrorRepresents an error that occurs regarding the global function eval()Float32ArrayRepresents a typed array of 32-bit floating point numbersFloat64ArrayRepresents a typed array of 64-bit floating point numbersFunctionRepresents a functionInfinityRepresents the mathematical Infinity valueInt8ArrayRepresents a typed array of 8-bit signed integersInt16ArrayRepresents a typed array of 16-bit signed integersInt32ArrayRepresents a typed array of 32-bit signed integersIntlProvides access to internationalization and localization functionalityisFiniteDetermines whether a value is a finite numberisNaNDetermines whether a value is NaN or notJSONProvides functionality to convert JavaScript values to and from the JSON formatMapRepresents a collection of values, where each value may occur only onceMathProvides access to mathematical functions and constantsNumberRepresents a numeric valueObjectRepresents the object that is the base of all JavaScript objectsparseFloatParses a string argument and returns a floating point numberparseIntParses a string argument and returns an integer of the specified radixPromiseRepresents the eventual completion (or failure) of an asynchronous operation, and its resulting valueProxyRepresents an object that is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc)queueMicrotaskQueues a microtask to be executedRangeErrorRepresents an error when a value is not in the set or range of allowed valuesReferenceErrorRepresents an error when a non-existent variable is referencedReflectProvides methods for interceptable JavaScript operationsRegExpRepresents a regular expression, allowing you to match

combinations of charactersSetRepresents a collection of values, where each value may occur only oncesetIntervalRepeatedly calls a function, with a fixed time delay between each callsetTimeoutCalls a function or evaluates an expression after a specified number of millisecondsSharedArrayBufferRepresents a generic, fixed-length raw binary data bufferStringRepresents a sequence of charactersstructuredCloneCreates a deep copy of a valueSymbolRepresents a unique and immutable data type that is used as the key of an object propertySyntaxErrorRepresents an error when trying to interpret syntactically invalid codeTypeErrorRepresents an error when a value is not of the expected typeUint8ArrayRepresents a typed array of 8-bit unsigned integersUint8ClampedArrayRepresents a typed array of 8-bit unsigned integers clamped to 0-255Uint32ArrayRepresents a typed array of 32-bit unsigned integersURIErrorRepresents an error when a global URI handling function was used in a wrong wayURLRepresents an object providing static methods used for creating object URLsURLPatternRepresents a URL patternURLSearchParamsRepresents a collection of key/value pairsWeakMapRepresents a collection of key/value pairs in which the keys are weakly referencedWeakSetRepresents a collection of objects in which each object may occur only onceWebAssemblyProvides access to WebAssembly
Next.js Specific Polyfills

AsyncLocalStorage

Environment Variables

You can use process.env to access Environment Variables for both next dev and next build.
Unsupported APIs

The Edge Runtime has some restrictions including:

Native Node.js APIs are not supported. For example, you can't read or write to the filesystem.
node_modules can be used, as long as they implement ES Modules and do not use native Node.js APIs.

Calling require directly is not allowed. Use ES Modules instead.

The following JavaScript language features are disabled, and will not work:
APIDescriptionevalEvaluates JavaScript code represented as a stringnew Function(evalString)Creates a new function with the code provided as an argumentWebAssembly.compileCompiles a WebAssembly module from a buffer sourceWebAssembly.instantiateCompiles and instantiates a WebAssembly module from a buffer source
In rare cases, your code could contain (or import) some dynamic code evaluation statements which can not be reached at runtime and which can not be removed by treeshaking.
You can relax the check to allow specific files with your Middleware or Edge API Route exported configuration:

```
 export const config = {
  runtime: 'edge', // for Edge API Routes only
  unstable_allowDynamic: [
    // allows a single file
    '/lib/utilities.js',
    // use a glob to allow anything in the function-bind 3rd party module
    '/node_modules/function-bind/**',
  ],
}
```

unstable_allowDynamic is a glob, or an array of globs, ignoring dynamic code evaluation for specific files. The globs are relative to your application root folder.
Be warned that if these statements are executed on the Edge, they will throw and cause a runtime error.
Next.js CLI
The Next.js CLI allows you to start, build, and export your application.
To get a list of the available CLI commands, run the following command inside your project directory:
Terminal npx next -h
(npx comes with npm 5.2+ and higher)
The output should look like this:
Terminal Usage

```
  $ next <command>

Available commands
  build, start, export, dev, lint, telemetry, info

Options
  --version, -v   Version number
  --help, -h      Displays this message

For more information run a command with the --help flag
  $ next build --help
```

You can pass any node arguments to next commands:

Terminal NODE_OPTIONS='--throw-deprecation' next
NODE_OPTIONS='-r esm' next
NODE_OPTIONS='--inspect' next

Good to know: Running next without a command is the same as running next dev

Build

next build creates an optimized production build of your application. The output displays information about each route.

Size – The number of assets downloaded when navigating to the page client-side. The size for each route only includes its dependencies.
First Load JS – The number of assets downloaded when visiting the page from the server. The amount of JS shared by all is shown as a separate metric.

Both of these values are compressed with gzip. The first load is indicated by green, yellow, or red. Aim for green for performant applications.
You can enable production profiling for React with the --profile flag in next build. This requires Next.js 9.5:
Terminal next build --profile
After that, you can use the profiler in the same way as you would in development.
You can enable more verbose build output with the --debug flag in next build. This requires Next.js 9.5.3:
Terminal next build --debug
With this flag enabled additional build output like rewrites, redirects, and headers will be shown.
Development

next dev starts the application in development mode with hot-code reloading, error reporting, and more:
The application will start at http://localhost:3000 by default. The default port can be changed with -p, like so:
Terminal npx next dev -p 4000
Or using the PORT environment variable:
Terminal PORT=4000 npx next dev

Good to know: PORT cannot be set in .env as booting up the HTTP server happens

before any other code is initialized.

You can also set the hostname to be different from the default of 0.0.0.0, this can be useful for making the application available for other devices on the network. The default hostname can be changed with -H, like so:
Terminal npx next dev -H 192.168.1.2
Production

next start starts the application in production mode. The application should be compiled with next build first.
The application will start at http://localhost:3000 by default. The default port can be changed with -p, like so:
Terminal npx next start -p 4000
Or using the PORT environment variable:
Terminal PORT=4000 npx next start

Good to know:
-PORT cannot be set in .env as booting up the HTTP server happens before any other code is initialized.

next start cannot be used with output: 'standalone' or output: 'export'.

Keep Alive Timeout

When deploying Next.js behind a downstream proxy (e.g. a load-balancer like AWS ELB/ALB) it's important to configure Next's underlying HTTP server with keep-alive timeouts that are larger than the downstream proxy's timeouts. Otherwise, once a keep-alive timeout is reached for a given TCP connection, Node.js will immediately terminate that connection without notifying the downstream proxy. This results in a proxy error whenever it attempts to reuse a connection that Node.js has already terminated.
To configure the timeout values for the production Next.js server, pass --keepAliveTimeout (in milliseconds) to next start, like so:
Terminal npx next start --keepAliveTimeout 70000
Lint

next lint runs ESLint for all files in the pages/, app/, components/, lib/, and src/ directories. It also
provides a guided setup to install any required dependencies if ESLint is not already configured in
your application.
If you have other directories that you would like to lint, you can specify them using the --dir
flag:
Terminal next lint --dir utils
Telemetry

Next.js collects completely anonymous telemetry data about general usage.
Participation in this anonymous program is optional, and you may opt-out if you'd not like to share any information.
To learn more about Telemetry, please read this document.
Next Info

next info prints relevant details about the current system which can be used to report Next.js bugs.
This information includes Operating System platform/arch/version, Binaries (Node.js, npm, Yarn, pnpm) and npm package versions (next, react, react-dom).
Running the following in your project's root directory:
Terminal next info
will give you information like this example:
Terminal

```
  Operating System:
    Platform: linux
    Arch: x64
    Version: #22-Ubuntu SMP Fri Nov 5 13:21:36 UTC 2021
  Binaries:
    Node: 16.13.0
    npm: 8.1.0
    Yarn: 1.22.17
    pnpm: 6.24.2
  Relevant packages:
    next: 12.0.8
```

react: 17.0.2
react-dom: 17.0.2

This information should then be pasted into GitHub Issues.
ArchitectureLearn about the Next.js architecture and how it works under the hood.
AccessibilityThe Next.js team is committed to making Next.js accessible to all developers (and their end-users). By adding accessibility features to Next.js by default, we aim to make the Web more inclusive for everyone.
Route Announcements

When transitioning between pages rendered on the server (e.g. using the <a href> tag) screen readers and other assistive technology announce the page title when the page loads so that users understand that the page has changed.
In addition to traditional page navigations, Next.js also supports client-side transitions for improved performance (using next/link). To ensure that client-side transitions are also announced to assistive technology, Next.js includes a route announcer by default. The Next.js route announcer looks for the page name to announce by first inspecting document.title, then the <h1> element, and finally the URL pathname. For the most accessible user experience, ensure that each page in your application has a unique and descriptive title.
Linting

Next.js provides an integrated ESLint experience out of the box, including custom rules for Next.js. By default, Next.js includes eslint-plugin-jsx-a11y to help catch accessibility issues early, including warning on:

aria-props
aria-proptypes
aria-unsupported-elements
role-has-required-aria-props
role-supports-aria-props

For example, this plugin helps ensure you add alt text to img tags, use correct aria-* attributes, use correct role attributes, and more.
Accessibility Resources

WebAIM WCAG checklist
WCAG 2.1 Guidelines
The A11y Project
Check color contrast ratios between foreground and background elements
Use prefers-reduced-motion when working with animations
Fast RefreshExamples
Fast Refresh Demo

Fast Refresh is a Next.js feature that gives you instantaneous feedback on edits made to your React components. Fast Refresh is enabled by default in all Next.js applications on 9.4 or newer. With Next.js Fast Refresh enabled, most edits should be visible within a second, without losing component state.
How It Works

If you edit a file that only exports React component(s), Fast Refresh will update the code only for that file, and re-render your component. You can edit anything in that file, including styles, rendering logic, event handlers, or effects.
If you edit a file with exports that aren't React components, Fast Refresh will re-run both that file, and the other files importing it. So if both Button.js and Modal.js import theme.js, editing theme.js will update both components.
Finally, if you edit a file that's imported by files outside of the React tree, Fast Refresh will fall back to doing a full reload. You might have a file which renders a React component but also exports a value that is imported by a non-React component. For example, maybe your component also exports a constant, and a non-React utility file imports it. In that case, consider migrating the constant to a separate file and importing it into both files. This will re-enable Fast Refresh to work. Other cases can usually be solved in a similar way.

Error Resilience

Syntax Errors

If you make a syntax error during development, you can fix it and save the file again. The error will disappear automatically, so you won't need to reload the app. You will not lose component state.

Runtime Errors

If you make a mistake that leads to a runtime error inside your component, you'll be greeted with a contextual overlay. Fixing the error will automatically dismiss the overlay, without reloading the app.

Component state will be retained if the error did not occur during rendering. If the error did occur during rendering, React will remount your application using the updated code.

If you have error boundaries in your app (which is a good idea for graceful failures in production), they will retry rendering on the next edit after a rendering error. This means having an error boundary can prevent you from always getting reset to the root app state. However, keep in mind that error boundaries shouldn't be too granular. They are used by React in production, and should always be designed intentionally.

Limitations

Fast Refresh tries to preserve local React state in the component you're editing, but only if it's safe to do so. Here's a few reasons why you might see local state being reset on every edit to a file:

Local state is not preserved for class components (only function components and Hooks preserve state).

The file you're editing might have other exports in addition to a React component.

Sometimes, a file would export the result of calling a higher-order component like HOC(WrappedComponent). If the returned component is a class, its state will be reset.

Anonymous arrow functions like export default () => ; cause Fast Refresh to not

preserve local component state. For large codebases you can use our name-default-component codemod.

As more of your codebase moves to function components and Hooks, you can expect state to be preserved in more cases.

Tips

Fast Refresh preserves React local state in function components (and Hooks) by default.

Sometimes you might want to force the state to be reset, and a component to be remounted. For example, this can be handy if you're tweaking an animation that only happens on mount. To do this, you can add // @refresh reset anywhere in the file you're editing. This directive is local to the file, and instructs Fast Refresh to remount components defined in that file on every edit.

You can put console.log or debugger; into the components you edit during development.

Fast Refresh and Hooks

When possible, Fast Refresh attempts to preserve the state of your component between edits. In particular, useState and useRef preserve their previous values as long as you don't change their arguments or the order of the Hook calls.

Hooks with dependencies—such as useEffect, useMemo, and useCallback—will always update during Fast Refresh. Their list of dependencies will be ignored while Fast Refresh is happening.

For example, when you edit useMemo(() => x * 2, [x]) to useMemo(() => x * 10, [x]), it will re-run even though x (the dependency) has not changed. If React didn't do that, your edit wouldn't reflect on the screen!

Sometimes, this can lead to unexpected results. For example, even a useEffect with an empty array of dependencies would still re-run once during Fast Refresh. However, writing code resilient to occasional re-running of useEffect is a good practice even
without Fast Refresh. It will make it easier for you to introduce new dependencies to it later on

and it's enforced by React Strict Mode,
which we highly recommend enabling.
Next.js CompilerThe Next.js Compiler, written in Rust using SWC, allows Next.js to transform and minify your JavaScript code for production. This replaces Babel for individual files and Terser for minifying output bundles.
Compilation using the Next.js Compiler is 17x faster than Babel and enabled by default since Next.js version 12. If you have an existing Babel configuration or are using unsupported features, your application will opt-out of the Next.js Compiler and continue using Babel.
Why SWC?

SWC is an extensible Rust-based platform for the next generation of fast developer tools.
SWC can be used for compilation, minification, bundling, and more – and is designed to be extended. It's something you can call to perform code transformations (either built-in or custom). Running those transformations happens through higher-level tools like Next.js.
We chose to build on SWC for a few reasons:

Extensibility: SWC can be used as a Crate inside Next.js, without having to fork the library or workaround design constraints.
Performance: We were able to achieve ~3x faster Fast Refresh and ~5x faster builds in Next.js by switching to SWC, with more room for optimization still in progress.
WebAssembly: Rust's support for WASM is essential for supporting all possible platforms and taking Next.js development everywhere.
Community: The Rust community and ecosystem are amazing and still growing.

Supported Features

Styled Components

We're working to port babel-plugin-styled-components to the Next.js Compiler.
First, update to the latest version of Next.js: npm install next@latest. Then, update your next.config.js file:

```
next.config.js module.exports = {
  compiler: {
    // see https://styled-components.com/docs/tooling#babel-plugin for more info on the options.
    styledComponents: boolean | {
      // Enabled by default in development, disabled in production to reduce file size,
      // setting this will override the default for all environments.
      displayName?: boolean,
      // Enabled by default.
      ssr?: boolean,
      // Enabled by default.
      fileName?: boolean,
      // Empty by default.
      topLevelImportPaths?: string[],
      // Defaults to ["index"].
      meaninglessFileNames?: string[],
      // Enabled by default.
      cssProp?: boolean,
      // Empty by default.
      namespace?: string,
      // Not supported yet.
      minify?: boolean,
      // Not supported yet.
      transpileTemplateLiterals?: boolean,
      // Not supported yet.
      pure?: boolean,
    },
  },
}
```

minify, transpileTemplateLiterals and pure are not yet implemented. You can follow the progress here. ssr and displayName transforms are the main requirement for using styled-components in Next.js.

Jest

The Next.js Compiler transpiles your tests and simplifies configuring Jest together with Next.js including:

Auto mocking of .css, .module.css (and their .scss variants), and image imports
Automatically sets up transform using SWC
Loading .env (and all variants) into process.env
Ignores node_modules from test resolving and transforms
Ignoring .next from test resolving

Loads next.config.js for flags that enable experimental SWC transforms

First, update to the latest version of Next.js: npm install next@latest. Then, update your jest.config.js file:
jest.config.js const nextJest = require('next/jest')

// Providing the path to your Next.js app which will enable loading next.config.js and .env files
const createJestConfig = nextJest({ dir: './' })

// Any custom config you want to pass to Jest
const customJestConfig = {
  setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],
}

// createJestConfig is exported in this way to ensure that next/jest can load the Next.js configuration, which is async
module.exports = createJestConfig(customJestConfig)
Relay

To enable Relay support:
next.config.js module.exports = {
  compiler: {
    relay: {
      // This should match relay.config.js
      src: './',
      artifactDirectory: './__generated__',
      language: 'typescript',
      eagerEsModules: false,
    },
  },
}

Good to know: In Next.js, all JavaScript files in pages directory are considered routes. So, for relay-compiler you'll need to specify artifactDirectory configuration settings outside of the pages, otherwise relay-compiler will generate files next to the source file in the __generated__ directory, and this file will be considered a route, which will break production builds.

Remove React Properties

Allows to remove JSX properties. This is often used for testing. Similar to babel-plugin-react-remove-properties.
To remove properties matching the default regex ^data-test:

```
next.config.js module.exports = {
  compiler: {
    reactRemoveProperties: true,
  },
}
```

To remove custom properties:

```
next.config.js module.exports = {
  compiler: {
    // The regexes defined here are processed in Rust so the syntax is different from
    // JavaScript `RegExp`s. See https://docs.rs/regex.
    reactRemoveProperties: { properties: ['^data-custom$'] },
  },
}
```

Remove Console

This transform allows for removing all console.* calls in application code (not node_modules). Similar to babel-plugin-transform-remove-console.
Remove all console.* calls:

```
next.config.js module.exports = {
  compiler: {
    removeConsole: true,
  },
}
```

Remove console.* output except console.error:

```
next.config.js module.exports = {
  compiler: {
    removeConsole: {
      exclude: ['error'],
    },
  },
}
```

Legacy Decorators

Next.js will automatically detect experimentalDecorators in jsconfig.json or tsconfig.json. Legacy decorators are commonly used with older versions of libraries like mobx.
This flag is only supported for compatibility with existing applications. We do not recommend using legacy decorators in new applications.
First, update to the latest version of Next.js: npm install next@latest. Then, update your jsconfig.json or tsconfig.json file:

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

importSource

Next.js will automatically detect jsxImportSource in jsconfig.json or tsconfig.json and apply that. This is commonly used with libraries like Theme UI.
First, update to the latest version of Next.js: npm install next@latest. Then, update your jsconfig.json or tsconfig.json file:

```
{
  "compilerOptions": {
    "jsxImportSource": "theme-ui"
  }
}
```

Emotion

We're working to port @emotion/babel-plugin to the Next.js Compiler.
First, update to the latest version of Next.js: npm install next@latest. Then, update your next.config.js file:
next.config.js

```
module.exports = {
  compiler: {
    emotion: boolean | {
      // default is true. It will be disabled when build type is production.
      sourceMap?: boolean,
      // default is 'dev-only'.
      autoLabel?: 'never' | 'dev-only' | 'always',
      // default is '[local]'.
```

```
      // Allowed values: `[local]` `[filename]` and `[dirname]`
      // This option only works when autoLabel is set to 'dev-only' or 'always'.
      // It allows you to define the format of the resulting label.
      // The format is defined via string where variable parts are enclosed in square
brackets [].
      // For example labelFormat: "my-classname--[local]", where [local] will be replaced
with the name of the variable the result is assigned to.
      labelFormat?: string,
      // default is undefined.
      // This option allows you to tell the compiler what imports it should
      // look at to determine what it should transform so if you re-export
      // Emotion's exports, you can still use transforms.
      importMap?: {
        [packageName: string]: {
          [exportName: string]: {
            canonicalImport?: [string, string],
            styledBaseImport?: [string, string],
          }
        }
      },
    },
  },
}
```

Minification

Next.js' swc compiler is used for minification by default since v13. This is 7x faster than Terser.

If Terser is still needed for any reason this can be configured.

```
next.config.js module.exports = {
  swcMinify: false,
}
```

Module Transpilation

Next.js can automatically transpile and bundle dependencies from local packages (like monorepos) or from external dependencies (node_modules). This replaces the next-transpile-modules package.

```
next.config.js module.exports = {
  transpilePackages: ['@acme/ui', 'lodash-es'],
```

}
Modularize Imports

Examples
modularize-imports

Allows to modularize imports, similar to babel-plugin-transform-imports.
Transforms member style imports of packages that use a “barrel file” (a single file that re-exports other modules):
```
 import { Row, Grid as MyGrid } from 'react-bootstrap'
import { merge } from 'lodash'
```
...into default style imports of each module. This prevents compilation of unused modules:
```
 import Row from 'react-bootstrap/Row'
import MyGrid from 'react-bootstrap/Grid'
import merge from 'lodash/merge'
```
Config for the above transform:
```
next.config.js module.exports = {
  modularizeImports: {
    'react-bootstrap': {
      transform: 'react-bootstrap/{{member}}',
    },
    lodash: {
      transform: 'lodash/{{member}}',
    },
  },
}
```
Handlebars variables and helper functions

This transform uses handlebars to template the replacement import path in the transform field. These variables and helper functions are available:

member: Has type string. The name of the member import.
lowerCase, upperCase, camelCase, kebabCase: Helper functions to convert a string to lower, upper, camel or kebab cases.
matches: Has type string[]. All groups matched by the regular expression. matches.[0] is the full match.

For example, you can use the kebabCase helper like this:
next.config.js module.exports = {
  modularizeImports: {
    'my-library': {
      transform: 'my-library/{{ kebabCase member }}',
    },
  },
}
The above config will transform your code as follows:
 // Before
import { MyModule } from 'my-library'

// After (`MyModule` was converted to `my-module`)
import MyModule from 'my-library/my-module'
You can also use regular expressions using Rust regex crate’s syntax:
next.config.js module.exports = {
  modularizeImports: {
    'my-library/?(((\\w*)?/?)*)': {
      transform: 'my-library/{{ matches.[1] }}/{{member}}',
    },
  },
}
The above config will transform your code as follows:
 // Before
import { MyModule } from 'my-library'
import { App } from 'my-library/components'
import { Header, Footer } from 'my-library/components/app'

// After
import MyModule from 'my-library/my-module'
import App from 'my-library/components/app'
import Header from 'my-library/components/app/header'
import Footer from 'my-library/components/app/footer'
Using named imports

By default, modularizeImports assumes that each module uses default exports. However, this may not always be the case — named exports may be used.
my-library/MyModule.ts // Using named export instead of default export
export const MyModule = {}

// my-library/index.ts
// The “barrel file” that re-exports `MyModule`

```
export { MyModule } from './MyModule'
```

In this case, you can use the skipDefaultConversion option to use named imports instead of default imports:

```
next.config.js module.exports = {
  modularizeImports: {
    'my-library': {
      transform: 'my-library/{{member}}',
      skipDefaultConversion: true,
    },
  },
}
```

The above config will transform your code as follows:

```
 // Before
import { MyModule } from 'my-library'

// After (imports `MyModule` using named import)
import { MyModule } from 'my-library/MyModule'
```

Preventing full import

If you use the preventFullImport option, the compiler will throw an error if you import a “barrel file” using default import. If you use the following config:

```
next.config.js module.exports = {
  modularizeImports: {
    lodash: {
      transform: 'lodash/{{member}}',
      preventFullImport: true,
    },
  },
}
```

The compiler will throw an error if you try to import the full lodash library (instead of using named imports):

```
 // Compiler error
import lodash from 'lodash'
```

Experimental Features

SWC Trace profiling

You can generate SWC's internal transform traces as chromium's trace event format.

```
next.config.js module.exports = {
  experimental: {
    swcTraceProfiling: true,
  },
}
```

Once enabled, swc will generate trace named as swc-trace-profile-${timestamp}.json under .next/. Chromium's trace viewer (chrome://tracing/, https://ui.perfetto.dev/), or compatible flamegraph viewer (https://www.speedscope.app/) can load & visualize generated traces.

SWC Plugins (Experimental)

You can configure swc's transform to use SWC's experimental plugin support written in wasm to customize transformation behavior.

```
next.config.js module.exports = {
  experimental: {
    swcPlugins: [
      [
        'plugin',
        {
          ...pluginOptions,
        },
      ],
    ],
  },
}
```

swcPlugins accepts an array of tuples for configuring plugins. A tuple for the plugin contains the path to the plugin and an object for plugin configuration. The path to the plugin can be an npm module package name or an absolute path to the .wasm binary itself.

Unsupported Features

When your application has a .babelrc file, Next.js will automatically fall back to using Babel for transforming individual files. This ensures backwards compatibility with existing applications that leverage custom Babel plugins.

If you're using a custom Babel setup, please share your configuration. We're working to port as many commonly used Babel transformations as possible, as well as supporting plugins in the future.
Version History

VersionChangesv13.1.0Module Transpilation and Modularize Imports stable.v13.0.0SWC Minifier enabled by default.v12.3.0SWC Minifier stable.v12.2.0SWC Plugins experimental support added.v12.1.0Added support for Styled Components, Jest, Relay, Remove React Properties, Legacy Decorators, Remove Console, and jsxImportSource.v12.0.0Next.js Compiler introduced.
Supported BrowsersNext.js supports modern browsers with zero configuration.

Chrome 64+
Edge 79+
Firefox 67+
Opera 51+
Safari 12+

Browserslist

If you would like to target specific browsers or features, Next.js supports Browserslist configuration in your package.json file. Next.js uses the following Browserslist configuration by default:
package.json {
```
  "browserslist": [
    "chrome 64",
    "edge 79",
    "firefox 67",
    "opera 51",
    "safari 12"
  ]
}
```
Polyfills

We inject widely used polyfills, including:

fetch() — Replacing: whatwg-fetch and unfetch.
URL — Replacing: the url package (Node.js API).
Object.assign() — Replacing: object-assign, object.assign, and core-js/object/assign.

If any of your dependencies includes these polyfills, they'll be eliminated automatically from the production build to avoid duplication.
In addition, to reduce bundle size, Next.js will only load these polyfills for browsers that require them. The majority of the web traffic globally will not download these polyfills.
Custom Polyfills

If your own code or any external npm dependencies require features not supported by your target browsers (such as IE 11), you need to add polyfills yourself.
In this case, you should add a top-level import for the specific polyfill you need in your Custom <App> or the individual component.
JavaScript Language Features

Next.js allows you to use the latest JavaScript features out of the box. In addition to ES6 features, Next.js also supports:

Async/await (ES2017)
Object Rest/Spread Properties (ES2018)
Dynamic import() (ES2020)
Optional Chaining (ES2020)
Nullish Coalescing (ES2020)
Class Fields and Static Properties (part of stage 3 proposal)
and more!

Server-Side Polyfills

In addition to fetch() on the client-side, Next.js polyfills fetch() in the Node.js environment. You can use fetch() in your server code (such as getStaticProps/ getServerSideProps) without using polyfills such as isomorphic-unfetch or node-fetch.

TypeScript Features

Next.js has built-in TypeScript support. Learn more here.
Customizing Babel Config (Advanced)

You can customize babel configuration. Learn more here.
TurbopackTurbopack (beta) is an incremental bundler optimized for JavaScript and TypeScript, written in Rust, and built into Next.js.
Usage

Turbopack can be used in Next.js in both the pages and app directories for faster local development. To enable Turbopack, use the --turbo flag when running the Next.js development server.
package.json {
```
  "scripts": {
    "dev": "next dev --turbo",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  }
}
```
Supported Features

To learn more about the currently supported features for Turbopack, view the documentation.
Unsupported Features

Turbopack currently only supports next dev and does not support next build. We are currently working on support for builds as we move closer towards stability.
Next.js CommunityWith over 4 million weekly downloads, Next.js has a large and active community of developers across the world. Here's how you can get involved in our community:
Contributing

There are a couple of ways you can contribute to the development of Next.js:

Documentation: Suggest improvements or even write new sections to help our users understand how to use Next.js.
Examples: Help developers integrate Next.js with other tools and services by creating a new example or improving an existing one.
Codebase: Learn more about the underlying architecture, contribute to bug fixes, errors, and suggest new features.

Discussions

If you have a question about Next.js, or want to help others, you're always welcome to join the conversation:

GitHub Discussions
Discord
Reddit

Social Media

Follow Next.js on Twitter for the latest updates, and subscribe to the Vercel YouTube channel for Next.js videos.
Code of Conduct

We believe in creating an inclusive, welcoming community. As such, we ask all members to adhere to our Code of Conduct. This document outlines our expectations for participant behavior. We invite you to read it and help us maintain a safe and respectful environment.