description: Learn about the built-in accessibility features of Next.js.

Accessibility

The Next.js team is committed to making Next.js accessible to all developers (and their end-users). By adding accessibility features to Next.js by default, we aim to make the Web more inclusive for everyone.

Route Announcements

When transitioning between pages rendered on the server (e.g. using the `<a href>` tag) screen readers and other assistive technology announce the page title when the page loads so that users understand that the page has changed.

In addition to traditional page navigations, Next.js also supports client-side transitions for improved performance (using `next/link`). To ensure that client-side transitions are also announced to assistive technology, Next.js includes a route announcer by default.

The Next.js route announcer looks for the page name to announce by first inspecting `document.title`, then the `<h1>` element, and finally the URL pathname. For the most accessible user experience, ensure that each page in your application has a unique and descriptive title.

Linting

Next.js provides an integrated ESLint experience out of the box, including custom rules for Next.js. By default, Next.js includes `eslint-plugin-jsx-a11y` to help catch accessibility issues early, including warning on:

- aria-props
- aria-proptypes
- aria-unsupported-elements
- role-has-required-aria-props
- role-supports-aria-props

For example, this plugin helps ensure you add alt text to `img` tags, use correct `aria-*` attributes, use correct `role` attributes, and more.

Disabling JavaScript

By default, Next.js prerenders pages to static HTML files. This means that JavaScript is not required to view the HTML markup from the server and is instead used to add interactivity on the client side.

If your application requires JavaScript to be disabled, and only HTML to be used, you can remove all JavaScript from your application using an experimental flag:

```
// next.config.js
export const config = {
  unstable_runtimeJS: false,
}
```

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

---

description: Add components from the AMP community to AMP pages, and make your pages more interactive.

Adding AMP Components

The AMP community provides many components to make AMP pages more interactive. Next.js will automatically import all components used on a page and there is no need to manually import AMP component scripts:

```
export const config = { amp: true }

function MyAmpPage() {
  const date = new Date()

  return (
    <div>
      <p>Some time: {date.toJSON()}</p>
      <amp-timeago
        width="0"
        height="15"
        datetime={date.toJSON()}
        layout="responsive"
      >
        .
      </amp-timeago>
    </div>
  )
}

export default MyAmpPage
```

The above example uses the `amp-timeago` component.

By default, the latest version of a component is always imported. If you want to customize the version, you can use `next/head`, as in the following example:

```
import Head from 'next/head'

export const config = { amp: true }

function MyAmpPage() {
  const date = new Date()
```

```
  return (
    <div>
      <Head>
        <script
          async
          key="amp-timeago"
          custom-element="amp-timeago"
          src="https://cdn.ampproject.org/v0/amp-timeago-0.1.js"
        />
      </Head>

      <p>Some time: {date.toJSON()}</p>
      <amp-timeago
        width="0"
        height="15"
        datetime={date.toJSON()}
        layout="responsive"
      >
        .
      </amp-timeago>
    </div>
  )
}

export default MyAmpPage
```

---

description: Learn how AMP pages are created when used together with export.

AMP in Static HTML export

When using Static HTML export statically prerender pages, Next.js will detect if the page supports AMP and change the exporting behavior based on that.

For example, the hybrid AMP page `pages/about.js` would output:

- `out/about.html` - HTML page with client-side React runtime
- `out/about.amp.html` - AMP page

And if `pages/about.js` is an AMP-only page, then it would output:

- `out/about.html` - Optimized AMP page

Next.js will automatically insert a link to the AMP version of your page in the HTML version, so you don't have to, like so:

```
<link rel="amphtml" href="/about.amp.html" />
```

And the AMP version of your page will include a link to the HTML page:

```
<link rel="canonical" href="/about" />
```

When `trailingSlash` is enabled the exported pages for `pages/about.js` would be:

- `out/about/index.html` - HTML page
- `out/about.amp/index.html` - AMP page

---

description: AMP pages are automatically validated by Next.js during development and on build. Learn more about it here.

AMP Validation

AMP pages are automatically validated with amphtml-validator during development. Errors and warnings will appear in the terminal where you started Next.js.

Pages are also validated during Static HTML export and any warnings / errors will be printed to the terminal. Any AMP errors will cause the export to exit with status code `1` because the export is not valid AMP.

Custom Validators

You can set up custom AMP validator in `next.config.js` as shown below:

```
module.exports = {
  amp: {
    validator: './custom_validator.js',
  },
}
```

Skip AMP Validation

To turn off AMP validation add the following code to `next.config.js`

```
experimental: {
  amp: {
    skipValidation: true
  }
}
```

---

description: With minimal config, and without leaving React, you can start adding AMP and improve the performance and speed of your pages.

AMP Support

▶ **Examples**

With Next.js you can turn any React page into an AMP page, with minimal config, and without leaving React.

You can read more about AMP in the official amp.dev site.

Enabling AMP

To enable AMP support for a page, and to learn more about the different AMP configs, read the API documentation for `next/amp`.

Caveats

- Only CSS-in-JS is supported. CSS Modules aren't supported by AMP pages at the moment. You can contribute CSS Modules support to Next.js.

Related

For more information on what to do next, we recommend the following sections:

AMP Components: Make your pages more interactive with AMP components.
AMP Validation: Learn about how Next.js validates AMP pages.

---

description: Using AMP with TypeScript? Extend your typings to allow AMP components.

TypeScript

AMP currently doesn't have built-in types for TypeScript, but it's in their roadmap (#13791).

As a workaround you can manually create a file called `amp.d.ts` inside your project and add these custom types.

---

description: Next.js automatically optimizes your app to be static HTML whenever possible. Learn how it works here.

Automatic Static Optimization

Next.js automatically determines that a page is static (can be prerendered) if it has no blocking data requirements. This determination is made by the absence of `getServerSideProps` and `getInitialProps` in the page.

This feature allows Next.js to emit hybrid applications that contain **both server-rendered and statically generated pages**.

> Statically generated pages are still reactive: Next.js will hydrate your application client-side to give it full interactivity.

One of the main benefits of this feature is that optimized pages require no server-side computation, and can be instantly streamed to the end-user from multiple CDN locations. The result is an *ultra fast* loading experience for your users.

How it works

If `getServerSideProps` or `getInitialProps` is present in a page, Next.js will switch to render the page on-demand, per-request (meaning Server-Side Rendering).

If the above is not the case, Next.js will **statically optimize** your page automatically by prerendering the page to static HTML.

During prerendering, the router's `query` object will be empty since we do not have `query` information to provide during this phase. After hydration, Next.js will trigger an update to your application to provide the route parameters in the `query` object.

The cases where the query will be updated after hydration triggering another render are:

- The page is a dynamic-route.
- The page has query values in the URL.
- Rewrites are configured in your `next.config.js` since these can have parameters that may need to be parsed and provided in the `query`.

To be able to distinguish if the query is fully updated and ready for use, you can leverage the `isReady` field on `next/router`.

> **Note**: Parameters added with dynamic routes to a page that's using `getStaticProps` will always be available inside the `query` object.

`next build` will emit `.html` files for statically optimized pages. For example, the result for the page `pages/about.js` would be:

```
.next/server/pages/about.html
```

And if you add `getServerSideProps` to the page, it will then be JavaScript, like so:

```
.next/server/pages/about.js
```

Caveats

- If you have a custom `App` with `getInitialProps` then this optimization will be turned off in pages without Static Generation.
- If you have a custom `Document` with `getInitialProps` be sure you check if `ctx.req` is defined before assuming the page is server-side rendered. `ctx.req` will be `undefined` for pages that are prerendered.
- Avoid using the `asPath` value on `next/router` in the rendering tree until the router's `isReady` field is `true`. Statically optimized pages only know `asPath` on the client and not the server, so using it as a prop may lead to mismatch errors. The `active-class-name` example demonstrates one way to use `asPath` as a prop.

---

description: Learn how to configure CI to cache Next.js builds

Continuous Integration (CI) Build Caching

To improve build performance, Next.js saves a cache to `.next/cache` that is shared between builds.

To take advantage of this cache in Continuous Integration (CI) environments, your CI workflow will need to be configured to correctly persist the cache between builds.

> If your CI is not configured to persist `.next/cache` between builds, you may see a No Cache Detected error.

Here are some example cache configurations for common CI providers:

Vercel

Next.js caching is automatically configured for you. There's no action required on your part.

CircleCI

Edit your `save_cache` step in `.circleci/config.yml` to include `.next/cache`:

```
steps:
  - save_cache:
      key: dependency-cache-{{ checksum "yarn.lock" }}
      paths:
        - ./node_modules
        - ./.next/cache
```

If you do not have a `save_cache` key, please follow CircleCI's documentation on setting up build caching.

Travis CI

Add or merge the following into your `.travis.yml`:

```
cache:
  directories:
    - $HOME/.cache/yarn
    - node_modules
    - .next/cache
```

GitLab CI

Add or merge the following into your `.gitlab-ci.yml`:

```
cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
    - node_modules/
    - .next/cache/
```

Netlify CI

Use Netlify Plugins with `@netlify/plugin-nextjs`.

AWS CodeBuild

Add (or merge in) the following to your `buildspec.yml`:

```
cache:
  paths:
    - 'node_modules/**/*' # Cache `node_modules` for faster `yarn` or `npm i`
    - '.next/cache/**/*' # Cache Next.js for faster application rebuilds
```

GitHub Actions

Using GitHub's actions/cache, add the following step in your workflow file:

```
uses: actions/cache@v3
with:
  # See here for caching with `yarn` https://github.com/actions/cache/blob/main/examples.md#node---yarn or you can leverage caching with actions/setup-n
  path: |
    ~/.npm
    ${{ github.workspace }}/.next/cache
  # Generate a new cache whenever packages or source files change.
  key: ${{ runner.os }}-nextjs-${{ hashFiles('**/package-lock.json') }}-${{ hashFiles('**.[jt]s', '**.[jt]sx') }}
  # If source files changed but packages didn't, rebuild from a prior cache.
  restore-keys: |
    ${{ runner.os }}-nextjs-${{ hashFiles('**/package-lock.json') }}-
```

Bitbucket Pipelines

Add or merge the following into your `bitbucket-pipelines.yml` at the top level (same level as `pipelines`):

```
definitions:
  caches:
    nextcache: .next/cache
```

Then reference it in the `caches` section of your pipeline's `step`:

```
- step:
    name: your_step_name
    caches:
      - node
      - nextcache
```

Heroku

Using Heroku's custom cache, add a `cacheDirectories` array in your top-level package.json:

```
"cacheDirectories": [".next/cache"]
```

Azure Pipelines

Using Azure Pipelines' Cache task, add the following task to your pipeline yaml file somewhere prior to the task that executes `next build`:

```
- task: Cache@2
  displayName: 'Cache .next/cache'
  inputs:
    key: next | $(Agent.OS) | yarn.lock
    path: '$(System.DefaultWorkingDirectory)/.next/cache'
```

---

description: Use codemods to update your codebase when upgrading Next.js to the latest version

Next.js Codemods

Next.js provides Codemod transformations to help upgrade your Next.js codebase when a feature is deprecated.

Codemods are transformations that run on your codebase programmatically. This allows for a large amount of changes to be applied without having to manually go through every file.

Usage

```
npx @next/codemod@latest <transform> <path>
```

- `transform` - name of transform, see available transforms below.
- `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 13.2

`built-in-next-font`

This codemod uninstalls `@next/font` 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'
```

Next.js 13

`new-link`

Safely removes `<a>` from `next/link` or adds `legacyBehavior` prop.

For example:

```
export default function Page() {
  return (
    <Link href="/about">
      <a>About Us</a>
    </Link>
  )
}
```

Transforms into:

```
export default function Page() {
  return <Link href="/about">About Us</Link>
}
```

`next-image-to-legacy-image`

This codemod safely migrates existing Next.js 10, 11, 12 applications importing `next/image` to the renamed `next/legacy/image` import in Next.js 13.

For example:

```
import Image1 from 'next/image'
import Image2 from 'next/future/image'

export default function Home() {
  return (
    <div>
      <Image1 src="/test.jpg" width="200" height="300" />
      <Image2 src="/test.png" width="500" height="400" />
    </div>
  )
}
```

Transforms into:

```
import Image1 from 'next/legacy/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>
  )
}
```

`next-image-experimental` (experimental)

This codemod dangerously migrates from `next/legacy/image` to the new `next/image` by adding inline styles and removing unused props. Please note this codemod is experimental and only covers static usage (such as `<Image src={img} layout="responsive" />`) but not dynamic usage (such as `<Image {...props} />`).

- Removes `layout` prop and adds `style`
- Removes `objectFit` prop and adds `style`
- Removes `objectPosition` prop and adds `style`
- Removes `lazyBoundary` prop
- Removes `lazyRoot` prop
- Changes next.config.js `loader` to "custom", removes `path`, and sets `loaderFile` to a new file.

Before: intrinsic

```
import Image from 'next/image'
import img from '../img.png'

function Page() {
  return <Image src={img} />
}
```

After: intrinsic

```
import Image from 'next/image'
import img from '../img.png'

const css = { maxWidth: '100%', height: 'auto' }
function Page() {
  return <Image src={img} style={css} />
}
```

Before: responsive

```
import Image from 'next/image'
import img from '../img.png'

function Page() {
  return <Image src={img} layout="responsive" />
}
```

After: responsive

```
import Image from 'next/image'
import img from '../img.png'

const css = { width: '100%', height: 'auto' }
function Page() {
  return <Image src={img} sizes="100vw" style={css} />
}
```

Before: fill

```
import Image from 'next/image'
import img from '../img.png'

function Page() {
  return <Image src={img} layout="fill" />
}
```

After: fill

```
import Image from 'next/image'
import img from '../img.png'

function Page() {
  return <Image src={img} sizes="100vw" fill />
}
```

Before: fixed

```
import Image from 'next/image'
import img from '../img.png'

function Page() {
  return <Image src={img} layout="fixed" />
}
```

After: fixed

```
import Image from 'next/image'
import img from '../img.png'

function Page() {
  return <Image src={img} />
}
```

Next.js 11

`cra-to-next` (experimental)

Migrates a Create React App project to Next.js; creating a pages directory 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 gradual adoption of Next.js specific features.

Please share any feedback related to this transform in this discussion.

Next.js 10

`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 <div>Hello World</div>
  }
}
```

Transforms into:

```
// my-component.js
import React from 'react'
export default class Home extends React.Component {
  render() {
    return <div>Hello World</div>
  }
}
```

Next.js 9

`name-default-component`

Transforms anonymous components into named components to make sure they work with Fast Refresh.

For example:

```
// my-component.js
export default function () {
  return <div>Hello World</div>
}
```

Transforms into:

```
// my-component.js
export default function MyComponent() {
  return <div>Hello World</div>
}
```

The component will have a camel cased name based on the name of the file, and it also works with arrow functions.

Usage

Go to your project

```
cd path-to-your-project/
```

Run the codemod:

```
npx @next/codemod@latest name-default-component
```

`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,
}
```

Usage

Go to your project

```
cd path-to-your-project/
```

Run the codemod:

```
npx @next/codemod@latest withamp-to-config
```

Next.js 6

`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.

Usage

Go to your project

```
cd path-to-your-project/
```

Run the codemod:

```
npx @next/codemod@latest url-to-withrouter
```

---

description: Learn about the Next.js Compiler, written in Rust, which transforms and minifies your Next.js application.

Next.js Compiler

▼ **Version History**

Version	Changes
`v13.1.0`	Module Transpilation and Modularize Imports stable.
`v13.0.0`	SWC Minifier enabled by default.
`v12.3.0`	SWC Minifier stable.
`v12.2.0`	SWC Plugins experimental support added.
`v12.1.0`	Added support for Styled Components, Jest, Relay, Remove React Properties, Legacy Decorators, Remove Console, and jsxImportSource.
`v12.0.0`	Next.js Compiler introduced.

The 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:

```
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,
    },
  },
}
```

NOTE: 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:

1. `member`: Has type `string`. The name of the member import.
2. `lowerCase`, `upperCase`, `camelCase`, `kebabCase`: Helper functions to convert a string to lower, upper, camel or kebab cases.
3. `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/MyModule'
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.

---

description: Control page initialization and add a layout that persists for all pages by overriding the default App component used by Next.js.

Custom `App`

> **Note**: Next.js 13 introduces the `app/` directory (beta). This new directory has support for layouts, nested routes, and uses Server Components by default. Inside `app/`, you can fetch data for your entire application inside layouts, including support for more granular nested layouts (with colocated data fetching).
>
> Learn more about incrementally adopting `app/`.

Next.js uses the `App` component to initialize pages. You can override it and control the page initialization and:

- Persist layouts between page changes
- Keeping state when navigating pages
- Inject additional data into pages
- Add global CSS

To override the default `App`, create the file `./pages/_app.js` as shown below:

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

The `Component` prop is the active `page`, so whenever you navigate between routes, `Component` will change to the new `page`. Therefore, any props you send to `Component` will be received by the `page`.

`pageProps` is an object with the initial props that were preloaded for your page by one of our data fetching methods, otherwise it's an empty object.

The `App.getInitialProps` receives a single argument called `context.ctx`. It's an object with the same set of properties as the `context` object in `getInitialProps`.

Caveats

- If your app is running and you added a custom `App`, you'll need to restart the development server. Only required if `pages/_app.js` didn't exist before.
- Adding a custom `getInitialProps` in your `App` will disable Automatic Static Optimization in pages without Static Generation.
- When you add `getInitialProps` in your custom app, you must `import App from "next/app"`, call `App.getInitialProps(appContext)` inside `getInitialProps` and merge the returned object into the return value.
- `App` does not support Next.js Data Fetching methods like `getStaticProps` or `getServerSideProps`. If you need global data fetching, consider incrementally adopting the `app/` directory.

TypeScript

If you're using TypeScript, take a look at our TypeScript documentation.

Related

For more information on what to do next, we recommend the following sections:

Automatic Static Optimization: Next.js automatically optimizes your app to be static HTML whenever possible. Learn how it works here.
Custom Error Page: Learn more about the built-in Error page.

---

description: Extend the default document markup added by Next.js.

Custom `Document`

> **Note**: Next.js 13 introduces the `app/` directory (beta). This new directory has support for layouts, nested routes, and uses Server Components by default. Inside `app/`, you can modify the initial `html` and `body` tags using a root layout.
>
> Learn more about incrementally adopting `app/`.

A custom `Document` can update the `<html>` and `<body>` tags used to render a Page. This file is only rendered on the server, so event handlers like `onClick` cannot be used in `_document`.

To override the default `Document`, create the file `pages/_document.js` as shown below:

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

export default function Document() {
  return (
    <Html>
      <Head />
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

The code above is the default `Document` added by Next.js. Custom attributes are allowed as props. For example, we might want to add `lang="en"` to the `<html>` tag:

```
<Html lang="en">
```

Or add a `className` to the `body` tag:

```
<body className="bg-white">
```

`<Html>`, `<Head />`, `<Main />` and `<NextScript />` are required for the page to be properly rendered.

Caveats

- The `<Head />` component used in `_document` is not the same as `next/head`. The `<Head />` component used here should only be used for any `<head>` code that is common for all pages. For all other cases, such as `<title>` tags, we recommend using `next/head` in your pages or components.
- React components outside of `<Main />` will not be initialized by the browser. Do *not* add application logic here or custom CSS (like `styled-jsx`). If you need shared components in all your pages (like a menu or a toolbar), read Layouts instead.
- `Document` currently does not support Next.js Data Fetching methods like `getStaticProps` or `getServerSideProps`.

Customizing `renderPage`

> **Note**: This is advanced and only needed for libraries like CSS-in-JS to support server-side rendering. This is not needed for built-in `styled-jsx` support.

For React 18 support, we recommend avoiding customizing `getInitialProps` and `renderPage`, if possible.

The `ctx` object shown below is equivalent to the one received in `getInitialProps`, with the addition of `renderPage`.

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

class MyDocument extends Document {
  static async getInitialProps(ctx) {
    const originalRenderPage = ctx.renderPage

    // Run the React rendering logic synchronously
    ctx.renderPage = () =>
      originalRenderPage({
        // Useful for wrapping the whole react tree
        enhanceApp: (App) => App,
        // Useful for wrapping in a per-page basis
        enhanceComponent: (Component) => Component,
      })

    // Run the parent `getInitialProps`, it now includes the custom `renderPage`
    const initialProps = await Document.getInitialProps(ctx)

    return initialProps
  }
```

```
  render() {
    return (
      <Html>
        <Head />
        <body>
          <Main />
          <NextScript />
        </body>
      </Html>
    )
  }
}

export default MyDocument
```

> **Note**: `getInitialProps` in `_document` is not called during client-side transitions.

TypeScript

You can use the built-in `DocumentContext` type and change the file name to `./pages/_document.tsx` like so:

```
import Document, { DocumentContext, DocumentInitialProps } from 'next/document'

class MyDocument extends Document {
  static async getInitialProps(
    ctx: DocumentContext
  ): Promise<DocumentInitialProps> {
    const initialProps = await Document.getInitialProps(ctx)

    return initialProps
  }
}

export default MyDocument
```

---

description: Override and extend the built-in Error page to handle custom errors.

Custom Error Page

404 Page

A 404 page may be accessed very often. Server-rendering an error page for every visit increases the load of the Next.js server. This can result in increased costs and slow experiences.

To avoid the above pitfalls, Next.js provides a static 404 page by default without having to add any additional files.

Customizing The 404 Page

To create a custom 404 page you can create a `pages/404.js` file. This file is statically generated at build time.

```
// pages/404.js
export default function Custom404() {
  return <h1>404 - Page Not Found</h1>
}
```

> **Note**: You can use `getStaticProps` inside this page if you need to fetch data at build time.

500 Page

Server-rendering an error page for every visit adds complexity to responding to errors. To help users get responses to errors as fast as possible, Next.js provides a static 500 page by default without having to add any additional files.

Customizing The 500 Page

To customize the 500 page you can create a `pages/500.js` file. This file is statically generated at build time.

```
// pages/500.js
export default function Custom500() {
  return <h1>500 - Server-side error occurred</h1>
}
```

> **Note**: You can use `getStaticProps` inside this page if you need to fetch data at build time.

More Advanced Error Page Customizing

500 errors are handled both client-side and server-side by the `Error` component. If you wish to override it, define the file `pages/_error.js` and add the following code:

```
function Error({ statusCode }) {
  return (
    <p>
      {statusCode
        ? `An error ${statusCode} occurred on server`
        : 'An error occurred on client'}
    </p>
  )
}

Error.getInitialProps = ({ res, err }) => {
  const statusCode = res ? res.statusCode : err ? err.statusCode : 404
  return { statusCode }
}

export default Error
```

> `pages/_error.js` is only used in production. In development you'll get an error with the call stack to know where the error originated from.

Reusing the built-in error page

If you want to render the built-in error page you can by importing the `Error` component:

```
import Error from 'next/error'

export async function getServerSideProps() {
  const res = await fetch('https://api.github.com/repos/vercel/next.js')
  const errorCode = res.ok ? false : res.status
  const json = await res.json()

  return {
    props: { errorCode, stars: json.stargazers_count },
  }
}

export default function Page({ errorCode, stars }) {
  if (errorCode) {
    return <Error statusCode={errorCode} />
  }

  return <div>Next stars: {stars}</div>
}
```

The `Error` component also takes `title` as a property if you want to pass in a text message along with a `statusCode`.

If you have a custom `Error` component be sure to import that one instead. `next/error` exports the default component used by Next.js.

Caveats

- `Error` does not currently support Next.js Data Fetching methods like `getStaticProps` or `getServerSideProps`.
- `_error`, like `_app`, is a reserved pathname. `_error` is used to define the customized layouts and behaviors of the error pages. `/_error` will render 404 when accessed directly via routing or rendering in a custom server.

---

description: Extend the babel preset added by Next.js with your own configs.

Customizing Babel Config

▶ **Examples**

Next.js includes the `next/babel` preset to your app, which includes everything needed to compile React applications and server-side code. But if you want to extend the default Babel configs, it's also possible.

To start, you only need to define a `.babelrc` file (or `babel.config.js`) at the top of your app. If such a file is found, it will be considered as the *source of truth*, and therefore it needs to define what Next.js needs as well, which is the `next/babel` preset.

Here's an example `.babelrc` file:

```
{
  "presets": ["next/babel"],
  "plugins": []
}
```

You can take a look at this file to learn about the presets included by `next/babel`.

To add presets/plugins **without configuring them**, you can do it this way:

```
{
  "presets": ["next/babel"],
  "plugins": ["@babel/plugin-proposal-do-expressions"]
}
```

To add presets/plugins **with custom configuration**, do it on the `next/babel` preset like so:

```
{
  "presets": [
    [
      "next/babel",
      {
        "preset-env": {},
        "transform-runtime": {},
        "styled-jsx": {},
        "class-properties": {}
      }
    ]
  ],
  "plugins": []
}
```

To learn more about the available options for each config, visit their documentation site.

> Next.js uses the **current** Node.js version for server-side compilations.
>
> The `modules` option on `"preset-env"` should be kept to `false`, otherwise webpack code splitting is turned off.

---

description: Extend the PostCSS config and plugins added by Next.js with your own.

Customizing PostCSS Config

▼ **Examples**

- Tailwind CSS Example

Default Behavior

Next.js compiles CSS for its built-in CSS support using PostCSS.

Out of the box, with no configuration, Next.js compiles CSS with the following transformations:

1. Autoprefixer automatically adds vendor prefixes to CSS rules (back to IE11).
2. Cross-browser Flexbox bugs are corrected to behave like the spec.
3. New CSS features are automatically compiled for Internet Explorer 11 compatibility:
   - `all` Property
   - Break Properties
   - `font-variant` Property
   - Gap Properties
   - Media Query Ranges

By default, CSS Grid and Custom Properties (CSS variables) are **not compiled** for IE11 support.

To compile CSS Grid Layout for IE11, you can place the following comment at the top of your CSS file:

```
/* autoprefixer grid: autoplace */
```

You can also enable IE11 support for CSS Grid Layout in your entire project by configuring autoprefixer with the configuration shown below (collapsed). See "Customizing Plugins" below for more information.

▶ **Click to view the configuration to enable CSS Grid Layout**

CSS variables are not compiled because it is not possible to safely do so. If you must use variables, consider using something like Sass variables which are compiled away by Sass.

Customizing Target Browsers

Next.js allows you to configure the target browsers (for Autoprefixer and compiled css features) through Browserslist.

To customize browserslist, create a `browserslist` key in your `package.json` like so:

```
{
  "browserslist": [">0.3%", "not dead", "not op_mini all"]
}
```

You can use the browsersl.ist tool to visualize what browsers you are targeting.

CSS Modules

No configuration is needed to support CSS Modules. To enable CSS Modules for a file, rename the file to have the extension `.module.css`.

You can learn more about Next.js' CSS Module support here.

Customizing Plugins

> **Warning**: When you define a custom PostCSS configuration file, Next.js **completely disables** the default behavior. Be sure to manually configure all the features you need compiled, including Autoprefixer. You also need to install any plugins included in your custom configuration manually, i.e. `npm install postcss-flexbugs-fixes postcss-preset-env`.

To customize the PostCSS configuration, create a `postcss.config.json` file in the root of your project.

This is the default configuration used by Next.js:

```
{
  "plugins": [
    "postcss-flexbugs-fixes",
    [
      "postcss-preset-env",
      {
        "autoprefixer": {
          "flexbox": "no-2009"
        },
        "stage": 3,
        "features": {
          "custom-properties": false
        }
      }
    ]
  ]
}
```

> **Note**: Next.js also allows the file to be named `.postcssrc.json`, or, to be read from the `postcss` key in `package.json`.

It is also possible to configure PostCSS with a `postcss.config.js` file, which is useful when you want to conditionally include plugins based on environment:

```
module.exports = {
  plugins:
    process.env.NODE_ENV === 'production'
      ? [
          'postcss-flexbugs-fixes',
          [
            'postcss-preset-env',
            {
              autoprefixer: {
                flexbox: 'no-2009',
              },
              stage: 3,
              features: {
                'custom-properties': false,
              },
            },
          ],
        ]
      : [
          // No transformations in development
        ],
}
```

> **Note**: Next.js also allows the file to be named `.postcssrc.js`.

Do **not use `require()`** to import the PostCSS Plugins. Plugins must be provided as strings.

> **Note**: If your `postcss.config.js` needs to support other non-Next.js tools in the same project, you must use the interoperable object-based format instead:

```
module.exports = {
  plugins: {
    'postcss-flexbugs-fixes': {},
    'postcss-preset-env': {
      autoprefixer: {
        flexbox: 'no-2009',
      },
      stage: 3,
      features: {
        'custom-properties': false,
      },
    },
  },
}
```

---

description: Start a Next.js app programmatically using a custom server.

Custom Server

▶ **Examples**

By default, Next.js includes its own server with `next start`. If you have an existing backend, you can still use it with Next.js (this is not a custom server). A custom Next.js server allows you to start a server 100% programmatically in order to use custom server patterns. Most of the time, you will not need this – but it's available for complete customization.

> **Note**: A custom server **cannot** be deployed on Vercel.
>
> Before deciding to use a custom server, please keep in mind that it should only be used when the integrated router of Next.js can't meet your app requirements. A custom server will remove important performance optimizations, like **serverless functions** and **Automatic Static Optimization**.

Take a look at the following example of a custom server:

```
// server.js
const { createServer } = require('http')
const { parse } = require('url')
const next = require('next')

const dev = process.env.NODE_ENV !== 'production'
const hostname = 'localhost'
const port = 3000
// when using middleware `hostname` and `port` must be provided below
const app = next({ dev, hostname, port })
const handle = app.getRequestHandler()

app.prepare().then(() => {
  createServer(async (req, res) => {
    try {
      // Be sure to pass `true` as the second argument to `url.parse`.
      // This tells it to parse the query portion of the URL.
      const parsedUrl = parse(req.url, true)
      const { pathname, query } = parsedUrl

      if (pathname === '/a') {
        await app.render(req, res, '/a', query)
      } else if (pathname === '/b') {
        await app.render(req, res, '/b', query)
      } else {
        await handle(req, res, parsedUrl)
      }
    } catch (err) {
      console.error('Error occurred handling', req.url, err)
      res.statusCode = 500
      res.end('Internal Server Error')
    }
  })
    .once('error', (err) => {
      console.error(err)
      process.exit(1)
    })
    .listen(port, () => {
      console.log(`> Ready on http://${hostname}:${port}`)
    })
})
```

> `server.js` doesn't go through babel or webpack. Make sure the syntax and sources this file requires are compatible with the current node version you are running.

To run the custom server you'll need to update the `scripts` in `package.json` like so:

```
"scripts": {
  "dev": "node server.js",
  "build": "next build",
  "start": "NODE_ENV=production node server.js"
}
```

The custom server uses the following import to connect the server with the Next.js application:

```
const next = require('next')
const app = next({})
```

The above `next` import is a function that receives an object with the following options:

- `dev`: `Boolean` - Whether or not to launch Next.js in dev mode. Defaults to `false`
- `dir`: `String` - Location of the Next.js project. Defaults to `'.'`
- `quiet`: `Boolean` - Hide error messages containing server information. Defaults to `false`
- `conf`: `object` - The same object you would use in next.config.js. Defaults to `{}`

The returned `app` can then be used to let Next.js handle requests as required.

Disabling file-system routing

By default, `Next` will serve each file in the `pages` folder under a pathname matching the filename. If your project uses a custom server, this behavior may result in the same content being served from multiple paths, which can present problems with SEO and UX.

To disable this behavior and prevent routing based on files in `pages`, open `next.config.js` and disable the `useFileSystemPublicRoutes` config:

```
module.exports = {
  useFileSystemPublicRoutes: false,
}
```

> Note that `useFileSystemPublicRoutes` disables filename routes from SSR; client-side routing may still access those paths. When using this option, you should guard against navigation to routes you do not want programmatically.
>
> You may also wish to configure the client-side router to disallow client-side redirects to filename routes; for that refer to `router.beforePopState`.

---

description: Debug your Next.js app.

Debugging

This documentation explains how you can debug your Next.js frontend and backend code with full source maps support using either the VS Code debugger or Chrome DevTools.

Any debugger that can attach to Node.js can also be used to debug a Next.js application. You can find more details in the Node.js Debugging Guide.

Debugging with VS Code

Create a file named `.vscode/launch.json` at the root of your project with the following content:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Next.js: debug server-side",
      "type": "node-terminal",
      "request": "launch",
      "command": "npm run dev"
    },
    {
      "name": "Next.js: debug client-side",
      "type": "chrome",
      "request": "launch",
      "url": "http://localhost:3000"
    },
    {
      "name": "Next.js: debug full stack",
      "type": "node-terminal",
      "request": "launch",
      "command": "npm run dev",
      "serverReadyAction": {
        "pattern": "started server on .+, url: (https?://.+)",
        "uriFormat": "%s",
        "action": "debugWithChrome"
      }
    }
  ]
}
```

`npm run dev` can be replaced with `yarn dev` if you're using Yarn. If you're changing the port number your application starts on, replace the `3000` in `http://localhost:3000` with the port you're using instead.

Now go to the Debug panel (`Ctrl+Shift+D` on Windows/Linux, ⇧+⌘+D on macOS), select a launch configuration, then press `F5` or select **Debug: Start Debugging** from the Command Palette to start your debugging session.

Using the Debugger in Jetbrains WebStorm

Click the drop down menu listing the runtime configuration, and click `Edit Configurations...`. Create a `Javascript Debug` debug configuration with `http://localhost:3000` as the URL. Customize to your liking (e.g. Browser for debugging, store as project file), and click `OK`. Run this debug configuration, and the selected browser should automatically open. At this point, you should have 2 applications in debug mode: the NextJS node application, and the client/ browser application.

Debugging with Chrome DevTools

Client-side code

Start your development server as usual by running `next dev`, `npm run dev`, or `yarn dev`. Once the server starts, open `http://localhost:3000` (or your alternate URL) in Chrome. Next, open Chrome's Developer Tools (`Ctrl+Shift+J` on Windows/Linux, ⌥+⌘+I on macOS), then go to the **Sources** tab.

Now, any time your client-side code reaches a `debugger` statement, code execution will pause and that file will appear in the debug area. You can also press `Ctrl+P` on Windows/Linux or ⌘+P on macOS to search for a file and set breakpoints manually. Note that when searching here, your source files will have paths starting with `webpack://_N_E/./`.

Server-side code

To debug server-side Next.js code with Chrome DevTools, you need to pass the `--inspect` flag to the underlying Node.js process:

```
NODE_OPTIONS='--inspect' next dev
```

If you're using `npm run dev` or `yarn dev` (see Getting Started) then you should update the `dev` script on your `package.json`:

```
"dev": "NODE_OPTIONS='--inspect' next dev"
```

Launching the Next.js dev server with the `--inspect` flag will look something like this:

```
Debugger listening on ws://127.0.0.1:9229/0cf90313-350d-4466-a748-cd60f4e47c95
For help, see: https://nodejs.org/en/docs/inspector
```

```
ready - started server on 0.0.0.0:3000, url: http://localhost:3000
```

> Be aware that running `NODE_OPTIONS='--inspect' npm run dev` or `NODE_OPTIONS='--inspect' yarn dev` won't work. This would try to start multiple debuggers on the same port: one for the npm/yarn process and one for Next.js. You would then get an error like `Starting inspector on 127.0.0.1:9229 failed: address already in use` in your console.

Once the server starts, open a new tab in Chrome and visit `chrome://inspect`, where you should see your Next.js application inside the **Remote Target** section. Click **inspect** under your application to open a separate DevTools window, then go to the **Sources** tab.

Debugging server-side code here works much like debugging client-side code with Chrome DevTools, except that when you search for files here with `Ctrl+P` or `⌘+P`, your source files will have paths starting with `webpack://{application-name}/./` (where `{application-name}` will be replaced with the name of your application according to your `package.json` file).

Debugging on Windows

Windows users may run into an issue when using `NODE_OPTIONS='--inspect'` as that syntax is not supported on Windows platforms. To get around this, install the `cross-env` package as a development dependency (`-D` with `npm` and `yarn`) and replace the `dev` script with the following.

```
"dev": "cross-env NODE_OPTIONS='--inspect' next dev",
```

`cross-env` will set the `NODE_OPTIONS` environment variable regardless of which platform you are on (including Mac, Linux, and Windows) and allow you to debug consistently across devices and operating systems.

> **Note**: Ensure Windows Defender is disabled on your machine. This external service will check *every file read*, which has been reported to greatly increase Fast Refresh time with `next dev`. This is a known issue, not related to Next.js, but it does affect Next.js development.

More information

To learn more about how to use a JavaScript debugger, take a look at the following documentation:

- Node.js debugging in VS Code: Breakpoints
- Chrome DevTools: Debug JavaScript

---

description: Dynamically import JavaScript modules and React Components and split your code into manageable chunks.

Dynamic Import

▼ **Examples**

- Dynamic Import

Next.js supports lazy loading external libraries with `import()` and React components with `next/dynamic`. Deferred loading helps improve the initial loading performance by decreasing the amount of JavaScript necessary to render the page. Components or libraries are only imported and included in the JavaScript bundle when they're used.

`next/dynamic` is a composite extension of `React.lazy` and `Suspense`, components can delay hydration until the Suspense boundary is resolved.

Example

By using `next/dynamic`, the header component will not be included in the page's initial JavaScript bundle. The page will render the Suspense `fallback` first, followed by the `Header` component when the `Suspense` boundary is resolved.

```
import dynamic from 'next/dynamic'

const DynamicHeader = dynamic(() => import('../components/header'), {
  loading: () => <p>Loading...</p>,
})

export default function Home() {
  return <DynamicHeader />
}
```

> **Note**: In `import('path/to/component')`, the path must be explicitly written. It can't be a template string nor a variable. Furthermore the `import()` has to be inside the `dynamic()` call for Next.js to be able to match webpack bundles / module ids to the specific `dynamic()` call and preload them before rendering. `dynamic()` can't be used inside of React rendering as it needs to be marked in the top level of the module for preloading to work, similar to `React.lazy`.

With named exports

To dynamically import a named export, you can return it from the Promise returned by `import()`:

```
// components/hello.js
export function Hello() {
  return <p>Hello!</p>
}

// pages/index.js
import dynamic from 'next/dynamic'

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

With no SSR

To dynamically load a component on the client side, you can use the `ssr` option to disable server-rendering. This is useful if an external dependency or component relies on browser APIs like `window`.

```
import dynamic from 'next/dynamic'

const DynamicHeader = dynamic(() => import('../components/header'), {
  ssr: false,
})
```

With external libraries

This example uses the external library `fuse.js` for fuzzy search. The module is only loaded in the browser after the user types in the search input.

```
import { useState } from 'react'

const names = ['Tim', 'Joe', 'Bel', 'Lee']

export default function Page() {
  const [results, setResults] = useState()

  return (
    <div>
      <input
        type="text"
        placeholder="Search"
        onChange={async (e) => {
          const { value } = e.currentTarget
          // Dynamically load fuse.js
          const Fuse = (await import('fuse.js')).default
          const fuse = new Fuse(names)

          setResults(fuse.search(value))
        }}
      />
      <pre>Results: {JSON.stringify(results, null, 2)}</pre>
    </div>
  )
}
```

---

description: Handle errors in your Next.js app.

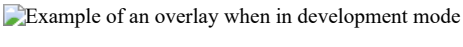
Error Handling

This documentation explains how you can handle development, server-side, and client-side errors.

Handling Errors in Development

When there is a runtime error during the development phase of your Next.js application, you will encounter an **overlay**. It is a modal that covers the webpage. It is only visible when the development server runs using `next dev`, `npm run dev`, or `yarn dev` and not in production. Fixing the error will automatically dismiss the overlay.

Here is an example of an overlay:


Example of an overlay when in development mode

Handling Server Errors

Next.js provides a static 500 page by default to handle server-side errors that occur in your application. You can also customize this page by creating a `pages/500.js` file.

Having a 500 page in your application does not show specific errors to the app user.

You can also use 404 page to handle specific runtime error like `file not found`.

Handling Client Errors

React Error Boundaries is a graceful way to handle a JavaScript error on the client so that the other parts of the application continue working. In addition to preventing the page from crashing, it allows you to provide a custom fallback component and even log error information.

To use Error Boundaries for your Next.js application, you must create a class component `ErrorBoundary` and wrap the `Component` prop in the `pages/_app.js` file. This component will be responsible to:

- Render a fallback UI after an error is thrown
- Provide a way to reset the Application's state
- Log error information

You can create an `ErrorBoundary` class component by extending `React.Component`. For example:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props)

    // Define a state variable to track whether is an error or not
    this.state = { hasError: false }
  }
  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI

    return { hasError: true }
  }
  componentDidCatch(error, errorInfo) {
    // You can use your own error logging service here
    console.log({ error, errorInfo })
  }
  render() {
    // Check if the error is thrown
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return (
        <div>
          <h2>Oops, there is an error!</h2>
          <button
            type="button"
            onClick={() => this.setState({ hasError: false })}
          >
            Try again?
          </button>
        </div>
      )
```

```
    }

    // Return children components in case of no error

    return this.props.children
  }
}

export default ErrorBoundary
```

The `ErrorBoundary` component keeps track of an `hasError` state. The value of this state variable is a boolean. When the value of `hasError` is `true`, then the `ErrorBoundary` component will render a fallback UI. Otherwise, it will render the children components.

After creating an `ErrorBoundary` component, import it in the `pages/_app.js` file to wrap the `Component` prop in your Next.js application.

```
// Import the ErrorBoundary component
import ErrorBoundary from '../components/ErrorBoundary'

function MyApp({ Component, pageProps }) {
  return (
    // Wrap the Component prop with ErrorBoundary component
    <ErrorBoundary>
      <Component {...pageProps} />
    </ErrorBoundary>
  )
}

export default MyApp
```

You can learn more about Error Boundaries in React's documentation.

Reporting Errors

To monitor client errors, use a service like Sentry, Bugsnag or Datadog.

---

description: Next.js has built-in support for internationalized routing and language detection. Learn more here.

Internationalized Routing

▶ **Examples**

Next.js has built-in support for internationalized (i18n) routing since `v10.0.0`. You can provide a list of locales, the default locale, and domain-specific locales and Next.js will automatically handle the routing.

The i18n routing support is currently meant to complement existing i18n library solutions like `react-intl`, `react-i18next`, `lingui`, `rosetta`, `next-intl`, `next-translate`, `next-multilingual`, `typesafe-i18n`, and others by streamlining the routes and locale parsing.

Getting started

To get started, add the `i18n` config to your `next.config.js` file.

Locales are UTS Locale Identifiers, a standardized format for defining locales.

Generally a Locale Identifier is made up of a language, region, and script separated by a dash: `language-region-script`. The region and script are optional. An example:

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

If user locale is `nl-BE` and it is not listed in your configuration, they will be redirected to `nl` if available, or to the default locale otherwise. If you don't plan to support all regions of a country, it is therefore a good practice to include country locales that will act as fallbacks.

```
// next.config.js
module.exports = {
  i18n: {
    // These are all the locales you want to support in
    // your application
    locales: ['en-US', 'fr', 'nl-NL'],
    // This is the default locale you want to be used when visiting
    // a non-locale prefixed path e.g. `/hello`
    defaultLocale: 'en-US',
    // This is a list of locale domains and the default locale they
    // should handle (these are only required when setting up domain routing)
    // Note: subdomains must be included in the domain value to be matched e.g. "fr.example.com".
    domains: [
      {
        domain: 'example.com',
        defaultLocale: 'en-US',
      },
      {
        domain: 'example.nl',
        defaultLocale: 'nl-NL',
      },
      {
        domain: 'example.fr',
        defaultLocale: 'fr',
        // an optional http field can also be used to test
        // locale domains locally with http instead of https
        http: true,
      },
    ],
  },
}
```

Locale Strategies

There are two locale handling strategies: Sub-path Routing and Domain Routing.

Sub-path Routing

Sub-path Routing puts the locale in the url path.

```
// next.config.js
module.exports = {
  i18n: {
    locales: ['en-US', 'fr', 'nl-NL'],
    defaultLocale: 'en-US',
  },
}
```

With the above configuration `en-US`, `fr`, and `nl-NL` will be available to be routed to, and `en-US` is the default locale. If you have a `pages/blog.js` the following urls would be available:

- `/blog`
- `/fr/blog`
- `/nl-nl/blog`

The default locale does not have a prefix.

Domain Routing

By using domain routing you can configure locales to be served from different domains:

```
// next.config.js
module.exports = {
  i18n: {
    locales: ['en-US', 'fr', 'nl-NL', 'nl-BE'],
    defaultLocale: 'en-US',

    domains: [
      {
        // Note: subdomains must be included in the domain value to be matched
        // e.g. www.example.com should be used if that is the expected hostname
        domain: 'example.com',
        defaultLocale: 'en-US',
      },
      {
        domain: 'example.fr',
        defaultLocale: 'fr',
      },
      {
        domain: 'example.nl',
        defaultLocale: 'nl-NL',
        // specify other locales that should be redirected
        // to this domain
        locales: ['nl-BE'],
      },
    ],
  },
}
```

For example if you have `pages/blog.js` the following urls will be available:

- `example.com/blog`
- `www.example.com/blog`
- `example.fr/blog`
- `example.nl/blog`
- `example.nl/nl-BE/blog`

Automatic Locale Detection

When a user visits the application root (generally `/`), Next.js will try to automatically detect which locale the user prefers based on the `Accept-Language` header and the current domain.

If a locale other than the default locale is detected, the user will be redirected to either:

- **When using Sub-path Routing:** The locale prefixed path
- **When using Domain Routing:** The domain with that locale specified as the default

When using Domain Routing, if a user with the `Accept-Language` header `fr;q=0.9` visits `example.com`, they will be redirected to `example.fr` since that domain handles the `fr` locale by default.

When using Sub-path Routing, the user would be redirected to `/fr`.

Prefixing the Default Locale

With Next.js 12 and Middleware, we can add a prefix to the default locale with a workaround.

For example, here's a `next.config.js` file with support for a few languages. Note the `"default"` locale has been added intentionally.

```
// next.config.js

module.exports = {
  i18n: {
    locales: ['default', 'en', 'de', 'fr'],
    defaultLocale: 'default',
    localeDetection: false,
  },
  trailingSlash: true,
}
```

Next, we can use Middleware to add custom routing rules:

```
// middleware.ts

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

const PUBLIC_FILE = /\.(.*)$/

export async function middleware(req: NextRequest) {
```

```
  if (
    req.nextUrl.pathname.startsWith('/_next') ||
    req.nextUrl.pathname.includes('/api/') ||
    PUBLIC_FILE.test(req.nextUrl.pathname)
  ) {
    return
  }

  if (req.nextUrl.locale === 'default') {
    const locale = req.cookies.get('NEXT_LOCALE')?.value || 'en'

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

This Middleware skips adding the default prefix to API Routes and public files like fonts or images. If a request is made to the default locale, we redirect to our prefix `/en`.

Disabling Automatic Locale Detection

The automatic locale detection can be disabled with:

```
// next.config.js
module.exports = {
  i18n: {
    localeDetection: false,
  },
}
```

When `localeDetection` is set to `false` Next.js will no longer automatically redirect based on the user's preferred locale and will only provide locale information detected from either the locale based domain or locale path as described above.

Accessing the locale information

You can access the locale information via the Next.js router. For example, using the `useRouter()` hook the following properties are available:

- `locale` contains the currently active locale.
- `locales` contains all configured locales.
- `defaultLocale` contains the configured default locale.

When pre-rendering pages with `getStaticProps` or `getServerSideProps`, the locale information is provided in the context provided to the function.

When leveraging `getStaticPaths`, the configured locales are provided in the context parameter of the function under `locales` and the configured defaultLocale under `defaultLocale`.

Transition between locales

You can use `next/link` or `next/router` to transition between locales.

For `next/link`, a `locale` prop can be provided to transition to a different locale from the currently active one. If no `locale` prop is provided, the currently active `locale` is used during client-transitions. For example:

```
import Link from 'next/link'

export default function IndexPage(props) {
  return (
    <Link href="/another" locale="fr">
      To /fr/another
    </Link>
  )
}
```

When using the `next/router` methods directly, you can specify the `locale` that should be used via the transition options. For example:

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

export default function IndexPage(props) {
  const router = useRouter()

  return (
    <div
      onClick={() => {
        router.push('/another', '/another', { locale: 'fr' })
      }}
    >
      to /fr/another
    </div>
  )
}
```

Note that to handle switching only the `locale` while preserving all routing information such as dynamic route query values or hidden href query values, you can provide the `href` parameter as an object:

```
import { useRouter } from 'next/router'
const router = useRouter()
const { pathname, asPath, query } = router
// change just the locale and maintain all other route information including href's query
router.push({ pathname, query }, asPath, { locale: nextLocale })
```

See here for more information on the object structure for `router.push`.

If you have a `href` that already includes the locale you can opt-out of automatically handling the locale prefixing:

```
import Link from 'next/link'

export default function IndexPage(props) {
  return (
    <Link href="/fr/another" locale={false}>
      To /fr/another
    </Link>
  )
}
```

Leveraging the NEXT_LOCALE cookie

Next.js supports overriding the accept-language header with a `NEXT_LOCALE=the-locale` cookie. This cookie can be set using a language switcher and then when a user comes back to the site it will leverage the locale specified in the cookie when redirecting from `/` to the correct locale location.

For example, if a user prefers the locale `fr` in their accept-language header but a `NEXT_LOCALE=en` cookie is set the `en` locale when visiting `/` the user will be redirected to the `en` locale location until the cookie is removed or expired.

Search Engine Optimization

Since Next.js knows what language the user is visiting it will automatically add the `lang` attribute to the `<html>` tag.

Next.js doesn't know about variants of a page so it's up to you to add the `hreflang` meta tags using `next/head`. You can learn more about `hreflang` in the Google Webmasters documentation.

How does this work with Static Generation?

> Note that Internationalized Routing does not integrate with `output: 'export'` as it does not leverage the Next.js routing layer. Hybrid Next.js applications that do not use `output: 'export'` are fully supported.

Dynamic Routes and `getStaticProps` Pages

For pages using `getStaticProps` with Dynamic Routes, all locale variants of the page desired to be prerendered need to be returned from `getStaticPaths`. Along with the `params` object returned for `paths`, you can also return a `locale` field specifying which locale you want to render. For example:

```
// pages/blog/[slug].js
export const getStaticPaths = ({ locales }) => {
  return {
    paths: [
      // if no `locale` is provided only the defaultLocale will be generated
      { params: { slug: 'post-1' }, locale: 'en-US' },
      { params: { slug: 'post-1' }, locale: 'fr' },
    ],
    fallback: true,
  }
}
```

For Automatically Statically Optimized and non-dynamic `getStaticProps` pages, **a version of the page will be generated for each locale**. This is important to consider because it can increase build times depending on how many locales are configured inside `getStaticProps`.

For example, if you have 50 locales configured with 10 non-dynamic pages using `getStaticProps`, this means `getStaticProps` will be called 500 times. 50 versions of the 10 pages will be generated during each build.

To decrease the build time of dynamic pages with `getStaticProps`, use a `fallback` mode. This allows you to return only the most popular paths and locales from `getStaticPaths` for prerendering during the build. Then, Next.js will build the remaining pages at runtime as they are requested.

Automatically Statically Optimized Pages

For pages that are automatically statically optimized, a version of the page will be generated for each locale.

Non-dynamic getStaticProps Pages

For non-dynamic `getStaticProps` pages, a version is generated for each locale like above. `getStaticProps` is called with each `locale` that is being rendered. If you would like to opt-out of a certain locale from being pre-rendered, you can return `notFound: true` from `getStaticProps` and this variant of the page will not be generated.

```
export async function getStaticProps({ locale }) {
  // Call an external API endpoint to get posts.
  // You can use any data fetching library
  const res = await fetch(`https://.../posts?locale=${locale}`)
  const posts = await res.json()

  if (posts.length === 0) {
    return {
      notFound: true,
    }
  }

  // By returning { props: posts }, the Blog component
  // will receive `posts` as a prop at build time
  return {
    props: {
      posts,
    },
  }
}
```

Limits for the i18n config

- `locales`: 100 total locales
- `domains`: 100 total locale domain items

> **Note**: These limits have been added initially to prevent potential performance issues at build time. You can workaround these limits with custom routing using Middleware in Next.js 12.

---

description: Learn how to use instrumentation to run code at server startup in your Next.js app

> **Note**: This feature is experimental. To use it, you must explicitly opt in by defining `experimental.instrumentationHook = true;` in your `next.config.js`.

`instrumentation.ts`

If you export a function named `register` from this file, we will call that function whenever a new Next.js server instance is bootstrapped. 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:

```
// /instrumentation.ts

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:

```
// /instrumentation.ts

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:

```
// /instrumentation.ts

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

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

---

description: Measure and track page performance using Next.js Speed Insights

Measuring performance

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.

Build Your Own

First, you will need to create a custom App component and define a `reportWebVitals` function:

```
// pages/_app.js
export function reportWebVitals(metric) {
  console.log(metric)
}

function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}

export default MyApp
```

This function is fired when the final values for any of the metrics have finished calculating on the page. You can use to log any of the results to the console or send to a particular endpoint.

The `metric` object returned to the function consists of a number of properties:

- `id`: Unique identifier for the metric in the context of the current page load
- `name`: Metric name
- `startTime`: First recorded timestamp of the performance entry in milliseconds (if applicable)
- `value`: Value, or duration in milliseconds, of the performance entry
- `label`: Type of metric (`web-vital` or `custom`)

There are two types of metrics that are tracked:

- Web Vitals
- Custom metrics

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)*

You can handle all the results of these metrics using the `web-vital` label:

```
export function reportWebVitals(metric) {
  if (metric.label === 'web-vital') {
    console.log(metric) // The metric object ({ id, name, startTime, value, label }) is logged to the console
```

```
  }
}
```

There's also the option of handling each of the metrics separately:

```
export function reportWebVitals(metric) {
  switch (metric.name) {
    case 'FCP':
      // handle FCP results
      break
    case 'LCP':
      // handle LCP results
      break
    case 'CLS':
      // handle CLS results
      break
    case 'FID':
      // handle FID results
      break
    case 'TTFB':
      // handle TTFB results
      break
    case 'INP':
      // handle INP results (note: INP is still an experimental metric)
      break
    default:
      break
  }
}
```

A third-party library, web-vitals, is used to measure these metrics. Browser compatibility depends on the particular metric, so refer to the Browser Support section to find out which browsers are supported.

Custom metrics

In addition to the core metrics listed above, there are some additional custom metrics that measure the time it takes for the page to hydrate and render:

- `Next.js-hydration`: Length of time it takes for the page to start and finish hydrating (in ms)
- `Next.js-route-change-to-render`: Length of time it takes for a page to start rendering after a route change (in ms)
- `Next.js-render`: Length of time it takes for a page to finish render after a route change (in ms)

You can handle all the results of these metrics using the `custom` label:

```
export function reportWebVitals(metric) {
  if (metric.label === 'custom') {
    console.log(metric) // The metric object ({ id, name, startTime, value, label }) is logged to the console
  }
}
```

There's also the option of handling each of the metrics separately:

```
export function reportWebVitals(metric) {
  switch (metric.name) {
    case 'Next.js-hydration':
      // handle hydration results
      break
    case 'Next.js-route-change-to-render':
      // handle route-change to render results
      break
    case 'Next.js-render':
      // handle render results
      break
    default:
      break
  }
}
```

These metrics work in all browsers that support the User Timing API.

Sending results to external systems

With the relay function, you can send results to any endpoint to measure and track real user performance on your site. For example:

```
export function reportWebVitals(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 })
  }
}
```

> **Note**: If you use Google Analytics, using the `id` value can allow you to construct metric distributions manually (to calculate percentiles, etc.)
>
> ```
> export function reportWebVitals({ id, name, label, value }) {
>   // 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', name, {
>     event_category:
>       label === 'web-vital' ? 'Web Vitals' : 'Next.js custom metric',
>     value: Math.round(name === 'CLS' ? value * 1000 : value), // values must be integers
>     event_label: id, // id unique to current page load
>     non_interaction: true, // avoids affecting bounce rate.
>   })
> }
> ```
>
> Read more about sending results to Google Analytics.

Web Vitals Attribution

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.

TypeScript

If you are using TypeScript, you can use the built-in type `NextWebVitalsMetric`:

```
// pages/_app.tsx

import type { AppProps, NextWebVitalsMetric } from 'next/app'

function MyApp({ Component, pageProps }: AppProps) {
  return <Component {...pageProps} />
}

export function reportWebVitals(metric: NextWebVitalsMetric) {
  console.log(metric)
}

export default MyApp
```

---

description: Learn how to use Middleware to run code before a request is completed.

Middleware

▼ **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
`v12.0.9`	Enforce absolute URLs in Edge Runtime (PR)
`v12.0.0`	Middleware (Beta) added

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, so you can personalize static files and pages. Common examples of Middleware would be authentication, A/B testing, localized pages, bot protection, and more. Regarding localized pages, you can start with i18n routing and implement Middleware for more advanced use cases.

> **Note**: If you were using Middleware prior to `12.2`, please see the upgrade guide.

Using Middleware

To begin using Middleware, follow the steps below:

1. Install the latest version of Next.js:

```
npm install next@latest
```

2. Create a `middleware.ts` (or `.js`) file at the same level as your `pages` (in the root or `src` directory)
3. Export a middleware function from the `middleware.ts` file:

```
// 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('/about-2', request.url))
}

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

Matching Paths

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

1. `headers` from `next.config.js`
2. `redirects` from `next.config.js`
3. Middleware (`rewrites`, `redirects`, etc.)
4. `beforeFiles` (`rewrites`) from `next.config.js`
5. Filesystem routes (`public/`, `_next/static/`, Pages, etc.)
6. `afterFiles` (`rewrites`) from `next.config.js`
7. Dynamic Routes (`/blog/[slug]`)
8. `fallback` (`rewrites`) from `next.config.js`

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

1. Custom matcher config
2. Conditional statements

Matcher

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

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

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

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

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

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

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

Configured matchers:

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

Read more details on path-to-regexp documentation.

> **Note**: 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:

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

Using Cookies

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

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

```
// 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')?.value
  console.log(cookie) // => 'fast'
  const allCookies = request.cookies.getAll()
  console.log(allCookies) // => [{ name: 'nextjs', value: 'fast' }]

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

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

  // Setting cookies on the response using the `ResponseCookies` API
  const response = NextResponse.next()
  response.cookies.set('vercel', 'fast')
  response.cookies.set({
    name: 'vercel',
    value: 'fast',
    path: '/test',
  })
  cookie = response.cookies.get('vercel')
  console.log(cookie) // => { name: 'vercel', value: 'fast', Path: '/test' }
  // 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.ts

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
}
```

> **Note**: 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.ts
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
}
```

Related

Edge Runtime Learn more about the supported Web APIs available.
Middleware API Reference Learn more about the supported APIs for Middleware.
Edge API Routes Build high performance APIs in Next.js.

---

description: Configure module path aliases that allow you to remap certain import paths.

Absolute Imports and Module path aliases

▶ **Examples**

Next.js automatically supports the `tsconfig.json` and `jsconfig.json` `"paths"` and `"baseUrl"` options since Next.js 9.4.

> **Note**: `jsconfig.json` can be used when you don't use TypeScript

> **Note**: you need to restart dev server to reflect modifications done in `tsconfig.json` / `jsconfig.json`

These options allow you to configure module aliases, for example a common pattern is aliasing certain directories to use absolute paths.

One useful feature of these options is that they integrate automatically into certain editors, for example vscode.

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.js
export default function Button() {
  return <button>Click me</button>
}

// pages/index.js
import Button from 'components/button'

export default function HomePage() {
  return (
    <>
      <h1>Hello World</h1>
      <Button />
    </>
  )
}
```

While `baseUrl` is useful you might want to add other aliases that don't match 1 on 1. For this TypeScript has the `"paths"` option.

Using `"paths"` allows you to configure module aliases. For example `@/components/*` to `components/*`.

An example of this configuration:

```
// tsconfig.json or jsconfig.json
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/components/*": ["components/*"]
    }
  }
}

// components/button.js
export default function Button() {
  return <button>Click me</button>
}

// pages/index.js
import Button from '@/components/button'

export default function HomePage() {
  return (
    <>
      <h1>Hello World</h1>
      <Button />
    </>
  )
}
```

Multi Zones

▼ **Examples**

- With Zones

A zone is a single deployment of a Next.js app. You can have multiple zones and merge them as a single app.

For example, let's say you have the following apps:

- An app for serving `/blog/**`
- Another app for serving all other pages

With multi zones support, you can merge both these apps into a single one allowing your customers to browse it using a single URL, but you can develop and deploy both apps independently.

How to define a zone

There are no zone related APIs. You only need to do the following:

- Make sure to keep only the pages you need in your app, meaning that an app can't have pages from another app, if app `A` has `/blog` then app `B` shouldn't have it too.
- Make sure to configure a basePath to avoid conflicts with pages and static files.

How to merge zones

You can merge zones using `rewrites` in one of the apps or any HTTP proxy.

For Next.js on Vercel applications, you can use a monorepo to deploy both apps with a single `git push`.

---

description: Learn how to instrument your Next.js app with OpenTelemetry.

> **Note**: This feature is experimental, you need to explicitly opt-in by providing `experimental.instrumentationHook = true;` in your `next.config.js`.

OpenTelemetry in Next.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.

> **Note**: 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 you need to customize your setup.

Using `@vercel/otel`

To get started, you must install `@vercel/otel`:

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

Next, create a custom `instrumentation.ts` file in the root of the project:

```
// instrumentation.ts
import { registerOTel } from '@vercel/otel'

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

> **Note**: We have created a basic with-opentelemetry example that you can use.

Manual OpenTelemetry configuration

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

Firstly you need to install OpenTelemetry packages:

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

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

```
  }
}

// instrumentation.node.ts
import { NodeSDK } from '@opentelemetry/sdk-node'
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-grpc'
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-http` instead of `@opentelemetry/exporter-trace-otlp-grpc` or you can specify more resource attributes.

Testing your instrumentation

You need an OpenTelemetry collector with a compatible backend to test OpenTelemetry traces locally. We recommend using our OpenTelemetry dev environment.

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

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

Deployment

Using OpenTelemetry Collector

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

Deploying on Vercel

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

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

Self-hosting

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

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

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

Custom Exporters

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

Custom Spans

You can add a custom span with OpenTelemetry APIs.

```
npm install @opentelemetry/api
```

The following example demonstrates a function that fetches GitHub stars and adds a custom `fetchGithubStars` span to track the fetch request's result:

```
import { trace } from '@opentelemetry/api'

export async function fetchGithubStars() {
  return await trace
    .getTracer('nextjs-example')
    .startActiveSpan('fetchGithubStars', async (span) => {
      try {
        return await getValue()
      } finally {
        span.end()
      }
    })
}
```

The `register` function will execute before your code runs in a new environment. You can start creating new spans, and they should be correctly added to the exported trace.

Default Spans in Next.js

Next.js automatically instruments several spans for you to provide useful insights into your application's performance.

Attributes on spans follow OpenTelemetry semantic conventions. We also add some custom attributes under the `next` namespace:

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

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

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

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

Attributes:

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

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

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

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

Attributes:

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

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

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

This span represents the fetch request executed in your code.

Attributes:

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

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

- `next.span_type`: `AppRouteRouteHandlers.runHandler`.

This span represents the execution of an API route handler in the app router.

Attributes:

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

`getServerSideProps [next.route]`

- `next.span_type`: `Render.getServerSideProps`.

This span represents the execution of `getServerSideProps` for a specific route.

Attributes:

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

`getStaticProps [next.route]`

- `next.span_type`: `Render.getStaticProps`.

This span represents the execution of `getStaticProps` for a specific route.

Attributes:

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

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

- `next.span_type`: `Render.renderDocument`.

This span represents the process of rendering the document for a specific route.

Attributes:

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

`generateMetadata [next.page]`

- `next.span_type`: `ResolveMetadata.generateMetadata`.

This span represents the process of generating metadata for a specific page (a single route can have multiple of these spans).

Attributes:

- `next.span_name`
- `next.span_type`
- `next.page`

---

description: Next.js automatically traces which files are needed by each page to allow for easy deployment of your application. Learn how it works here.

Output File Tracing

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`:

```
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.

> **Note**: If `distDir` is configured, the standalone build can be found at `<distDir>/standalone`. Additionally, static files should be copied to `standalone/<distDir>/static` if they should be served by the integrated server.
>
> **Note**: `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.
>
> **Note**: If your project uses Image Optimization with the default `loader`, you must install `sharp` as a dependency:

```
npm i sharp
```

```
yarn add sharp
```

```
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
    },
  },
}
```

---

description: Next.js has the preview mode for statically generated pages. You can learn how it works here.

Preview Mode

> This document is for Next.js versions 9.3 and up. If you're using older versions of Next.js, refer to our previous documentation.

▼ Examples

- WordPress Example (Demo)
- DatoCMS Example (Demo)
- TakeShape Example (Demo)
- Sanity Example (Demo)
- Prismic Example (Demo)
- Contentful Example (Demo)
- Strapi Example (Demo)
- Prepr Example (Demo)
- Agility CMS Example (Demo)
- Cosmic Example (Demo)
- ButterCMS Example (Demo)
- Storyblok Example (Demo)
- GraphCMS Example (Demo)
- Kontent Example (Demo)
- Umbraco Heartcore Example (Demo)
- Plasmic Example (Demo)
- Enterspeed Example (Demo)
- Makeswift Example (Demo)

In the Pages documentation and the Data Fetching documentation, we talked about how to pre-render a page at build time (**Static Generation**) using `getStaticProps` and `getStaticPaths`.

Static Generation is useful when your pages fetch data from a headless CMS. However, it's not ideal when you're writing a draft on your headless CMS and want to **preview** the draft immediately on your page. You'd want Next.js to render these pages at **request time** instead of build time and fetch the draft content instead of the published content. You'd want Next.js to bypass Static Generation only for this specific case.

Next.js has a feature called **Preview Mode** which solves this problem. Here are instructions on how to use it.

Step 1. Create and access a preview API route

> Take a look at the API Routes documentation first if you're not familiar with Next.js API Routes.

First, create a **preview API route**. It can have any name - e.g. `pages/api/preview.js` (or `.ts` if using TypeScript).

In this API route, you need to call `setPreviewData` on the response object. The argument for `setPreviewData` should be an object, and this can be used by `getStaticProps` (more on this later). For now, we'll use `{}`.

```
export default function handler(req, res) {
  // ...
  res.setPreviewData({})
  // ...
}
```

`res.setPreviewData` sets some **cookies** on the browser which turns on the preview mode. Any requests to Next.js containing these cookies will be considered as the **preview mode**, and the behavior for statically generated pages will change (more on this later).

You can test this manually by creating an API route like below and accessing it from your browser manually:

```
// A simple example for testing it manually from your browser.
// If this is located at pages/api/preview.js, then
// open /api/preview from your browser.
export default function handler(req, res) {
```

```
  res.setPreviewData({})
  res.end('Preview mode enabled')
}
```

If you use your browser's developer tools, you'll notice that the `__prerender_bypass` and `__next_preview_data` cookies will be set on this request.

Securely accessing it from your Headless CMS

In practice, you'd want to call this API route *securely* from your headless CMS. The specific steps will vary depending on which headless CMS you're using, but here are some common steps you could take.

These steps assume that the headless CMS you're using supports setting **custom preview URLs**. If it doesn't, you can still use this method to secure your preview URLs, but you'll need to construct and access the preview URL manually.

First, you should create a **secret token string** using a token generator of your choice. This secret will only be known by your Next.js app and your headless CMS. This secret prevents people who don't have access to your CMS from accessing preview URLs.

Second, if your headless CMS supports setting custom preview URLs, specify the following as the preview URL. (This assumes that your preview API route is located at `pages/api/preview.js`.)

```
https://<your-site>/api/preview?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 preview. If you want to preview `/posts/foo`, then you should use `&slug=/posts/foo`.

Your headless CMS might allow you to include a variable in the preview URL so that `<path>` can be set dynamically based on the CMS's data like so: `&slug=/posts/{entry.fields.slug}`

Finally, in the preview API route:

- Check that the secret matches and that the `slug` parameter exists (if not, the request should fail).
-
- Call `res.setPreviewData`.
- Then redirect the browser to the path specified by `slug`. (The following example uses a 307 redirect).

```
export default async (req, res) => {
  // Check the secret and next parameters
  // This secret should only be known to this API route and the CMS
  if (req.query.secret !== 'MY_SECRET_TOKEN' || !req.query.slug) {
    return res.status(401).json({ message: 'Invalid token' })
  }

  // 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(req.query.slug)

  // If the slug doesn't exist prevent preview mode from being enabled
  if (!post) {
    return res.status(401).json({ message: 'Invalid slug' })
  }

  // Enable Preview Mode by setting the cookies
  res.setPreviewData({})

  // Redirect to the path from the fetched post
  // We don't redirect to req.query.slug as that might lead to open redirect vulnerabilities
  res.redirect(post.slug)
}
```

If it succeeds, then the browser will be redirected to the path you want to preview with the preview mode cookies being set.

Step 2. Update `getStaticProps`

The next step is to update `getStaticProps` to support the preview mode.

If you request a page which has `getStaticProps` with the preview mode cookies set (via `res.setPreviewData`), then `getStaticProps` will be called at **request time** (instead of at build time).

Furthermore, it will be called with a `context` object where:

- `context.preview` will be `true`.
- `context.previewData` will be the same as the argument used for `setPreviewData`.

```
export async function getStaticProps(context) {
  // If you request this page with the preview mode cookies set:
  //
  // - context.preview will be true
  // - context.previewData will be the same as
  //   the argument used for `setPreviewData`.
}
```

We used `res.setPreviewData({})` in the preview API route, so `context.previewData` will be `{}`. You can use this to pass session information from the preview API route to `getStaticProps` if necessary.

If you're also using `getStaticPaths`, then `context.params` will also be available.

Fetch preview data

You can update `getStaticProps` to fetch different data based on `context.preview` and/or `context.previewData`.

For example, your headless CMS might have a different API endpoint for draft posts. If so, you can use `context.preview` to modify the API endpoint URL like below:

```
export async function getStaticProps(context) {
  // If context.preview is true, append "/preview" to the API endpoint
  // to request draft data instead of published data. This will vary
  // based on which headless CMS you're using.
  const res = await fetch(`https://.../${context.preview ? 'preview' : ''}`)
  // ...
}
```

That's it! If you access the preview API route (with `secret` and `slug`) from your headless CMS or manually, you should now be able to see the preview content. And if you update your draft without publishing, you should be able to preview the draft.

```
# Set this as the preview URL on your headless CMS or access manually,
# and you should be able to see the preview.
https://<your-site>/api/preview?secret=<token>&slug=<path>
```

More Details

> **Note**: during rendering `next/router` exposes an `isPreview` flag, see the router object docs for more info.

Specify the Preview Mode duration

`setPreviewData` takes an optional second parameter which should be an options object. It accepts the following keys:

- `maxAge`: Specifies the number (in seconds) for the preview session to last for.
- `path`: Specifies the path the cookie should be applied under. Defaults to `/` enabling preview mode for all paths.

```
setPreviewData(data, {
  maxAge: 60 * 60, // The preview mode cookies expire in 1 hour
  path: '/about', // The preview mode cookies apply to paths with /about
})
```

Clear the Preview Mode cookies

By default, no expiration date is set for Preview Mode cookies, so the preview session ends when the browser is closed.

To clear the Preview Mode cookies manually, create an API route that calls `clearPreviewData()`:

```
// pages/api/clear-preview-mode-cookies.js

export default function handler(req, res) {
  res.clearPreviewData({})
}
```

Then, send a request to `/api/clear-preview-mode-cookies` to invoke the API Route. If calling this route using `next/link`, you must pass `prefetch={false}` to prevent calling `clearPreviewData` during link prefetching.

If a path was specified in the `setPreviewData` call, you must pass the same path to `clearPreviewData`:

```
// pages/api/clear-preview-mode-cookies.js

export default function handler(req, res) {
  const { path } = req.query

  res.clearPreviewData({ path })
}
```

`previewData` size limits

You can pass an object to `setPreviewData` and have it be available in `getStaticProps`. However, because the data will be stored in a cookie, there's a size limitation. Currently, preview data is limited to 2KB.

Works with `getServerSideProps`

The preview mode works on `getServerSideProps` as well. It will also be available on the `context` object containing `preview` and `previewData`.

Works with API Routes

API Routes will have access to `preview` and `previewData` under the request object. For example:

```
export default function myApiRoute(req, res) {
  const isPreview = req.preview
  const previewData = req.previewData
  // ...
}
```

Unique per `next build`

Both the bypass cookie value and the private key for encrypting the `previewData` change when `next build` is completed. This ensures that the bypass cookie can't be guessed.

> **Note**: To test Preview Mode locally over HTTP your browser will need to allow third-party cookies and local storage access.

Learn more

The following pages might also be useful.

Data Fetching: Learn more about data fetching in Next.js.
API Routes: Learn more about API routes in Next.js.
Environment Variables: Learn more about environment variables in Next.js.

React 18

Next.js 13 requires using React 18, unlocking:

- Streaming SSR
- React Server Components
- Edge and Node.js Runtimes
- New APIs like `startTransition` and more.

Streaming SSR

In Next.js 13, you can start using the `app/` directory (beta) to take advantage of streaming server-rendering. Learn more by reading the `app/` directory (beta) documentation:

- Streaming and Suspense
- Instant Loading UI

Deploy the `app/` directory example to try Streaming SSR.

React Server Components

In Next.js 13, you can start using the `app/` directory (beta) which use Server Components by default. Learn more by reading the `app/` directory (beta) documentation:

- Rendering Fundamentals
- Server and Client Components

Deploy the `app/` directory example to try Server Components.

Edge and Node.js Runtimes

Next.js has two **server runtimes** where you can render parts of your application code: the **Node.js Runtime** and the **Edge Runtime**. Depending on your deployment infrastructure, both runtimes support streaming.

By default, Next.js uses the Node.js runtime. Middleware and Edge API Routes use the Edge runtime.

Learn more about the different runtimes.

React Server Components

React Server 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.

In Next.js 13, you can start using the `app/` directory (beta) which uses Server Components by default. Learn more by reading the `app/` directory (beta) documentation:

- Rendering Fundamentals
- Server and Client Components

Deploy the `app/` directory example to try Server Components.

Streaming SSR

Streaming allows you to incrementally render parts of your UI to the client.

In Next.js 13, you can start using the `app/` directory (beta) to take advantage of streaming server-rendering. Learn more by reading the `app/` directory (beta) documentation:

- Streaming and Suspense
- Instant Loading UI

Deploy the `app/` directory example to try Streaming SSR.

---

description: Learn more about the switchable runtimes (Edge and Node.js) in Next.js.

Edge and Node.js Runtimes

Next.js has two **server runtimes** where you can render parts of your application code: the **Node.js Runtime** and the **Edge Runtime**. Depending on your deployment infrastructure, both runtimes support streaming.

By default, Next.js uses the Node.js runtime. Middleware and Edge API Routes use the Edge runtime.

Page Runtime Option

On each page, you can optionally export a `runtime` config set to either `'nodejs'` or `'experimental-edge'`:

```
// pages/index.js
export default function Index () { ... }

export function getServerSideProps() { ... }

export const config = {
  runtime: 'experimental-edge',
}
```

Runtime Differences

	Node (Server)	Node (Serverless)	Edge
Name	`nodejs`	`nodejs`	`edge` or `experimental-edge` if using Next.js Rendering
Cold Boot	/	~250ms	Instant
HTTP Streaming	Yes	Yes	Yes
IO	All	All	`fetch`
Scalability	/	High	Highest
Security	Normal	High	High
Latency	Normal	Low	Lowest
Code Size	/	50 MB	4 MB
NPM Packages	All	All	A smaller subset

Next.js' default runtime configuration is good for most use cases, but there are still many reasons to change to one runtime over the other one.

For example, for API routes that rely on native Node.js APIs, they need to run with the Node.js Runtime. However, if an API only uses something like cookie-based authentication, using Middleware and the Edge Runtime will be a better choice due to its lower latency as well as better scalability.

Edge API Routes

Edge API Routes enable you to build high performance APIs with Next.js using the Edge Runtime.

```
export const config = {
  runtime: 'edge',
}

export default (req) => new Response('Hello world!')
```

Related

Edge Runtime Learn more about the supported Web APIs available.
Middleware API Reference Learn more about the supported APIs for Middleware.
Edge API Routes Build high performance APIs in Next.js.

---

description: Improve the security of your Next.js application by adding HTTP response headers.

Security Headers

To improve the security of your application, you can use `headers` in `next.config.js` to apply HTTP response headers to all routes in your application.

```
// next.config.js

// You can choose which headers to add to the list
// after learning more below.
const securityHeaders = []

module.exports = {
  async headers() {
    return [
      {
        // Apply these headers to all routes in your application.
        source: '/:path*',
        headers: securityHeaders,
      },
    ]
  },
}
```

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()
}
```

References

- MDN
- Varun Naik
- Scott Helme
- Mozilla Observatory

Related

For more information, we recommend the following sections:

Headers: Add custom HTTP headers to your Next.js app.

---

description: Enables browser source map generation during the production build.

Source Maps

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.

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.

Caveats

- Adding source maps can increase `next build` time
- Increases memory usage during `next build`

---

description: Save pages under the `src` directory as an alternative to the root `pages` directory.

`src` Directory

Pages can also be added under `src/pages` as an alternative to the root `pages` directory.

The `src` directory is very common in many apps and Next.js supports it by default.

Caveats

- `src/pages` will be ignored if `pages` is present in the root directory
- Config files like `next.config.js` and `tsconfig.json`, as well as environment variables, should be inside the root directory, moving them to `src` won't work. Same goes for the `public` directory

Related

For more information on what to do next, we recommend the following sections:

Pages: Learn more about what pages are in Next.js

---

description: Export your Next.js app to static HTML, and run it standalone without the need of a Node.js server.

Static HTML Export

▶ **Examples**

Next.js can be used to generate static applications, including using React in the browser without the need for a Node.js server.

The core of Next.js has been designed to enable starting as a static site (or Single-Page Application), if desired, and later upgrade to use powerful, dynamic features that require a server. For example, Incremental Static Regeneration, Internationalized Routing, and more.

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.

Usage

Update your `next.config.js` file to include `output: 'export'` like the following:

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

module.exports = nextConfig
```

Then run `next build` to generate an `out` directory containing the HTML/CSS/JS static assets.

You can utilize `getStaticProps` and `getStaticPaths` to generate an HTML file for each page in your `pages` directory (or more for dynamic routes).

If you want to change the output directory, you can configure `distDir` like the following:

```
/**
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  output: 'export',
  distDir: 'dist',
}

module.exports = nextConfig
```

In this example, `next build` will generate a `dist` directory containing the HTML/CSS/JS static assets.

Learn more about Setting a custom build directory.

If you want to change the output directory structure to always include a trailing slash, you can configure `trailingSlash` like the following:

```
/**
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  output: 'export',
  trailingSlash: true,
}

module.exports = nextConfig
```

This will change links so that `href="/about"` will instead be `href="/about/"`. It will also change the output so that `out/about.html` will instead emit `out/about/index.html`.

Learn more about Trailing Slash.

Supported Features

The majority of core Next.js features needed to build a static site are supported, including:

- Dynamic Routes when using `getStaticPaths`
- Prefetching with `next/link`
- Preloading JavaScript
- Dynamic Imports
- Any styling options (e.g. CSS Modules, styled-jsx)
- Client-side data fetching
- `getStaticProps`
- `getStaticPaths`
- Image Optimization using a custom loader

Unsupported Features

Features that require a Node.js server, or dynamic logic that cannot be computed during the build process, are not supported:

- Image Optimization (default loader)
- Internationalized Routing
- API Routes
- Rewrites
- Redirects
- Headers
- Middleware
- Incremental Static Regeneration
- `getStaticPaths` with `fallback: true`
- `getStaticPaths` with `fallback: 'blocking'`
- `getServerSideProps`

`getInitialProps`

It's possible to use the `getInitialProps` API instead of `getStaticProps`, but it comes with a few caveats:

- `getInitialProps` cannot be used alongside `getStaticProps` or `getStaticPaths` on any given page. If you have dynamic routes, instead of using `getStaticPaths` you'll need to configure the `exportPathMap` parameter in your `next.config.js` file to let the exporter know which HTML files it should output.
- When `getInitialProps` is called during export, the `req` and `res` fields of its `context` parameter will be empty objects, since during export there is no server running.
- `getInitialProps` **will be called on every client-side navigation**, if you'd like to only fetch data at build-time, switch to `getStaticProps`.
- `getInitialProps` should fetch from an API and cannot use Node.js-specific libraries or the file system like `getStaticProps` can.

We recommend migrating towards `getStaticProps` over `getInitialProps` whenever possible.

next export

> **Warning**: "next export" is deprecated since Next.js 13.3 in favor of "output: 'export'" configuration.

In versions of Next.js prior to 13.3, there was no configuration option in next.config.js and instead there was a separate command for `next export`.

This could be used by updating your `package.json` file to include `next export` like the following:

```
"scripts": {
  "build": "next build && next export"
}
```

Running `npm run build` will generate an `out` directory.

`next export` builds an HTML version of your app. During `next build`, `getStaticProps` and `getStaticPaths` will generate an HTML file for each page in your `pages` directory (or more for dynamic routes). Then, `next export` will copy the already exported files into the correct directory. `getInitialProps` will generate the HTML files during `next export` instead of `next build`.

> **Warning**: Using `exportPathMap` is deprecated and is overridden by `getStaticPaths` inside `pages`. We recommend not to use them together.

---

description: Turbopack, an incremental bundler built with Rust, can be used with Next.js 13 using the --turbo flag for faster local development.

Turbopack (beta)

Turbopack is an incremental bundler optimized for JavaScript and TypeScript, written in Rust, and built into Next.js 13.

On large applications, Turbopack updates 700x faster than Webpack.

Usage

Turbopack can be used in Next.js 13 in both the `pages` and `app` directories:

1. Create a Next.js 13 project with Turbopack

```
npx create-next-app@latest --example with-turbopack
```

2. Start the Next.js development server (with Turbopack)

```
next dev --turbo
```

Supported Features

To learn more about the currently supported features for Turbopack, view the documentation.

---

description: Learn how to use @next/mdx in your Next.js project.

Using MDX with Next.js

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 components within your content, helping you to bring your pages to life.

Next.js supports MDX through a number of different means, this page will outline some of the ways you can begin integrating MDX into your Next.js project.

Why use MDX?

Authoring in markdown is an intuitive way to write content, its terse syntax, once adopted, can enable you to write content that is both readable and maintainable. Because you can use `HTML` elements in your markdown, you can also get creative when styling your markdown pages.

However, because markdown is essentially static content, you can't create *dynamic* content based on user interactivity. Where MDX shines is in its ability to let you create and use your React components directly in the markup. This opens up a wide range of possibilities when composing your sites pages with interactivity in mind.

MDX Plugins

Internally MDX uses remark and rehype. Remark is a markdown processor powered by a plugins ecosystem. This plugin ecosystem lets you parse code, transform `HTML` elements, change syntax, extract frontmatter, and more. Using remark-gfm to enable GitHub flavored markdown (GFM) is a popular option.

Rehype is an `HTML` processor, also powered by a plugin ecosystem. Similar to remark, these plugins let you manipulate, sanitize, compile and configure all types of data, elements and content.

To use a plugin from either remark or rehype, you will need to add it to the MDX packages config.

`@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` directory.

Setup `@next/mdx` in Next.js

The following steps outline how to setup `@next/mdx` in your Next.js project:

1. Install the required packages:

```
npm install @next/mdx @mdx-js/loader @mdx-js/react
```

2. Require the package and configure to support top level `.mdx` pages. The following adds the `options` object key allowing you to pass in any plugins:

```
// next.config.js

const withMDX = require('@next/mdx')({
  extension: /\.mdx?$/,
  options: {
    // If you use remark-gfm, you'll need to use next.config.mjs
    // as the package is ESM only
    // https://github.com/remarkjs/remark-gfm#install
    remarkPlugins: [],
    rehypePlugins: [],
    // If you use `MDXProvider`, uncomment the following line.
    // providerImportSource: "@mdx-js/react",
  },
})

/** @type {import('next').NextConfig} */
const nextConfig = {
  // Configure pageExtensions to include md and mdx
  pageExtensions: ['ts', 'tsx', 'js', 'jsx', 'md', 'mdx'],
  // Optionally, add any other Next.js config below
  reactStrictMode: true,
}

// Merge MDX config with Next.js config
module.exports = withMDX(nextConfig)
```

3. Create a new MDX page within the `/pages` directory:

```
- /pages
  - my-mdx-page.mdx
- package.json
```

Using Components, Layouts and Custom Elements

You can now import a React component directly inside your MDX page:

```
import { MyComponent } from 'my-components'

# My MDX page

This is a list in markdown:

- One
- Two
- Three

Checkout my React component:

<MyComponent/>
```

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
```

Layouts

To add a layout to your MDX page, create a new component and import it into the MDX page. Then you can wrap the MDX page with your layout component:

```
import { MyComponent, MyLayoutComponent } from 'my-components'

export const meta = {
author: 'Rich Haines'
}

# My MDX Page with a Layout

This is a list in markdown:
```

```
- One
- Two
- Three

Checkout my React component:

<MyComponent/>

export default ({ children }) => <MyLayoutComponent meta={meta}>{children}</MyLayoutComponent>
```

Custom Elements

One of the pleasant aspects of using markdown, is that it maps to native `HTML` elements, making writing fast, and intuitive:

```
# H1 heading

## H2 heading

This is a list in markdown:

- One
- Two
- Three
```

The above generates the following `HTML`:

```
<h1>H1 heading</h1>

<h2>H2 heading</h2>

<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.

Using 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

---

description: The Next.js CLI allows you to start, build, and export your application. Learn more about it here.

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:

```
npx next -h
```

(npx comes with npm 5.2+ and higher)

The output should look like this:

```
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:

```
NODE_OPTIONS='--throw-deprecation' next
NODE_OPTIONS='-r esm' next
NODE_OPTIONS='--inspect' next
```

> **Note**: 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:

```
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:

```
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:

```
npx next dev -p 4000
```

Or using the `PORT` environment variable:

```
PORT=4000 npx next dev
```

> **Note**: `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:

```
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:

```
npx next start -p 4000
```

Or using the `PORT` environment variable:

```
PORT=4000 npx next start
```

> **Note**: `PORT` cannot be set in `.env` as booting up the HTTP server happens before any other code is initialized.

> **Note**: `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:

```
npx next start --keepAliveTimeout 70000
```

Lint

`next lint` runs ESLint for all files in the `pages/`, `app` (only if the experimental `appDir` feature is enabled), `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:

```
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.

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:

```
next info
```

will give you information like this example:

```
    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.

---

description: Create Next.js apps in one command with create-next-app.

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. To get started, use the following command:

Interactive

You can create a new project interactively by running:

```
npx create-next-app@latest
# or
yarn create next-app
# or
pnpm create next-app
```

You will be asked for the name of your project, and then whether you want to create a TypeScript project:

```
✓ Would you like to use TypeScript with this project? … No / Yes
```

Select **Yes** to install the necessary types/dependencies and create a new TS project.

Non-interactive

You can also pass command line arguments to set up a new project non-interactively. See `create-next-app --help`:

```
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)

  --no-tailwind

    Initialize without Tailwind CSS config.

  --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 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`).
- **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.

Related

For more information on what to do next, we recommend the following sections:

Pages: Learn more about what pages are in Next.js.
CSS Support: Use the built-in CSS support to add custom styles to your app.
CLI: Learn more about the Next.js CLI.

---

description: Enable Server-Side Rendering in a page and do initial data population with `getInitialProps`.

getInitialProps

> **Note**: Next.js 13 introduces the `app/` directory (beta). This new directory has support for colocated data fetching at the component level, using the new React `use` hook and an extended `fetch` Web API.
>
> Learn more about incrementally adopting `app/`.

`getInitialProps` enables server-side rendering in a page and allows you to do **initial data population**, it means sending the page with the data already populated from the server. This is especially useful for SEO.

`getInitialProps` will disable Automatic Static Optimization.

`getInitialProps` is an `async` function that can be added to any page as a `static method`. Take a look at the following example:

```
function Page({ stars }) {
  return <div>Next stars: {stars}</div>
}

Page.getInitialProps = async (ctx) => {
  const res = await fetch('https://api.github.com/repos/vercel/next.js')
  const json = await res.json()
  return { stars: json.stargazers_count }
}

export default Page
```

Or using a class component:

```
import React from 'react'

class Page extends React.Component {
  static async getInitialProps(ctx) {
    const res = await fetch('https://api.github.com/repos/vercel/next.js')
    const json = await res.json()
```

```
    return { stars: json.stargazers_count }
  }

  render() {
    return <div>Next stars: {this.props.stars}</div>
  }
}

export default Page
```

`getInitialProps` is used to asynchronously fetch some data, which then populates `props`.

Data returned from `getInitialProps` is serialized when server rendering, similar to what `JSON.stringify` does. Make sure the returned object from `getInitialProps` is a plain `Object` and not using `Date`, `Map` or `Set`.

For the initial page load, `getInitialProps` will run on the server only. `getInitialProps` will then run on the client when navigating to a different route via the `next/link` component or by using `next/router`. However, if `getInitialProps` is used in a custom `_app.js`, and the page being navigated to implements `getServerSideProps`, then `getInitialProps` will run on the server.

Context Object

`getInitialProps` receives a single argument called `context`, it's an object with the following properties:

- `pathname` - Current route. That is the path of the page in `/pages`
- `query` - Query string section of URL parsed as an object
- `asPath` - `String` of the actual path (including the query) shown in the browser
- `req` - HTTP request object (server only)
- `res` - HTTP response object (server only)
- `err` - Error object if any error is encountered during the rendering

Caveats

- `getInitialProps` can **not** be used in children components, only in the default export of every page
- If you are using server-side only modules inside `getInitialProps`, make sure to import them properly, otherwise it'll slow down your app

  Note that irrespective of rendering type, any `props` will be passed to the page component and can be viewed on the client-side in the initial HTML. This is to allow the page to be hydrated correctly. Make sure that you don't pass any sensitive information that shouldn't be available on the client in `props`.

TypeScript

If you're using TypeScript, you can use the `NextPage` type for function components:

```
import { NextPage } from 'next'

interface Props {
  userAgent?: string;
}

const Page: NextPage<Props> = ({ userAgent }) => (
  <main>Your user agent: {userAgent}</main>
)

Page.getInitialProps = async ({ req }) => {
  const userAgent = req ? req.headers['user-agent'] : navigator.userAgent
  return { userAgent }
}

export default Page
```

And for `React.Component`, you can use `NextPageContext`:

```
import React from 'react'
import { NextPageContext } from 'next'

interface Props {
  userAgent?: string;
}

export default class Page extends React.Component<Props> {
  static async getInitialProps({ req }: NextPageContext) {
    const userAgent = req ? req.headers['user-agent'] : navigator.userAgent
    return { userAgent }
  }

  render() {
    const { userAgent } = this.props
    return <main>Your user agent: {userAgent}</main>
  }
}
```

Related

For more information on what to do next, we recommend the following sections:

Data Fetching: Learn more about data fetching in Next.js.

---

description: API reference for `getServerSideProps`. Learn how to fetch data on each request with Next.js.

getServerSideProps

▶ **Version History**

> **Note**: Next.js 13 introduces the `app/` directory (beta). This new directory has support for colocated data fetching at the component level, using the new React `use` hook and an extended `fetch` Web API.
>
> Learn more about incrementally adopting `app/`.

When exporting a function called `getServerSideProps` (Server-Side Rendering) from a page, Next.js will pre-render this page on each request using the data returned by `getServerSideProps`. This is useful if you want to fetch data that changes often, and have the page update to show the most current data.

```
export async function getServerSideProps(context) {
  return {
    props: {}, // will be passed to the page component as props
  }
}
```

You can import modules in top-level scope for use in `getServerSideProps`. Imports used will **not be bundled for the client-side**. This means you can write **server-side code directly in `getServerSideProps`**, including fetching data from your database.

Context parameter

The `context` parameter is an object containing the following keys:

- `params`: If this page uses a dynamic route, `params` contains the route parameters. If the page name is `[id].js` , then `params` will look like `{ id: ... }`.
- `req`: The `HTTP` IncomingMessage object, with an additional `cookies` prop, which is an object with string keys mapping to string values of cookies.
- `res`: The `HTTP` response object.
- `query`: An object representing the query string, including dynamic route parameters.
- `preview`: `preview` is `true` if the page is in the Preview Mode and `false` otherwise.
- `previewData`: The preview data set by `setPreviewData`.
- `resolvedUrl`: A normalized version of the request `URL` that strips the `_next/data` prefix for client transitions and includes original query values.
- `locale` contains the active locale (if enabled).
- `locales` contains all supported locales (if enabled).
- `defaultLocale` contains the configured default locale (if enabled).

getServerSideProps return values

The `getServerSideProps` function should return an object with **any one of the following** properties:

`props`

The `props` object is a key-value pair, where each value is received by the page component. It should be a serializable object so that any props passed, could be serialized with `JSON.stringify`.

```
export async function getServerSideProps(context) {
  return {
    props: { message: `Next.js is awesome` }, // will be passed to the page component as props
  }
}
```

`notFound`

The `notFound` boolean allows the page to return a `404` status and 404 Page. With `notFound: true`, the page will return a `404` even if there was a successfully generated page before. This is meant to support use cases like user-generated content getting removed by its author.

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

  if (!data) {
    return {
      notFound: true,
    }
  }

  return {
    props: { data }, // will be passed to the page component as props
  }
}
```

`redirect`

The `redirect` object allows redirecting to internal and external resources. It should match the shape of `{ destination: string, permanent: boolean }`. In some rare cases, you might need to assign a custom status code for older `HTTP` clients to properly redirect. In these cases, you can use the `statusCode` property instead of the `permanent` property, but not both.

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

  if (!data) {
    return {
      redirect: {
        destination: '/',
        permanent: false,
      },
    }
  }

  return {
    props: {}, // will be passed to the page component as props
  }
}
```

getServerSideProps with TypeScript

The type of `getServerSideProps` can be specified using `GetServerSideProps` from `next`:

```
import { GetServerSideProps } from 'next'

type Data = { ... }

export const getServerSideProps: GetServerSideProps<{ data: Data }> = async (context) => {
  const res = await fetch('https://.../data')
  const data: Data = await res.json()

  return {
    props: {
```

```
      data,
    },
  }
}
```

If you want to get inferred typings for your props, you can use `InferGetServerSidePropsType<typeof getServerSideProps>`:

```
import { InferGetServerSidePropsType } from 'next'
import { GetServerSideProps } from 'next'

type Data = { ... }

export const getServerSideProps: GetServerSideProps<{ data: Data }> = async () => {
  const res = await fetch('https://.../data')
  const data: Data = await res.json()

  return {
    props: {
      data,
    },
  }
}

function Page({ data }: InferGetServerSidePropsType<typeof getServerSideProps>) {
  // will resolve data to type Data
}

export default Page
```

Implicit typing for `getServerSideProps` will also work properly:

```
import { InferGetServerSidePropsType } from 'next'

type Data = { ... }

export const getServerSideProps = async () => {
  const res = await fetch('https://.../data')
  const data: Data = await res.json()

  return {
    props: {
      data,
    },
  }
}

function Page({ data }: InferGetServerSidePropsType<typeof getServerSideProps>) {
  // will resolve data to type Data
}

export default Page
```

Related

For more information on what to do next, we recommend the following sections:

Data Fetching: Learn more about data fetching in Next.js.

---

description: API reference for `getStaticPaths`. Learn how to fetch data and generate static pages with `getStaticPaths`.

getStaticPaths

▶ **Version History**

When exporting a function called `getStaticPaths` from a page that uses Dynamic Routes, Next.js will statically pre-render all the paths specified by `getStaticPaths`.

```
export async function getStaticPaths() {
  return {
    paths: [
      { params: { ... } } // See the "paths" section below
    ],
    fallback: true, false or "blocking" // See the "fallback" section below
  };
}
```

getStaticPaths return values

The `getStaticPaths` function should return an object with the following **required** properties:

`paths`

The `paths` key determines which paths will be pre-rendered. For example, suppose that you have a page that uses Dynamic Routes named `pages/posts/[id].js`. If you export `getStaticPaths` from this page and return the following for `paths`:

```
return {
  paths: [
    { params: { id: '1' }},
    {
      params: { id: '2' },
      // with i18n configured the locale for the path can be returned as well
      locale: "en",
    },
  ],
  fallback: ...
}
```

Then, Next.js will statically generate `/posts/1` and `/posts/2` during `next build` using the page component in `pages/posts/[id].js`.

The value for each `params` object must match the parameters used in the page name:

- If the page name is `pages/posts/[postId]/[commentId]`, then `params` should contain `postId` and `commentId`.
- If the page name uses catch-all routes like `pages/[...slug]`, then `params` should contain `slug` (which is an array). If this array is `['hello', 'world']`, then Next.js will statically generate the page at `/hello/world`.
- If the page uses an optional catch-all route, use `null`, `[]`, `undefined` or `false` to render the root-most route. For example, if you supply `slug: false` for `pages/[[...slug]]`, Next.js will statically generate the page `/`.

The `params` strings are **case-sensitive** and ideally should be normalized to ensure the paths are generated correctly. For example, if `WoRLD` is returned for a param it will only match if `WoRLD` is the actual path visited, not `world` or `World`.

Separate of the `params` object a `locale` field can be returned when i18n is configured, which configures the locale for the path being generated.

`fallback: false`

If `fallback` is `false`, then any paths not returned by `getStaticPaths` will result in a **404 page**.

When `next build` is run, Next.js will check if `getStaticPaths` returned `fallback: false`, it will then build **only** the paths returned by `getStaticPaths`. This option is useful if you have a small number of paths to create, or new page data is not added often. If you find that you need to add more paths, and you have `fallback: false`, you will need to run `next build` again so that the new paths can be generated.

The following example pre-renders one blog post per page called `pages/posts/[id].js`. The list of blog posts will be fetched from a CMS and returned by `getStaticPaths`. Then, for each page, it fetches the post data from a CMS using `getStaticProps`.

```
// pages/posts/[id].js

function Post({ post }) {
  // Render post...
}

// This function gets called at build time
export async function getStaticPaths() {
  // Call an external API endpoint to get posts
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to pre-render based on posts
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

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

// This also gets called at build time
export async function getStaticProps({ params }) {
  // params contains the post `id`.
  // If the route is like /posts/1, then params.id is 1
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  // Pass post data to the page via props
  return { props: { post } }
}

export default Post
```

`fallback: true`

▶ **Examples**

If `fallback` is `true`, then the behavior of `getStaticProps` changes in the following ways:

- The paths returned from `getStaticPaths` will be rendered to `HTML` at build time by `getStaticProps`.
- The paths that have not been generated at build time will **not** result in a 404 page. Instead, Next.js will serve a "fallback" version of the page on the first request to such a path. Web crawlers, such as Google, won't be served a fallback and instead the path will behave as in `fallback: 'blocking'`.
- When a page with `fallback: true` is navigated to through `next/link` or `next/router` (client-side) Next.js will *not* serve a fallback and instead the page will behave as `fallback: 'blocking'`.
- In the background, Next.js will statically generate the requested path `HTML` and `JSON`. This includes running `getStaticProps`.
- When complete, the browser receives the `JSON` for the generated path. This will be used to automatically render the page with the required props. From the user's perspective, the page will be swapped from the fallback page to the full page.
- At the same time, Next.js adds this path to the list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, like other pages pre-rendered at build time.

  Note: `fallback: true` is not supported when using `output: 'export'`.

When is `fallback: true` useful?

`fallback: true` is useful if your app has a very large number of static pages that depend on data (such as a very large e-commerce site). If you want to pre-render all product pages, the builds would take a very long time.

Instead, you may statically generate a small subset of pages and use `fallback: true` for the rest. When someone requests a page that is not generated yet, the user will see the page with a loading indicator or skeleton component.

Shortly after, `getStaticProps` finishes and the page will be rendered with the requested data. From now on, everyone who requests the same page will get the statically pre-rendered page.

This ensures that users always have a fast experience while preserving fast builds and the benefits of Static Generation.

`fallback: true` will not *update* generated pages, for that take a look at Incremental Static Regeneration.

`fallback: 'blocking'`

If `fallback` is `'blocking'`, new paths not returned by `getStaticPaths` will wait for the `HTML` to be generated, identical to SSR (hence why *blocking*), and then be cached for future requests so it only happens once per path.

`getStaticProps` will behave as follows:

- The paths returned from `getStaticPaths` will be rendered to `HTML` at build time by `getStaticProps`.
- The paths that have not been generated at build time will **not** result in a 404 page. Instead, Next.js will SSR on the first request and return the generated `HTML`.

- When complete, the browser receives the `HTML` for the generated path. From the user's perspective, it will transition from "the browser is requesting the page" to "the full page is loaded". There is no flash of loading/fallback state.
- At the same time, Next.js adds this path to the list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, like other pages pre-rendered at build time.

`fallback: 'blocking'` will not *update* generated pages by default. To update generated pages, use Incremental Static Regeneration in conjunction with `fallback: 'blocking'`.

> **Note**: `fallback: 'blocking'` is not supported when using `output: 'export'`.

Fallback pages

In the "fallback" version of a page:

- The page's props will be empty.
- Using the router, you can detect if the fallback is being rendered, `router.isFallback` will be `true`.

The following example showcases using `isFallback`:

```
// pages/posts/[id].js
import { useRouter } from 'next/router'

function Post({ post }) {
  const router = useRouter()

  // If the page is not yet generated, this will be displayed
  // initially until getStaticProps() finishes running
  if (router.isFallback) {
    return <div>Loading...</div>
  }

  // Render post...
}

// This function gets called at build time
export async function getStaticPaths() {
  return {
    // Only `/posts/1` and `/posts/2` are generated at build time
    paths: [{ params: { id: '1' } }, { params: { id: '2' } }],
    // Enable statically generating additional pages
    // For example: `/posts/3`
    fallback: true,
  }
}

// This also gets called at build time
export async function getStaticProps({ params }) {
  // params contains the post `id`.
  // If the route is like /posts/1, then params.id is 1
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  // Pass post data to the page via props
  return {
    props: { post },
    // Re-generate the post at most once per second
    // if a request comes in
    revalidate: 1,
  }
}

export default Post
```

getStaticPaths with TypeScript

For TypeScript, you can use the `GetStaticPaths` type from `next`:

```
import { GetStaticPaths } from 'next'

export const getStaticPaths: GetStaticPaths = async () => {
  // ...
}
```

---

description: API reference for `getStaticProps`. Learn how to use `getStaticProps` to generate static pages with Next.js.

getStaticProps

▶ **Version History**

> **Note**: Next.js 13 introduces the `app/` directory (beta). This new directory has support for colocated data fetching at the component level, using the new React `use` hook and an extended `fetch` Web API.
>
> Learn more about incrementally adopting `app/`.

Exporting a function called `getStaticProps` will pre-render a page at build time using the props returned from the function:

```
export async function getStaticProps(context) {
  return {
    props: {}, // will be passed to the page component as props
  }
}
```

You can import modules in top-level scope for use in `getStaticProps`. Imports used will **not be bundled for the client-side**. This means you can write **server-side code directly in `getStaticProps`**, including fetching data from your database.

Context parameter

The `context` parameter is an object containing the following keys:

- `params` contains the route parameters for pages using dynamic routes. For example, if the page name is `[id].js` , then `params` will look like `{ id: ... }`. You should use this together with `getStaticPaths`, which we'll explain later.
- `preview` is `true` if the page is in the Preview Mode and `undefined` otherwise.
- `previewData` contains the preview data set by `setPreviewData`.
- `locale` contains the active locale (if enabled).
- `locales` contains all supported locales (if enabled).
- `defaultLocale` contains the configured default locale (if enabled).

getStaticProps return values

The `getStaticProps` function should return an object containing either `props`, `redirect`, or `notFound` followed by an **optional** `revalidate` property.

`props`

The `props` object is a key-value pair, where each value is received by the page component. It should be a serializable object so that any props passed, could be serialized with `JSON.stringify`.

```
export async function getStaticProps(context) {
  return {
    props: { message: `Next.js is awesome` }, // will be passed to the page component as props
  }
}
```

`revalidate`

The `revalidate` property is the amount in seconds after which a page re-generation can occur (defaults to `false` or no revalidation).

```
// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// revalidation is enabled and a new request comes in
export async function getStaticProps() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  return {
    props: {
      posts,
    },
    // Next.js will attempt to re-generate the page:
    // - When a request comes in
    // - At most once every 10 seconds
    revalidate: 10, // In seconds
  }
}
```

Learn more about Incremental Static Regeneration.

The cache status of a page leveraging ISR can be determined by reading the value of the `x-nextjs-cache` response header. The possible values are the following:

- `MISS` - the path is not in the cache (occurs at most once, on the first visit)
- `STALE` - the path is in the cache but exceeded the revalidate time so it will be updated in the background
- `HIT` - the path is in the cache and has not exceeded the revalidate time

`notFound`

The `notFound` boolean allows the page to return a `404` status and 404 Page. With `notFound: true`, the page will return a `404` even if there was a successfully generated page before. This is meant to support use cases like user-generated content getting removed by its author. Note, `notFound` follows the same `revalidate` behavior described here.

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

  if (!data) {
    return {
      notFound: true,
    }
  }

  return {
    props: { data }, // will be passed to the page component as props
  }
}
```

> **Note**: `notFound` is not needed for `fallback: false` mode as only paths returned from `getStaticPaths` will be pre-rendered.

`redirect`

The `redirect` object allows redirecting to internal or external resources. It should match the shape of `{ destination: string, permanent: boolean }`.

In some rare cases, you might need to assign a custom status code for older `HTTP` clients to properly redirect. In these cases, you can use the `statusCode` property instead of the `permanent` property, **but not both**. You can also set `basePath: false` similar to redirects in `next.config.js`.

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

  if (!data) {
    return {
      redirect: {
        destination: '/',
        permanent: false,
        // statusCode: 301
      },
    }
  }

  return {
    props: { data }, // will be passed to the page component as props
  }
}
```

If the redirects are known at build-time, they should be added in `next.config.js` instead.

Reading files: Use `process.cwd()`

Files can be read directly from the filesystem in `getStaticProps`.

In order to do so you have to get the full path to a file.

Since Next.js compiles your code into a separate directory you can't use `__dirname` as the path it returns will be different from the pages directory.

Instead you can use `process.cwd()` which gives you the directory where Next.js is being executed.

```
import { promises as fs } from 'fs'
import path from 'path'

// posts will be populated at build time by getStaticProps()
function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>
          <h3>{post.filename}</h3>
          <p>{post.content}</p>
        </li>
      ))}
    </ul>
  )
}

// This function gets called at build time on server-side.
// It won't be called on client-side, so you can even do
// direct database queries.
export async function getStaticProps() {
  const postsDirectory = path.join(process.cwd(), 'posts')
  const filenames = await fs.readdir(postsDirectory)

  const posts = filenames.map(async (filename) => {
    const filePath = path.join(postsDirectory, filename)
    const fileContents = await fs.readFile(filePath, 'utf8')

    // Generally you would parse/transform the contents
    // For example you can transform markdown to HTML here

    return {
      filename,
      content: fileContents,
    }
  })
  // By returning { props: { posts } }, the Blog component
  // will receive `posts` as a prop at build time
  return {
    props: {
      posts: await Promise.all(posts),
    },
  }
}

export default Blog
```

getStaticProps with TypeScript

The type of `getStaticProps` can be specified using `GetStaticProps` from `next`:

```
import { GetStaticProps } from 'next'

type Post = {
  author: string
  content: string
}

export const getStaticProps: GetStaticProps<{ posts: Post[] }> = async (
  context
) => {
  const res = await fetch('https://.../posts')
  const posts: Post[] = await res.json()

  return {
    props: {
      posts,
    },
  }
}
```

If you want to get inferred typings for your props, you can use `InferGetStaticPropsType<typeof getStaticProps>`:

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

type Post = {
  author: string
  content: string
}

export const getStaticProps: GetStaticProps<{ posts: Post[] }> = async () => {
  const res = await fetch('https://.../posts')
  const posts: Post[] = await res.json()

  return {
    props: {
      posts,
    },
  }
}

function Blog({ posts }: InferGetStaticPropsType<typeof getStaticProps>) {
  // will resolve posts to type Post[]
}

export default Blog
```

Implicit typing for `getStaticProps` will also work properly:

```
import { InferGetStaticPropsType } from 'next'

type Post = {
  author: string
  content: string
}

export const getStaticProps = async () => {
  const res = await fetch('https://.../posts')
  const posts: Post[] = await res.json()

  return {
    props: {
      posts,
    },
  }
}

function Blog({ posts }: InferGetStaticPropsType<typeof getStaticProps>) {
  // will resolve posts to type Post[]
}

export default Blog
```

Related

For more information on what to do next, we recommend the following sections:

Data Fetching: Learn more about data fetching in Next.js.

---

description: The Next.js Edge Runtime is based on standard Web APIs. Learn more about the supported APIs available.

Edge Runtime

The Next.js Edge Runtime is based on standard Web APIs, which is used by Middleware and Edge API Routes.

Network APIs

The Edge Runtime supports the following network APIs:

API	Description
`fetch`	Fetches a resource
`Request`	Represents an HTTP request
`Response`	Represents an HTTP response
`Headers`	Represents HTTP headers
`FetchEvent`	Represents a fetch event
`addEventListener`	Adds an event listener
`FormData`	Represents form data
`File`	Represents a file
`Blob`	Represents a blob
`URLSearchParams`	Represents URL search parameters

Encoding APIs

The Edge Runtime supports the following encoding APIs:

API	Description
`TextEncoder`	Encodes a string into a Uint8Array
`TextDecoder`	Decodes a Uint8Array into a string
`atob`	Decodes a base-64 encoded string
`btoa`	Encodes a string in base-64

Stream APIs

The Edge Runtime supports the following stream APIs:

API	Description
`ReadableStream`	Represents a readable stream
`WritableStream`	Represents a writable stream
`WritableStreamDefaultWriter`	Represents a writer of a WritableStream
`TransformStream`	Represents a transform stream
`ReadableStreamDefaultReader`	Represents a reader of a ReadableStream
`ReadableStreamBYOBReader`	Represents a reader of a ReadableStream

Crypto APIs

The Edge Runtime supports the following crypto APIs:

API	Description
`crypto`	Provides access to the cryptographic functionality of the platform
`SubtleCrypto`	Provides access to common cryptographic primitives, like hashing, signing, encryption or decryption
`CryptoKey`	Represents a cryptographic key

API	Description

Web Standard APIs

The Edge Runtime supports the following web standard APIs:

API	Description
AbortController	Allows you to abort one or more DOM requests as and when desired
DOMException	Represents an error that occurs in the DOM
structuredClone	Creates a deep copy of a value
URLPattern	Represents a URL pattern
Array	Represents an array of values
ArrayBuffer	Represents a generic, fixed-length raw binary data buffer
Atomics	Provides atomic operations as static methods
BigInt	Represents a whole number with arbitrary precision
BigInt64Array	Represents a typed array of 64-bit signed integers
BigUint64Array	Represents a typed array of 64-bit unsigned integers
Boolean	Represents a logical entity and can have two values: `true` and `false`
clearInterval	Cancels a timed, repeating action which was previously established by a call to `setInterval()`
clearTimeout	Cancels a timed, repeating action which was previously established by a call to `setTimeout()`
console	Provides access to the browser's debugging console
DataView	Represents a generic view of an `ArrayBuffer`
Date	Represents a single moment in time in a platform-independent format
decodeURI	Decodes a Uniform Resource Identifier (URI) previously created by `encodeURI` or by a similar routine
decodeURIComponent	Decodes a Uniform Resource Identifier (URI) component previously created by `encodeURIComponent` or by a similar routine
encodeURI	Encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character
encodeURIComponent	Encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character
Error	Represents an error when trying to execute a statement or accessing a property
EvalError	Represents an error that occurs regarding the global function `eval()`
Float32Array	Represents a typed array of 32-bit floating point numbers
Float64Array	Represents a typed array of 64-bit floating point numbers
Function	Represents a function
Infinity	Represents the mathematical Infinity value
Int8Array	Represents a typed array of 8-bit signed integers
Int16Array	Represents a typed array of 16-bit signed integers
Int32Array	Represents a typed array of 32-bit signed integers
Intl	Provides access to internationalization and localization functionality
isFinite	Determines whether a value is a finite number
isNaN	Determines whether a value is `NaN` or not
JSON	Provides functionality to convert JavaScript values to and from the JSON format
Map	Represents a collection of values, where each value may occur only once
Math	Provides access to mathematical functions and constants
Number	Represents a numeric value
Object	Represents the object that is the base of all JavaScript objects
parseFloat	Parses a string argument and returns a floating point number
parseInt	Parses a string argument and returns an integer of the specified radix
Promise	Represents the eventual completion (or failure) of an asynchronous operation, and its resulting value
Proxy	Represents an object that is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc)
RangeError	Represents an error when a value is not in the set or range of allowed values
ReferenceError	Represents an error when a non-existent variable is referenced
Reflect	Provides methods for interceptable JavaScript operations
RegExp	Represents a regular expression, allowing you to match combinations of characters
Set	Represents a collection of values, where each value may occur only once
setInterval	Repeatedly calls a function, with a fixed time delay between each call
setTimeout	Calls a function or evaluates an expression after a specified number of milliseconds
SharedArrayBuffer	Represents a generic, fixed-length raw binary data buffer
String	Represents a sequence of characters
Symbol	Represents a unique and immutable data type that is used as the key of an object property
SyntaxError	Represents an error when trying to interpret syntactically invalid code
TypeError	Represents an error when a value is not of the expected type
Uint8Array	Represents a typed array of 8-bit unsigned integers
Uint8ClampedArray	Represents a typed array of 8-bit unsigned integers clamped to 0-255
Uint32Array	Represents a typed array of 32-bit unsigned integers
URIError	Represents an error when a global URI handling function was used in a wrong way
URL	Represents an object providing static methods used for creating object URLs
URLSearchParams	Represents a collection of key/value pairs
WeakMap	Represents a collection of key/value pairs in which the keys are weakly referenced
WeakSet	Represents a collection of objects in which each object may occur only once
WebAssembly	Provides access to WebAssembly

Environment Variables

You can use `process.env` to access Environment Variables for both `next dev` and `next build`.

Running `console.log` on `process.env` **will not** show all your Environment Variables. You have to access the variables directly as shown below:

```
console.log(process.env)
// { NEXT_RUNTIME: 'edge' }
console.log(process.env.TEST_VARIABLE)
// value
```

Compatible Node.js Modules

The following modules can be imported with and without the `node:` prefix when using the `import` statement:

Module	Description
`async_hooks`	Manage asynchronous resources lifecycles with `AsyncLocalStorage`. Supports the WinterCG subset of APIs
`events`	Facilitate event-driven programming with custom event emitters and listeners. This API is fully supported
`buffer`	Efficiently manipulate binary data using fixed-size, raw memory allocations with `Buffer`. Every primitive compatible with `Uint8Array` accepts `Buffer` too
`assert`	Provide a set of assertion functions for verifying invariants in your code
`util`	Offer various utility functions where we include `promisify`/`callbackify` and `types`

Also, `Buffer` and `AsyncLocalStorage` are globally exposed to maximize compatibility with existing Node.js nodules.

Unsupported APIs

The Edge Runtime has some restrictions including:

- Some Node.js APIs other than the ones listed above **are not supported**. For example, you can't read or write to the filesystem
- `node_modules` *can* be used, as long as they implement ES Modules and do not use native Node.js APIs
- Calling `require` directly is **not allowed**. Use ES Modules instead

The following JavaScript language features are disabled, and **will not work:**

API	Description
`eval`	Evaluates JavaScript code represented as a string
`new Function(evalString)`	Creates a new function with the code provided as an argument
`WebAssembly.compile`	Compiles a WebAssembly module from a buffer source
`WebAssembly.instantiate`	Compiles and instantiates a WebAssembly module from a buffer source

In rare cases, your code could contain (or import) some dynamic code evaluation statements which *can not be reached at runtime* and which can not be removed by treeshaking. You can relax the check to allow specific files with your Middleware or Edge API Route exported configuration:

```
export const config = {
  runtime: 'edge', // for Edge API Routes only
  unstable_allowDynamic: [
    '/lib/utilities.js', // allows a single file
    '/node_modules/function-bind/**', // use a glob to allow anything in the function-bind 3rd party module
  ],
}
```

`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*.

Related

Middleware Run code before a request is completed.
Middleware API Reference Learn more about the supported APIs for Middleware.
Edge API Routes Build high performance APIs in Next.js.

---

description: Learn more about setting a base path in Next.js

Base Path

▶ **Version History**

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:

```
module.exports = {
  basePath: '/docs',
}
```

> **Note**: 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
```

---

description: In development mode, pages include an indicator to let you know if your new code it's being compiled. You can opt-out of it here.

Build indicator

When you edit your code, and Next.js is compiling the application, a compilation indicator appears in the bottom right corner of the page.

> **Note**: 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`:

```
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:

```
module.exports = {
  devIndicators: {
    buildActivity: false,
  },
}
```

---

description: A custom asset prefix allows you serve static assets from a CDN. Learn more about it here.

CDN Support with Asset Prefix

> **Attention**: Deploying to Vercel automatically configures a global CDN for your Next.js project. You do not need to manually setup an Asset Prefix.

> **Note**: 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:

```
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
- `/_next/data/` requests for `getServerSideProps` pages. These requests will always be made against the main domain since they're not static.
- `/_next/data/` requests for `getStaticProps` pages. These requests will always be made against the main domain to support Incremental Static Generation, even if you're not using it (for consistency).

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.
Static File Serving: Serve static files, like images, in the public directory.

---

description: Next.js provides gzip compression to compress rendered content and static files, it only works with the server target. Learn more about it here.

Compression

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:

```
module.exports = {
  compress: false,
}
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Configure how Next.js will dispose and keep in memory pages created in development.

Configuring 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:

```
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,
  },
}
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Configure the build id, which is used to identify the current build in which your application is being served.

Configuring the Build ID

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:

```
module.exports = {
  generateBuildId: async () => {
    // You can, for example, get the latest git commit hash here
    return 'my-build-id'
  },
}
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Custom configuration for the next/image loader

Custom Image Loader Configuration

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure `next.config.js` with the following:

```
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

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, quality, width }) {
  const url = new URL(`https://example.com${src}`)
  url.searchParams.set('fm', 'webp')
  url.searchParams.set('w', width.toString())
  url.searchParams.set('q', quality.toString() || '75')
  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.toString() || '75')
  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.toString() || '75')
  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.toString() || '50')
  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}`
}
```

Related

Image Optimization Learn how to optimize images with the Image component.
Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Extend the default page extensions used by Next.js when resolving pages in the pages directory.

Custom Page Extensions

You can extend the default Page extensions (`.tsx`, `.ts`, `.jsx`, `.js`) used by Next.js. Inside `next.config.js`, add the `pageExtensions` config:

```
module.exports = {
  pageExtensions: ['mdx', 'md', 'jsx', 'js', 'tsx', 'ts'],
}
```

Changing these values affects *all* Next.js pages, including the following:

- `middleware.js`
- `pages/_document.js`
- `pages/_app.js`
- `pages/api/`

For example, if you reconfigure `.ts` page extensions to `.page.ts`, you would need to rename pages like `_app.page.ts`.

Including non-page files in the `pages` directory

You can colocate test files or other files used by components in the `pages` directory. Inside `next.config.js`, add the `pageExtensions` config:

```
module.exports = {
  pageExtensions: ['page.tsx', 'page.ts', 'page.jsx', 'page.js'],
}
```

Then, rename your pages to have a file extension that includes `.page` (e.g. rename `MyPage.tsx` to `MyPage.page.tsx`). Ensure you rename *all* Next.js pages, including the files mentioned above.

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Extend the default webpack config added by Next.js.

Custom Webpack Config

> **Note**: 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
- Customizing babel configuration

Some commonly asked for features are available as plugins:

- @next/mdx
- @next/bundle-analyzer

In order to extend our usage of `webpack`, you can define a function that extends its config inside `next.config.js`, like so:

```
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.

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Next.js will generate etags for every page by default. Learn more about how to disable etag generation here.

Disabling ETag Generation

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:

```
module.exports = {
  generateEtags: false,
}
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Next.js will automatically use HTTP Keep-Alive by default. Learn more about how to disable HTTP Keep-Alive here.

Disabling HTTP Keep-Alive

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:

```
module.exports = {
  httpAgentOptions: {
    keepAlive: false,
  },
}
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Next.js will add the `x-powered-by` header by default. Learn to opt-out of it here.

Disabling x-powered-by

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:

```
module.exports = {
  poweredByHeader: false,
}
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Learn to add and access environment variables in your Next.js application at build time.

Environment Variables

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**

Note: 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:

```
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>
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.
Built-in support for Environment Variables: Learn more about the new support for environment variables.

---

description: Customize the pages that will be exported as HTML files when using `next export`.

exportPathMap

> This feature is exclusive to `next export` and currently **deprecated** in favor of `getStaticPaths` with `pages` or `generateStaticParams` with `app`.

▶ **Examples**

`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:

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

> **Note**: 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:

```
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:

```
next export -o outdir
```

> **Warning**: Using `exportPathMap` is deprecated and is overridden by `getStaticPaths` inside `pages`. We recommend not to use them together.

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.
Static HTML Export: Export your Next.js app to static HTML.

---

description: Add custom HTTP headers to your Next.js app.

Headers

▼ **Examples**

- Headers

▶ **Version History**

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`:

```
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`.

```
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):

```
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`:

```
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`:

```
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:

```
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`.

```
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:

```
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.

```
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.

Related

For more information, we recommend the following sections:

Security Headers: Improve the security of your Next.js application by adding HTTP response headers.

---

description: Next.js reports ESLint errors and warnings during builds by default. Learn how to opt-out of this behavior here.

Ignoring 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:

```
module.exports = {
  eslint: {
    // Warning: This allows production builds to successfully complete even if
    // your project has ESLint errors.
    ignoreDuringBuilds: true,
  },
}
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.
ESLint: Get started with ESLint in Next.js.

---

description: Next.js reports TypeScript errors by default. Learn to opt-out of this behavior here.

Ignoring TypeScript Errors

Next.js fails your **production build** (`next build`) when TypeScript errors are present in your project.

If you'd like Next.js to dangerously produce production code even when your application has errors, you can disable the built-in type checking step.

If disabled, be sure you are running type checks as part of your build or deploy process, otherwise this can be very dangerous.

Open `next.config.js` and enable the `ignoreBuildErrors` option in the `typescript` config:

```
module.exports = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.
TypeScript: Get started with TypeScript in Next.js.

---

description: learn more about the configuration file used by Next.js to handle your application.

next.config.js

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:

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

module.exports = nextConfig
```

If you need ECMAScript modules, you can use `next.config.mjs`:

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

export default nextConfig
```

You can also use a function:

```
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:

```
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.

---

description: The complete Next.js runtime is now Strict Mode-compliant, learn how to opt-in

React Strict Mode

> **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>`.

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Add redirects to your Next.js app.

Redirects

▼ **Examples**

- Redirects

▶ **Version History**

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`:

```
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):

```
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`:

```
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`:

```
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:

```
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`.

```
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:

```
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.

```
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.

---

description: Add rewrites to your Next.js app.

Rewrites

▼ **Examples**

- Rewrites

▶ **Version History**

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`:

```
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:

```
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*`,
      },
    ],
  }
 },
}
```

Note: rewrites in `beforeFiles` do not check the filesystem/dynamic routes immediately after matching a source, they continue until all `beforeFiles` have been checked.

The order Next.js routes are checked is:

1. headers are checked/applied
2. redirects are checked/applied
3. `beforeFiles` rewrites are checked/applied
4. static files from the public directory, `_next/static` files, and non-dynamic pages are checked/served
5. `afterFiles` rewrites are checked/applied, if one of these rewrites is matched we check dynamic routes/static files after each match
6. `fallback` rewrites are checked/applied, these are applied before rendering the 404 page and after dynamic routes/all static assets have been checked. If you use fallback: true/'blocking' in `getStaticPaths`, the fallback `rewrites` defined in your `next.config.js` will *not* be run.

Rewrite parameters

When using parameters in a rewrite the parameters will be passed in the query by default when none of the parameters are used in the `destination`.

```
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.

```
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`.

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

Note: 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):

```
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`:

```
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`:

```
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:

```
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`.

```
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**

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.

```
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.

```
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

```
module.exports = {
  async rewrites() {
    return {
      fallback: [
        {
          source: '/:path*',
          destination: `https://custom-routes-proxying-endpoint.vercel.app/:path*`,
        },
      ],
    }
  },
}
```

See additional information on incremental adoption in the docs here.

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:

```
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.

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

---

description: Add client and server runtime configuration to your Next.js app.

Runtime Configuration

> **Note**: 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 in order to avoid initialization overhead.

To add runtime configuration to your app, open `next.config.js` and add the `publicRuntimeConfig` and `serverRuntimeConfig` configs:

```
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 (
    <div>
      <Image
        src={`${publicRuntimeConfig.staticFolder}/logo.png`}
        alt="logo"
        layout="fill"
      />
    </div>
  )
}

export default MyImage
```

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.
Environment Variables: Access environment variables in your Next.js application at build time.

---

description: Set a custom build directory to use instead of the default .next directory.

Setting a custom build directory

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:

```
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.

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Optimized pages include an indicator to let you know if it's being statically optimized. You can opt-out of it here.

Static Optimization Indicator

> **Note**: This indicator was removed in Next.js version 10.0.1. We recommend upgrading to the latest version of Next.js.

When a page qualifies for Automatic Static Optimization we show an indicator to let you know.

This is helpful since automatic static optimization can be very beneficial and knowing immediately in development if the page qualifies can be useful.

In some cases this indicator might not be useful, like when working on electron applications. To remove it open `next.config.js` and disable the `autoPrerender` config in `devIndicators`:

```
module.exports = {
  devIndicators: {
    autoPrerender: false,
  },
}
```

---

description: Configure Next.js pages to resolve with or without a trailing slash.

Trailing Slash

▶ **Version History**

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:

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

With this option set, urls like `/about` will redirect to `/about/`.

Related

Introduction to next.config.js: Learn more about the configuration file used by Next.js.

---

description: Configure Next.js with Turbopack-specific options

Turbopack-specific options (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:

```
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`:

```
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.

---

description: Configure Next.js to allow importing modules from external URLs (experimental).

URL Imports

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`:

```
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"
```

---

description: Enable AMP in a page, and control the way Next.js adds AMP to the page with the AMP config.

next/amp

▶ **Examples**

AMP support is one of our advanced features, you can read more about AMP here.

To enable AMP, add the following config to your page:

```
export const config = { amp: true }
```

The `amp` config accepts the following values:

- `true` - The page will be AMP-only
- `'hybrid'` - The page will have two versions, one with AMP and another one with HTML

To learn more about the `amp` config, read the sections below.

AMP First Page

Take a look at the following example:

```
export const config = { amp: true }

function About(props) {
  return <h3>My AMP About Page!</h3>
}

export default About
```

The page above is an AMP-only page, which means:

- The page has no Next.js or React client-side runtime
- The page is automatically optimized with AMP Optimizer, an optimizer that applies the same transformations as AMP caches (improves performance by up to 42%)
- The page has a user-accessible (optimized) version of the page and a search-engine indexable (unoptimized) version of the page

Hybrid AMP Page

Take a look at the following example:

```
import { useAmp } from 'next/amp'

export const config = { amp: 'hybrid' }

function About(props) {
  const isAmp = useAmp()

  return (
    <div>
      <h3>My AMP About Page!</h3>
      {isAmp ? (
        <amp-img
          width="300"
          height="300"
          src="/my-img.jpg"
          alt="a cool image"
          layout="responsive"
        />
      ) : (
        <img width="300" height="300" src="/my-img.jpg" alt="a cool image" />
      )}
    </div>
  )
}

export default About
```

The page above is a hybrid AMP page, which means:

- The page is rendered as traditional HTML (default) and AMP HTML (by adding `?amp=1` to the URL)
- The AMP version of the page only has valid optimizations applied with AMP Optimizer so that it is indexable by search-engines

The page uses `useAmp` to differentiate between modes, it's a React Hook that returns `true` if the page is using AMP, and `false` otherwise.

---

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

next/font

▶ **Version History**

This API reference will help you understand how to use `next/font/google` and `next/font/local`. For features and usage, please see the Optimizing Fonts page.

Font function arguments

For usage, review Google Fonts and Local Fonts.

Key	`font/google`	`font/local`	Data type	Required
`src`	❌	✅	String or Array of Objects	Required
`weight`	✅	✅	String or Array	Required/Optional
`style`	✅	✅	String or Array	Optional
`subsets`	✅	❌	Array of Strings	Optional
`axes`	✅	❌	Array of Strings	Optional
`display`	✅	✅	String	Optional
`preload`	✅	✅	Boolean	Optional
`fallback`	✅	✅	Array of Strings	Optional
`adjustFontFallback`	✅	✅	Boolean or String	Optional
`variable`	✅	✅	String	Optional
`declarations`	❌	✅	Array of Objects	Optional

`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`

`axes`

Some variable fonts have extra `axes` that can be included. By default, only the font weight is included to keep the file size down. The possible values of `axes` depend on the specific font.

Used in `next/font/google`

- Optional

Examples:

- `axes: ['slnt']`: An array with value `slnt` for the `Inter` variable font which has `slnt` as additional `axes` as shown here. You can find the possible `axes` values for your font by using the filter on the Google variable fonts page and looking for axes other than `wght`

`display`

The font `display` with possible string values of `'auto'`, `'block'`, `'swap'`, `'fallback'` or `'optional'` with default value of `'swap'`.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `display: 'optional'`: A string assigned to the `optional` value

`preload`

A boolean value that specifies whether the font should be preloaded or not. The default is `true`.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `preload: false`

`fallback`

The fallback font to use if the font cannot be loaded. An array of strings of fallback fonts with no default.

- Optional

Used in `next/font/google` and `next/font/local`

Examples:

- `fallback: ['system-ui', 'arial']`: An array setting the fallback fonts to `system-ui` or `arial`

`adjustFontFallback`

- For `next/font/google`: A boolean value that sets whether an automatic fallback font should be used to reduce Cumulative Layout Shift. The default is `true`.
- For `next/font/local`: A string or boolean `false` value that sets whether an automatic fallback font should be used to reduce Cumulative Layout Shift. The possible values are `'Arial'`, `'Times New Roman'` or `false`. The default is `'Arial'`.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `adjustFontFallback: false`: for ``next/font/google`
- `adjustFontFallback: 'Times New Roman'`: for `next/font/local`

`variable`

A string value to define the CSS variable name to be used if the style is applied with the CSS variable method.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `variable: '--my-font'`: The CSS variable `--my-font` is declared

`declarations`

An array of font face descriptor key-value pairs that define the generated `@font-face` further.

Used in `next/font/local`

- Optional

Examples:

- `declarations: [{ prop: 'ascent-override', value: '90%' }]`

Applying Styles

You can apply the font styles in three ways:

- `className`
- `style`
- CSS Variables

`className`

Returns a read-only CSS `className` for the loaded font to be passed to an HTML element.

```
<p className={inter.className}>Hello, Next.js!</p>
```

`style`

Returns a read-only CSS `style` object for the loaded font to be passed to an HTML element, including `style.fontFamily` to access the font family name and fallback fonts.

```
<p style={inter.style}>Hello World</p>
```

CSS Variables

If you would like to set your styles in an external style sheet and specify additional options there, use the CSS variable method.

In addition to importing the font, also import the CSS file where the CSS variable is defined and set the variable option of the font loader object as follows:

```
// pages/index.js
import { Inter } from 'next/font/google'
import styles from '../styles/component.module.css'

const inter = Inter({
  variable: '--inter-font',
})
```

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.

```
// pages/index.js
<main className={inter.variable}>
  <p className={styles.text}>Hello World</p>
</main>
```

Define the `text` selector class in the `component.module.css` CSS file as follows:

```
/* styles/component.module.css */
.text {
  font-family: var(--inter-font);
  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`.

Next Steps

Font Optimization Learn how to optimize fonts with the Font module.

---

description: Add custom elements to the `head` of your page with the built-in Head component.

next/head

▶ **Examples**

We expose a built-in component for appending elements to the `head` of the page:

```
import Head from 'next/head'

function IndexPage() {
  return (
    <div>
      <Head>
        <title>My page title</title>
      </Head>
      <p>Hello world!</p>
    </div>
  )
}

export default IndexPage
```

To avoid duplicate tags in your `head` you can use the `key` property, which will make sure the tag is only rendered once, as in the following example:

```
import Head from 'next/head'

function IndexPage() {
  return (
    <div>
      <Head>
        <title>My page title</title>
        <meta property="og:title" content="My page title" key="title" />
      </Head>
      <Head>
        <meta property="og:title" content="My new title" key="title" />
      </Head>
      <p>Hello world!</p>
    </div>
  )
}

export default IndexPage
```

In this case only the second `<meta property="og:title" />` is rendered. `meta` tags with duplicate `key` attributes are automatically handled.

> The contents of `head` get cleared upon unmounting the component, so make sure each page completely defines what it needs in `head`, without making assumptions about what other pages added.

`title`, `meta` or any other elements (e.g. `script`) need to be contained as **direct** children of the `Head` element, or wrapped into maximum one level of `<React.Fragment>` or arrays—otherwise the tags won't be correctly picked up on client-side navigations.

> We recommend using next/script in your component instead of manually creating a `<script>` in `next/head`.

---

description: Enable Image Optimization with the built-in Image component.

next/image

▼ **Examples**

- Image Component

▶ **Version History**

> **Note**: This page is the API reference for the `next/image` component. For a feature overview and usage information, please see the Image Component and Image Optimization documentation.
>
> **Note**: If you are using a version of Next.js prior to 13, you'll want to use the next/legacy/image documentation since the component was renamed.

This `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.

Known Browser Bugs

- 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"`

Required Props

The `<Image />` component requires the following properties.

src

Must be one of the following:

1. A statically imported image file, or
2. 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:

```
import Image from 'next/image'

const myLoader = ({ src, width, quality }) => {
  return `https://example.com/${src}?w=${width}&q=${quality || 75}`
}

const MyImage = (props) => {
  return (
    <Image
      loader={myLoader}
      src="me.png"
      alt="Picture of the author"
```

```
      width={500}
      height={500}
    />
  )
}
```

Alternatively, you can use the loaderFile configuration in next.config.js to configure every instance of `next/image` in your application, without passing a prop.

fill

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.

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'
const Example = () => (
  <div className="grid-element">
    <Image
      src="/example.png"
      fill
      sizes="(max-width: 768px) 100vw,
             (max-width: 1200px) 50vw,
             33vw"
    />
  </div>
)
```

This example `sizes` could have a dramatic effect on performance metrics. Without the `33vw` sizes, the image selected from the server would be 3 times as wide as it needs to be. Because file size is proportional to the square of the width, without `sizes` the user would download an image that's 9 times larger than necessary.

Learn more about `srcset` and `sizes`:

- web.dev
- mdn

quality

The quality of the optimized image, an integer between `1` and `100`, where `100` is the best quality and therefore largest file size. Defaults to `75`.

priority

When true, the image will be considered high priority and preload. Lazy loading is automatically disabled for images using `priority`.

You should use the `priority` property on any image detected as the Largest Contentful Paint (LCP) element. It may be appropriate to have multiple priority images, as different images may be the LCP element for different viewport sizes.

Should only be used when the image is visible above the fold. Defaults to `false`.

placeholder

A placeholder to use while the image is loading. Possible values are `blur` or `empty`. Defaults to `empty`.

When `blur`, the `blurDataURL` property will be used as the placeholder. If `src` is an object from a static import and the imported image is `.jpg`, `.png`, `.webp`, or `.avif`, then `blurDataURL` will be automatically populated.

For dynamic images, you must provide the `blurDataURL` property. Solutions such as 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.

Also keep in mind that the required `width` and `height` props can interact with your styling. If you use styling to modify an image's `width`, you must set the `height="auto"` style as well, or your image will be distorted.

onLoadingComplete

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.

onLoad

A callback function that is invoked when the image is loaded.

Note that the load event might occur before the placeholder is removed and the image is fully decoded.

Instead, use `onLoadingComplete`.

onError

A callback function that is invoked if the image fails to load.

loading

> **Attention**: This property is only meant for advanced usage. Switching an image to load with `eager` will normally **hurt performance**.
>
> We recommend using the `priority` property instead, which properly loads the image eagerly for nearly all use cases.

The loading behavior of the image. Defaults to `lazy`.

When `lazy`, defer loading the image until it reaches a calculated distance from the viewport.

When `eager`, load the image immediately.

Learn more

blurDataURL

A Data URL to be used as a placeholder image before the `src` image successfully loads. Only takes effect when combined with `placeholder="blur"`.

Must be a base64-encoded image. It will be enlarged and blurred, so a very small image (10px or less) is recommended. Including larger images as placeholders may harm your application performance.

Try it out:

- Demo the default `blurDataURL` prop
- Demo the shimmer effect with `blurDataURL` prop
- Demo the color effect with `blurDataURL` prop

You can also generate a solid color Data URL to match the image.

unoptimized

When true, the source image will be served as-is instead of changing quality, size, or format. Defaults to `false`.

```
import Image from 'next/image'

const UnoptimizedImage = (props) => {
  return <Image {...props} unoptimized />
}
```

Since Next.js 12.3.0, this prop can be assigned to all images by updating `next.config.js` with the following configuration:

```
module.exports = {
  images: {
    unoptimized: true,
  },
}
```

Other Props

Other properties on the `<Image />` component will be passed to the underlying `img` element with the exception of the following:

- `srcSet`. Use Device Sizes instead.
- `decoding`. It is always `"async"`.

Configuration Options

Remote Patterns

To protect your application from malicious users, configuration is required in order to use external images. This ensures that only external images from your account can be served from the Next.js Image Optimization API. These external images can be configured with the `remotePatterns` property in your `next.config.js` file, as shown below:

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

```
      {
        protocol: 'https',
        hostname: 'example.com',
        port: '',
        pathname: '/account123/**',
      },
    ],
  },
}
```

> **Note**: The example above will ensure the `src` property of `next/image` must start with `https://example.com/account123/`. Any other protocol, hostname, port, or unmatched path will respond with 400 Bad Request.

Below is another example of the `remotePatterns` property in the `next.config.js` file:

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
      },
    ],
  },
}
```

> **Note**: 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

> **Note**: We recommend using `remotePatterns` instead so you can restrict protocol and pathname.

Similar to `remotePatterns`, the `domains` configuration can be used to provide a list of allowed hostnames for external images.

However, the `domains` configuration does not support wildcard pattern matching and it cannot restrict protocol, port, or pathname.

Below is an example of the `domains` property in the `next.config.js` file:

```
module.exports = {
  images: {
    domains: ['assets.acme.com'],
  },
}
```

Loader Configuration

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure the `loaderFile` in your `next.config.js` like the following:

```
module.exports = {
  images: {
    loader: 'custom',
    loaderFile: './my/image/loader.js',
  },
}
```

This must point to a file relative to the root of your Next.js application. The file must export a default function that returns a string, for example:

```
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

Advanced

The following configuration is for advanced use cases and is usually not necessary. If you choose to configure the properties below, you will override any changes to the Next.js defaults in future updates.

Device Sizes

If you know the expected device widths of your users, you can specify a list of device width breakpoints using the `deviceSizes` property in `next.config.js`. These widths are used when the `next/image` component uses `sizes` prop to ensure the correct image is served for user's device.

If no configuration is provided, the default below is used.

```
module.exports = {
  images: {
    deviceSizes: [640, 750, 828, 1080, 1200, 1920, 2048, 3840],
  },
}
```

Image Sizes

You can specify a list of image widths using the `images.imageSizes` property in your `next.config.js` file. These widths are concatenated with the array of device sizes to form the full array of sizes used to generate image srcsets.

The reason there are two separate lists is that imageSizes is only used for images which provide a `sizes` prop, which indicates that the image is less than the full width of the screen. **Therefore, the sizes in imageSizes should all be smaller than the smallest size in deviceSizes.**

If no configuration is provided, the default below is used.

```
module.exports = {
  images: {
    imageSizes: [16, 32, 48, 64, 96, 128, 256, 384],
  },
}
```

Acceptable Formats

The default Image Optimization API will automatically detect the browser's supported image formats via the request's `Accept` header.

If the `Accept` head matches more than one of the configured formats, the first match in the array is used. Therefore, the array order matters. If there is no match (or the source image is animated), the Image Optimization API will fallback to the original image's format.

If no configuration is provided, the default below is used.

```
module.exports = {
  images: {
    formats: ['image/webp'],
  },
}
```

You can enable AVIF support with the following configuration.

```
module.exports = {
  images: {
    formats: ['image/avif', 'image/webp'],
  },
}
```

> **Note**: 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.
>
> **Note**: 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.

Minimum Cache TTL

You can configure the Time to Live (TTL) in seconds for cached optimized images. In many cases, it's better to use a Static Image Import which will automatically hash the file contents and cache the image forever with a `Cache-Control` header of `immutable`.

```
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`.

Disable Static Imports

The default behavior allows you to import static files such as `import icon from './icon.png'` and then pass that to the `src` property.

In some cases, you may wish to disable this feature if it conflicts with other plugins that expect the import to behave differently.

You can disable static image imports inside your `next.config.js`:

```
module.exports = {
  images: {
    disableStaticImages: true,
  },
}
```

Dangerously Allow SVG

The default loader does not optimize SVG images for a few reasons. First, SVG is a vector format meaning it can be resized losslessly. Second, SVG has many of the same features as HTML/CSS, which can lead to vulnerabilities without proper Content Security Policy (CSP) headers.

If you need to serve SVG images with the default Image Optimization API, you can set `dangerouslyAllowSVG` inside your `next.config.js`:

```
module.exports = {
  images: {
    dangerouslyAllowSVG: true,
    contentDispositionType: 'attachment',
    contentSecurityPolicy: "default-src 'self'; script-src 'none'; sandbox;",
  },
}
```

In addition, it is strongly recommended to also set `contentDispositionType` to force the browser to download the image, as well as `contentSecurityPolicy` to prevent scripts embedded in the image from executing.

Animated Images

The default loader will automatically bypass Image Optimization for animated images and serve the image as-is.

Auto-detection for animated files is best-effort and supports GIF, APNG, and WebP. If you want to explicitly bypass Image Optimization for a given animated image, use the unoptimized prop.

Related

For an overview of the Image component features and usage guidelines, see:

Images Learn how to display and optimize images with the Image component.

---

description: Backwards compatible Image Optimization with the Legacy Image component.

next/legacy/image

▼ **Examples**

- Legacy Image Component

▶ **Version History**

Starting with Next.js 13, the `next/image` component was rewritten to improve both the performance and developer experience. In order to provide a backwards compatible upgrade solution, the old `next/image` was renamed to `next/legacy/image`.

Comparison

Compared to `next/legacy/image`, the new `next/image` component has the following changes:

- Removes `<span>` wrapper around `<img>` in favor of native computed aspect ratio
- Adds support for canonical `style` prop
  - Removes `layout` prop in favor of `style` or `className`
  - Removes `objectFit` prop in favor of `style` or `className`
  - Removes `objectPosition` prop in favor of `style` or `className`
- Removes `IntersectionObserver` implementation in favor of native lazy loading
  - Removes `lazyBoundary` prop since there is no native equivalent
  - Removes `lazyRoot` prop since there is no native equivalent
- Removes `loader` config in favor of `loader` prop
- Changed `alt` prop from optional to required
- Changed `onLoadingComplete` callback to receive reference to `<img>` element

Required Props

The `<Image />` component requires the following properties.

src

Must be one of the following:

1. A statically imported image file, or
2. A path string. This can be either an absolute external URL, or an internal path depending on the loader prop or loader configuration.

When using an external URL, you must add it to remotePatterns in `next.config.js`.

width

The `width` property can represent either the *rendered* width or *original* width in pixels, depending on the `layout` and `sizes` properties.

When using `layout="intrinsic"` or `layout="fixed"` the `width` property represents the *rendered* width in pixels, so it will affect how large the image appears.

When using `layout="responsive"`, `layout="fill"`, the `width` property represents the *original* width in pixels, so it will only affect the aspect ratio.

The `width` property is required, except for statically imported images, or those with `layout="fill"`.

height

The `height` property can represent either the *rendered* height or *original* height in pixels, depending on the `layout` and `sizes` properties.

When using `layout="intrinsic"` or `layout="fixed"` the `height` property represents the *rendered* height in pixels, so it will affect how large the image appears.

When using `layout="responsive"`, `layout="fill"`, the `height` property represents the *original* height in pixels, so it will only affect the aspect ratio.

The `height` property is required, except for statically imported images, or those with `layout="fill"`.

Optional Props

The `<Image />` component accepts a number of additional properties beyond those which are required. This section describes the most commonly-used properties of the Image component. Find details about more rarely-used properties in the Advanced Props section.

layout

The layout behavior of the image as the viewport changes size.

`layout`	Behavior	`srcSet`	`sizes`	Has wrapper and sizer
`intrinsic` (default)	Scale *down* to fit width of container, up to image size	`1x`, `2x` (based on imageSizes)	N/A	yes
`fixed`	Sized to `width` and `height` exactly	`1x`, `2x` (based on imageSizes)	N/A	yes
`responsive`	Scale to fit width of container	`640w`, `750w`, ... `2048w`, `3840w` (based on imageSizes and deviceSizes)	`100vw`	yes
`fill`	Grow in both X and Y axes to fill container	`640w`, `750w`, ... `2048w`, `3840w` (based on imageSizes and deviceSizes)	`100vw`	yes

- Demo the `intrinsic` layout (default)
  - When `intrinsic`, the image will scale the dimensions down for smaller viewports, but maintain the original dimensions for larger viewports.
- Demo the `fixed` layout
  - When `fixed`, the image dimensions will not change as the viewport changes (no responsiveness) similar to the native `img` element.
- Demo the `responsive` layout
  - When `responsive`, the image will scale the dimensions down for smaller viewports and scale up for larger viewports.
  - Ensure the parent element uses `display: block` in their stylesheet.
- Demo the `fill` layout
  - When `fill`, the image will stretch both width and height to the dimensions of the parent element, provided the parent element is relative.
  - This is usually paired with the `objectFit` property.
  - Ensure the parent element has `position: relative` in their stylesheet.
- Demo background image

loader

A custom function used to resolve URLs. Setting the loader as a prop on the Image component overrides the default loader defined in the `images` section of `next.config.js`.

A `loader` is a function returning a URL string for the image, given the following parameters:

- `src`
- `width`
- `quality`

Here is an example of using a custom loader:

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

const myLoader = ({ src, width, quality }) => {
  return `https://example.com/${src}?w=${width}&q=${quality || 75}`
}

const MyImage = (props) => {
  return (
    <Image
      loader={myLoader}
      src="me.png"
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

sizes

A string that provides information about how wide the image will be at different breakpoints. The value of `sizes` will greatly affect performance for images using `layout="responsive"` or `layout="fill"`. It will be ignored for images using `layout="intrinsic"` or `layout="fixed"`.

The `sizes` property serves two important purposes related to image performance:

First, the value of `sizes` is used by the browser to determine which size of the image to download, from `next/legacy/image`'s automatically-generated source set. When the browser chooses, it does not yet know the size of the image on the page, so it selects an image that is the same size or larger than the viewport. The `sizes` property allows you to tell the browser that the image will actually be smaller than full screen. If you don't specify a `sizes` value, a default value of `100vw` (full screen width) is used.

Second, the `sizes` value is parsed and used to trim the values in the automatically-created source set. If the `sizes` property includes sizes such as `50vw`, which represent a percentage of the viewport width, then the source set is trimmed to not include any values which are too small to ever be necessary.

For example, if you know your styling will cause an image to be full-width on mobile devices, in a 2-column layout on tablets, and a 3-column layout on desktop displays, you should include a sizes property such as the following:

```
import Image from 'next/legacy/image'
const Example = () => (
  <div className="grid-element">
    <Image
      src="/example.png"
      layout="fill"
      sizes="(max-width: 768px) 100vw,
             (max-width: 1200px) 50vw,
             33vw"
    />
  </div>
)
```

This example `sizes` could have a dramatic effect on performance metrics. Without the `33vw` sizes, the image selected from the server would be 3 times as wide as it needs to be. Because file size is proportional to the square of the width, without `sizes` the user would download an image that's 9 times larger than necessary.

Learn more about `srcset` and `sizes`:

- web.dev
- mdn

quality

The quality of the optimized image, an integer between `1` and `100` where `100` is the best quality. Defaults to `75`.

priority

When true, the image will be considered high priority and preload. Lazy loading is automatically disabled for images using `priority`.

You should use the `priority` property on any image detected as the Largest Contentful Paint (LCP) element. It may be appropriate to have multiple priority images, as different images may be the LCP element for different viewport sizes.

Should only be used when the image is visible above the fold. Defaults to `false`.

placeholder

A placeholder to use while the image is loading. Possible values are `blur` or `empty`. Defaults to `empty`.

When `blur`, the `blurDataURL` property will be used as the placeholder. If `src` is an object from a static import and the imported image is `.jpg`, `.png`, `.webp`, or `.avif`, then `blurDataURL` will be automatically populated.

For dynamic images, you must provide the `blurDataURL` property. Solutions such as 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.

Note that all `layout` modes apply their own styles to the image element, and these automatic styles take precedence over the `style` prop.

Also keep in mind that the required `width` and `height` props can interact with your styling. If you use styling to modify an image's `width`, you must set the `height="auto"` style as well, or your image will be distorted.

objectFit

Defines how the image will fit into its parent container when using `layout="fill"`.

This value is passed to the object-fit CSS property for the `src` image.

objectPosition

Defines how the image is positioned within its parent element when using `layout="fill"`.

This value is passed to the object-position CSS property applied to the image.

onLoadingComplete

A callback function that is invoked once the image is completely loaded and the placeholder has been removed.

The `onLoadingComplete` function accepts one parameter, an object with the following properties:

- `naturalWidth`
- `naturalHeight`

loading

> **Attention**: This property is only meant for advanced usage. Switching an image to load with `eager` will normally **hurt performance**.
>
> We recommend using the `priority` property instead, which properly loads the image eagerly for nearly all use cases.

The loading behavior of the image. Defaults to `lazy`.

When `lazy`, defer loading the image until it reaches a calculated distance from the viewport.

When `eager`, load the image immediately.

Learn more

blurDataURL

A Data URL to be used as a placeholder image before the `src` image successfully loads. Only takes effect when combined with `placeholder="blur"`.

Must be a base64-encoded image. It will be enlarged and blurred, so a very small image (10px or less) is recommended. Including larger images as placeholders may harm your application performance.

Try it out:

- Demo the default `blurDataURL` prop
- Demo the shimmer effect with `blurDataURL` prop
- Demo the color effect with `blurDataURL` prop

You can also generate a solid color Data URL to match the image.

lazyBoundary

A string (with similar syntax to the margin property) that acts as the bounding box used to detect the intersection of the viewport with the image and trigger lazy loading. Defaults to `"200px"`.

If the image is nested in a scrollable parent element other than the root document, you will also need to assign the lazyRoot prop.

Learn more

lazyRoot

A React Ref pointing to the scrollable parent element. Defaults to `null` (the document viewport).

The Ref must point to a DOM element or a React component that forwards the Ref to the underlying DOM element.

Example pointing to a DOM element

```
import Image from 'next/legacy/image'
import React from 'react'

const Example = () => {
  const lazyRoot = React.useRef(null)

  return (
    <div ref={lazyRoot} style={{ overflowX: 'scroll', width: '500px' }}>
      <Image lazyRoot={lazyRoot} src="/one.jpg" width="500" height="500" />
      <Image lazyRoot={lazyRoot} src="/two.jpg" width="500" height="500" />
    </div>
  )
}
```

Example pointing to a React component

```
import Image from 'next/legacy/image'
import React from 'react'

const Container = React.forwardRef((props, ref) => {
  return (
    <div ref={ref} style={{ overflowX: 'scroll', width: '500px' }}>
      {props.children}
    </div>
  )
})

const Example = () => {
  const lazyRoot = React.useRef(null)

  return (
    <Container ref={lazyRoot}>
      <Image lazyRoot={lazyRoot} src="/one.jpg" width="500" height="500" />
      <Image lazyRoot={lazyRoot} src="/two.jpg" width="500" height="500" />
    </Container>
  )
}
```

Learn more

unoptimized

When true, the source image will be served as-is instead of changing quality, size, or format. Defaults to `false`.

```
import Image from 'next/image'

const UnoptimizedImage = (props) => {
  return <Image {...props} unoptimized />
}
```

Since Next.js 12.3.0, this prop can be assigned to all images by updating `next.config.js` with the following configuration:

```
module.exports = {
  images: {
    unoptimized: true,
  },
}
```

Other Props

Other properties on the `<Image />` component will be passed to the underlying `img` element with the exception of the following:

- `srcSet`. Use Device Sizes instead.
- `ref`. Use `onLoadingComplete` instead.
- `decoding`. It is always `"async"`.

Configuration Options

Remote Patterns

To protect your application from malicious users, configuration is required in order to use external images. This ensures that only external images from your account can be served from the Next.js Image Optimization API. These external images can be configured with the `remotePatterns` property in your `next.config.js` file, as shown below:

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'example.com',
        port: '',
        pathname: '/account123/**',
      },
    ],
  },
}
```

> **Note**: The example above will ensure the `src` property of `next/legacy/image` must start with `https://example.com/account123/`. Any other protocol, hostname, port, or unmatched path will respond with 400 Bad Request.

Below is another example of the `remotePatterns` property in the `next.config.js` file:

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
      },
    ],
  },
}
```

> **Note**: The example above will ensure the `src` property of `next/legacy/image` must start with `https://img1.example.com` or `https://me.avatar.example.com` or any number of subdomains. 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

> **Note**: We recommend using `remotePatterns` instead so you can restrict protocol and pathname.

Similar to `remotePatterns`, the `domains` configuration can be used to provide a list of allowed hostnames for external images.

However, the `domains` configuration does not support wildcard pattern matching and it cannot restrict protocol, port, or pathname.

Below is an example of the `domains` property in the `next.config.js` file:

```
module.exports = {
  images: {
    domains: ['assets.acme.com'],
  },
}
```

Loader Configuration

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure the `loader` and `path` prefix in your `next.config.js` file. This allows you to use relative URLs for the Image `src` and automatically generate the correct absolute URL for your provider.

```
module.exports = {
  images: {
    loader: 'imgix',
    path: 'https://example.com/myaccount/',
  },
}
```

Built-in Loaders

The following Image Optimization cloud providers are included:

- Default: Works automatically with `next dev`, `next start`, or a custom server
- Vercel: Works automatically when you deploy on Vercel, no configuration necessary. Learn more
- Imgix: `loader: 'imgix'`
- Cloudinary: `loader: 'cloudinary'`
- Akamai: `loader: 'akamai'`
- Custom: `loader: 'custom'` use a custom cloud provider by implementing the `loader` prop on the `next/legacy/image` component

If you need a different provider, you can use the `loader` prop with `next/legacy/image`.

> Images can not be optimized at build time using `output: 'export'`, only on-demand. To use `next/legacy/image` with `output: 'export'`, you will need to use a different loader than the default. Read more in the discussion.

> The `next/legacy/image` component's default loader uses `squoosh` because it is quick to install and suitable for a development environment. When using `next start` in your production environment, it is strongly recommended that you install `sharp` by running `yarn add sharp` in your project directory. This is not necessary for Vercel deployments, as `sharp` is installed automatically.

Advanced

The following configuration is for advanced use cases and is usually not necessary. If you choose to configure the properties below, you will override any changes to the Next.js defaults in future updates.

Device Sizes

If you know the expected device widths of your users, you can specify a list of device width breakpoints using the `deviceSizes` property in `next.config.js`. These widths are used when the `next/legacy/image` component uses `layout="responsive"` or `layout="fill"` to ensure the correct image is served for user's device.

If no configuration is provided, the default below is used.

```
module.exports = {
  images: {
    deviceSizes: [640, 750, 828, 1080, 1200, 1920, 2048, 3840],
  },
}
```

Image Sizes

You can specify a list of image widths using the `images.imageSizes` property in your `next.config.js` file. These widths are concatenated with the array of device sizes to form the full array of sizes used to generate image srcsets.

The reason there are two separate lists is that imageSizes is only used for images which provide a `sizes` prop, which indicates that the image is less than the full width of the screen. **Therefore, the sizes in imageSizes should all be smaller than the smallest size in deviceSizes.**

If no configuration is provided, the default below is used.

```
module.exports = {
  images: {
    imageSizes: [16, 32, 48, 64, 96, 128, 256, 384],
  },
}
```

Acceptable Formats

The default Image Optimization API will automatically detect the browser's supported image formats via the request's `Accept` header.

If the `Accept` head matches more than one of the configured formats, the first match in the array is used. Therefore, the array order matters. If there is no match (or the source image is animated), the Image Optimization API will fallback to the original image's format.

If no configuration is provided, the default below is used.

```
module.exports = {
  images: {
    formats: ['image/webp'],
  },
}
```

You can enable AVIF support with the following configuration.

```
module.exports = {
  images: {
    formats: ['image/avif', 'image/webp'],
  },
}
```

> **Note**: 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.

Caching Behavior

The following describes the caching algorithm for the default loader. For all other loaders, please refer to your cloud provider's documentation.

Images are optimized dynamically upon request and stored in the `<distDir>/cache/images` directory. The optimized image file will be served for subsequent requests until the expiration is reached. When a request is made that matches a cached but expired file, the expired image is served stale immediately. Then the image is optimized again in the background (also called revalidation) and saved to the cache with the new expiration date.

The cache status of an image can be determined by reading the value of the `x-nextjs-cache` (`x-vercel-cache` when deployed on Vercel) response header. The possible values are the following:

- `MISS` - the path is not in the cache (occurs at most once, on the first visit)
- `STALE` - the path is in the cache but exceeded the revalidate time so it will be updated in the background
- `HIT` - the path is in the cache and has not exceeded the revalidate time

The expiration (or rather Max Age) is defined by either the `minimumCacheTTL` configuration or the upstream image `Cache-Control` header, whichever is larger. Specifically, the `max-age` value of the `Cache-Control` header is used. If both `s-maxage` and `max-age` are found, then `s-maxage` is preferred. The `max-age` is also passed-through to any downstream clients including CDNs and browsers.

- You can configure `minimumCacheTTL` to increase the cache duration when the upstream image does not include `Cache-Control` header or the value is very low.
- You can configure `deviceSizes` and `imageSizes` to reduce the total number of possible generated images.
- You can configure formats to disable multiple formats in favor of a single image format.

Minimum Cache TTL

You can configure the Time to Live (TTL) in seconds for cached optimized images. In many cases, it's better to use a Static Image Import which will automatically hash the file contents and cache the image forever with a `Cache-Control` header of `immutable`.

```
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`.

Disable Static Imports

The default behavior allows you to import static files such as `import icon from './icon.png'` and then pass that to the `src` property.

In some cases, you may wish to disable this feature if it conflicts with other plugins that expect the import to behave differently.

You can disable static image imports inside your `next.config.js`:

```
module.exports = {
  images: {
    disableStaticImages: true,
  },
}
```

Dangerously Allow SVG

The default loader does not optimize SVG images for a few reasons. First, SVG is a vector format meaning it can be resized losslessly. Second, SVG has many of the same features as HTML/CSS, which can lead to vulnerabilities without proper Content Security Policy (CSP) headers.

If you need to serve SVG images with the default Image Optimization API, you can set `dangerouslyAllowSVG` inside your `next.config.js`:

```
module.exports = {
  images: {
    dangerouslyAllowSVG: true,
    contentDispositionType: 'attachment',
    contentSecurityPolicy: "default-src 'self'; script-src 'none'; sandbox;",
  },
}
```

In addition, it is strongly recommended to also set `contentDispositionType` to force the browser to download the image, as well as `contentSecurityPolicy` to prevent scripts embedded in the image from executing.

Animated Images

The default loader will automatically bypass Image Optimization for animated images and serve the image as-is.

Auto-detection for animated files is best-effort and supports GIF, APNG, and WebP. If you want to explicitly bypass Image Optimization for a given animated image, use the unoptimized prop.

Related

For an overview of the Image component features and usage guidelines, see:

Images Learn how to display and optimize images with the Image component.

---

description: Enable client-side transitions between routes with the built-in Link component.

next/link

▶ **Examples**

Before moving forward, we recommend you to read Routing Introduction first.

Client-side transitions between routes can be enabled via the `Link` component exported by `next/link`.

For an example, consider a `pages` directory with the following files:

- `pages/index.js`
- `pages/about.js`
- `pages/blog/[slug].js`

We can have a link to each of these pages like so:

```
import Link from 'next/link'

function Home() {
  return (
    <ul>
      <li>
        <Link href="/">Home</Link>
      </li>
      <li>
        <Link href="/about">About Us</Link>
      </li>
      <li>
        <Link href="/blog/hello-world">Blog Post</Link>
      </li>
    </ul>
  )
}

export default Home
```

`Link` accepts the following props:

- `href` - The path or URL to navigate to. This is the only required prop. It can also be an object, see example here
- `as` - Optional decorator for the path that will be shown in the browser URL bar. Before Next.js 9.5.3 this was used for dynamic routes, check our previous docs to see how it worked. When this path differs from the one provided in `href` the previous `href`/`as` behavior is used as shown in the previous docs.
- `legacyBehavior` - Changes behavior so that child must be `<a>`. Defaults to `false`.
- `passHref` - Forces `Link` to send the `href` property to its child. Defaults to `false`
- `prefetch` - Prefetch the page in the background. Defaults to `true`. Any `<Link />` that is in the viewport (initially or through scroll) will be preloaded. Prefetch can be disabled by passing `prefetch={false}`. When `prefetch` is set to `false`, prefetching will still occur on hover. Pages using Static Generation will preload `JSON` files with the data for faster page transitions. Prefetching is only enabled in production.
- `replace` - Replace the current `history` state instead of adding a new url into the stack. Defaults to `false`
- `scroll` - Scroll to the top of the page after a navigation. Defaults to `true`
- `shallow` - Update the path of the current page without rerunning `getStaticProps`, `getServerSideProps` or `getInitialProps`. Defaults to `false`
- `locale` - The active locale is automatically prepended. `locale` allows for providing a different locale. When `false` `href` has to include the locale as the default behavior is disabled.

Note: when `legacyBehavior` is not set to `true`, all `anchor` tag properties can be passed to `next/link` as well such as, `className`, `onClick`, etc.

If the route has dynamic segments

There is nothing to do when linking to a dynamic route, including catch all routes, since Next.js 9.5.3 (for older versions check our previous docs). However, it can become quite common and handy to use interpolation or an URL Object to generate the link.

For example, the dynamic route `pages/blog/[slug].js` will match the following link:

```
import Link from 'next/link'

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

```
          <Link href={`/blog/${encodeURIComponent(post.slug)}`}>
            {post.title}
          </Link>
        </li>
      ))}
    </ul>
  )
}

export default Posts
```

If the child is `<a>` tag

```
import Link from 'next/link'

function Legacy() {
  return (
    <Link href="/about" legacyBehavior>
      <a>About Us</a>
    </Link>
  )
}

export default Legacy
```

If the child is a custom component that wraps an `<a>` tag

If the child of `Link` is a custom component that wraps an `<a>` tag, you must add `passHref` to `Link`. This is necessary if you're using libraries like styled-components. Without this, the `<a>` tag will not have the `href` attribute, which hurts your site's accessibility and might affect SEO. If you're using ESLint, there is a built-in rule `next/link-passhref` to ensure correct usage of `passHref`.

```
import Link from 'next/link'
import styled from 'styled-components'

// This creates a custom component that wraps an <a> tag
const RedLink = styled.a`
  color: red;
`

function NavLink({ href, name }) {
  return (
    <Link href={href} passHref legacyBehavior>
      <RedLink>{name}</RedLink>
    </Link>
  )
}

export default NavLink
```

- If you're using emotion's JSX pragma feature (`@jsx jsx`), you must use `passHref` even if you use an `<a>` tag directly.
- The component should support `onClick` property to trigger navigation correctly

If the child is a functional component

If the child of `Link` is a functional component, in addition to using `passHref` and `legacyBehavior`, you must wrap the component in `React.forwardRef`:

```
import Link from 'next/link'

// `onClick`, `href`, and `ref` need to be passed to the DOM element
// for proper handling
const MyButton = React.forwardRef(({ onClick, href }, ref) => {
  return (
    <a href={href} onClick={onClick} ref={ref}>
      Click Me
    </a>
  )
})

function Home() {
  return (
    <Link href="/about" passHref legacyBehavior>
      <MyButton />
    </Link>
  )
}

export default Home
```

With URL Object

`Link` can also receive a URL object and it will automatically format it to create the URL string. Here's how to do it:

```
import Link from 'next/link'

function Home() {
  return (
    <ul>
      <li>
        <Link
          href={{
            pathname: '/about',
            query: { name: 'test' },
          }}
        >
          About us
        </Link>
      </li>
      <li>
        <Link
          href={{
            pathname: '/blog/[slug]',
            query: { slug: 'my-post' },
          }}
        >
```

```
          Blog Post
        </Link>
      </li>
    </ul>
  )
}

export default Home
```

The above example has a link to:

- A predefined route: `/about?name=test`
- A dynamic route: `/blog/my-post`

You can use every property as defined in the Node.js URL module documentation.

Replace the URL instead of push

The default behavior of the `Link` component is to `push` a new URL into the `history` stack. You can use the `replace` prop to prevent adding a new entry, as in the following example:

```
<Link href="/about" replace>
  About us
</Link>
```

Disable scrolling to the top of the page

The default behavior of `Link` is to scroll to the top of the page. When there is a hash defined it will scroll to the specific id, like a normal `<a>` tag. To prevent scrolling to the top / hash `scroll={false}` can be added to `Link`:

```
<Link href="/#hashid" scroll={false}>
  Disables scrolling to the top
</Link>
```

With Next.js 13 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 have 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 (inside `pages/`):

```
// pages/index.js
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>
  )
}
```

> **Note**: If you're using Dynamic Routes, you'll need to adapt your `as` and `href` props. For example, if you have a Dynamic Route like `/dashboard/[user]` that you want to present differently via middleware, you would write: `<Link href={{ pathname: '/dashboard/authed/[user]', query: { user: username } }} as="/dashboard/[user]">Profile</Link>`.

---

description: Learn more about the API of the Next.js Router, and access the router instance in your page with the useRouter hook.

next/router

> Before moving forward, we recommend you to read Routing Introduction first.

useRouter

If you want to access the `router` object inside any function component in your app, you can use the `useRouter` hook, take a look at the following example:

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

function ActiveLink({ children, href }) {
  const router = useRouter()
  const style = {
    marginRight: 10,
    color: router.asPath === href ? 'red' : 'black',
  }

  const handleClick = (e) => {
    e.preventDefault()
```

```
    router.push(href)
  }

  return (
    <a href={href} onClick={handleClick} style={style}>
      {children}
    </a>
  )
}

export default ActiveLink
```

`useRouter` is a React Hook, meaning it cannot be used with classes. You can either use withRouter or wrap your class in a function component.

`router` object

The following is the definition of the `router` object returned by both `useRouter` and `withRouter`:

- `pathname`: `String` - The path for current route file that comes after `/pages`. Therefore, `basePath`, `locale` and trailing slash (`trailingSlash: true`) are not included.
- `query`: `Object` - The query string parsed to an object, including dynamic route parameters. It will be an empty object during prerendering if the page doesn't use Server-side Rendering. Defaults to `{}`
- `asPath`: `String` - The path as shown in the browser including the search params and respecting the `trailingSlash` configuration. `basePath` and `locale` are not included.
- `isFallback`: `boolean` - Whether the current page is in fallback mode.
- `basePath`: `String` - The active basePath (if enabled).
- `locale`: `String` - The active locale (if enabled).
- `locales`: `String[]` - All supported locales (if enabled).
- `defaultLocale`: `String` - The current default locale (if enabled).
- `domainLocales`: `Array<{domain, defaultLocale, locales}>` - Any configured domain locales.
- `isReady`: `boolean` - Whether the router fields are updated client-side and ready for use. Should only be used inside of `useEffect` methods and not for conditionally rendering on the server. See related docs for use case with automatically statically optimized pages
- `isPreview`: `boolean` - Whether the application is currently in preview mode.

  Using the `asPath` field may lead to a mismatch between client and server if the page is rendered using server-side rendering or automatic static optimization. Avoid using `asPath` until the `isReady` field is `true`.

The following methods are included inside `router`:

router.push

▶ **Examples**

Handles client-side transitions, this method is useful for cases where `next/link` is not enough.

```
router.push(url, as, options)
```

- `url`: `UrlObject | String` - The URL to navigate to (see Node.JS URL module documentation for `UrlObject` properties).
- `as`: `UrlObject | String` - Optional decorator for the path that will be shown in the browser URL bar. Before Next.js 9.5.3 this was used for dynamic routes, check our previous docs to see how it worked. When this path differs from the one provided in `href` the previous `href`/`as` behavior is used as shown in the previous docs
- `options` - Optional object with the following configuration options:
  - `scroll` - Optional boolean, controls scrolling to the top of the page after navigation. Defaults to `true`
  - `shallow`: Update the path of the current page without rerunning `getStaticProps`, `getServerSideProps` or `getInitialProps`. Defaults to `false`
  - `locale` - Optional string, indicates locale of the new page

  You don't need to use `router.push` for external URLs. window.location is better suited for those cases.

Usage

Navigating to `pages/about.js`, which is a predefined route:

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

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

  return (
    <button type="button" onClick={() => router.push('/about')}>
      Click me
    </button>
  )
}
```

Navigating `pages/post/[pid].js`, which is a dynamic route:

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

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

  return (
    <button type="button" onClick={() => router.push('/post/abc')}>
      Click me
    </button>
  )
}
```

Redirecting the user to `pages/login.js`, useful for pages behind authentication:

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

// Here you would fetch and return the user
const useUser = () => ({ user: null, loading: false })

export default function Page() {
  const { user, loading } = useUser()
  const router = useRouter()

  useEffect(() => {
    if (!(user || loading)) {
      router.push('/login')
    }
```

```
  }, [user, loading])

  return <p>Redirecting...</p>
}
```

Resetting state after navigation

When navigating to the same page in Next.js, the page's state **will not** be reset by default as React does not unmount unless the parent component has changed.

```
// pages/[slug].js
import Link from 'next/link'
import { useState } from 'react'
import { useRouter } from 'next/router'

export default function Page(props) {
  const router = useRouter()
  const [count, setCount] = useState(0)
  return (
    <div>
      <h1>Page: {router.query.slug}</h1>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase count</button>
      <Link href="/one">one</Link> <Link href="/two">two</Link>
    </div>
  )
}
```

In the above example, navigating between `/one` and `/two` **will not** reset the count . The `useState` is maintained between renders because the top-level React component, `Page`, is the same.

If you do not want this behavior, you have a couple of options:

1. Manually ensure each state is updated using `useEffect`. In the above example, that could look like:

```
useEffect(() => {
  setCount(0)
}, [router.query.slug])
```

2. Use a React `key` to tell React to remount the component. To do this for all pages, you can use a custom app:

```
// pages/_app.js
import { useRouter } from 'next/router'

export default function MyApp({ Component, pageProps }) {
  const router = useRouter()
  return <Component key={router.asPath} {...pageProps} />
}
```

With URL object

You can use a URL object in the same way you can use it for `next/link`. Works for both the `url` and `as` parameters:

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

export default function ReadMore({ post }) {
  const router = useRouter()

  return (
    <button
      type="button"
      onClick={() => {
        router.push({
          pathname: '/post/[pid]',
          query: { pid: post.id },
        })
      }}
    >
      Click here to read more
    </button>
  )
}
```

router.replace

Similar to the `replace` prop in `next/link`, `router.replace` will prevent adding a new URL entry into the `history` stack.

```
router.replace(url, as, options)
```

- The API for `router.replace` is exactly the same as the API for `router.push`.

Usage

Take a look at the following example:

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

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

  return (
    <button type="button" onClick={() => router.replace('/home')}>
      Click me
    </button>
  )
}
```

router.prefetch

Prefetch pages for faster client-side transitions. This method is only useful for navigations without `next/link`, as `next/link` takes care of prefetching pages automatically.

> This is a production only feature. Next.js doesn't prefetch pages in development.

```
router.prefetch(url, as, options)
```

- `url` - The URL to prefetch, including explicit routes (e.g. `/dashboard`) and dynamic routes (e.g. `/product/[id]`)
- `as` - Optional decorator for `url`. Before Next.js 9.5.3 this was used to prefetch dynamic routes, check our previous docs to see how it worked
- `options` - Optional object with the following allowed fields:
  - `locale` - allows providing a different locale from the active one. If `false`, `url` has to include the locale as the active locale won't be used.

Usage

Let's say you have a login page, and after a login, you redirect the user to the dashboard. For that case, we can prefetch the dashboard to make a faster transition, like in the following example:

```
import { useCallback, useEffect } from 'react'
import { useRouter } from 'next/router'

export default function Login() {
  const router = useRouter()
  const handleSubmit = useCallback((e) => {
    e.preventDefault()

    fetch('/api/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        /* Form data */
      }),
    }).then((res) => {
      // Do a fast client-side transition to the already prefetched dashboard page
      if (res.ok) router.push('/dashboard')
    })
  }, [])

  useEffect(() => {
    // Prefetch the dashboard page
    router.prefetch('/dashboard')
  }, [router])

  return (
    <form onSubmit={handleSubmit}>
      {/* Form fields */}
      <button type="submit">Login</button>
    </form>
  )
}
```

router.beforePopState

In some cases (for example, if using a Custom Server), you may wish to listen to popstate and do something before the router acts on it.

```
router.beforePopState(cb)
```

- `cb` - The function to run on incoming `popstate` events. The function receives the state of the event as an object with the following props:
  - `url`: `String` - the route for the new state. This is usually the name of a `page`
  - `as`: `String` - the url that will be shown in the browser
  - `options`: `Object` - Additional options sent by router.push

If `cb` returns `false`, the Next.js router will not handle `popstate`, and you'll be responsible for handling it in that case. See Disabling file-system routing.

Usage

You could use `beforePopState` to manipulate the request, or force a SSR refresh, as in the following example:

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

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

  useEffect(() => {
    router.beforePopState(({ url, as, options }) => {
      // I only want to allow these two routes!
      if (as !== '/' && as !== '/other') {
        // Have SSR render bad routes as a 404.
        window.location.href = as
        return false
      }

      return true
    })
  }, [router])

  return <p>Welcome to the page</p>
}
```

router.back

Navigate back in history. Equivalent to clicking the browser's back button. It executes `window.history.back()`.

Usage

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

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

  return (
    <button type="button" onClick={() => router.back()}>
      Click here to go back
    </button>
  )
}
```

router.reload

Reload the current URL. Equivalent to clicking the browser's refresh button. It executes `window.location.reload()`.

Usage

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

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

  return (
    <button type="button" onClick={() => router.reload()}>
      Click here to reload
    </button>
  )
}
```

router.events

▶ **Examples**

You can listen to different events happening inside the Next.js Router. Here's a list of supported events:

- `routeChangeStart(url, { shallow })` - Fires when a route starts to change
- `routeChangeComplete(url, { shallow })` - Fires when a route changed completely
- `routeChangeError(err, url, { shallow })` - Fires when there's an error when changing routes, or a route load is cancelled
  - `err.cancelled` - Indicates if the navigation was cancelled
- `beforeHistoryChange(url, { shallow })` - Fires before changing the browser's history
- `hashChangeStart(url, { shallow })` - Fires when the hash will change but not the page
- `hashChangeComplete(url, { shallow })` - Fires when the hash has changed but not the page

> **Note**: Here `url` is the URL shown in the browser, including the `basePath`.

Usage

For example, to listen to the router event `routeChangeStart`, open or create `pages/_app.js` and subscribe to the event, like so:

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

export default function MyApp({ Component, pageProps }) {
  const router = useRouter()

  useEffect(() => {
    const handleRouteChange = (url, { shallow }) => {
      console.log(
        `App is changing to ${url} ${
          shallow ? 'with' : 'without'
        } shallow routing`
      )
    }

    router.events.on('routeChangeStart', handleRouteChange)

    // If the component is unmounted, unsubscribe
    // from the event with the `off` method:
    return () => {
      router.events.off('routeChangeStart', handleRouteChange)
    }
  }, [router])

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

> We use a Custom App (`pages/_app.js`) for this example to subscribe to the event because it's not unmounted on page navigations, but you can subscribe to router events on any component in your application.

Router events should be registered when a component mounts (useEffect or componentDidMount / componentWillUnmount) or imperatively when an event happens.

If a route load is cancelled (for example, by clicking two links rapidly in succession), `routeChangeError` will fire. And the passed `err` will contain a `cancelled` property set to `true`, as in the following example:

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

export default function MyApp({ Component, pageProps }) {
  const router = useRouter()

  useEffect(() => {
    const handleRouteChangeError = (err, url) => {
      if (err.cancelled) {
        console.log(`Route to ${url} was cancelled!`)
      }
    }

    router.events.on('routeChangeError', handleRouteChangeError)

    // If the component is unmounted, unsubscribe
    // from the event with the `off` method:
    return () => {
      router.events.off('routeChangeError', handleRouteChangeError)
    }
  }, [router])

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

Potential ESLint errors

Certain methods accessible on the `router` object return a Promise. If you have the ESLint rule, no-floating-promises enabled, consider disabling it either globally, or for the affected line.

If your application needs this rule, you should either `void` the promise – or use an `async` function, `await` the Promise, then void the function call. **This is not applicable when the method is called from inside an `onClick` handler**.

The affected methods are:

- `router.push`
- `router.replace`
- `router.prefetch`

Potential solutions

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

// Here you would fetch and return the user
const useUser = () => ({ user: null, loading: false })

export default function Page() {
  const { user, loading } = useUser()
  const router = useRouter()

  useEffect(() => {
    // disable the linting on the next line - This is the cleanest solution
    // eslint-disable-next-line no-floating-promises
    router.push('/login')

    // void the Promise returned by router.push
    if (!(user || loading)) {
      void router.push('/login')
    }
    // or use an async function, await the Promise, then void the function call
    async function handleRouteChange() {
      if (!(user || loading)) {
        await router.push('/login')
      }
    }
    void handleRouteChange()
  }, [user, loading])

  return <p>Redirecting...</p>
}
```

withRouter

If `useRouter` is not the best fit for you, `withRouter` can also add the same `router` object to any component.

Usage

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

function Page({ router }) {
  return <p>{router.pathname}</p>
}

export default withRouter(Page)
```

TypeScript

To use class components with `withRouter`, the component needs to accept a router prop:

```
import React from 'react'
import { withRouter, NextRouter } from 'next/router'

interface WithRouterProps {
  router: NextRouter
}

interface MyComponentProps extends WithRouterProps {}

class MyComponent extends React.Component<MyComponentProps> {
  render() {
    return <p>{this.props.router.pathname}</p>
  }
}

export default withRouter(MyComponent)
```

---

description: Optimize third-party script loading with the built-in Script component.

next/script

▶ **Version History**

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.

```
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:

Prop	Example	Values	Required
src	`src="http://example.com/script"`	String	Required unless inline script is used
strategy	`strategy="lazyOnload"`	String	Optional
onLoad	`onLoad={onLoadFunc}`	Function	Optional
onReady	`onReady={onReadyFunc}`	Function	Optional
onError	`onError={onErrorFunc}`	Function	Optional

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.

> **Note**: 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 `pages/_document.js` and are designed to load scripts that are needed by the entire site (i.e. the script will load when any page in the application has been loaded server-side).

This strategy should only be used for critical scripts that need to be fetched before any part of the page becomes interactive.

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

export default function Document() {
  return (
    <Html>
      <Head />
      <body>
        <Main />
        <NextScript />
        <Script
          src="https://example.com/script.js"
          strategy="beforeInteractive"
        />
      </body>
    </Html>
  )
}
```

> **Note**: 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.

```
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.

```
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`

> **Note**: 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`:

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

`worker` scripts can **only currently be used in the `pages/` directory**:

```
import Script from 'next/script'

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

`onLoad`

> **Note**: `onLoad` can't be used with `beforeInteractive` – consider using `onReady` instead. Learn more about usage of `onLoad` in the `app/` directory.

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.

```
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`

> **Note**: Learn more about usage of `onReady` in the `app/` directory.

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 everytime the component is mounted:

```
import { useRef } from 'react'
import Script from 'next/script'

export default function Page() {
  const mapRef = useRef()

  return (
    <>
      <div ref={mapRef}></div>
      <Script
        id="google-maps"
        src="https://maps.googleapis.com/maps/api/js"
        onReady={() => {
          new google.maps.Map(mapRef.current, {
            center: { lat: -34.397, lng: 150.644 },
            zoom: 8,
          })
        }}
      />
    </>
  )
}
```

`onError`

> **Note**: `onError` cannot be used with the `beforeInteractive` loading strategy. Learn more about usage of `onError` in the `app/` directory.

Sometimes it is helpful to catch when a script fails to load. These errors can be handled with the onError property:

```
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        onError={(e) => {
          console.error('Script failed to load', e)
        }}
      />
    </>
  )
}
```

Next Steps

Optimizing Scripts Learn how to optimize third-party scripts in your Next.js application.

---

description: Learn about the server-only helpers for Middleware and Edge API Routes.

next/server

`next/server` provides server-only helpers for use in Middleware and Edge API Routes.

NextRequest

The `NextRequest` object is an extension of the native `Request` interface, with the following added methods and properties:

- `cookies` - A RequestCookies instance with cookies from the `Request`. It reads/mutates the `Cookie` header of the request. See also Using cookies in Middleware.
    - `get` - 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.
    - `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.
    - `set` - A method that takes an object with properties of `CookieListItem` as defined in the W3C CookieStore API spec.
    - `delete` - A method that takes either a cookie `name` or a list of names. and removes the cookies matching the name(s). Returns `true` for deleted and `false` for undeleted cookies.
    - `has` - A method that takes a cookie `name` and returns a `boolean` based on if the cookie exists (`true`) or not (`false`).
    - `clear` - A method that takes no argument and will effectively remove the `Cookie` header.
- `nextUrl`: Includes an extended, parsed, URL object that gives you access to Next.js specific properties such as `pathname`, `basePath`, `trailingSlash` and `i18n`. Includes the following properties:
    - `basePath` (`string`)
    - `buildId` (`string || undefined`)
    - `defaultLocale` (`string || undefined`)
    - `domainLocale`
        - `defaultLocale`: (`string`)
        - `domain`: (`string`)
        - `http`: (`boolean || undefined`)
        - `locales`: (`string[] || undefined`)
    - `locale` (`string || undefined`)
    - `url` (`URL`)
- `ip`: (`string || undefined`) - Has the IP address of the `Request`. This information is provided by your hosting platform.
- `geo` - Has the geographic location from the `Request`. This information is provided by your hosting platform. Includes the following properties:
    - `city` (`string || undefined`)
    - `country` (`string || undefined`)
    - `region` (`string || undefined`)
    - `latitude` (`string || undefined`)
    - `longitude` (`string || undefined`)

You can use the `NextRequest` object as a direct replacement for the native `Request` interface, giving you more control over how you manipulate the request.

`NextRequest` can be imported from `next/server`:

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

NextFetchEvent

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

The `waitUntil()` method can be used to prolong the execution of the function if you have other background work to make.

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

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

  return NextResponse.next()
}
```

The `NextFetchEvent` object can be imported from `next/server`:

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

NextResponse

The `NextResponse` class extends the native `Response` interface, with the following:

Public Methods

Public methods are available on an instance of the `NextResponse` class. Depending on your use case, you can create an instance and assign to a variable, then access the following public methods:

- `cookies` - A ResponseCookies instance with the cookies from the `Response`. It reads/mutates the `Set-Cookie` header of the response. See also Using cookies in Middleware.
  - `get` - 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.
  - `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.
  - `set` - A method that takes an object with properties of `CookieListItem` as defined in the W3C CookieStore API spec.
  - `delete` - A method that takes either a cookie `name` or a list of names. and removes the cookies matching the name(s). Returns `true` for deleted and `false` for undeleted cookies.

Static Methods

The following static methods are available on the `NextResponse` class directly:

- `redirect()` - Returns a `NextResponse` with a redirect set
- `rewrite()` - Returns a `NextResponse` with a rewrite set
- `next()` - Returns a `NextResponse` that will continue the middleware chain

To use the methods above, **you must return the `NextResponse` object** returned. `NextResponse` can be imported from `next/server`:

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

userAgent

The `userAgent` helper allows you to interact with the user agent object from the request. It is abstracted from the native `Request` object, and is an opt in feature. It has the following properties:

- `isBot`: (`boolean`) Whether the request comes from a known bot
- `browser`
  - `name`: (`string || undefined`) The name of the browser
  - `version`: (`string || undefined`) The version of the browser, determined dynamically
- `device`
  - `model`: (`string || undefined`) The model of the device, determined dynamically
  - `type`: (`string || undefined`) The type of the browser, can be one of the following values: `console`, `mobile`, `tablet`, `smarttv`, `wearable`, `embedded`, or `undefined`
  - `vendor`: (`string || undefined`) The vendor of the device, determined dynamically
- `engine`
  - `name`: (`string || undefined`) The name of the browser engine, could be one of the following values: `Amaya`, `Blink`, `EdgeHTML`, `Flow`, `Gecko`, `Goanna`, `iCab`, `KHTML`, `Links`, `Lynx`, `NetFront`, `NetSurf`, `Presto`, `Tasman`, `Trident`, `w3m`, `WebKit` or `undefined`
  - `version`: (`string || undefined`) The version of the browser engine, determined dynamically, or `undefined`
- `os`
  - `name`: (`string || undefined`) The name of the OS, could be `undefined`
  - `version`: (`string || undefined`) The version of the OS, determined dynamically, or `undefined`
- `cpu`
  - `architecture`: (`string || undefined`) The architecture of the CPU, could be one of the following values: `68k`, `amd64`, `arm`, `arm64`, `armhf`, `avr`, `ia32`, `ia64`, `irix`, `irix64`, `mips`, `mips64`, `pa-risc`, `ppc`, `sparc`, `sparc64` or `undefined`

`userAgent` can be imported from `next/server`:

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

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

export function middleware(request: NextRequest) {
  const url = request.nextUrl
  const { device } = userAgent(request)
  const viewport = device.type === 'mobile' ? 'mobile' : 'desktop'
  url.searchParams.set('viewport', viewport)
  return NextResponse.rewrite(url)
}
```

FAQ

Why does `redirect` use 307 and 308?

When using `redirect()` you may notice that the status codes used are `307` for a temporary redirect, and `308` for a permanent redirect. While traditionally a `302` was used for a temporary redirect, and a `301` for a permanent redirect, many browsers changed the request method of the redirect, from a `POST` to `GET` request when using a `302`, regardless of the origins request method.

Taking the following example of a redirect from `/users` to `/people`, if you make a `POST` request to `/users` to create a new user, and are conforming to a `302` temporary redirect, the request method will be changed from a `POST` to a `GET` request. This doesn't make sense, as to create a new user, you should be making a `POST` request to `/people`, and not a `GET` request.

The introduction of the `307` status code means that the request method is preserved as `POST`.

- `302` - Temporary redirect, will change the request method from `POST` to `GET`
- `307` - Temporary redirect, will preserve the request method as `POST`

The `redirect()` method uses a `307` by default, instead of a `302` temporary redirect, meaning your requests will *always* be preserved as `POST` requests.

If you want to cause a `GET` response to a `POST` request, use `303`.

Learn more about HTTP Redirects.

How do I access Environment Variables?

`process.env` can be used to access Environment Variables from Edge Middleware. They are evaluated during `next build`:

Works	Does not work
`console.log(process.env.MY_ENV_VARIABLE)`	`const getEnv = name => process.env[name]`
`const { MY_ENV_VARIABLE } = process.env`	
`const { "MY-ENV-VARIABLE": MY_ENV_VARIABLE } = process.env`	

Related

Edge Runtime Learn more about the supported Web APIs available.
Middleware Run code before a request is completed.

---

description: You can add the dynamic routes used for pages to API Routes too. Learn how it works here.

Dynamic API Routes

▼ **Examples**

- Basic API Routes

API routes support dynamic routes, and follow the same file naming rules used for `pages`.

For example, the API route `pages/api/post/[pid].js` has the following code:

```
export default function handler(req, res) {
  const { pid } = req.query
  res.end(`Post: ${pid}`)
}
```

Now, a request to `/api/post/abc` will respond with the text: `Post: abc`.

Index routes and Dynamic API routes

A very common RESTful pattern is to set up routes like this:

- `GET api/posts` - gets a list of posts, probably paginated
- `GET api/posts/12345` - gets post id 12345

We can model this in two ways:

- Option 1:
  - `/api/posts.js`
  - `/api/posts/[postId].js`
- Option 2:
  - `/api/posts/index.js`
  - `/api/posts/[postId].js`

Both are equivalent. A third option of only using `/api/posts/[postId].js` is not valid because Dynamic Routes (including Catch-all routes - see below) do not have an `undefined` state and `GET api/posts` will not match `/api/posts/[postId].js` under any circumstances.

Catch all API routes

API Routes can be extended to catch all paths by adding three dots (`...`) inside the brackets. For example:

- `pages/api/post/[...slug].js` matches `/api/post/a`, but also `/api/post/a/b`, `/api/post/a/b/c` and so on.

  Note: You can use names other than `slug`, such as: `[...param]`

Matched parameters will be sent as a query parameter (`slug` in the example) to the page, and it will always be an array, so, the path `/api/post/a` will have the following `query` object:

```
{ "slug": ["a"] }
```

And in the case of `/api/post/a/b`, and any other matching path, new parameters will be added to the array, like so:

```
{ "slug": ["a", "b"] }
```

An API route for `pages/api/post/[...slug].js` could look like this:

```
export default function handler(req, res) {
  const { slug } = req.query
  res.end(`Post: ${slug.join(', ')}`)
}
```

Now, a request to `/api/post/a/b/c` will respond with the text: `Post: a, b, c`.

Optional catch all API routes

Catch all routes can be made optional by including the parameter in double brackets (`[[...slug]]`).

For example, `pages/api/post/[[...slug]].js` will match `/api/post`, `/api/post/a`, `/api/post/a/b`, and so on.

The main difference between catch all and optional catch all routes is that with optional, the route without the parameter is also matched (`/api/post` in the example above).

The `query` objects are as follows:

```
{ } // GET `/api/post` (empty object)
{ "slug": ["a"] } // `GET /api/post/a` (single-element array)
{ "slug": ["a", "b"] } // `GET /api/post/a/b` (multi-element array)
```

Caveats

- Predefined API routes take precedence over dynamic API routes, and dynamic API routes over catch all API routes. Take a look at the following examples:
  - `pages/api/post/create.js` - Will match `/api/post/create`
  - `pages/api/post/[pid].js` - Will match `/api/post/1`, `/api/post/abc`, etc. But not `/api/post/create`

- `pages/api/post/[...slug].js` - Will match `/api/post/1/2`, `/api/post/a/b/c`, etc. But not `/api/post/create`, `/api/post/abc`

Related

For more information on what to do next, we recommend the following sections:

Dynamic Routes: Learn more about the built-in dynamic routes.

---

description: Edge API Routes enable you to build high performance APIs directly inside your Next.js application.

Edge API Routes

Edge API Routes enable you to build high performance APIs with Next.js. Using the Edge Runtime, they are often faster than Node.js-based API Routes. This performance improvement does come with constraints, like not having access to native Node.js APIs. Instead, Edge API Routes are built on standard Web APIs.

Any file inside the folder `pages/api` is mapped to `/api/*` and will be treated as an API endpoint instead of a page. They are server-side only bundles and won't increase your client-side bundle size.

Examples

Basic

```
export const config = {
  runtime: 'edge',
}

export default (req) => new Response('Hello world!')
```

JSON Response

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

export const config = {
  runtime: 'edge',
}

export default async function handler(req: NextRequest) {
  return new Response(
    JSON.stringify({
      name: 'Jim Halpert',
    }),
    {
      status: 200,
      headers: {
        'content-type': 'application/json',
      },
    }
  )
}
```

Cache-Control

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

export const config = {
  runtime: 'edge',
}

export default async function handler(req: NextRequest) {
  return new Response(
    JSON.stringify({
      name: 'Jim Halpert',
    }),
    {
      status: 200,
      headers: {
        'content-type': 'application/json',
        'cache-control': 'public, s-maxage=1200, stale-while-revalidate=600',
      },
    }
  )
}
```

Query Parameters

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

export const config = {
  runtime: 'edge',
}

export default async function handler(req: NextRequest) {
  const { searchParams } = new URL(req.url)
  const email = searchParams.get('email')
  return new Response(email)
}
```

Forwarding Headers

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

export const config = {
  runtime: 'edge',
}

export default async function handler(req: NextRequest) {
  const authorization = req.cookies.get('authorization')?.value
  return fetch('https://backend-api.com/api/protected', {
    method: req.method,
```

```
    headers: {
      authorization,
    },
    redirect: 'manual',
  })
}
```

Configuring Regions (for deploying)

You may want to restrict your edge function to specific regions when deploying so that you can colocate near your data sources ensuring lower response times which can be achieved as shown.

> **Note**: This configuration is available in `v12.3.2` of Next.js and up.

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

export const config = {
  regions: ['sfo1', 'iad1'], // defaults to 'all'
}

export default async function handler(req: NextRequest) {
  const myData = await getNearbyData()
  return NextResponse.json(myData)
}
```

Differences between API Routes

Edge API Routes use the Edge Runtime, whereas API Routes use the Node.js runtime.

Edge API Routes can stream responses from the server and run *after* cached files (e.g. HTML, CSS, JavaScript) have been accessed. Server-side streaming can help improve performance with faster Time To First Byte (TTFB).

> **Note**: Using Edge Runtime with `getServerSideProps` does not give you access to the response object. If you need access to `res`, you should use the Node.js runtime by setting `runtime: 'nodejs'`.

View the supported APIs and unsupported APIs for the Edge Runtime.

---

description: Next.js supports API Routes, which allow you to build your API without leaving your Next.js app. Learn how it works here.

API Routes

▼ **Examples**

- Basic API Routes
- API Routes with GraphQL
- API Routes with REST
- API Routes with CORS

API routes provide a solution to build your **API** with Next.js.

Any file inside the folder `pages/api` is mapped to `/api/*` and will be treated as an API endpoint instead of a `page`. They are server-side only bundles and won't increase your client-side bundle size.

For example, the following API route `pages/api/user.js` returns a `json` response with a status code of `200`:

```
export default function handler(req, res) {
  res.status(200).json({ name: 'John Doe' })
}
```

> **Note**: API Routes will be affected by `pageExtensions` configuration in `next.config.js`.

For an API route to work, you need to export a function as default (a.k.a **request handler**), which then receives the following parameters:

- `req`: An instance of http.IncomingMessage, plus some pre-built middlewares
- `res`: An instance of http.ServerResponse, plus some helper functions

To handle different HTTP methods in an API route, you can use `req.method` in your request handler, like so:

```
export default function handler(req, res) {
  if (req.method === 'POST') {
    // Process a POST request
  } else {
    // Handle any other HTTP method
  }
}
```

To fetch API endpoints, take a look into any of the examples at the start of this section.

Use Cases

For new projects, you can build your entire API with API Routes. If you have an existing API, you do not need to forward calls to the API through an API Route. Some other use cases for API Routes are:

- Masking the URL of an external service (e.g. `/api/secret` instead of `https://company.com/secret-url`)
- Using Environment Variables on the server to securely access external services.

Caveats

- API Routes do not specify CORS headers, meaning they are **same-origin only** by default. You can customize such behavior by wrapping the request handler with the CORS request helpers.
- API Routes can't be used with `output: 'export'`

Related

For more information on what to do next, we recommend the following sections:

API Routes Request Helpers: learn about the built-in helpers for the request.
Response Helpers: learn about the built-in methods for the response.
TypeScript: Add TypeScript to your API Routes.

---

description: API Routes provide built-in request helpers that parse the incoming request. Learn more about them here.

API Routes Request Helpers

▼ **Examples**

- API Routes Request Helpers
- API Routes with CORS

API Routes provide built-in request helpers which parse the incoming request (`req`):

- `req.cookies` - An object containing the cookies sent by the request. Defaults to `{}`
- `req.query` - An object containing the query string. Defaults to `{}`
- `req.body` - An object containing the body parsed by `content-type`, or `null` if no body was sent

Custom config

Every API Route can export a `config` object to change the default configuration, which is the following:

```
export const config = {
  api: {
    bodyParser: {
      sizeLimit: '1mb',
    },
  },
}
```

The `api` object includes all config options available for API Routes.

`bodyParser` is automatically enabled. If you want to consume the body as a `Stream` or with `raw-body`, you can set this to `false`.

One use case for disabling the automatic `bodyParsing` is to allow you to verify the raw body of a **webhook** request, for example from GitHub.

```
export const config = {
  api: {
    bodyParser: false,
  },
}
```

`bodyParser.sizeLimit` is the maximum size allowed for the parsed body, in any format supported by bytes, like so:

```
export const config = {
  api: {
    bodyParser: {
      sizeLimit: '500kb',
    },
  },
}
```

`externalResolver` is an explicit flag that tells the server that this route is being handled by an external resolver like *express* or *connect*. Enabling this option disables warnings for unresolved requests.

```
export const config = {
  api: {
    externalResolver: true,
  },
}
```

`responseLimit` is automatically enabled, warning when an API Routes' response body is over 4MB.

If you are not using Next.js in a serverless environment, and understand the performance implications of not using a CDN or dedicated media host, you can set this limit to `false`.

```
export const config = {
  api: {
    responseLimit: false,
  },
}
```

`responseLimit` can also take the number of bytes or any string format supported by `bytes`, for example `1000`, `'500kb'` or `'3mb'`. This value will be the maximum response size before a warning is displayed. Default is 4MB. (see above)

```
export const config = {
  api: {
    responseLimit: '8mb',
  },
}
```

Extending the `req`/`res` objects with TypeScript

For better type-safety, it is not recommended to extend the `req` and `res` objects. Instead, use functions to work with them:

```
// utils/cookies.ts

import { serialize, CookieSerializeOptions } from 'cookie'
import { NextApiResponse } from 'next'

/**
 * This sets `cookie` using the `res` object
```

```
 */

export const setCookie = (
  res: NextApiResponse,
  name: string,
  value: unknown,
  options: CookieSerializeOptions = {}
) => {
  const stringValue =
    typeof value === 'object' ? 'j:' + JSON.stringify(value) : String(value)

  if (typeof options.maxAge === 'number') {
    options.expires = new Date(Date.now() + options.maxAge * 1000)
  }

  res.setHeader('Set-Cookie', serialize(name, stringValue, options))
}

// pages/api/cookies.ts

import { NextApiRequest, NextApiResponse } from 'next'
import { setCookie } from '../../utils/cookies'

const handler = (req: NextApiRequest, res: NextApiResponse) => {
  // Calling our pure function using the `res` object, it will add the `set-cookie` header
  // Add the `set-cookie` header on the main domain and expire after 30 days
  setCookie(res, 'Next.js', 'api-middleware!', { path: '/', maxAge: 2592000 })
  // Return the `set-cookie` header so we can display it in the browser and show that it works!
  res.end(res.getHeader('Set-Cookie'))
}

export default handler
```

If you can't avoid these objects from being extended, you have to create your own type to include the extra properties:

```
// pages/api/foo.ts

import { NextApiRequest, NextApiResponse } from 'next'
import { withFoo } from 'external-lib-foo'

type NextApiRequestWithFoo = NextApiRequest & {
  foo: (bar: string) => void
}

const handler = (req: NextApiRequestWithFoo, res: NextApiResponse) => {
  req.foo('bar') // we can now use `req.foo` without type errors
  res.end('ok')
}

export default withFoo(handler)
```

Keep in mind this is not safe since the code will still compile even if you remove `withFoo()` from the export.

---

description: API Routes include a set of Express.js-like methods for the response to help you creating new API endpoints. Learn how it works here.

API Routes Response Helpers

The Server Response object, (often abbreviated as `res`) includes a set of Express.js-like helper methods to improve the developer experience and increase the speed of creating new API endpoints.

The included helpers are:

- `res.status(code)` - A function to set the status code. `code` must be a valid HTTP status code
- `res.json(body)` - Sends a JSON response. `body` must be a serializable object
- `res.send(body)` - Sends the HTTP response. `body` can be a `string`, an `object` or a `Buffer`
- `res.redirect([status,] path)` - Redirects to a specified path or URL. `status` must be a valid HTTP status code. If not specified, `status` defaults to "307" "Temporary redirect".
- `res.revalidate(urlPath)` - Revalidate a page on demand using `getStaticProps`. `urlPath` must be a `string`.

Setting the status code of a response

When sending a response back to the client, you can set the status code of the response.

The following example sets the status code of the response to `200` (`OK`) and returns a `message` property with the value of `Hello from Next.js!` as a JSON response:

```
export default function handler(req, res) {
  res.status(200).json({ message: 'Hello from Next.js!' })
}
```

Sending a JSON response

When sending a response back to the client you can send a JSON response, this must be a serializable object. In a real world application you might want to let the client know the status of the request depending on the result of the requested endpoint.

The following example sends a JSON response with the status code `200` (`OK`) and the result of the async operation. It's contained in a try catch block to handle any errors that may occur, with the appropriate status code and error message caught and sent back to the client:

```
export default async function handler(req, res) {
  try {
    const result = await someAsyncOperation()
    res.status(200).json({ result })
  } catch (err) {
    res.status(500).json({ error: 'failed to load data' })
  }
}
```

Sending a HTTP response

Sending an HTTP response works the same way as when sending a JSON response. The only difference is that the response body can be a `string`, an `object` or a `Buffer`.

The following example sends a HTTP response with the status code `200` (`OK`) and the result of the async operation.

```
export default async function handler(req, res) {
  try {
    const result = await someAsyncOperation()
    res.status(200).send({ result })
  } catch (err) {
    res.status(500).send({ error: 'failed to fetch data' })
  }
}
```

Redirects to a specified path or URL

Taking a form as an example, you may want to redirect your client to a specified path or URL once they have submitted the form.

The following example redirects the client to the `/` path if the form is successfully submitted:

```
export default async function handler(req, res) {
  const { name, message } = req.body
  try {
    await handleFormInputAsync({ name, message })
    res.redirect(307, '/')
  } catch (err) {
    res.status(500).send({ error: 'failed to fetch data' })
  }
}
```

Adding TypeScript types

You can make your response handlers more type-safe by importing the `NextApiRequest` and `NextApiResponse` types from `next`, in addition to those, you can also type your response data:

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

type ResponseData = {
  message: string
}

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<ResponseData>
) {
  res.status(200).json({ message: 'Hello from Next.js!' })
}
```

> **Note**: The body of `NextApiRequest` is `any` because the client may include any payload. You should validate the type/shape of the body at runtime before using it.

To view more examples using types, check out the TypeScript documentation.

If you prefer to view your examples within a real projects structure you can checkout our examples repository:

- Basic API Routes
- API Routes with REST

---

description: Learn about authentication patterns in Next.js apps and explore a few examples.

Authentication

Authentication verifies who a user is, while authorization controls what a user can access. Next.js supports multiple authentication patterns, each designed for different use cases. This page will go through each case so that you can choose based on your constraints.

Authentication Patterns

The first step to identifying which authentication pattern you need is understanding the data-fetching strategy you want. We can then determine which authentication providers support this strategy. There are two main patterns:

- Use static generation to server-render a loading state, followed by fetching user data client-side.
- Fetch user data server-side to eliminate a flash of unauthenticated content.

Authenticating Statically Generated Pages

Next.js automatically determines that a page is static if there are no blocking data requirements. This means the absence of `getServerSideProps` and `getInitialProps` in the page. Instead, your page can render a loading state from the server, followed by fetching the user client-side.

One advantage of this pattern is it allows pages to be served from a global CDN and preloaded using `next/link`. In practice, this results in a faster TTI (Time to Interactive).

Let's look at an example for a profile page. This will initially render a loading skeleton. Once the request for a user has finished, it will show the user's name:

```
// pages/profile.js

import useUser from '../lib/useUser'
import Layout from '../components/Layout'

const Profile = () => {
  // Fetch the user client-side
  const { user } = useUser({ redirectTo: '/login' })

  // Server-render loading state
  if (!user || user.isLoggedIn === false) {
    return <Layout>Loading...</Layout>
  }

  // Once the user request finishes, show the user
  return (
    <Layout>
```

```
      <h1>Your Profile</h1>
      <pre>{JSON.stringify(user, null, 2)}</pre>
    </Layout>
  )
}

export default Profile
```

You can view this example in action. Check out the `with-iron-session` example to see how it works.

Authenticating Server-Rendered Pages

If you export an `async` function called `getServerSideProps` from a page, Next.js will pre-render this page on each request using the data returned by `getServerSideProps`.

```
export async function getServerSideProps(context) {
  return {
    props: {}, // Will be passed to the page component as props
  }
}
```

Let's transform the profile example to use server-side rendering. If there's a session, return `user` as a prop to the `Profile` component in the page. Notice there is not a loading skeleton in this example.

```
// pages/profile.js

import withSession from '../lib/session'
import Layout from '../components/Layout'

export const getServerSideProps = withSession(async function ({ req, res }) {
  const { user } = req.session

  if (!user) {
    return {
      redirect: {
        destination: '/login',
        permanent: false,
      },
    }
  }

  return {
    props: { user },
  }
})

const Profile = ({ user }) => {
  // Show the user. No loading state is required
  return (
    <Layout>
      <h1>Your Profile</h1>
      <pre>{JSON.stringify(user, null, 2)}</pre>
    </Layout>
  )
}

export default Profile
```

An advantage of this pattern is preventing a flash of unauthenticated content before redirecting. It's important to note fetching user data in `getServerSideProps` will block rendering until the request to your authentication provider resolves. To prevent creating a bottleneck and increasing your TTFB (Time to First Byte), you should ensure your authentication lookup is fast. Otherwise, consider static generation.

Authentication Providers

Now that we've discussed authentication patterns, let's look at specific providers and explore how they're used with Next.js.

Bring Your Own Database

▼ Examples

- with-iron-session
- next-auth-example

If you have an existing database with user data, you'll likely want to utilize an open-source solution that's provider agnostic.

- If you want a low-level, encrypted, and stateless session utility use `iron-session`.
- If you want a full-featured authentication system with built-in providers (Google, Facebook, GitHub…), JWT, JWE, email/password, magic links and more… use `next-auth`.

Both of these libraries support either authentication pattern. If you're interested in Passport, we also have examples for it using secure and encrypted cookies:

- with-passport
- with-passport-and-next-connect

Other Providers

To see examples with other authentication providers, check out the examples folder.

▼ Examples

- Auth0
- Clerk
- Firebase
- Magic
- Nhost
- Ory
- Supabase
- Supertokens
- Userbase

Related

For more information on what to do next, we recommend the following sections:

Pages: Learn more about pages and the different pre-rendering methods in Next.js.
Data Fetching: Learn more about data fetching in Next.js.

---

description: Next.js supports including CSS files as Global CSS or CSS Modules, using `styled-jsx` for CSS-in-JS, or any other CSS-in-JS solution! Learn more here.

Built-In CSS Support

▼ **Examples**

- Basic CSS Example
- With Tailwind CSS

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

Adding a Global Stylesheet

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

For example, consider the following stylesheet named `styles.css`:

```
body {
  font-family: 'SF Pro Text', 'SF Pro Icons', 'Helvetica Neue', 'Helvetica',
    'Arial', sans-serif;
  padding: 20px 20px 60px;
  max-width: 680px;
  margin: 0 auto;
}
```

Create a `pages/_app.js` file if not already present. Then, `import` the `styles.css` file.

```
import '../styles.css'

// This default export is required in a new `pages/_app.js` file.
export default function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

These styles (`styles.css`) will apply to all pages and components in your application. Due to the global nature of stylesheets, and to avoid conflicts, you may **only import them inside `pages/_app.js`**.

In development, expressing stylesheets this way allows your styles to be hot reloaded as you edit them—meaning you can keep application state.

In production, all CSS files will be automatically concatenated into a single minified `.css` file.

Import styles from `node_modules`

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

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

```
// pages/_app.js
import 'bootstrap/dist/css/bootstrap.css'

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

For importing CSS required by a third party component, you can do so in your component. For example:

```
// components/ExampleDialog.js
import { useState } from 'react'
import { Dialog } from '@reach/dialog'
import VisuallyHidden from '@reach/visually-hidden'
import '@reach/dialog/styles.css'

function ExampleDialog(props) {
  const [showDialog, setShowDialog] = useState(false)
  const open = () => setShowDialog(true)
  const close = () => setShowDialog(false)

  return (
    <div>
      <button onClick={open}>Open Dialog</button>
      <Dialog isOpen={showDialog} onDismiss={close}>
        <button className="close-button" onClick={close}>
          <VisuallyHidden>Close</VisuallyHidden>
          <span aria-hidden>×</span>
        </button>
        <p>Hello there. I am a dialog</p>
      </Dialog>
    </div>
  )
}
```

Adding Component-Level CSS

Next.js supports CSS Modules using the `[name].module.css` file naming convention.

CSS Modules locally scope CSS by automatically creating a unique class name. This allows you to use the same CSS class name in different files without worrying about collisions.

This behavior makes CSS Modules the ideal way to include component-level CSS. CSS Module files **can be imported anywhere in your application**.

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

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

```
/*
You do not need to worry about .error {} colliding with any other `.css` or
`.module.css` files!
*/
.error {
  color: white;
  background-color: red;
}
```

Then, create `components/Button.js`, importing and using the above CSS file:

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

export function Button() {
  return (
    <button
      type="button"
      // Note how the "error" class is accessed as a property on the imported
      // `styles` object.
      className={styles.error}
    >
      Destroy
    </button>
  )
}
```

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

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

Sass Support

Next.js allows you to import Sass using both the `.scss` and `.sass` extensions. You can use component-level Sass via CSS Modules and the `.module.scss` or `.module.sass` extension.

Before you can use Next.js' built-in Sass support, be sure to install `sass`:

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

Sass support has the same benefits and restrictions as the built-in CSS support detailed above.

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

Customizing Sass Options

If you want to configure the Sass compiler you can do so by using `sassOptions` in `next.config.js`.

For example to add `includePaths`:

```
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:

```
/* variables.module.scss */
$primary-color: #64ff00;

:export {
  primaryColor: $primary-color;
}

// pages/_app.js
import variables from '../styles/variables.module.scss'

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

CSS-in-JS

▶ **Examples**

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

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

export default HiThere
```

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

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

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

```
function HelloWorld() {
  return (
    <div>
      Hello world
      <p>scoped!</p>
      <style jsx>{`
        p {
          color: blue;
        }
        div {
          background: red;
        }
        @media (max-width: 600px) {
          div {
            background: blue;
          }
        }
      `}</style>
      <style global jsx>{`
        body {
          background: black;
        }
      `}</style>
    </div>
  )
}

export default HelloWorld
```

Please see the styled-jsx documentation for more examples.

FAQ

Does it work with JavaScript disabled?

Yes, if you disable JavaScript the CSS will still be loaded in the production build (`next start`). During development, we require JavaScript to be enabled to provide the best developer experience with Fast Refresh.

Related

For more information on what to do next, we recommend the following sections:

Customizing PostCSS Config: Extend the PostCSS config and plugins added by Next.js with your own.

---

description: 'Learn about client-side data fetching, and how to use SWR, a data fetching React hook library that handles caching, revalidation, focus tracking, refetching on interval and more.'

Client-side data fetching

Client-side data fetching is useful when your page doesn't require SEO indexing, when you don't need to pre-render your data, or when the content of your pages needs to update frequently. Unlike the server-side rendering APIs, you can use client-side data fetching at the component level.

If done at the page level, the data is fetched at runtime, and the content of the page is updated as the data changes. When used at the component level, the data is fetched at the time of the component mount, and the content of the component is updated as the data changes.

It's important to note that using client-side data fetching can affect the performance of your application and the load speed of your pages. This is because the data fetching is done at the time of the component or pages mount, and the data is not cached.

Client-side data fetching with useEffect

The following example shows how you can fetch data on the client side using the useEffect hook.

```
import { useState, useEffect } from 'react'

function Profile() {
  const [data, setData] = useState(null)
  const [isLoading, setLoading] = useState(false)

  useEffect(() => {
    setLoading(true)
    fetch('/api/profile-data')
      .then((res) => res.json())
      .then((data) => {
        setData(data)
        setLoading(false)
      })
  }, [])

  if (isLoading) return <p>Loading...</p>
  if (!data) return <p>No profile data</p>

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

Client-side data fetching with SWR

The team behind Next.js has created a React hook library for data fetching called **SWR**. It is **highly recommended** if you are fetching data on the client-side. It handles caching, revalidation, focus tracking, refetching on intervals, and more.

Using the same example as above, we can now use SWR to fetch the profile data. SWR will automatically cache the data for us and will revalidate the data if it becomes stale.

For more information on using SWR, check out the SWR docs.

```
import useSWR from 'swr'

const fetcher = (...args) => fetch(...args).then((res) => res.json())

function Profile() {
  const { data, error } = useSWR('/api/profile-data', fetcher)

  if (error) return <div>Failed to load</div>
  if (!data) return <div>Loading...</div>

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.bio}</p>
    </div>
  )
}
```

Related

For more information on what to do next, we recommend the following sections:

Routing: Learn more about routing in Next.js.

---

description: Fetch data on each request with `getServerSideProps`.

getServerSideProps

If you export a function called `getServerSideProps` (Server-Side Rendering) from a page, Next.js will pre-render this page on each request using the data returned by `getServerSideProps`.

```
export async function getServerSideProps(context) {
  return {
    props: {}, // will be passed to the page component as props
  }
}
```

> Note that irrespective of rendering type, any `props` will be passed to the page component and can be viewed on the client-side in the initial HTML. This is to allow the page to be hydrated correctly. Make sure that you don't pass any sensitive information that shouldn't be available on the client in `props`.

When does getServerSideProps run

`getServerSideProps` only runs on server-side and never runs on the browser. If a page uses `getServerSideProps`, then:

- When you request this page directly, `getServerSideProps` runs at request time, and this page will be pre-rendered with the returned props
- When you request this page on client-side page transitions through `next/link` or `next/router`, Next.js sends an API request to the server, which runs `getServerSideProps`

`getServerSideProps` returns JSON which will be used to render the page. All this work will be handled automatically by Next.js, so you don't need to do anything extra as long as you have `getServerSideProps` defined.

You can use the next-code-elimination tool to verify what Next.js eliminates from the client-side bundle.

`getServerSideProps` can only be exported from a **page**. You can't export it from non-page files.

Note that you must export `getServerSideProps` as a standalone function — it will **not** work if you add `getServerSideProps` as a property of the page component.

The `getServerSideProps` API reference covers all parameters and props that can be used with `getServerSideProps`.

When should I use getServerSideProps

You should use `getServerSideProps` only if you need to render a page whose data must be fetched at request time. This could be due to the nature of the data or properties of the request (such as `authorization` headers or geo location). Pages using `getServerSideProps` will be server side rendered at request time and only be cached if cache-control headers are configured.

If you do not need to render the data during the request, then you should consider fetching data on the client side or `getStaticProps`.

getServerSideProps or API Routes

It can be tempting to reach for an API Route when you want to fetch data from the server, then call that API route from `getServerSideProps`. This is an unnecessary and inefficient approach, as it will cause an extra request to be made due to both `getServerSideProps` and API Routes running on the server.

Take the following example. An API route is used to fetch some data from a CMS. That API route is then called directly from `getServerSideProps`. This produces an additional call, reducing performance. Instead, directly import the logic used inside your API Route into `getServerSideProps`. This could mean calling a CMS, database, or other API directly from inside `getServerSideProps`.

getServerSideProps with Edge API Routes

`getServerSideProps` can be used with both Serverless and Edge Runtimes, and you can set props in both. However, currently in Edge Runtime, you do not have access to the response object. This means that you cannot — for example — add cookies in `getServerSideProps`. To have access to the response object, you should **continue to use the Node.js runtime**, which is the default runtime.

You can explicitly set the runtime on a per-page basis by modifying the `config`, for example:

```
export const config = {
  runtime: 'nodejs',
}
```

Fetching data on the client side

If your page contains frequently updating data, and you don't need to pre-render the data, you can fetch the data on the client side. An example of this is user-specific data:

- First, immediately show the page without data. Parts of the page can be pre-rendered using Static Generation. You can show loading states for missing data
- Then, fetch the data on the client side and display it when ready

This approach works well for user dashboard pages, for example. Because a dashboard is a private, user-specific page, SEO is not relevant and the page doesn't need to be pre-rendered. The data is frequently updated, which requires request-time data fetching.

Using getServerSideProps to fetch data at request time

The following example shows how to fetch data at request time and pre-render the result.

```
function Page({ data }) {
  // Render data...
}

// This gets called on every request
export async function getServerSideProps() {
  // Fetch data from external API
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  // Pass data to the page via props
  return { props: { data } }
}

export default Page
```

Caching with Server-Side Rendering (SSR)

You can use caching headers (`Cache-Control`) inside `getServerSideProps` to cache dynamic responses. For example, using `stale-while-revalidate`.

```
// This value is considered fresh for ten seconds (s-maxage=10).
// If a request is repeated within the next 10 seconds, the previously
// cached value will still be fresh. If the request is repeated before 59 seconds,
// the cached value will be stale but still render (stale-while-revalidate=59).
//
// In the background, a revalidation request will be made to populate the cache
// with a fresh value. If you refresh the page, you will see the new value.
export async function getServerSideProps({ req, res }) {
  res.setHeader(
    'Cache-Control',
    'public, s-maxage=10, stale-while-revalidate=59'
  )

  return {
    props: {},
  }
}
```

Learn more about caching.

Does getServerSideProps render an error page

If an error is thrown inside `getServerSideProps`, it will show the `pages/500.js` file. Check out the documentation for 500 page to learn more on how to create it. During development this file will not be used and the dev overlay will be shown instead.

Related

For more information on what to do next, we recommend the following sections:

getServerSideProps API Reference Read the API Reference for getServerSideProps

---

description: Fetch data and generate static pages with `getStaticProps`. Learn more about this API for data fetching in Next.js.

getStaticPaths

If a page has Dynamic Routes and uses `getStaticProps`, it needs to define a list of paths to be statically generated.

When you export a function called `getStaticPaths` (Static Site Generation) from a page that uses dynamic routes, Next.js will statically pre-render all the paths specified by `getStaticPaths`.

```
// pages/posts/[id].js

// Generates `/posts/1` and `/posts/2`
export async function getStaticPaths() {
  return {
    paths: [{ params: { id: '1' } }, { params: { id: '2' } }],
    fallback: false, // can also be true or 'blocking'
  }
}

// `getStaticPaths` requires using `getStaticProps`
export async function getStaticProps(context) {
  return {
    // Passed to the page component as props
    props: { post: {} },
  }
}

export default function Post({ post }) {
  // Render post...
}
```

The `getStaticPaths` API reference covers all parameters and props that can be used with `getStaticPaths`.

When should I use getStaticPaths?

You should use `getStaticPaths` if you're statically pre-rendering pages that use dynamic routes and:

- The data comes from a headless CMS
- The data comes from a database
- The data comes from the filesystem
- The data can be publicly cached (not user-specific)
- The page must be pre-rendered (for SEO) and be very fast — `getStaticProps` generates `HTML` and `JSON` files, both of which can be cached by a CDN for performance

When does getStaticPaths run

`getStaticPaths` will only run during build in production, it will not be called during runtime. You can validate code written inside `getStaticPaths` is removed from the client-side bundle with this tool.

How does getStaticProps run with regards to getStaticPaths

- `getStaticProps` runs during `next build` for any `paths` returned during build
- `getStaticProps` runs in the background when using `fallback: true`
- `getStaticProps` is called before initial render when using `fallback: blocking`

Where can I use getStaticPaths

- `getStaticPaths` **must** be used with `getStaticProps`
- You **cannot** use `getStaticPaths` with `getServerSideProps`
- You can export `getStaticPaths` from a Dynamic Route that also uses `getStaticProps`
- You **cannot** export `getStaticPaths` from non-page file (e.g. your `components` folder)
- You must export `getStaticPaths` as a standalone function, and not a property of the page component

Runs on every request in development

In development (`next dev`), `getStaticPaths` will be called on every request.

Generating paths on-demand

`getStaticPaths` allows you to control which pages are generated during the build instead of on-demand with `fallback`. Generating more pages during a build will cause slower builds.

You can defer generating all pages on-demand by returning an empty array for `paths`. This can be especially helpful when deploying your Next.js application to multiple environments. For example, you can have faster builds by generating all pages on-demand for previews (but not production builds). This is helpful for sites with hundreds/thousands of static pages.

```
// pages/posts/[id].js

export async function getStaticPaths() {
  // When this is true (in preview environments) don't
  // prerender any static pages
  // (faster builds, but slower initial page load)
  if (process.env.SKIP_BUILD_STATIC_GENERATION) {
    return {
      paths: [],
      fallback: 'blocking',
    }
  }

  // Call an external API endpoint to get posts
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to prerender based on posts
  // In production environments, prerender all pages
  // (slower builds, but faster initial page load)
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

  // { fallback: false } means other routes should 404
  return { paths, fallback: false }
}
```

Related

For more information on what to do next, we recommend the following sections:

getStaticPaths API Reference Read the API Reference for getStaticPaths

---

description: Fetch data and generate static pages with `getStaticProps`. Learn more about this API for data fetching in Next.js.

getStaticProps

If you export a function called `getStaticProps` (Static Site Generation) from a page, Next.js will pre-render this page at build time using the props returned by `getStaticProps`.

```
export async function getStaticProps(context) {
  return {
    props: {}, // will be passed to the page component as props
  }
}
```

> Note that irrespective of rendering type, any `props` will be passed to the page component and can be viewed on the client-side in the initial HTML. This is to allow the page to be hydrated correctly. Make sure that you don't pass any sensitive information that shouldn't be available on the client in `props`.

When should I use getStaticProps?

You should use `getStaticProps` if:

- The data required to render the page is available at build time ahead of a user's request
- The data comes from a headless CMS
- The page must be pre-rendered (for SEO) and be very fast — `getStaticProps` generates `HTML` and `JSON` files, both of which can be cached by a CDN for performance
- The data can be publicly cached (not user-specific). This condition can be bypassed in certain specific situation by using a Middleware to rewrite the path.

When does getStaticProps run

`getStaticProps` always runs on the server and never on the client. You can validate that code written inside `getStaticProps` is removed from the client-side bundle with this tool.

- `getStaticProps` always runs during `next build`
- `getStaticProps` runs in the background when using `fallback: true`
- `getStaticProps` is called before initial render when using `fallback: blocking`
- `getStaticProps` runs in the background when using `revalidate`
- `getStaticProps` runs on-demand in the background when using `revalidate()`

When combined with Incremental Static Regeneration, `getStaticProps` will run in the background while the stale page is being revalidated, and the fresh page served to the browser.

`getStaticProps` does not have access to the incoming request (such as query parameters or HTTP headers) as it generates static HTML. If you need access to the request for your page, consider using Middleware in addition to `getStaticProps`.

Using getStaticProps to fetch data from a CMS

The following example shows how you can fetch a list of blog posts from a CMS.

```
// posts will be populated at build time by getStaticProps()
function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>{post.title}</li>
      ))}
    </ul>
  )
}

// This function gets called at build time on server-side.
// It won't be called on client-side, so you can even do
// direct database queries.
export async function getStaticProps() {
  // Call an external API endpoint to get posts.
  // You can use any data fetching library
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // By returning { props: { posts } }, the Blog component
  // will receive `posts` as a prop at build time
  return {
    props: {
      posts,
    },
  }
}

export default Blog
```

The `getStaticProps` API reference covers all parameters and props that can be used with `getStaticProps`.

Write server-side code directly

As `getStaticProps` runs only on the server-side, it will never run on the client-side. It won't even be included in the JS bundle for the browser, so you can write direct database queries without them being sent to browsers.

This means that instead of fetching an **API route** from `getStaticProps` (that itself fetches data from an external source), you can write the server-side code directly in `getStaticProps`.

Take the following example. An API route is used to fetch some data from a CMS. That API route is then called directly from `getStaticProps`. This produces an additional call, reducing performance. Instead, the logic for fetching the data from the CMS can be shared by using a `lib/` directory. Then it can be shared with `getStaticProps`.

```
// lib/load-posts.js

// The following function is shared
// with getStaticProps and API routes
// from a `lib/` directory
export async function loadPosts() {
  // Call an external API endpoint to get posts
  const res = await fetch('https://.../posts/')
  const data = await res.json()

  return data
}

// pages/blog.js
import { loadPosts } from '../lib/load-posts'

// This function runs only on the server side
export async function getStaticProps() {
  // Instead of fetching your `/api` route you can call the same
  // function directly in `getStaticProps`
  const posts = await loadPosts()

  // Props returned will be passed to the page component
  return { props: { posts } }
}
```

Alternatively, if you are **not** using API routes to fetch data, then the `fetch()` API *can* be used directly in `getStaticProps` to fetch data.

To verify what Next.js eliminates from the client-side bundle, you can use the next-code-elimination tool.

Statically generates both HTML and JSON

When a page with `getStaticProps` is pre-rendered at build time, in addition to the page HTML file, Next.js generates a JSON file holding the result of running `getStaticProps`.

This JSON file will be used in client-side routing through `next/link` or `next/router`. When you navigate to a page that's pre-rendered using `getStaticProps`, Next.js fetches this JSON file (pre-computed at build time) and uses it as the props for the page component. This means that client-side page transitions will **not** call `getStaticProps` as only the exported JSON is used.

When using Incremental Static Generation, `getStaticProps` will be executed in the background to generate the JSON needed for client-side navigation. You may see this in the form of multiple requests being made for the same page, however, this is intended and has no impact on end-user performance.

Where can I use getStaticProps

`getStaticProps` can only be exported from a **page**. You **cannot** export it from non-page files, `_app`, `_document`, or `_error`.

One of the reasons for this restriction is that React needs to have all the required data before the page is rendered.

Also, you must use export `getStaticProps` as a standalone function — it will **not** work if you add `getStaticProps` as a property of the page component.

> **Note**: if you have created a custom app, ensure you are passing the `pageProps` to the page component as shown in the linked document, otherwise the props will be empty.

Runs on every request in development

In development (`next dev`), `getStaticProps` will be called on every request.

Preview Mode

You can temporarily bypass static generation and render the page at **request time** instead of build time using **Preview Mode**. For example, you might be using a headless CMS and want to preview drafts before they're published.

Related

For more information on what to do next, we recommend the following sections:

getStaticProps API Reference Read the API Reference for getStaticProps

---

description: 'Learn how to create or update static pages at runtime with Incremental Static Regeneration.'

Incremental Static Regeneration

▶ **Examples**
▶ **Version History**

Next.js allows you to create or update static pages *after* you've built your site. Incremental Static Regeneration (ISR) enables you to use static-generation on a per-page basis, **without needing to rebuild the entire site**. With ISR, you can retain the benefits of static while scaling to millions of pages.

> **Note**: The `experimental-edge` runtime is currently not compatible with ISR, although you can leverage `stale-while-revalidate` by setting the `cache-control` header manually.

To use ISR, add the `revalidate` prop to `getStaticProps`:

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

// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// revalidation is enabled and a new request comes in
export async function getStaticProps() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  return {
    props: {
      posts,
    },
    // Next.js will attempt to re-generate the page:
    // - When a request comes in
    // - At most once every 10 seconds
    revalidate: 10, // In seconds
  }
}

// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// the path has not been generated.
export async function getStaticPaths() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to pre-render based on posts
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

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

```
export default Blog
```

When a request is made to a page that was pre-rendered at build time, it will initially show the cached page.

- Any requests to the page after the initial request and before 10 seconds are also cached and instantaneous.
- After the 10-second window, the next request will still show the cached (stale) page
- Next.js triggers a regeneration of the page in the background.
- Once the page generates successfully, Next.js will invalidate the cache and show the updated page. If the background regeneration fails, the old page would still be unaltered.

When a request is made to a path that hasn't been generated, Next.js will server-render the page on the first request. Future requests will serve the static file from the cache. ISR on Vercel persists the cache globally and handles rollbacks.

> **Note**: 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.

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 from someone visiting that page after the minute has passed.

Starting with `v12.2.0`, Next.js supports On-Demand Incremental Static Regeneration to manually purge the Next.js cache for a specific page. This makes it easier to update your site when:

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

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

> **Note**: Middleware won't be executed for On-Demand ISR requests. Instead, call `revalidate()` on the *exact* path that you want revalidated. For example, if you have `pages/blog/[slug].js` and a rewrite from `/post-1` -> `/blog/post-1`, you would need to call `res.revalidate('/blog/post-1')`.

Using On-Demand Revalidation

First, create a secret token only known by your Next.js app. This secret will be used to prevent unauthorized access to the revalidation API Route. You can access the route (either manually or with a webhook) with the following URL structure:

```
https://<your-site.com>/api/revalidate?secret=<token>
```

Next, add the secret as an Environment Variable to your application. Finally, create the revalidation API Route:

```
// pages/api/revalidate.js

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

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

View our demo to see on-demand revalidation in action and provide feedback.

Testing on-Demand ISR during development

When running locally with `next dev`, `getStaticProps` is invoked on every request. To verify your on-demand ISR configuration is correct, you will need to create a production build and start the production server:

```
$ next build
$ next start
```

Then, you can confirm that static pages have successfully revalidated.

Error handling and revalidation

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

```
export async function getStaticProps() {
  // If this request throws an uncaught error, Next.js will
  // not invalidate the currently shown page and
  // retry getStaticProps on the next request.
  const res = await fetch('https://.../posts')
  const posts = await res.json()

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

  // If the request was successful, return the posts
  // and revalidate every 10 seconds.
  return {
    props: {
      posts,
    },
    revalidate: 10,
```

```
  }
}
```

Self-hosting ISR

Incremental Static Regeneration (ISR) works on self-hosted Next.js sites out of the box when you use `next start`.

You can use this approach when deploying to container orchestrators such as Kubernetes or HashiCorp Nomad. By default, generated assets will be stored in-memory on each pod. This means that each pod will have its own copy of the static files. Stale data may be shown until that specific pod is hit by a request.

To ensure consistency across all pods, you can disable in-memory caching. This will inform the Next.js server to only leverage assets generated by ISR in the file system.

You can use a shared network mount in your Kubernetes pods (or similar setup) to reuse the same file-system cache between different containers. By sharing the same mount, the `.next` folder which contains the `next/image` cache will also be shared and re-used.

To disable in-memory caching, set `isrMemoryCacheSize` to `0` in your `next.config.js` file:

```
module.exports = {
  experimental: {
    // Defaults to 50MB
    isrMemoryCacheSize: 0,
  },
}
```

> **Note**: You might need to consider a race condition between multiple pods trying to update the cache at the same time, depending on how your shared mount is configured.

Related

For more information on what to do next, we recommend the following sections:

Dynamic routing Learn more about dynamic routing in Next.js with getStaticPaths.

---

description: 'Next.js allows you to fetch data in multiple ways, with pre-rendering, server-side rendering or static-site generation, and incremental static regeneration. Learn how to manage your application data in Next.js.'

Data Fetching Overview

▶ **Examples**

> **Note**: Next.js 13 introduces the `app/` directory (beta). This new directory has support for colocated data fetching at the component level, using the new React `use` hook and an extended `fetch` Web API.
>
> Learn more about incrementally adopting `app/`.

Data fetching in Next.js allows you to render your content in different ways, depending on your application's use case. These include pre-rendering with **Server-side Rendering** or **Static Generation**, and updating or creating content at runtime with **Incremental Static Regeneration**.

SSR: Server-side rendering Learn more about server-side rendering in Next.js with getServerSideProps.
SSG: Static-site generation Learn more about static site generation in Next.js with getStaticProps.
CSR: Client-side rendering Learn more about client side rendering in Next.js with SWR.
Dynamic routing Learn more about dynamic routing in Next.js with getStaticPaths.
ISR: Incremental Static Regeneration Learn more about Incremental Static Regeneration in Next.js.

Learn more

Preview Mode: Learn more about the preview mode in Next.js.
Routing: Learn more about routing in Next.js.
TypeScript: Add TypeScript to your pages.

---

description: Learn to add and access environment variables in your Next.js application.

Environment Variables

▶ **Examples**

Next.js comes with built-in support for environment variables, which allows you to do the following:

- Use `.env.local` to load environment variables
- Expose environment variables to 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`.

An example `.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 Next.js data fetching methods and API routes.

For example, using `getStaticProps`:

```
// pages/index.js
export async function getStaticProps() {
  const db = await myDB.connect({
    host: process.env.DB_HOST,
```

```
    username: process.env.DB_USER,
    password: process.env.DB_PASS,
  })
  // ...
}
```

Note: In order to keep server-only secrets safe, environment variables are evaluated at build time, so only environment variables *actually* used will be included. This means that `process.env` is not a standard JavaScript object, so you're not able to use object destructuring. Environment variables must be referenced as e.g. `process.env.PUBLISHABLE_KEY`, *not* `const { PUBLISHABLE_KEY } = process.env`.

Note: Next.js will automatically expand variables (`$VAR`) inside of your `.env*` files. This allows you to reference other secrets, like so:

```
# .env
HOSTNAME=localhost
PORT=8080
HOST=http://$HOSTNAME:$PORT
```

If you are trying to use a variable with a `$` in the actual value, it needs to be escaped like so: `\$`.

For example:

```
# .env
A=abc

# becomes "preabc"
WRONG=pre$A

# becomes "pre$A"
CORRECT=pre\$A
```

Note: If you are using a `/src` folder, please note that Next.js will load the .env files **only** from the parent folder and **not** from the `/src` folder.

Exposing Environment Variables to the Browser

By default environment variables are only available in the Node.js environment, meaning they won't be exposed to the browser.

In order to expose a variable to the browser you have to prefix the variable with `NEXT_PUBLIC_`. For example:

```
NEXT_PUBLIC_ANALYTICS_ID=abcdefghijk
```

This loads `process.env.NEXT_PUBLIC_ANALYTICS_ID` into the Node.js environment automatically, allowing you to use it anywhere in your code. The value will be inlined into JavaScript sent to the browser because of the `NEXT_PUBLIC_` prefix. This inlining occurs at build time, so your various `NEXT_PUBLIC_` envs need to be set when the project is built.

```
// 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.

Note: `.env`, `.env.development`, and `.env.production` files should be included in your repository as they define defaults. **`.env*.local` should be added to `.gitignore`**, as those files are intended to be ignored. `.env.local` is where secrets can be stored.

Environment Variables on Vercel

When deploying your Next.js application to Vercel, Environment Variables can be configured in the Project Settings.

All types of Environment Variables should be configured there. Even Environment Variables used in Development – which can be downloaded onto your local device afterwards.

If you've configured Development Environment Variables you can pull them into a `.env.local` for usage on your local machine using the following command:

```
vercel env pull .env.local
```

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).

> **Note**: similar to Default Environment Variables, `.env.test` file should be included in your repository, but `.env.test.local` shouldn't, as `.env*.local` are intended to be ignored through `.gitignore`.

While running unit tests you can make sure to load your environment variables the same way Next.js does by leveraging the `loadEnvConfig` function from the `@next/env` package.

```
// The below can be used in a Jest global setup file or similar for your testing set-up
import { loadEnvConfig } from '@next/env'

export default async () => {
  const projectDir = process.cwd()
  loadEnvConfig(projectDir)
}
```

Environment Variable Load Order

Environment variables are looked up in the following places, in order, stopping once the variable is found.

1. `process.env`
2. `.env.$(NODE_ENV).local`
3. `.env.local` (Not checked when `NODE_ENV` is `test`.)
4. `.env.$(NODE_ENV)`
5. `.env`

For example, if `NODE_ENV` is `development` and you define a variable in both `.env.development.local` and `.env`, the value in `.env.development.local` will be used.

> **Note**: The allowed values for `NODE_ENV` are `production`, `development` and `test`.

---

description: Next.js provides an integrated ESLint experience by default. These conformance rules help you use Next.js in the optimal way.

ESLint

Next.js provides an integrated ESLint experience out of the box. Add `next lint` as a script to `package.json`:

```
"scripts": {
  "lint": "next lint"
}
```

Then run `npm run lint` or `yarn lint`:

```
yarn lint
```

If you don't already have ESLint configured in your application, you will be guided through the installation and configuration process.

```
yarn lint

# You'll see a prompt like this:
#
# ? How would you like to configure ESLint?
#
# ❯   Base configuration + Core Web Vitals rule-set (recommended)
#     Base configuration
#     None
```

One of the following three options can be selected:

- **Strict**: Includes Next.js' base ESLint configuration along with a stricter Core Web Vitals rule-set. This is the recommended configuration for developers setting up ESLint for the first time.

  ```
  {
    "extends": "next/core-web-vitals"
  }
  ```

- **Base**: Includes Next.js' base ESLint configuration.

  ```
  {
    "extends": "next"
  }
  ```

- **Cancel**: Does not include any ESLint configuration. Only select this option if you plan on setting up your own custom ESLint configuration.

If either of the two configuration options are selected, Next.js will automatically install `eslint` and `eslint-config-next` as 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

Rule	Description
✓ @next/next/google-font-display	Enforce font-display behavior with Google Fonts.
✓ @next/next/google-font-preconnect	Ensure `preconnect` is used with Google Fonts.
✓ @next/next/inline-script-id	Enforce `id` attribute on `next/script` components with inline content.
✓ @next/next/next-script-for-ga	Prefer `next/script` component when using the inline script for Google Analytics.
✓ @next/next/no-assign-module-variable	Prevent assignment to the `module` variable.
✓ @next/next/no-before-interactive-script-outside-document	Prevent usage of `next/script`'s `beforeInteractive` strategy outside of `pages/_document.js`.
✓ @next/next/no-css-tags	Prevent manual stylesheet tags.
✓ @next/next/no-document-import-in-page	Prevent importing `next/document` outside of `pages/_document.js`.
✓ @next/next/no-duplicate-head	Prevent duplicate usage of `<Head>` in `pages/_document.js`.
✓ @next/next/no-head-element	Prevent usage of `<head>` element.
✓ @next/next/no-head-import-in-document	Prevent usage of `next/head` in `pages/_document.js`.
✓ @next/next/no-html-link-for-pages	Prevent usage of `<a>` elements to navigate to internal Next.js pages.
✓ @next/next/no-img-element	Prevent usage of `<img>` element due to slower LCP and higher bandwidth.
✓ @next/next/no-page-custom-font	Prevent page-only custom fonts.
✓ @next/next/no-script-component-in-head	Prevent usage of `next/script` in `next/head` component.
✓ @next/next/no-styled-jsx-in-document	Prevent usage of `styled-jsx` in `pages/_document.js`.
✓ @next/next/no-sync-scripts	Prevent synchronous scripts.
✓ @next/next/no-title-in-document-head	Prevent usage of `<title>` with `Head` component from `next/document`.
✓ @next/next/no-typos	Prevent common typos in Next.js's data fetching functions
✓ @next/next/no-unwanted-polyfillio	Prevent duplicate polyfills from Polyfill.io.

If you already have ESLint configured in your application, we recommend extending from this plugin directly instead of including `eslint-config-next` unless a few conditions are met. Refer to the Recommended Plugin Ruleset to learn more.

Custom Settings

`rootDir`

If you're using `eslint-plugin-next` in a project where Next.js isn't installed in your root directory (such as a monorepo), you can tell `eslint-plugin-next` where to find your Next.js application using the `settings` property in your `.eslintrc`:

```
{
  "extends": "next",
  "settings": {
    "next": {
      "rootDir": "packages/my-app/"
    }
  }
}
```

`rootDir` can be a path (relative or absolute), a glob (i.e. `"packages/*/"`), or an array of paths and/or globs.

Linting Custom Directories and Files

By default, Next.js will run ESLint for all files in the `pages/`, `app` (only if the experimental `appDir` feature is enabled), `components/`, `lib/`, and `src/` directories. However, you can specify which directories using the `dirs` option in the `eslint` config in `next.config.js` for production builds:

```
module.exports = {
  eslint: {
    dirs: ['pages', 'utils'], // Only run ESLint on the 'pages' and 'utils' directories during production builds (next build)
  },
}
```

Similarly, the `--dir` and `--file` flags can be used for `next lint` to lint specific directories and files:

```
next lint --dir pages --dir utils --file bar.js
```

Caching

To improve performance, information of files processed by ESLint are cached by default. This is stored in `.next/cache` or in your defined build directory. If you include any ESLint rules that depend on more than the contents of a single source file and need to disable the cache, use the `--no-cache` flag with `next lint`.

```
next lint --no-cache
```

Disabling Rules

If you would like to modify or disable any rules provided by the supported plugins (`react`, `react-hooks`, `next`), you can directly change them using the `rules` property in your `.eslintrc`:

```
{
  "extends": "next",
  "rules": {
    "react/no-unescaped-entities": "off",
    "@next/next/no-page-custom-font": "off"
  }
}
```

Core Web Vitals

The `next/core-web-vitals` rule set is enabled when `next lint` is run for the first time and the **strict** option is selected.

```
{
  "extends": "next/core-web-vitals"
}
```

`next/core-web-vitals` updates `eslint-plugin-next` to error on a number of rules that are warnings by default if they affect Core Web Vitals.

> The `next/core-web-vitals` entry point is automatically included for new applications built with Create Next App.

Usage With Other Tools

Prettier

ESLint also contains code formatting rules, which can conflict with your existing Prettier setup. We recommend including eslint-config-prettier in your ESLint config to make ESLint and Prettier work together.

First, install the dependency:

```
npm install --save-dev eslint-config-prettier
# or
yarn add --dev eslint-config-prettier
```

Then, add `prettier` to your existing ESLint config:

```
{
  "extends": ["next", "prettier"]
}
```

lint-staged

If you would like to use `next lint` with lint-staged to run the linter on staged git files, you'll have to add the following to the `.lintstagedrc.js` file in the root of your project in order to specify usage of the `--file` flag.

```
const path = require('path')

const buildEslintCommand = (filenames) =>
  `next lint --fix --file ${filenames
    .map((f) => path.relative(process.cwd(), f))
    .join(' --file ')}`

module.exports = {
  '*.{js,jsx,ts,tsx}': [buildEslintCommand],
}
```

Migrating Existing Config

Recommended Plugin Ruleset

If you already have ESLint configured in your application and any of the following conditions are true:

- You have one or more of the following plugins already installed (either separately or through a different config such as `airbnb` or `react-app`):
  - `react`
  - `react-hooks`
  - `jsx-a11y`
  - `import`
- You've defined specific `parserOptions` that are different from how Babel is configured within Next.js (this is not recommended unless you have customized your Babel configuration)
- You have `eslint-plugin-import` installed with Node.js and/or TypeScript resolvers defined to handle imports

Then we recommend either removing these settings if you prefer how these properties have been configured within `eslint-config-next` or extending directly from the Next.js ESLint plugin instead:

```
module.exports = {
  extends: [
    //...
    'plugin:@next/next/recommended',
  ],
}
```

The plugin can be installed normally in your project without needing to run `next lint`:

```
npm install --save-dev @next/eslint-plugin-next
# or
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:

```
{
  "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.

---

description: Next.js' Fast Refresh is a new hot reloading experience that gives you instantaneous feedback on edits made to your React components.

Fast Refresh

▼ **Examples**

- 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 () => <div />;` cause Fast Refresh to not preserve local component state. For large codebases you can use our `name-default-component` codemod.

As more of your codebase moves to function components and Hooks, you can expect state to be preserved in more cases.

Tips

- Fast Refresh preserves React local state in function components (and Hooks) by default.
- Sometimes you might want to *force* the state to be reset, and a component to be remounted. For example, this can be handy if you're tweaking an animation that only happens on mount. To do this, you can add `// @refresh reset` anywhere in the file you're editing. This directive is local to the file, and instructs Fast Refresh to remount components defined in that file on every edit.
- You can put `console.log` or `debugger;` into the components you edit during development.

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.

---

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

Optimizing Fonts

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

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

Overview

`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.**

To get started, import 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.

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

```
// pages/_app.js
import { Inter } from 'next/font/google'

// If loading a variable font, you don't need to specify the font weight
const inter = Inter({ subsets: ['latin'] })

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={inter.className}>
      <Component {...pageProps} />
    </main>
  )
}
```

If you can't use a variable font, you will **need to specify a weight**:

```
// pages/_app.js
import { Roboto } from 'next/font/google'

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

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={roboto.className}>
      <Component {...pageProps} />
    </main>
  )
}
```

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

```
const roboto = Roboto({
  weight: ['400', '700'],
  style: ['normal', 'italic'],
  subsets: ['latin'],
})
```

> **Note**: You can use _ for fonts with spaces in the name. For example `Titillium Web` should be `Titillium_Web`.

Apply the font in `<head>`

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

```
// pages/_app.js
import { Inter } from 'next/font/google'

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

export default function MyApp({ Component, pageProps }) {
  return (
    <>
      <style jsx global>{`
        html {
          font-family: ${inter.style.fontFamily};
        }
      `}</style>
      <Component {...pageProps} />
    </>
  )
}
```

Single page usage

To use the font on a single page, add it to the specific page as shown below:

```
// pages/index.js
import { Inter } from 'next/font/google'

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

export default function Home() {
  return (
    <div className={inter.className}>
      <p>Hello World</p>
    </div>
  )
}
```

Specifying a subset

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

```
// pages/_app.js
const inter = Inter({ subsets: ['latin'] })
```

View the Font API Reference for more information.

Local Fonts

Import `next/font/local` and specify the `src` of your local font file. We recommend using **variable fonts** for the best performance and flexibility.

```
// pages/_app.js
import localFont from 'next/font/local'

// Font files can be colocated inside of `pages`
const myFont = localFont({ src: './my-font.woff2' })

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={myFont.className}>
      <Component {...pageProps} />
    </main>
  )
}
```

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

```
const roboto = localFont({
  src: [
    {
      path: './Roboto-Regular.woff2',
      weight: '400',
      style: 'normal',
    },
    {
      path: './Roboto-Italic.woff2',
      weight: '400',
      style: 'italic',
    },
    {
      path: './Roboto-Bold.woff2',
      weight: '700',
      style: 'normal',
    },
    {
      path: './Roboto-BoldItalic.woff2',
      weight: '700',
      style: 'italic',
    },
  ],
})
```

View the Font API Reference for more information.

With Tailwind CSS

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

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

```
// pages/_app.js
import { Inter } from 'next/font/google'

const inter = Inter({
  subsets: ['latin'],
  variable: '--font-inter',
})

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={`${inter.variable} font-sans`}>
      <Component {...pageProps} />
    </main>
  )
}
```

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

```
// tailwind.config.js
const { fontFamily } = require('tailwindcss/defaultTheme')

/** @type {import('tailwindcss').Config} \*/
module.exports = {
  content: [
    './pages/**/*.{js,ts,jsx,tsx}',
    './components/**/*.{js,ts,jsx,tsx}',
  ],
  theme: {
    extend: {
      fontFamily: {
        sans: ['var(--font-inter)', ...fontFamily.sans],
      },
    },
  },
  plugins: [],
}
```

You can now use the `font-sans` utility class 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 route/s based on the type of file where it is used:

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

Reusing fonts

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

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

Next Steps

Font API Reference See the API Reference for the Font module.
Image Optimization Learn how to optimize images with the Image component.

---

description: Next.js supports built-in image optimization, as well as third party loaders for Imgix, Cloudinary, and more! Learn more here.

Image Component and Image Optimization

▼ **Examples**

- Image Component

The Next.js Image component, `next/image`, is an extension of the HTML `<img>` element, evolved for the modern web. It includes a variety of built-in performance optimizations to help you achieve good Core Web Vitals. These scores are an important measurement of user experience on your website, and are factored into Google's search rankings.

Some of the optimizations built into the Image component include:

- **Improved Performance:** Always serve correctly sized image for each device, using modern image formats
- **Visual Stability:** Prevent Cumulative Layout Shift automatically
- **Faster Page Loads:** Images are only loaded when they enter the viewport, with optional blur-up placeholders
- **Asset Flexibility:** On-demand image resizing, even for images stored on remote servers

Using the Image Component

To add an image to your application, import the `next/image` component:

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

Now, you can define the `src` for your image (either local or remote).

Local Images

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

```
import profilePic from '../assets/me.png'
```

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

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

```
import Image from 'next/image'
import profilePic from '../assets/me.png'

function Home() {
  return (
    <>
      <h1>My Homepage</h1>
      <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
      />
      <p>Welcome to my homepage!</p>
    </>
  )
}
```

Remote Images

To use a remote image, the `src` property should be a URL string, which can be relative or absolute. Because 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:

```
import Image from 'next/image'

export default function Home() {
  return (
    <>
      <h1>My Homepage</h1>
      <Image
        src="/me.png"
        alt="Picture of the author"
        width={500}
        height={500}
      />
      <p>Welcome to my homepage!</p>
    </>
  )
}
```

Learn more about the sizing requirements in `next/image`.

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:

```
import Image from 'next/image'

export default function Home() {
  return (
    <>
      <h1>My Homepage</h1>
      <Image
        src="/me.png"
        alt="Picture of the author"
        width={500}
        height={500}
        priority
      />
      <p>Welcome to my homepage!</p>
    </>
  )
}
```

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

Image Sizing

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

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

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

What if I don't know the size of my images?

If you are accessing images from a source without knowledge of the images' sizes, there are several things you can do:

Use `fill`

The `fill` prop allows your image to be sized by its parent element. Consider using CSS to give the image's parent element space on the page along `sizes` prop to match any media query break points. You can also use `object-fit` with `fill`, `contain`, or `cover`, and `object-position` to define how the image should occupy that space.

Normalize your images

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

Modify your API calls

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

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

Styling

Styling the Image component is similar to styling a normal `<img>` element, but there are a few guidelines to keep in mind:

Use `className` or `style`, not `styled-jsx`

In most cases, we recommend using the `className` prop. This can be an imported CSS Module, a global stylesheet, etc.

You can also use the `style` prop to assign inline styles.

You cannot use styled-jsx because it's scoped to the current component (unless you mark the style as `global`).

When using `fill`, the parent element must have `position: relative`

This is necessary for the proper rendering of the image element in that layout mode.

When using `fill`, the parent element must have `display: block`

This is the default for `<div>` elements but should be specified otherwise.

Properties

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

Styling Examples

For examples of the Image component used with the various styles, see the Image Component Demo.

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.

Related

For more information on what to do next, we recommend the following sections:

next/image See all available properties for the Image component

---

description: Learn how to share components and state between Next.js pages with Layouts.

Layouts

> **Note**: Next.js 13 introduces the `app/` directory (beta). This new directory has support for layouts, nested routes, and uses Server Components by default. Inside `app/`, you can fetch data for your entire application inside layouts, including support for more granular nested layouts (with colocated data fetching).
>
> Learn more about incrementally adopting `app/`.

The React model allows us to deconstruct a page into a series of components. Many of these components are often reused between pages. For example, you might have the same navigation bar and footer on every page.

```
// components/layout.js

import Navbar from './navbar'
import Footer from './footer'

export default function Layout({ children }) {
  return (
    <>
      <Navbar />
      <main>{children}</main>
      <Footer />
    </>
  )
}
```

Examples

Single Shared Layout with Custom App

If you only have one layout for your entire application, you can create a Custom App and wrap your application with the layout. Since the `<Layout />` component is re-used when changing pages, its component state will be preserved (e.g. input values).

```
// pages/_app.js

import Layout from '../components/layout'

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

Per-Page Layouts

If you need multiple layouts, you can add a property `getLayout` to your page, allowing you to return a React component for the layout. This allows you to define the layout on a *per-page basis*. Since we're returning a function, we can have complex nested layouts if desired.

```
// pages/index.js

import Layout from '../components/layout'
import NestedLayout from '../components/nested-layout'

export default function Page() {
  return (
    /** Your content */
  )
}

Page.getLayout = function getLayout(page) {
  return (
    <Layout>
      <NestedLayout>{page}</NestedLayout>
    </Layout>
```

```
  )
}

// pages/_app.js

export default function MyApp({ Component, pageProps }) {
  // Use the layout defined at the page level, if available
  const getLayout = Component.getLayout || ((page) => page)

  return getLayout(<Component {...pageProps} />)
}
```

When navigating between pages, we want to *persist* page state (input values, scroll position, etc.) for a Single-Page Application (SPA) experience.

This layout pattern enables state persistence because the React component tree is maintained between page transitions. With the component tree, React can understand which elements have changed to preserve state.

> **Note**: This process is called reconciliation, which is how React understands which elements have changed.

With TypeScript

When using TypeScript, you must first create a new type for your pages which includes a `getLayout` function. Then, you must create a new type for your `AppProps` which overrides the `Component` property to use the previously created type.

```
// pages/index.tsx

import type { ReactElement } from 'react'
import Layout from '../components/layout'
import NestedLayout from '../components/nested-layout'
import type { NextPageWithLayout } from './_app'

const Page: NextPageWithLayout = () => {
  return <p>hello world</p>
}

Page.getLayout = function getLayout(page: ReactElement) {
  return (
    <Layout>
      <NestedLayout>{page}</NestedLayout>
    </Layout>
  )
}

export default Page

// pages/_app.tsx

import type { ReactElement, ReactNode } from 'react'
import type { NextPage } from 'next'
import type { AppProps } from 'next/app'

export type NextPageWithLayout<P = {}, IP = P> = NextPage<P, IP> & {
  getLayout?: (page: ReactElement) => ReactNode
}

type AppPropsWithLayout = AppProps & {
  Component: NextPageWithLayout
}

export default function MyApp({ Component, pageProps }: AppPropsWithLayout) {
  // Use the layout defined at the page level, if available
  const getLayout = Component.getLayout ?? ((page) => page)

  return getLayout(<Component {...pageProps} />)
}
```

Data Fetching

Inside your layout, you can fetch data on the client-side using `useEffect` or a library like SWR. Because this file is not a Page, you cannot use `getStaticProps` or `getServerSideProps` currently.

```
// components/layout.js

import useSWR from 'swr'
import Navbar from './navbar'
import Footer from './footer'

export default function Layout({ children }) {
  const { data, error } = useSWR('/api/navigation', fetcher)

  if (error) return <div>Failed to load</div>
  if (!data) return <div>Loading...</div>

  return (
    <>
      <Navbar links={data.links} />
      <main>{children}</main>
      <Footer />
    </>
  )
}
```

For more information on what to do next, we recommend the following sections:

Pages: Learn more about what pages are in Next.js.
Custom App: Learn more about how Next.js initializes pages.

---

description: Next.js pages are React Components exported in a file in the pages directory. Learn how they work here.

Pages

> **Note**: Next.js 13 introduces the `app/` directory (beta). This new directory has support for layouts, nested routes, and uses Server Components by default. Inside `app/`, you can fetch data for your entire application inside layouts, including support for more granular nested layouts (with colocated data fetching).
>
> Learn more about incrementally adopting `app/`.

In Next.js, a **page** is a React Component exported from a `.js`, `.jsx`, `.ts`, or `.tsx` file in the `pages` directory. Each page is associated with a route based on its file name.

Example: If you create `pages/about.js` that exports a React component like below, it will be accessible at `/about`.

```
export default function About() {
  return <div>About</div>
}
```

Pages with Dynamic Routes

Next.js supports pages with dynamic routes. For example, if you create a file called `pages/posts/[id].js`, then it will be accessible at `posts/1`, `posts/2`, etc.

> To learn more about dynamic routing, check the Dynamic Routing documentation.

Pre-rendering

By default, Next.js **pre-renders** every page. This means that Next.js generates HTML for each page in advance, instead of having it all done by client-side JavaScript. Pre-rendering can result in better performance and SEO.

Each generated HTML is associated with minimal JavaScript code necessary for that page. When a page is loaded by the browser, its JavaScript code runs and makes the page fully interactive. (This process is called *hydration*.)

Two forms of Pre-rendering

Next.js has two forms of pre-rendering: **Static Generation** and **Server-side Rendering**. The difference is in **when** it generates the HTML for a page.

- **Static Generation (Recommended)**: The HTML is generated at **build time** and will be reused on each request.
- **Server-side Rendering**: The HTML is generated on **each request**.

Importantly, Next.js lets you **choose** which pre-rendering form you'd like to use for each page. You can create a "hybrid" Next.js app by using Static Generation for most pages and using Server-side Rendering for others.

We **recommend** using **Static Generation** over Server-side Rendering for performance reasons. Statically generated pages can be cached by CDN with no extra configuration to boost performance. However, in some cases, Server-side Rendering might be the only option.

You can also use **Client-side data fetching** along with Static Generation or Server-side Rendering. That means some parts of a page can be rendered entirely by client side JavaScript. To learn more, take a look at the Data Fetching documentation.

Static Generation

▶ **Examples**

If a page uses **Static Generation**, the page HTML is generated at **build time**. That means in production, the page HTML is generated when you run `next build`. This HTML will then be reused on each request. It can be cached by a CDN.

In Next.js, you can statically generate pages **with or without data**. Let's take a look at each case.

Static Generation without data

By default, Next.js pre-renders pages using Static Generation without fetching data. Here's an example:

```
function About() {
  return <div>About</div>
}

export default About
```

Note that this page does not need to fetch any external data to be pre-rendered. In cases like this, Next.js generates a single HTML file per page during build time.

Static Generation with data

Some pages require fetching external data for pre-rendering. There are two scenarios, and one or both might apply. In each case, you can use these functions that Next.js provides:

1. Your page **content** depends on external data: Use `getStaticProps`.
2. Your page **paths** depend on external data: Use `getStaticPaths` (usually in addition to `getStaticProps`).

Scenario 1: Your page content depends on external data

Example: Your blog page might need to fetch the list of blog posts from a CMS (content management system).

```
// TODO: Need to fetch `posts` (by calling some API endpoint)
//       before this page can be pre-rendered.
export default function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>{post.title}</li>
      ))}
    </ul>
  )
}
```

To fetch this data on pre-render, Next.js allows you to `export` an `async` function called `getStaticProps` from the same file. This function gets called at build time and lets you pass fetched data to the page's `props` on pre-render.

```
export default function Blog({ posts }) {
  // Render posts...
}

// This function gets called at build time
export async function getStaticProps() {
```

```
  // Call an external API endpoint to get posts
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // By returning { props: { posts } }, the Blog component
  // will receive `posts` as a prop at build time
  return {
    props: {
      posts,
    },
  }
}
```

To learn more about how `getStaticProps` works, check out the Data Fetching documentation.

Scenario 2: Your page paths depend on external data

Next.js allows you to create pages with **dynamic routes**. For example, you can create a file called `pages/posts/[id].js` to show a single blog post based on `id`. This will allow you to show a blog post with `id: 1` when you access `posts/1`.

> To learn more about dynamic routing, check the Dynamic Routing documentation.

However, which `id` you want to pre-render at build time might depend on external data.

Example: suppose that you've only added one blog post (with `id: 1`) to the database. In this case, you'd only want to pre-render `posts/1` at build time.

Later, you might add the second post with `id: 2`. Then you'd want to pre-render `posts/2` as well.

So your page **paths** that are pre-rendered depend on external data**.** To handle this, Next.js lets you `export` an `async` function called `getStaticPaths` from a dynamic page (`pages/posts/[id].js` in this case). This function gets called at build time and lets you specify which paths you want to pre-render.

```
// This function gets called at build time
export async function getStaticPaths() {
  // Call an external API endpoint to get posts
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to pre-render based on posts
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

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

Also in `pages/posts/[id].js`, you need to export `getStaticProps` so that you can fetch the data about the post with this `id` and use it to pre-render the page:

```
export default function Post({ post }) {
  // Render post...
}

export async function getStaticPaths() {
  // ...
}

// This also gets called at build time
export async function getStaticProps({ params }) {
  // params contains the post `id`.
  // If the route is like /posts/1, then params.id is 1
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  // Pass post data to the page via props
  return { props: { post } }
}
```

To learn more about how `getStaticPaths` works, check out the Data Fetching documentation.

When should I use Static Generation?

We recommend using **Static Generation** (with and without data) whenever possible because your page can be built once and served by CDN, which makes it much faster than having a server render the page on every request.

You can use Static Generation for many types of pages, including:

- Marketing pages
- Blog posts and portfolios
- E-commerce product listings
- Help and documentation

You should ask yourself: "Can I pre-render this page **ahead** of a user's request?" If the answer is yes, then you should choose Static Generation.

On the other hand, Static Generation is **not** a good idea if you cannot pre-render a page ahead of a user's request. Maybe your page shows frequently updated data, and the page content changes on every request.

In cases like this, you can do one of the following:

- Use Static Generation with **Client-side data fetching:** You can skip pre-rendering some parts of a page and then use client-side JavaScript to populate them. To learn more about this approach, check out the Data Fetching documentation.
- Use **Server-Side Rendering:** Next.js pre-renders a page on each request. It will be slower because the page cannot be cached by a CDN, but the pre-rendered page will always be up-to-date. We'll talk about this approach below.

Server-side Rendering

> Also referred to as "SSR" or "Dynamic Rendering".

If a page uses **Server-side Rendering**, the page HTML is generated on **each request**.

To use Server-side Rendering for a page, you need to `export` an `async` function called `getServerSideProps`. This function will be called by the server on every request.

For example, suppose that your page needs to pre-render frequently updated data (fetched from an external API). You can write `getServerSideProps` which fetches this data and passes it to `Page` like below:

```
export default function Page({ data }) {
  // Render data...
}

// This gets called on every request
export async function getServerSideProps() {
  // Fetch data from external API
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  // Pass data to the page via props
  return { props: { data } }
}
```

As you can see, `getServerSideProps` is similar to `getStaticProps`, but the difference is that `getServerSideProps` is run on every request instead of on build time.

To learn more about how `getServerSideProps` works, check out our Data Fetching documentation

Summary

We've discussed two forms of pre-rendering for Next.js.

- **Static Generation (Recommended):** The HTML is generated at **build time** and will be reused on each request. To make a page use Static Generation, either export the page component, or export `getStaticProps` (and `getStaticPaths` if necessary). It's great for pages that can be pre-rendered ahead of a user's request. You can also use it with Client-side Rendering to bring in additional data.
- **Server-side Rendering:** The HTML is generated on **each request**. To make a page use Server-side Rendering, export `getServerSideProps`. Because Server-side Rendering results in slower performance than Static Generation, use this only if absolutely necessary.

Learn more

We recommend you to read the following sections next:

Data Fetching: Learn more about data fetching in Next.js.
Preview Mode: Learn more about the preview mode in Next.js.
Routing: Learn more about routing in Next.js.
TypeScript: Add TypeScript to your pages.

---

description: Optimize your third-party scripts with the built-in `next/script` component.

Optimizing Scripts

▶ **Examples**

The **Script component**, `next/script`, allows you to optimally load third-party scripts anywhere in your Next.js application. It is an extension of the HTML `<script>` element and enables you to choose between multiple loading strategies to fit your use case.

Overview

Websites often use third-party scripts to add functionality like analytics, ads, customer support widgets, and consent management. However, this can introduce problems that impact both user and developer experience:

- Some third-party scripts decrease loading performance and can degrade the user experience, especially if they are blocking the page content from being displayed.
- Developers are often unsure where and how to load third-party scripts in an application without impacting page performance.

Browsers load and execute `<script>` elements based on the order of placement in HTML and the usage of `async` and `defer` attributes. However, using the native `<script>` element creates some challenges:

- As your application grows in size and complexity, it becomes increasingly difficult to manage the loading order of third-party scripts.
- Streaming and Suspense improve page performance by rendering and hydrating new content as soon as possible, but `<script>` attributes (like `defer`) are incompatible without additional work.

The Script component solves these problems by providing a declarative API for loading third-party scripts. It provides a set of built-in loading strategies that can be used to optimize the loading sequence of scripts with support for streaming. Each of the strategies provided by the Script component uses the best possible combination of React and Web APIs to ensure that scripts are loaded with minimal impact to page performance.

Usage

To get started, import the `next/script` component:

```
import Script from 'next/script'
```

Page Scripts

To load a third-party script in a single route, import `next/script` and include the script directly in your page component:

```
import Script from 'next/script'

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

The script will only be fetched and executed when this specific page is loaded on the browser.

Application Scripts

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

```
import Script from 'next/script'

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

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.

> **Note**: You should rarely need to load a third-party script for every page of your application. We recommend only including third-party scripts in specific pages in order to minimize any unnecessary impact to performance.

Strategy

Although the default behavior of `next/script` allows you to load third-party scripts in any page, 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.

> **Note**: Once a `next/script` component has been loaded by the browser, it will stay in the DOM and client-side navigations won't re-execute the script.

Offloading Scripts To A Web Worker (experimental)

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

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

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

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

Then, run `next` (normally `npm run dev` or `yarn dev`) and Next.js will guide you through the installation of the required packages to finish the setup:

```
npm run dev

# You'll see instructions like these:
#
# Please install Partytown by running:
#
#         npm install @builder.io/partytown
#
# ...
```

Once setup is complete, defining `strategy="worker"` will automatically instantiate Partytown in your application and offload the script to a web worker.

```
import Script from 'next/script'

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

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" strategy="afterInteractive">
  {`document.getElementById('banner').classList.remove('hidden')`}
</Script>
```

Or by using the `dangerouslySetInnerHTML` property:

```
<Script
  id="show-banner"
  strategy="afterInteractive"
  dangerouslySetInnerHTML={{
    __html: `document.getElementById('banner').classList.remove('hidden')`,
  }}
/>
```

> **Note**: 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.

```
import Script from 'next/script'

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

Refer to the `next/script` API reference to learn more about each event handler and view examples.

Additional Attributes

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

```
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        id="example-script"
        nonce="XUENAJFW"
        data-test="script"
      />
    </>
  )
}
```

Next Steps

next/script API Reference View the API for the Script component.

---

description: Next.js allows you to serve static files, like images, in the public directory. You can learn how it works here.

Static File Serving

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

For example, if you add `me.png` inside `public`, the following code will access the image:

```
import Image from 'next/image'

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

export default Avatar
```

> **Note**: `next/image` requires Next.js 10 or later.

This folder is also useful for `robots.txt`, `favicon.ico`, Google Site Verification, and any other static files (including `.html`)!

> **Note**: Be sure the directory is named `public`. The name cannot be changed and is the only directory used to serve static assets.

> **Note**: Be sure to not have a static file with the same name as a file in the `pages/` directory, as this will result in an error.

> Read more: https://nextjs.org/docs/messages/conflicting-public-file-page

> **Note**: 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.

---

description: Browser support and which JavaScript features are supported by Next.js.

Supported Browsers and Features

Next.js supports **modern browsers** with zero configuration.

- Chrome 64+
- Edge 79+
- Firefox 67+
- Opera 51+
- Safari 12+

Browserslist

If you would like to target specific browsers or features, Next.js supports Browserslist configuration in your `package.json` file. Next.js uses the following Browserslist configuration by default:

```
{
  "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.

---

description: Next.js supports TypeScript by default and has built-in types for pages and the API. You can get started with TypeScript in Next.js here.

TypeScript

▶ **Version History**

Next.js provides an integrated TypeScript experience, including zero-configuration set up and built-in types for Pages, APIs, and more.

- Clone and deploy the TypeScript starter
- View an example application

`create-next-app` support

You can create a TypeScript project with `create-next-app` using the `--ts, --typescript` flag like so:

```
npx create-next-app@latest --ts
# or
yarn create next-app --typescript
# or
pnpm create next-app --ts
```

Existing projects

To get started in an existing project, create an empty `tsconfig.json` file in the root folder:

```
touch tsconfig.json
```

Next.js will automatically configure this file with default values. Providing your own `tsconfig.json` with custom compiler options is also supported.

You can also provide a relative path to a tsconfig.json file by setting `typescript.tsconfigPath` prop inside your `next.config.js` file.

Starting in `v12.0.0`, Next.js uses SWC by default to compile TypeScript and TSX for faster builds.

> Next.js will use Babel to handle TypeScript if `.babelrc` is present. This has some caveats and some compiler options are handled differently.

Then, run `next` (normally `npm run dev` or `yarn dev`) and Next.js will guide you through the installation of the required packages to finish the setup:

```
npm run dev

# You'll see instructions like these:
```

```
#
# Please install TypeScript, @types/react, and @types/node by running:
#
#         yarn add --dev typescript @types/react @types/node
#
# ...
```

You're now ready to start converting files from `.js` to `.tsx` and leveraging the benefits of TypeScript!

> A file named `next-env.d.ts` will be created at the root of your project. This file ensures Next.js types are picked up by the TypeScript compiler. **You should not remove it or edit it** as it can change at any time. This file should not be committed and should be ignored by version control (e.g. inside your `.gitignore` file).
>
> TypeScript `strict` mode is turned off by default. When you feel comfortable with TypeScript, it's recommended to turn it on in your `tsconfig.json`.
>
> Instead of editing `next-env.d.ts`, you can include additional types by adding a new file e.g. `additional.d.ts` and then referencing it in the `include` array in your `tsconfig.json`.

By default, Next.js will do type checking as part of `next build`. We recommend using code editor type checking during development.

If you want to silence the error reports, refer to the documentation for Ignoring TypeScript errors.

Static Generation and Server-side Rendering

For `getStaticProps`, `getStaticPaths`, and `getServerSideProps`, you can use the `GetStaticProps`, `GetStaticPaths`, and `GetServerSideProps` types respectively:

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

export const getStaticProps: GetStaticProps = async (context) => {
  // ...
}

export const getStaticPaths: GetStaticPaths = async () => {
  // ...
}

export const getServerSideProps: GetServerSideProps = async (context) => {
  // ...
}
```

> If you're using `getInitialProps`, you can follow the directions on this page.

API Routes

The following is an example of how to use the built-in types for API routes:

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

export default function handler(req: NextApiRequest, res: NextApiResponse) {
  res.status(200).json({ name: 'John Doe' })
}
```

You can also type the response data:

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

type Data = {
  name: string
}

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<Data>
) {
  res.status(200).json({ name: 'John Doe' })
}
```

Custom `App`

If you have a custom `App`, you can use the built-in type `AppProps` and change file name to `./pages/_app.tsx` like so:

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

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

Path aliases and baseUrl

Next.js automatically supports the `tsconfig.json` `"paths"` and `"baseUrl"` options.

You can learn more about this feature on the Module Path aliases documentation.

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.

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.

Ignoring TypeScript Errors

Next.js fails your **production build** (`next build`) when TypeScript errors are present in your project.

If you'd like Next.js to dangerously produce production code even when your application has errors, you can disable the built-in type checking step.

If disabled, be sure you are running type checks as part of your build or deploy process, otherwise this can be very dangerous.

Open `next.config.js` and enable the `ignoreBuildErrors` option in the `typescript` config:

```
module.exports = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```

---

description: Learn how to deploy your Next.js app to production, either managed or self-hosted.

Deployment

Congratulations, you are ready to deploy your Next.js application to production. This document 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:

```
{
  "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.

1. Install Docker on your machine
2. Clone the with-docker example
3. Build your container: `docker build -t nextjs-docker .`
4. 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

> **Note**: 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

> **Note**: 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.

Note: 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

Sometimes you might want to run some cleanup code on process signals like `SIGTERM` or `SIGINT`.

You can do that by setting the env variable `NEXT_MANUAL_SIG_HANDLE` to `true` and then register a handler for that signal inside your `_document.js` file. Please note that you need to register env variable directly in the system env variable, not in the `.env` file.

```
// package.json
{
  "scripts": {
    "dev": "NEXT_MANUAL_SIG_HANDLE=true 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)
  })
}
```

Related

For more information on what to do next, we recommend the following sections:

Going to Production: Ensure the best performance and user experience.

---

description: Get to know more about Next.js with the frequently asked questions.

Frequently Asked Questions

- ▶ Is Next.js production ready?
- ▶ How do I fetch data in Next.js?
- ▶ Why does Next.js have its own Router?
- ▶ Can I use Next.js with my favorite JavaScript library?
- ▶ Can I use Next.js with GraphQL?
- ▶ Can I use Next.js with Redux?
- ▶ Can I make a Next.js Progressive Web App (PWA)?
- ▶ Can I use a CDN for static assets?
- ▶ How can I change the internal webpack config?
- ▶ What is Next.js inspired by?

---

description: Get started with Next.js in the official documentation, and learn more about all our features!

> Next.js 13 was recently released, learn more and see the upgrade guide. Version 13 also introduces beta features like the `app` directory that works alongside the `pages` directory (stable) for incremental adoption. You can continue using `pages` in Next.js 13, but if you want to try the new `app` features, see the new beta docs.

Getting Started

Welcome to the Next.js documentation!

If you're new to Next.js, we recommend starting with the learn course. The interactive course with quizzes will guide you through everything you need to know to use Next.js.

If you have questions about anything related to Next.js, you're always welcome to ask our community on GitHub Discussions.

System Requirements

- Node.js 16.8.0 or newer
- MacOS, Windows (including WSL), and Linux are supported

Automatic Setup

We recommend creating a new Next.js app using `create-next-app`, which sets up everything automatically for you. (You don't need to create an empty directory. `create-next-app` will make one for you.) To create a project, run:

```
npx create-next-app@latest
# or
yarn create next-app
# or
pnpm create next-app
```

If you want to start with a TypeScript project you can use the `--typescript` flag:

```
npx create-next-app@latest --typescript
# or
yarn create next-app --typescript
# or
pnpm create next-app --typescript
```

After the installation is complete:

- Run `npm run dev` or `yarn dev` or `pnpm dev` to start the development server on `http://localhost:3000`
- Visit `http://localhost:3000` to view your application
- Edit `pages/index.js` and see the updated result in your browser

For more information on how to use `create-next-app`, you can review the `create-next-app` documentation.

Manual Setup

Install `next`, `react` and `react-dom` in your project:

```
npm install next react react-dom
# or
yarn add next react react-dom
# or
pnpm add next react react-dom
```

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

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

These scripts refer to the different stages of developing an application:

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

Create two directories `pages` and `public` at the root of your application:

- `pages` - Associated with a route based on their file name. For example, `pages/about.js` is mapped to `/about`
- `public` - Stores static assets such as images, fonts, etc. Files inside `public` directory can then be referenced by your code starting from the base URL (`/`).

Next.js is built around the concept of pages. A page is a React Component exported from a `.js`, `.jsx`, `.ts`, or `.tsx` file in the `pages` directory. You can even add dynamic route parameters with the filename.

Inside the `pages` directory, add the `index.js` file to get started. This is the page that is rendered when the user visits the root of your application.

Populate `pages/index.js` with the following contents:

```
function HomePage() {
  return <div>Welcome to Next.js!</div>
}

export default HomePage
```

After the set up is complete:

- Run `npm run dev` or `yarn dev` or `pnpm dev` to start the development server on `http://localhost:3000`
- Visit `http://localhost:3000` to view your application
- Edit `pages/index.js` and see the updated result in your browser

So far, we get:

- Automatic compilation and bundling
- React Fast Refresh
- Static generation and server-side rendering of `pages/`
- Static file serving through `public/` which is mapped to the base URL (`/`)

In addition, any Next.js application is ready for production from the start. Read more in our Deployment documentation.

Related

For more information on what to do next, we recommend the following sections:

Pages: Learn more about what pages are in Next.js.
CSS Support: Built-in CSS support to add custom styles to your app.
CLI: Learn more about the Next.js CLI.

---

description: Before taking your Next.js application to production, here are some recommendations to ensure the best user experience.

Going to Production

Before taking your Next.js application to production, here are some recommendations to ensure the best user experience.

In General

- Use caching wherever possible.
- Ensure your database and backend are deployed in the same region.
- Aim to ship the least amount of JavaScript possible.
- Defer loading heavy JavaScript bundles until needed.
- Ensure logging is set up.
- Ensure error handling is set up.
- Configure the 404 (Not Found) and 500 (Error) pages.
- Ensure you are measuring performance.
- Run Lighthouse to check for performance, best practices, accessibility, and SEO. For best results, use a production build of Next.js and use incognito in your browser so results aren't affected by extensions.
- Review Supported Browsers and Features.
- Improve performance using:
  - `next/image` and Automatic Image Optimization
  - Automatic Font Optimization
  - Script Optimization
- Improve loading performance

Caching

▼ Examples

- ssr-caching

Caching improves response times and reduces the number of requests to external services. Next.js automatically adds caching headers to immutable assets served from `/_next/static` including JavaScript, CSS, static images, and other media.

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

`Cache-Control` headers set in `next.config.js` will be overwritten in production to ensure that static assets can be cached effectively. If you need to revalidate the cache of a page that has been statically generated, you can do so by setting `revalidate` in the page's `getStaticProps` function. If you're using `next/image`, you can configure the `minimumCacheTTL` for the default Image Optimization loader.

> **Note**: When running your application locally with `next dev`, your headers are overwritten to prevent caching locally.

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
```

You can also use caching headers inside `getServerSideProps` and API Routes for dynamic responses. For example, using `stale-while-revalidate`.

```
// This value is considered fresh for ten seconds (s-maxage=10).
// If a request is repeated within the next 10 seconds, the previously
// cached value will still be fresh. If the request is repeated before 59 seconds,
// the cached value will be stale but still render (stale-while-revalidate=59).
//
// In the background, a revalidation request will be made to populate the cache
// with a fresh value. If you refresh the page, you will see the new value.
export async function getServerSideProps({ req, res }) {
  res.setHeader(
    'Cache-Control',
    'public, s-maxage=10, stale-while-revalidate=59'
  )

  return {
    props: {},
  }
}
```

By default, `Cache-Control` headers will be set differently depending on how your page fetches data.

- If the page uses `getServerSideProps` or `getInitialProps`, it will use the default `Cache-Control` header set by `next start` in order to prevent accidental caching of responses that cannot be cached. If you want a different cache behavior while using `getServerSideProps`, use `res.setHeader('Cache-Control', 'value_you_prefer')` inside of the function as shown above.
- If the page is using `getStaticProps`, it will have a `Cache-Control` header of `s-maxage=REVALIDATE_SECONDS, stale-while-revalidate`, or if `revalidate` is *not* used, `s-maxage=31536000, stale-while-revalidate` to cache for the maximum age possible.

  Note: Your deployment provider must support caching for dynamic responses. If you are self-hosting, you will need to add this logic yourself using a key/value store like Redis. If you are using Vercel, Edge Caching works without configuration.

Reducing JavaScript Size

▼ Examples

- with-dynamic-import

To reduce the amount of JavaScript sent to the browser, you can use the following tools to understand what is included inside each JavaScript bundle:

- Import Cost – Display the size of the imported package inside VSCode.
- Package Phobia – Find the cost of adding a new dev dependency to your project.
- Bundle Phobia - Analyze how much a dependency can increase bundle sizes.
- Webpack Bundle Analyzer – Visualize the size of webpack output files with an interactive, zoomable treemap.
- bundlejs - An online tool to quickly bundle & minify your projects, while viewing the compressed gzip/brotli bundle size, all running locally on your browser.

Each file inside your `pages/` directory will automatically be code split into its own JavaScript bundle during `next build`. You can also use Dynamic Imports to lazy-load components and libraries. For example, you might want to defer loading your modal code until a user clicks the open button.

Logging

▼ Examples

- Pino and Logflare Example

Since Next.js runs on both the client and server, there are multiple forms of logging supported:

- `console.log` in the browser
- `stdout` on the server

If you want a structured logging package, we recommend Pino. If you're using Vercel, there are pre-built logging integrations compatible with Next.js.

Error Handling

▼ Examples

- with-sentry

When an unhandled exception occurs, you can control the experience for your users with the 500 page. We recommend customizing this to your brand instead of the default Next.js theme.

You can also log and track exceptions with a tool like Sentry. This example shows how to catch & report errors on both the client and server-side, using the Sentry SDK for Next.js. There's also a Sentry integration for Vercel.

Loading Performance

To improve loading performance, you first need to determine what to measure and how to measure it. Core Web Vitals is a good industry standard that is measured using your own web browser. If you are not familiar with the metrics of Core Web Vitals, review this blog post and determine which specific metric/s will be your drivers for loading performance. Ideally, you would want to measure the loading performance in the following environments:

- In the lab, using your own computer or a simulator.
- In the field, using real-world data from actual visitors.
- Local, using a test that runs on your device.
- Remote, using a test that runs in the cloud.

Once you are able to measure the loading performance, use the following strategies to improve it iteratively so that you apply one strategy, measure the new performance and continue tweaking until you do not see much improvement. Then, you can move on to the next strategy.

- Use caching regions that are close to the regions where your database or API is deployed.
- As described in the caching section, use a `stale-while-revalidate` value that will not overload your backend.
- Use Incremental Static Regeneration to reduce the number of requests to your backend.
- Remove unused JavaScript. Review this blog post to understand what Core Web Vitals metrics bundle size affects and what strategies you can use to reduce it, such as:
  - Setting up your Code Editor to view import costs and sizes
  - Finding alternative smaller packages
  - Dynamically loading components and dependencies

Related

For more information on what to do next, we recommend the following sections:

Deployment: Take your Next.js application to production.

---

title: Building Forms with Next.js description: Learn how to create forms with Next.js, from the form HTML element to advanced concepts with React.

Building Forms with Next.js

A web form has a **client-server** relationship. They are used to send data handled by a web server for processing and storage. The form itself is the client, and the server is any storage mechanism that can be used to store, retrieve and send data when needed.

This guide will teach you how to create a web form with Next.js.

Part 1: HTML Form

HTML forms are built using the `<form>` tag. It takes a set of attributes and fields to structure the form for features like text fields, checkboxes, dropdown menus, buttons, radio buttons, etc.

Here's the syntax of an HTML form:

```
<!-- Basic HTML Form -->
<form action="/send-data-here" method="post">
  <label for="first">First name:</label>
  <input type="text" id="first" name="first" />
  <label for="last">Last name:</label>
  <input type="text" id="last" name="last" />
  <button type="submit">Submit</button>
</form>
```

The front-end looks like this:


html forms

The HTML `<form>` tag acts as a container for different `<input>` elements like `text` field and submit `button`. Let's study each of these elements:

- `action`: An attribute that specifies where the form data is sent when the form is submitted. It's generally a URL (an absolute URL or a relative URL).
- `method`: Specifies the HTTP method, i.e., `GET` or `POST` used to send data while submitting the form.
- `<label>`: An element that defines the label for other form elements. Labels aid accessibility, especially for screen readers.
- `<input>`: The form element that is widely used to structure the form fields. It depends significantly on the value of the `type` attribute. Input types can be `text`, `checkbox`, `email`, `radio`, and more.
- `<button>`: Represents a clickable button that's used to submit the form data.

Form Validation

A process that checks if the information provided by a user is correct or not. Form validation also ensures that the provided information is in the correct format (e.g. there's an @ in the email field). These are of two types:

- **Client-side**: Validation is done in the browser
- **Server-side**: Validation is done on the server

Though both of these types are equally important, this guide will focus on client-side validation only.

Client-side validation is further categorized as:

- **Built-in**: Uses HTML-based attributes like `required`, `type`, `minLength`, `maxLength`, `pattern`, etc.
- **JavaScript-based**: Validation that's coded with JavaScript.

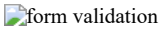
Built-in Form Validation Using `required`, `type`, `minLength`, `maxLength`

- `required`: Specifies which fields must be filled before submitting the form.
- `type`: Specifies the data's type (i.e a number, email address, string, etc).
- `minLength`: Specifies minimum length for the text data string.
- `maxLength`: Specifies maximum length for the text data string.

So, a form using this attributes may look like:

```
<!-- HTML Form with Built-in Validation -->
<form action="/send-data-here" method="post">
  <label for="roll">Roll Number</label>
  <input
    type="text"
    id="roll"
    name="roll"
    required
    minlength="10"
    maxlength="20"
  />
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" required />
  <button type="submit">Submit</button>
</form>
```

With these validation checks in place, when a user tries to submit an empty field for Name, it gives an error that pops right in the form field. Similarly, a roll number can only be entered if it's 10-20 characters long.



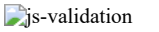
JavaScript-based Form Validation

Form Validation is important to ensure that a user has submitted the correct data, in a correct format. JavaScript offers an additional level of validation along with HTML native form attributes on the client side. Developers generally prefer validating form data through JavaScript because its data processing is faster when compared to server-side validation, however front-end validation may be less secure in some scenarios as a malicious user could always send malformed data to your server.

The following example shows using JavaScript to validate a form:

```
<form onsubmit="validateFormWithJS()">
  <label for="rollNumber">Roll Number:</label>
  <input type="text" name="rollNumber" id="rollNumber" />

  <label for="name">Name:</label>
  <input type="text" name="name" id="name" />

  <button type="submit">Submit</button>
</form>

<script>
  function validateFormWithJS() {
    const name = document.querySelector('#name').value
    const rollNumber = document.querySelector('#rollNumber').value

    if (!name) {
      alert('Please enter your name.')
      return false
    }

    if (rollNumber.length < 3) {
      alert('Roll Number should be at least 3 digits long.')
      return false
    }
  }
</script>
```

The HTML script tag is used to embed any client-side JavaScript. It can either contain inline scripting statements (as shown in the example above) or point to an external script file via the `src` attribute. This example validates the name and roll number of a user. The `validateFormWithJS()` function does not allow an empty name field, and the roll number must be at least three digits long. The validation is performed when you hit the Submit button. You are not redirected to the next page until the given values are correct.

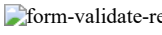


Form Validation Using Regular Expressions

JavaScript validation with Regular Expressions uses the `pattern` HTML attribute. A regular expression (commonly known as RegEx) is an object that describes a pattern of characters. You can only apply the `pattern` attribute to the `<input>` element. This way, you can validate the input value using Regular Expressions (RegEx) by defining your own rules. Once again, if the value does not match the defined pattern, the input will give an error. The below example shows using the `pattern` attribute on an `input` element:

```
<form action="/action_page.php">
  <label for="pswrd">Password:</label>
  <input
    type="password"
    id="pswrd"
    name="pswrd"
    pattern="[a-z0-9]{1,15}"
    title="Password should be digits (0 to 9) or alphabets (a to z)."
  />

  <button type="submit">Submit</button>
</form>
```

The password form field must only contain digits (0 to 9), lowercase alphabets (a to z) and it must be no more than 15 characters in length. No other characters (#,$,&, etc.) are allowed. The rule in RegEx is written as `[a-z0-9]{1,15}`.



> To learn more about HTML forms, check out the MDN Web Docs.

Part 2: Project Setup

In the following section you will be creating forms in React using Next.js.

Create a new Next.js app. You can use the create-next-app for a quick start. In your command line terminal, run the following:

```
npx create-next-app
```

Answer the questions to create your project, and give it a name, this example uses `next-forms`. Next `cd` into this directory, and run `npm run dev` or `yarn dev` command to start the development server.

Open the URL printed in the terminal to ensure that your app is running successfully.

Part 3: Setting up a Next.js Form API Route

Both the client and the server will be built using Next.js. For the server part, create an API endpoint where you will send the form data.

Next.js offers a file-based system for routing that's built on the concept of pages. Any file inside the folder `pages/api` is mapped to `/api/*` and will be treated as an API endpoint instead of a page. This API endpoint is going to be server-side only.

Go to `pages/api`, create a file called `form.js` and paste this code written in Node.js:

```
export default function handler(req, res) {
  // Get data submitted in request's body.
  const body = req.body

  // Optional logging to see the responses
  // in the command line where next.js app is running.
```

```
  console.log('body: ', body)

  // Guard clause checks for first and last name,
  // and returns early if they are not found
  if (!body.first || !body.last) {
    // Sends a HTTP bad request error code
    return res.status(400).json({ data: 'First or last name not found' })
  }

  // Found the name.
  // Sends a HTTP success code
  res.status(200).json({ data: `${body.first} ${body.last}` })
}
```

This form `handler` function will receive the request `req` from the client (i.e. submitted form data). And in return, it'll send a response `res` as JSON that will have both the first and the last name. You can access this API endpoint at `http://localhost:3000/api/form` or replace the localhost URL with an actual Vercel deployment when you deploy.

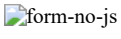
> Moreover, you can also attach this API to a database like MongoDB or Google Sheets. This way, your submitted form data will be securely stored for later use. For this guide, no database is used. Instead, the same data is returned to the user to demo how it's done.

Form Submission without JavaScript

You can now use `/api/form` relative endpoint inside the `action` attribute of the form. You are sending form data to the server when the form is submitted via `POST` HTTP method (which is used to send data).

```
<form action="/api/form" method="post">
  <label for="first">First name:</label>
  <input type="text" id="first" name="first" />
  <label for="last">Last name:</label>
  <input type="text" id="last" name="last" />
  <button type="submit">Submit</button>
</form>
```

If you submit this form, it will submit the data to the forms API endpoint `/api/form`. The server then responds, generally handling the data and loading the URL defined by the action attribute, causing a new page load. So in this case you'll be redirected to `http://localhost:3000/api/form` with the following response from the server.



Part 4: Configuring Forms in Next.js

You have created a Next.js API Route for form submission. Now it's time to configure the client (the form itself) inside Next.js using React. The first step will be extending your knowledge of HTML forms and converting it into React (using JSX).

Here's the same form in a React function component written using JSX.

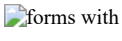
```
export default function Form() {
  return (
    <form action="/api/form" method="post">
      <label htmlFor="first">First Name</label>
      <input type="text" id="first" name="first" required />

      <label htmlFor="last">Last Name</label>
      <input type="text" id="last" name="last" required />

      <button type="submit">Submit</button>
    </form>
  )
}
```

Here's what changed:

- The `for` attribute is changed to `htmlFor`. (Since `for` is a keyword associated with the "for" loop in JavaScript, React elements use `htmlFor` instead.)
- The `action` attribute now has a relative URL which is the form API endpoint.

This completes the basic structure of your Next.js-based form.

> You can view the entire source code of next-forms example repo that we're creating here as a working example. Feel free to clone it and start right away. This demo is built with create-next-app, and you can preview the basic form CSS styles inside `/styles/global.css` file.



Part 5: Form Submission without JavaScript

JavaScript brings interactivity to our web applications, but sometimes you need to control the JavaScript bundle from being too large, or your sites visitors might have JavaScript disabled.

There are several reasons why users disable JavaScript:

- Addressing bandwidth constraints
- Increasing device (phone or laptop) battery life
- For privacy so they won't be tracked with analytical scripts

Regardless of the reason, disabling JavaScript will impact site functionality partially, if not completely.

Next open the `next-forms` directory. Inside the `/pages` directory, create a file `no-js-form.js`.

> **Quick Tip**: In Next.js, a page is a React Component exported from a `.js`, `.jsx`, `.ts`, or `.tsx` file in the pages directory. Each page is associated with a route based on its file name.
>
> Example: If you create `pages/no-js-form.js`, it will be accessible at `your-domain.tld/no-js-form`.

Let's use the same code from above:

```
export default function PageWithoutJSbasedForm() {
  return (
    <form action="/api/form" method="post">
      <label htmlFor="first">First Name</label>
      <input type="text" id="first" name="first" required />

      <label htmlFor="last">Last Name</label>
```

```
      <input type="text" id="last" name="last" required />

      <button type="submit">Submit</button>
    </form>
  )
}
```

With JavaScript disabled, when you hit the Submit button, an event is triggered, which collects the form data and sends it to our forms API endpoint as defined in the `action` attribute and using `POST` HTTP `method`. You'll be redirected to the `/api/form` endpoint since that's how form `action` works.

The form data will be submitted on the server as a request `req` to the form handler function written above. It will process the data and return a JSON string as a response `res` with your submitted name included.

> To improve the experience here, as a response you can redirect the user to a page and thank them for submitting the form.

Part 6: Form Submission with JavaScript Enabled

Inside `/pages`, you'll create another file called `js-form.js`. This will create a `/js-form` page on your Next.js app.

Now, as soon as the form is submitted, we prevent the form's default behavior of reloading the page. We'll take the form data, convert it to JSON string, and send it to our server, the API endpoint. Finally, our server will respond with the name submitted. All of this with a basic JavaScript `handleSubmit()` function.

Here's what this function looks like. It's well documented for you to understand each step:

```
export default function PageWithJSbasedForm() {
  // Handles the submit event on form submit.
  const handleSubmit = async (event) => {
    // Stop the form from submitting and refreshing the page.
    event.preventDefault()

    // Get data from the form.
    const data = {
      first: event.target.first.value,
      last: event.target.last.value,
    }

    // Send the data to the server in JSON format.
    const JSONdata = JSON.stringify(data)

    // API endpoint where we send form data.
    const endpoint = '/api/form'

    // Form the request for sending data to the server.
    const options = {
      // The method is POST because we are sending data.
      method: 'POST',
      // Tell the server we're sending JSON.
      headers: {
        'Content-Type': 'application/json',
      },
      // Body of the request is the JSON data we created above.
      body: JSONdata,
    }

    // Send the form data to our forms API on Vercel and get a response.
    const response = await fetch(endpoint, options)

    // Get the response data from server as JSON.
    // If server returns the name submitted, that means the form works.
    const result = await response.json()
    alert(`Is this your full name: ${result.data}`)
  }
  return (
    // We pass the event to the handleSubmit() function on submit.
    <form onSubmit={handleSubmit}>
      <label htmlFor="first">First Name</label>
      <input type="text" id="first" name="first" required />

      <label htmlFor="last">Last Name</label>
      <input type="text" id="last" name="last" required />

      <button type="submit">Submit</button>
    </form>
  )
}
```

It's a Next.js page with a React function component called `PageWithJSbasedForm` with a `<form>` element written in JSX. There's no action on the `<form>` element. Instead, we use the `onSubmit` event handler to send data to our `{handleSubmit}` function.

The `handleSubmit()` function processes your form data through a series of steps:

- The `event.preventDefault()` stops the `<form>` element from refreshing the entire page.
- We created a JavaScript object called `data` with the `first` and `last` values from the form.
- JSON is a language-agnostic data transfer format. So we use `JSON.stringify(data)` to convert the data to JSON.
- We then use `fetch()` to send the data to our `/api/form` endpoint using JSON and HTTP `POST` method.
- Server sends back a response with the name submitted. Woohoo! 🥳

Conclusion

This guide has covered the following:

- The basic HTML `form` element
- Understanding forms with React.js
- Validating forms data with and without JavaScript
- Using Next.js API Routes to handle `req` and `res` from the client and server

For more details go through Next.js Learn Course.

---

description: Learn how to transition an existing Create React App project to Next.js.

Migrating from Create React App

This guide will help you understand how to transition from an existing non-ejected Create React App project to Next.js. Migrating to Next.js will allow you to:

- Choose which data fetching strategy you want on a per-page basis.
- Use Incremental Static Regeneration to update *existing* pages by re-rendering them in the background as traffic comes in.
- Use API Routes.

And more! Let's walk through a series of steps to complete the migration.

Updating `package.json` and dependencies

The first step towards migrating to Next.js is to update `package.json` and dependencies. You should:

- Remove `react-scripts` (but keep `react` and `react-dom`). If you're using React Router, you can also remove `react-router-dom`.
- Install `next`.
- Add Next.js related commands to `scripts`. One is `next dev`, which runs a development server at `localhost:3000`. You should also add `next build` and `next start` for creating and starting a production build.

Here's an example `package.json`:

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start"
  },
  "dependencies": {
    "next": "latest",
    "react": "latest",
    "react-dom": "latest"
  }
}
```

Static Assets and Compiled Output

Create React App uses the `public` directory for the entry HTML file as well as static assets, but Next.js only uses it for static assets. When migrating from Create React App, the location of the `public` directory remains the same.

- Move any images, fonts, or other static assets to `public`.
- Convert `index.html` (the entry point of your application) to Next.js. Any `<head>` code should be moved to a custom `_document.js`. Any shared layout between all pages should be moved to a custom `_app.js`.
- See Styling for CSS/Sass files.
- Add `.next` to `.gitignore`.

Creating Routes & Linking

With Create React App, you're likely using React Router. Instead of using a third-party library, Next.js includes its own file-system based routing.

- Create a `pages` directory at the root of your project.
- Then, move the `src/App.js` file to `pages/index.js`. This file is the index page of your Next.js application. Populate this file with code that is used to display the index route in your Create React App.
- Convert all other `Route` components to new files in the `pages` directory.
- For routes that require dynamic content (e.g. `/blog/:slug`), you can use Dynamic Routes with Next.js (e.g. `pages/blog/[slug].js`). The value of `slug` is accessible through a query parameter. For example, the route `/blog/first-post` would forward the query object `{ 'slug': 'first-post' }` to `pages/blog/[slug].js` (learn more here).

For more information, see Migrating from React Router.

Styling

Next.js has built-in support for CSS, Sass and CSS-in-JS.

With Create React App, you can import `.css` files directly inside React components. Next.js allows you to do the same, but requires these files to be CSS Modules. For global styles, you'll need a custom `_app.js` to add a global stylesheet.

Safely Accessing Web APIs

With client-side rendered applications (like Create React App), you can access `window`, `localStorage`, `navigator`, and other Web APIs out of the box.

Since Next.js uses pre-rendering, you'll need to safely access those Web APIs only when you're on the client-side. For example, the following code snippet will allow access to `window` only on the client-side.

```
if (typeof window !== 'undefined') {
  // You now have access to `window`
}
```

A recommended way of accessing Web APIs safely is by using the `useEffect` hook, which only executes client-side:

```
import { useEffect } from 'react'

useEffect(() => {
  // You now have access to `window`
}, [])
```

Image Component and Image Optimization

Since version **10.0.0**, Next.js has a built-in Image Component and Automatic Image Optimization.

The Next.js Image Component, `next/image`, is an extension of the HTML `<img>` element, evolved for the modern web.

The Automatic Image Optimization allows for resizing, optimizing, and serving images in modern formats like WebP when the browser supports it. This avoids shipping large images to devices with a smaller viewport. It also allows Next.js to automatically adopt future image formats and serve them to browsers that support those formats.

Instead of optimizing images at build time, Next.js optimizes images on-demand, as users request them. Your build times aren't increased, whether shipping 10 images or 10 million images.

```
import Image from 'next/image'

export default function Home() {
  return (
    <>
      <h1>My Homepage</h1>
      <Image
        src="/me.png"
        alt="Picture of the author"
        width={500}
        height={500}
      />
      <p>Welcome to my homepage!</p>
    </>
  )
}
```

Environment Variables

Next.js has support for `.env` Environment Variables similar to Create React App. The main difference is the prefix used to expose environment variables on the client-side.

- Change all environment variables with the `REACT_APP_` prefix to `NEXT_PUBLIC_`.
- Server-side environment variables will be available at build-time and in API Routes.

Search Engine Optimization

Most Create React App examples use `react-helmet` to assist with adding `meta` tags for proper SEO. With Next.js, we use `next/head` to add `meta` tags to your `<head />` element. For example, here's an SEO component with Create React App:

```
// src/components/seo.js

import { Helmet } from 'react-helmet'

export default function SEO({ description, title, siteTitle }) {
  return (
    <Helmet
      title={title}
      titleTemplate={siteTitle ? `%s | ${siteTitle}` : null}
      meta={[
        {
          name: `description`,
          content: description,
        },
        {
          property: `og:title`,
          content: title,
        },
        {
          property: `og:description`,
          content: description,
        },
        {
          property: `og:type`,
          content: `website`,
        },
        {
          name: `twitter:card`,
          content: `summary`,
        },
        {
          name: `twitter:creator`,
          content: twitter,
        },
        {
          name: `twitter:title`,
          content: title,
        },
        {
          name: `twitter:description`,
          content: description,
        },
      ]}
    />
  )
}
```

And here's the same example using Next.js.

```
// src/components/seo.js

import Head from 'next/head'

export default function SEO({ description, title, siteTitle }) {
  return (
    <Head>
      <title>{`${title} | ${siteTitle}`}</title>
      <meta name="description" content={description} />
      <meta property="og:type" content="website" />
      <meta property="og:title" content={title} />
      <meta property="og:description" content={description} />
      <meta property="og:site_name" content={siteTitle} />
      <meta property="twitter:card" content="summary" />
      <meta property="twitter:creator" content={config.social.twitter} />
      <meta property="twitter:title" content={title} />
      <meta property="twitter:description" content={description} />
    </Head>
  )
}
```

Single-Page App (SPA)

If you want to move your existing Create React App to Next.js and keep a Single-Page App, you can move your old application's entry point to an Optional Catch-All Route named `pages/[[...app]].js`.

```
// pages/[[...app]].js

import { useState, useEffect } from 'react'
import CreateReactAppEntryPoint from '../components/app'

function App() {
  const [isMounted, setIsMounted] = useState(false)

  useEffect(() => {
    setIsMounted(true)
  }, [])

  if (!isMounted) {
    return null
  }

  return <CreateReactAppEntryPoint />
}

export default App
```

Ejected Create React App

If you've ejected Create React App, here are some things to consider:

- If you have custom file loaders set up for CSS, Sass, or other assets, this is all built-in with Next.js.
- If you've manually added new JavaScript features (e.g. Optional Chaining) or Polyfills, check to see what's included by default with Next.js.
- If you have a custom code splitting setup, you can remove that. Next.js has automatic code splitting on a per-page basis.
- You can customize your PostCSS setup with Next.js without ejecting from the framework.
- You should reference the default Babel config and Webpack config of Next.js to see what's included by default.

Learn More

You can learn more about Next.js by completing our starter tutorial. If you have questions or if this guide didn't work for you, feel free to reach out to our community on GitHub Discussions.

---

description: Learn how to transition an existing Gatsby project to Next.js.

Migrating from Gatsby

This guide will help you understand how to transition from an existing Gatsby project to Next.js. Migrating to Next.js will allow you to:

- Choose which data fetching strategy you want on a per-page basis.
- Use Incremental Static Regeneration to update *existing* pages by re-rendering them in the background as traffic comes in.
- Use API Routes.

And more! Let's walk through a series of steps to complete the migration.

Updating `package.json` and dependencies

The first step towards migrating to Next.js is to update `package.json` and dependencies. You should:

- Remove all Gatsby-related packages (but keep `react` and `react-dom`).
- Install `next`.
- Add Next.js related commands to `scripts`. One is `next dev`, which runs a development server at `localhost:3000`. You should also add `next build` and `next start` for creating and starting a production build.

Here's an example `package.json` (view diff):

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start"
  },
  "dependencies": {
    "next": "latest",
    "react": "latest",
    "react-dom": "latest"
  }
}
```

Static Assets and Compiled Output

Gatsby uses the `public` directory for the compiled output, whereas Next.js uses it for static assets. Here are the steps for migration (view diff):

- Remove `.cache/` and `public` from `.gitignore` and delete both directories.
- Rename Gatsby's `static` directory as `public`.
- Add `.next` to `.gitignore`.

Creating Routes

Both Gatsby and Next support a `pages` directory, which uses file-system based routing. Gatsby's directory is `src/pages`, which is also supported by Next.js.

Gatsby creates dynamic routes using the `createPages` API inside of `gatsby-node.js`. With Next, we can use Dynamic Routes inside of `pages` to achieve the same effect. Rather than having a `template` directory, you can use the React component inside your dynamic route file. For example:

- **Gatsby:** `createPages` API inside `gatsby-node.js` for each blog post, then have a template file at `src/templates/blog-post.js`.
- **Next:** Create `pages/blog/[slug].js` which contains the blog post template. The value of `slug` is accessible through a query parameter. For example, the route `/blog/first-post` would forward the query object `{ 'slug': 'first-post' }` to `pages/blog/[slug].js` (learn more here).

Styling

With Gatsby, global CSS imports are included in `gatsby-browser.js`. With Next, you should create a custom `_app.js` for global CSS. When migrating, you can copy over your CSS imports directly and update the relative file path, if necessary. Next.js has built-in CSS support.

Links

The Gatsby `Link` and Next.js `Link` component have a slightly different API.

```
// Gatsby

import { Link } from 'gatsby'

export default function Home() {
  return <Link to="/blog">blog</Link>
}

// Next.js

import Link from 'next/link'

export default function Home() {
  return <Link href="/blog">blog</Link>
}
```

Update any import statements, switch `to` to `href`.

Data Fetching

The largest difference between Gatsby and Next.js is how data fetching is implemented. Gatsby is opinionated with GraphQL being the default strategy for retrieving data across your application. With Next.js, you get to choose which strategy you want (GraphQL is one supported option).

Gatsby uses the `graphql` tag to query data in the pages of your site. This may include local data, remote data, or information about your site configuration. Gatsby only allows the creation of static pages. With Next.js, you can choose on a per-page basis which data fetching strategy you want. For example, `getServerSideProps` allows you to do server-side rendering. If you wanted to generate a static page, you'd export `getStaticProps` / `getStaticPaths` inside the page, rather than using `pageQuery`. For example:

```
// src/pages/[slug].js

// Install remark and remark-html
import { remark } from 'remark'
import html from 'remark-html'
import { getPostBySlug, getAllPosts } from '../lib/blog'

export async function getStaticProps({ params }) {
  const post = getPostBySlug(params.slug)
  const markdown = await remark()
    .use(html)
    .process(post.content || '')
  const content = markdown.toString()

  return {
    props: {
      ...post,
      content,
    },
  }
}

export async function getStaticPaths() {
  const posts = getAllPosts()

  return {
    paths: posts.map((post) => {
      return {
        params: {
          slug: post.slug,
        },
      }
    }),
    fallback: false,
  }
}
```

You'll commonly see Gatsby plugins used for reading the file system (`gatsby-source-filesystem`), handling markdown files (`gatsby-transformer-remark`), and so on. For example, the popular starter blog example has 15 Gatsby specific packages. Next takes a different approach. It includes common features directly inside the framework, and gives the user full control over integrations with external packages. For example, rather than abstracting reading from the file system to a plugin, you can use the native Node.js `fs` package inside `getStaticProps` / `getStaticPaths` to read from the file system.

```
// src/lib/blog.js

// Install gray-matter and date-fns
import matter from 'gray-matter'
import { parseISO, format } from 'date-fns'
import fs from 'fs'
import { join } from 'path'

// Add markdown files in `src/content/blog`
const postsDirectory = join(process.cwd(), 'src', 'content', 'blog')

export function getPostBySlug(slug) {
  const realSlug = slug.replace(/\.md$/, '')
  const fullPath = join(postsDirectory, `${realSlug}.md`)
  const fileContents = fs.readFileSync(fullPath, 'utf8')
  const { data, content } = matter(fileContents)
  const date = format(parseISO(data.date), 'MMMM dd, yyyy')

  return { slug: realSlug, frontmatter: { ...data, date }, content }
}

export function getAllPosts() {
  const slugs = fs.readdirSync(postsDirectory)
  const posts = slugs.map((slug) => getPostBySlug(slug))

  return posts
}
```

Image Component and Image Optimization

Next.js has a built-in Image Component and Automatic Image Optimization.

The Next.js Image Component, `next/image`, is an extension of the HTML `<img>` element, evolved for the modern web.

The Automatic Image Optimization allows for resizing, optimizing, and serving images in modern formats like WebP when the browser supports it. This avoids shipping large images to devices with a smaller viewport. It also allows Next.js to automatically adopt future image formats and serve them to browsers that support those formats.

Migrating from Gatsby Image

Instead of optimizing images at build time, Next.js optimizes images on-demand, as users request them. Unlike static site generators and static-only solutions, your build times aren't increased, whether shipping 10 images or 10 million images.

This means you can remove common Gatsby plugins like:

- `gatsby-image`
- `gatsby-transformer-sharp`
- `gatsby-plugin-sharp`

Instead, use the built-in `next/image` component and Automatic Image Optimization.

> The `next/image` component's default loader is not supported when using `output: 'export'`. However, other loader options will work.

```
import Image from 'next/image'
import profilePic from '../assets/me.png'

export default function Home() {
  return (
    <>
      <h1>My Homepage</h1>
      <Image
        src={profilePic}
        alt="Picture of the author"
        // When "responsive", similar to "fluid" from Gatsby
        // When "intrinsic", similar to "fluid" with maxWidth from Gatsby
        // When "fixed", similar to "fixed" from Gatsby
        layout="responsive"
        // Optional, similar to "blur-up" from Gatsby
        placeholder="blur"
        // Optional, similar to "width" in Gatsby GraphQL
        width={500}
        // Optional, similar to "height" in Gatsby GraphQL
        height={500}
      />
      <p>Welcome to my homepage!</p>
    </>
  )
}
```

Site Configuration

With Gatsby, your site's metadata (name, description, etc.) is located inside `gatsby-config.js`. This is then exposed through the GraphQL API and consumed through a `pageQuery` or a static query inside a component.

With Next.js, we recommend creating a config file similar to below. You can then import this file anywhere without having to use GraphQL to access your site's metadata.

```
// src/config.js

export default {
  title: 'Starter Blog',
  author: {
    name: 'Lee Robinson',
    summary: 'who loves Next.js.',
  },
  description: 'A starter blog converting Gatsby -> Next.',
  social: {
    twitter: 'leeerob',
  },
}
```

Search Engine Optimization

Most Gatsby examples use `react-helmet` to assist with adding `meta` tags for proper SEO. With Next.js, we use `next/head` to add `meta` tags to your `<head />` element. For example, here's an SEO component with Gatsby:

```
// src/components/seo.js

import { Helmet } from 'react-helmet'

export default function SEO({ description, title, siteTitle }) {
  return (
    <Helmet
      title={title}
      titleTemplate={siteTitle ? `%s | ${siteTitle}` : null}
      meta={[
        {
          name: `description`,
          content: description,
        },
        {
          property: `og:title`,
          content: title,
        },
        {
          property: `og:description`,
          content: description,
        },
        {
          property: `og:type`,
          content: `website`,
        },
        {
```

```
          name: `twitter:card`,
          content: `summary`,
        },
        {
          name: `twitter:creator`,
          content: twitter,
        },
        {
          name: `twitter:title`,
          content: title,
        },
        {
          name: `twitter:description`,
          content: description,
        },
      ]}
    />
  )
}
```

And here's the same example using Next.js, including reading from a site config file.

```
// src/components/seo.js

import Head from 'next/head'
import config from '../config'

export default function SEO({ description, title }) {
  const siteTitle = config.title

  return (
    <Head>
      <title>{`${title} | ${siteTitle}`}</title>
      <meta name="description" content={description} />
      <meta property="og:type" content="website" />
      <meta property="og:title" content={title} />
      <meta property="og:description" content={description} />
      <meta property="og:site_name" content={siteTitle} />
      <meta property="twitter:card" content="summary" />
      <meta property="twitter:creator" content={config.social.twitter} />
      <meta property="twitter:title" content={title} />
      <meta property="twitter:description" content={description} />
    </Head>
  )
}
```

Learn more

Take a look at this pull request for more details on how an app can be migrated from Gatsby to Next.js. If you have questions or if this guide didn't work for you, feel free to reach out to our community on GitHub Discussions.

---

description: Learn how to migrate from React Router to file-system based routes with Next.js.

Migrating from React Router

This guide will help you understand how to transition from React Router to file-system based routes with Next.js. Using `next/link` and `next/router` will allow you to:

- Decrease bundle size by removing React Router as a dependency.
- Define your application routes through the file system.
- Utilize the latest improvements to the Next.js framework.

Basics

First, uninstall React Router. You'll be migrating to the built-in routing with Next.js.

```
npm uninstall react-router-dom
```

The `Link` component for performing client-side route transitions is slightly different from React Router.

```
// Before (React Router)
import { Link } from 'react-router-dom'

export default function App() {
  return <Link to="/about">About</Link>
}

// After (Next.js)
import Link from 'next/link'

export default function App() {
  return (
    <Link href="/about">
      About
    </Link>
  )
}
```

Most React applications that use React Router have a top-level navigation file, containing a list of routes. For example:

```
import { BrowserRouter as Router, Switch, Route } from 'react-router-dom'

export default function App() {
  return (
    <Router>
      <Switch>
        <Route path="/about">
          <h1>About</h1>
        </Route>
        <Route path="/blog">
          <h1>Blog</h1>
        </Route>
        <Route path="/">
```

```
          <h1>Home</h1>
        </Route>
      </Switch>
    </Router>
  )
}
```

With Next.js, you can express the same application structure in the file system. When a file is added to the `pages` directory it's automatically available as a route.

- `pages/about.js` → `/about`
- `pages/blog.js` → `/blog`
- `pages/index.js` → `/`

Nested Routes

In the example below, routes like `/blog/my-post` would render the `Post` component. If a slug was not provided, it would render the list of all blog posts.

```
import {
  BrowserRouter as Router,
  Switch,
  Route,
  useRouteMatch,
  useParams,
} from 'react-router-dom'

export default function Blog() {
  // Nested route under /blog
  const match = useRouteMatch()

  return (
    <Router>
      <Switch>
        <Route path={`${match.path}/:slug`}>
          <Post />
        </Route>
        <Route path={match.path}>
          <h1>All Blog Posts</h1>
        </Route>
      </Switch>
    </Router>
  )
}

function Post() {
  const { slug } = useParams()
  return <h1>Post Slug: {slug}</h1>
}
```

Rather than using the `:slug` syntax inside your `Route` component, Next.js uses the `[slug]` syntax in the file name for Dynamic Routes. We can transform this to Next.js by creating two new files, `pages/blog/index.js` (showing all pages) and `pages/blog/[slug].js` (showing an individual post).

```
// pages/blog/index.js

export default function Blog() {
  return <h1>All Blog Posts</h1>
}

// pages/blog/[slug].js

import { useRouter } from 'next/router'

export default function Post() {
  const router = useRouter()
  const { slug } = router.query

  return <h1>Post Slug: {slug}</h1>
}
```

Server Rendering

Next.js has built-in support for Server-side Rendering. This means you can remove any instances of `StaticRouter` in your code.

Code Splitting

Next.js has built-in support for Code Splitting. This means you can remove any instances of:

- `@loadable/server`, `@loadable/babel-plugin`, and `@loadable/webpack-plugin`
- Modifications to your `.babelrc` for `@loadable/babel-plugin`

Each file inside your `pages/` directory will be code split into its own JavaScript bundle during the build process. Next.js also supports ES2020 dynamic `import()` for JavaScript. With it you can import JavaScript modules dynamically and work with them. They also work with SSR.

For more information, read about Dynamic Imports.

Scroll Restoration

Next.js has built-in support for Scroll Restoration. This means you can remove any custom `ScrollToTop` components you have defined.

The default behavior of `next/link` and `next/router` is to scroll to the top of the page. You can also disable this if you prefer.

Learn More

For more information on what to do next, we recommend the following sections:

Routing: Learn more about routing in Next.js.
Dynamic Routes: Learn more about the built-in dynamic routes.
Pages: Enable client-side transitions with next/link.

description: Learn different strategies for incrementally adopting Next.js into your development workflow.

Incrementally Adopting Next.js

▶ **Examples**

Next.js has been designed for gradual adoption. With Next.js, you can continue using your existing code and add as much (or as little) React as you need. By starting small and incrementally adding more pages, you can prevent derailing feature work by avoiding a complete rewrite.

Strategies

Subpath

The first strategy is to configure your server or proxy such that, everything under a specific subpath points to a Next.js app. For example, your existing website might be at `example.com`, and you might configure your proxy such that `example.com/store` serves a Next.js e-commerce store.

Using `basePath`, you can configure your Next.js application's assets and links to automatically work with your new subpath `/store`. Since each page in Next.js is its own standalone route, pages like `pages/products.js` will route to `example.com/store/products` in your application.

```
// next.config.js

module.exports = {
  basePath: '/store',
}
```

To learn more about `basePath`, take a look at our documentation.

Rewrites

The second strategy is to create a new Next.js app that points to the root URL of your domain. Then, you can use `rewrites` inside `next.config.js` to have some subpaths to be proxied to your existing app.

For example, let's say you created a Next.js app to be served from `example.com` with the following `next.config.js`. Now, requests for the pages you've added to this Next.js app (e.g. `/about` if you've added `pages/about.js`) will be handled by Next.js, and requests for any other route (e.g. `/dashboard`) will be proxied to `proxy.example.com`.

> **Note**: If you use fallback: true/'blocking' in `getStaticPaths`, the catch-all fallback `rewrites` defined in `next.config.js` will not be run. They are instead caught by the `getStaticPaths` fallback.

```
// next.config.js

module.exports = {
  async rewrites() {
    return {
      // After checking all Next.js pages (including dynamic routes)
      // and static files we proxy any other requests
      fallback: [
        {
          source: '/:path*',
          destination: `https://proxy.example.com/:path*`,
        },
      ],
    }

    // For versions of Next.js < v10.1 you can use a no-op rewrite instead
    return [
      // we need to define a no-op rewrite to trigger checking
      // all pages/static files before we attempt proxying
      {
        source: '/:path*',
        destination: '/:path*',
      },
      {
        source: '/:path*',
        destination: `https://proxy.example.com/:path*`,
      },
    ]
  },
}
```

To learn more about rewrites, take a look at our documentation.

> **Note**: If you are incrementally migrating to a dynamic route (e.g. `[slug].js`) and using `fallback: true` or `fallback: 'blocking'` along with a fallback `rewrite`, ensure you consider the case where pages are not found. When Next.js matches the dynamic route it stops checking any further routes. Using `notFound: true` in `getStaticProps` will return the 404 page without applying the fallback `rewrite`. If this is not desired, you can use `getServerSideProps` with `stale-while-revalidate` Cache-Control headers when returning your props. Then, you can *manually* proxy to your existing backend using something like http-proxy instead of returning `notFound: true`.

Micro-Frontends with Monorepos and Subdomains

Next.js and Vercel make it straightforward to adopt micro frontends and deploy as a monorepo. This allows you to use subdomains to adopt new applications incrementally. Some benefits of micro-frontends:

- Smaller, more cohesive and maintainable codebases.
- More scalable organizations with decoupled, autonomous teams.
- The ability to upgrade, update, or even rewrite parts of the frontend in a more incremental fashion.

Once your monorepo is set up, push changes to your Git repository as usual and you'll see the commits deployed to the Vercel projects you've connected.

Conclusion

To learn more, read about subpaths and rewrites or deploy a Next.js monorepo.

---

description: Dynamic Routes are pages that allow you to add custom params to your URLs. Start creating Dynamic Routes and learn more here.

Dynamic Routes

▶ **Examples**

Defining routes by using predefined paths is not always enough for complex applications. In Next.js you can add brackets to a page (`[param]`) to create a dynamic route (a.k.a. url slugs, pretty urls, and others).

Consider the following page `pages/post/[pid].js`:

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

const Post = () => {
  const router = useRouter()
  const { pid } = router.query

  return <p>Post: {pid}</p>
}

export default Post
```

Any route like `/post/1`, `/post/abc`, etc. will be matched by `pages/post/[pid].js`. The matched path parameter will be sent as a query parameter to the page, and it will be merged with the other query parameters.

For example, the route `/post/abc` will have the following `query` object:

```
{ "pid": "abc" }
```

Similarly, the route `/post/abc?foo=bar` will have the following `query` object:

```
{ "foo": "bar", "pid": "abc" }
```

However, route parameters will override query parameters with the same name. For example, the route `/post/abc?pid=123` will have the following `query` object:

```
{ "pid": "abc" }
```

Multiple dynamic route segments work the same way. The page `pages/post/[pid]/[comment].js` will match the route `/post/abc/a-comment` and its `query` object will be:

```
{ "pid": "abc", "comment": "a-comment" }
```

Client-side navigations to dynamic routes are handled with `next/link`. If we wanted to have links to the routes used above it will look like this:

```
import Link from 'next/link'

function Home() {
  return (
    <ul>
      <li>
        <Link href="/post/abc">Go to pages/post/[pid].js</Link>
      </li>
      <li>
        <Link href="/post/abc?foo=bar">Also goes to pages/post/[pid].js</Link>
      </li>
      <li>
        <Link href="/post/abc/a-comment">
          Go to pages/post/[pid]/[comment].js
        </Link>
      </li>
    </ul>
  )
}

export default Home
```

Read our docs for Linking between pages to learn more.

Catch all routes

▶ **Examples**

Dynamic routes can be extended to catch all paths by adding three dots (`...`) inside the brackets. For example:

- `pages/post/[...slug].js` not only matches `/post/a`, but also `/post/a/b`, `/post/a/b/c`, and so on.
- `pages/post/[...slug].js` does not match `/post`.

> **Note**: You can use names other than `slug`, such as: `[...param]`

Matched parameters will be sent as a query parameter (`slug` in the example) to the page, and it will always be an array, so, the path `/post/a` will have the following `query` object:

```
{ "slug": ["a"] }
```

And in the case of `/post/a/b`, and any other matching path, new parameters will be added to the array, like so:

```
{ "slug": ["a", "b"] }
```

Optional catch all routes

Catch all routes can be made optional by including the parameter in double brackets (`[[...slug]]`).

For example, `pages/post/[[...slug]].js` will match `/post`, `/post/a`, `/post/a/b`, and so on.

The main difference between catch all and optional catch all routes is that with optional, the route without the parameter is also matched (`/post` in the example above).

The `query` objects are as follows:

```
{ } // GET `/post` (empty object)
{ "slug": ["a"] } // `GET /post/a` (single-element array)
{ "slug": ["a", "b"] } // `GET /post/a/b` (multi-element array)
```

Caveats

- Predefined routes take precedence over dynamic routes, and dynamic routes over catch all routes. Take a look at the following examples:
  - `pages/post/create.js` - Will match `/post/create`
  - `pages/post/[pid].js` - Will match `/post/1`, `/post/abc`, etc. But not `/post/create`
  - `pages/post/[...slug].js` - Will match `/post/1/2`, `/post/a/b/c`, etc. But not `/post/create`, `/post/abc`
- Pages that are statically optimized by Automatic Static Optimization will be hydrated without their route parameters provided, i.e `query` will be an empty object (`{}`).

  After hydration, Next.js will trigger an update to your application to provide the route parameters in the `query` object.

Related

For more information on what to do next, we recommend the following sections:

next/link: Enable client-side transitions with next/link.
Routing: Learn more about routing in Next.js.

---

description: Client-side navigations are also possible using the Next.js Router instead of the Link component. Learn more here.

Imperatively

▶ **Examples**

`next/link` should be able to cover most of your routing needs, but you can also do client-side navigations without it, take a look at the documentation for `next/router`.

The following example shows how to do basic page navigations with `useRouter`:

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

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

  return (
    <button onClick={() => router.push('/about')}>
      Click here to read more
    </button>
  )
}
```

---

description: Next.js has a built-in, opinionated, and file-system based Router. You can learn how it works here.

Routing

Next.js has a file-system based router built on the concept of pages.

When a file is added to the `pages` directory, it's automatically available as a route.

The files inside the `pages` directory can be used to define most common patterns.

Index routes

The router will automatically route files named `index` to the root of the directory.

- `pages/index.js` → `/`
- `pages/blog/index.js` → `/blog`

Nested routes

The router supports nested files. If you create a nested folder structure, files will automatically be routed in the same way still.

- `pages/blog/first-post.js` → `/blog/first-post`
- `pages/dashboard/settings/username.js` → `/dashboard/settings/username`

Dynamic route segments

To match a dynamic segment, you can use the bracket syntax. This allows you to match named parameters.

- `pages/blog/[slug].js` → `/blog/:slug` (`/blog/hello-world`)
- `pages/[username]/settings.js` → `/:username/settings` (`/foo/settings`)
- `pages/post/[...all].js` → `/post/*` (`/post/2020/id/title`)

  Check out the Dynamic Routes documentation to learn more about how they work.

Linking between pages

The Next.js router allows you to do client-side route transitions between pages, similar to a single-page application.

A React component called `Link` is provided to do this client-side route transition.

```
import Link from 'next/link'

function Home() {
  return (
    <ul>
      <li>
        <Link href="/">Home</Link>
      </li>
      <li>
        <Link href="/about">About Us</Link>
```

```
      </li>
      <li>
        <Link href="/blog/hello-world">Blog Post</Link>
      </li>
    </ul>
  )
}

export default Home
```

The example above uses multiple links. Each one maps a path (`href`) to a known page:

- `/` → `pages/index.js`
- `/about` → `pages/about.js`
- `/blog/hello-world` → `pages/blog/[slug].js`

Any `<Link />` in the viewport (initially or through scroll) will be prefetched by default (including the corresponding data) for pages using Static Generation. The corresponding data for server-rendered routes is fetched *only when* the `<Link />` is clicked.

Linking to dynamic paths

You can also use interpolation to create the path, which comes in handy for dynamic route segments. For example, to show a list of posts which have been passed to the component as a prop:

```
import Link from 'next/link'

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

export default Posts
```

`encodeURIComponent` is used in the example to keep the path utf-8 compatible.

Alternatively, using a URL Object:

```
import Link from 'next/link'

function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link
            href={{
              pathname: '/blog/[slug]',
              query: { slug: post.slug },
            }}
          >
            {post.title}
          </Link>
        </li>
      ))}
    </ul>
  )
}

export default Posts
```

Now, instead of using interpolation to create the path, we use a URL object in `href` where:

- `pathname` is the name of the page in the `pages` directory. `/blog/[slug]` in this case.
- `query` is an object with the dynamic segment. `slug` in this case.

Injecting the router

▶ **Examples**

To access the `router` object in a React component you can use `useRouter` or `withRouter`.

In general we recommend using `useRouter`.

Learn more

The router is divided in multiple parts:

next/link: Handle client-side navigations.
next/router: Leverage the router API in your pages.

---

description: You can use shallow routing to change the URL without triggering a new page change. Learn more here.

Shallow Routing

▶ **Examples**

Shallow routing allows you to change the URL without running data fetching methods again, that includes `getServerSideProps`, `getStaticProps`, and `getInitialProps`.

You'll receive the updated `pathname` and the `query` via the `router` object (added by `useRouter` or `withRouter`), without losing state.

To enable shallow routing, set the `shallow` option to `true`. Consider the following example:

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

// Current URL is '/'
function Page() {
  const router = useRouter()

  useEffect(() => {
    // Always do navigations after the first render
    router.push('/?counter=10', undefined, { shallow: true })
  }, [])

  useEffect(() => {
    // The counter changed!
  }, [router.query.counter])
}

export default Page
```

The URL will get updated to `/?counter=10`. and the page won't get replaced, only the state of the route is changed.

You can also watch for URL changes via `componentDidUpdate` as shown below:

```
componentDidUpdate(prevProps) {
  const { pathname, query } = this.props.router
  // verify props have changed to avoid an infinite loop
  if (query.counter !== prevProps.router.query.counter) {
    // fetch data based on the new query
  }
}
```

Caveats

Shallow routing **only** works for URL changes in the current page. For example, let's assume we have another page called `pages/about.js`, and you run this:

```
router.push('/?counter=10', '/about?counter=10', { shallow: true })
```

Since that's a new page, it'll unload the current page, load the new one and wait for data fetching even though we asked to do shallow routing.

When shallow routing is used with middleware it will not ensure the new page matches the current page like previously done without middleware. This is due to middleware being able to rewrite dynamically and can't be verified client-side without a data fetch which is skipped with shallow, so a shallow route change must always be treated as shallow.

---

description: Learn how to set up Next.js with three commonly used testing tools — Cypress, Playwright, Jest, and React Testing Library.

Testing

▼ **Examples**

- Next.js with Cypress
- Next.js with Playwright
- Next.js with Jest and React Testing Library
- Next.js with Vitest

Learn how to set up Next.js with commonly used testing tools: Cypress, Playwright, and Jest with React Testing Library.

Cypress

Cypress is a test runner used for **End-to-End (E2E)** and **Component Testing**.

Quickstart

You can use `create-next-app` with the with-cypress example to quickly get started.

```
npx create-next-app@latest --example with-cypress with-cypress-app
```

Manual setup

To get started with Cypress, install the `cypress` package:

```
npm install --save-dev cypress
```

Add Cypress to the `package.json` scripts field:

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

Run Cypress for the first time to generate examples that use their recommended folder structure:

```
npm run cypress
```

You can look through the generated examples and the Writing Your First Test section of the Cypress Documentation to help you get familiar with Cypress.

Should I use E2E or Component Tests?

The Cypress docs contain a guide on the difference between these two types of tests and when it is appropriate to use each.

Creating your first Cypress E2E test

Assuming the following two Next.js pages:

```
// pages/index.js
import Link from 'next/link'

export default function Home() {
  return (
    <nav>
      <h1>Homepage</h1>
      <Link href="/about">About</Link>
    </nav>
  )
}

// pages/about.js
export default function About() {
  return (
    <div>
      <h1>About Page</h1>
      <Link href="/">Homepage</Link>
    </div>
  )
}
```

Add a test to check your navigation is working correctly:

```
// cypress/e2e/app.cy.js

describe('Navigation', () => {
  it('should navigate to the about page', () => {
    // Start from the index page
    cy.visit('http://localhost:3000/')

    // Find a link with an href attribute containing "about" and click it
    cy.get('a[href*="about"]').click()

    // The new url should include "/about"
    cy.url().should('include', '/about')

    // The new page should contain an h1 with "About page"
    cy.get('h1').contains('About Page')
  })
})
```

You can use `cy.visit("/")` instead of `cy.visit("http://localhost:3000/")` if you add `baseUrl: 'http://localhost:3000'` to the `cypress.config.js` configuration file.

Creating your first Cypress component test

Component tests build and mount a specific component without having to bundle your whole application or launch a server. This allows for more performant tests that still provide visual feedback and the same API used for Cypress E2E tests.

> **Note**: Since component tests do not launch a Next.js server, capabilities like `<Image />` and `getServerSideProps` which rely on a server being available will not function out-of-the-box. See the Cypress Next.js docs for examples of getting these features working within component tests.

Assuming the same components from the previous section, add a test to validate a component is rendering the expected output:

```
// pages/about.cy.js
import AboutPage from './about.js'

describe('<AboutPage />', () => {
  it('should render and display expected content', () => {
    // Mount the React component for the About page
    cy.mount(<AboutPage />)

    // The new page should contain an h1 with "About page"
    cy.get('h1').contains('About Page')

    // Validate that a link with the expected URL is present
    // *Following* the link is better suited to an E2E test
    cy.get('a[href="/"]').should('be.visible')
  })
})
```

Running your Cypress tests

E2E Tests

Since Cypress E2E tests are testing a real Next.js application they require the Next.js server to be running prior to starting Cypress. We recommend running your tests against your production code to more closely resemble how your application will behave.

Run `npm run build` and `npm run start`, then run `npm run cypress -- --e2e` in another terminal window to start Cypress and run your E2E testing suite.

> **Note**: Alternatively, you can install the `start-server-and-test` package and add it to the `package.json` scripts field: `"test": "start-server-and-test start http://localhost:3000 cypress"` to start the Next.js production server in conjunction with Cypress. Remember to rebuild your application after new changes.

Component Tests

Run `npm run cypress -- --component` to start Cypress and execute your component testing suite.

Getting ready for Continuous Integration (CI)

You will have noticed that running Cypress so far has opened an interactive browser which is not ideal for CI environments. You can also run Cypress headlessly using the `cypress run` command:

```
// package.json

"scripts": {
  //...
  "e2e": "start-server-and-test dev http://localhost:3000 \"cypress open --e2e\"",
  "e2e:headless": "start-server-and-test dev http://localhost:3000 \"cypress run --e2e\"",
```

```
  "component": "cypress open --component",
  "component:headless": "cypress run --component"
}
```

You can learn more about Cypress and Continuous Integration from these resources:

- Cypress Continuous Integration Docs
- Cypress GitHub Actions Guide
- Official Cypress GitHub Action

Playwright

Playwright is a testing framework that lets you automate Chromium, Firefox, and WebKit with a single API. You can use it to write **End-to-End (E2E)** and **Integration** tests across all platforms.

Quickstart

The fastest way to get started is to use `create-next-app` with the with-playwright example. This will create a Next.js project complete with Playwright all set up.

```
npx create-next-app@latest --example with-playwright with-playwright-app
```

Manual setup

You can also use `npm init playwright` to add Playwright to an existing `NPM` project.

To manually get started with Playwright, install the `@playwright/test` package:

```
npm install --save-dev @playwright/test
```

Add Playwright to the `package.json` scripts field:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "test:e2e": "playwright test",
}
```

Creating your first Playwright end-to-end test

Assuming the following two Next.js pages:

```
// pages/index.js
import Link from 'next/link'

export default function Home() {
  return (
    <nav>
      <Link href="/about">About</Link>
    </nav>
  )
}

// pages/about.js
export default function About() {
  return (
    <div>
      <h1>About Page</h1>
    </div>
  )
}
```

Add a test to verify that your navigation is working correctly:

```
// e2e/example.spec.ts

import { test, expect } from '@playwright/test'

test('should navigate to the about page', async ({ page }) => {
  // Start from the index page (the baseURL is set via the webServer in the playwright.config.ts)
  await page.goto('http://localhost:3000/')
  // Find an element with the text 'About Page' and click on it
  await page.click('text=About')
  // The new URL should be "/about" (baseURL is used there)
  await expect(page).toHaveURL('http://localhost:3000/about')
  // The new page should contain an h1 with "About Page"
  await expect(page.locator('h1')).toContainText('About Page')
})
```

You can use `page.goto("/")` instead of `page.goto("http://localhost:3000/")`, if you add `"baseURL": "http://localhost:3000"` to the `playwright.config.ts` configuration file.

Running your Playwright tests

Since Playwright is testing a real Next.js application, it requires the Next.js server to be running prior to starting Playwright. It is recommended to run your tests against your production code to more closely resemble how your application will behave.

Run `npm run build` and `npm run start`, then run `npm run test:e2e` in another terminal window to run the Playwright tests.

> **Note**: Alternatively, you can use the `webServer` feature to let Playwright start the development server and wait until it's fully available.

Running Playwright on Continuous Integration (CI)

Playwright will by default run your tests in the headless mode. To install all the Playwright dependencies, run `npx playwright install-deps`.

You can learn more about Playwright and Continuous Integration from these resources:

- Getting started with Playwright
- Use a development server
- Playwright on your CI provider

Jest and React Testing Library

Jest and React Testing Library are frequently used together for **Unit Testing**. There are three ways you can start using Jest within your Next.js application:

1. Using one of our quickstart examples
2. With the Next.js Rust Compiler
3. With Babel

The following sections will go through how you can set up Jest with each of these options:

Quickstart

You can use `create-next-app` with the with-jest example to quickly get started with Jest and React Testing Library:

```
npx create-next-app@latest --example with-jest with-jest-app
```

Setting up Jest (with the Rust Compiler)

Since the release of Next.js 12, Next.js now has built-in configuration for Jest.

To set up Jest, install `jest`, `jest-environment-jsdom`, `@testing-library/react`, `@testing-library/jest-dom`:

```
npm install --save-dev jest jest-environment-jsdom @testing-library/react @testing-library/jest-dom
```

Create a `jest.config.mjs` file in your project's root directory and add the following:

```
// jest.config.mjs
import nextJest from 'next/jest.js'

const createJestConfig = nextJest({
  // Provide the path to your Next.js app to load next.config.js and .env files in your test environment
  dir: './',
})

// Add any custom config to be passed to Jest
/** @type {import('jest').Config} */
const config = {
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],

  testEnvironment: 'jest-environment-jsdom',
}

// createJestConfig is exported this way to ensure that next/jest can load the Next.js config which is async
export default createJestConfig(config)
```

Under the hood, `next/jest` is automatically configuring Jest for you, including:

- Setting up `transform` using SWC
- Auto mocking stylesheets (`.css`, `.module.css`, and their scss variants), image imports and `next/font`
- Loading `.env` (and all variants) into `process.env`
- Ignoring `node_modules` from test resolving and transforms
- Ignoring `.next` from test resolving
- Loading `next.config.js` for flags that enable SWC transforms

> **Note**: To test environment variables directly, load them manually in a separate setup script or in your `jest.config.js` file. For more information, please see Test Environment Variables.

Setting up Jest (with Babel)

If you opt out of the Rust Compiler, you will need to manually configure Jest and install `babel-jest` and `identity-obj-proxy` in addition to the packages above.

Here are the recommended options to configure Jest for Next.js:

```
// jest.config.js
module.exports = {
  collectCoverage: true,
  // on node 14.x coverage provider v8 offers good speed and more or less good report
  coverageProvider: 'v8',
  collectCoverageFrom: [
    '**/*.{js,jsx,ts,tsx}',
    '!**/*.d.ts',
    '!**/node_modules/**',
    '!<rootDir>/out/**',
    '!<rootDir>/.next/**',
    '!<rootDir>/*.config.js',
    '!<rootDir>/coverage/**',
  ],
  moduleNameMapper: {
    // Handle CSS imports (with CSS modules)
    // https://jestjs.io/docs/webpack#mocking-css-modules
    '^.+\.module\.(css|sass|scss)$': 'identity-obj-proxy',

    // Handle CSS imports (without CSS modules)
    '^.+\.(css|sass|scss)$': '<rootDir>/__mocks__/styleMock.js',

    // Handle image imports
    // https://jestjs.io/docs/webpack#handling-static-assets
    '^.+\.(png|jpg|jpeg|gif|webp|avif|ico|bmp|svg)$/i': `<rootDir>/__mocks__/fileMock.js`,

    // Handle module aliases
    '^@/components/(.*)$': '<rootDir>/components/$1',
  },
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],
  testPathIgnorePatterns: ['<rootDir>/node_modules/', '<rootDir>/.next/'],
  testEnvironment: 'jsdom',
  transform: {
    // Use babel-jest to transpile tests with the next/babel preset
    // https://jestjs.io/docs/configuration#transform-objectstring-pathtotransformer--pathtotransformer-object
    '^.+\.(js|jsx|ts|tsx)$': ['babel-jest', { presets: ['next/babel'] }],
  },
  transformIgnorePatterns: [
```

```
    '/node_modules/',
    '^.+\.module\.(css|sass|scss)$',
  ],
}
```

You can learn more about each configuration option in the Jest docs.

Handling stylesheets and image imports

Stylesheets and images aren't used in the tests but importing them may cause errors, so they will need to be mocked. Create the mock files referenced in the configuration above - `fileMock.js` and `styleMock.js` - inside a `__mocks__` directory:

```
// __mocks__/fileMock.js
module.exports = {
  src: '/img.jpg',
  height: 24,
  width: 24,
  blurDataURL: '',
}

// __mocks__/styleMock.js
module.exports = {}
```

For more information on handling static assets, please refer to the Jest Docs.

Optional: Extend Jest with custom matchers

`@testing-library/jest-dom` includes a set of convenient custom matchers such as `.toBeInTheDocument()` making it easier to write tests. You can import the custom matchers for every test by adding the following option to the Jest configuration file:

```
// jest.config.js
setupFilesAfterEnv: ['<rootDir>/jest.setup.js']
```

Then, inside `jest.setup.js`, add the following import:

```
// jest.setup.js
import '@testing-library/jest-dom/extend-expect'
```

If you need to add more setup options before each test, it's common to add them to the `jest.setup.js` file above.

Optional: Absolute Imports and Module Path Aliases

If your project is using Module Path Aliases, you will need to configure Jest to resolve the imports by matching the paths option in the `jsconfig.json` file with the `moduleNameMapper` option in the `jest.config.js` file. For example:

```
// tsconfig.json or jsconfig.json
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/components/*": ["components/*"]
    }
  }
}

// jest.config.js
moduleNameMapper: {
  '^@/components/(.*)$': '<rootDir>/components/$1',
}
```

Creating your tests:

Add a test script to package.json

Add the Jest executable in watch mode to the `package.json` scripts:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "test": "jest --watch"
}
```

`jest --watch` will re-run tests when a file is changed. For more Jest CLI options, please refer to the Jest Docs.

Create your first tests

Your project is now ready to run tests. Follow Jest's convention by adding tests to the `__tests__` folder in your project's root directory.

For example, we can add a test to check if the `<Home />` component successfully renders a heading:

```
// __tests__/index.test.jsx

import { render, screen } from '@testing-library/react'
import Home from '../pages/index'
import '@testing-library/jest-dom'

describe('Home', () => {
  it('renders a heading', () => {
    render(<Home />)

    const heading = screen.getByRole('heading', {
      name: /welcome to next\.js!/i,
    })

    expect(heading).toBeInTheDocument()
  })
})
```

Optionally, add a snapshot test to keep track of any unexpected changes to your `<Home />` component:

```
// __tests__/snapshot.js

import { render } from '@testing-library/react'
import Home from '../pages/index'
```

```
it('renders homepage unchanged', () => {
  const { container } = render(<Home />)
  expect(container).toMatchSnapshot()
})
```

> **Note**: Test files should not be included inside the pages directory because any files inside the pages directory are considered routes.

Running your test suite

Run `npm run test` to run your test suite. After your tests pass or fail, you will notice a list of interactive Jest commands that will be helpful as you add more tests.

For further reading, you may find these resources helpful:

- Jest Docs
- React Testing Library Docs
- Testing Playground - use good testing practices to match elements.

Community Packages and Examples

The Next.js community has created packages and articles you may find helpful:

- next-router-mock for Storybook.
- Test Preview Vercel Deploys with Cypress by Gleb Bahmutov.

For more information on what to read next, we recommend:

Test Environment Variables Learn more about the test environment variables.

---

description: Learn how to upgrade Next.js.

Upgrade Guide

Upgrading from 12 to 13

To update to Next.js version 13, run the following command using your preferred package manager:

```
npm i next@latest react@latest react-dom@latest eslint-config-next@latest
# or
yarn add next@latest react@latest react-dom@latest eslint-config-next@latest
# or
pnpm up next react react-dom eslint-config-next --latest
```

v13 Summary

- The Supported Browsers have been changed to drop Internet Explorer and target modern browsers.
- The minimum Node.js version has been bumped from 12.22.0 to 14.18.0, since 12.x has reached end-of-life.
- The minimum React version has been bumped from 17.0.2 to 18.2.0.
- The `swcMinify` configuration property was changed from `false` to `true`. See Next.js Compiler for more info.
- The `next/image` import was renamed to `next/legacy/image`. The `next/future/image` import was renamed to `next/image`. A codemod is available to safely and automatically rename your imports.
- The `next/link` child can no longer be `<a>`. Add the `legacyBehavior` prop to use the legacy behavior or remove the `<a>` to upgrade. A codemod is available to automatically upgrade your code.
- The `target` configuration property has been removed and superseded by Output File Tracing.

Migrating shared features

Next.js 13 introduces a new `app` directory with new features and conventions. However, upgrading to Next.js 13 does **not** require using the new `app` directory.

You can continue using `pages` with new features that work in both directories, such as the updated Image component, Link component, Script component, and Font optimization.

`<Image/>` Component

Next.js 12 introduced many improvements to the Image Component with a temporary import: `next/future/image`. These improvements included less client-side JavaScript, easier ways to extend and style images, better accessibility, and native browser lazy loading.

Starting in Next.js 13, this new behavior is now the default for `next/image`.

There are two codemods to help you migrate to the new Image Component:

- next-image-to-legacy-image: This codemod will safely and automatically rename `next/image` imports to `next/legacy/image` to maintain the same behavior as Next.js 12. We recommend running this codemod to quickly update to Next.js 13 automatically.
- next-image-experimental: After running the previous codemod, you can optionally run this experimental codemod to upgrade `next/legacy/image` to the new `next/image`, which will remove unused props and add inline styles. Please note this codemod is experimental and only covers static usage (such as `<Image src={img} layout="responsive" />`) but not dynamic usage (such as `<Image {...props} />`).

Alternatively, you can manually update by following the migration guide and also see the legacy comparison.

`<Link>` Component

The `<Link>` Component no longer requires manually adding an `<a>` tag as a child. This behavior was added as an experimental option in version 12.2 and is now the default. In Next.js 13, `<Link>` always renders `<a>` and allows you to forward props to the underlying tag.

For example:

```
import Link from 'next/link'

// Next.js 12: `<a>` has to be nested otherwise 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`. If incrementally adopting `app`, read the upgrade guide.

Font Optimization

Previously, Next.js helped you optimize fonts by inlining font CSS. Version 13 introduces the new `next/font` module which gives you the ability to customize your font loading experience while still ensuring great performance and privacy.

See Optimizing Fonts to learn how to use `next/font`.

Upgrading to 12.2

If you were using Middleware prior to `12.2`, please see the upgrade guide for more information.

Upgrading from 11 to 12

Minimum Node.js version

The minimum Node.js version has been bumped from 12.0.0 to 12.22.0 which is the first version of Node.js with native ES Modules support.

Upgrade React version to latest

The minimum required React version is `17.0.2`. To upgrade you can run the following command in the terminal:

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

Or using `yarn`:

```
yarn add react@latest react-dom@latest
```

Upgrade Next.js version to 12

To upgrade you can run the following command in the terminal:

```
npm install next@12
```

or

```
yarn add next@12
```

SWC replacing Babel

Next.js now uses a Rust-based compiler, SWC, to compile JavaScript/TypeScript. This new compiler is up to 17x faster than Babel when compiling individual files and allows for up to 5x faster Fast Refresh.

Next.js provides full backwards compatibility with applications that have custom Babel configuration. All transformations that Next.js handles by default like styled-jsx and tree-shaking of `getStaticProps` / `getStaticPaths` / `getServerSideProps` have been ported to Rust.

When an application has a custom Babel configuration, Next.js will automatically opt-out of using SWC for compiling JavaScript/Typescript and will fall back to using Babel in the same way that it was used in Next.js 11.

Many of the integrations with external libraries that currently require custom Babel transformations will be ported to Rust-based SWC transforms in the near future. These include but are not limited to:

- Styled Components
- Emotion
- Relay

In order to prioritize transforms that will help you adopt SWC, please provide your `.babelrc` on the feedback thread.

SWC replacing Terser for minification

You can opt-in to replacing Terser with SWC for minifying JavaScript up to 7x faster using a flag in `next.config.js`:

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

Minification using SWC is an opt-in flag to ensure it can be tested against more real-world Next.js applications before it becomes the default in Next.js 12.1. If you have feedback about minification, please leave it on the feedback thread.

Improvements to styled-jsx CSS parsing

On top of the Rust-based compiler, we've implemented a new CSS parser based on the CSS parser that was used for the styled-jsx Babel transform. This new parser has improved handling of CSS and now errors when invalid CSS is used that would previously slip through and cause unexpected behavior.

Because of this change, invalid CSS will throw an error during development and `next build`. This change only affects styled-jsx usage.

`next/image` changed wrapping element

`next/image` now renders the `<img>` inside a `<span>` instead of `<div>`.

If your application has specific CSS targeting span, for example, `.container span`, upgrading to Next.js 12 might incorrectly match the wrapping element inside the `<Image>` component. You can avoid this by restricting the selector to a specific class such as `.container span.item` and updating the relevant component with that className, such as `<span className="item" />`.

If your application has specific CSS targeting the `next/image` `<div>` tag, for example `.container div`, it may not match anymore. You can update the selector `.container span`, or preferably, add a new `<div className="wrapper">` wrapping the `<Image>` component and target that instead such as `.container .wrapper`.

The `className` prop is unchanged and will still be passed to the underlying `<img>` element.

See the documentation for more info.

Next.js' HMR connection now uses a WebSocket

Previously, Next.js used a server-sent events connection to receive HMR events. Next.js 12 now uses a WebSocket connection.

In some cases when proxying requests to the Next.js dev server, you will need to ensure the upgrade request is handled correctly. For example, in `nginx` you would need to add the following configuration:

```
location /_next/webpack-hmr {
    proxy_pass http://localhost:3000/_next/webpack-hmr;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}
```

For custom servers, such as `express`, you may need to use `app.all` to ensure the request is passed correctly, for example:

```
app.all('/_next/webpack-hmr', (req, res) => {
  nextjsRequestHandler(req, res)
})
```

Webpack 4 support has been removed

If you are already using webpack 5 you can skip this section.

Next.js has adopted webpack 5 as the default for compilation in Next.js 11. As communicated in the webpack 5 upgrading documentation Next.js 12 removes support for webpack 4.

If your application is still using webpack 4 using the opt-out flag you will now see an error linking to the webpack 5 upgrading documentation.

`target` option deprecated

If you do not have `target` in `next.config.js` you can skip this section.

The target option has been deprecated in favor of built-in support for tracing what dependencies are needed to run a page.

During `next build`, Next.js will automatically trace each page and its dependencies to determine all of the files that are needed for deploying a production version of your application.

If you are currently using the `target` option set to `serverless` please read the documentation on how to leverage the new output.

Upgrading from version 10 to 11

Upgrade React version to latest

Most applications already use the latest version of React, with Next.js 11 the minimum React version has been updated to 17.0.2.

To upgrade you can run the following command:

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

Or using `yarn`:

```
yarn add react@latest react-dom@latest
```

Upgrade Next.js version to 11

To upgrade you can run the following command in the terminal:

```
npm install next@11
```

or

```
yarn add next@11
```

Webpack 5

Webpack 5 is now the default for all Next.js applications. If you did not have custom webpack configuration your application is already using webpack 5. If you do have custom webpack configuration you can refer to the Next.js webpack 5 documentation for upgrading guidance.

Cleaning the `distDir` is now a default

The build output directory (defaults to `.next`) is now cleared by default except for the Next.js caches. You can refer to the cleaning `distDir` RFC for more information.

If your application was relying on this behavior previously you can disable the new default behavior by adding the `cleanDistDir: false` flag in `next.config.js`.

`PORT` is now supported for `next dev` and `next start`

Next.js 11 supports the `PORT` environment variable to set the port the application has to run on. Using `-p`/`--port` is still recommended but if you were prohibited from using `-p` in any way you can now use `PORT` as an alternative:

Example:

```
PORT=4000 next start
```

`next.config.js` customization to import images

Next.js 11 supports static image imports with `next/image`. This new feature relies on being able to process image imports. If you previously added the `next-images` or `next-optimized-images` packages you can either move to the new built-in support using `next/image` or disable the feature:

```
module.exports = {
  images: {
    disableStaticImages: true,
  },
}
```

Remove `super.componentDidCatch()` from `pages/_app.js`

The `next/app` component's `componentDidCatch` has been deprecated since Next.js 9 as it's no longer needed and has since been a no-op, in Next.js 11 it has been removed.

If your `pages/_app.js` has a custom `componentDidCatch` method you can remove `super.componentDidCatch` as it is no longer needed.

Remove `Container` from `pages/_app.js`

This export has been deprecated since Next.js 9 as it's no longer needed and has since been a no-op with a warning during development. In Next.js 11 it has been removed.

If your `pages/_app.js` imports `Container` from `next/app` you can remove `Container` as it has been removed. Learn more in the documentation.

Remove `props.url` usage from page components

This property has been deprecated since Next.js 4 and has since shown a warning during development. With the introduction of `getStaticProps` / `getServerSideProps` these methods already disallowed usage of `props.url`. In Next.js 11 it has been removed completely.

You can learn more in the documentation.

Remove `unsized` property on `next/image`

The `unsized` property on `next/image` was deprecated in Next.js 10.0.1. You can use `layout="fill"` instead. In Next.js 11 `unsized` was removed.

Remove `modules` property on `next/dynamic`

The `modules` and `render` option for `next/dynamic` have been deprecated since Next.js 9.5 showing a warning that it has been deprecated. This was done in order to make `next/dynamic` close to `React.lazy` in API surface. In Next.js 11 the `modules` and `render` options have been removed.

This option hasn't been mentioned in the documentation since Next.js 8 so it's less likely that your application is using it.

If your application does use `modules` and `render` you can refer to the documentation.

Remove `Head.rewind`

`Head.rewind` has been a no-op since Next.js 9.5, in Next.js 11 it was removed. You can safely remove your usage of `Head.rewind`.

Moment.js locales excluded by default

Moment.js includes translations for a lot of locales by default. Next.js now automatically excludes these locales by default to optimize bundle size for applications using Moment.js.

To load a specific locale use this snippet:

```
import moment from 'moment'
import 'moment/locale/ja'

moment.locale('ja')
```

You can opt-out of this new default by adding `excludeDefaultMomentLocales: false` to `next.config.js` if you do not want the new behavior, do note it's highly recommended to not disable this new optimization as it significantly reduces the size of Moment.js.

Update usage of `router.events`

In case you're accessing `router.events` during rendering, in Next.js 11 `router.events` is no longer provided during pre-rendering. Ensure you're accessing `router.events` in `useEffect`:

```
useEffect(() => {
  const handleRouteChange = (url, { shallow }) => {
    console.log(
      `App is changing to ${url} ${
        shallow ? 'with' : 'without'
      } shallow routing`
    )
  }

  router.events.on('routeChangeStart', handleRouteChange)

  // If the component is unmounted, unsubscribe
  // from the event with the `off` method:
  return () => {
    router.events.off('routeChangeStart', handleRouteChange)
  }
}, [router])
```

If your application uses `router.router.events` which was an internal property that was not public please make sure to use `router.events` as well.

React 16 to 17

React 17 introduced a new JSX Transform that brings a long-time Next.js feature to the wider React ecosystem: Not having to `import React from 'react'` when using JSX. When using React 17 Next.js will automatically use the new transform. This transform does not make the `React` variable global, which was an unintended side-effect of the previous Next.js implementation. A codemod is available to automatically fix cases where you accidentally used `React` without importing it.

Upgrading from version 9 to 10

There were no breaking changes between version 9 and 10.

To upgrade run the following command:

```
npm install next@10
```

Or using `yarn`:

```
yarn add next@10
```

Upgrading from version 8 to 9

Preamble

Production Deployment on Vercel

If you previously configured `routes` in your `vercel.json` file for dynamic routes, these rules can be removed when leveraging Next.js 9's new Dynamic Routing feature.

Next.js 9's dynamic routes are **automatically configured on Vercel** and do not require any `vercel.json` customization.

You can read more about Dynamic Routing here.

Check your Custom (`pages/_app.js`)

If you previously copied the Custom `<App>` example, you may be able to remove your `getInitialProps`.

Removing `getInitialProps` from `pages/_app.js` (when possible) is important to leverage new Next.js features!

The following `getInitialProps` does nothing and may be removed:

```
class MyApp extends App {
  // Remove me, I do nothing!
  static async getInitialProps({ Component, ctx }) {
    let pageProps = {}

    if (Component.getInitialProps) {
      pageProps = await Component.getInitialProps(ctx)
    }

    return { pageProps }
  }

  render() {
    // ... etc
  }
}
```

Breaking Changes

`@zeit/next-typescript` is no longer necessary

Next.js will now ignore usage `@zeit/next-typescript` and warn you to remove it. Please remove this plugin from your `next.config.js`.

Remove references to `@zeit/next-typescript/babel` from your custom `.babelrc` (if present).

Usage of `fork-ts-checker-webpack-plugin` should also be removed from your `next.config.js`.

TypeScript Definitions are published with the `next` package, so you need to uninstall `@types/next` as they would conflict.

The following types are different:

> This list was created by the community to help you upgrade, if you find other differences please send a pull request to this list to help other users.

From:

```
import { NextContext } from 'next'
import { NextAppContext, DefaultAppIProps } from 'next/app'
import { NextDocumentContext, DefaultDocumentIProps } from 'next/document'
```

to

```
import { NextPageContext } from 'next'
import { AppContext, AppInitialProps } from 'next/app'
import { DocumentContext, DocumentInitialProps } from 'next/document'
```

The `config` key is now a named export on a page

You may no longer export a custom variable named `config` from a page (i.e. `export { config }` / `export const config ...`). This exported variable is now used to specify page-level Next.js configuration like Opt-in AMP and API Route features.

You must rename a non-Next.js-purposed `config` export to something different.

`next/dynamic` no longer renders "loading..." by default while loading

Dynamic components will not render anything by default while loading. You can still customize this behavior by setting the `loading` property:

```
import dynamic from 'next/dynamic'

const DynamicComponentWithCustomLoading = dynamic(
  () => import('../components/hello2'),
  {
    loading: () => <p>Loading</p>,
  }
)
```

`withAmp` has been removed in favor of an exported configuration object

Next.js now has the concept of page-level configuration, so the `withAmp` higher-order component has been removed for consistency.

This change can be **automatically migrated by running the following commands at the root of your Next.js project:**

```
curl -L https://github.com/vercel/next-codemod/archive/master.tar.gz | tar -xz --strip=2 next-codemod-master/transforms/withamp-to-config.js npx jscodes
```

To perform this migration by hand, or view what the codemod will produce, see below:

Before

```
import { withAmp } from 'next/amp'

function Home() {
  return <h1>My AMP Page</h1>
}

export default withAmp(Home)
// or
export default withAmp(Home, { hybrid: true })
```

After

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

export const config = {
  amp: true,
  // or
  amp: 'hybrid',
}
```

`next export` no longer exports pages as `index.html`

Previously, exporting `pages/about.js` would result in `out/about/index.html`. This behavior has been changed to result in `out/about.html`.

You can revert to the previous behavior by creating a `next.config.js` with the following content:

```
// next.config.js
module.exports = {
  trailingSlash: true,
}
```

`./pages/api/` is treated differently

Pages in `./pages/api/` are now considered API Routes. Pages in this directory will no longer contain a client-side bundle.

Deprecated Features

`next/dynamic` has deprecated loading multiple modules at once

The ability to load multiple modules at once has been deprecated in `next/dynamic` to be closer to React's implementation (`React.lazy` and `Suspense`).

Updating code that relies on this behavior is relatively straightforward! We've provided an example of a before/after to help you migrate your application:

Before

```
import dynamic from 'next/dynamic'

const HelloBundle = dynamic({
  modules: () => {
    const components = {
      Hello1: () => import('../components/hello1').then((m) => m.default),
      Hello2: () => import('../components/hello2').then((m) => m.default),
    }

    return components
  },
  render: (props, { Hello1, Hello2 }) => (
    <div>
      <h1>{props.title}</h1>
      <Hello1 />
      <Hello2 />
    </div>
  ),
})

function DynamicBundle() {
  return <HelloBundle title="Dynamic Bundle" />
}

export default DynamicBundle
```

After

```
import dynamic from 'next/dynamic'

const Hello1 = dynamic(() => import('../components/hello1'))
const Hello2 = dynamic(() => import('../components/hello2'))

function HelloBundle({ title }) {
  return (
    <div>
      <h1>{title}</h1>
      <Hello1 />
      <Hello2 />
    </div>
  )
}

function DynamicBundle() {
  return <HelloBundle title="Dynamic Bundle" />
}

export default DynamicBundle
```