

Learn to code – free 3,000-hour curriculum

NOVEMBER 19, 2019 / #FRONTEND

The Next.js Handbook – Learn Next.js for Beginners



Flavio Copes

The ~~NEXT~~.JS
handbook

I wrote this tutorial to help you quickly learn Next.js and get familiar with how it works.

It's ideal for you if you have zero to little knowledge of Next.js, you have used React in the past, and you are looking forward diving more into the React ecosystem, in particular server-side rendering.

Learn to code – free 3,000-hour curriculum

Note: you can download a PDF / ePub / Mobi version of this tutorial so you can read it offline!

Index

1. [Introduction](#)
2. [The main features provided by Next.js](#)
3. [Next.js vs Gatsby vs create-react-app](#)
4. [How to install Next.js](#)
5. [View source to confirm SSR is working](#)
6. [The app bundles](#)
7. [What's that icon in the bottom right?](#)
8. [Install the React DevTools](#)
9. [Other debugging techniques you can use](#)
10. [Adding a second page to the site](#)
11. [Linking the two pages](#)
12. [Dynamic content with the router](#)
13. [Prefetching](#)
14. [Using the router to detect the active link](#)
15. [Using next/router](#)
16. [Feed data to the components using getInitialProps\(\)](#)
17. [CSS](#)
18. [Populating the head tag with custom tags](#)
19. [Adding a wrapper component](#)

Learn to code — free 3,000-hour curriculum

22. [Deploying the production version](#)
23. [Deploying on Now](#)
24. [Analyzing the app bundles](#)
25. [Lazy loading modules](#)
26. [Where to go from here](#)

Introduction

Working on a modern JavaScript application powered by React is awesome until you realize that there are a couple problems related to rendering all the content on the client-side.

First, the page takes longer to become visible to the user, because before the content loads, all the JavaScript must load, and your application needs to run to determine what to show on the page.

Second, if you are building a publicly available website, you have a content SEO issue. Search engines are getting better at running and indexing JavaScript apps, but it's much better if we can send them content instead of letting them figure it out.

The solution to both of those problems is **server rendering**, also called **static pre-rendering**.

[Next.js](#) is one React framework to do all of this in a very simple way, but it's not limited to this. It's advertised by its creators as a **zero-configuration, single-command toolchain for React apps**.

It provides a common structure that allows you to easily build a frontend React application, and transparently handles server-side

Learn to code – free 3,000-hour curriculum

The main features provided by Next.js

Here is a non-exhaustive list of the main Next.js features:

Hot Code Reloading

Next.js reloads the page when it detects any change saved to disk.

Automatic Routing

Any URL is mapped to the filesystem, to files put in the `pages` folder, and you don't need any configuration (you have customization options of course).

Single File Components

Using `styled-jsx`, completely integrated as built by the same team, it's trivial to add styles scoped to the component.

Server Rendering

You can render React components on the server side, before sending the HTML to the client.

Ecosystem Compatibility

Next.js plays well with the rest of the JavaScript, Node, and React ecosystem.

Automatic Code Splitting

Pages are rendered with just the libraries and JavaScript that they need, no more. Instead of generating one single JavaScript file

Learn to code – free 3,000-hour curriculum

Loading a page only loads the JavaScript necessary for that particular page.

Next.js does that by analyzing the resources imported.

If only one of your pages imports the Axios library, for example, that specific page will include the library in its bundle.

This ensures your first page load is as fast as it can be, and only future page loads (if they will ever be triggered) will send the JavaScript needed to the client.

There is one notable exception. Frequently used imports are moved into the main JavaScript bundle if they are used in at least half of the site pages.

Prefetching

The `Link` component, used to link together different pages, supports a `prefetch` prop which automatically prefetches page resources (including code missing due to code splitting) in the background.

Dynamic Components

You can import JavaScript modules and React Components dynamically.

Static Exports

Using the `next export` command, Next.js allows you to export a fully static site from your app.

Learn to code – free 3,000-hour curriculum

TypeScript support.

Next.js vs Gatsby vs `create-react-app`

Next.js, [Gatsby](#), and [create-react-app](#) are amazing tools we can use to power our applications.

Let's first say what they have in common. They all have React under the hood, powering the entire development experience. They also abstract [webpack](#) and all those low level things that we used to configure manually in the good old days.

`create-react-app` does not help you generate a server-side-rendered app easily. Anything that comes with it (SEO, speed...) is only provided by tools like Next.js and Gatsby.

When is Next.js better than Gatsby?

They can both help with **server-side rendering**, but in 2 different ways.

The end result using Gatsby is a static site generator, without a server. You build the site, and then you deploy the result of the build process statically on Netlify or another static hosting site.

Next.js provides a backend that can server side render a response to request, allowing you to create a dynamic website, which means you will deploy it on a platform that can run Node.js.

Learn to code – free 3,000-hour curriculum

If my goal was to build a static site, I'd have a hard time choosing and perhaps Gatsby has a better ecosystem of plugins, including many for blogging in particular.

Gatsby is also heavily based on [GraphQL](#), something you might really like or dislike depending on your opinions and needs.

How to install Next.js

To install Next.js, you need to have Node.js installed.

Make sure that you have the latest version of Node. Check with running `node -v` in your terminal, and compare it to the latest LTS version listed on <https://nodejs.org/>.

After you install Node.js, you will have the `npm` command available into your command line.

If you have any trouble at this stage, I recommend the following tutorials I wrote for you:

- [How to install Node.js](#)
- [How to update Node.js](#)
- [An introduction to the npm package manager](#)
- [Unix Shells Tutorial](#)
- [How to use the macOS terminal](#)
- [The Bash Shell](#)

Learn to code – free 3,000-hour curriculum

We can choose 2 routes now: using `create-next-app` or the classic approach which involves installing and setting up a Next app manually.

Using `create-next-app`

If you're familiar with `create-react-app`, `create-next-app` is the same thing - except it creates a Next app instead of a React app, as the name implies.

I assume you have already installed Node.js, which, from version 5.2 (2+ years ago at the time of writing), comes with the `npx` command bundled. This handy tool lets us download and execute a JavaScript command, and we'll use it like this:

```
npx create-next-app
```

The command asks the application name (and creates a new folder for you with that name), then downloads all the packages it needs (`react`, `react-dom`, `next`), sets the `package.json` to:

Learn to code – free 3,000-hour curriculum

The screenshot shows a code editor interface with a sidebar and a main editor area. The sidebar on the left lists project files and folders:

- OPEN EDITORS
- package.json
- MY-APP
 - .next
 - .vscode
 - components (selected)
 - pages
 - index.js
 - public
 - .gitignore
 - package-lock.json
 - package.json
- OUTLINE
- NPM SCRIPTS

The main editor area displays the contents of the package.json file:

```
1 "name": "my-app",
2 "version": "0.1.0",
3 "private": true,
4 "scripts": {
5   "dev": "next dev",
6   "build": "next build",
7   "start": "next start"
8 },
9 "dependencies": {
10   "next": "9.1.3",
11   "react": "16.11.0",
12   "react-dom": "16.11.0"
13 }
14
15
16
```

At the bottom of the editor, there are status indicators: 0 errors, 0 warnings, initializing JS/TS language features, Ln 1, Col 1, Spaces: 2, UTF-8, LF, JSON, and a few small icons.

and you can immediately run the sample app by running `npm run dev`:

Learn to code — free 3,000-hour curriculum

```
Success! Created my-app at /Users/flaviocopes/dev/nextjs/testing/my-app
Inside that directory, you can run several commands:
```

```
npm run dev
  Starts the development server.

npm run build
  Builds the app for production.

npm start
  Runs the built app in production mode.
```

We suggest that you begin by typing:

```
cd my-app
npm run dev

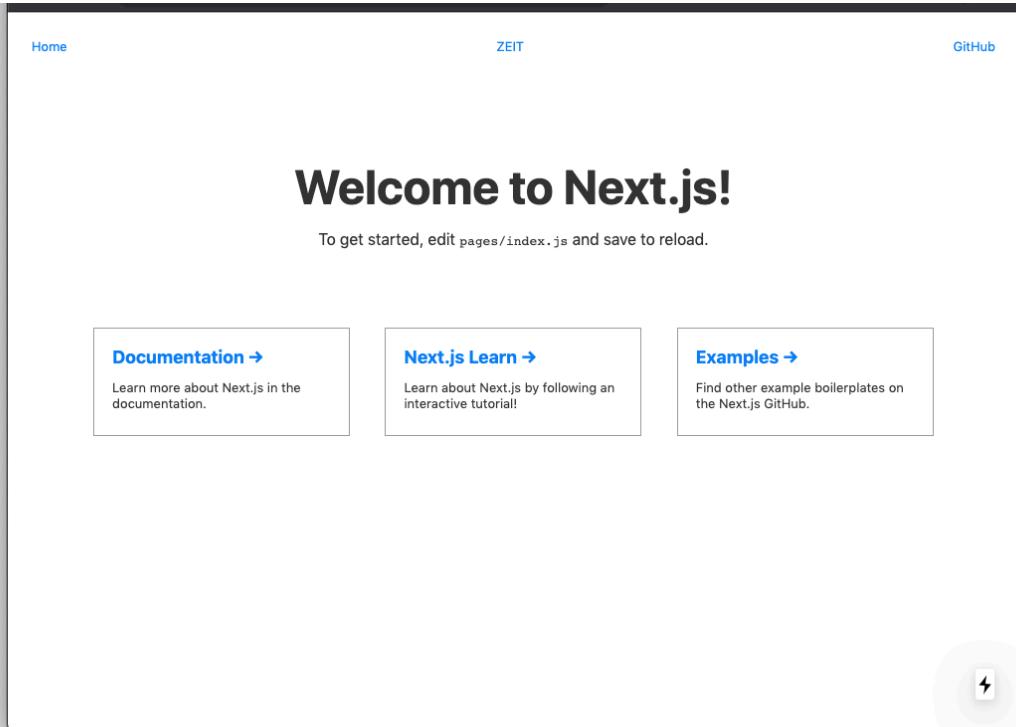
[→ testing code my-app/
[→ testing cd my-app/
[→ my-app npm run dev

> my-app@0.1.0 dev /Users/flaviocopes/dev/nextjs/testing/my-app
> next dev

[ wait ] starting the development server ...
[ info ] waiting on http://localhost:3000 ...
[ ready ] compiled successfully - ready on http://localhost:3000
[ wait ] compiling ...
[ ready ] compiled successfully - ready on http://localhost:3000
```

And here's the result on <http://localhost:3000>:

Learn to code – free 3,000-hour curriculum



This is the recommended way to start a Next.js application, as it gives you structure and sample code to play with. There's more than just that default sample application; you can use any of the examples stored at <https://github.com/zeit/next.js/tree/canary/examples> using the `--example` option. For example try:

```
npx create-next-app --example blog-starter
```

Which gives you an immediately usable blog instance with syntax highlighting too:

Learn to code – free 3,000-hour curriculum

The screenshot shows a web browser displaying a blog post from the 'Next.js Starter Blog'. The post is titled 'Syntax highlighting' and was published on May 19, 2018, by Juan Olvera. The page includes a navigation bar with links to 'Next.js Starter Blog' and 'About'. Below the title, there is a snippet of JavaScript code for a component named 'tick'. The code uses JSX and includes a call to 'ReactDOM.render'. A yellow vertical bar highlights the first line of this code. Below the component definition, there is another snippet of Python code using the 'requests' library to make a GET request to GitHub's API. This code prints the response headers, encoding, and text. A third snippet shows a portion of a CSS file with rules for the 'body' element. The code is presented in a monospaced font with color-coded syntax highlighting.

Manually create a Next.js app

You can avoid `create-next-app` if you feel like creating a Next app from scratch. Here's how: create an empty folder anywhere you like, for example in your home folder, and go into it:

```
mkdir nextjs
cd nextjs
```

and create your first Next project directory:

```
mkdir firstproject
cd firstproject
```

Learn to code – free 3,000-hour curriculum

```
npm init -y
```

The `-y` option tells `npm` to use the default settings for a project, populating a sample `package.json` file.



A screenshot of a terminal window titled "firstproject — fish /Users/flaviocopes/dev/nextjs/firstproject — -fish — 71x21". The command `npm init -y` is run, followed by the message "Wrote to /Users/flaviocopes/dev/nextjs/firstproject/package.json:". The terminal then displays the contents of the generated `package.json` file:

```
{  
  "name": "firstproject",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC"  
}
```

The terminal prompt "`→ firstproject`" is visible at the bottom.

Now install Next and React:

```
npm install next react react-dom
```

Your project folder should now have 2 files:

- `package.json` ([see my tutorial on it](#))

Learn to code – free 3,000-hour curriculum
and the node_modules folder.

Open the project folder using your favorite editor. My favorite editor is [VS Code](#). If you have that installed, you can run `code .` in your terminal to open the current folder in the editor (if the command does not work for you, see [this](#))

Open `package.json`, which now has this content:

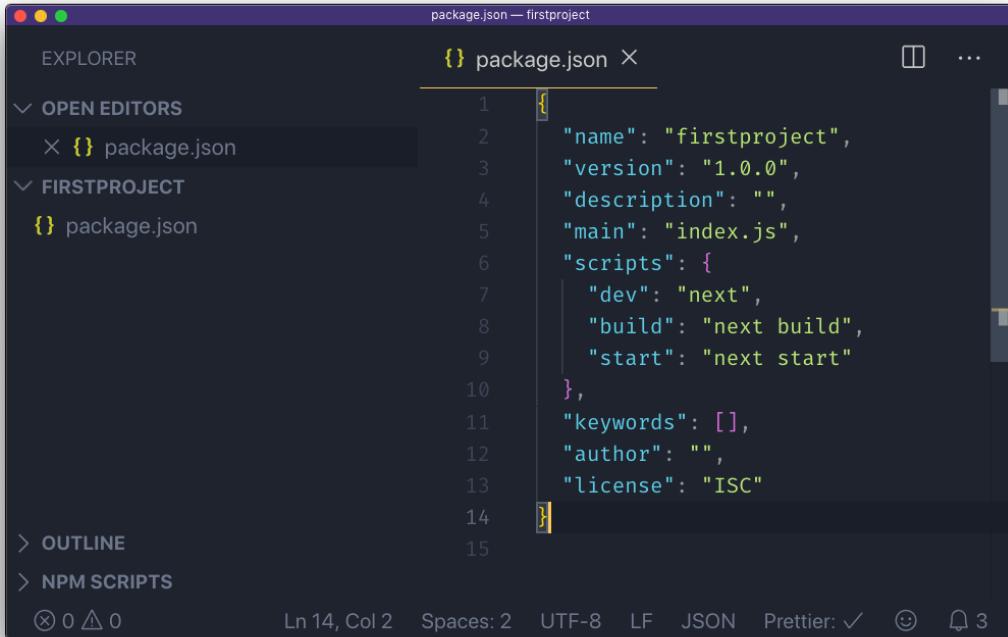
```
{  
  "name": "firstproject",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "next": "^9.1.2",  
    "react": "^16.11.0",  
    "react-dom": "^16.11.0"  
  }  
}
```

and replace the `scripts` section with:

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

Learn to code – free 3,000-hour curriculum

Tip: use "dev": "next -p 3001", to change the port and run, in this example, on port 3001.



The screenshot shows a code editor window with a dark theme. The left sidebar is titled 'EXPLORER' and shows two entries: 'OPEN EDITORS' containing a package.json file, and 'FIRSTPROJECT' containing a package.json file. The main area is titled '{ package.json }' and displays the contents of the package.json file. The file content is as follows:

```
1  {
2    "name": "firstproject",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "dev": "next",
8      "build": "next build",
9      "start": "next start"
10    },
11    "keywords": [],
12    "author": "",
13    "license": "ISC"
14  }
15
```

At the bottom of the editor, there are status indicators: Ln 14, Col 2, Spaces: 2, UTF-8, LF, JSON, Prettier: ✓, and a few other icons.

Now create a `pages` folder, and add an `index.js` file.

In this file, let's create our first React component.

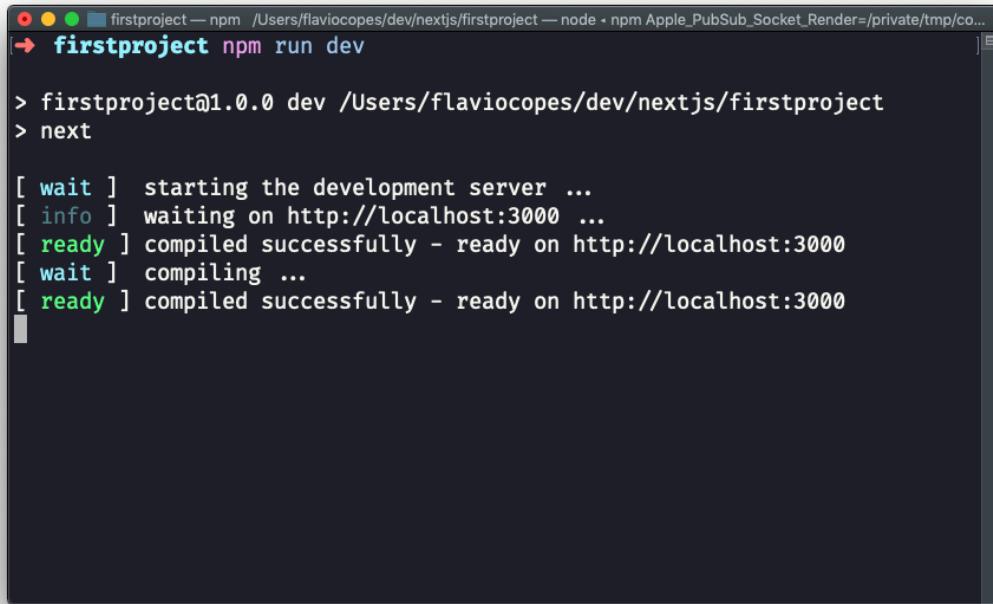
We're going to use it as the default export:

```
const Index = () => (
  <div>
    <h1>Home page</h1>
  </div>
)

export default Index
```

Learn to code — free 3,000-hour curriculum

This will make the app available on port 3000, on localhost.



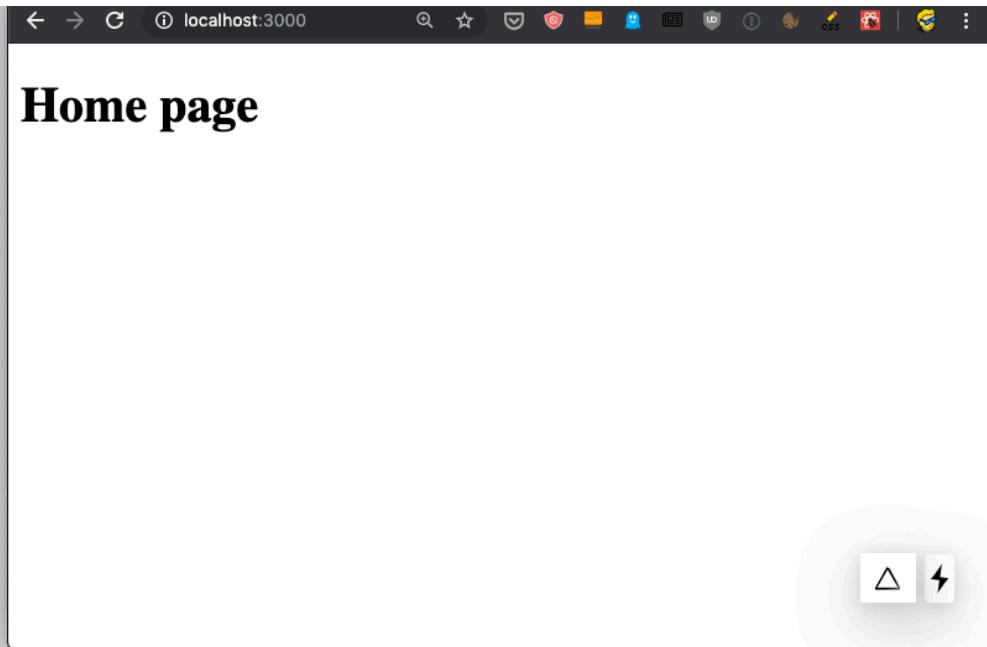
```
firstproject — npm /Users/flaviocopes/dev/nextjs/firstproject — node -e npm Apple_PubSub_Socket_Render=/private/tmp/co...
| ➔ firstproject npm run dev

> firstproject@1.0.0 dev /Users/flaviocopes/dev/nextjs/firstproject
> next

[ wait ]  starting the development server ...
[ info ]  waiting on http://localhost:3000 ...
[ ready ]  compiled successfully - ready on http://localhost:3000
[ wait ]  compiling ...
[ ready ]  compiled successfully - ready on http://localhost:3000
```

Open <http://localhost:3000> in your browser to see it.

Learn to code – free 3,000-hour curriculum



View source to confirm SSR is working

Let's now check the application is working as we expect it to work.

It's a Next.js app, so it should be **server side rendered**.

It's one of the main selling points of Next.js: if we create a site using Next.js, the site pages are rendered on the server, which delivers HTML to the browser.

This has 3 major benefits:

- The client does not need to instantiate React to render, which makes the site faster to your users.
- Search engines will index the pages without needing to run the client-side JavaScript. Something Google started doing,

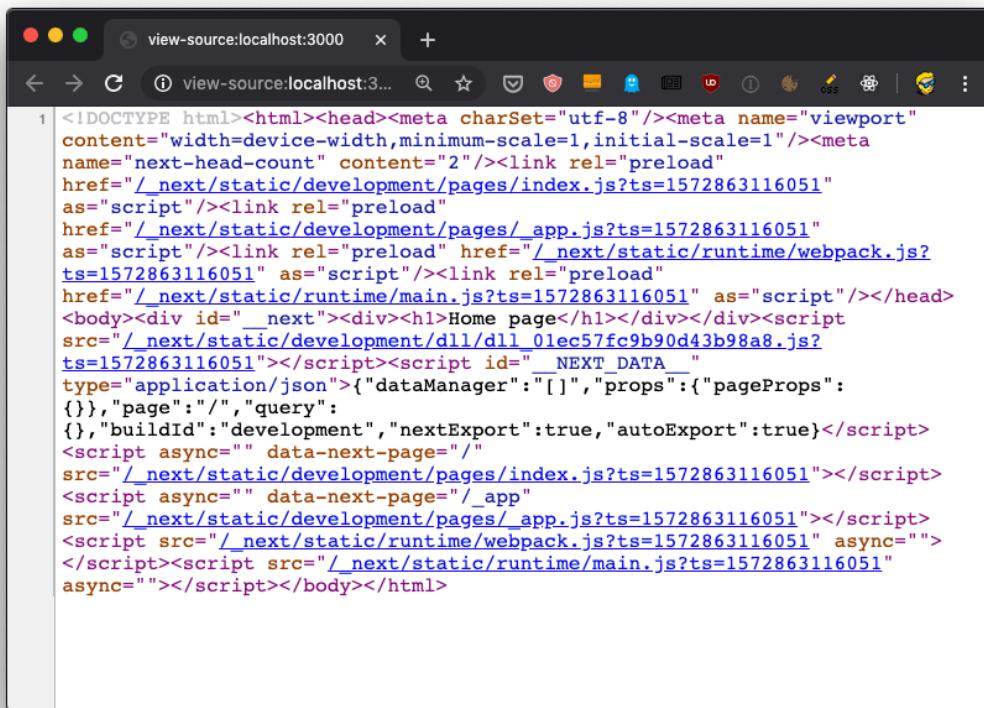
Learn to code – free 3,000-hour curriculum

- You can have social media meta tags, useful to add preview images, customize title and description for any of your pages shared on Facebook, Twitter and so on.

Let's view the source of the app.

Using Chrome you can right-click anywhere in the page, and press

View Page Source.



The screenshot shows a browser window titled "view-source:localhost:3000". The page content is the raw HTML source code of a Next.js application. It includes the standard `<!DOCTYPE html>`, `<html>`, `<head>` section with meta tags for charset, viewport, and links to static files like `/_next/static/development/pages/_app.js`. The `<body>` section contains a single `<div id="__next">` element, which itself contains an `<h1>Home page</h1>` heading. Below this, there are several `<script>` tags for JavaScript bundles, including one with the ID `__NEXT_DATA__` containing JSON data for the page manager. The entire code is wrapped in a `<html></html>` tag.

If you view the source of the page, you'll see the `<div><h1>Home page</h1></div>` snippet in the HTML body, along with a bunch of JavaScript files - the app bundles.

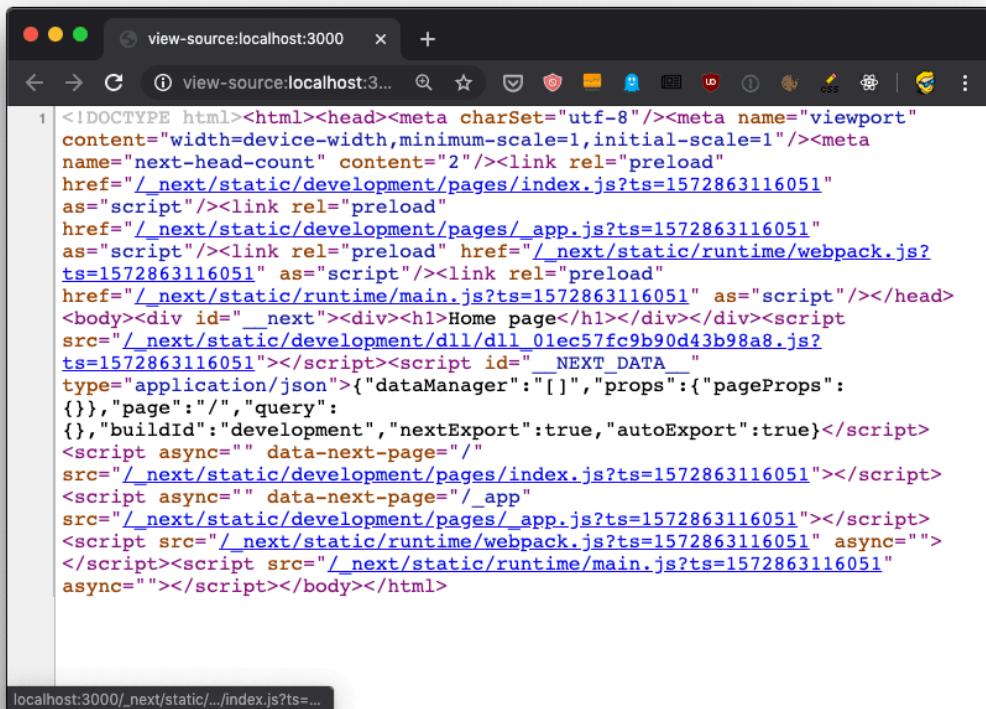
We don't need to set up anything, SSR (server-side rendering) is already working for us.

Learn to code – free 3,000-hour curriculum

Next.js there should be no difference in the result inside the browser - a server-rendered page should look exactly like a client-rendered page.

The app bundles

When we viewed the page source, we saw a bunch of JavaScript files being loaded:



A screenshot of a browser window showing the page source code. The title bar says "view-source:localhost:3000". The page content is a large block of HTML and JavaScript code, primarily in blue and black. It includes meta tags, links to static files like index.js, _app.js, and runtime/webpack.js, and a script block containing JSON data for the pageProps prop. The bottom status bar shows the URL "localhost:3000/_next/static/.../index.js?ts=...".

```
<!DOCTYPE html><html><head><meta charset="utf-8"/><meta name="viewport" content="width=device-width,minimum-scale=1,initial-scale=1"/><meta name="next-head-count" content="2"/><link rel="preload" href="/_next/static/development/pages/index.js?ts=1572863116051" as="script"/><link rel="preload" href="/_next/static/development/pages/_app.js?ts=1572863116051" as="script"/><link rel="preload" href="/_next/static/runtime/webpack.js?ts=1572863116051" as="script"/><link rel="preload" href="/_next/static/runtime/main.js?ts=1572863116051" as="script"/></head><body><div id="__next"><div><h1>Home page</h1></div></div><script src="/_next/static/development/dll/dll_01ec57fc9b90d43b98a8.js?ts=1572863116051"></script><script id="__NEXT_DATA__" type="application/json">{"dataManager":[],"props":{"pageProps":{},"page":"/","query":{},"buildId":"development","nextExport":true,"autoExport":true}}</script><script async="" data-next-page="/" src="/_next/static/development/pages/index.js?ts=1572863116051"></script><script async="" data-next-page="/_app" src="/_next/static/development/pages/_app.js?ts=1572863116051"></script><script src="/_next/static/runtime/webpack.js?ts=1572863116051" async=""></script><script src="/_next/static/runtime/main.js?ts=1572863116051" async=""></script></body></html>
```

Let's start by putting the code in an HTML formatter to get it formatted better, so we humans can get a better chance at understanding it:

Learn to code – free 3,000-hour curriculum

```

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,minimum-scale=1" />
    <meta name="next-head-count" content="2" />
    <link rel="preload" href="/_next/static/development/pages/index.js" as="script" />
    <link rel="preload" href="/_next/static/development/pages/_app.js" as="script" />
    <link rel="preload" href="/_next/static/runtime/webpack.js?ts=1572863116051" as="script" />
    <link rel="preload" href="/_next/static/runtime/main.js?ts=1572863116051" as="script" />
</head>

<body>
    <div id="__next">
        <div>
            <h1>Home page</h1></div>
        </div>
        <script src="/_next/static/development/dll/dll_01ec57fc9b90d4f3a2e0c3333333333.js" type="module" as="script" />
        <script id="__NEXT_DATA__" type="application/json">{"data":{},"pageProps":{}}</script>
        <script async="" data-next-page="/" src="/_next/static/development/pages/index.js" as="script" />
        <script async="" data-next-page="/_app" src="/_next/static/development/pages/_app.js" as="script" />
        <script src="/_next/static/runtime/webpack.js?ts=1572863116051" as="script" />
        <script src="/_next/static/runtime/main.js?ts=1572863116051" as="script" />
    </div>
</body>

</html>

```

We have 4 JavaScript files being declared to be preloaded in the head , using `rel="preload" as="script"` :

- `/_next/static/development/pages/index.js` (96 LOC)
- `/_next/static/development/pages/_app.js` (5900 LOC)
- `/_next/static/runtime/webpack.js` (939 LOC)
- `/_next/static/runtime/main.js` (12k LOC)

This tells the browser to start loading those files as soon as possible, before the normal rendering flow starts. Without those, scripts

Learn to code – free 3,000-hour curriculum

Then those 4 files are loaded at the end of the `body`, along with `/_next/static/development/dll/dll_01ec57fc9b90d43b98a8.js` (31k LOC), and a JSON snippet that sets some defaults for the page data:

```
<script id="__NEXT_DATA__" type="application/json">
{
  "dataManager": "[]",
  "props": {
    "pageProps": {}
  },
  "page": "/",
  "query": {},
  "buildId": "development",
  "nextExport": true,
  "autoExport": true
}
</script>
```

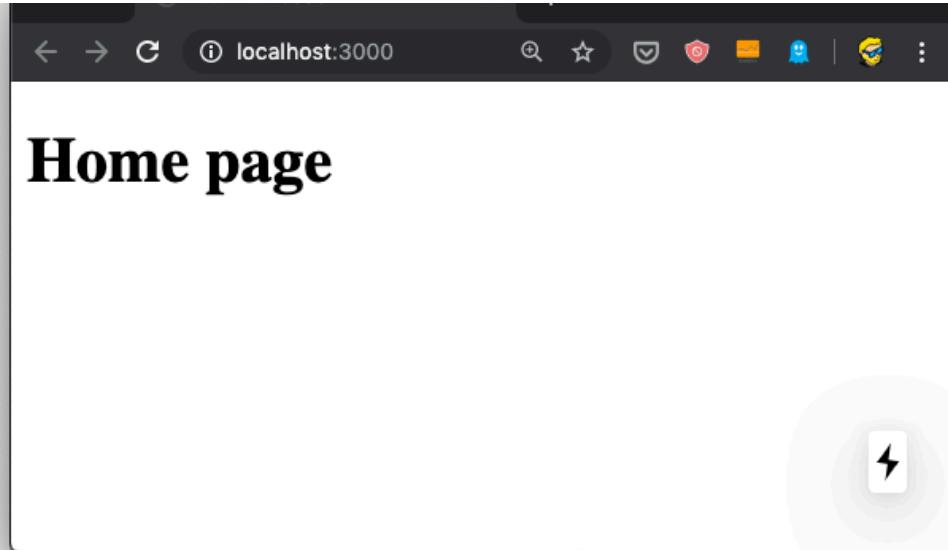
The 4 bundle files loaded are already implementing one feature called code splitting. The `index.js` file provides the code needed for the `index` component, which serves the `/` route, and if we had more pages we'd have more bundles for each page, which will then only be loaded if needed - to provide a more performant load time for the page.

What's that icon on the bottom right?

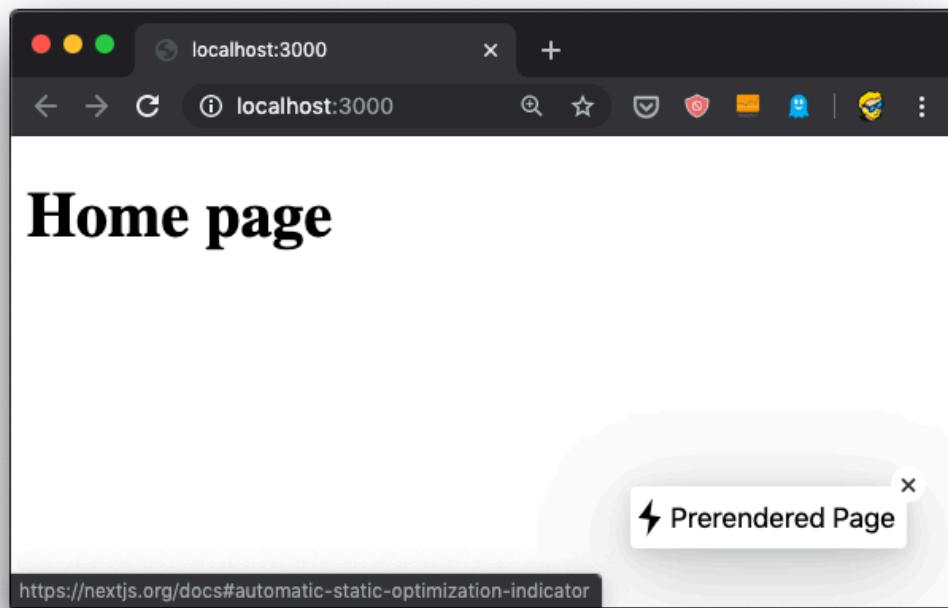
Did you see that little icon at the bottom right of the page, which looks like a lightning?

[Forum](#)[Donate](#)

Learn to code – free 3,000-hour curriculum



If you hover it, it's going to say "Prerendered Page":



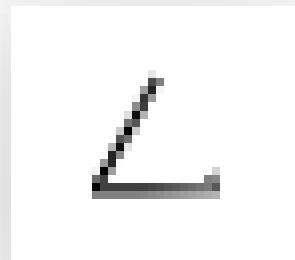
Learn to code – free 3,000-hour curriculum

fetched at invocation time, and it can be prerendered and built as a static HTML file at build time (when we run `npm run build`).

Next can determine this by the absence of the `getInitialProps()` method attached to the page component.

When this is the case, our page can be even faster because it will be served statically as an HTML file rather than going through the Node.js server that generates the HTML output.

Another useful icon that might appear next to it, or instead of it on non-prerendered pages, is a little animated triangle:



This is a compilation indicator, and appears when you save a page and Next.js is compiling the application before hot code reloading kicks in to reload the code in the application automatically.

It's a really nice way to immediately determine if the app has already been compiled and you can test a part of it you're working on.

Learn to code – free 3,000-hour curriculum

to install (if you haven't already) is the React Developer Tools.

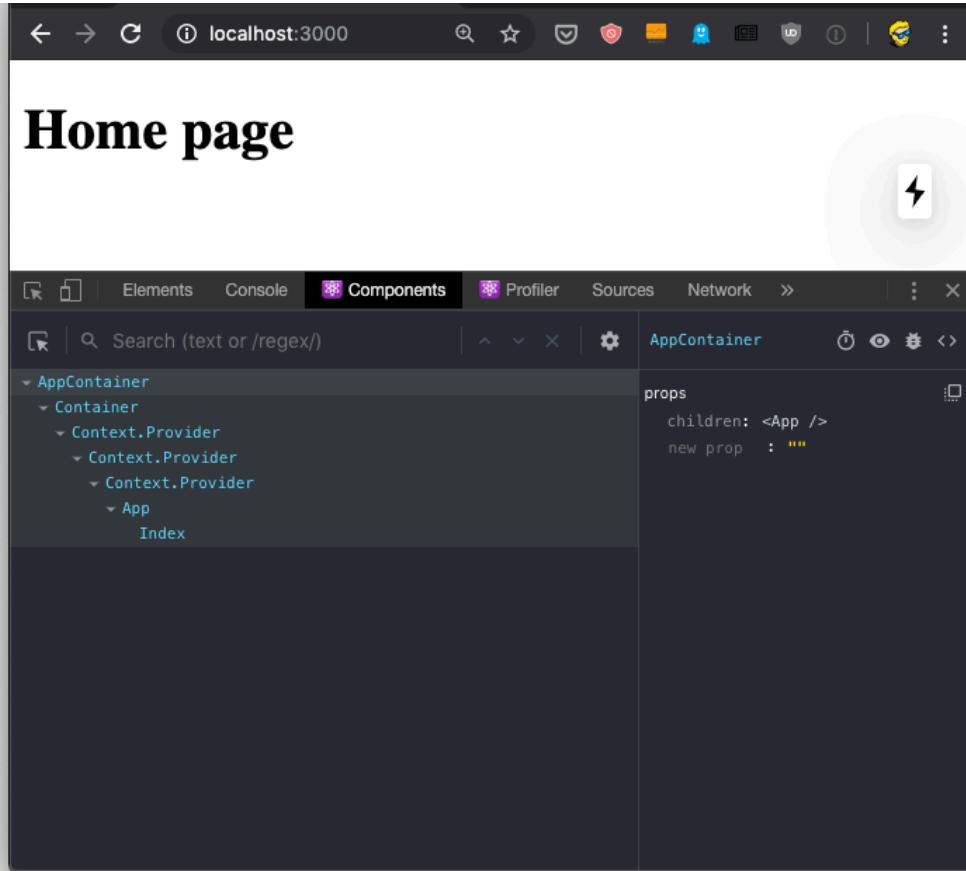
Available for both [Chrome](#) and [Firefox](#), the React Developer Tools are an essential instrument you can use to inspect a React application.

Now, the React Developer Tools are not specific to Next.js but I want to introduce them because you might not be 100% familiar with all the tools React provides. It's best to go a little into debugging tooling than assuming you already know them.

They provide an inspector that reveals the React components tree that builds your page, and for each component you can go and check the props, the state, hooks, and lots more.

Once you have installed the React Developer Tools, you can open the regular browser devtools (in Chrome, it's right-click in the page, then click `Inspect`) and you'll find 2 new panels: **Components** and **Profiler**.

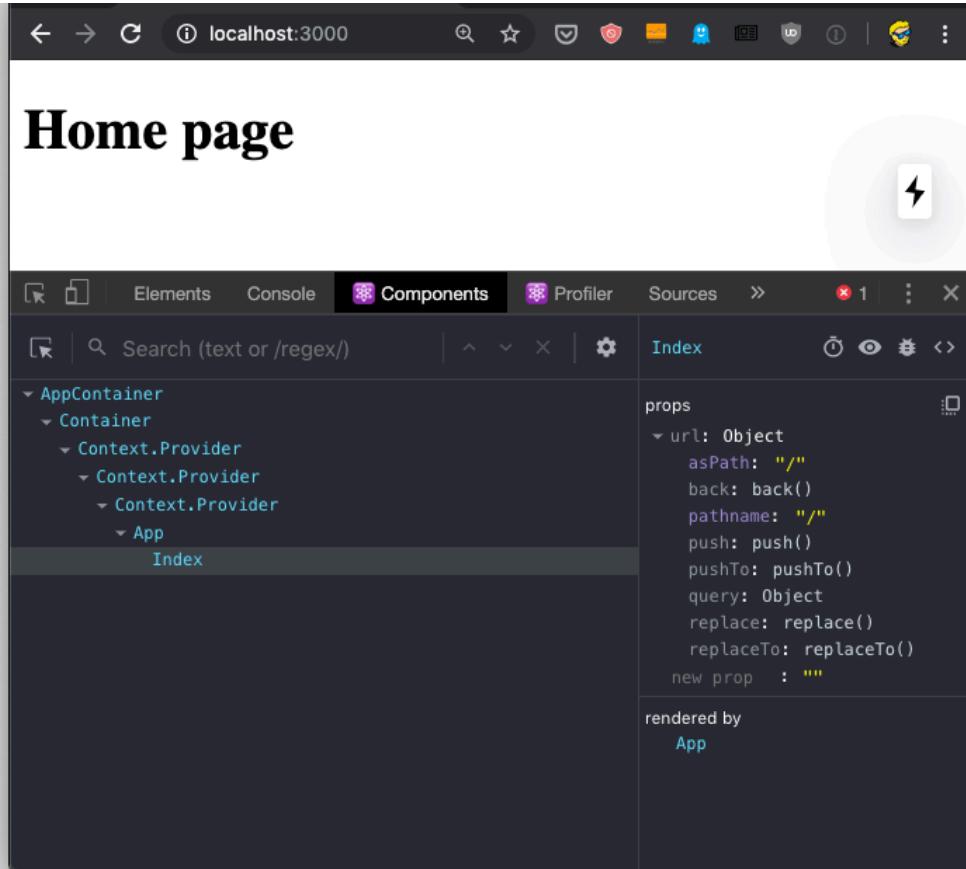
Learn to code – free 3,000-hour curriculum



If you move the mouse over the components, you'll see that in the page, the browser will select the parts that are rendered by that component.

If you select any component in the tree, the right panel will show you a reference to the parent component, and the props passed to it:

Learn to code – free 3,000-hour curriculum

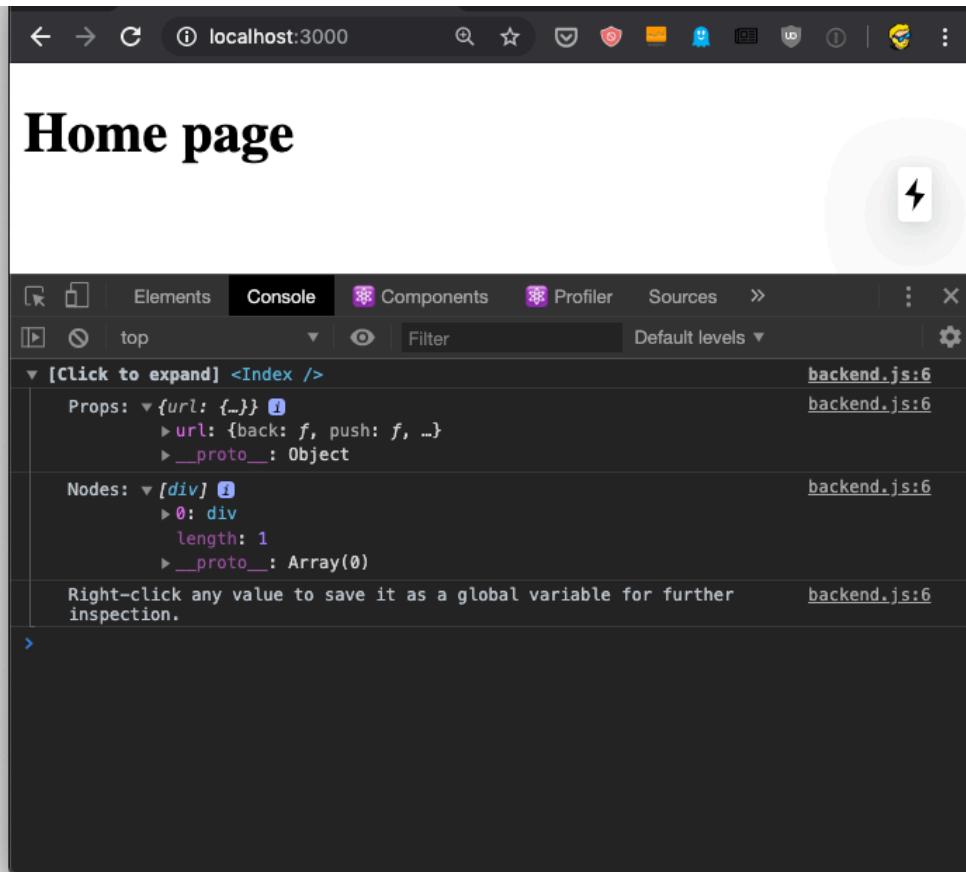


You can easily navigate by clicking around the component names.

You can click the eye icon in the Developer Tools toolbar to inspect the DOM element, and also if you use the first icon, the one with the mouse icon (which conveniently sits under the similar regular DevTools icon), you can hover an element in the browser UI to directly select the React component that renders it.

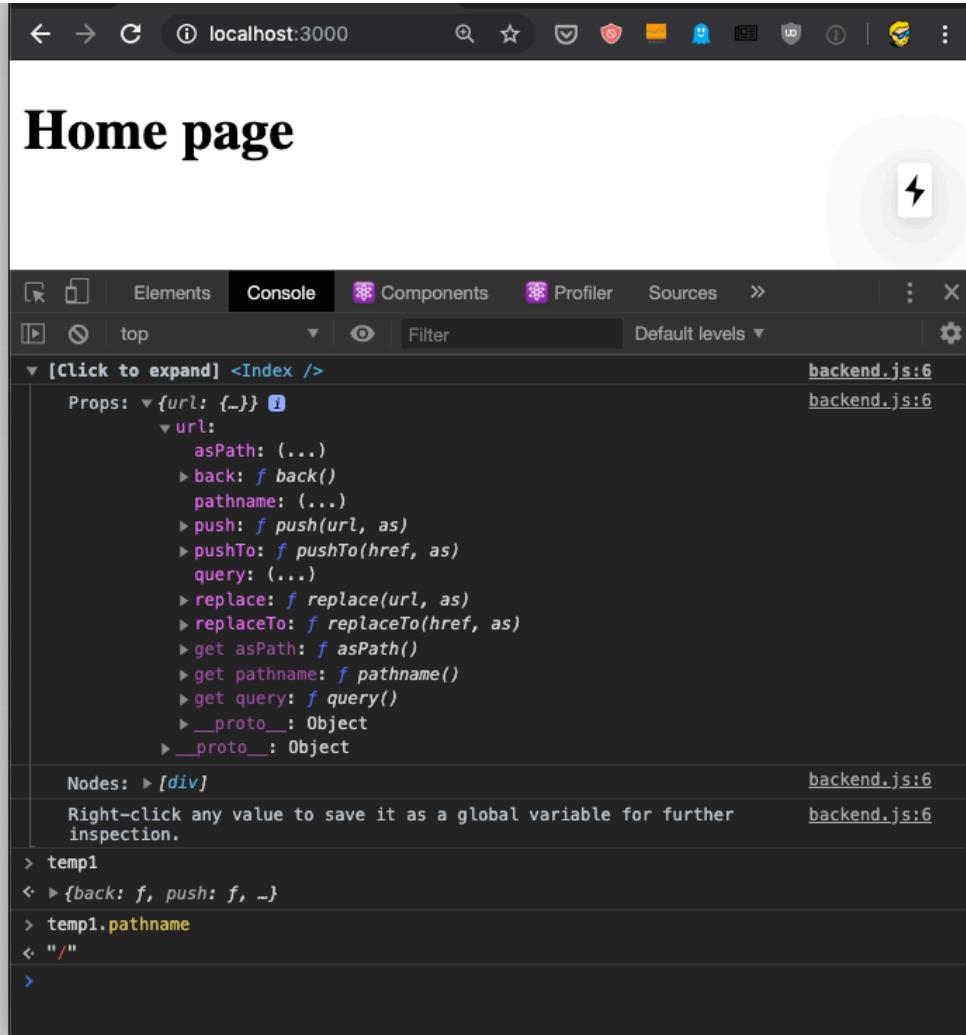
You can use the `bug` icon to log a component data to the console.

Learn to code – free 3,000-hour curriculum



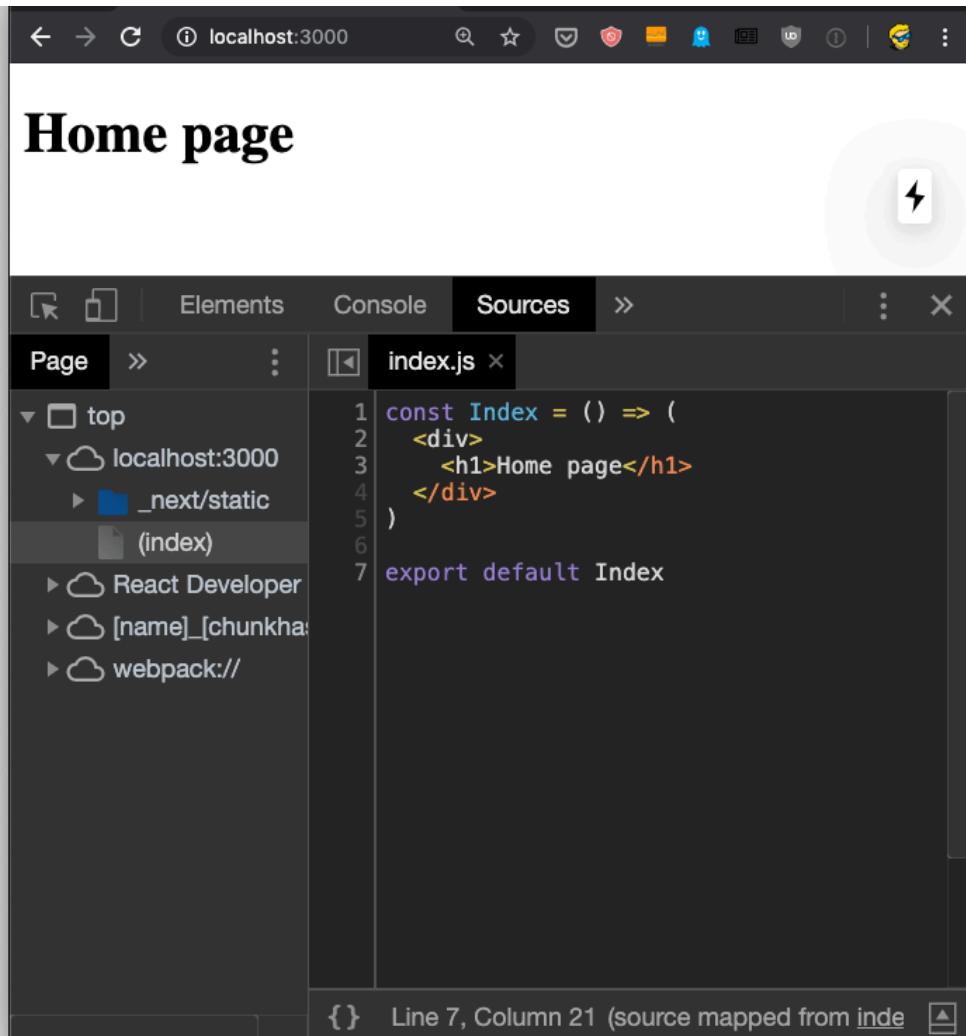
This is pretty awesome because once you have the data printed there, you can right-click any element and press "Store as a global variable". For example here I did it with the `url` prop, and I was able to inspect it in the console using the temporary variable assigned to it, `temp1`:

Learn to code – free 3,000-hour curriculum



Using **Source Maps**, which are loaded by Next.js automatically in development mode, from the Components panel we can click the `<>` code and the DevTools will switch to the Source panel, showing us the component source code:

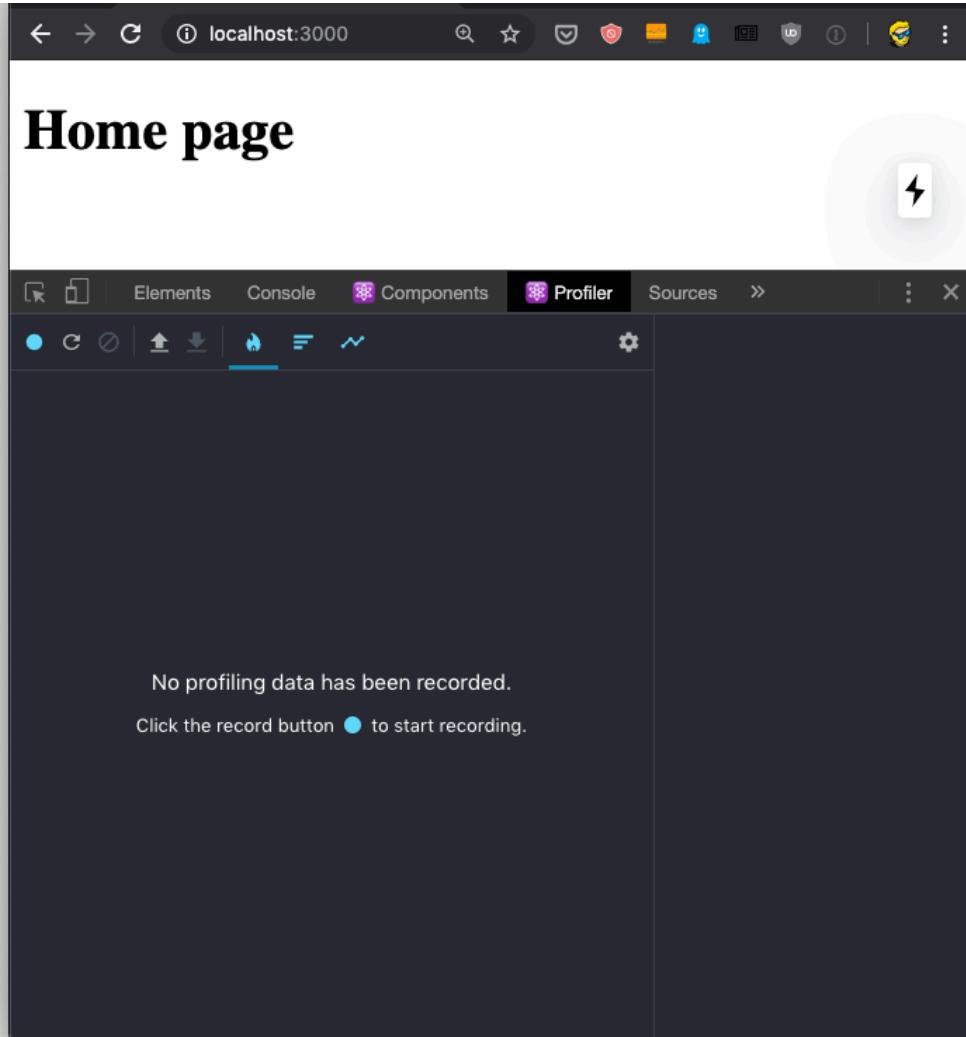
Learn to code – free 3,000-hour curriculum



```
const Index = () => (
  <div>
    <h1>Home page</h1>
  </div>
)
export default Index
```

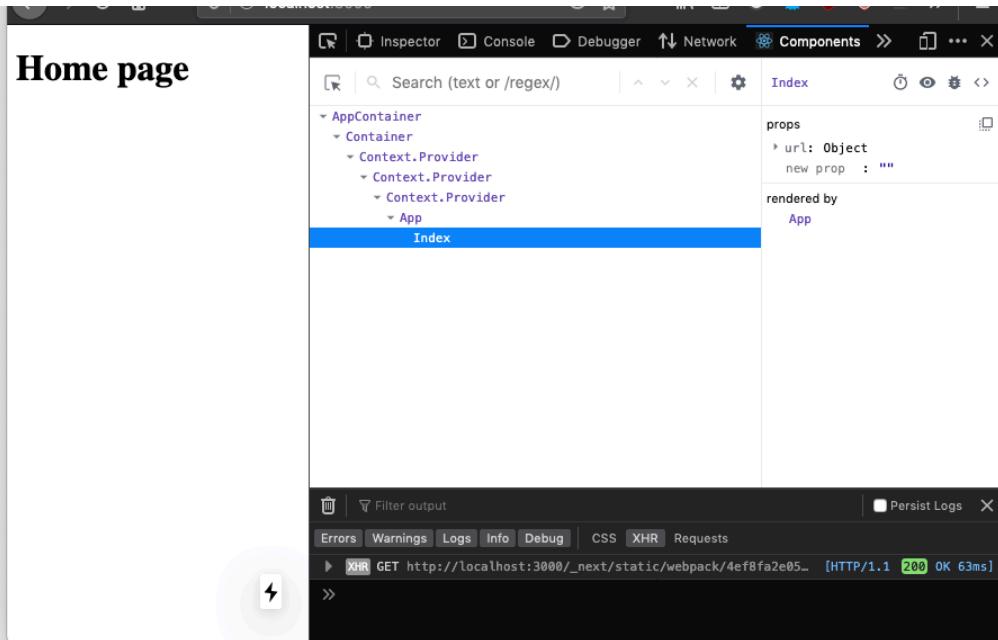
The **Profiler** tab is even more awesome, if possible. It allows us to **record an interaction** in the app, and see what happens. I cannot show an example yet, because it needs at least 2 components to create an interaction, and we have just one now. I'll talk about this later.

Learn to code – free 3,000-hour curriculum



I showed all screenshots using Chrome, but the React Developer Tools works in the same way in Firefox:

Learn to code – free 3,000-hour curriculum



Other debugging techniques you can use

In addition to the React Developer Tools, which are essential to building a Next.js application, I want to emphasize 2 ways to debug Next.js apps.

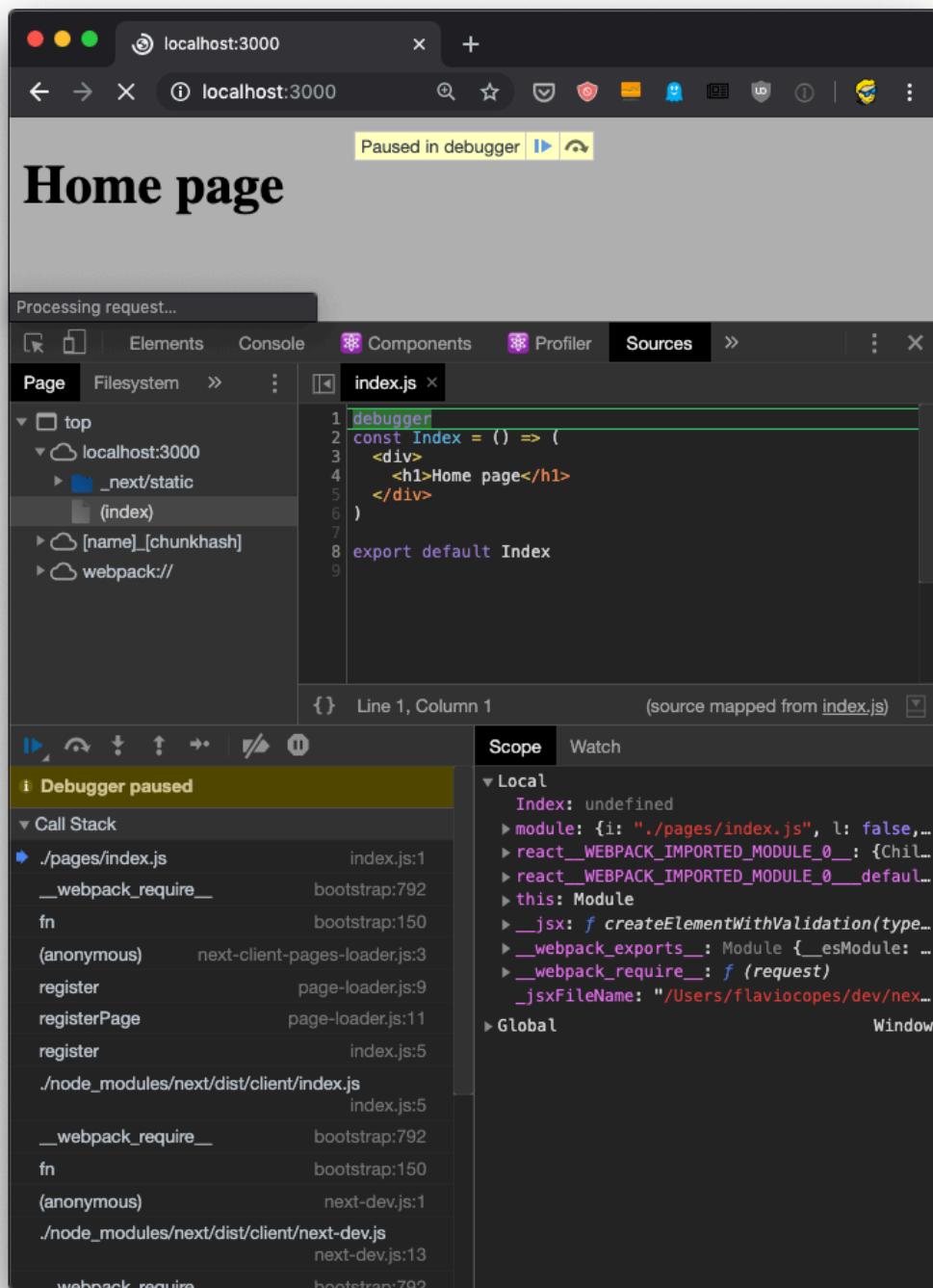
The first is obviously `console.log()` and all the [other Console API](#) tools. The way Next apps work will make a log statement work in the browser console OR in the terminal where you started Next using `npm run dev`.

In particular, if the page loads from the server, when you point the URL to it, or you hit the refresh button / cmd/ctrl-R, any console logging happens in the terminal.

Learn to code — free 3,000-hour curriculum

Just remember if you are surprised by missing logging.

Another tool that is essential is the `debugger` statement. Adding this statement to a component will pause the browser rendering the page:



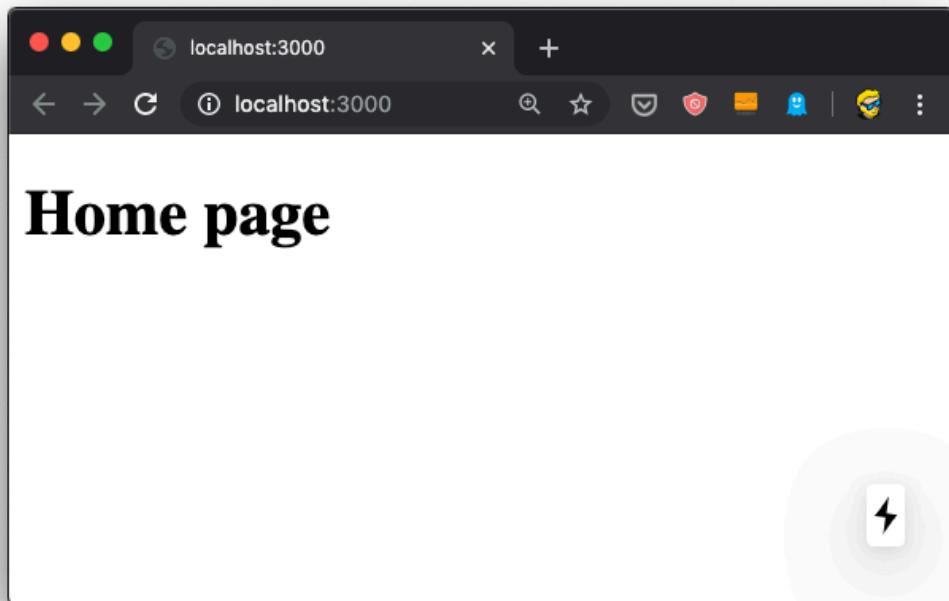
Learn to code – free 3,000-hour curriculum

inspect values and run your app one line at a time.

You can also use the VS Code debugger to debug server-side code. I mention this technique and [this tutorial](#) to set this up.

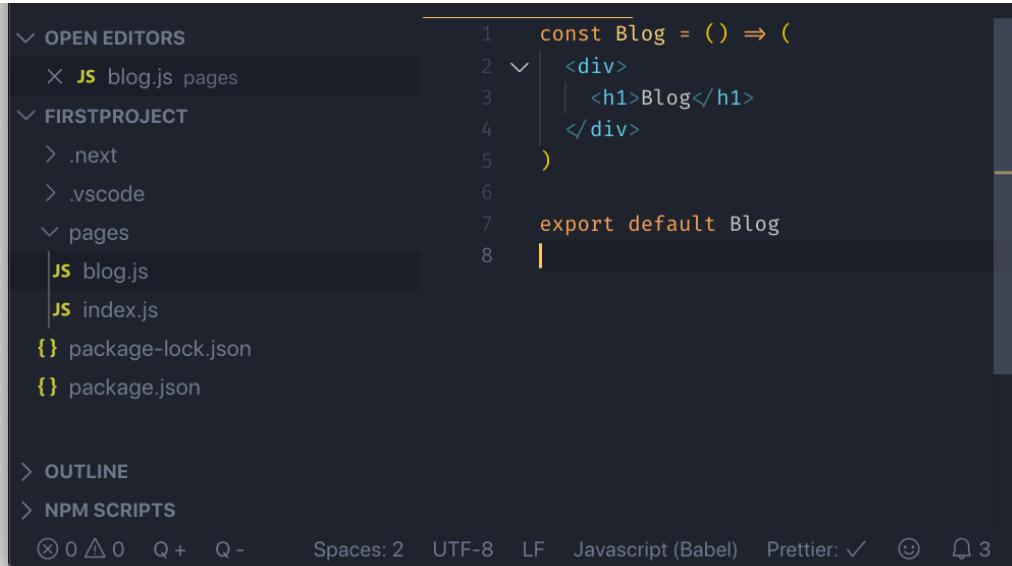
Adding a second page to the site

Now that we have a good grasp of the tools we can use to help us develop Next.js apps, let's continue from where we left our first app:



I want to add a second page to this website, a blog. It's going to be served into `/blog`, and for the time being it will just contain a simple static page, just like our first `index.js` component:

Learn to code – free 3,000-hour curriculum



```
1  const Blog = () => (
2    <div>
3      <h1>Blog</h1>
4    </div>
5  )
6
7  export default Blog
8
```

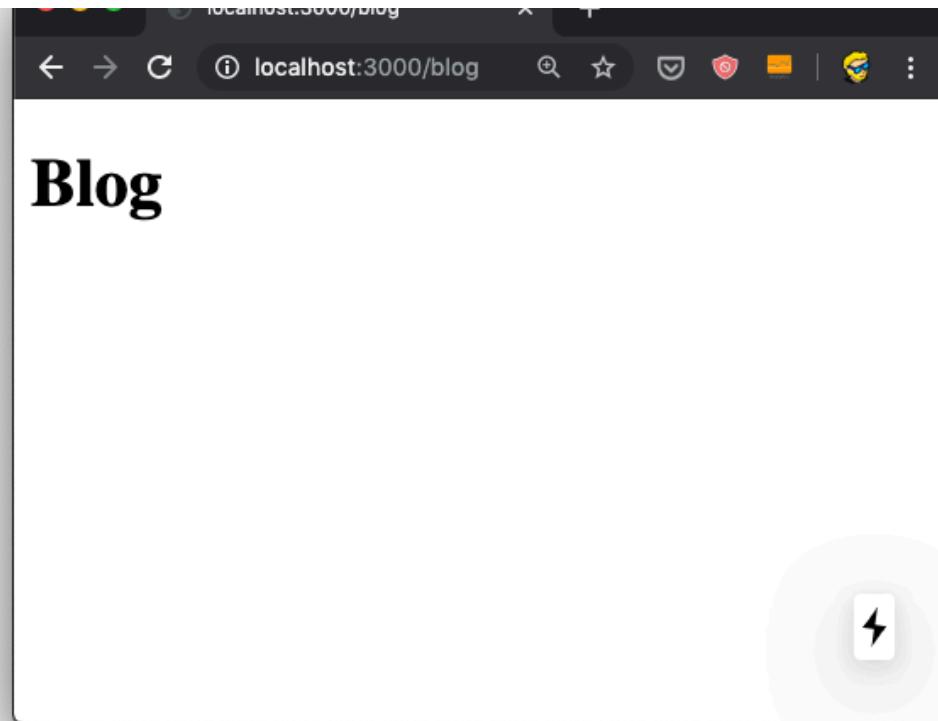
The code editor interface shows a sidebar with project files: OPEN EDITORS (JS blog.js pages), FIRSTPROJECT (.next, .vscode, pages), and a selected JS blog.js file. Below the sidebar is the main code editor area with the code above. At the bottom, there are navigation icons (refresh, search, etc.) and status indicators: Spaces: 2, UTF-8, LF, Javascript (Babel), Prettier: ✓, and a file count of 3.

After saving the new file, the `npm run dev` process already running is already capable of rendering the page, without the need to restart it.

When we hit the URL <http://localhost:3000/blog> we have the new page:

[Forum](#)[Donate](#)

Learn to code — free 3,000-hour curriculum



and here's what the terminal told us:

```
[ event ] build page: /blog
[ wait ]  compiling ...
[ ready ] compiled successfully - ready on http://localhost:3000
```

A screenshot of a terminal window. The window title is 'firstproject — npm /Users/flaviocopes/dev/nextjs/firstproject — node -v npm Apple_PubSub_Socket_Render=/private/t...'. The terminal output shows three lines of text: '[event] build page: /blog', '[wait] compiling ...', and '[ready] compiled successfully - ready on http://localhost:3000'. The background of the terminal window is dark.

Learn to code – free 3,000-hour curriculum

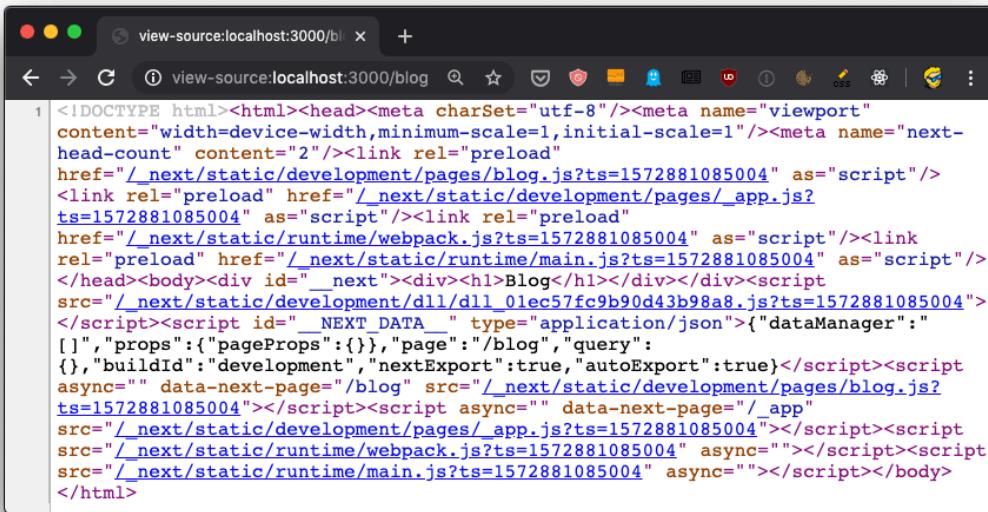
You can create a `pages/hey/ho` page, and that page will show up on the URL <http://localhost:3000/hey/ho>.

What does not matter, for the URL purposes, is the component name inside the file.

Try going and viewing the source of the page, when loaded from the server it will list `/_next/static/development/pages/blog.js` as one of the bundles loaded, and not

`/_next/static/development/pages/index.js` like in the home page.

This is because thanks to automatic code splitting we don't need the bundle that serves the home page. Just the bundle that serves the blog page.



The screenshot shows the 'view-source' tab of a browser window. The address bar indicates the URL is `view-source:localhost:3000/blog`. The page content is the raw HTML source code of the blog page. It includes standard head meta tags, a link to the blog page's static bundle (`blog.js?ts=1572881085004`), and links to the app bundle (`app.js?ts=1572881085004`) and runtime webpack bundle (`runtime/webpack.js?ts=1572881085004`). The body contains a `<h1>Blog</h1>` heading and a `<script id="__NEXT_DATA__" type="application/json">` block containing JSON data for the data manager. The JSON object includes properties like `dataManager`, `props` (with `pageProps`), `page` (set to `/blog`), `query`, `buildId` (set to `"development"`), and `nextExport` (set to `true`). There are also `autoExport` and `data-next-page` attributes set to `/blog`.

```

<!DOCTYPE html><html><head><meta charset="utf-8" /><meta name="viewport" content="width=device-width,minimum-scale=1,initial-scale=1" /><meta name="next-head-count" content="2" /><link rel="preload" href="/_next/static/development/pages/blog.js?ts=1572881085004" as="script"/><link rel="preload" href="/_next/static/development/pages/_app.js?ts=1572881085004" as="script"/><link rel="preload" href="/_next/static/runtime/webpack.js?ts=1572881085004" as="script"/><link rel="preload" href="/_next/static/runtime/main.js?ts=1572881085004" as="script"/></head><body><div id="__next"><div><h1>Blog</h1></div></div><script src="/_next/static/development/dll/dll_01ec57fc9b90d43b98a8.js?ts=1572881085004"></script><script id="__NEXT_DATA__" type="application/json">{"dataManager": [], "props": {"pageProps": {}}, "page": "/blog", "query": {}, "buildId": "development", "nextExport": true, "autoExport": true}</script><script async="" data-next-page="/blog" src="/_next/static/development/pages/blog.js?ts=1572881085004"></script><script async="" data-next-page="/_app" src="/_next/static/development/pages/_app.js?ts=1572881085004"></script><script src="/_next/static/runtime/webpack.js?ts=1572881085004" async=""></script><script src="/_next/static/runtime/main.js?ts=1572881085004" async=""></script></body></html>

```

We can also just export an anonymous function from `blog.js`:

Learn to code – free 3,000-hour curriculum

```
</div>
)
```

or if you prefer the non-arrow function syntax:

```
export default function() {
  return (
    <div>
      <h1>Blog</h1>
    </div>
  )
}
```

Linking the two pages

Now that we have 2 pages, defined by `index.js` and `blog.js`, we can introduce links.

Normal HTML links within pages are done using the `a` tag:

```
<a href="/blog">Blog</a>
```

We can't do that in Next.js.

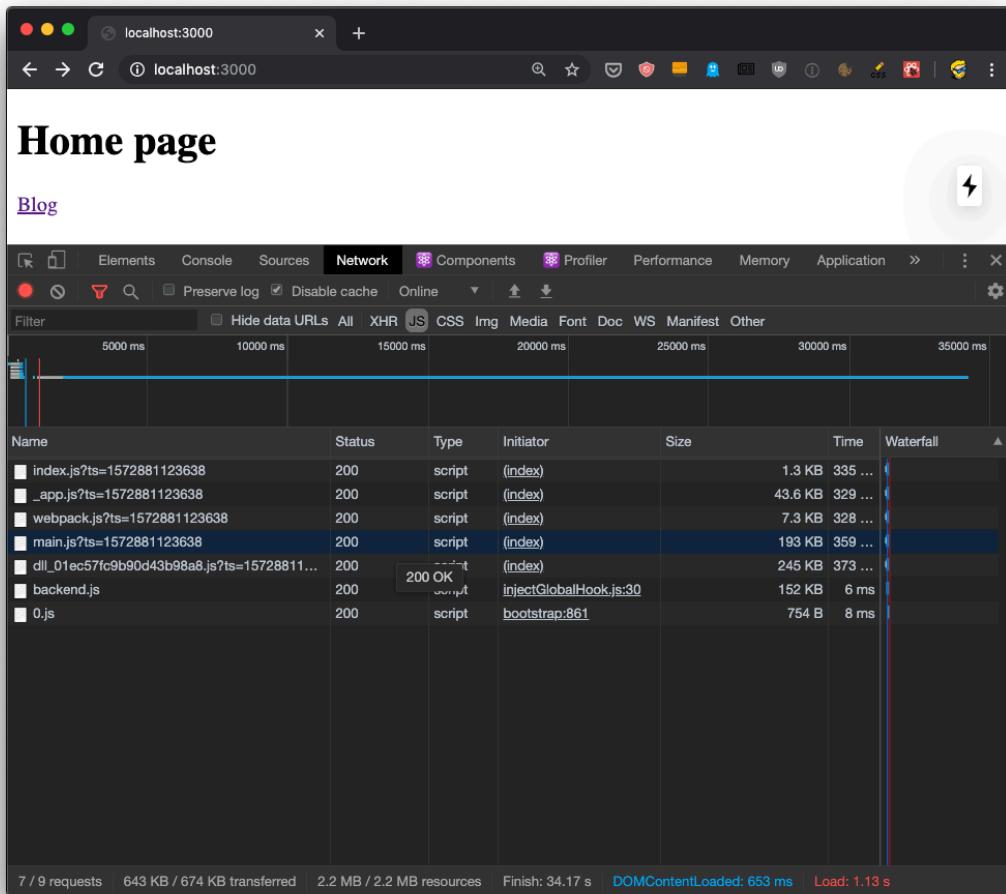
Why? We technically *can*, of course, because this is the Web and *on the Web things never break* (that's why we can still use the `<marquee>` tag). But one of the main benefits of using Next is that once a page is loaded, transitions to other page are very fast thanks to client-side rendering.

Learn to code – free 3,000-hour curriculum

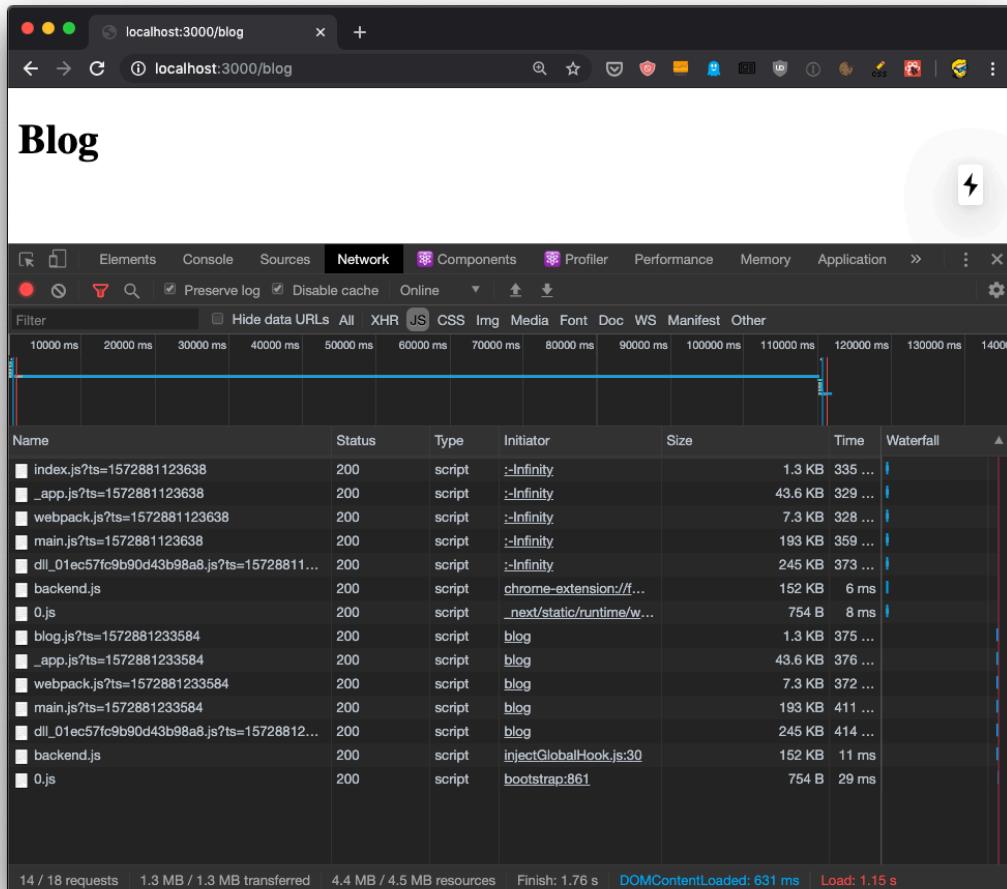
```
const Index = () => (
  <div>
    <h1>Home page</h1>
    <a href='/blog'>Blog</a>
  </div>
)

export default Index
```

Now open the **DevTools**, and the **Network** panel in particular. The first time we load `http://localhost:3000/` we get all the page bundles loaded:



Learn to code – free 3,000-hour curriculum



We got all that JavaScript from the server, again! But.. we don't need all that JavaScript if we already got it. We'd just need the `blog.js` page bundle, the only one that's new to the page.

To fix this problem, we use a component provided by Next, called Link.

We import it:

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

Learn to code – free 3,000-hour curriculum

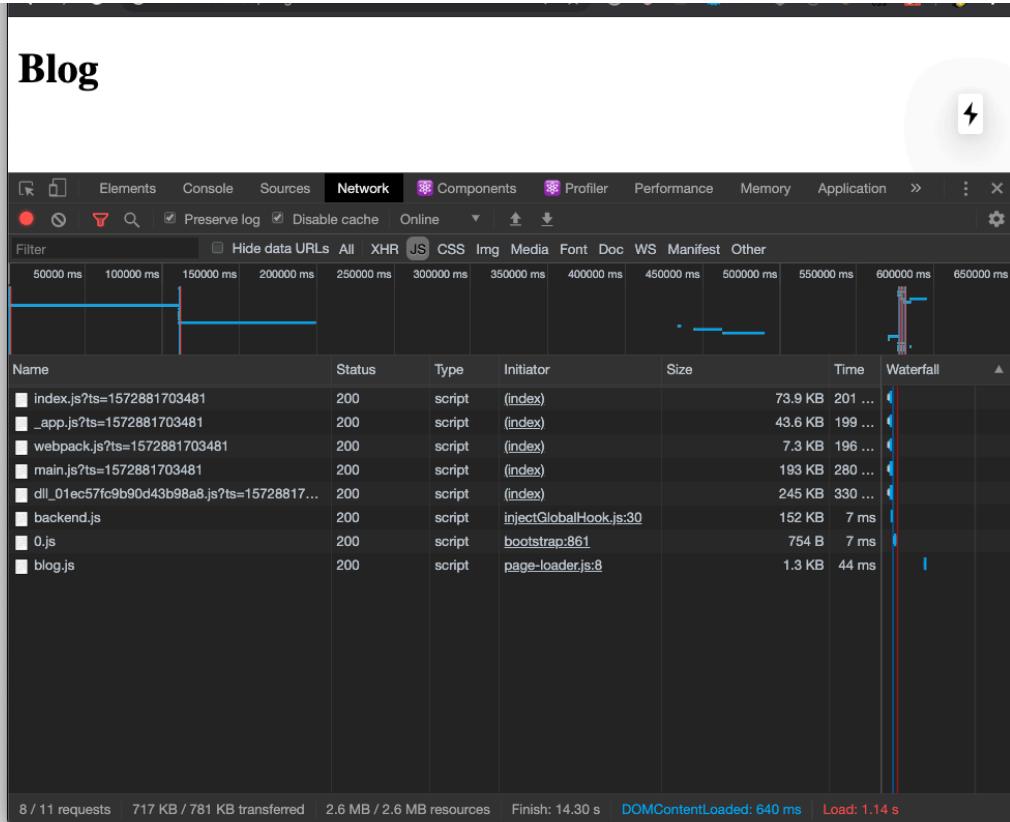
```
import Link from 'next/link'

const Index = () => (
  <div>
    <h1>Home page</h1>
    <Link href='/blog'>
      <a>Blog</a>
    </Link>
  </div>
)

export default Index
```

Now if you retry the thing we did previously, you'll be able to see that only the `blog.js` bundle is loaded when we move to the blog page:

Learn to code – free 3,000-hour curriculum



and the page loaded so faster than before, the browser usual spinner on the tab didn't even appear. Yet the URL changed, as you can see. This is working seamlessly with the browser [History API](#).

This is client-side rendering in action.

What if you now press the back button? Nothing is being loaded, because the browser still has the old `index.js` bundle in place, ready to load the `/index` route. It's all automatic!

Dynamic content with the router

In the previous chapter we saw how to link the home to the blog page.

Learn to code – free 3,000-hour curriculum

Blog posts have a dynamic URL. For example a post titled "Hello World" might have the URL `/blog/hello-world`. A post titled "My second post" might have the URL `/blog/my-second-post`.

This content is dynamic, and might be taken from a database, markdown files or more.

Next.js can serve dynamic content based on a **dynamic URL**.

We create a dynamic URL by creating a dynamic page with the `[]` syntax.

How? We add a `pages/blog/[id].js` file. This file will handle all the dynamic URLs under the `/blog/` route, like the ones we mentioned above: `/blog/hello-world`, `/blog/my-second-post` and more.

In the file name, `[id]` inside the square brackets means that anything that's dynamic will be put inside the `id` parameter of the **query property of the router**.

Ok, that's a bit too many things at once.

What's the router?

The router is a library provided by Next.js.

We import it from `next/router`:

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

Learn to code – free 3,000-hour curriculum

```
const router = useRouter()
```

Once we have this router object, we can extract information from it.

In particular we can get the dynamic part of the URL in the `[id].js` file by accessing `router.query.id`.

The dynamic part can also just be a portion of the URL, like `post-[id].js`.

So let's go on and apply all those things in practice.

Create the file `pages/blog/[id].js`:

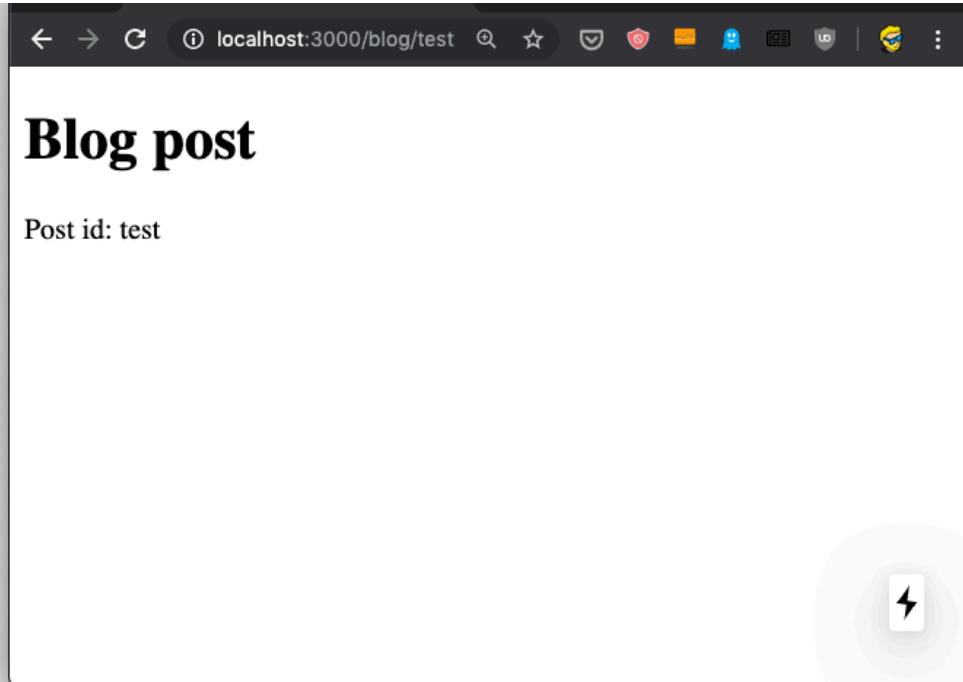
```
import { useRouter } from 'next/router'

export default () => {
  const router = useRouter()

  return (
    <>
      <h1>Blog post</h1>
      <p>Post id: {router.query.id}</p>
    </>
  )
}
```

Now if you go to the `http://localhost:3000/blog/test` router, you should see this:

Learn to code – free 3,000-hour curriculum



We can use this `id` parameter to gather the post from a list of posts. From a database, for example. To keep things simple we'll add a `posts.json` file in the project root folder:

```
{
  "test": {
    "title": "test post",
    "content": "Hey some post content"
  },
  "second": {
    "title": "second post",
    "content": "Hey this is the second post content"
  }
}
```

Now we can import it and lookup the post from the `id` key:

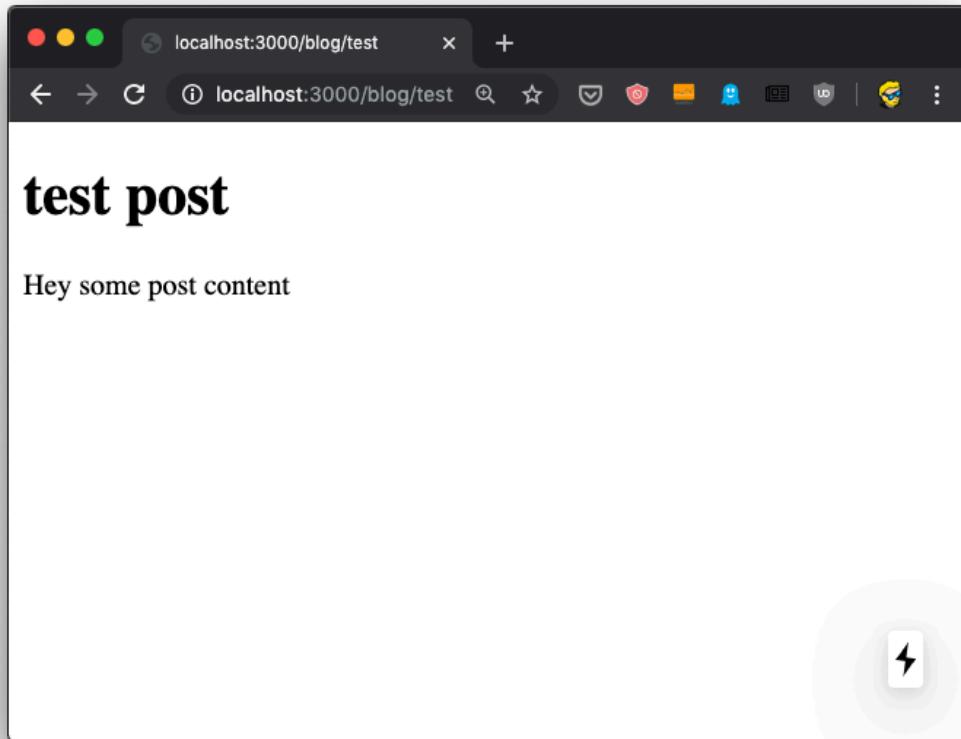
Learn to code – free 3,000-hour curriculum

```
export default () => {
  const router = useRouter()

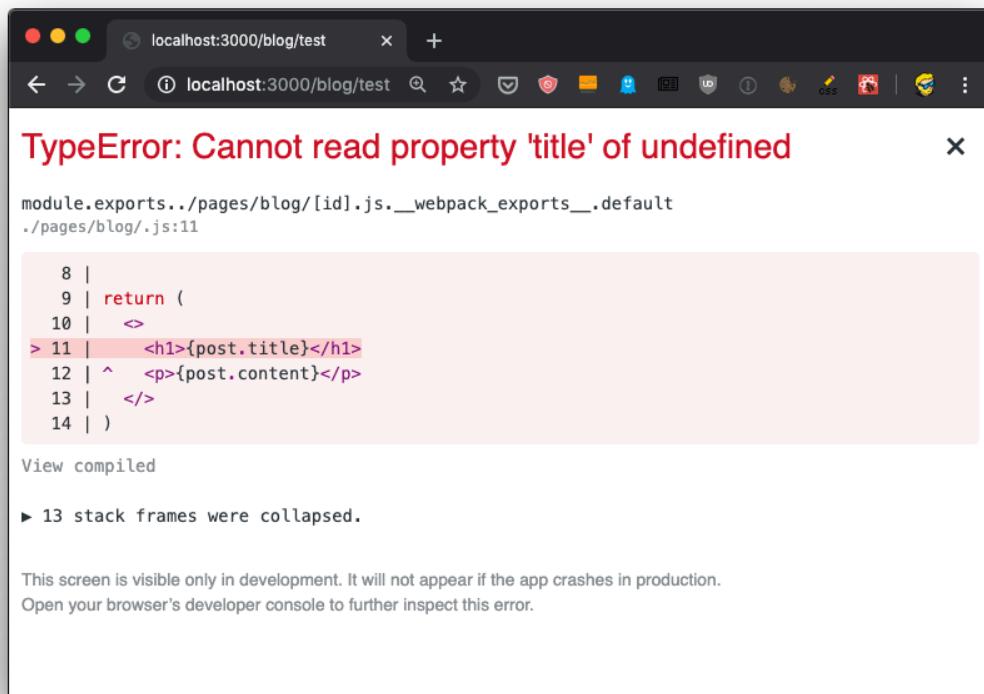
  const post = posts[router.query.id]

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

Reloading the page should show us this result:



Learn to code – free 3,000-hour curriculum



Why? Because.. during rendering, when the component is initialized, the data is not there yet. We'll see how to provide the data to the component with `getInitialProps` in the next lesson.

For now, add a little `if (!post) return <p></p>` check before returning the JSX:

```
import { useRouter } from 'next/router'
import posts from '../..../posts.json'

export default () => {
  const router = useRouter()

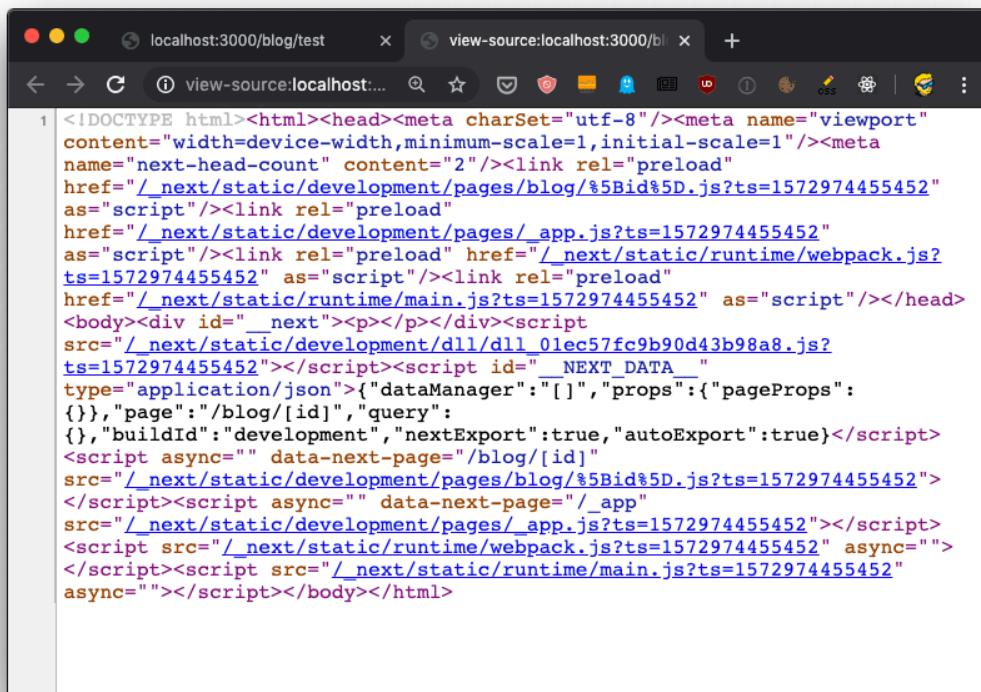
  const post = posts[router.query.id]
  if (!post) return <p></p>
```

Learn to code – free 3,000-hour curriculum

```
</>
)
}
```

Now things should work. Initially the component is rendered without the dynamic `router.query.id` information. After rendering, Next.js triggers an update with the query value and the page displays the correct information.

And if you view source, there is that empty `<p>` tag in the HTML:



The screenshot shows a browser window with the URL `localhost:3000/blog/test` in the address bar. The title bar says "view-source:localhost:...". The main content area displays the generated HTML code. The code includes a DOCTYPE declaration, meta tags for charset, viewport, and device width, and various script tags for Next.js runtime and static assets. A prominent empty `<p></p>` tag is visible within a `<div id="__next">` container.

```
<!DOCTYPE html><html><head><meta charset="utf-8"/><meta name="viewport" content="width=device-width,minimum-scale=1,initial-scale=1"/><meta name="next-head-count" content="2"/><link rel="preload" href="/_next/static/development/pages/blog/%5Bid%5D.js?ts=1572974455452" as="script"/><link rel="preload" href="/_next/static/development/pages/_app.js?ts=1572974455452" as="script"/><link rel="preload" href="/_next/static/runtime/webpack.js?ts=1572974455452" as="script"/><link rel="preload" href="/_next/static/runtime/main.js?ts=1572974455452" as="script"/></head><body><div id="__next"><p></p></div><script src="/_next/static/development/dll/dll_01ec57fc9b90d43b98a8.js?ts=1572974455452"></script><script id="__NEXT_DATA__" type="application/json">{"dataManager":[],"props":{"pageProps":{}}, "page":"/blog/[id]","query":{},"buildId":"development","nextExport":true,"autoExport":true}</script><script async="" data-next-page="/blog/[id]" src="/_next/static/development/pages/blog/%5Bid%5D.js?ts=1572974455452"></script><script async="" data-next-page="/_app" src="/_next/static/development/pages/_app.js?ts=1572974455452"></script><script src="/_next/static/runtime/webpack.js?ts=1572974455452" as="script"></script><script src="/_next/static/runtime/main.js?ts=1572974455452" as="script"></script></body></html>
```

We'll soon fix this issue that fails to implement SSR and this harms both loading times for our users, SEO and social sharing as we already discussed.

Learn to code – free 3,000-hour curriculum

```
import posts from '../posts.json'

const Blog = () => (
  <div>
    <h1>Blog</h1>

    <ul>
      {Object.entries(posts).map((value, index) => {
        return <li key={index}>{value[1].title}</li>
      })}
    </ul>
  </div>
)

export default Blog
```

And we can link them to the individual post pages, by importing `Link` from `next/link` and using it inside the posts loop:

```
import Link from 'next/link'
import posts from '../posts.json'

const Blog = () => (
  <div>
    <h1>Blog</h1>

    <ul>
      {Object.entries(posts).map((value, index) => {
        return (
          <li key={index}>
            <Link href={`/blog/[id]`} as={`/blog/' + value[0]}>
              <a>{value[1].title}</a>
            </Link>
          </li>
        )
      })}
    </ul>
  </div>
)
```

[Learn to code – free 3,000-hour curriculum](#)

Prefetching

I mentioned previously how the `Link` Next.js component can be used to create links between 2 pages, and when you use it, Next.js **transparently handles frontend routing** for us, so when a user clicks a link, frontend takes care of showing the new page without triggering a new client/server request and response cycle, as it normally happens with web pages.

There's another thing that Next.js does for you when you use `Link`.

As soon as an element wrapped within `<Link>` appears in the viewport (which means it's visible to the website user), Next.js prefetches the URL it points to, as long as it's a local link (on your website), making the application super fast to the viewer.

This behavior is only being triggered in **production mode** (we'll talk about this in-depth later), which means you have to stop the application if you are running it with `npm run dev`, compile your production bundle with `npm run build` and run it with `npm run start` instead.

Using the Network inspector in the DevTools you'll notice that any links above the fold, at page load, start the prefetching as soon as the `load` event has been fired on your page (triggered when the page is fully loaded, and happens after the `DOMContentLoaded` event).

Any other `Link` tag not in the viewport will be prefetched when the user scrolls and it

Learn to code – free 3,000-hour curriculum

Header.

You can opt out from prefetching individual `Link` instances by setting the `prefetch` prop to `false`:

```
<Link href="/a-link" prefetch={false}>
  <a>A link</a>
</Link>
```

Using the router to detect the active link

One very important feature when working with links is determining what is the current URL, and in particular assigning a class to the active link, so we can style it differently from the other ones.

This is especially useful in your site header, for example.

The Next.js default `Link` component offered in `next/link` does not do this automatically for us.

We can create a `Link` component ourselves, and we store it in a file `Link.js` in the Components folder, and import that instead of the default `next/link`.

In this component, we'll first import React from `react`, `Link` from `next/link` and the `useRouter` hook from `next/router`.

Inside the component we determine if the current path name matches the `href` prop of the component, and if so we append the

Learn to code – free 3,000-hour curriculum

`React.cloneElement()`:

```
import React from 'react'
import Link from 'next/link'
import { useRouter } from 'next/router'

export default ({ href, children }) => {
  const router = useRouter()

  let className = children.props.className || ''
  if (router.pathname === href) {
    className = `${className} selected`
  }

  return <Link href={href}>{React.cloneElement(children, { class: className })}</Link>
}
```

Using `next/router`

We already saw how to use the `Link` component to declaratively handle routing in Next.js apps.

It's really handy to manage routing in JSX, but sometimes you need to trigger a routing change programmatically.

In this case, you can access the Next.js Router directly, provided in the `next/router` package, and call its `push()` method.

Here's an example of accessing the router:

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

Learn to code – free 3,000-hour curriculum

Once we get the router object by invoking `useRouter()`, we can use its methods.

This is the client side router, so methods should only be used in frontend facing code. The easiest way to ensure this is to wrap calls in the `useEffect()` React hook, or inside `componentDidMount()` in React stateful components.

The ones you'll likely use the most are `push()` and `prefetch()`.

`push()` allows us to programmatically trigger a URL change, in the frontend:

```
router.push('/login')
```

`prefetch()` allows us to programmatically prefetch a URL, useful when we don't have a `Link` tag which automatically handles prefetching for us:

```
router.prefetch('/login')
```

Full example:

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

Learn to code – free 3,000-hour curriculum

```
    router.prefetch('/login')
  )
}
```

You can also use the router to listen for route change events.

Feed data to the components using `getInitialProps`

In the previous chapter we had an issue with dynamically generating the post page, because the component required some data up front, and when we tried to get the data from the JSON file:

```
import { useRouter } from 'next/router'
import posts from '../..../posts.json'

export default () => {
  const router = useRouter()

  const post = posts[router.query.id]

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

we got this error:

Learn to code – free 3,000-hour curriculum



```
module.exports = async ({ params }) => {
  const post = await db.posts.findFirst({ where: { id: params.id } })
  if (!post) {
    return {
      status: 404,
      message: 'Post not found'
    }
  }
  return {
    props: {
      post
    }
  }
}
```

TypeError: Cannot read property 'title' of undefined

./pages/blog/[id].js:11

```
8 |
9 |   return (
10|   <>
> 11|   <h1>{post.title}</h1>
12|   ^  <p>{post.content}</p>
13|   </>
14| )
```

View compiled

▶ 13 stack frames were collapsed.

This screen is visible only in development. It will not appear if the app crashes in production.
Open your browser's developer console to further inspect this error.

How do we solve this? And how do we make SSR work for dynamic routes?

We must provide the component with props, using a special function called `getInitialProps()` which is attached to the component.

To do so, first we name the component:

```
const Post = () => {
  // ...
}

export default Post
```

then we add the function to it:

Learn to code – free 3,000-hour curriculum

```
Post.getInitialProps = () => {
  // ...
}

export default Post
```

This function gets an object as its argument, which contains several properties. In particular, the thing we are interested into now is that we get the `query` object, the one we used previously to get the post id.

So we can get it using the *object destructuring syntax*:

```
Post.getInitialProps = ({ query }) => {
  // ...
}
```

Now we can return the post from this function:

```
Post.getInitialProps = ({ query }) => {
  return {
    post: posts[query.id]
  }
}
```

And we can also remove the import of `useRouter`, and we get the post from the `props` property passed to the `Post` component:

Learn to code – free 3,000-hour curriculum

```

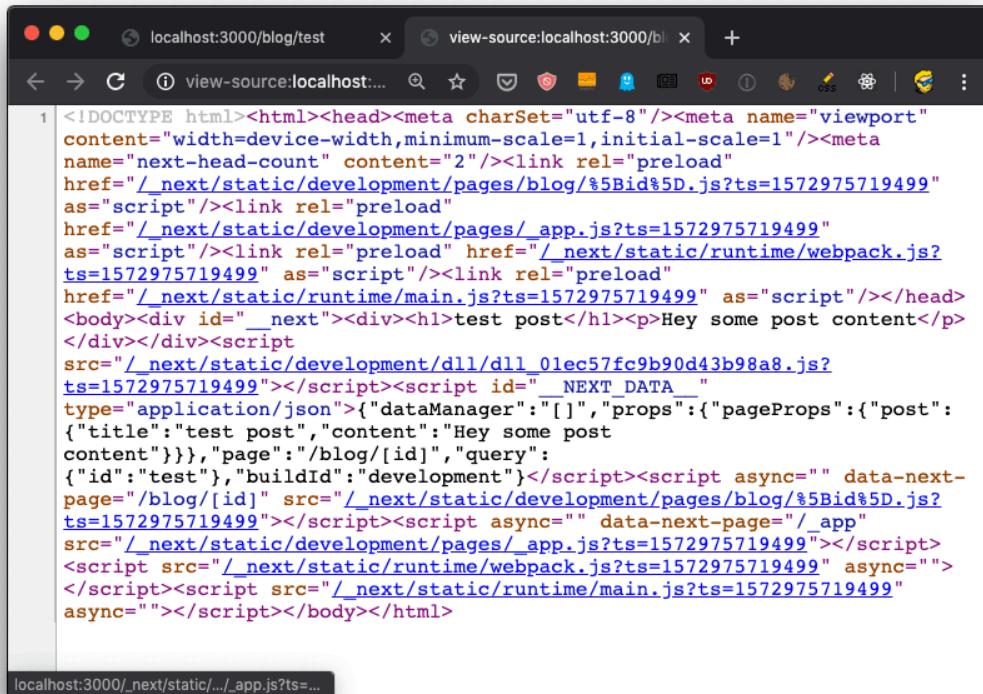
return (
  <div>
    <h1>{props.post.title}</h1>
    <p>{props.post.content}</p>
  </div>
)
}

Post.getInitialProps = ({ query }) => {
  return {
    post: posts[query.id]
  }
}

export default Post

```

Now there will be no error, and SSR will be working as expected, as you can see checking view source:



The screenshot shows a browser window with the URL `localhost:3000/blog/test` in the address bar. The title bar also says `view-source:localhost:3000/blog/test`. The main content area displays the rendered HTML of the page, which includes an `<h1>test post</h1>` heading and a `<p>Hey some post content</p>` paragraph. Below the rendered content, the browser's developer tools are open, specifically the Network tab, showing a list of resources being loaded. One resource is highlighted with a blue border: `/_next/static/runtime/main.js?ts=1572975719499`. This indicates that the page is being served via static pre-rendering.

Learn to code – free 3,000-hour curriculum

[Learn components as we code.](#)

It's important to note that `getInitialProps` gets, in the context object it receives, in addition to the `query` object these other properties:

- `pathname` : the `path` section of URL
- `asPath` - String of the actual path (including the query) shows in the browser

which in the case of calling `http://localhost:3000/blog/test` will respectively result to:

- `/blog/[id]`
- `/blog/test`

And in the case of server side rendering, it will also receive:

- `req` : the HTTP request object
- `res` : the HTTP response object
- `err` : an error object

`req` and `res` will be familiar to you if you've done any Node.js coding.

CSS

How do we style React components in Next.js?

Learn to code – free 3,000-hour curriculum

But Next.js comes with `styled-jsx` built-in, because that's a library built by the same people working on Next.js.

And it's a pretty cool library that provides us scoped CSS, which is great for maintainability because the CSS is only affecting the component it's applied to.

I think this is a great approach at writing CSS, without the need to apply additional libraries or preprocessors that add complexity.

To add CSS to a React component in Next.js we insert it inside a snippet in the JSX, which start with

```
<style jsx>{`
```

and ends with

```
`}</style>
```

Inside this weird blocks we write plain CSS, as we'd do in a `.css` file:

```
<style jsx>{
  h1 {
    font-size: 3rem;
  }
`}</style>
```

Learn to code – free 3,000-hour curriculum

```
const Index = () => (
  <div>
    <h1>Home page</h1>

    <style jsx>{
      h1 {
        font-size: 3rem;
      }
    }</style>
  </div>
)

export default Index
```

Inside the block we can use interpolation to dynamically change the values. For example here we assume a `size` prop is being passed by the parent component, and we use it in the `styled-jsx` block:

```
const Index = props => (
  <div>
    <h1>Home page</h1>

    <style jsx>{
      h1 {
        font-size: ${props.size}rem;
      }
    }</style>
  </div>
)
```

If you want to apply some CSS globally, not scoped to a component, you add the `global` keyword to the `style` tag:

Learn to code – free 3,000-hour curriculum

```
  }  
`}</style>
```

If you want to import an external CSS file in a Next.js component, you have to first install `@zeit/next-css`:

```
npm install @zeit/next-css
```

and then create a configuration file in the root of the project, called `next.config.js`, with this content:

```
const withCSS = require('@zeit/next-css')  
module.exports = withCSS()
```

After restarting the Next app, you can now import CSS like you normally do with JavaScript libraries or components:

```
import './style.css'
```

You can also import a SASS file directly, using the `@zeit/next-sass` library instead.

Populating the head tag with custom tags

Learn to code – free 3,000-hour curriculum

This is handy when:

- you want to customize the page title
- you want to change a meta tag

How can you do so?

Inside every component you can import the `Head` component from `next/head` and include it in your component JSX output:

```
import Head from 'next/head'

const House = props => (
  <div>
    <Head>
      <title>The page title</title>
    </Head>
    {/* the rest of the JSX */}
  </div>
)

export default House
```

You can add any HTML tag you'd like to appear in the `<head>` section of the page.

When mounting the component, Next.js will make sure the tags inside `Head` are added to the heading of the page. Same when unmounting the component, Next.js will take care of removing those tags.

Learn to code – free 3,000-hour curriculum

chrome window, a common base layer, and you just want to change what's inside.

There's a nav bar, a sidebar, and then the actual content.

How do you build such system in Next.js?

There are 2 ways. One is using a Higher Order Component, by creating a `components/Layout.js` component:

```
export default Page => {
  return () => (
    <div>
      <nav>
        <ul>....</ul>
      </nav>
      <main>
        <Page />
      </main>
    </div>
  )
}
```

In there we can import separate components for heading and/or sidebar, and we can also add all the CSS we need.

And you use it in every page like this:

```
import withLayout from '../components/Layout.js'

const Page = () => <p>Here's a page!</p>

export default withLayout(Page)
```

Learn to code – free 3,000-hour curriculum
to call `getInitialProps()` on a page.

Why?

Because `getInitialProps()` gets only called on the page component. But if we export the Higher Order Component `withLayout()` from a page, `Page.getInitialProps()` is not called. `withLayout.getInitialProps()` would.

To avoid unnecessarily complicating our codebase, the alternative approach is to use props:

```
export default props => (
  <div>
    <nav>
      <ul>....</ul>
    </nav>
    <main>
      {props.content}
    </main>
  </div>
)
```

and in our pages now we use it like this:

```
import Layout from '../components/Layout.js'

const Page = () => (
  <Layout content={(
    <p>Here's a page!</p>
  )} />
)
```

Learn to code – free 3,000-hour curriculum

```
import Layout from '../components/Layout.js'

const Page = () => (
  <Layout content={(
    <p>Here's a page!</p>
  )} />
)

Page.getInitialProps = ({ query }) => {
  // ...
}
```

API Routes

In addition to creating **page routes**, which means pages are served to the browser as Web pages, Next.js can create **API routes**.

This is a very interesting feature because it means that Next.js can be used to create a frontend for data that is stored and retrieved by Next.js itself, transferring JSON via fetch requests.

API routes live under the `/pages/api/` folder and are mapped to the `/api` endpoint.

This feature is very useful when creating applications.

In those routes, we write Node.js code (rather than React code). It's a paradigm shift, you move from the frontend to the backend, but very seamlessly.

Learn to code – free 3,000-hour curriculum

Say you have a list of comments stored in a `comments.json` file:

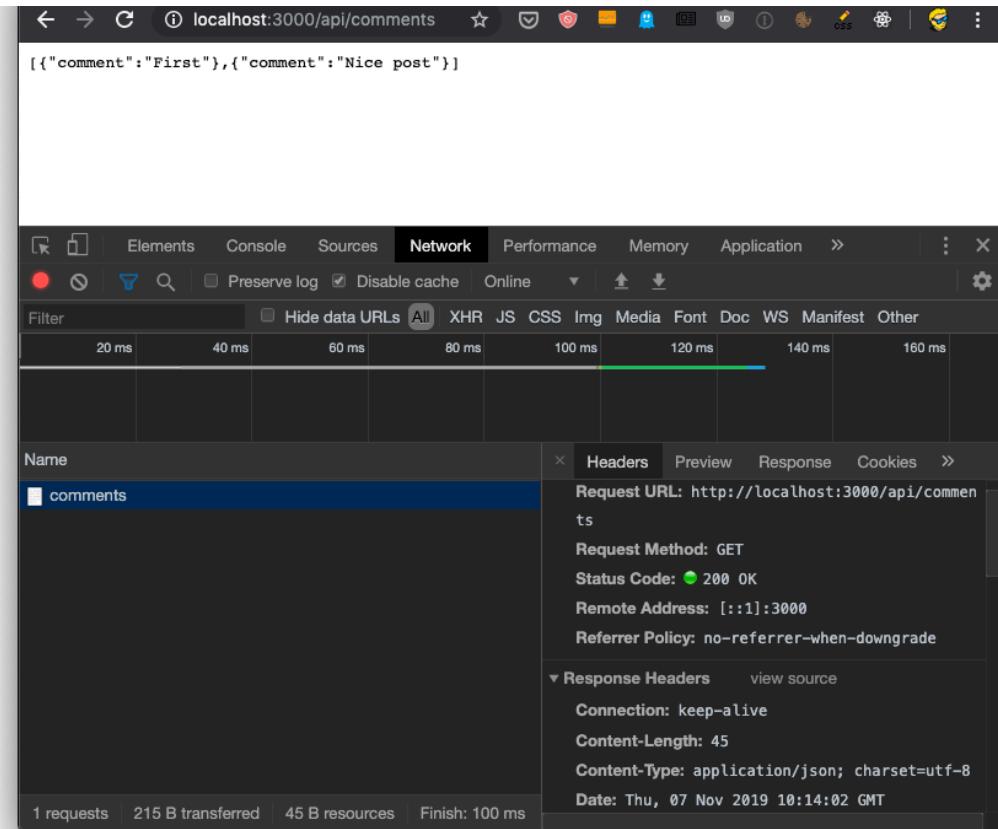
```
[  
  {  
    "comment": "First"  
  },  
  {  
    "comment": "Nice post"  
  }  
]
```

Here's a sample code, which returns to the client the list of comments:

```
import comments from './comments.json'  
  
export default (req, res) => {  
  res.status(200).json(comments)  
}
```

It will listen on the `/api/comments` URL for GET requests, and you can try calling it using your browser:

Learn to code – free 3,000-hour curriculum



API routes can also use **dynamic routing** like pages, use the `[]` syntax to create a dynamic API route, like

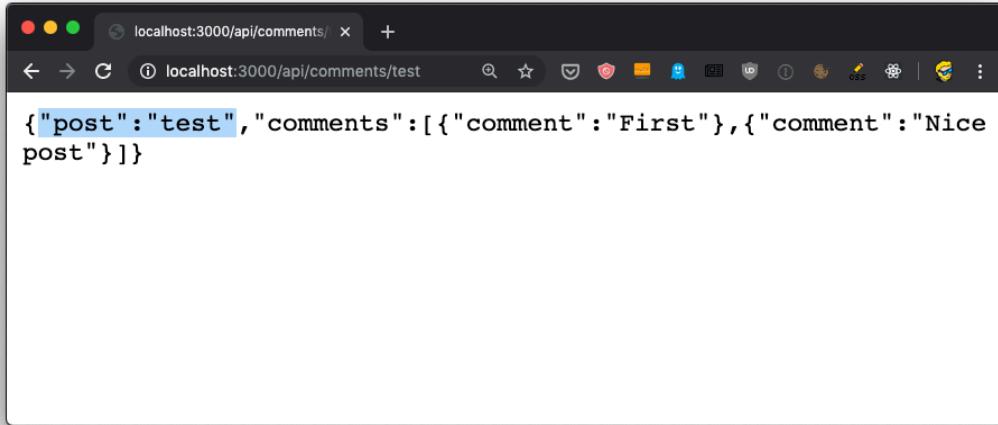
`/pages/api/comments/[id].js` which will retrieve the comments specific to a post id.

Inside the `[id].js` you can retrieve the `id` value by looking it up inside the `req.query` object:

```
import comments from '../comments.json'

export default (req, res) => {
  res.status(200).json({ post: req.query.id, comments })
}
```

Learn to code – free 3,000-hour curriculum



In dynamic pages, you'd need to import `useRouter` from `next/router`, then get the router object using `const router = useRouter()`, and then we'd be able to get the `id` value using `router.query.id`.

In the server-side it's all easier, as the query is attached to the request object.

If you do a POST request, all works in the same way - it all goes through that default export.

To separate POST from GET and other HTTP methods (PUT, DELETE), lookup the `req.method` value:

```
export default (req, res) => {
  switch (req.method) {
    case 'GET':
      // ...
      break
    case 'POST':
      // ...
      break
```

Learn to code – free 3,000-hour curriculum

}

In addition to `req.query` and `req.method` we already saw, we have access to cookies by referencing `req.cookies`, the request body in `req.body`.

Under the hoods, this is all powered by [Micro](#), a library that powers asynchronous HTTP microservices, made by the same team that built Next.js.

You can make use of any Micro middleware in our API routes to add more functionality.

Run code only on the server side or client side

In your page components, you can execute code only in the server-side or on the client-side, by checking the `window` property.

This property is only existing inside the browser, so you can check

```
if (typeof window === 'undefined') {  
}
```

and add the server-side code in that block.

Similarly, you can execute client-side code only by checking

Learn to code – free 3,000-hour curriculum

JS Tip: We use the `typeof` operator here because we can't detect a value to be undefined in other ways. We can't do `if (window === undefined)` because we'd get a "window is not defined" runtime error

Next.js, as a build-time optimization, also removes the code that uses those checks from bundles. A client-side bundle will not include the content wrapped into a `if (typeof window === 'undefined') {}` block.

Deploying the production version

Deploying an app is always left last in tutorials.

Here I want to introduce it early, just because it's so easy to deploy a Next.js app that we can dive into it now, and then move on to other more complex topics later on.

Remember in the "How to install Next.js" chapter I told you to add those 3 lines to the `package.json` `script` section:

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

Learn to code — free 3,000-hour curriculum

code reloading, two very useful features while debugging.

The same command can be invoked to build the website passing the `build` flag, by running `npm run build`. Then, the same command can be used to start the production app passing the `start` flag, by running `npm run start`.

Those 2 commands are the ones we must invoke to successfully deploy the production version of our site locally. The production version is highly optimized and does not come with source maps and other things like hot code reloading that would not be beneficial to our end users.

So, let's create a production deploy of our app. Build it using:

```
npm run build
```

Learn to code — free 3,000-hour curriculum

```
Compiled successfully.

Automatically optimizing pages

  Page      Size    Files Packages
  /          4.18 kB   0       3
  [ /_app     218 kB   0       2
  [ /_document
  [ /_error    7.69 kB   0       2
  [ ⚡ /blog     4.92 kB   1       3
  σ /blog/[id] 659 B    1       2

σ (Server)      page will be server rendered (i.e. getInitialProps)
⚡ (Static File) page was prerendered as static HTML

→ firstproject
```

The output of the command tells us that some routes (/ and /blog) are now prerendered as static HTML, while /blog/[id] will be served by the Node.js backend.

Then you can run `npm run start` to start the production server locally:

```
npm run start
```

```
firstproject — npm /Users/flaviocopes/dev/nextjs/firstproject — node + npm Apple_PubSub_Socket_Render=/private/tmp/com...
→ firstproject npm run start
[

> firstproject@1.0.0 start /Users/flaviocopes/dev/nextjs/firstproject
> next start

> Ready on http://localhost:3000
|
```

Learn to code – free 3,000-hour curriculum
[Build app, locally.](#)

Deploying on Now

In the previous chapter we deployed the Next.js application locally.

How do we deploy it to a real web server, so other people can access it?

One of the most simple ways to deploy a Next application is through the **Now** platform created by [Zeit](#), the same company that created the Open Source project Next.js. You can use Now to deploy Node.js apps, Static Websites, and much more.

Now makes the deployment and distribution step of an app very, very simple and fast, and in addition to Node.js apps, they also support deploying Go, PHP, Python and other languages.

You can think of it as the "cloud", as you don't really know where your app will be deployed, but you know that you will have a URL where you can reach it.

Now is free to start using, with generous free plan that currently includes 100GB of hosting, 1000 [serverless](#) functions invocations per day, 1000 builds per month, 100GB of bandwidth per month, and one [CDN](#) location. The [pricing.page](#) helps get an idea of the costs if you need more.

The best way to start using Now is by using the official Now CLI:

Learn to code — free 3,000-hour curriculum

Once the command is available, run

```
now login
```

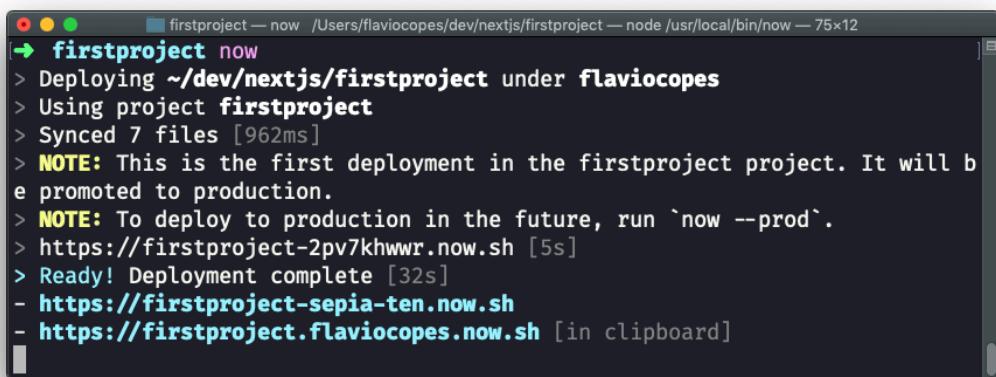
and the app will ask you for your email.

If you haven't registered already, create an account on <https://zeit.co/signup> before continuing, then add your email to the CLI client.

Once this is done, from the Next.js project root folder run

```
now
```

and the app will be instantly deployed to the Now cloud, and you'll be given the unique app URL:



```
firstproject — now /Users/flaviocopes/dev/nextjs/firstproject — node /usr/local/bin/now — 75x12
[→ firstproject now
> Deploying ~/dev/nextjs/firstproject under flaviocopes
> Using project firstproject
> Synced 7 files [962ms]
> NOTE: This is the first deployment in the firstproject project. It will be promoted to production.
> NOTE: To deploy to production in the future, run `now --prod`.
> https://firstproject-2pv7khwwr.now.sh [5s]
> Ready! Deployment complete [32s]
- https://firstproject-sepia-ten.now.sh
- https://firstproject.flaviocopes.now.sh [in clipboard]
```

Learn to code – free 3,000-hour curriculum

We can see 3 different URLs in the output given in the image:

- <https://firstproject-2pv7khwwr.now.sh>
- <https://firstproject-sepia-ten.now.sh>
- <https://firstproject.flaviocopes.now.sh>

Why so many?

The first is the URL identifying the deploy. Every time we deploy the app, this URL will change.

You can test immediately by changing something in the project code, and running `now` again:

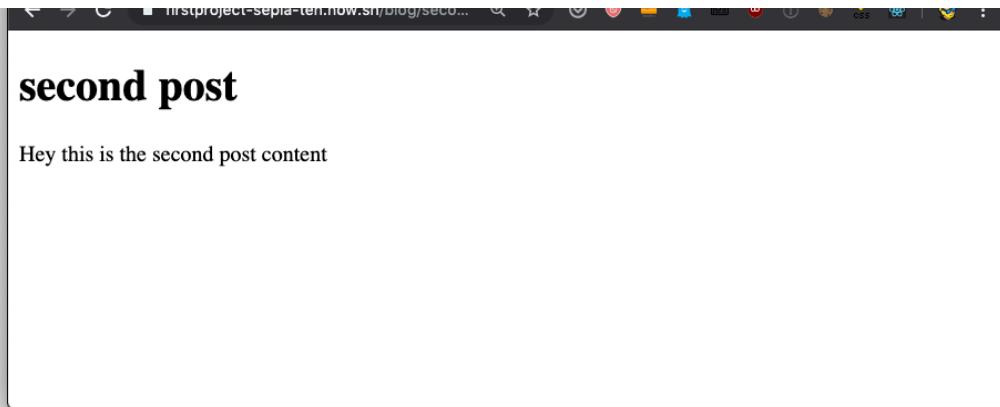


```
firstproject — fish /Users/flaviocopes/dev/nextjs/firstproject — -fish — 75x8
[green arrow] ➔ firstproject now
> Deploying ~/dev/nextjs/firstproject under flaviocopes
> Using project firstproject
> Synced 1 file [982ms]
> https://firstproject-rdcwfkwt6.now.sh [4s]
> Ready! Deployed to https://firstproject.flaviocopes.now.sh [in clipboard]
[38s]
[green arrow] ➔ firstproject
```

The other 2 URLs will not change. The first is a random one, the second is your project name (which defaults to the current project folder), your account name and then `.now.sh`.

If you visit the URL, you will see the app deployed to production.

Learn to code – free 3,000-hour curriculum



You can configure Now to serve the site to your own custom domain or subdomain, but I will not dive into that right now.

The `now.sh` subdomain is enough for our testing purposes.

Analyzing the app bundles

Next provides us a way to analyze the code bundles that are generated.

Open the `package.json` file of the app and in the `scripts` section add those 3 new commands:

```
"analyze": "cross-env ANALYZE=true next build",
"analyze:server": "cross-env BUNDLE_ANALYZE=server next build",
"analyze:browser": "cross-env BUNDLE_ANALYZE=client next build"
```

Like this:

Learn to code – free 3,000-hour curriculum

```
"description": "",  
"main": "index.js",  
"scripts": {  
  "dev": "next",  
  "build": "next build",  
  "start": "next start",  
  "analyze": "cross-env ANALYZE=true next build",  
  "analyze:server": "cross-env BUNDLE_ANALYZE=server next build",  
  "analyze:browser": "cross-env BUNDLE_ANALYZE=client next build"  
},  
"keywords": [],  
"author": "",  
"license": "ISC",  
"dependencies": {  
  "next": "^9.1.2",  
  "react": "^16.11.0",  
  "react-dom": "^16.11.0"  
}  
}
```



then install those 2 packages:

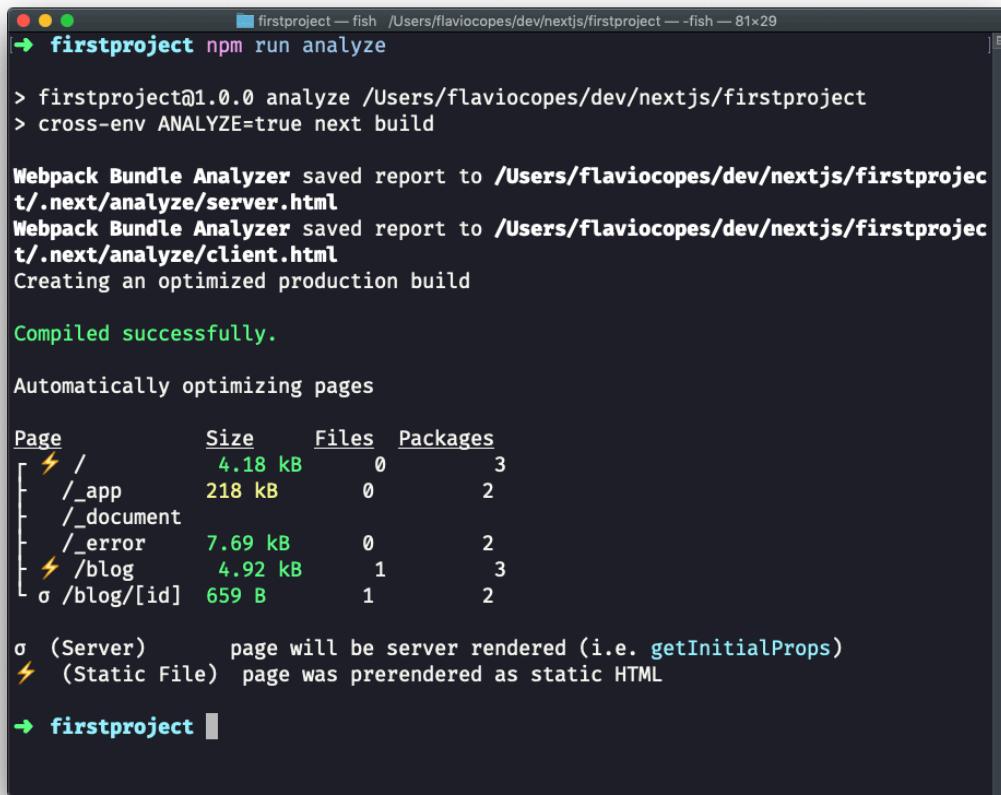
```
npm install --dev cross-env @next/bundle-analyzer
```

Create a `next.config.js` file in the project root, with this content:

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

Now run the command

Learn to code — free 3,000-hour curriculum



```
firstproject — fish /Users/flaviocopes/dev/nextjs/firstproject — -fish — 81x29
→ firstproject npm run analyze

> firstproject@1.0.0 analyze /Users/flaviocopes/dev/nextjs/firstproject
> cross-env ANALYZE=true next build

Webpack Bundle Analyzer saved report to /Users/flaviocopes/dev/nextjs/firstprojec
t/.next/analyze/server.html
Webpack Bundle Analyzer saved report to /Users/flaviocopes/dev/nextjs/firstprojec
t/.next/analyze/client.html
Creating an optimized production build

Compiled successfully.

Automatically optimizing pages

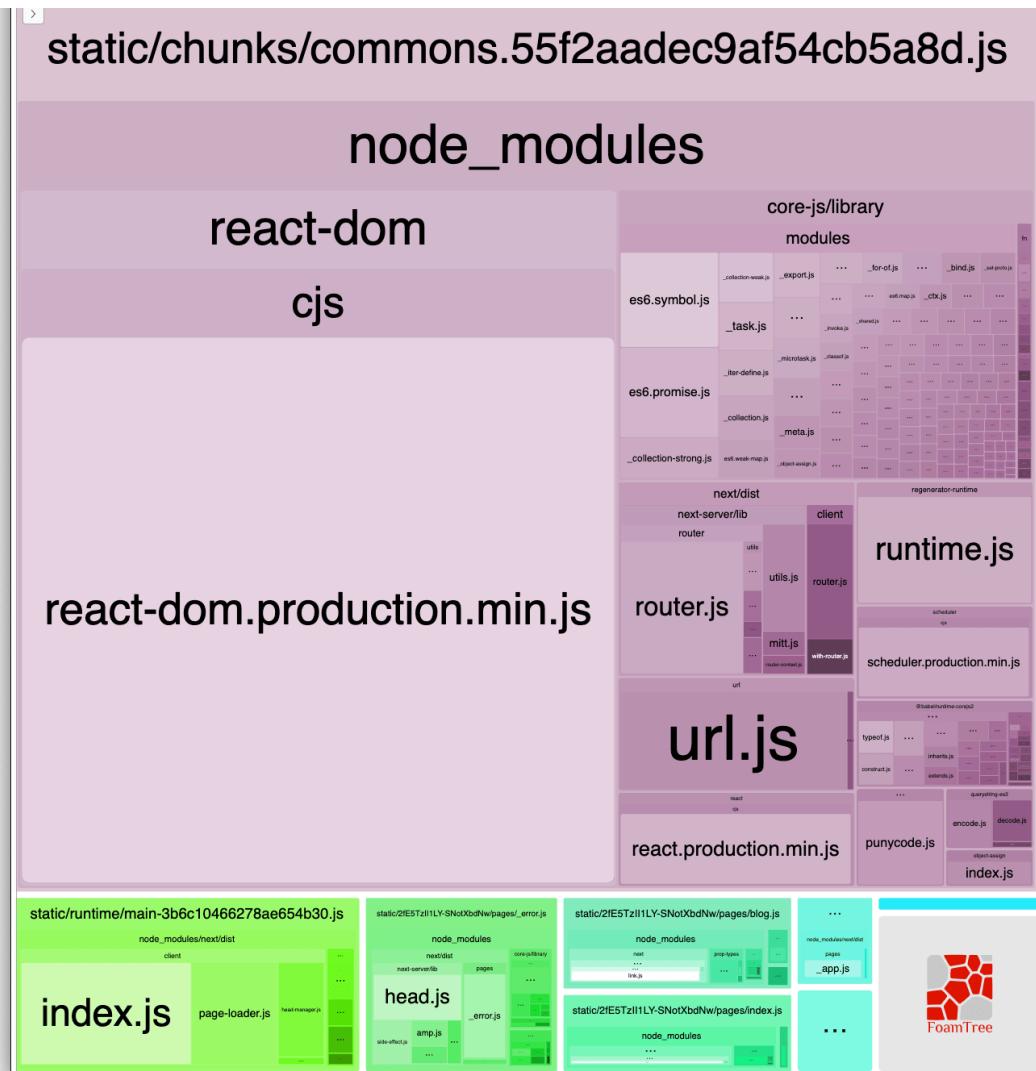
Page      Size    Files  Packages
⚡ /       4.18 kB   0      3
[  /_app   218 kB    0      2
  /_document
  /_error   7.69 kB   0      2
  ⚡ /blog   4.92 kB   1      3
  ⚡ /blog/[id] 659 B   1      2

σ (Server)    page will be server rendered (i.e. getInitialProps)
⚡ (Static File) page was prerendered as static HTML

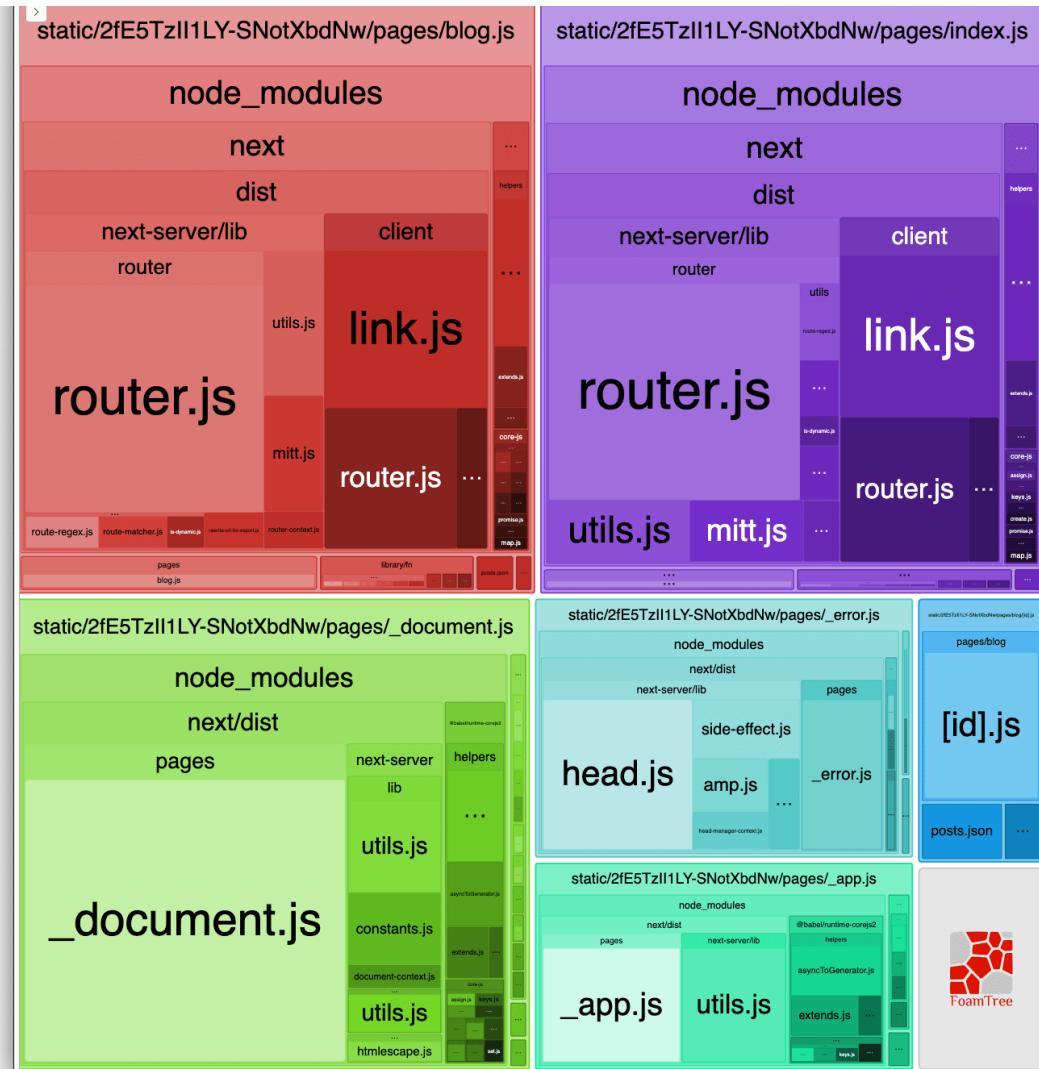
→ firstproject
```

This should open 2 pages in the browser. One for the client bundles, and one for the server bundles:

Learn to code – free 3,000-hour curriculum



Learn to code – free 3,000-hour curriculum



This is incredibly useful. You can inspect what's taking the most space in the bundles, and you can also use the sidebar to exclude bundles, for an easier visualization of the smaller ones:

Learn to code – free 3,000-hour curriculum



Lazy loading modules

Being able to visually analyze a bundle is great because we can optimize our application very easily.

Say we need to load the Moment library in our blog posts. Run:

```
npm install moment
```

Learn to code – free 3,000-hour curriculum

/blog and /blog/[id].

We import it in pages/blog/[id].js :

```
import moment from 'moment'

...

const Post = props => {
  return (
    <div>
      <h1>{props.post.title}</h1>
      <p>Published on {moment().format('ddd D MMMM YYYY')}</p>
      <p>{props.post.content}</p>
    </div>
  )
}
```

I'm just adding today's date, as an example.

This will include Moment.js in the blog post page bundle, as you can see by running `npm run analyze` :

Learn to code — free 3,000-hour curriculum

```
> firstproject@1.0.0 analyze /Users/flaviocopes/dev/nextjs/firstproject
> cross-env ANALYZE=true next build

Webpack Bundle Analyzer saved report to /Users/flaviocopes/dev/nextjs/firstproject/.next/analyze/server.html
Webpack Bundle Analyzer saved report to /Users/flaviocopes/dev/nextjs/firstproject/.next/analyze/client.html
Creating an optimized production build

Compiled successfully.

Automatically optimizing pages

Page          Size    Files  Packages
⚡ /           4.18 kB   0      3
└─/_app       218 kB    0      2
└─/_document
└─/_error     7.69 kB   0      2
└─σ /blog     5.66 kB   1      3
└─σ /blog/[id] 350 kB   1      3

σ (Server)    page will be server rendered (i.e. getInitialProps)
⚡ (Static File) page was prerendered as static HTML

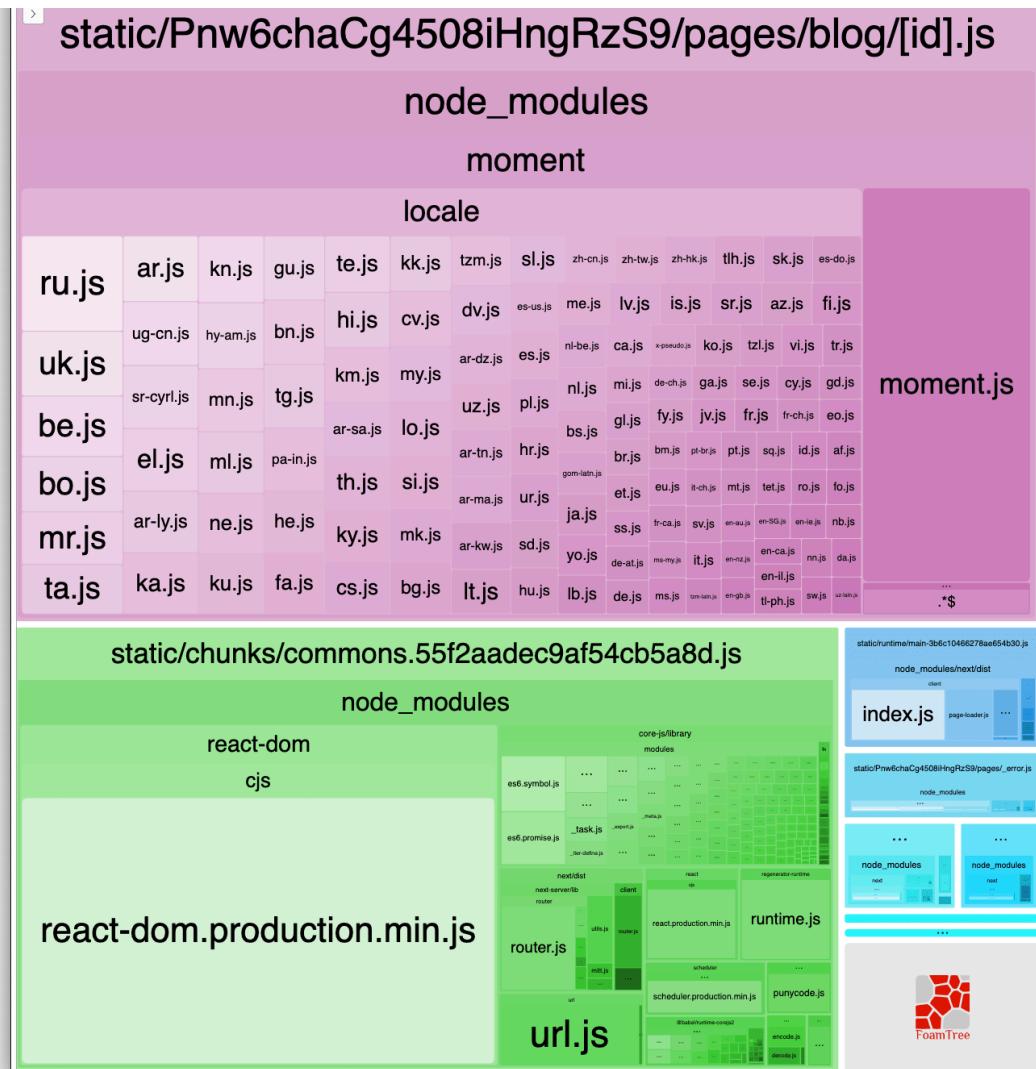
→ firstproject
```

See that we now have a red entry in `/blog/[id]`, the route that we added Moment.js to!

It went from ~1kB to 350kB, quite a big deal. And this is because the Moment.js library itself is 349kB.

The client bundles visualization now shows us that the bigger bundle is the page one, which before was very little. And 99% of its code is Moment.js.

Learn to code – free 3,000-hour curriculum



Every time we load a blog post we are going to have all this code transferred to the client. Which is not ideal.

One fix would be to look for a library with a smaller size, as Moment.js is not known for being lightweight (especially out of the box with all the locales included), but let's assume for the sake of the example that we must use it.

What we can do instead is separating all the Moment code in a **separate bundle**.

Learn to code – free 3,000-hour curriculum

Remember that we can't return complex objects inside the `getInitialProps()` returned object, so we calculate the date inside it:

```
import posts from '../posts.json'

const Post = props => {
  return (
    <div>
      <h1>{props.post.title}</h1>
      <p>Published on {props.date}</p>
      <p>{props.post.content}</p>
    </div>
  )
}

Post.getInitialProps = async ({ query }) => {
  const moment = (await import('moment')).default
  return {
    date: moment.format('ddd D MMMM YYYY'),
    post: posts[query.id]
  }
}

export default Post
```

See that special call to `.default()` after `await import`? It's needed to reference the default export in a dynamic import (see <https://v8.dev/features/dynamic-import>)

Now if we run `npm run analyze` again, we can see this:

Learn to code — free 3,000-hour curriculum

```
> firstproject@1.0.0 analyze /Users/flaviocopes/dev/nextjs/firstproject
> cross-env ANALYZE=true next build

Webpack Bundle Analyzer saved report to /Users/flaviocopes/dev/nextjs/firstproject/.next/analyze/server.html
Webpack Bundle Analyzer saved report to /Users/flaviocopes/dev/nextjs/firstproject/.next/analyze/client.html
Creating an optimized production build

Compiled successfully.

Automatically optimizing pages

Page          Size    Files  Packages
⚡ /           4.18 kB   0      3
└─/_app       219 kB    0      2
└─/_document
└─/_error     7.69 kB   0      2
└─σ /blog     5.66 kB   1      3
└─σ /blog/[id] 1.46 kB   1      3

σ (Server)    page will be server rendered (i.e. getInitialProps)
⚡ (Static File) page was prerendered as static HTML

→ firstproject
```

Our `/blog/[id]` bundle is again very small, as Moment has been moved to its own bundle file, loaded separately by the browser.

Where to go from here

There is a lot more to know about Next.js. I didn't talk about managing user sessions with login, serverless, managing databases, and so on.

The goal of this Handbook is not to teach you everything, but instead it aims to introduce you, gradually, to all the power of Next.js.

The next step I recommend is to take a good read at the [Next.js official documentation](#) to find out more about all the features and functionality I didn't talk about, and take a look at all the additional

Learn to code – free 3,000-hour curriculum

You can reach me on Twitter [@flaviocopes](#).

Also check out my website, [flaviocopes.com](#).

Note: you can download a PDF / ePub / Mobi version of this tutorial so you can read it offline!



Flavio Copes

Read [more posts](#).

If you read this far, thank the author to show them you care.

[Say Thanks](#)

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

[Get started](#)

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) charity organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can [make a tax-deductible donation here](#).

Learn to code – free 3,000-hour curriculum

Learn CSS Transform	Build a Static Blog	Build an AI Chatbot
What is Programming?	Python Code Examples	Open Source for Devs
HTTP Networking in JS	Write React Unit Tests	Learn Algorithms in JS
How to Write Clean Code	Learn PHP	Learn Java
Learn Swift	Learn Golang	Learn Node.js
Learn CSS Grid	Learn Solidity	Learn Express.js
Learn JS Modules	Learn Apache Kafka	REST API Best Practices
Front-End JS Development	Learn to Build REST APIs	Intermediate TS and React
Command Line for Beginners	Intro to Operating Systems	Learn to Build GraphQL APIs
OSS Security Best Practices	Distributed Systems Patterns	Software Architecture Patterns

Mobile App



Our Charity

[About](#) [Alumni Network](#) [Open Source](#) [Shop](#) [Support](#) [Sponsors](#) [Academic Honesty](#)
[Code of Conduct](#) [Privacy Policy](#) [Terms of Service](#) [Copyright Policy](#)