



NEXT.JS HANDBOOK

FLAVIO COPES

Table of Contents

[Preface](#)

[The Next.js Handbook](#)

[Conclusion](#)

Preface

The Next.js Handbook follows the 80/20 rule: learn in 20% of the time the 80% of a topic.

In particular, the goal is to get you up to speed quickly with Next.js.

This book is written by Flavio. I **publish programming tutorials** on my blog flaviocopes.com and [The Valley Of Code](#).

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

Enjoy!

The Next.js Handbook

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 rendering for you.

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 containing all the app code, the app is broken up automatically by Next.js in several different resources.

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.

TypeScript Support

Next.js is written in TypeScript and as such comes with an excellent 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.

Next.js *can* generate a static site too, but I would not say it's its main use case.

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](#)

Now that you have Node, updated to the latest version, and `npm`, we're set!

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

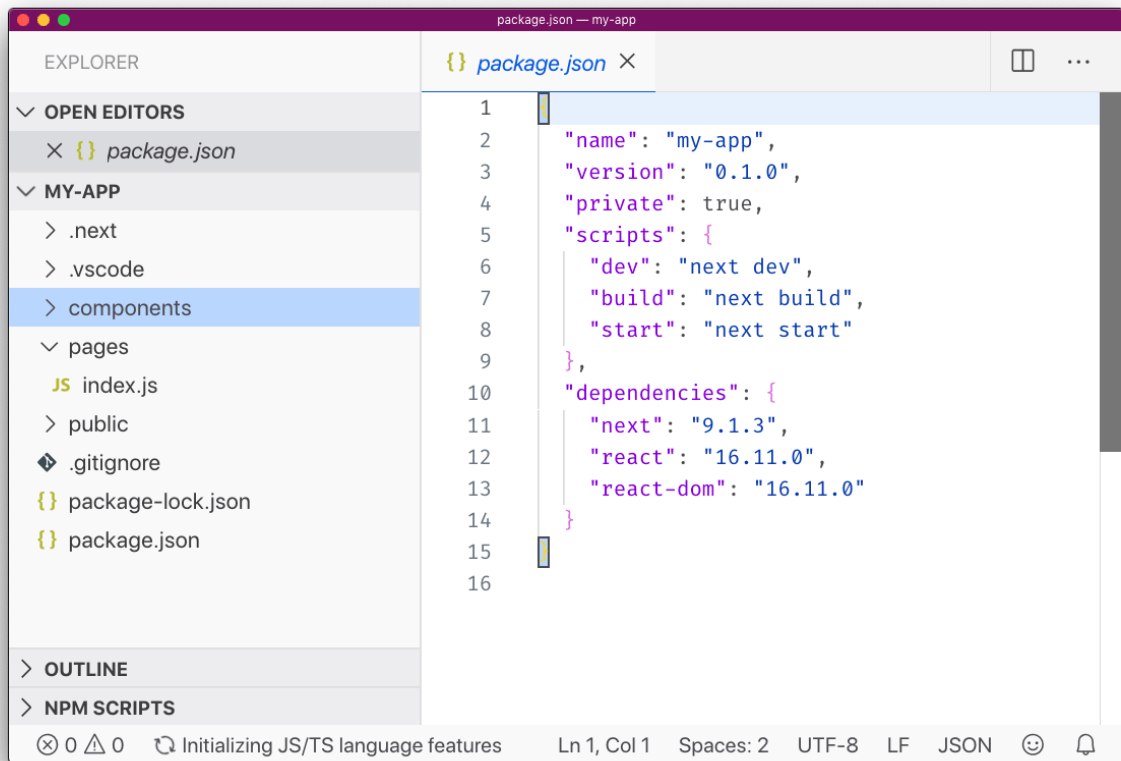
Using create-next-app

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:



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

```
my-app — npm /Users/flaviocopes/dev/nextjs/testing/my-app — node · npm Apple_PubSub_Socket_Re...

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:

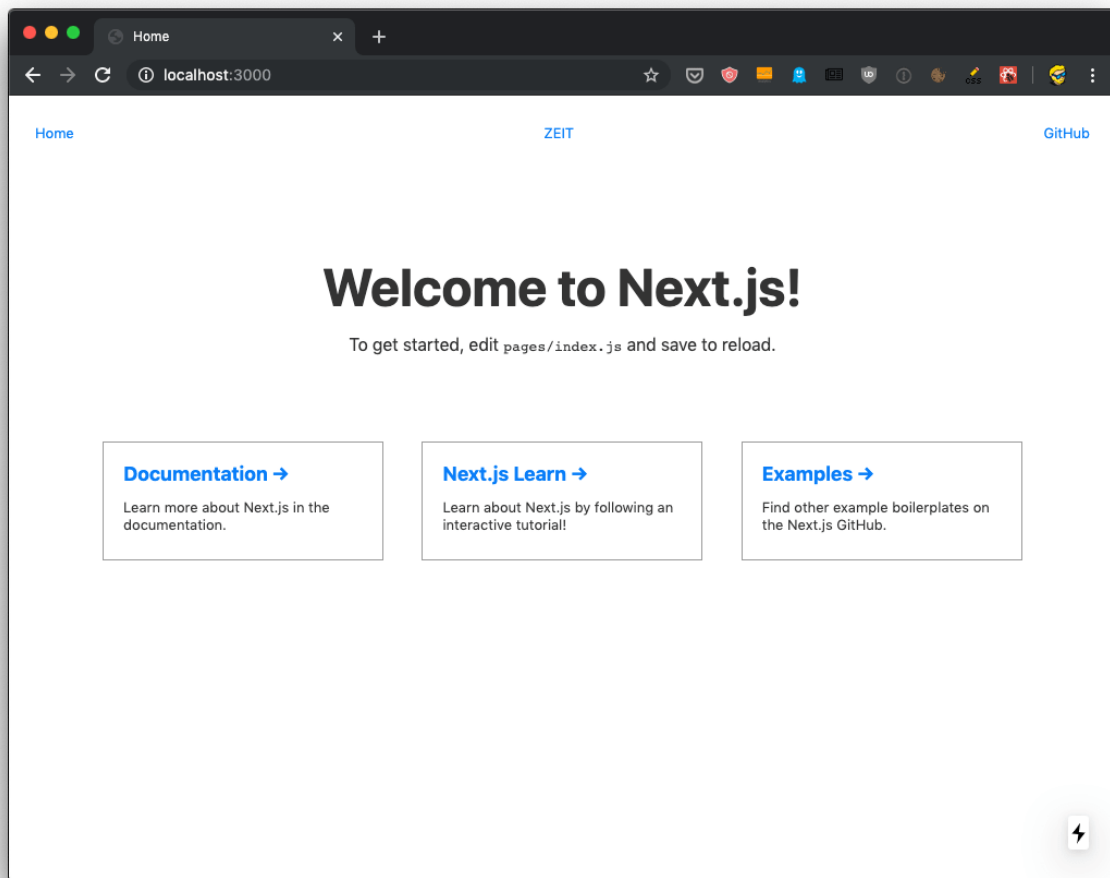
  cd my-app
  npm run dev

[→ 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
0
[ wait ] compiling ...
[ ready ] compiled successfully - ready on http://localhost:3000
0
```

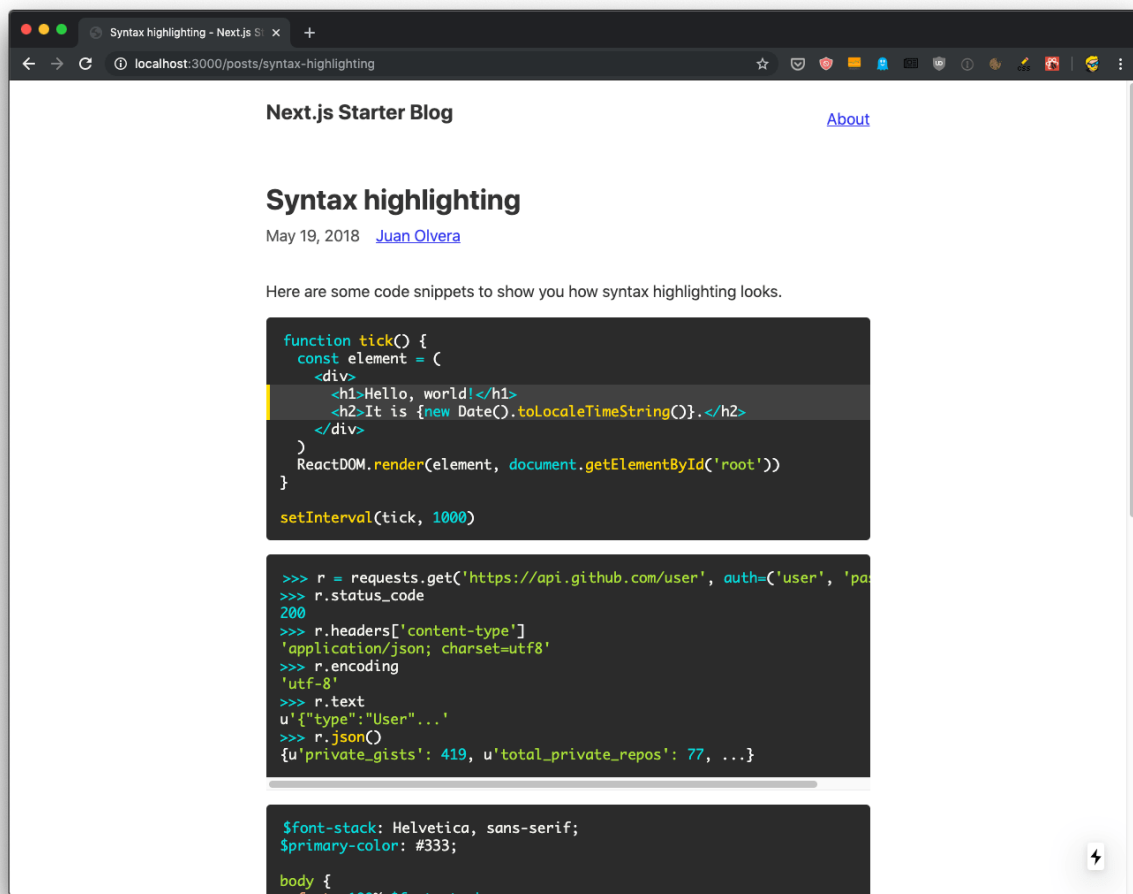
And here's the result on <http://localhost:3000>:



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:



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
```

Now use the `npm` command to initialize it as a Node project:

```
npm init -y
```

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

A terminal window titled 'firstproject' showing the command 'npm init -y' being executed. The output shows the command writing to a 'package.json' file with the following content: { "name": "firstproject", "version": "1.0.0", "description": "", "main": "index.js", "scripts": { "test": "echo \"Error: no test specified\" && exit 1" }, "keywords": [], "author": "", "license": "ISC" }. The prompt then returns to 'firstproject' with a cursor.

```
firstproject — fish /Users/flaviocopes/dev/nextjs/firstproject — -fish — 71x21
→ firstproject npm init -y
Wrote to /Users/flaviocopes/dev/nextjs/firstproject/package.json:

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

→ firstproject
```

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

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:

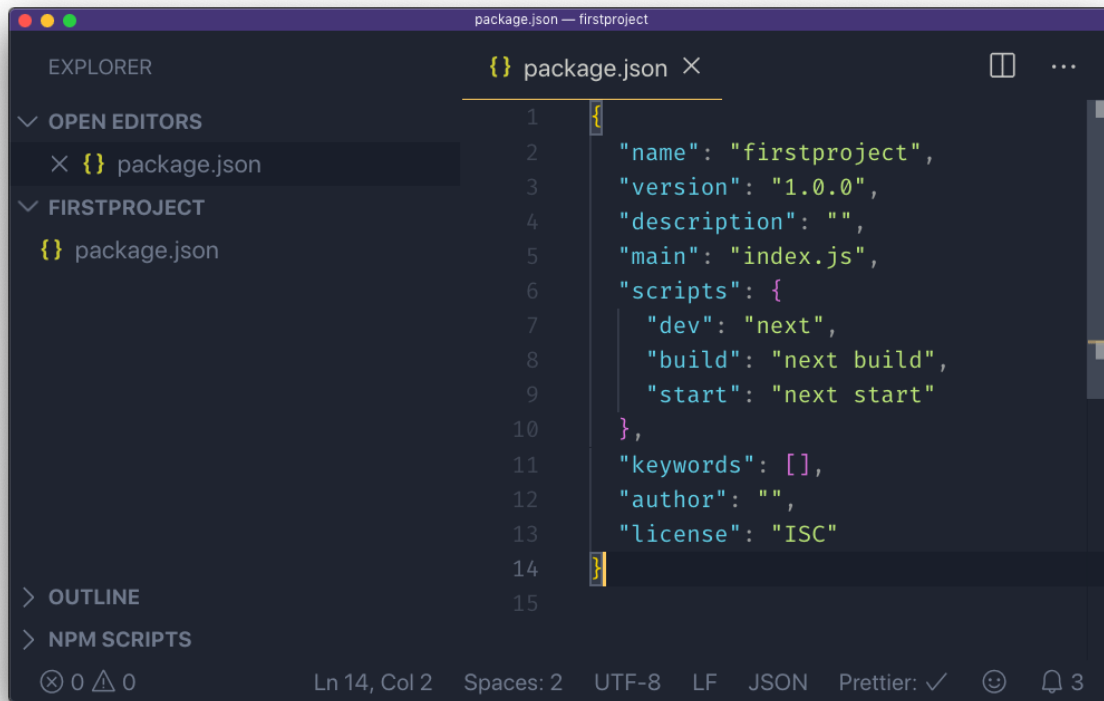
```
{
  "name": "firstproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "next": "^9.1.2",
    "react": "^16.11.0",
    "react-dom":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

and replace the `scripts` section with:

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

to add the Next.js build commands, which we're going to use soon.

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



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
```

Now using the terminal, run `npm run dev` to start the Next development server.

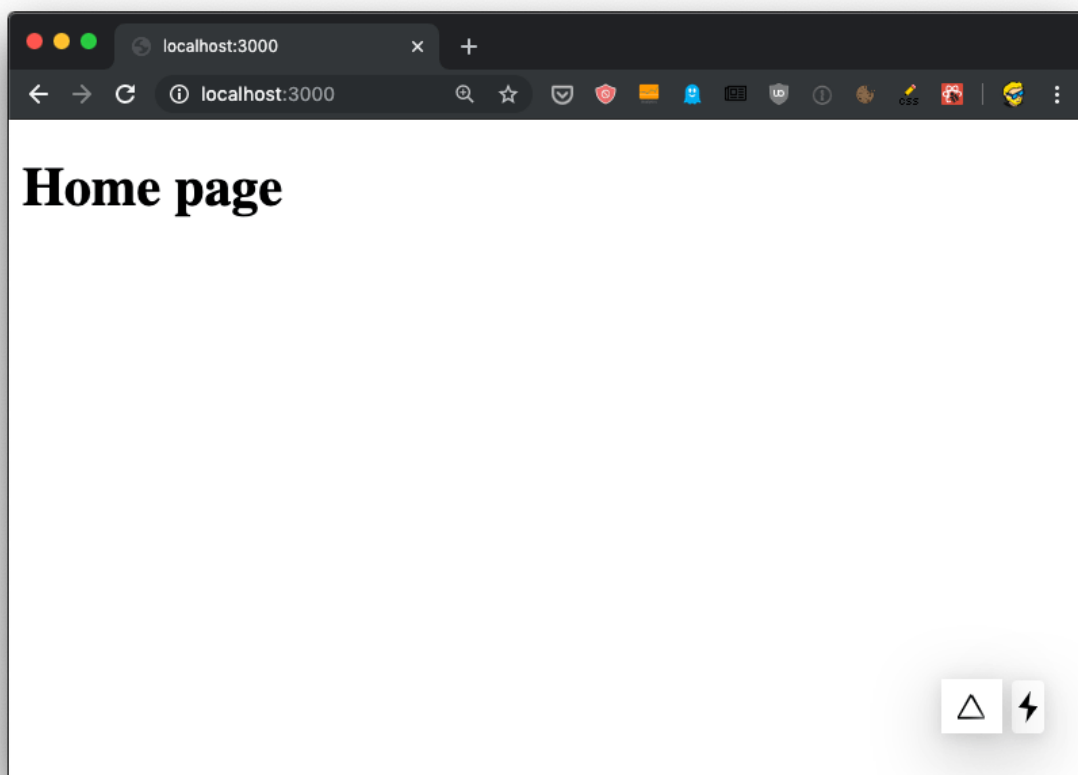
This will make the app available on port 3000, on localhost.

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

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

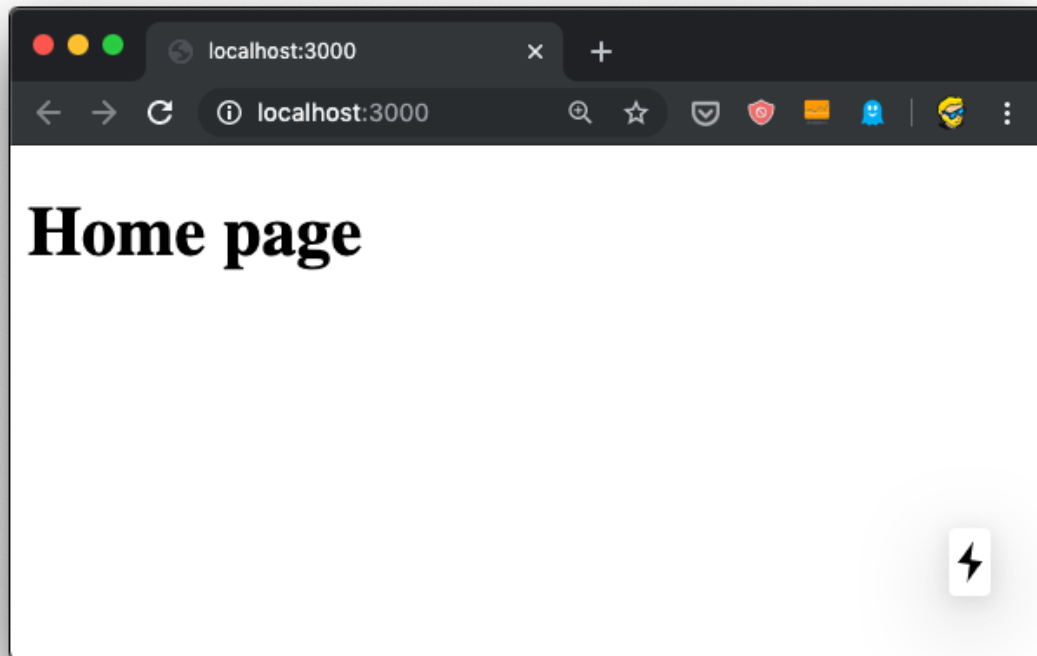
[ wait ] starting the development server ...
[ info ] waiting on http://localhost:3000 ...
[ ready ] compiled successfully - ready on http://localhost:3000
[ wait ] compiling ...
[ ready ] compiled successfully - ready on http://localhost:3000
```

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

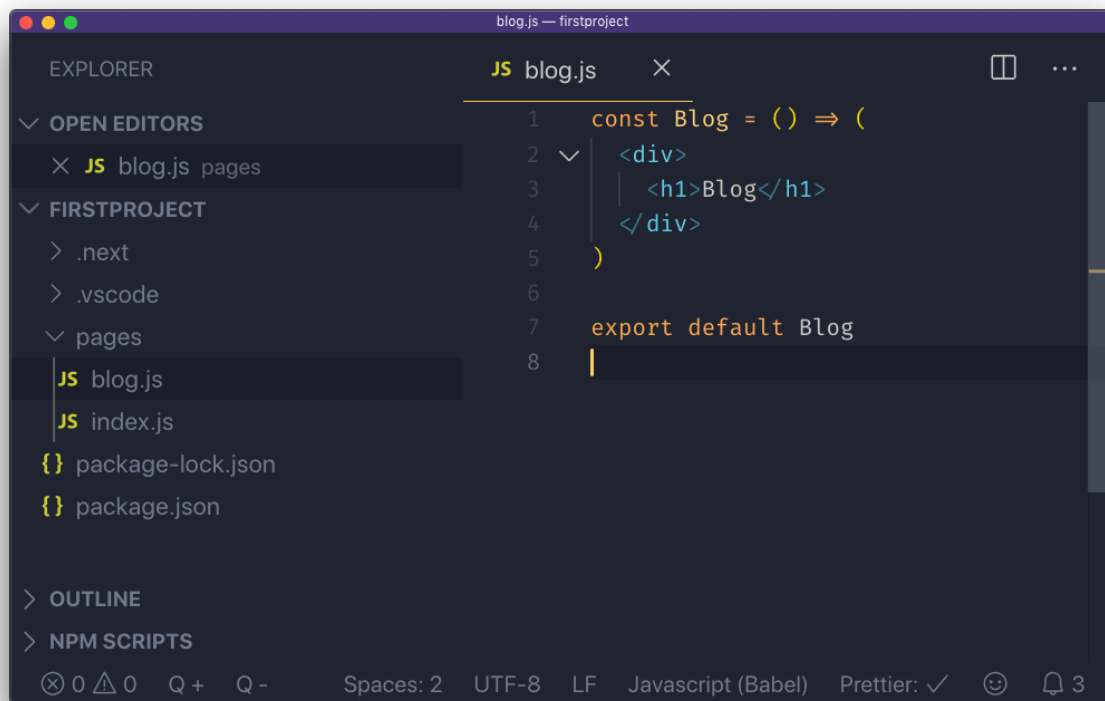


Adding a second page to the site

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:

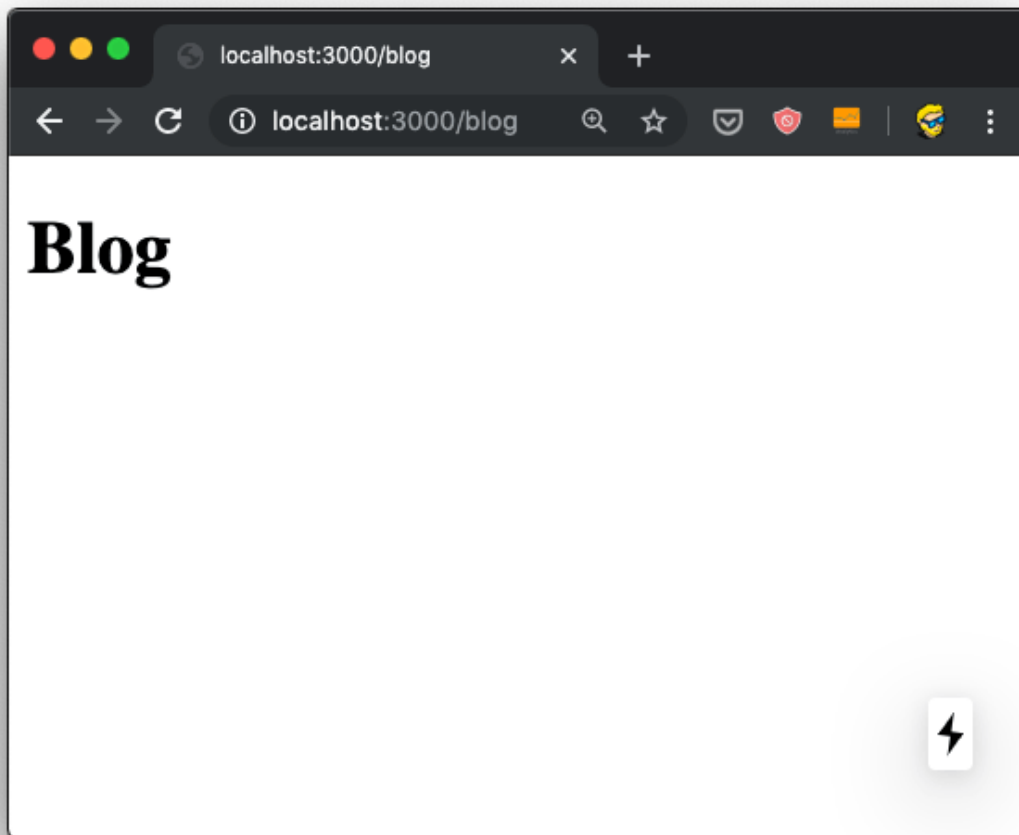


I want to add a second page to this website, a blog. It's going to be served into `/blog`, and for the time being it will just contain a simple static page, just like our first `index.js` component:



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:



and here's what the terminal told us:

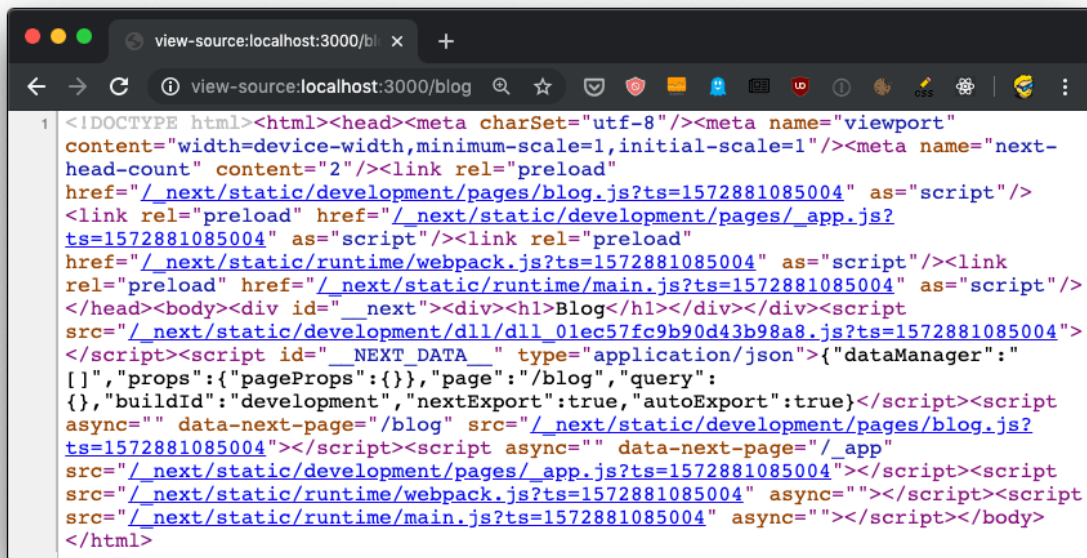
```
firstproject — npm /Users/flaviocopes/dev/nextjs/firstproject — node • npm Apple_PubSub_Socket_Render=/private/t...
[ event ] build page: /blog
[ wait ] compiling ...
[ ready ] compiled successfully - ready on http://localhost:3000
```

Now the fact that the URL is `/blog` depends on just the filename, and its position under the `pages` folder.

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.



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

```
export default () => (
  <div>
    <h1>Blog</h1>
  </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>
```

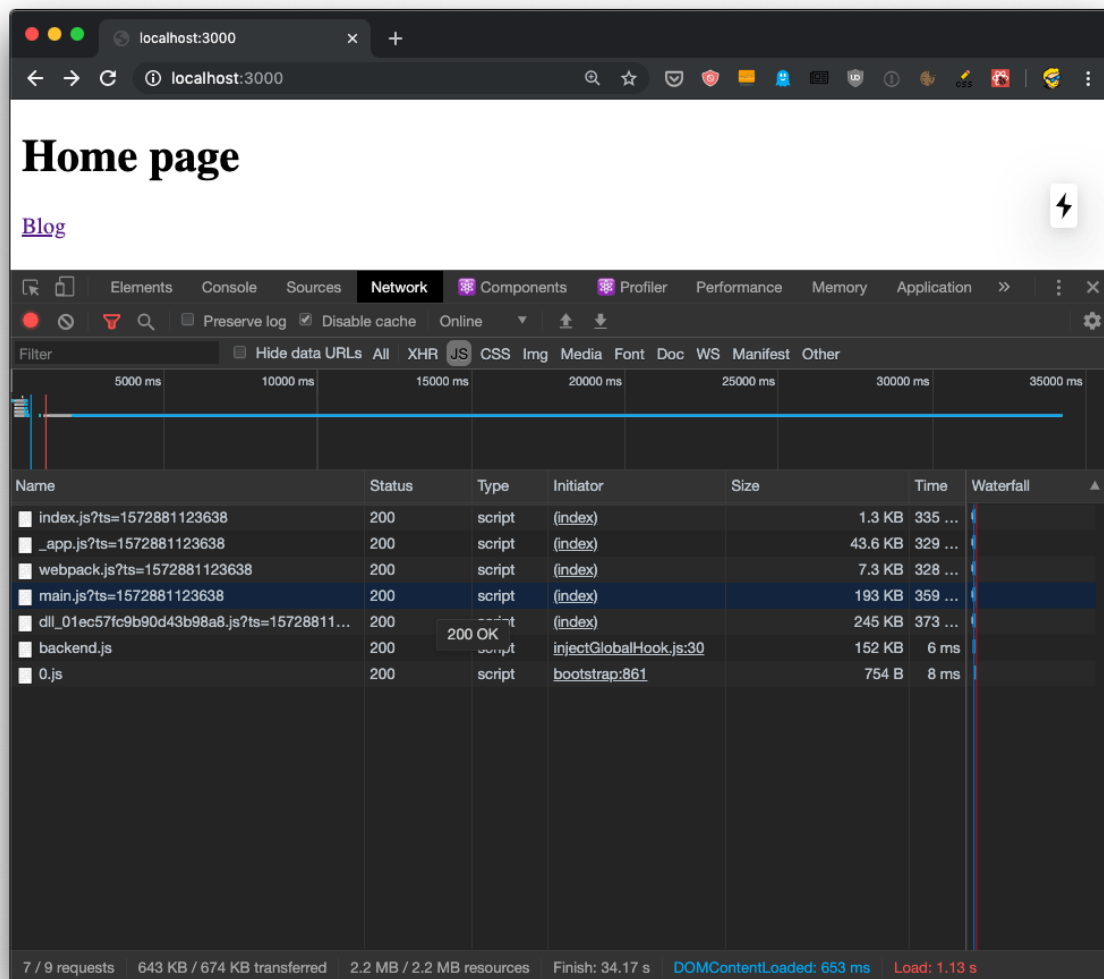
We can't do that in Next.js.

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.

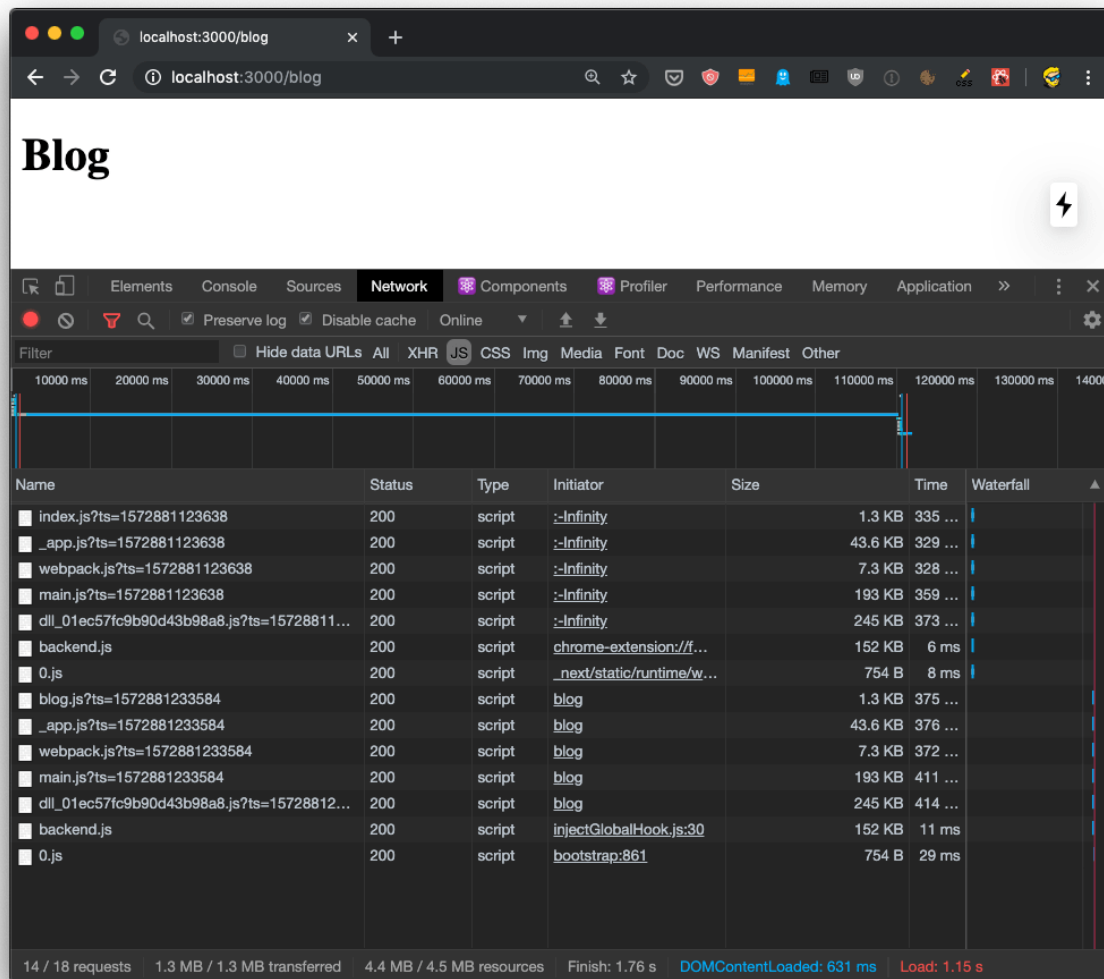
If you use a plain `a` link:

```
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:



Now if you click the "Preserve log" button (to avoid clearing the Network panel), and click the "Blog" link, this is what happens:



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'
```

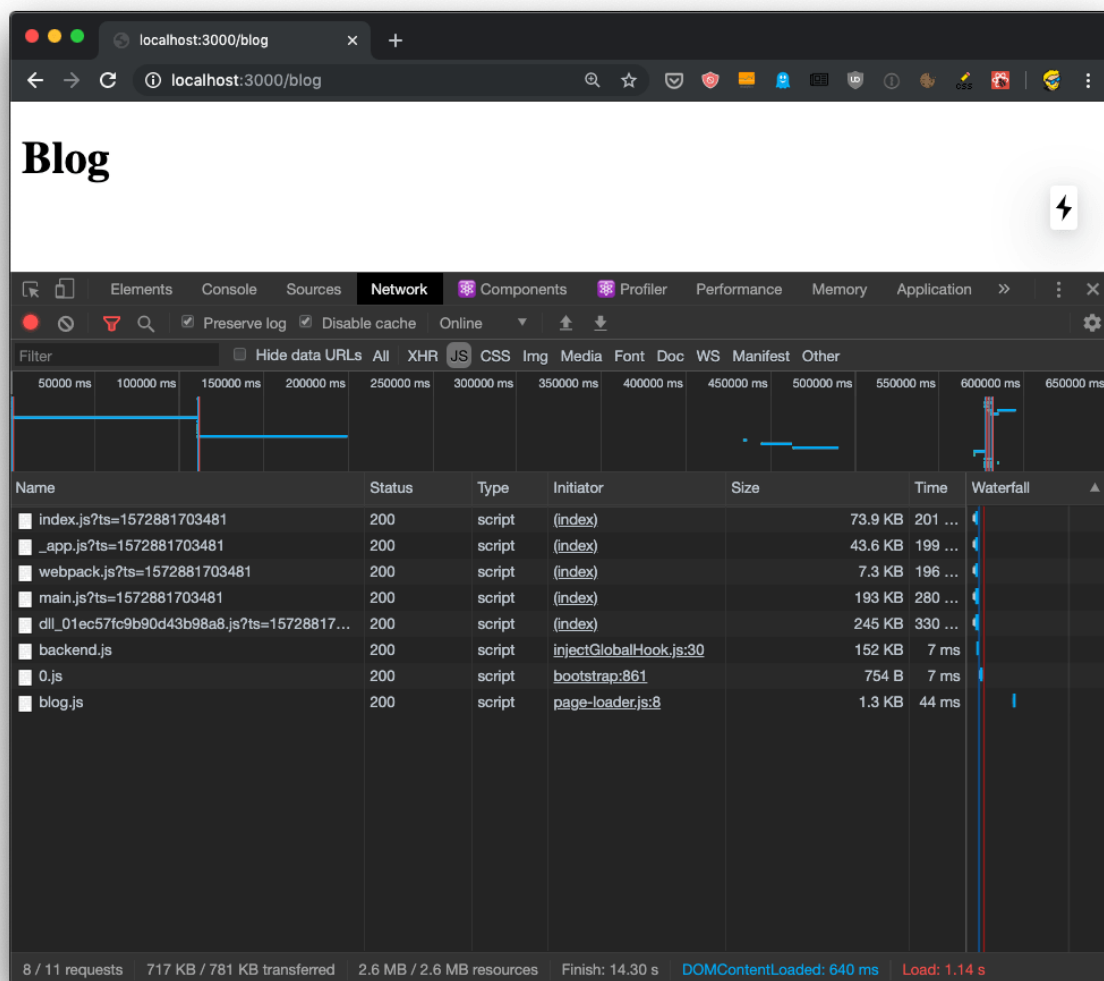
and then we use it to wrap our link, like this:

```
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:



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.

A blog is a great use case for Next.js, one we'll continue to explore in this chapter by adding **blog posts**.

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'
```

and once we have `useRouter` , we instantiate the router object using:

```
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.

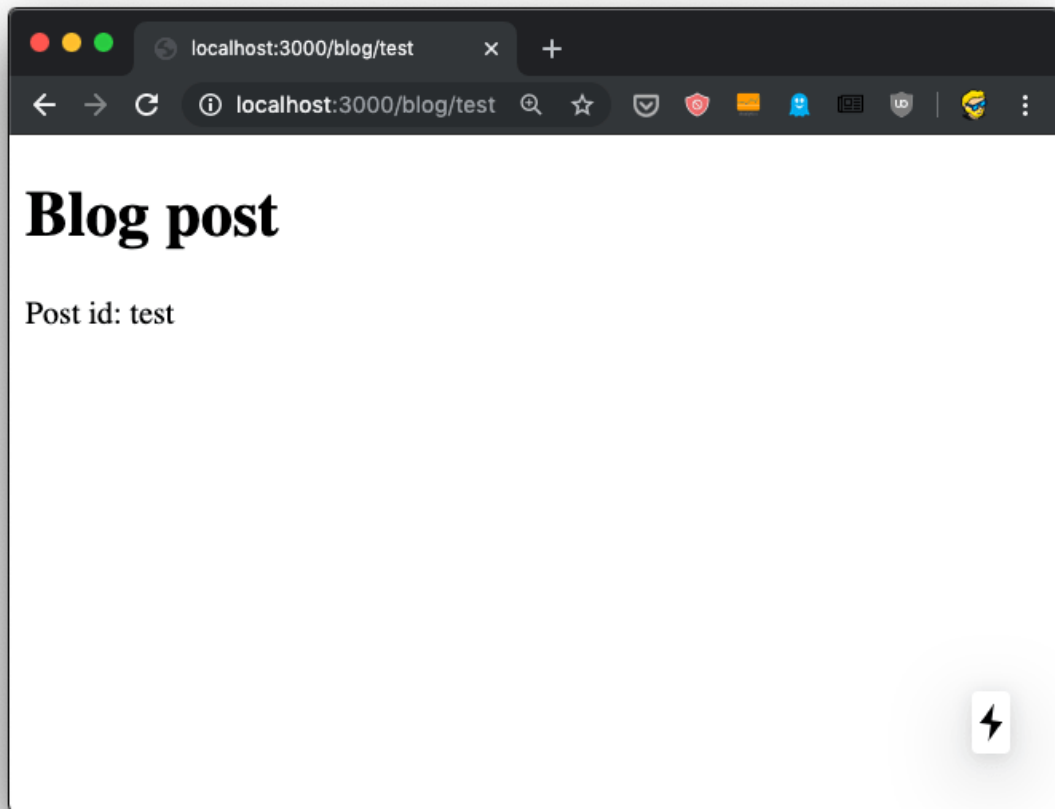
Create the file `pages/blog/[id].js` :

```
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:



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:

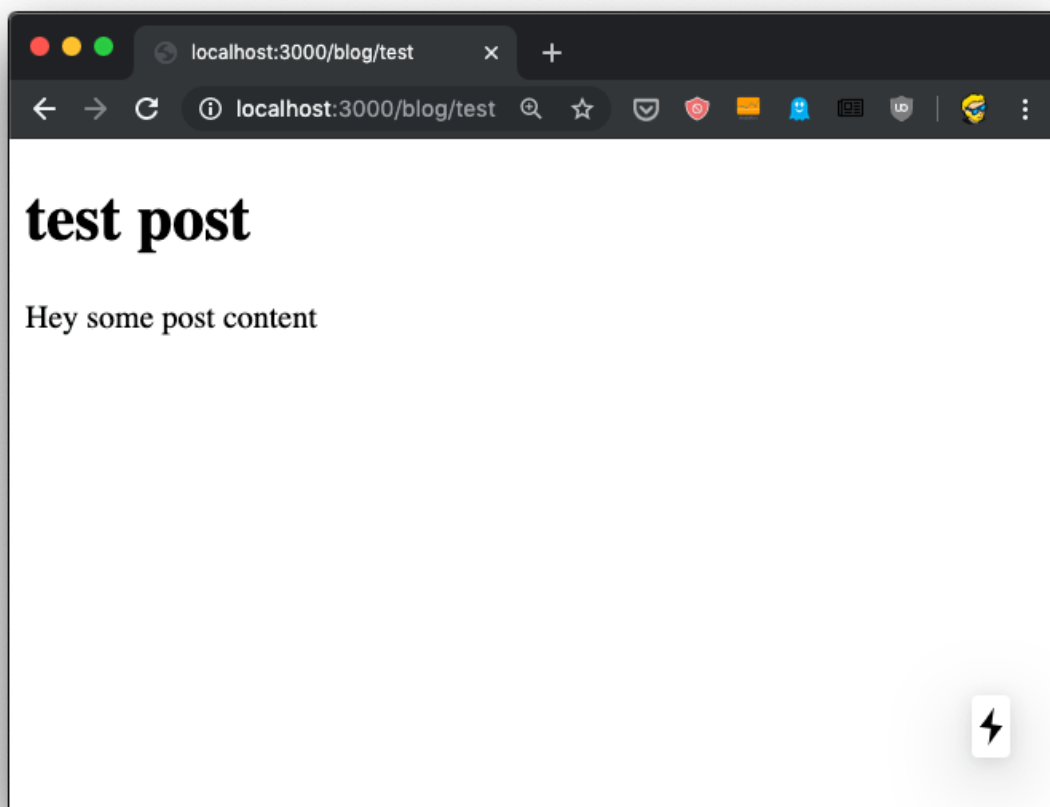
```
import { useRouter } from 'next/router'
import posts from '../posts.json'

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

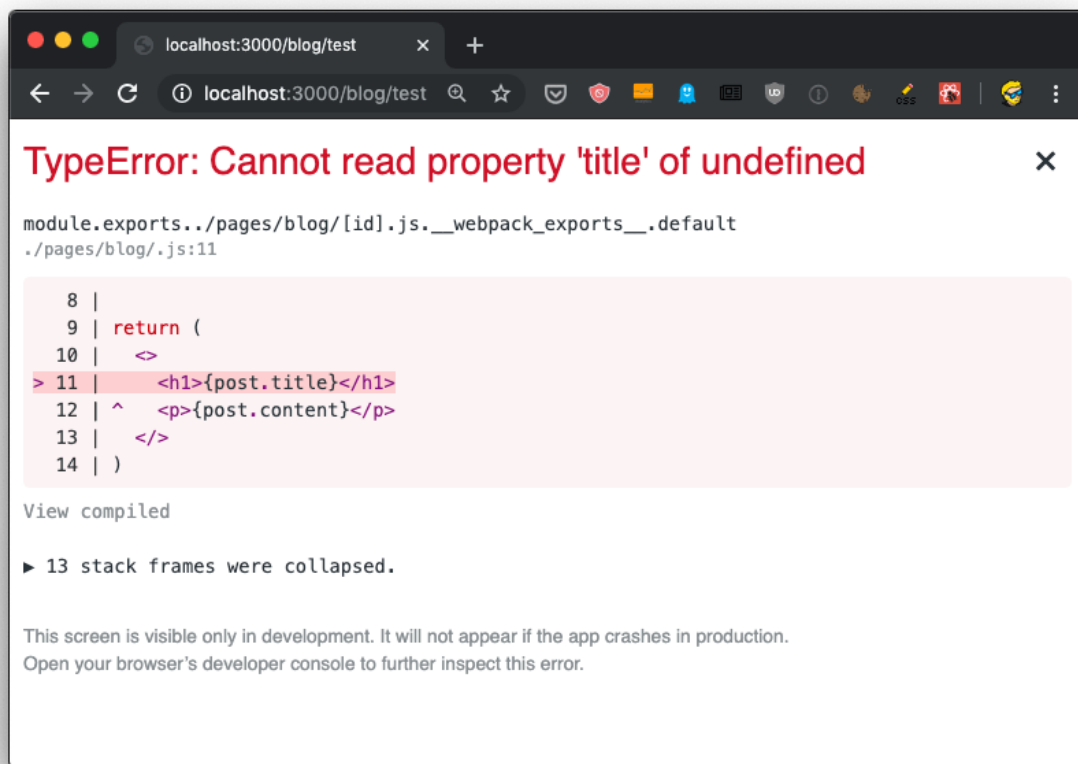
  const post = posts[router.query.id]

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

Reloading the page should show us this result:



But it's not! Instead, we get an error in the console, and an error in the browser, too:



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()

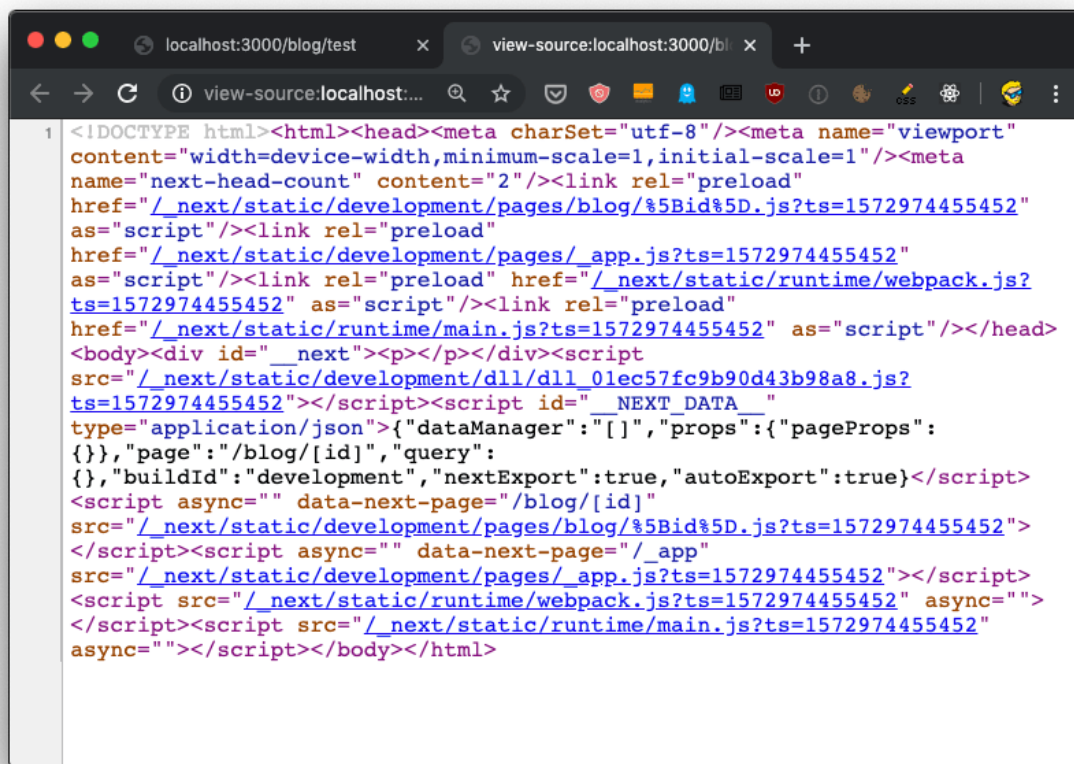
  const post = posts[router.query.id]
  if (!post) return <p></p>

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

```

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:



```
1 <!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" async="">
</script><script src="/_next/static/runtime/main.js?ts=1572974455452"
async=""></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.

We can complete the blog example by listing those posts in `pages/blog.js` :

```
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>
)

export default Blog
```

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

Prefetching is automatic on high speed connections (Wifi and 3g+ connections, unless the browser sends the `Save-Data` HTTP 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 `selected` class to the children.

We finally return this children with the updated class, using `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, { className })}</
}
```

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'

export default () => {
  const router = useRouter()
  //...
}
```

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'

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

  useEffect(() => {
    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:

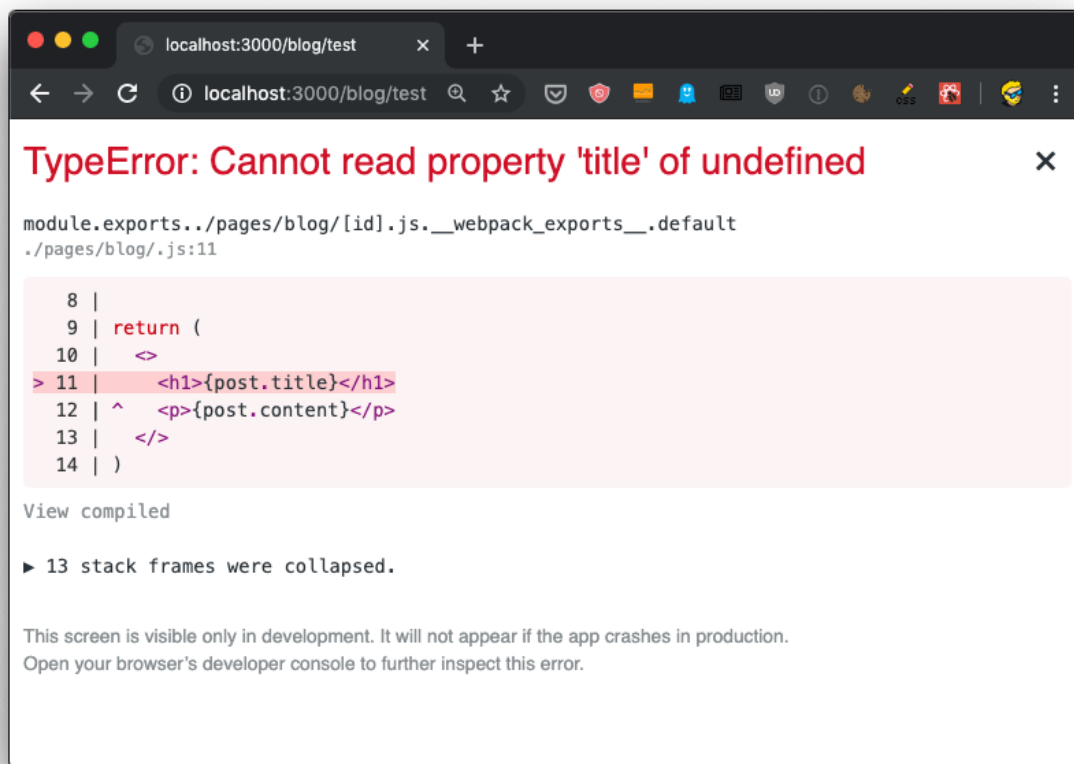
```
import { useRouter } from 'next/router'
import posts from '../../posts.json'

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

  const post = posts[router.query.id]

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

we got this error:



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:

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

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:

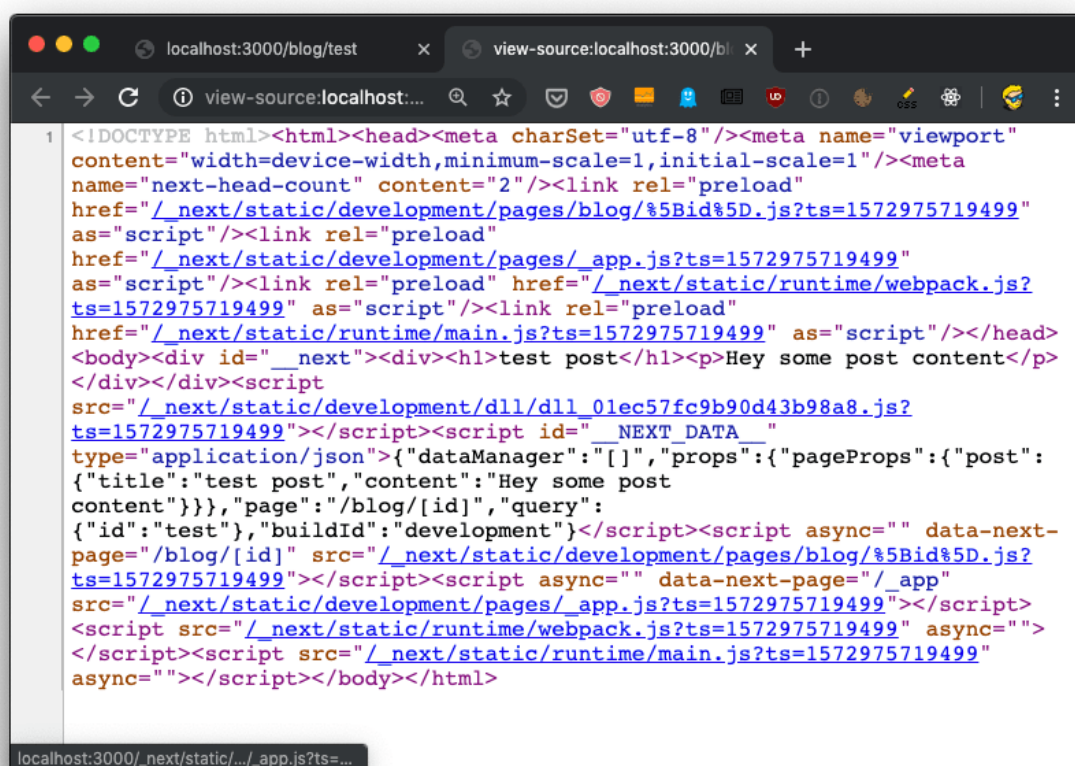
```
import posts from '../..posts.json'

const Post = (props) => {
  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 `getInitialProps` function will be executed on the server side, but also on the client side, when we navigate to a new page using the `Link` component as we did.

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?

We have a lot of freedom, because we can use whatever library we prefer.

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>
```

You write it inside the JSX, like this:

```
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:

```
<style jsx global>{`  
  body {  
    margin: 0;  
  }  
`}</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

From any Next.js page component, you can add information to the page header.

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.

Adding a wrapper component

All the pages on your site look more or less the same. There's a 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)
```

But I found this works only for simple cases, where you don't need 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>} />
```

This approach lets us use `getInitialProps()` from within our page component, with the only downside of having to write the component JSX inside the `content` prop:

```
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.

Say you have a `/pages/api/comments.js` file, whose goal is to return the comments of a blog post as JSON.

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

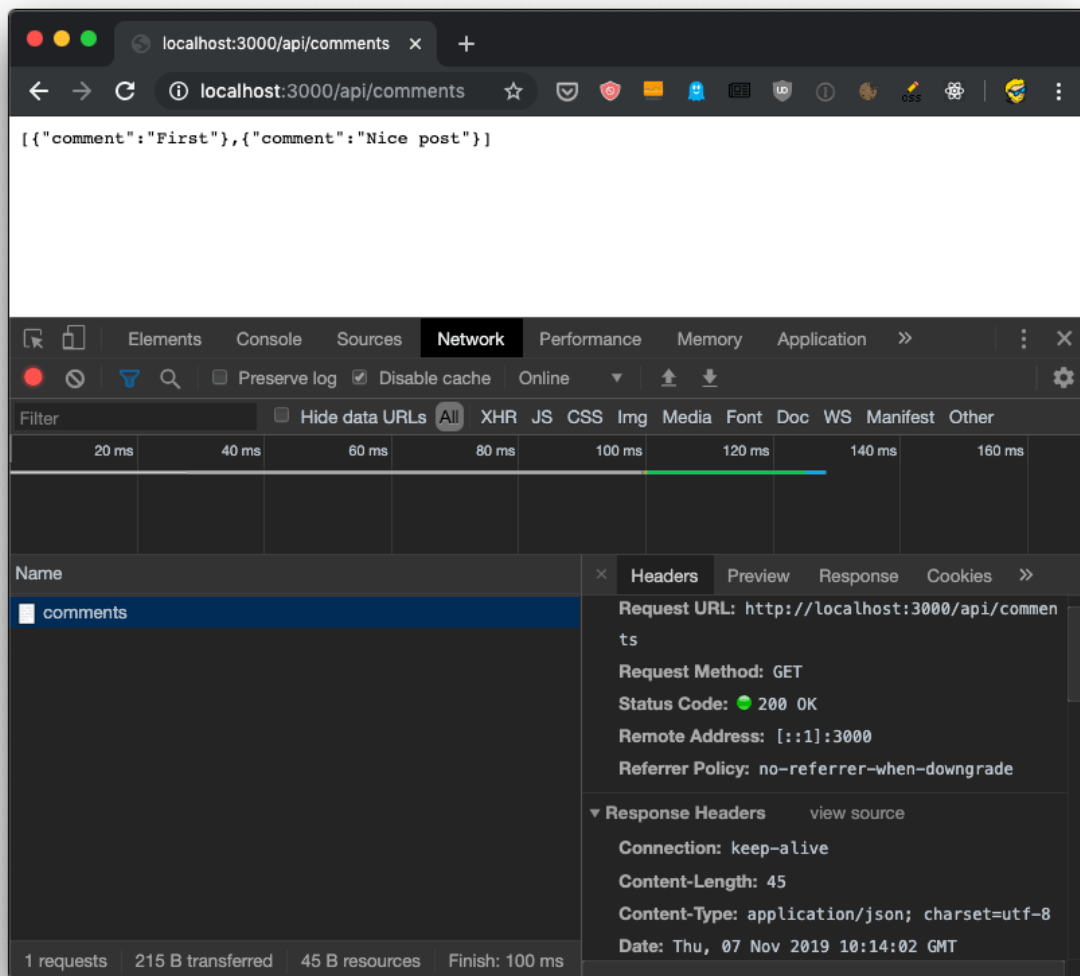
```
[
  {
    "comment": "First"
  },
  {
    "comment": "Nice post"
  }
]
```

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:



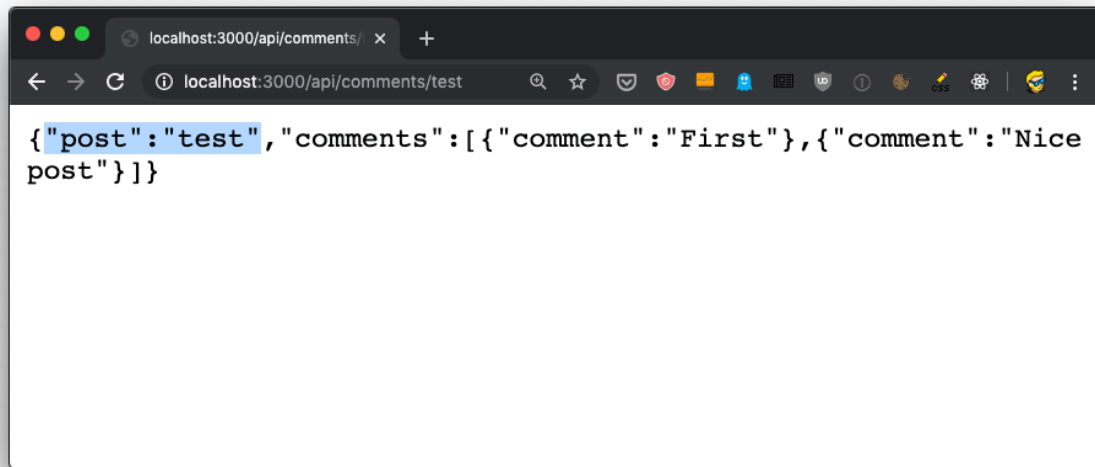
API routes can also use **dynamic routing** like pages, use the `[]` syntax to create a dynamic API route, like `/pages/api/comments/[id].js` which will retrieve the comments specific to a post id.

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 })
}
```

Heres you can see the above code in action:



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
    default: //Method Not Allowed
      res.status(405).end()
      break
  }
}
```

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 hood, 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 on the server side, or on the 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

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

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"  
}
```

We used `npm run dev` up to now, to call the `next` command installed locally in `node_modules/next/dist/bin/next`. This started the development server, which provided us **source maps** and **hot 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
```

```
firstproject — fish /Users/flaviocopes/dev/nextjs/firstproject — fish — 75x21
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
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
```

Visiting <http://localhost:3000> will show us the production version of the app, locally.

Conclusion

Thanks a lot for reading this book.

For more, head over to [The Valley Of Code](#).

Send any feedback, errata or opinions at hello@thevalleyofcode.com