NEXT.js (PAGES ROUTER) HANDBOOK

FLAVIO COPES

Preface

This book aims to be an introduction to Next.js, in particular using its Pages Router.

While Next.js has recently introduced the App Router, which supports React Server components, the Pages Router is still used in countless applications built in the past, and still maintained.

You might work on a project that uses it, or uses both the App Router and the Pages Router at the same time.

If you're unfamiliar with JavaScript, TypeScript or React, I highly recommend reading my handbooks on those topics.

After reading this book I'd recommend checking out the other books on Web Development freely available on my website.

This book was published in early 2025.

Legal

Introduction

Through this book we will learn Next.js, which is in my opinion the best tool to create web applications with React.

And it’s the perfect way to create a Node.js API in a Web Application without maintaining multiple different codebases, like one for the frontend and one for the backend.

We call it a *full-stack* framework.

Why do we need Next.js on top of React?

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* like React does by default.

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 are **server rendering** and **static pre-rendering**, both of which are provided by Next.js.

How to install Next.js

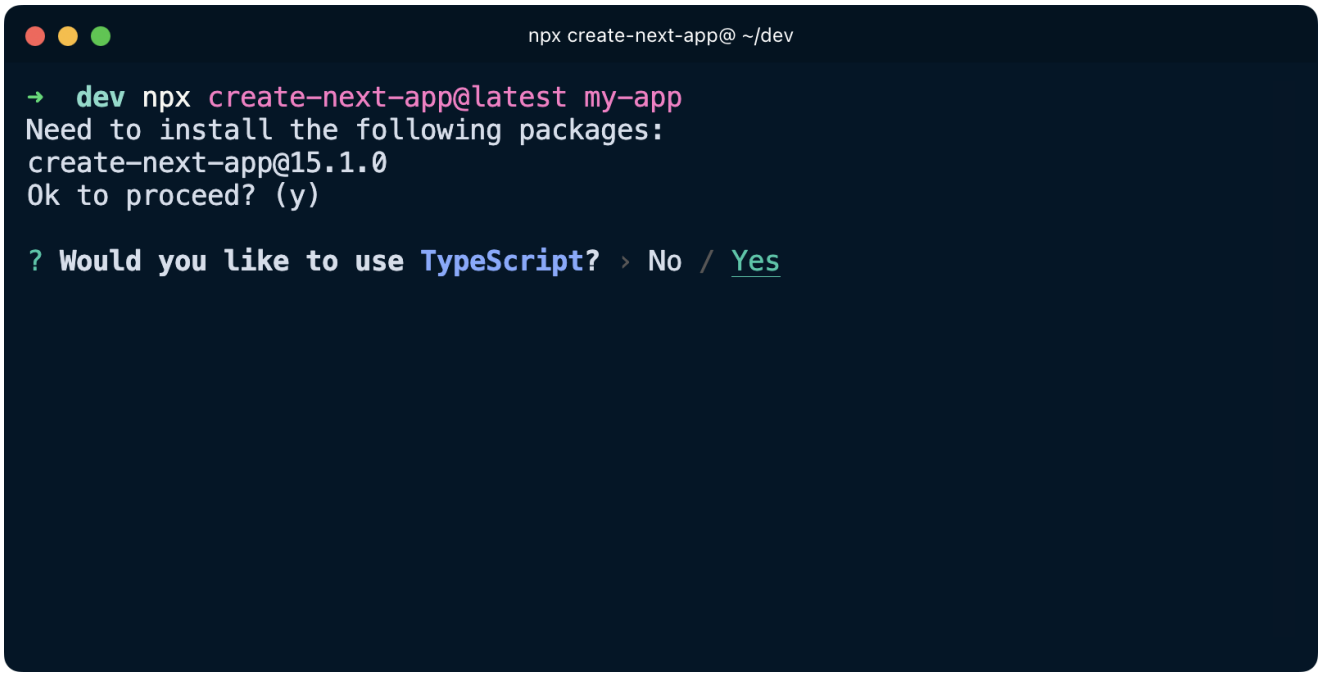
To install Next.js, you need to have Node.js installed.

Make sure you have that installed, a recent version is highly recommended. See my Node.js Handbook if you’re unsure how to proceed.

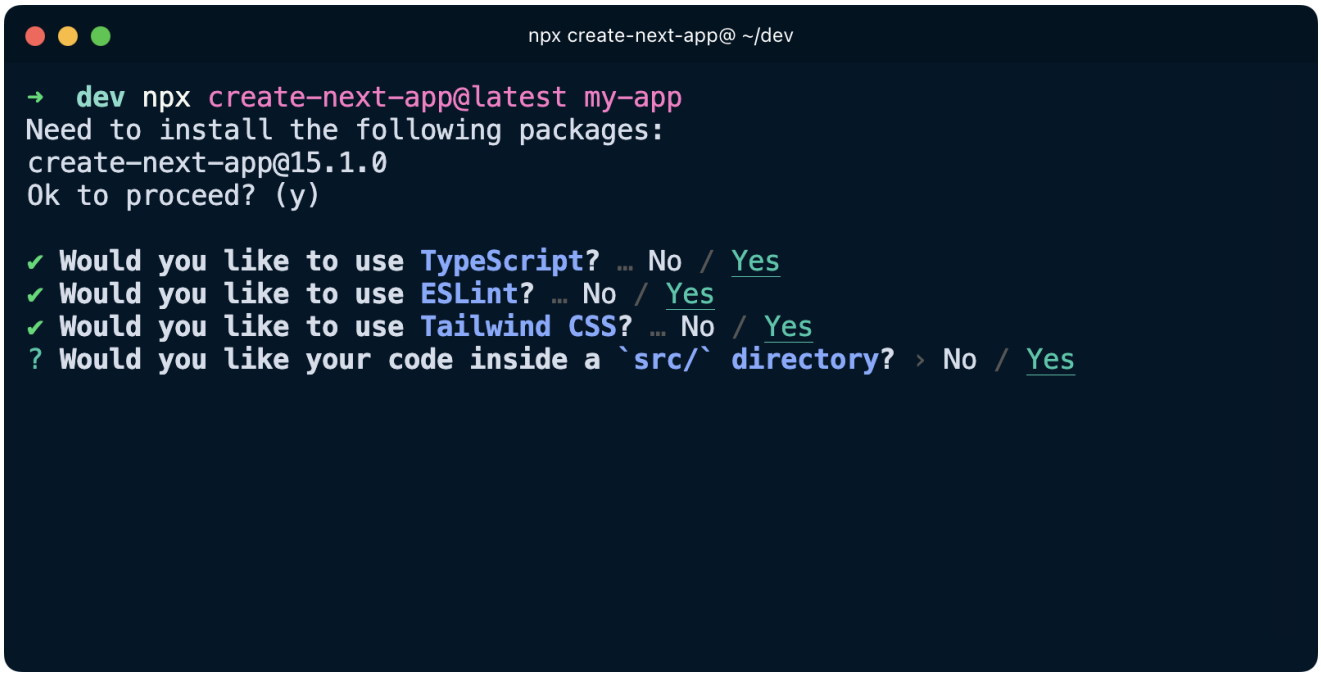
Once you’re done, we use `create-next-app` to create a Next.js app, in this way

```
npx create-next-app@latest my-app
```

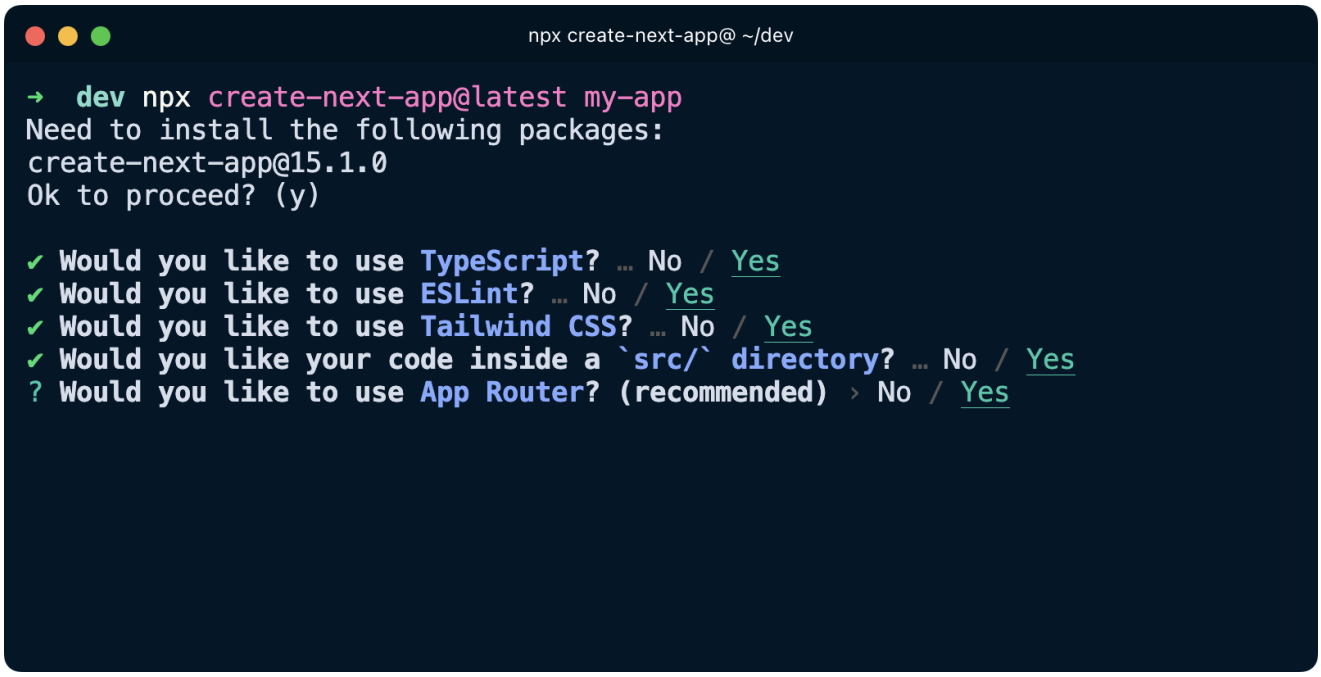
You can choose “Yes” when they ask you if you want to use TypeScript:



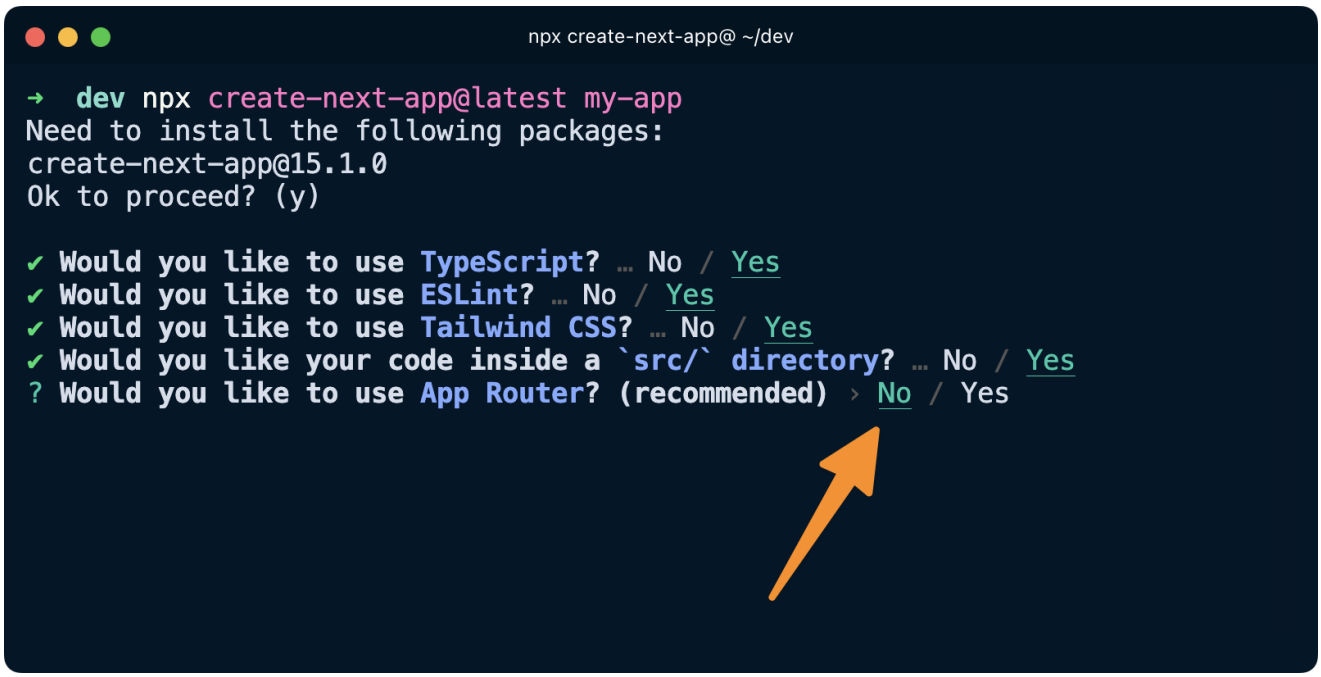
Then click “Yes” to use ESLint and Tailwind CSS:



Next, pick “yes” when it asks to use the `src/` directory:

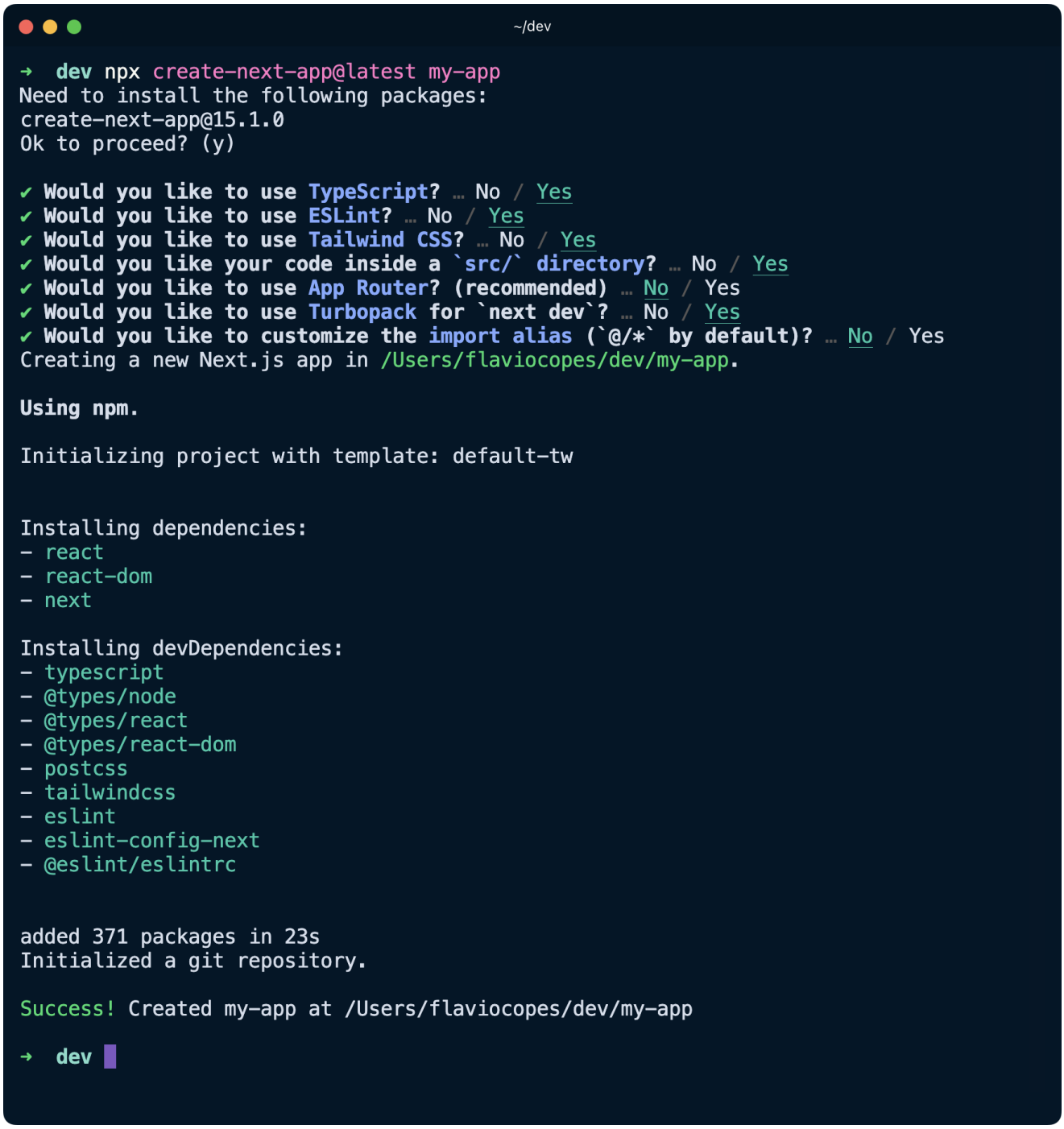


and “No” to configure Next.js to only use the Pages Router, because this book focuses on it rather than the App Router:



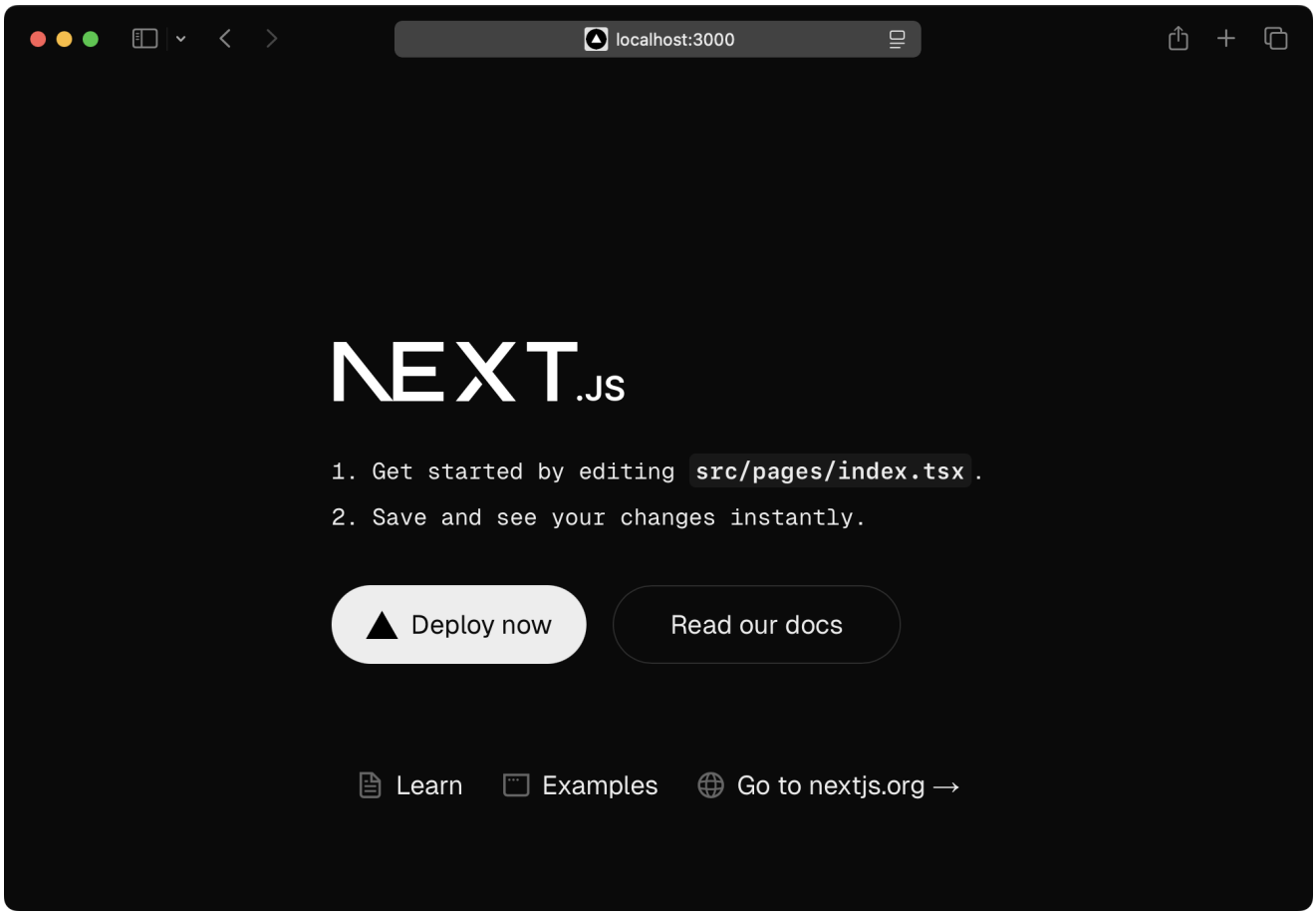
> NOTE: I have another book on the App Router. You should learn the Pages Router only if your company uses it, the future of Next.js is all about the App Router.

Continue the installation with the default options:



Now you can immediately run the sample app by going in the folder `my-app` and running `npm run dev`:

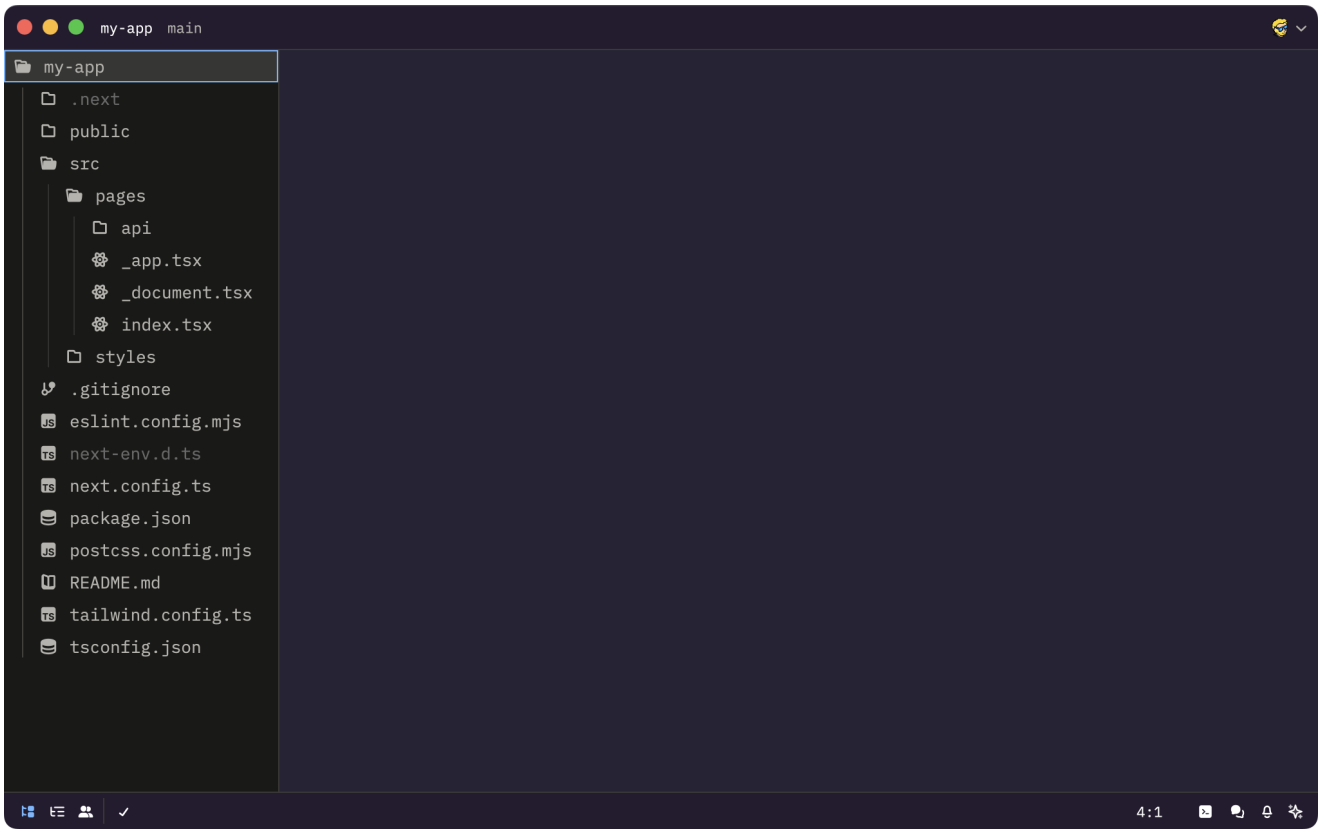
And here's the result on http://localhost:3000:



🤦 Note: if you have some other app running on port 3000, maybe one you worked on before but forgot to stop, the app will automatically start on port 3001 3002 and so on. Just check what `npm run dev` prints to the terminal.

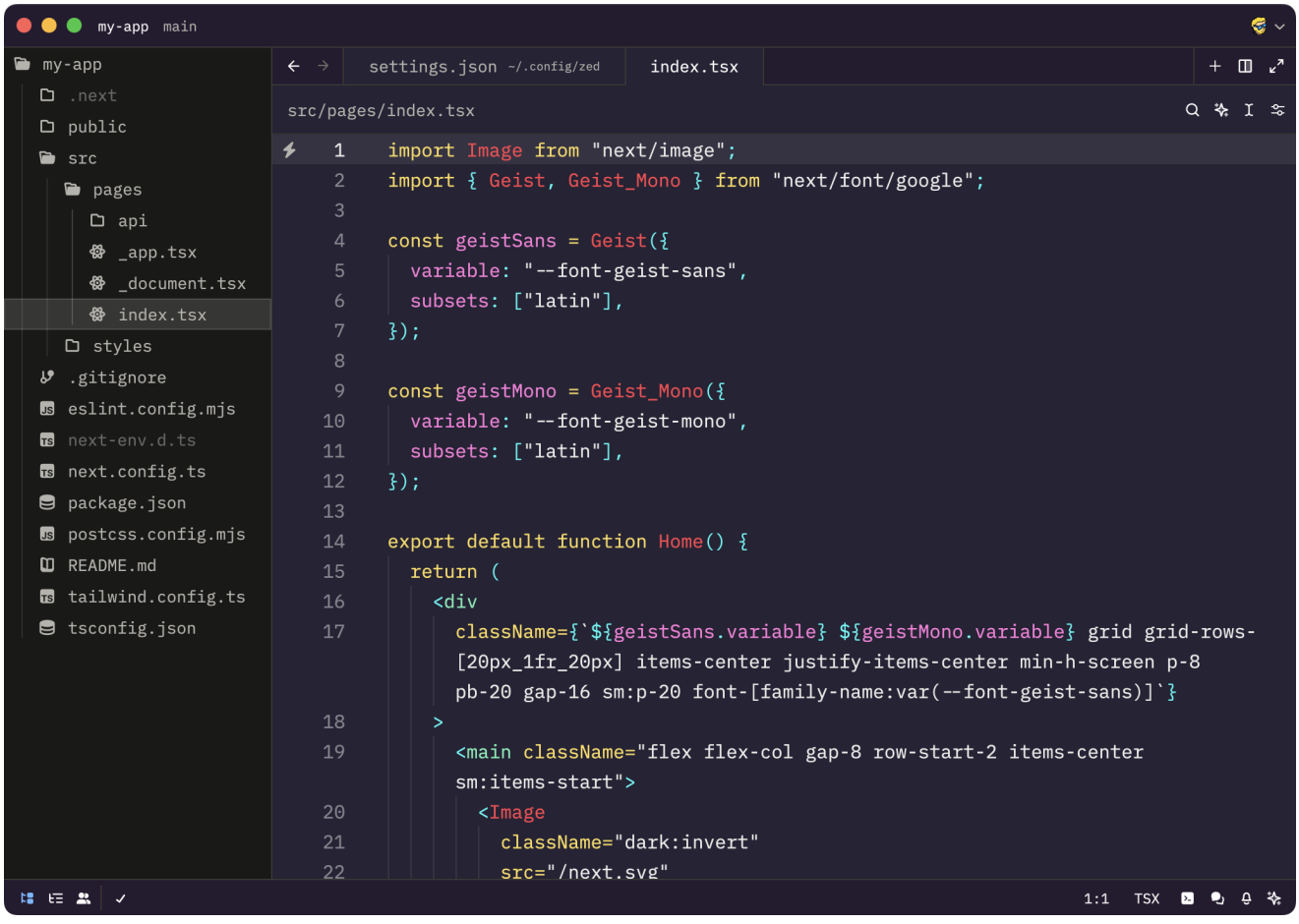
Now open the app's folder in your favorite editor.

We have a bunch of files that serve as the initial configuration and structure for the default app.



Open `src/pages/index.tsx`.

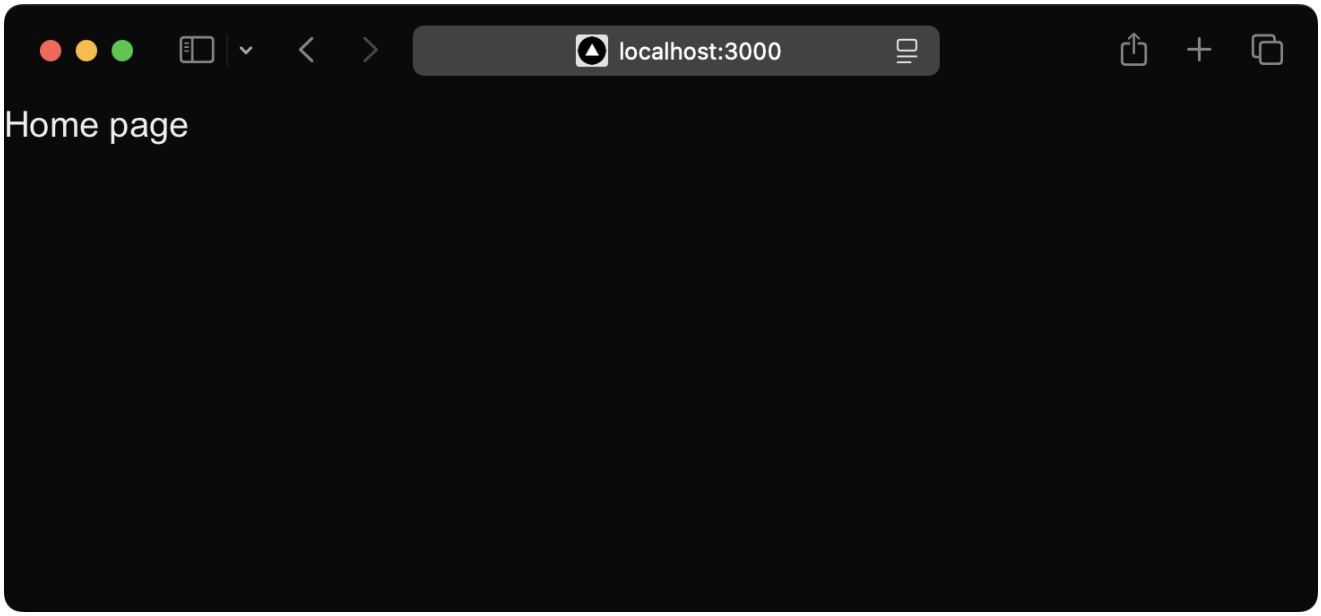
This is the file that defines the homepage content:



Select all of the content in this file, and add this instead, so we can start simple:

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

Save and this will be the result in the browser



Next.js vs Vite

Before reading this book, I highly recommend reading the **React Beginner's Handbook**, written by myself as well. In that book I explain how to use **Vite** to create a React application.

That's super helpful to get us up and running with React.

Vite is a **pure frontend tool**, and it has no support for the backend.

It's great for creating SPA (Single Page Application) apps, for example dashboards.

Using Next.js, we can create a backend to fetch data, and alter data (adding new data, or modifying existing data). Next.js can also help us with **server-side rendering**, which is essential for most sites to deliver a speedy and SEO-optimized experience to our users. Next.js also allows us to create both static and dynamic websites, depending on our needs. It's quite flexible.

And it provides support for various things needed by every website like file-based **routing**, image optimization, API routes and more.

Depending on what you're working on, you might prefer one tool over the other.

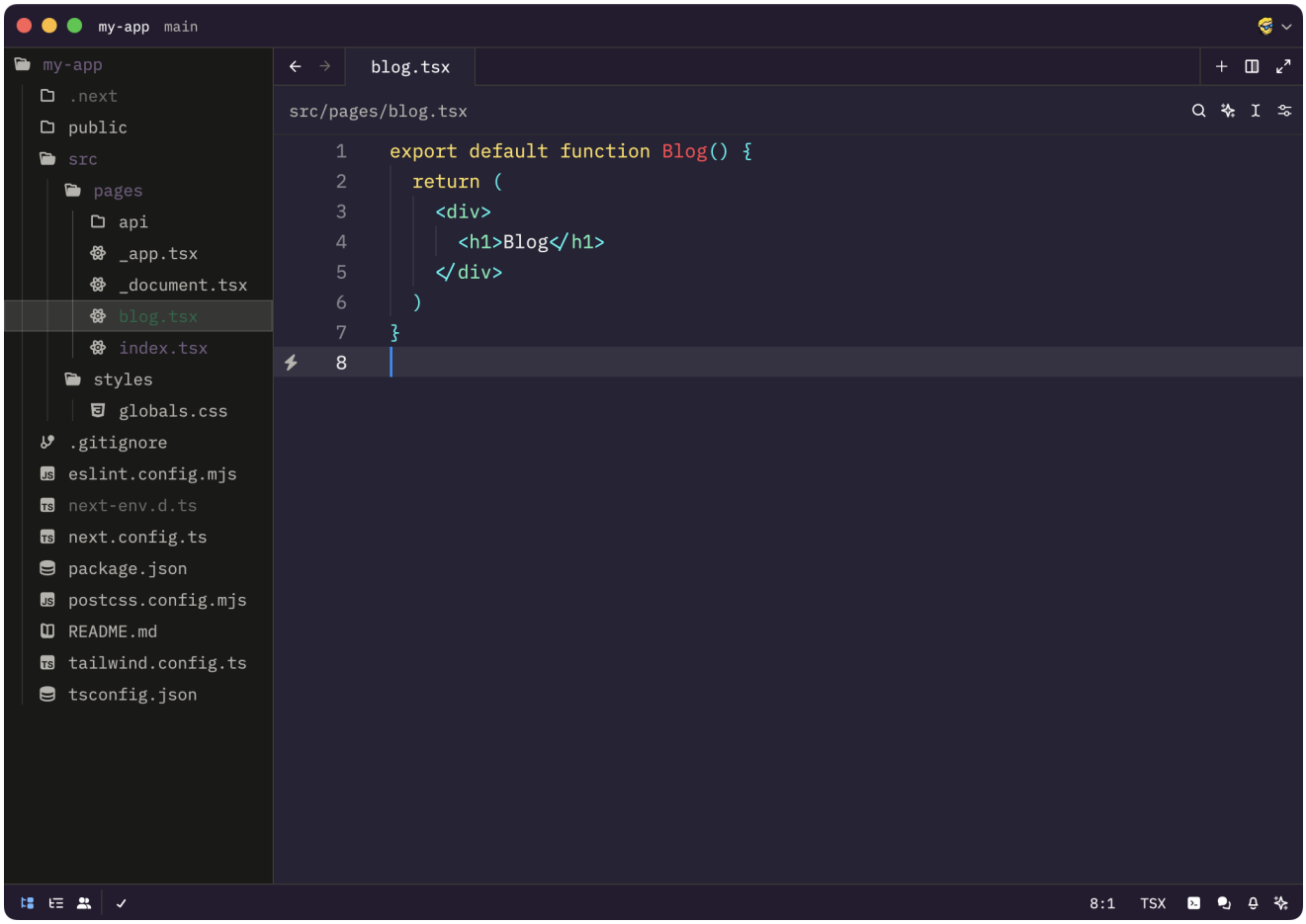
Experience will help you with that.

Adding a second page to the site

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 page component `src/pages/index.tsx`.

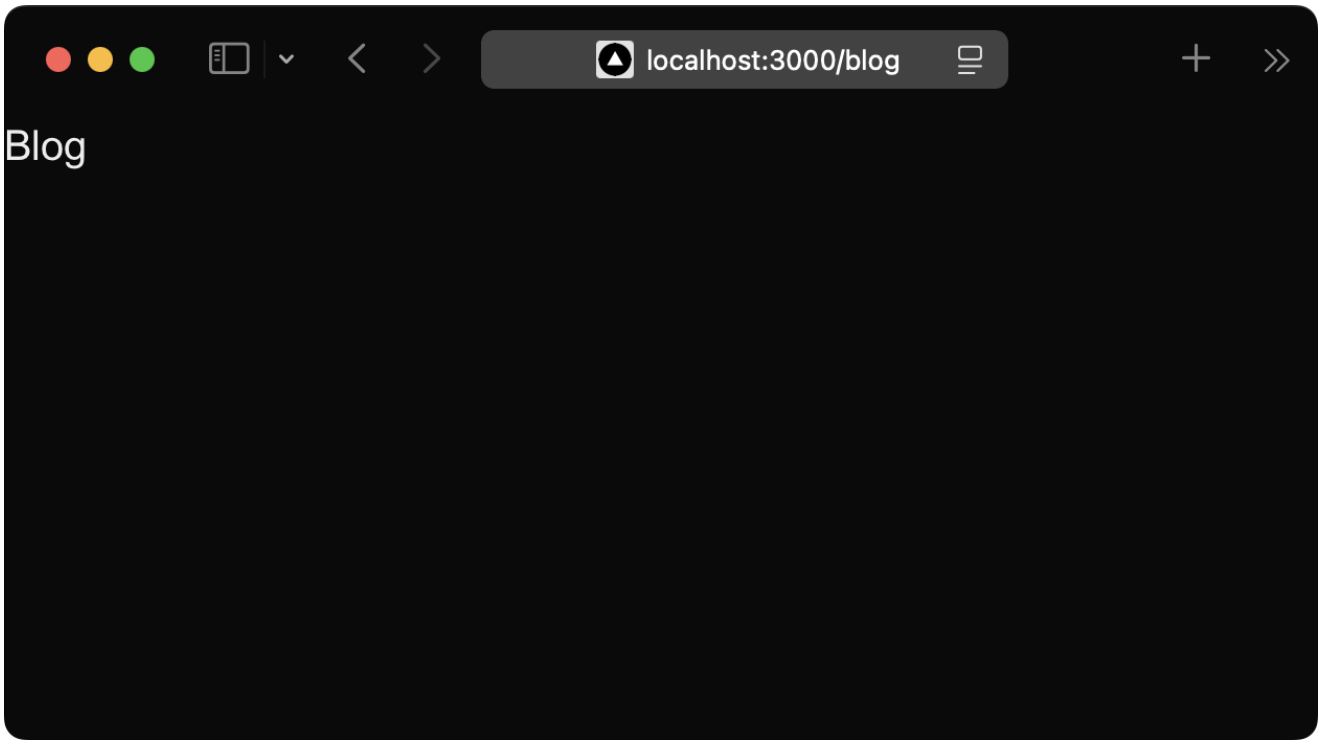
Create a new file in `src/pages/blog.tsx`, with this content:

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



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 like we had to do for Node.js projects.

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



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

Subfolders work in the same way, you could create a `pages/hey/ho.tsx` page, and that page would show up on the URL http://localhost:3000/hey/ho.

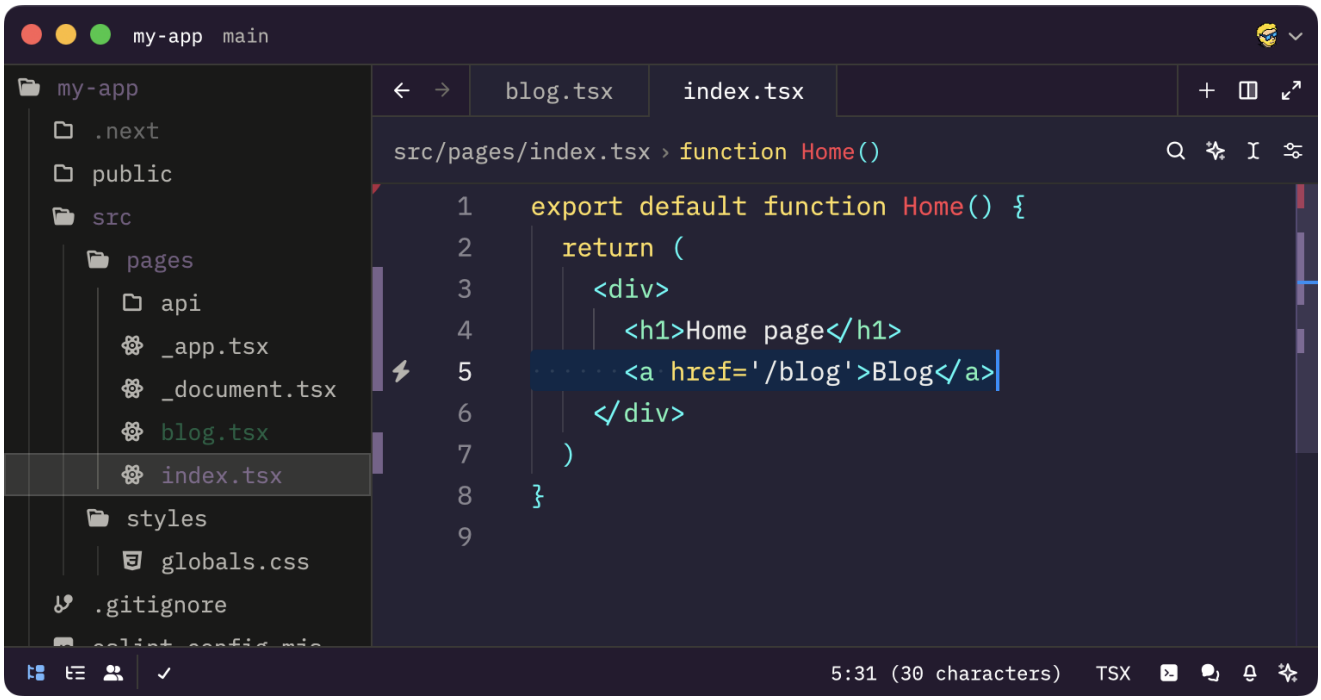
Linking the two pages

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

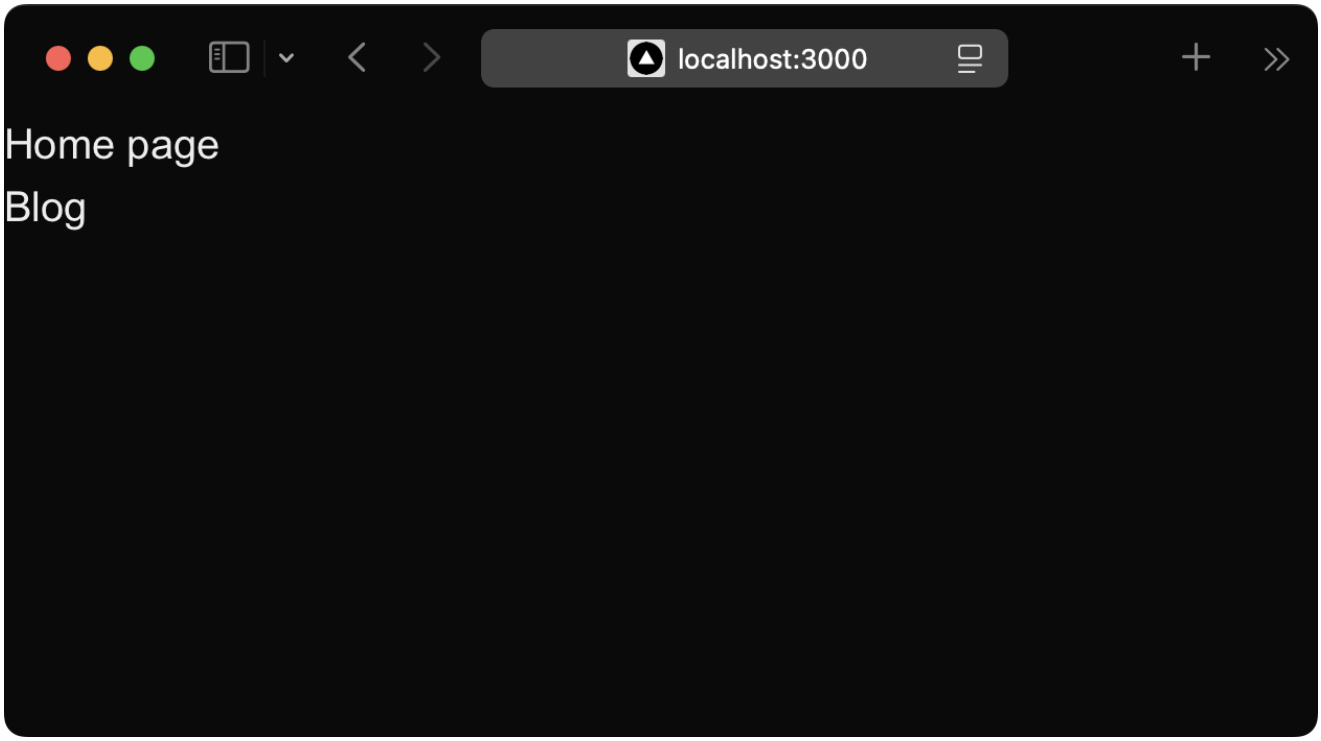
Normal HTML links within pages are done using the `a` tag:

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

We can use this way in Next.js too:



And it works:



Notice the link is not styled due to the Tailwind CSS preflight that removes all default styles, but if you hover the link, you can click it. And if you click that link, the browser will do a full reload of the page, and render the blog page, like it happens normally in Web pages.

However, 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 navigation**.

That's something powered by React, and it makes our page transitions very fast.

To enable that, you need to use the `<Link>` component offered by Next.js.

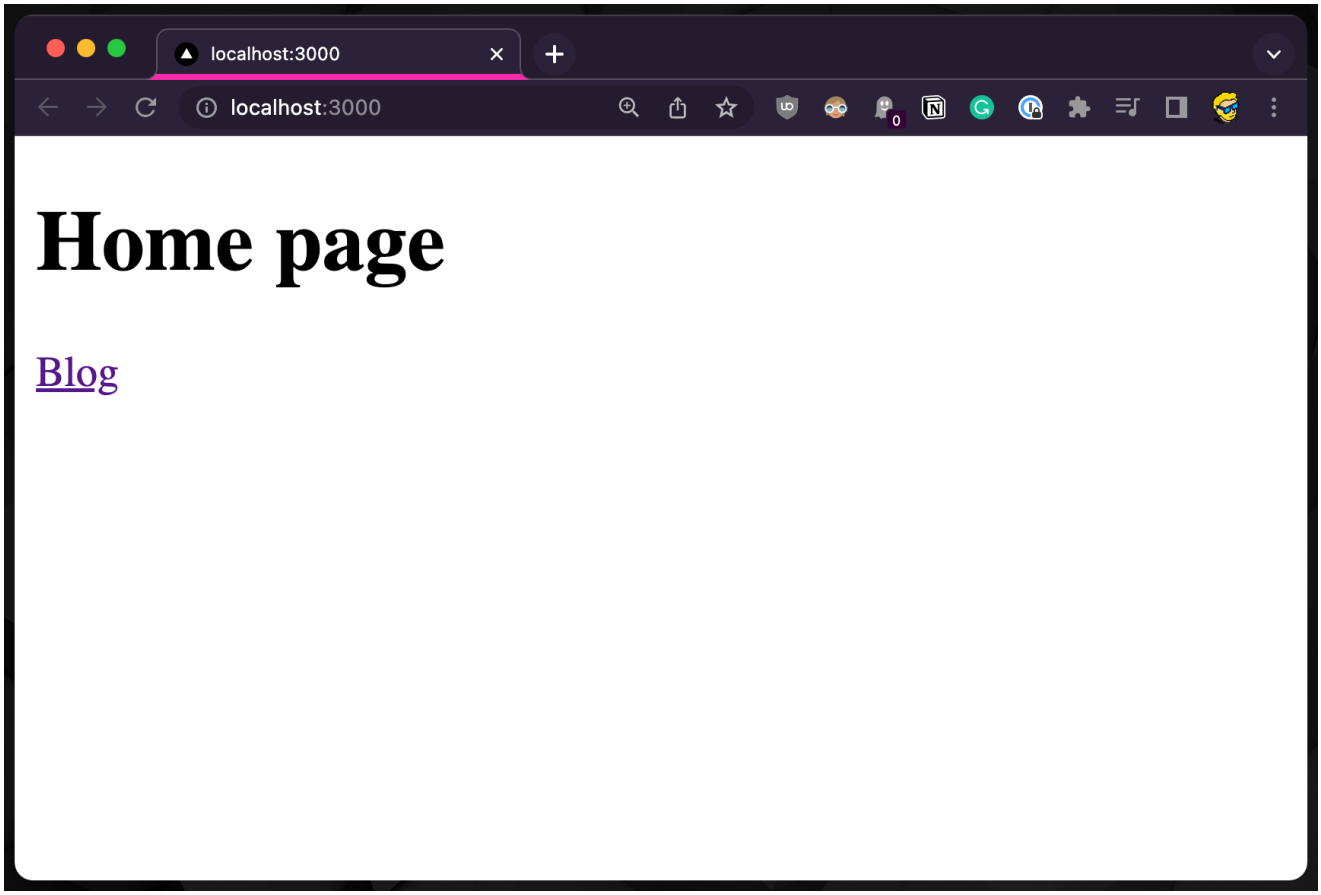
It’s a bit of additional work, but worth it.

Not only the page transition will be faster, but less data will be sent to the client.

We import the component from `next/link` and then we use it to wrap our link, like this:

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

export default function Home() {
  return (
    <div>
      <h1>Home page</h1>
      **<Link href='/blog'>Blog</Link>**
    </div>
  )
}
```

And the link will appear in the same way:



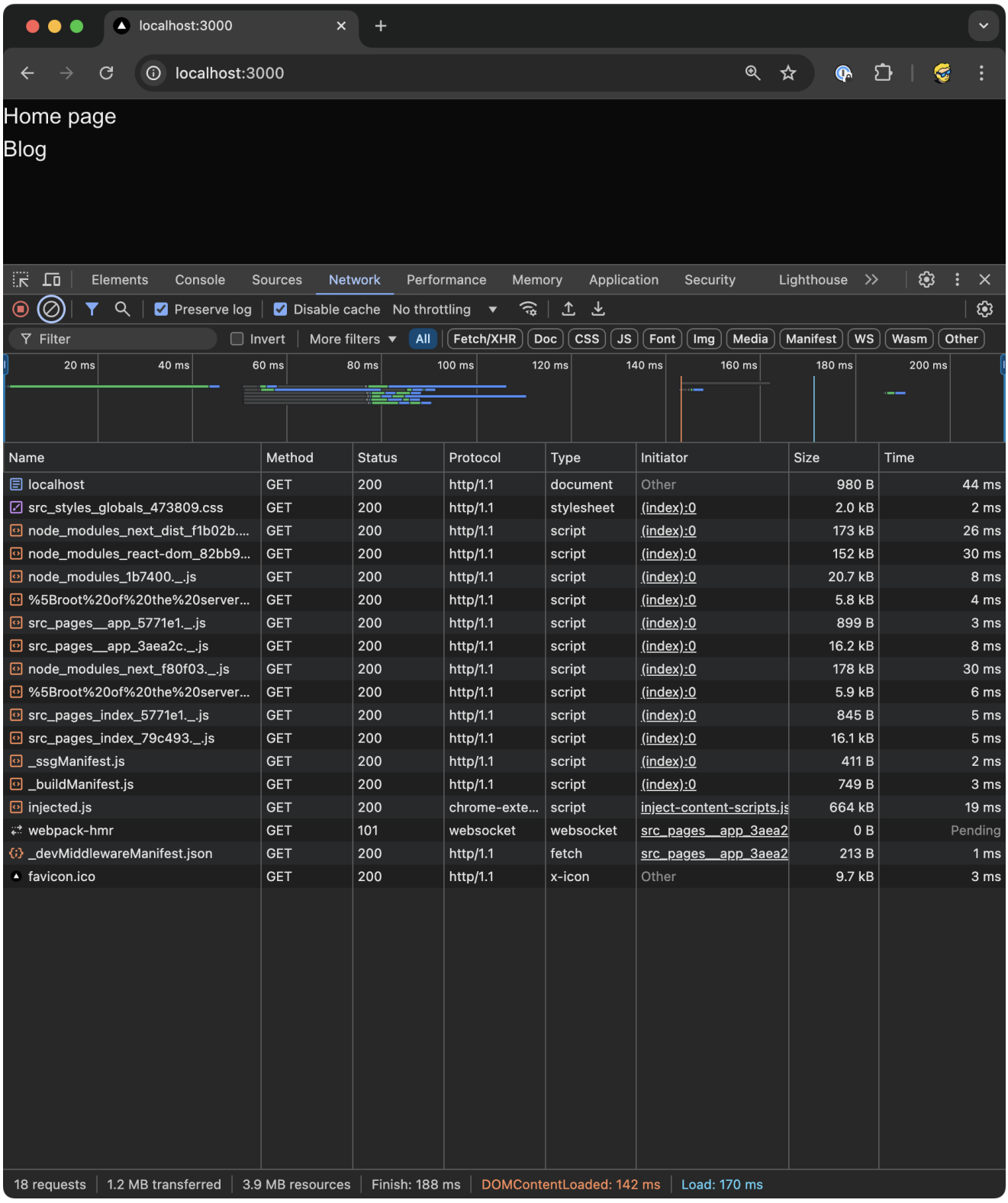
But now the link behaves differently.

Let’s do this experiment to see the difference in practice.

Use the `<a>` link in your page, then open the browser **DevTools** by right-clicking in the page and clicking “Inspect”.

Then in the DevTools open the **Network panel.**

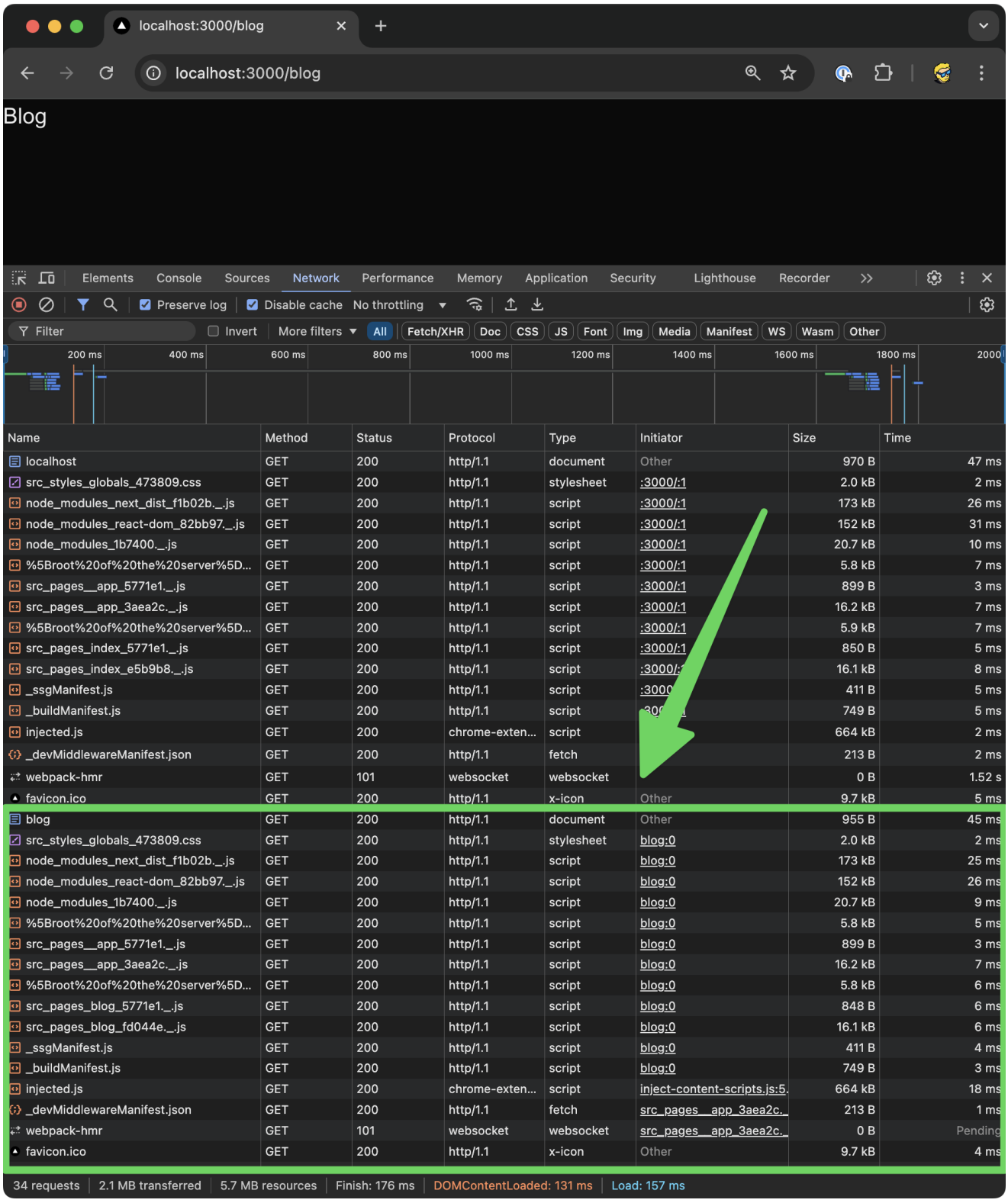
The first time we load `http://localhost:3000/` we get all the *page bundles* loaded, that's all the code Next.js needs to run in development mode:



It's ~1.2MB of resources transferred (this is development mode so Next.js needs to load more resources, production will be *a lot* less heavy due to all the optimizations and less client-side JavaScript needed).

On the Chrome Developer Tools, enable the "Preserve log" button to avoid clearing the Network panel when you click the link.

Restore the old `<a>` link we used before to add navigation between pages and when you click the "Blog" link, this is what happens:



We got a bunch of stuff from the server, again! Another ~1.2MB of data.

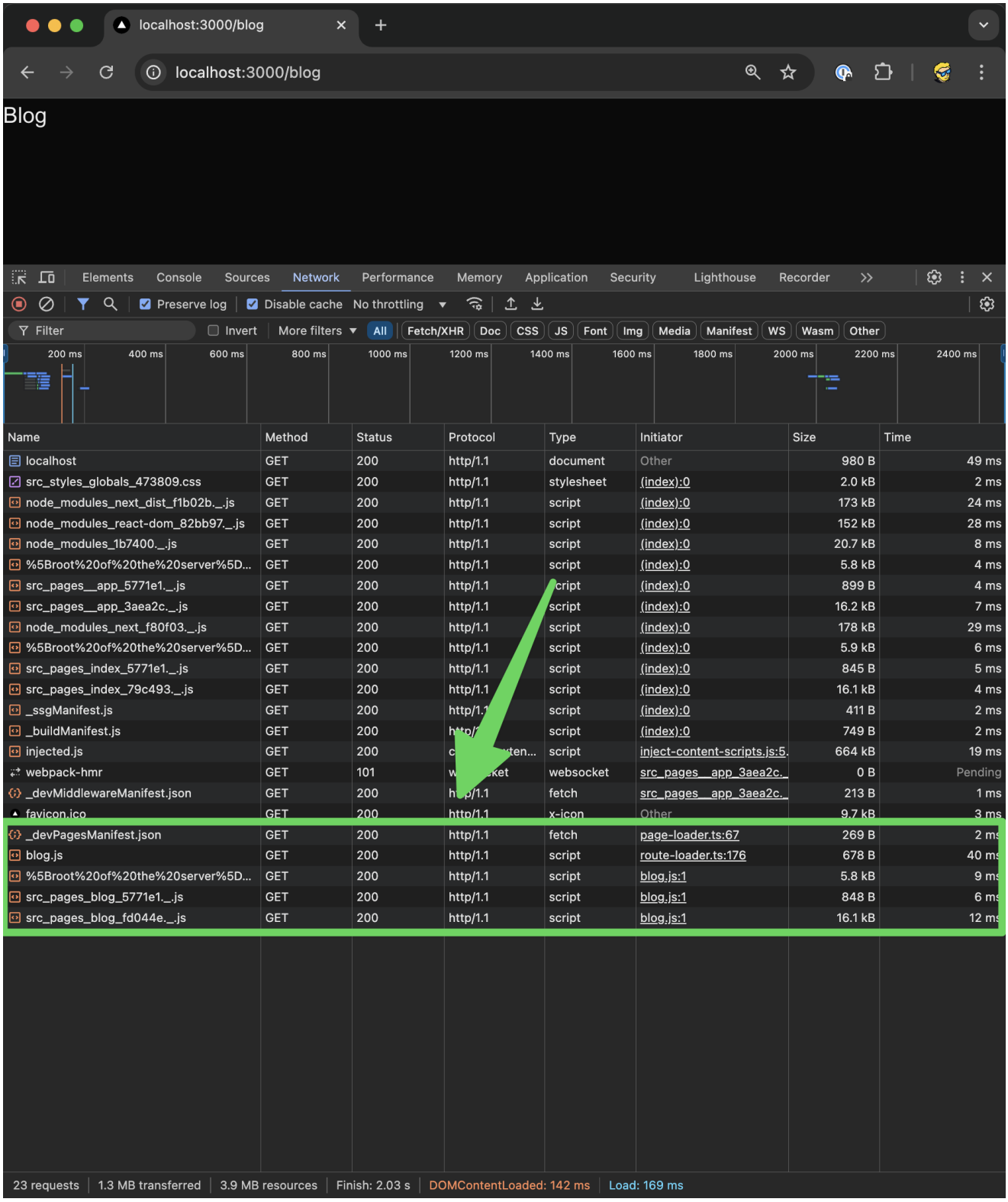
Next.js is downloading again a ton of stuff it was already loaded in the browser.

But.. we don't need all that JavaScript if we already got it.

We'd just need the new page bundle, the only one that's new to the page.

To fix this problem, we use the `Link` component provided by Next.js

Using that, if you retry the thing we did previously, you'll be able to see that only a couple kB of files are loaded when we click the link to the blog page:



And the page loaded so much faster than before.

This is client-side navigation in action.

What if you now press the back button and then click the link again? Notice nothing is being loaded in the network panel anymore, because the browser now has all the information

needed to render the pages, it's all automatic!

Dynamic content with the router

In the previous lesson 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 `src/pages/blog` folder, and inside it we add a file named `[id].tsx`.

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 to handle navigation.

To use it, 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].tsx` file by accessing `router.query.id`.

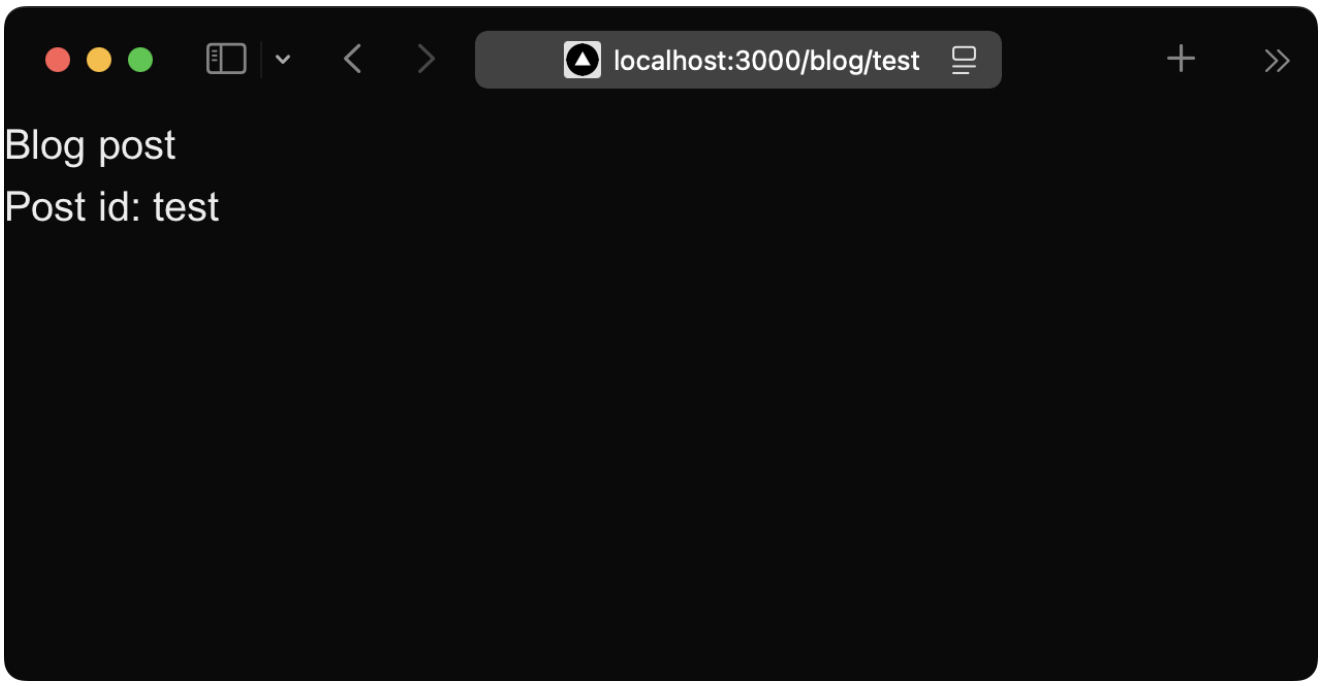
> The dynamic part can also just be a portion of the URL, like `post-[id].tsx`.

So let's go on and apply all those things in practice in our first Next.js project.

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

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

export default function BlogPost() {
  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` page, you should see this:



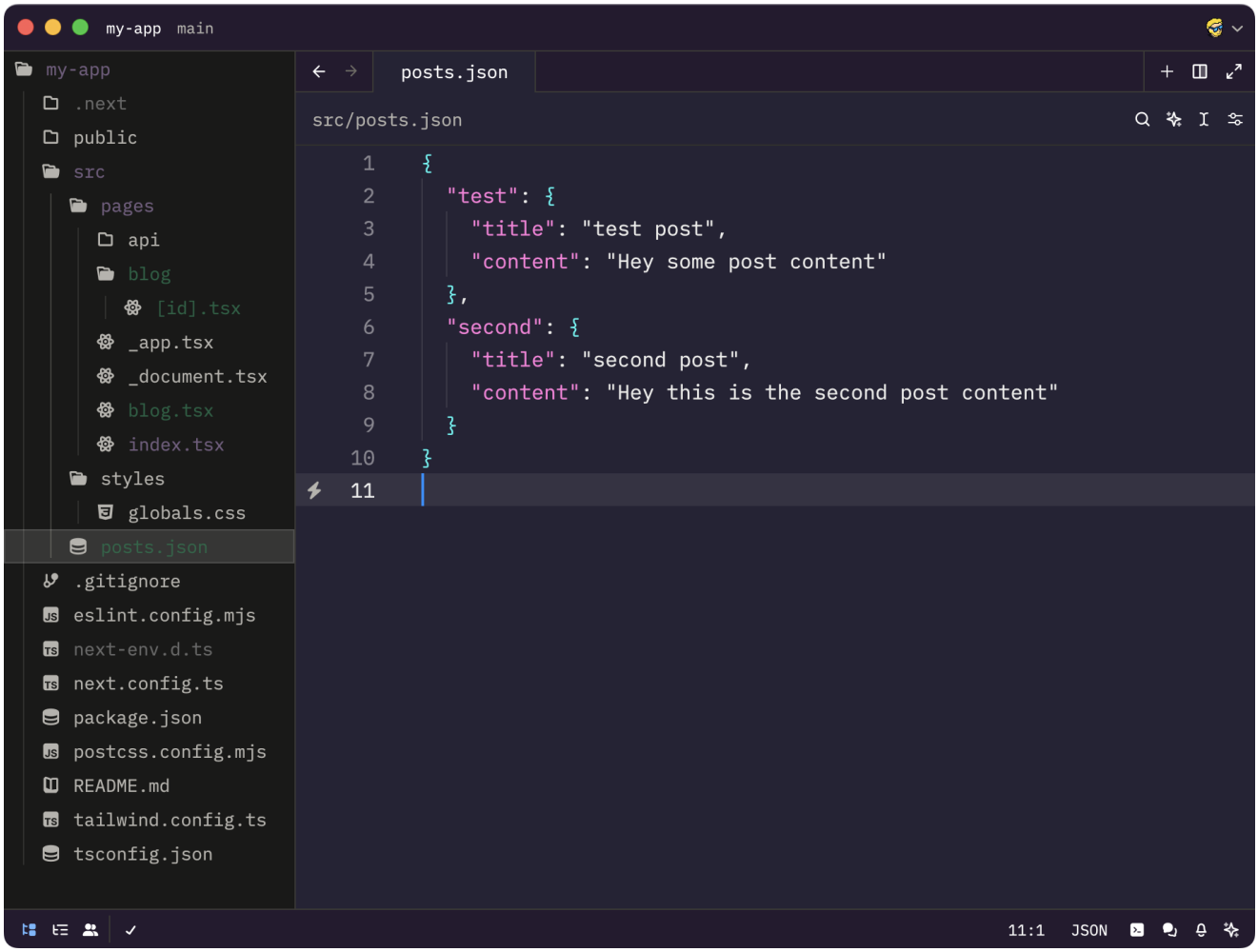
We can use this `id` parameter to gather the post from a list of posts from a database, for example.

To keep things simple we'll use a JSON file.

Create a `posts.json` file in the `src` 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 define the types of the data, and lookup the post from the `id` key in the single blog post page we just created.

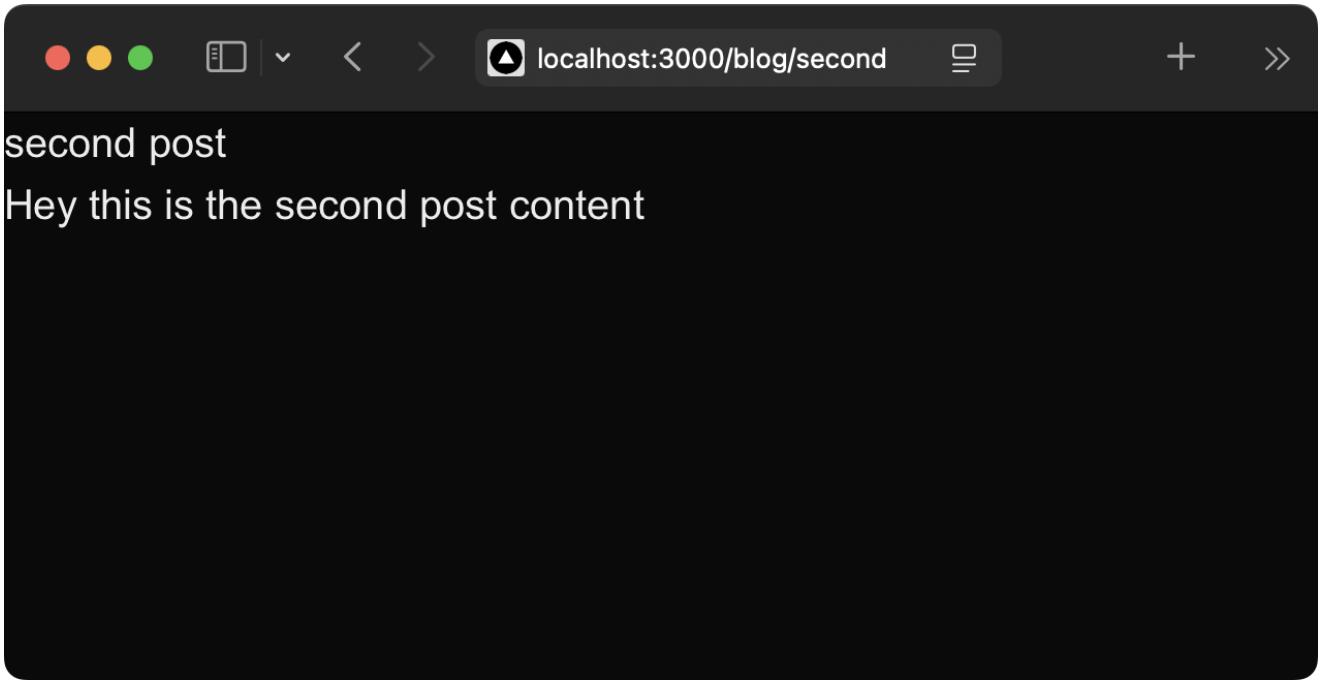
```
import { useRouter } from 'next/router'
**import posts from '@/posts.json'

interface Post {
  title: string
  content: string
}

interface Posts {
  [key: string]: Post
}**

export default function BlogPost() {
  const router = useRouter()
```

```
  **const post = postsData[router.query.id as string]

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

Let’s also handle the loading state, and the case where the URL does not match any post:

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

interface Post {
  title: string
  content: string
}

interface Posts {
  [key: string]: Post
}

const postsData = posts as Posts

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

  **if (router.isFallback || !router.query.id) {
    return <div>Loading...</div>
  }**

  const post = postsData[router.query.id as string]

  **if (!post) {
    return <div>Post not found</div>
  }**

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

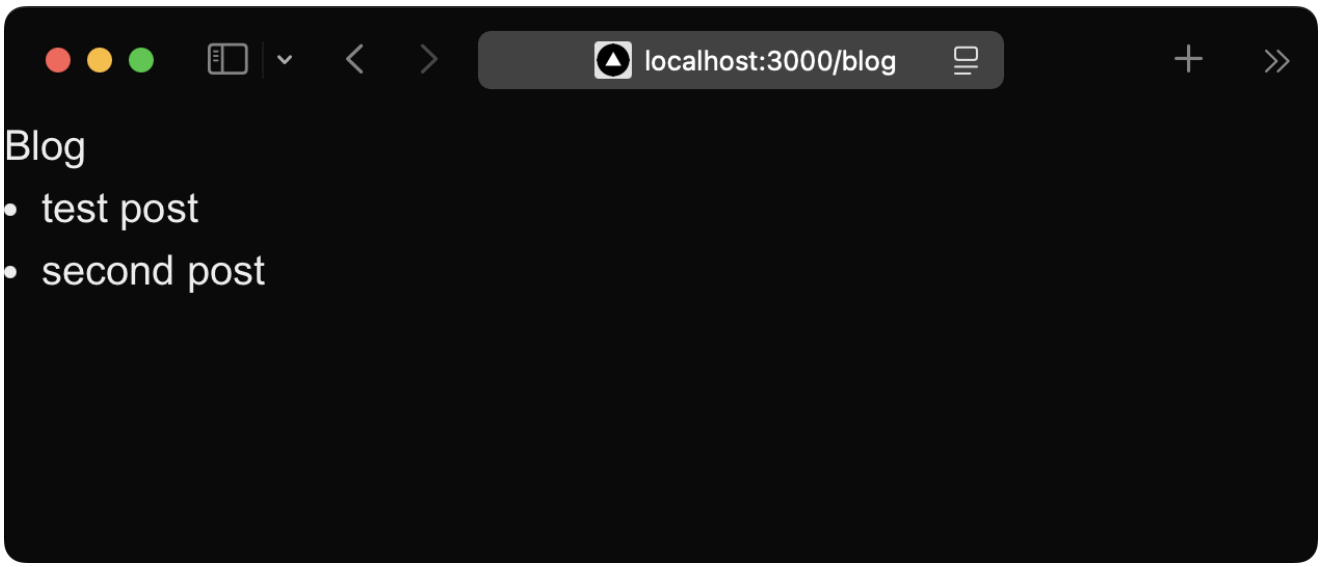
Now things should work. Initially the component is rendered without the dynamic `router.query.id` information. After rendering, Next.js triggers an update with the query value and the page displays the correct information.



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

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

interface Post {
  title: string
  content: string
}

interface Posts {
  [key: string]: Post
}

const postsData = posts as Posts**

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

      **<ul className='list-disc list-inside'>
        {Object.keys(posts).map((id, index) => {
          return <li key={index}>{postsData[id].title}</li>
        })}
      </ul>**
    </div>
```

```
  )
}
```



See here a short description for `Object.keys()`. We use it to loop over the posts object as if it was an array.

We're duplicating the type definition now, so let's move that to the file `src/types/posts.ts`:

```
export interface Post {
  title: string
  content: string
}

export interface Posts {
  [key: string]: Post
}
```

Import this in `src/pages/blog/[id].tsx`:

```
import { useRouter } from 'next/router'
import posts from '@/posts.json'
**import type { Posts } from '@/types/posts'**

~~interface Post {
  title: string
  content: string
}

interface Posts {
  [key: string]: Post
}~~
```

```
const postsData = posts as Posts

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

  if (router.isFallback || !router.query.id) {
    return <div>Loading...</div>
  }

  const post = postsData[router.query.id as string]

  if (!post) {
    return <div>Post not found</div>
  }

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

and `src/pages/blog.tsx`:

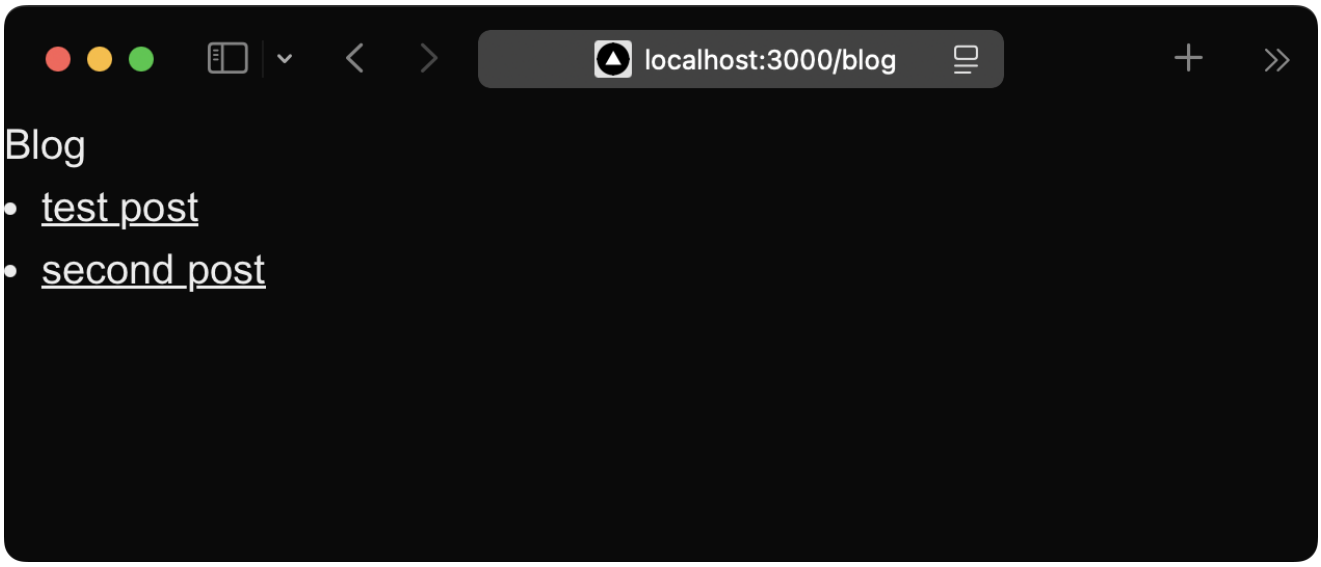
```
import posts from '@/posts.json'
**import type { Posts } from '@/types/posts'**

~~interface Post {
  title: string
  content: string
}

interface Posts {
  [key: string]: Post
}~~

const postsData = posts as Posts

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

      <ul className='list-disc list-inside'>
        {Object.keys(posts).map((id, index) => {
          return <li key={index}>{postsData[id].title}</li>
        })}
```

```
      </ul>
    </div>
  )
}
```

We can link each post in the list to the individual post pages, by importing `Link` from `next/link` and using it inside the posts loop in `src/pages/blog.tsx`:

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

import posts from '@/posts.json'
import type { Posts } from '@/types/posts'

const postsData = posts as Posts

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

      <ul className='list-disc list-inside'>
        {Object.keys(posts).map((id, index) => {
          return (
            <li key={index}>
              **<Link className='underline' href={'/blog/' + id}>
                {postsData[id].title}
              </Link>**
            </li>
          )
        })}
      </ul>
    </div>
  )
}
```

Here’s the result:



Data fetching

On a website we can have 2 kinds of pages:

- Pages that do not need any data from the server
- Dynamic pages that need to fetch data before rendering

The first kind of page does not need anything special to “exist”.

But if you need a page to get data from a database or the network, for example, you’ll need to add a function to your page components called `getServerSideProps`.

This Next.js function has the task of fetching data and returning it as an object with the `props` property.

```
export async function getServerSideProps() {
  return {
    props: {
      //... the props returned
    }
  }
}
```

These props are then passed to the main page component.

It’s important to note that **this function runs on the server**, not client-side.

When a page component has this function associated, whenever a user visits the URL, the page is rendered from the server.

The server must do some work before the page is fully served, so the page will load slower than static pages that don’t have this function, where there is no data processing involved at all.

Note that Next.js will first render a page without the data, and then when the data becomes available it will add the data to the page. This gives you the option to send the user a skeleton of the page quickly, but then the user will see a “loading..” screen until the data is ready.

Sometimes server-side data fetching is the only way to provide useful information, for example when a database is involved.

Here’s an example where we load the details of the user with id `332` from the database (`User` in this case is a Prisma model that lets us access the database):

```
export async function getServerSideProps() {
  const user = await User.findOne(332)
  return {
    props: {
                          user
                }
  }
}
```

Then we can use this data in the component by getting the `user` from its props:

```
export default function User(**{ user }**) {
  return (
    <p>This is user {user.name}</p>
  )
}

export async function getServerSideProps() {
  const user = await User.findOne(332)
  return {
    props: {
                          user
                }
  }
}
```

You can also fetch data from the network.

For example let’s do a `fetch()` API call **server-side** to get something from the network in `getServerSideProps`:

```
export async function getServerSideProps() {
  const res = await fetch(
    `https://dog.ceo/api/breeds/image/random`)
  const data = await res.json()
  return {
```

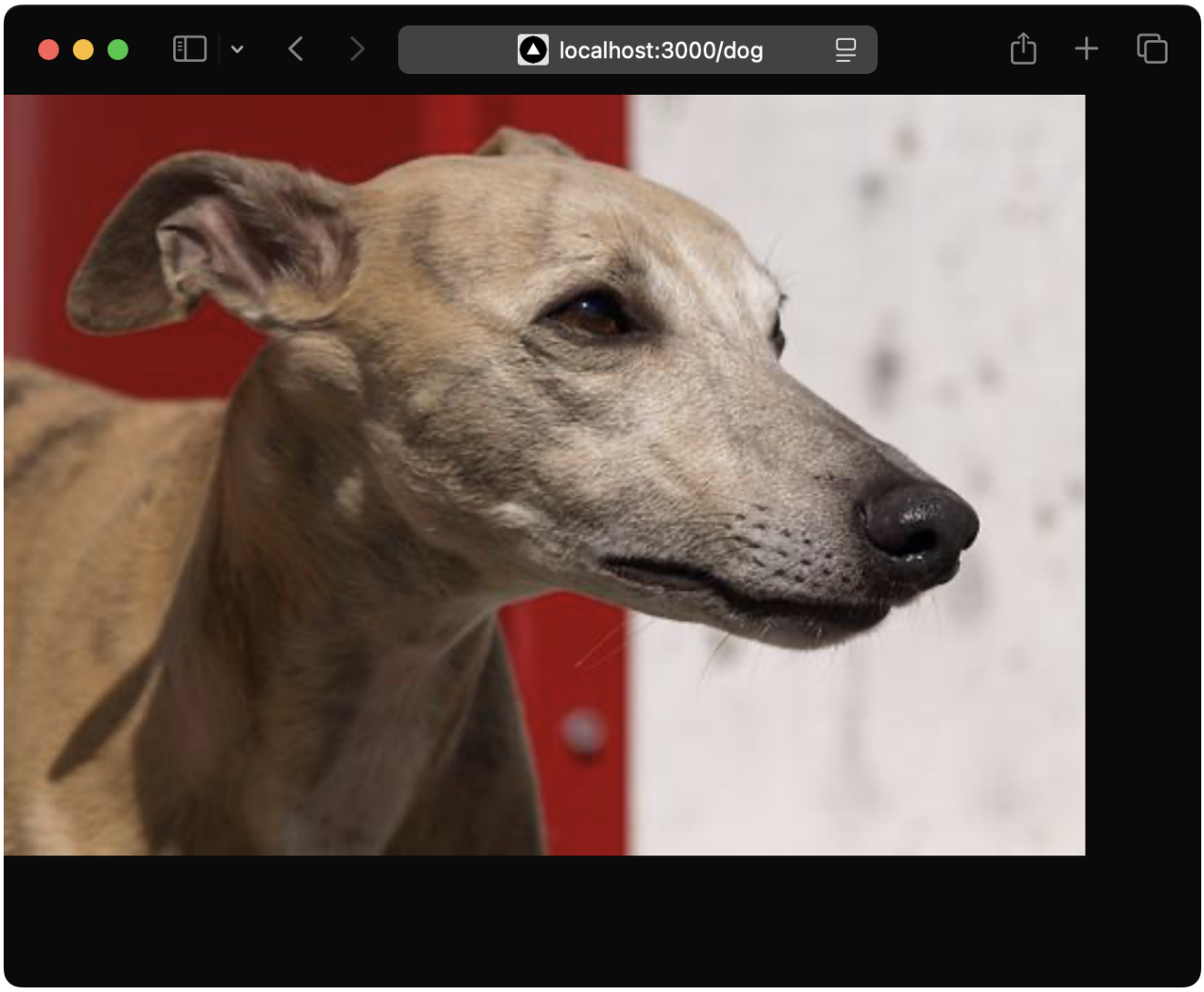
```
    props: {
                        image: data.message
                }
  }
}
```

Now we can use this data in the component:

```
export default function DogImage({ image }) {
  return <img src={image ?? ''} alt='A picture of a dog' />
}

export async function getServerSideProps() {
  const res = await fetch(`https://dog.ceo/api/breeds/image/random`)
  const data = await res.json()
  return {
    props: {
                        image: data.message
                }
  }
}
```

This is server-side data fetching.

Save this page component into a `src/pages/dog.tsx` file and you should get the following result (the dog picture is random and will change on every page load):



Depending on what you're trying to do, sometimes you might want to load this data from the browser, **client-side**.

Let's see how.

We do so by avoiding the use of `getServerSideProps` and by instead using the React hook `useEffect()` , which is executed client-side when the component is loaded:

```
import { useState, useEffect } from 'react'
****
export default function DogImage() {
  **const [image, setImage] = useState(null)

  useEffect(() => {
    async function getData() {
            const res = await fetch(
        `https://dog.ceo/api/breeds/image/random`)
            const data = await res.json()
            setImage(data.message)
                }
    getData()
  }, [])**
```

```
  return <img src={image ?? ''} alt='A picture of a dog' />
}

~~export async function getServerSideProps() {
  const res = await fetch(`https://dog.ceo/api/breeds/image/random`)
  const data = await res.json()
  return {
    props: {
                        image: data.message
                }
  }
}~~
```

So we discussed two differences: fetching server-side, or client-side.

There's another way in Next.js, **static data fetching**, and we'll see it in the next lesson.

Static data fetching at build time

We talked about data fetching when a user visits a page, both in the backend and in the frontend.

Next.js, with its Pages Router, also offers another way. It's called **static data fetching**.

Suppose you have a blog. You have a set of blog posts, perhaps published on a service like Contentful or Sanity. Or on a headless Wordpress install.

You can tell Next.js to fetch that content at build time, and generate static pages that are then served to the user without further action.

It's the best of both worlds: your data is dynamic in nature, but you create static pages from it.

How does it work?

You have to define and export 2 functions in your page component:

- `getStaticPaths`
- `getStaticProps`

The first defines the dynamic URLs that the page allows.

Remember a few lessons back how we made a page that served blog posts?

We had a `posts.json` file in the project root folder containing the list of blog posts, and a `src/pages/blog/[id].tsx` page serving each post, which looks like this at the moment:

```
import { useRouter } from 'next/router'
import posts from '@/posts.json'
import type { Posts } from '@/types/posts'

const postsData = posts as Posts

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

  if (router.isFallback || !router.query.id) {
    return <div>Loading...</div>
  }

  const post = postsData[router.query.id as string]

  if (!post) {
    return <div>Post not found</div>
  }

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

Blog posts are server rendered. On each call we look for the post data server-side, and the HTML is rendered.

Since this data never changes, we can statically render posts at build time.

To do so, add a `getStaticPaths` function that exports the ids of the posts we defined in the JSON file:

```
import { useRouter } from 'next/router'
import posts from '@/posts.json'
import type { Posts } from '@/types/posts'

const postsData = posts as Posts

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

  if (router.isFallback || !router.query.id) {
    return <div>Loading...</div>
  }
```

```
  const post = postsData[router.query.id as string]

  if (!post) {
    return <div>Post not found</div>
  }

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

**export const getStaticPaths = async () => {
  return {
    paths: Object.keys(posts).map((id) => ({ params: { id } })),
    fallback: false,
  }
}**
```

Now add a `getStaticProps` function that is called for every one of those paths array you returned from `getStaticPaths`. We also define its returned type interface:

```
import { useRouter } from 'next/router'
import posts from '@/posts.json'
import type { Posts, Post } from '@/types/posts'
**import type { GetStaticProps } from 'next'**

const postsData = posts as Posts

**interface BlogPostProps {
  post: Post
}

export default function BlogPost({ post }: BlogPostProps) {**
  const router = useRouter()

  if (router.isFallback || !router.query.id) {
    return <div>Loading...</div>
  }

  if (!post) {
    return <div>Post not found</div>
  }

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

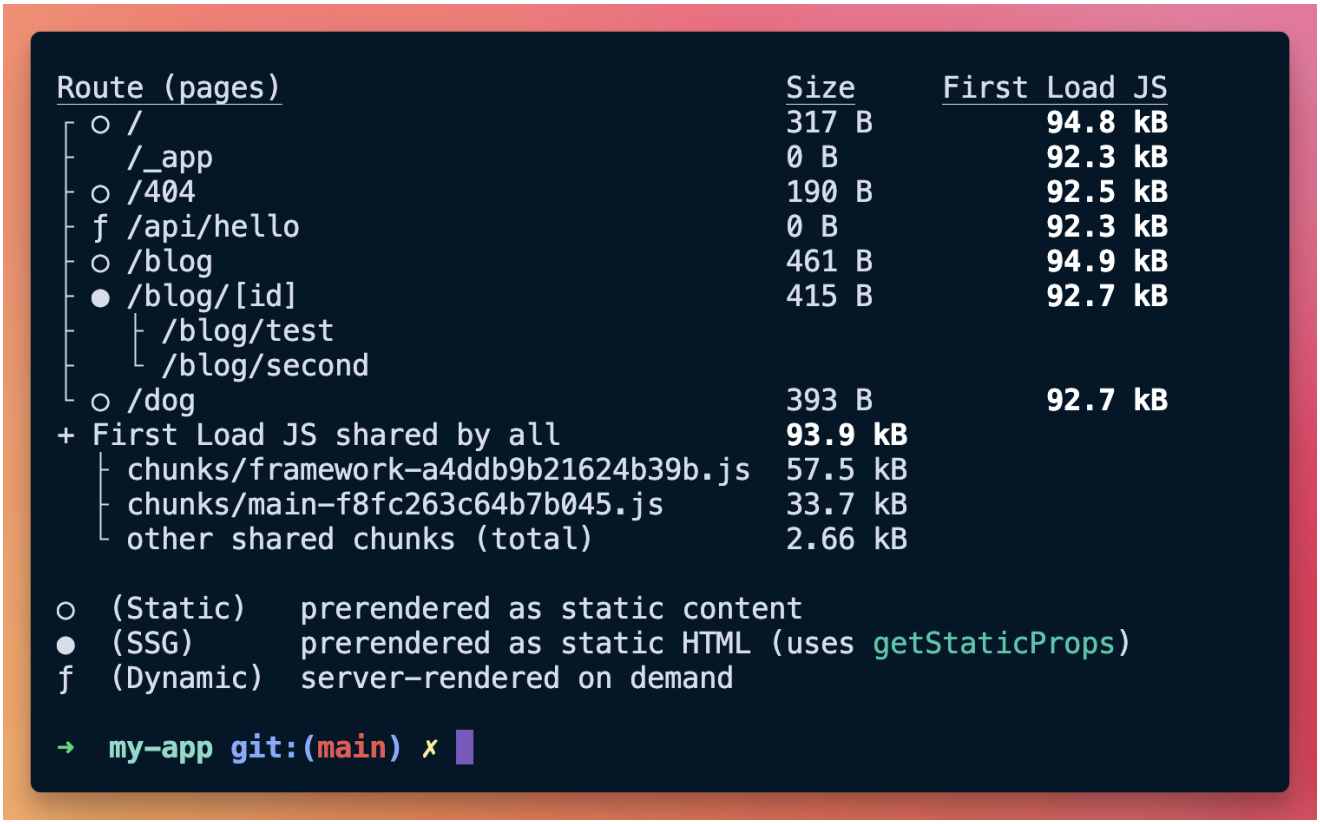
```
      <p>{post.content}</p>
    </>
  )
}

export const getStaticPaths = async () => {
  return {
    paths: Object.keys(posts).map((id) => ({ params: { id } })),
    fallback: false,
  }
}

**export const getStaticProps: GetStaticProps<BlogPostProps> = async (
  context
) => {
  const { id } = context.params as { id: string }

  return {
    props: {
      post: postsData[id],
    },
  }
}**
```

Now the page component receives the `post` parameter as its prop, and it does not need to load a router and do any kind of client-side data lookup, because Next.js does this at build time:

```
import { useRouter } from 'next/router'
import posts from '@/posts.json'
import type { Posts, Post } from '@/types/posts'
import type { GetStaticProps } from 'next'

const postsData = posts as Posts

interface BlogPostProps {
  post: Post
}

**export default function BlogPost({ post }: BlogPostProps) {**
  const router = useRouter()

  if (router.isFallback || !router.query.id) {
    return <div>Loading...</div>
  }

  if (!post) {
```

```
    return <div>Post not found</div>
  }

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

export const getStaticPaths = async () => {
  return {
    paths: Object.keys(posts).map((id) => ({ params: { id } })),
    fallback: false,
  }
}

export const getStaticProps: GetStaticProps<BlogPostProps> = async (
  context
) => {
  const { id } = context.params as { id: string }

  return {
    props: {
      post: postsData[id],
    },
  }
}
```

The way you can see the pages of the posts are now statically rendered is by looking at the output of `npm run build` which builds the production version of the Next.js website, the white circle near `/blog/[id]` means SSG (statically rendered = prerendered as static HTML):



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 using the Pages Router.

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.ts` file, whose goal is to return the comments of a blog post as JSON.

Create a `comments.json` file in the `src` folder, like we did for `posts.json` previously. This file will store a list of comments:

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

```
  }
]
```

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

```
import { NextApiRequest, NextApiResponse } from 'next'
import comments from '@/comments.json'

export default function handler(req: NextApiRequest, res:
NextApiResponse) {
  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].tsx` which will retrieve the comments specific to a post id.

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

```
import { NextApiRequest, NextApiResponse } from 'next'
import comments from '@/comments.json'

export default function handler(req: NextApiRequest, res:
NextApiResponse) {
  res.status(200).json({ post: req.query.id, comments })
}
```

Heres you can see the above code in action:

Remember how 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 API routes it's easier to do that, as the query is **attached to the request object:**

```
export default function handler(req: NextApiRequest, res:
NextApiResponse) {
  console.log(req.query.id)
  res.end()
}
```

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 function handler(req: NextApiRequest, res:
NextApiResponse) {
  switch (req.method) {
    case 'GET':
      //...handle the GET request here
      break
    case 'POST':
      //...handle the POST request here
      break
    default:
            //no other method is allowed,
            //so we return a ["405 Method Not Allowed"]
(https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/405) error
      res.status(405).end()
      break
```

```
  }
}
```

In addition to `req.query` and `req.method` we already saw, we can have access to the request **body** in `req.body`.

Deploying on Vercel

Let's now deploy our Next.js website to the Internet using Vercel.

Vercel makes Next.js. So they optimize their hosting for serving Next.js websites. While you can host Next.js elsewhere, including Netlify or self hosting on a VPS, I think it makes perfect sense to use Vercel in our case.

They have a generous free plan, too, so it's worth taking a look.

First create an account on vercel.com.

Once you're in, you can import any repository you have hosted on GitHub, or you can use the Vercel CLI.

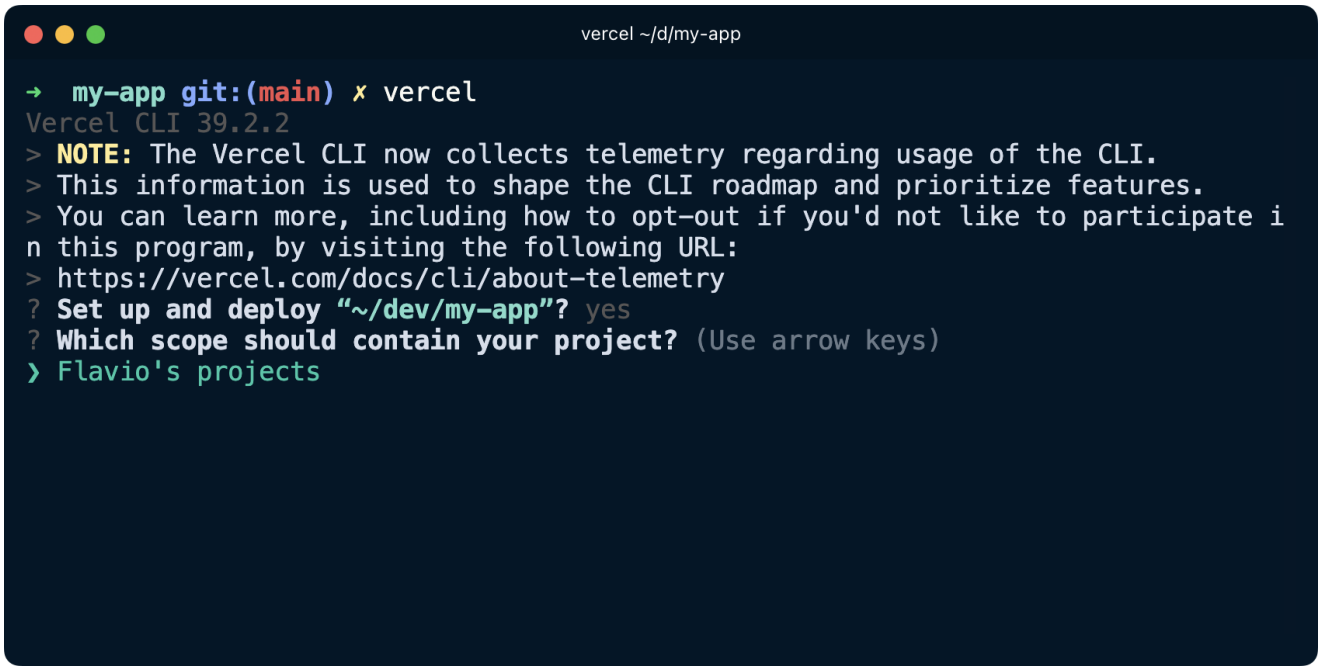
Install the CLI using

```
npm i -g vercel
```

Then from the website root folder, run

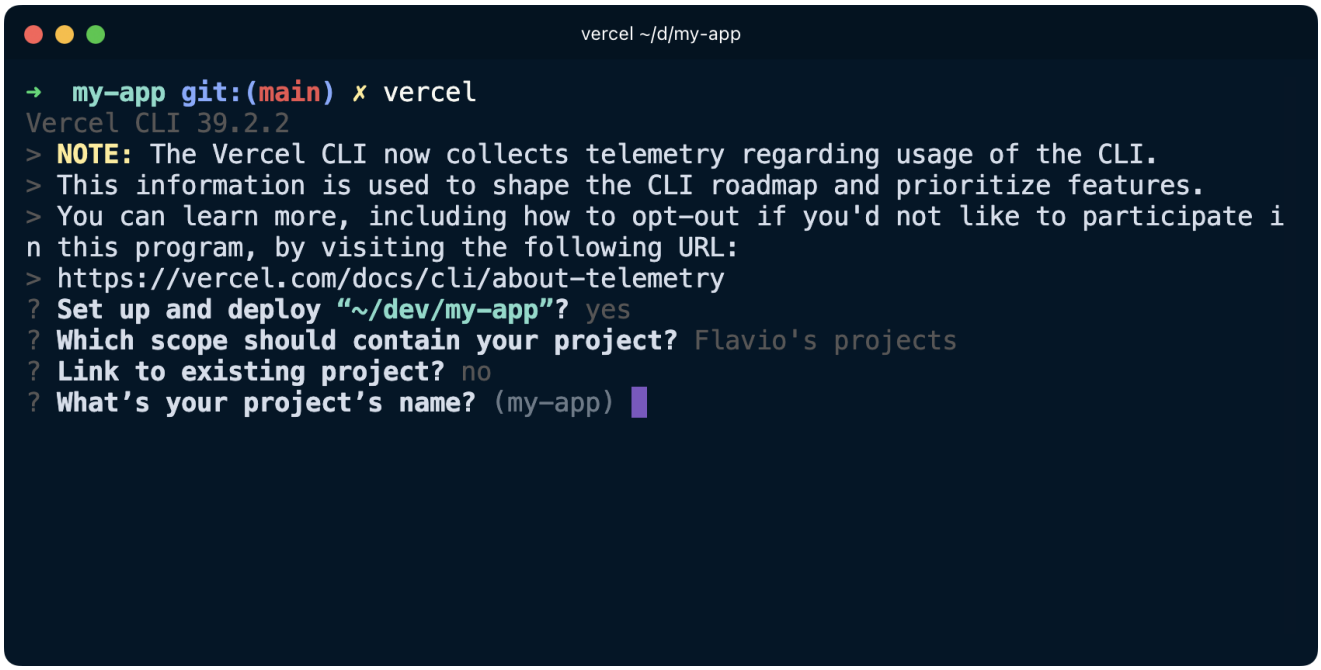
```
vercel
```

Answer `Y` to the question, then press enter to use your default account:



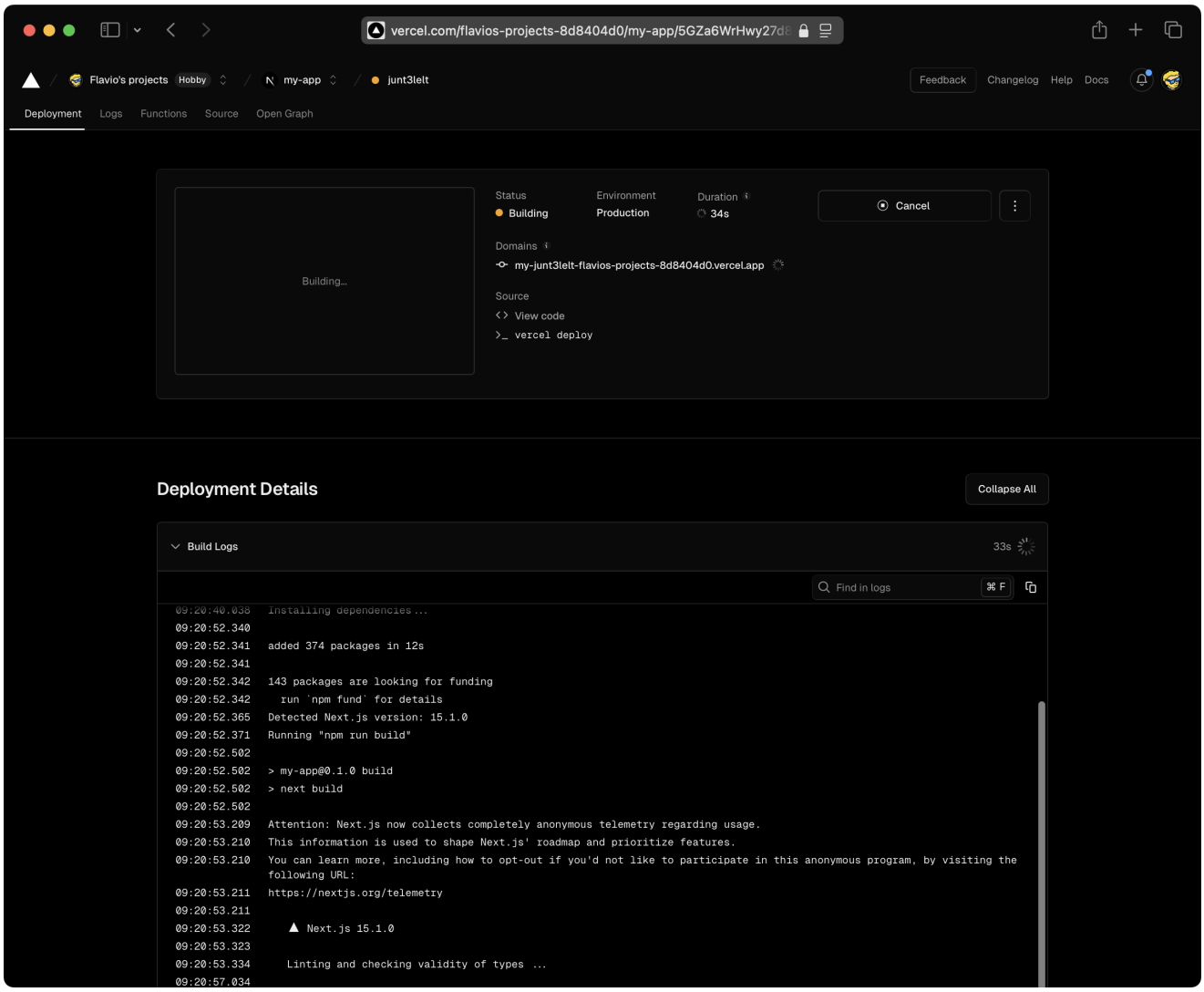
We want to create a new project, so press `N` here:

Now set the project name:

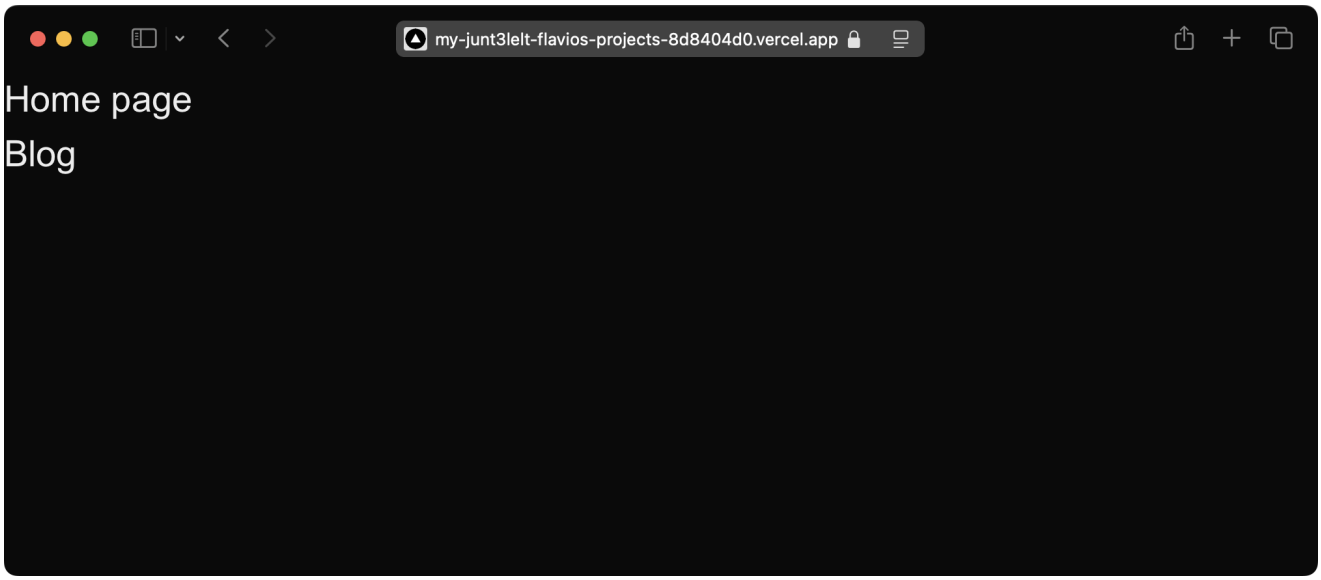


And choose to deploy the project in this folder:

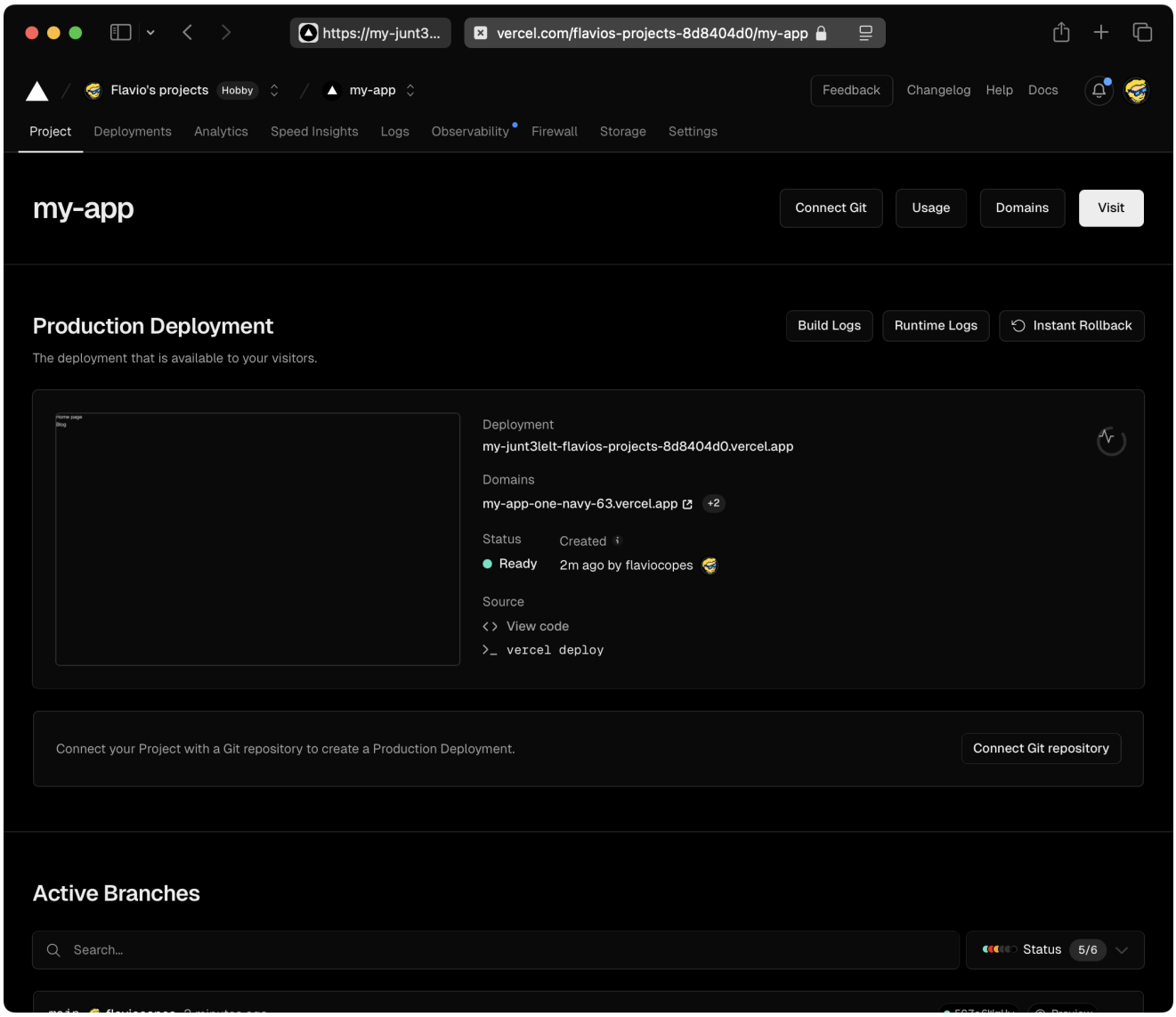
The CLI will do its job, and deploy the site on Vercel, giving you a way to inspect the deployment process:



Finally, it will give you the site URL on a `.vercel.app` subdomain (which will be the app's URL until you set up a custom domain), from where you'll see your site:



Now your site is also visible from the Vercel dashboard:



To push an update, run the `vercel` command again, since you set up the site before, this time the build will immediately start: