Build a Blog Using Next.js

Diana MacDonald

Build a Blog Using Next.js

Diana MacDonald

Queensland, Australia

https://didoesdigital.com/project/nextjs-blog-book/

Published by Diana MacDonald.

For enquires, please e-mail nextjs@didoesdigital.com

The companion repository of source code for this book is available to readers on GitHub via the book's product page, located at https://didoesdigital.com/project/nextjs-blog-book/

Table of Contents

Introduction

This book describes how to build a Next.js blog in 2024 using all the latest practices, and a static export. It includes using local MDX files that let you write posts with Markdown, JSX, and rich content. I've included considerations for tech choices along the way. There are caveats before getting started and some editor setup guidance.

There is an accompanying Git repository with the code for the blog on GitHub: https://github.com/didoesdigital/nextjs-blog. Refer to this if you get stuck.

If you find this book useful, please consider supporting my work by paying for the book on Ko-fi: https://ko-fi.com/didoesdigital.

Here are the main technologies used in the blog:

- Next.js v14 with App Router
- A static export
- TypeScript v5
- Tailwind CSS v3
- MDX v3 via `@next/mdx` v14
- RSS via the `rss` package
- Eslint
- Prettier
- Tailwind Prettier plugin
- Google Analytics

We'll also cover setting up dark mode, favicons, Open Graph images, syntax highlighting, styling MDX content, linked headings, a sitemap, `robots.txt`, and static sites for production. Static

sites don't require an app server. I'll assume you have some familiarity with React and web fundamentals.

There are better choices than Next.js for personal blogs

⚠️ Before we get started, a word of warning so you know what you're getting into.

While this book demonstrates how you might build a personal blog using Next.js, it's just an example of a *something* you could build with Next.js to get the hang of it and even take advantage of static exports.

I had specific motivations to build my site and blog in Next.js using React Server Components with lots of customisation, but if I were looking for a standard, statically generated site just for a blog, I'd consider Eleventy or Astro or Zola or something else.

While Vercel provide Next.js templates for blogs and portfolios, they're not as full featured as you might hope. Other static site generators have more mature ecosystems and plugins for blogs to get you up and running faster.

Next.js lets you build rich, interactive experiences using React. If you need that interactivity and the convenience of a familiar front-end framework, it

can be a great choice. If you just want a quick, maintainable blog, you might want to consider an alternative static site generator option.

Templates

We're going to use the Create Next App Command Line Interface (CLI) to kick us off but we're not going to use a template. Given how much you'd need to build or rebuild to make a full blog using the latest features, the templates don't offer much.

If you're curious about some reasons why we're not using a template:

- The templates aren't as up-to-date with the latest Next.js features as the Create Next App CLI starter.

- The blog starter kit template includes Markdown, but not MDX, which means without further work it is tedious to include rich content such as social media post embeds, videos, syntax highlighted code blocks, or IDs and links on headings.
- The blog starter template uses a custom `markdownToHtml` pipeline using `remark-html` instead of Next.js's recommended `@next/mdx` package and config. Similarly, it uses the `gray-matter` package to parse front matter instead of Next.js's built-in `metadata` object. These approaches have their trade-offs but seem to grate against the built-in Next.js features. Other templates parse YAML-style front-matter *using regex*, which is wild.
- The blog starter template doesn't include a sitemap or RSS feed, or support for categories, tags, or search.

- The blog starter template has the occasional stray artifact from older versions of Next.js, such as the `as` attribute on `Link` components, which is no longer required, and duplicate `public/` directories. There are also immediate `metabase` console warnings.
- Other templates, such as the Portfolio Blog Starter, use the `next-mdx-remote` package for *remote* MDX—which is unnecessary if you want to keep your MDX posts in your git repo—and don't use the `@next/mdx` package recommended by Next.js.
- The blog templates don't use a static export and they don't include useful defaults like `next.config.mjs` or `.eslintrc.json` files.

Conventions in this book

For code blocks, filenames are included in the preceding paragraph.

Where a file that has already been introduced includes new changes, I only repeat the lines of code that are necessary for context. Otherwise, I use comments like `//` or `{/* */}` to indicate that existing code has been omitted.

Long lines in code blocks in the book's PDFs wrap onto the next line using a hard line break. If you copy and paste code from the PDF, you may need to join those broken lines back together. You can reference any linked documentation or the code in the repository if you're having trouble.

Troubleshooting

Be precise with directories and files

Next.js relies on directory and filename conventions. This is convenient for producing desired behaviour by default, but can lead to confusing errors in some situations. Keep in mind:

- `app/` and `pages/` directories are special, so avoid directories with those names except in the exact places where Next.js expects them for routing, which is discussed in the first chapter.
- There are special filenames that have specific behaviour, such as `page.tsx`. They are documented in Next.js's file conventions. Only use those names as intended by Next.js.
- Avoid whitespace in paths and filenames. Be mindful when copying and pasting code from the book or the repository.
- Avoid extraneous empty directories.

Restart the dev server

For a lot of changes you make, the dev server will auto-reload. Occasionally, however, some changes won't be picked up, such as changes to config files or packages. If something you've changed isn't applying, try restarting the server.

Check the terminal for logs

If you're used to checking the browser for the result of `console.log` and similar statements, remember to check the terminal instead.

Import aliases using @

If you see errors related to resolving modules containing an `@` symbol, it may be related to path aliases. Review your `tsconfig.json` file and your file structure.

Expected paths

There are some parts of the system that expect files in certain places. If you move files or directories around, you may need to review configuration paths, such as the `content` array in the Tailwind config file, `paths` in the `tsconfig.json` file, or scripts in the `package.json`. Until you're comfortable with how each part works, I recommend sticking with the paths and filenames I've used in the book.

Next.js cache

Next.js is incredibly clever at efficiently rebuilding your site when you make changes, but sometimes it can get into a weird state. On rare occasions, you may try removing the `.next/cache` directory and restarting the server.

Chapter 1: Set Up a Next.js Site and VS Code

Getting started

Create a new project directory using the recommended `create-next-app` automatic installation CLI. By supplying no arguments, it will kick off an interactive setup process. If `npx` prompts you to install packages, type y and Return. We'll choose `Yes` for TypeScript, ESLint, Tailwind CSS, `src/` directory, and `App Router`. We won't customise the default import alias. To accept the

default `create-next-app` options, type Return for each prompt. Otherwise, use arrow keys to switch between them.

```
$ npx create-next-app@14.2.3
✔ What is your project named? …
next14-static-export-mdx-blog
✔ Would you like to use TypeScript? …
No / Yes
✔ Would you like to use ESLint? …
No / Yes
✔ Would you like to use Tailwind CSS?
… No / Yes
✔ Would you like to use src/
directory? … No / Yes
✔ Would you like to use App Router?
(recommended) … No / Yes
✔ Would you like to customize the
default import alias (@/)? … No / Yes
```

Change directory into your new project:

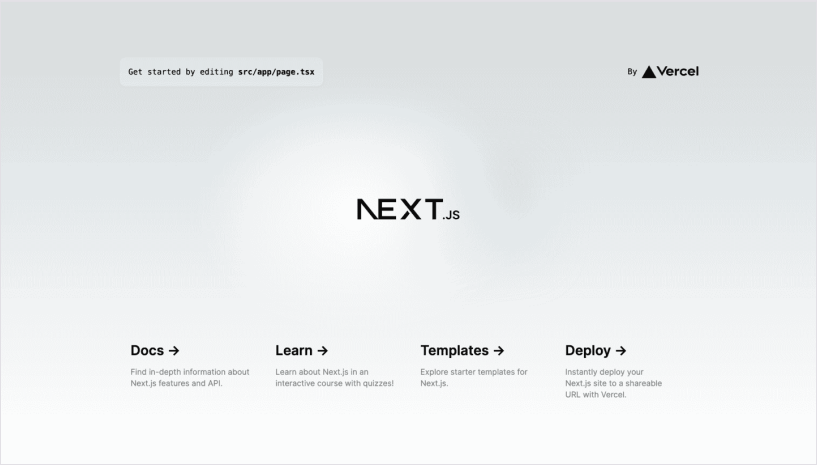
```
cd next14-static-export-mdx-blog
```

Run the blog

Run the `dev` script from `package.json` and visit `http://localhost:3000/`:

```
npm run dev
```

Here's how it should look:



The site at http://localhost:3000/

In dark mode, the text and background colours will be inverted.

You can now stop the server with ctrl + c. You can resume `npm run dev` after we finish setting up.

Inspect the source code

Here's how most of our folders and files look:

```
README.md
next-env.d.ts
next.config.mjs
node_modules/
package-lock.json
package.json
postcss.config.mjs
public/
└── next.svg
└── vercel.svg
src/
└── app/
    └── favicon.ico
    └── globals.css
    └── layout.tsx
    └── page.tsx
tailwind.config.ts
tsconfig.json
```

App Router vs Page Router

Use App Router. From the Next.js *Page* Router docs:

> Before Next.js 13, the Pages Router was the main way to create routes in Next.js. It used an intuitive file-system router to map each file to a route. The Pages Router is still supported in newer versions of Next.js, but we recommend migrating to the new App Router to leverage React's latest features.

While some of the newer React features available with App Router aren't needed for a static blog, App Router is the default in the latest versions of the Next.js CLI. You can use a static blog built with App Router as a stepping stone to towards more dynamic and interactive sites.

When searching for or within Next.js documentation, make sure you're looking at the modern "App Router" version of the docs. There is an on-page toggle to switch between "App Router" and the older "Page Router". There are also tabs for "App Router" and "Page Router" results in the Next.js documentation search command menu. Outside of the Next.js docs, there are lots of links to the older "Page Router" so keep an eye out for changes to the URL and on-page toggle.

Editor config

TypeScript

According to the docs on Next.js TypeScript Plugin:

> Next.js includes a custom TypeScript plugin and type checker, which VSCode and other code editors can use for advanced type-checking and auto-completion.

If you're using VS Code, open a Typescript (`.tsx`) file and follow Next.js's instructions:

> You can enable the plugin in VS Code by:
>
> 1. Opening the command palette (Ctrl/⌘ + Shift + P)
> 2. Searching for “TypeScript: Select TypeScript Version”
> 3. Selecting “Use Workspace Version”

That should add the following line to a `.vscode/settings.json` file containing the following:

```
{
  "typescript.tsdk": "node_modules/typescript/lib"
}
```

You may or may not want to add this workspace settings file to your `.gitignore` file. If you want to enforce shared VS Code settings with others, you can Git commit the `.vscode/settings.json` file and any changes to it that should be shared.

Tailwind

Check out the docs on Tailwind editor setup.

If you open the `globals.css` file and see a warning like `Unknown at rule @tailwindcss(unknownAtRules)`, you can set the language mode to `tailwindcss`. If you always want to treat the `globals.css` file or all `*.css` files as Tailwind CSS, you can configure one of these file associations in your `settings.json`:

```
{
  "typescript.tsdk": "node_modules/
typescript/lib",
  "files.associations": {
    "*.css": "tailwindcss",
    "globals.css": "tailwindcss"
  }
}
```

If you also use the Git Lens extension and it's too noisy, you can add:

```
{
  "typescript.tsdk": "node_modules/
typescript/lib",
  "[tailwindcss]": {
    "gitlens.codeLens.scopes":
["document"]
  },
  "files.associations": {
    "*.css": "tailwindcss"
  }
}
```

Linting and formatting

Eslint

We'll check if Eslint is good to go. For details, see the Next.js eslint docs.

There should be a `lint` script in the `package.json` for running `next lint`:

```
{
  "scripts": {
    "lint": "next lint"
  }
}
```

This script uses the `eslint` and `eslint-config-next` dependencies:

```
{
  "devDependencies": {
    "eslint": "^8",
    "eslint-config-next": "14.2.2"
  }
}
```

There should be an `.eslintrc.json` file in the root of the project to use the Next.js core web vitals config. While you can specify the one `"next/core-web-vitals"` configuration file here in `extends` as a string, we will use an array so that it's easy to add more configurations later:

```
{
  "extends": ["next/core-web-vitals"]
}
```

Now run `npm run lint`. You should see no warnings or errors:

```
$ npm run lint

> next14-static-export-mdx-blog@0.1.0
lint
> next lint
✔ No ESLint warnings or errors
```

Prettier

There are useful Prettier docs and Next.js Prettier docs.

Install the `prettier` package and commit the changes to `package.json` and `package-lock.json`:

```
npm install --save-dev --save-exact
prettier
```

Create an empty config file called `.prettierrc` in the root of the project to let your editor and other tools know you're using Prettier:

```
{}
```

Add a `.prettierignore` file so that the Prettier CLI and editors don't format files they shouldn't. Note, Prettier will already ignore items listed in your `.gitignore`, such as `/.next/` and `next-env.d.ts`:

```
public/
package-lock.json
```

Because we're using Eslint and Prettier, install the `eslint-config-prettier` package so that Eslint and Prettier play nicely together:

```
npm install --save-dev eslint-config-
prettier
```

Commit the changes.

Update the `.eslintrc.json` file:

```
{
  "extends": ["next/core-web-vitals",
"prettier"]
}
```

You can use your editor to run prettier manually on each file using a command like “Format Document” or on file save using a setting like `"editor.formatOnSave": true,`.

Otherwise, you can add a script to `package.json`, e.g.:

```
{
  "scripts": {
    "format": "npx prettier --write 'src/**/*.{js,ts,jsx,tsx,mdx}' --log-level 'warn'"
  }
}
```

Run `npm run format` to format all the files to check that it works and commit your changes.

Tailwind and Prettier

If you'd like your Tailwind classes sorted, check out the Tailwind Prettier plugin docs.

Install the plugin (and `prettier` if you haven't already):

```
npm install -D prettier-plugin-tailwindcss
```

Update `.prettierrc` to use the new plugin:

```
{
  "plugins": ["prettier-plugin-tailwindcss"]
}
```

Run `npm run format` to format all the files. This should move around a lot of Tailwind classes so be sure to commit those changes.

Static exports

For a simple deployment of static HTML/CSS/JavaScript files, we can use Next.js's support for static exports. Skim the docs to see which Next.js features are supported and which are not. Using a static export means you don't need to run a node server, which can simplify deployment and some security considerations. One drawback, however, is that Next.js docs don't always cover how you need to adjust your code to work with static exports.

Update the `next.config.mjs` file in the root of the project to include `output: 'export'`:

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

export default nextConfig;
```

To run the `next build` command, use the `package.json` script:

```
npm run build
```

This will create a new `out/` directory containing the static files.

To see the static files in action, run a local server:

```
npx http-server ./out/
```

Press `y` if prompted to install the `http-server` package.

You should then see exactly the same blog when you visit `http://localhost:8080/`.

Keep checking that `npm run build` works after every new feature you add.

Chapter 2: MDX and Metadata

Add MDX

MDX lets you use JSX in your Markdown content. We can follow the Next.js MDX docs to add MDX to our blog using `@next/mdx`.

> You might see mentions around the internet of `next-mdx-remote` or other MDX packages. Next.js specifically recommends `@next/mdx`. Alex Chan mentions a few reasons why you might avoid `next-mdx-remote`.

Install the necessary dependencies and types:

```
npm install @next/mdx @mdx-js/loader
@mdx-js/react
npm install --save-dev @types/mdx
```

Commit dependency changes.

We need to configure the `next.config.mjs` file to use the MDX loader and allow MDX pages.

```
import createMDX from "@next/mdx";

/**
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  output: "export",
  pageExtensions: ["js", "jsx", "md",
"mdx", "ts", "tsx"],
};

const withMDX = createMDX({
  // Add Markdown plugins here, as
desired
});

export default withMDX(nextConfig);
```

Note: You could also replace `createMDX` with `nextMDX` if that's clearer to you.

We also need to make the **required** `src/mdx-components.tsx` file:

```
import type { MDXComponents } from
"mdx/types";

export function
useMDXComponents(components:
MDXComponents): MDXComponents {
  return {
    ...components,
  };
}
```

If you don't add that file, you might get a mysterious error like:

```
Error: createContext only works in Client
Components. Add the "use client"
directive at the top of the file to use
it. Read more: https://nextjs.org/docs/
messages/context-in-server-component
```

It isn't terribly obvious from the message that the answer is to add the missing `mdx-components.tsx` next to the `app/` directory. Since our `app/` directory is in `src/app/`, we need our file to be at `src/mdx-components.tsx`.

The App Router lets us define routes using directories and files in the directory called `app/`, which we have inside `src/`. From routes inside `src/app/`, we will import our blog post MDX files, which we can store wherever we like. We'll store the MDX content in its own directory, `src/blog/`, outside of `app/`.

Create your first MDX file in a `src/blog/` directory e.g. `src/blog/first-mdx-post.mdx`:

```
# First MDX post

Hello world!
```

Scaffold a new blog page component in `src/app/blog/[slug]/page.tsx` to show the MDX blog post. In this path, `[slug]` is a Dynamic Segment because it is wrapped in square brackets. This blog `page.tsx` will need a `BlogPage` component to render the page using the dynamic segment's slug from `params`:

```
type BlogPageProps = { params: { slug:
string } };

export default function
BlogPage({ params }: BlogPageProps) {
  return (
    <div className="container mx-auto
p-4">
      <h2 className="text-xl">{params.
slug}</h2>
    </div>
  );
}
```

And it will need a `generateStaticParams` function to statically generate routes for the blog posts:

```
export async function
generateStaticParams() {
  const blogPosts = ["first-mdx-
post"]; // FIXME: Read from file
system
  const blogStaticParams =
blogPosts.map((post) => ({
    slug: post,
  }));

  return blogStaticParams;
}
```

At this point, your file structure within `src/` should look like this:

```
src/
├── app/
│   ├── blog/
│   │   └── [slug]/
│   │       └── page.tsx
│   ├── favicon.ico
│   ├── globals.css
│   ├── layout.tsx
│   └── page.tsx
├── blog/
│   └── first-mdx-post.mdx
└── mdx-components.tsx
```

Now when you run `npm run dev` and visit `http://localhost:3000/blog/first-mdx-post`, you should see the slug name on the page: `first-mdx-post`. This means the route is working. We'll come back to that `FIXME` in another chapter.

It's automatically using the root `layout.tsx`, which means we can see an awful repeating gradient background. Let's fix that in `globals.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;

:root {
  --foreground-rgb: 0, 0, 0;
  --background-rgb: 255, 255, 255;
}

@media (prefers-color-scheme: dark) {
  :root {
    --foreground-rgb: 255, 255, 255;
  }
}

body {
  color: rgb(var(--foreground-rgb));
  background: rgb(var(--background-
rgb));
}
```

We want to show the actual MDX post content though. Let's update `src/app/blog/[slug]/page.tsx` to dynamically import the MDX content and render it:

```
import dynamic from "next/dynamic";

//

export default function
BlogPage({ params }: BlogPageProps) {
  const BlogMarkdown = dynamic(() =>
import("@/blog/" + params.slug +
".mdx"));

  return (
    <div className="container mx-auto
p-4">
      <h2 className="text-xl">{params.
slug}</h2>
      <BlogMarkdown />
    </div>
  );
}
```

If you see an error like `Module not found: Can't resolve '@/blog/'`, you might need to review your `tsconfig.json` file and your file structure. Next.js has built-in support for path aliases, so when you set a `baseUrl` like `"baseUrl": "."` and a `paths` entry like `"paths": { "@/*": ["./src/*"] }`, TypeScript can find files you reference using the `@/` alias.

Now when you run `npm run dev` and visit `http://localhost:3000/blog/first-mdx-post`, you should see the slug name on the page as well as the blog content:

The first MDX post at http://localhost:3000/blog/first-mdx-post

Remember to keep checking that `npm run build` still works as expected.

MDX metadata

We can use Next.js's static metadata object to specify the title and description of the MDX post.

The metadata object sets the page's title and meta description in the HTML head.

We'll also reuse that metadata title in the blog post component and the blog listing index page.

First, update the `src/blog/first-mdx-post.mdx` file to include a metadata export:

```
export const metadata = {
  title: "My first MDX blog post",
  description: "A short MDX blog
post.",
};

# First MDX post

Hello world!
```

Let's create some reusable functionality to import a post's metadata.

We'll create a new directory using Next.js's colocation and private folder organisation features, `src/app/blog/_lib/`.

Create a new file inside the new directory: `src/app/blog/_lib/getBlogPostData.ts`. For types, we'll need `PostMetadata` and `BlogPostData`:

```
import type { Metadata } from "next/
types";

export type PostMetadata = Metadata &
{
  title: string;
  description: string;
};

export type BlogPostData = {
  slug: string;
  metadata: Metadata;
};
```

We'll also create a function to import the metadata from the MDX file:

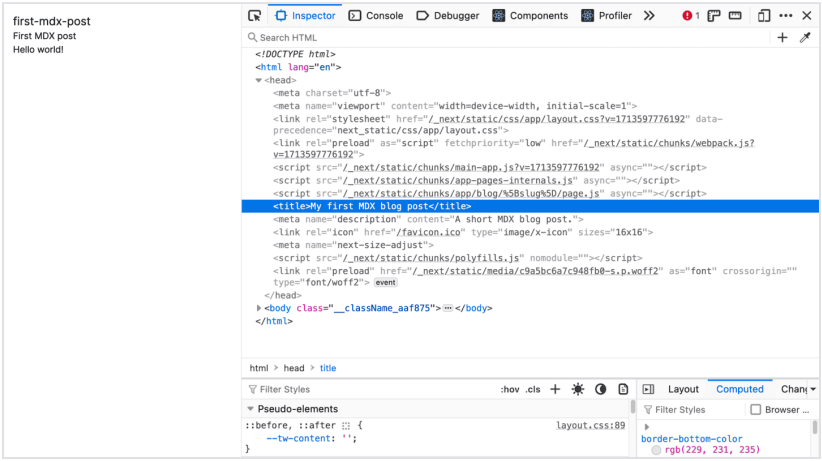
```
import { notFound } from "next/
navigation";

export async function
getBlogPostMetadata(slug: string):
Promise<BlogPostData> {
  try {
    const file = await import("@/
blog/" + slug + ".mdx");

    if (file?.metadata) {
      if (!file.metadata.title || !
file.metadata.description) {
        throw new Error(`Missing some
required metadata fields in: ${slug}
`);
      }

      return {
        slug,
        metadata: file.metadata,
      };
    } else {
      throw new Error(`Unable to find
metadata for ${slug}.mdx`);
    }
```

```
  } catch (error: any) {
    console.error(error?.message);
    return notFound();
  }
}
```

Update `src/app/blog/[slug]/page.tsx` to generate dynamic metadata from the MDX file:

```
import { getBlogPostMetadata } from
"@/app/blog/_lib/getBlogPostData";
import type { Metadata } from "next/
types";
//

export async function
generateMetadata({
  params,
}: BlogPageProps): Promise<Metadata> {
  const { metadata } = await
getBlogPostMetadata(params.slug);

  if (metadata) {
    return metadata;
  } else {
    throw new
Error(`No metadata found for blog
post: ${params.slug}`);
  }
}
```

Now when you run `npm run dev`, you should see your title and description appear in the head of the page:

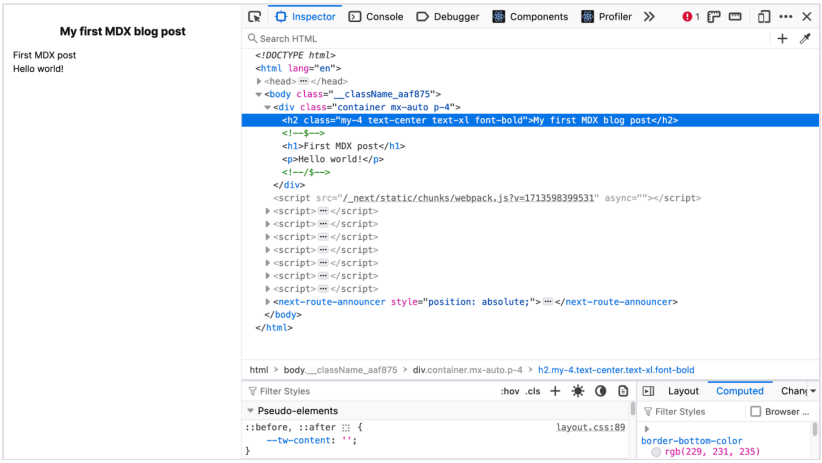


The <title> tag and <meta name="description"> tag show the data from the MDX metadata object

We can use that metadata title as the heading of the blog post. Update `src/app/blog/[slug]/page.tsx`:

```
export default async function
BlogPage({ params }: BlogPageProps) {
  const { metadata } = await
getBlogPostMetadata(params.slug);
  const title = `${metadata.title ??
""}`;

  const BlogMarkdown = dynamic(() =>
import("@/blog/" + params.slug +
".mdx"));

  return (
    <div className="container mx-auto
p-4">
      <h2 className="my-4 text-center
text-xl font-bold">{title}</h2>
      <BlogMarkdown />
    </div>
  );
}
```

We use Next.js's dynamic import to lazy load the metadata. We've made the function `async` so that we can `await` the metadata. We've used the metadata title in a styled heading and removed the slug from the page.



The title now appears as a styled heading on the blog post

Does `npm run build` still work? Keep checking!

Chapter 3: Site Styling

Fonts

Let's add some fonts. Check out the Next.js fonts docs.

We'll use 1 non-variable serif font for the main body text, 1 sans-serif font for buttons, and 1 monospace font for code blocks.

First, import the Google fonts and export font `const`s in a `src/app/_components/fonts/fonts.ts` file:

```
import { Crimson_Text, Overpass_Mono,
Work_Sans } from "next/font/google";

export const crimson = Crimson_Text({
  weight: ["400", "600", "700"],
  style: ["normal", "italic"],
  subsets: ["latin"],
  display: "swap",
  variable: "--font-crimson",
});

export const workSans = Work_Sans({
  // Variable font so we don't set
weight: [],
  style: ["normal"],
  subsets: ["latin"],
  display: "swap",
  variable: "--font-work-sans",
});

export const overpassMono =
Overpass_Mono({
  // Variable font so we don't set
weight: [],
  style: ["normal"],
  subsets: ["latin"],
```

```
  display: "swap",
  variable: "--font-overpass-mono",
});
```

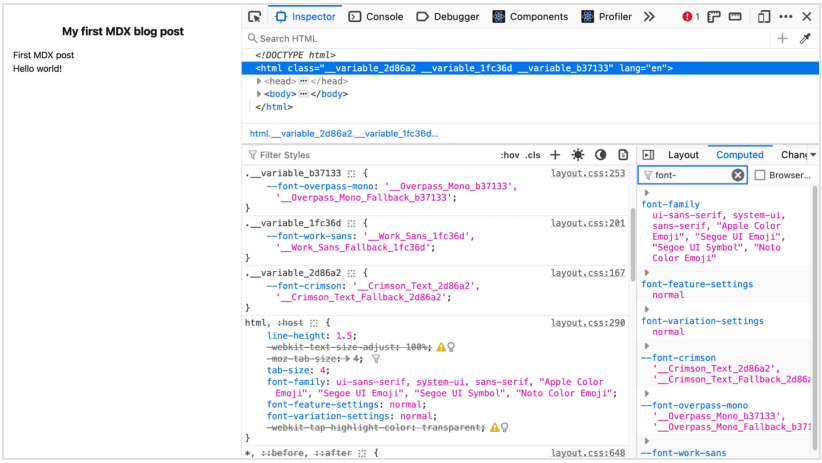
We've used the `variable` option to set CSS custom properties for each font. This will let us use the fonts with our Tailwind CSS and, optionally, in the `globals.css` file.

Now we'll use those variables in `src/app/layout.tsx`:

```
import { crimson, workSans,
overpassMono } from "@/app/
_components/fonts/fonts";

// …

<html
  lang="en"
  className={`${crimson.variable} ${wo
rkSans.variable} ${overpassMono.variab
le}`}
>
  <body>{children}</body>
</html>
```

Be sure to remove the references to Inter, including the variable on the `body` tag.

We should now see Tailwind ugly class names on the `html` tag that set the font CSS properties to unique font names for the Google fonts:



Our 3 custom font family names are available as CSS variables

Next, we'll update our `tailwind.config.ts` file to use those font variables.

You can add the `fontFamily` property directly inside the `theme` object to *replace* Tailwind's font family settings. For example:

```
const config: Config = {
  theme: {
    fontFamily: {
      sans: [
        "var(--font-work-sans)",
        "ui-sans-serif",
        "system-ui",
        "Apple Color Emoji",
        "Segoe UI Emoji",
        "Segoe UI Symbol",
        "Noto Color Emoji",
        "sans-serif",
      ],
      serif: [
        "var(--font-crimson)",
        "ui-serif",
        "Georgia",
        "Cambria",
        "Times New Roman",
        "Times",
        "serif",
      ],
      mono: [
        "var(--font-overpass-mono)",
        "ui-monospace",
        "SFMono-Regular",
```

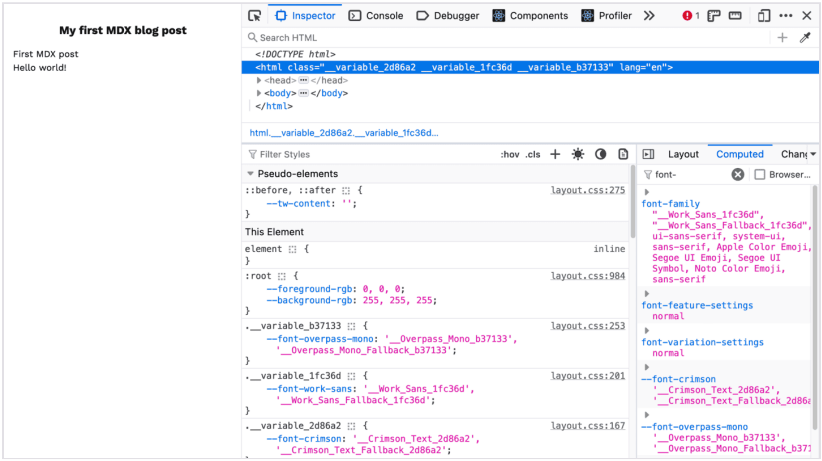
```
        "Menlo",
        "Monaco",
        "Consolas",
        "Liberation Mono",
        "Courier New",
        "monospace",
      ],
    },
    //
  },
  //
};
```

The other option would be to add `fontFamily` to an `extend` object inside `theme` to *extend* Tailwind's settings:

```
// tailwind.config.ts
const config: Config = {
  theme: {
    extend: {
      fontFamily: {
        serif: [
          "var(--font-crimson)",
          "ui-serif",
          "Georgia",
          "Cambria",
          "Times New Roman",
          "Times",
          "serif",
        ],
        //
      },
      //
    },
  },
  //
};
```

We'll stick with the first option. Apart from our custom font CSS variables, the rest of the font family names are the same as the default Tailwind settings.

Now, your CSS should look like this:



The HTML element now has a font-family applied including the unique custom font family names for Work Sans shown in the Computed styles panel

Our fonts are now *available* to use and Tailwind has automatically applied the sans-serif font to the `html` element.

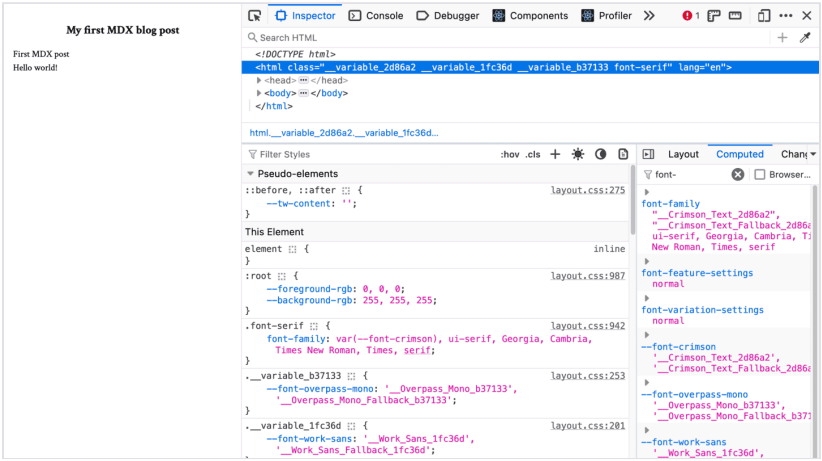
Typography

The Tailwind docs on adding base styles suggest adding global colors and font styles like this:

```
<html lang="en" class="text-gray-900
bg-gray-100 font-serif"></html>
```

We can add the `font-serif` class to the `html` or `body` tag in the `layout.tsx` file so that our body text uses Crimson text:

```
<html
  lang="en"
  className={`${crimson.variable} $
{workSans.variable} $
{overpassMono.variable} font-serif`}
>
  <body>{children}</body>
</html>
```



The .font-serif class uses var(--font-crimson) to apply the unique font family name for Crimson Text

When we want to apply a sans serif font to a button, we can add the `font-sans` class to the button. When we want to apply the monospace font to a code block, we can add the `font-mono` class to the code block. We can also use our CSS variables directly in CSS.

Colors

To configure our own brand colors, we *could* replace the Tailwind colors directly with our colors in the `tailwind.config.ts` file like this:

```
theme: {
  colors: {
    black: "#ffffff",
    blue: {
      "50": "#f0f9ff",
      "100": "#e1f3fd",
      "200": "#bbe8fc",
      "300": "#80d6f9",
      "400": "#3cc1f4",
      "500": "#13a9e4",
      "600": "#0689c3",
      "700": "#066d9e",
      "800": "#0a5c82",
      "900": "#0e4c6c",
      "950": "#0b3954",
    }
  }
}
```

But instead, we'll use CSS variables. This makes it possible to easily access the colors in our CSS and TSX files. It could also enable theme switching.

First, update the `globals.css` with color CSS variables for your brand colors (and move these base styles into the `base` layer before `utilities`):

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer base {
  :root {
    --color-blue-50: #f0f9ff;
    --color-blue-100: #e1f3fd;
    --color-blue-200: #bbe8fc;
    --color-blue-300: #80d6f9;
    --color-blue-400: #3cc1f4;
    --color-blue-500: #13a9e4;
    --color-blue-600: #0689c3;
    --color-blue-700: #066d9e;
    --color-blue-800: #0a5c82;
    --color-blue-900: #0e4c6c;
    --color-blue-950: #0b3954;
    --color-teal-50: #eefffc;
    --color-teal-100: #c5fffa;
    --color-teal-200: #8bfff5;
```

```
    --color-teal-300: #4afef0;
    --color-teal-400: #15ece2;
    --color-teal-500: #00d0c9;
    --color-teal-600: #00a8a5;
    --color-teal-700: #008080;
    --color-teal-800: #066769;
    --color-teal-900: #0a5757;
    --color-teal-950: #003235;
    --color-neutral-50: #f6f6f6;
    --color-neutral-100: #e7e7e7;
    --color-neutral-200: #d1d1d1;
    --color-neutral-300: #b0b0b0;
    --color-neutral-400: #888888;
    --color-neutral-500: #737373;
    --color-neutral-600: #5d5d5d;
    --color-neutral-700: #4f4f4f;
    --color-neutral-800: #454545;
    --color-neutral-900: #3d3d3d;
    --color-neutral-950: #262626;

    --foreground-rgb: 0, 0, 0;
    --background-rgb: 255, 255, 255;
  }

  @media (prefers-color-scheme: dark)
{
    :root {
```

```
      --foreground-rgb: 255, 255, 255;
    }
  }

  body {
    color: rgb(var(--foreground-rgb));
    background: rgb(var(--background-
rgb));
  }
}

@layer utilities {
  /* */
}
```

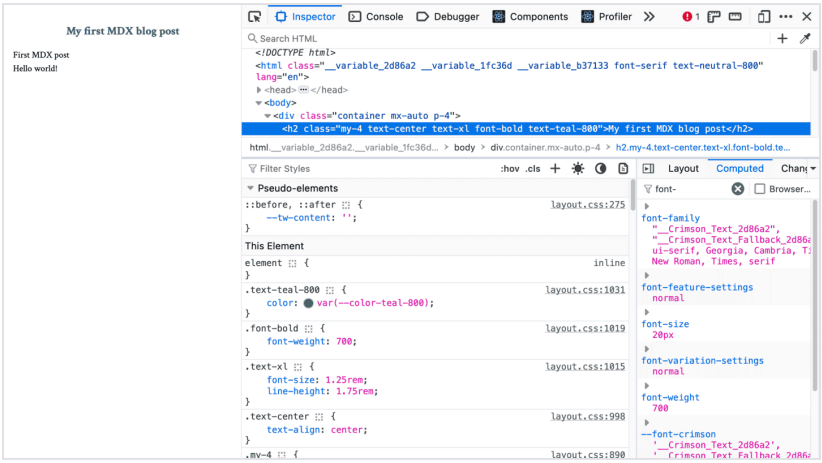
Then in the `tailwind.config.ts` file, add the CSS variables to the `theme` (or `theme.extend`) object:

```
theme: {
  colors: {
    transparent: "transparent",
    current: "currentColor",
    black: "#000000",
```

```
    white: "#ffffff",
    blue: {
      50: "var(--color-blue-50)",
      100: "var(--color-blue-100)",
      200: "var(--color-blue-200)",
      300: "var(--color-blue-300)",
      400: "var(--color-blue-400)",
      500: "var(--color-blue-500)",
      600: "var(--color-blue-600)",
      700: "var(--color-blue-700)",
      800: "var(--color-blue-800)",
      900: "var(--color-blue-900)",
      950: "var(--color-blue-950)",
    },
    teal: {
      50: "var(--color-teal-50)",
      100: "var(--color-teal-100)",
      200: "var(--color-teal-200)",
      300: "var(--color-teal-300)",
      400: "var(--color-teal-400)",
      500: "var(--color-teal-500)",
      600: "var(--color-teal-600)",
      700: "var(--color-teal-700)",
      800: "var(--color-teal-800)",
      900: "var(--color-teal-900)",
      950: "var(--color-teal-950)",
    },
```

```
    neutral: {
      50: "var(--color-neutral-50)",
      100: "var(--color-neutral-100)",
      200: "var(--color-neutral-200)",
      300: "var(--color-neutral-300)",
      400: "var(--color-neutral-400)",
      500: "var(--color-neutral-500)",
      600: "var(--color-neutral-600)",
      700: "var(--color-neutral-700)",
      800: "var(--color-neutral-800)",
      900: "var(--color-neutral-900)",
      950: "var(--color-neutral-950)",
    }
  }
}
```

By adding `text-neutral-800` to the `html` and `text-teal-800` to the blog post heading in our TSX files, we can see some brand colors in action:



A teal heading

Global site styles

If you have truly global styles you wish to apply to HTML elements, use the `base` layer of your global CSS file:

```
/* src/app/globals.css */
@layer base {
  a[href] {
    text-decoration-style: wavy;
  }
}
```

If you add styling there for text elements such as `h1` or `p`, I suggest that you *don't* add any margins to them. Use Tailwind classes for spacing.

Chapter 4: Markdown Styling

Styling Markdown approaches

We're going to cover multiple approaches to styling Markdown content in Next.js that can be used separately or together:

- Defining components for elements in the MDX content and applying Tailwind classes to them
- Using CSS modules to style elements in the MDX content
- Using the Tailwind typography plugin

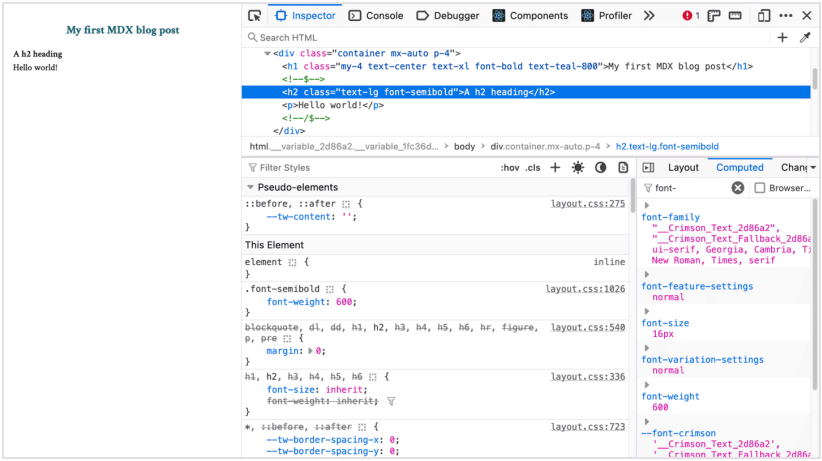
Markdown components

Our MDX content is unstyled. Let's set up some components in the `src/mdx-components.tsx` file to add Tailwind classes to heading elements:

```
export function
useMDXComponents(components:
MDXComponents): MDXComponents {
  return {
    h2: ({ children, ...props }) => (
      <h2 className="text-lg font-
semibold" {...props}>
        {children}
      </h2>
    ),
    ...components,
  };
}
```

To improve the heading hierarchy, we should also update the post title to be a `h1` in `src/app/blog/[slug]/page.tsx` and the first heading in the MDX file to be a `h2` i.e. change `# First MDX post` to `## A h2 heading`.

After manually refreshing the page, you should then see the Tailwind classes present on the heading element:



A Tailwind styled heading element from the MDX content

Generate classes for all Tailwind content

Tailwind won't generate more styles than it needs to and it relies on your configuration to know which classes to generate. To make sure Tailwind class names you use in MDX content get styles generated, add the blog and `mdx-components.tsx` paths to the `content` array in the `tailwind.config.ts` file:

```
const config: Config = {
  content: [
    "./src/app/**/*.
{js,ts,jsx,tsx,mdx}",
    "./src/blog/**/*.
{js,ts,jsx,tsx,mdx}",
    "./src/mdx-components.tsx",
  ],
  //
};
```

Now Tailwind classes found in these files will have generated styles.

Markdown styles using CSS modules

If we want to style not just the elements but how they work together, we can use CSS modules to style them. First, create a `src/app/blog/_components/markdown/markdown.module.css` file:

```
.markdown > h2 + p::first-line {
  @apply text-lg leading-snug
tracking-widest;
}
```

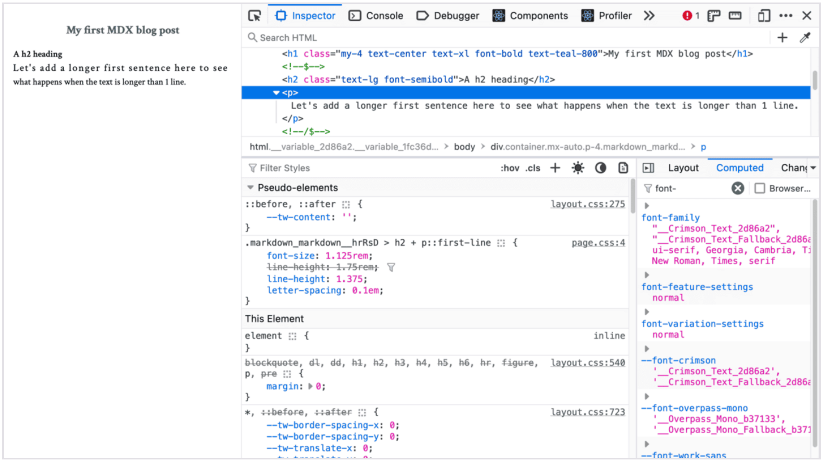
Then update the `page` component to import the CSS module and apply the “markdown” class to the container `div`:

```
import markdownStyles from "@/app/
blog/_components/markdown/
markdown.module.css";

//

export default async function
BlogPage({ params }: BlogPageProps) {
  //

  return (

<div className={`container mx-auto p-4
${markdownStyles["markdown"]}`}>
      <h1 className="my-4 text-center
text-xl font-bold text-teal-800">
        {title}
      </h1>
      <BlogMarkdown />
    </div>
  );
}
```

Let's test that by updating our MDX post:

```
## A h2 heading

Let's add a longer first sentence here
to see what happens when the text is
longer than 1 line.
```

Now we should see the first line of the paragraph styled differently:



The first line of the paragraph is styled differently and the Markdown class name is unique

Tailwind typography plugin

The `@tailwindcss/typography` plugin for Tailwind is designed to make prose like a blog post look good, with minimal effort, in a Tailwind way. The post Introducing Tailwind CSS Typography elaborates on how it fits into Tailwind's model and the Tailwind typography repo includes the docs.

Install the plugin package:

```
npm install -D @tailwindcss/typography
```

Update `tailwind.config.ts`:

```
const config: Config = {
  /* */
  plugins: [require("@tailwindcss/
typography")],
};
```

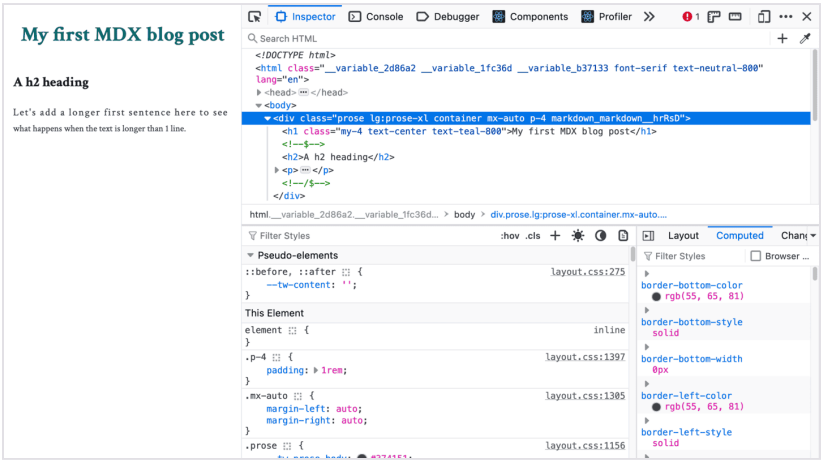
When you use the `.prose` class, and even breakpoint-specific prose classes like `lg:prose-xl` `dark:prose-invert`, your MDX content should look better:

```
export default async function
BlogPage({ params }: BlogPageProps) {
  //

  return (
    <div
      className={`prose lg:prose-xl da
rk:prose-invert container mx-auto p-4
${markdownStyles["markdown"]}`}
    >
      <h1 className="my-4 text-center
text-teal-800">{title}</h1>
      <BlogMarkdown />
    </div>
  );
}
```

With some amendments, we could rely entirely on `.prose` to set the font size and weight for our headings. First, remove the manually added `text-xl font-bold` classes from the `h1`. Second, remove the `h2` entry from the `mdx-components.tsx` file:

```
import type { MDXComponents } from
"mdx/types";

export function
useMDXComponents(components:
MDXComponents): MDXComponents {
  return {
    ...components,
  };
}
```

Now the size and weight of our headings are styled entirely by the `.prose` class:



Headings styled thanks to .prose

You can mix and match the 3 approaches to style your MDX content.

Chapter 5: Next.js Images

Add images

Let's add an image. We'll explore some approaches and challenges with images in MDX and static exports before we get to a solution. You should expect to see some errors along the way, which I'll include and cover how to address each one.

It's possible to throw an image tag into a post using Markdown syntax like this:

```
![A cryptic motionless bush stone-
curlew snuggled into wood chips](/
assets/images/curlew.jpg)
```

If you're happy with that, you can skip to the next chapter.

But let's assume you're interested in customising your images and using the `next/image` component that is encouraged by the Next.js docs:

```
import Image from "next/image";

<Image
  src="/assets/images/curlew.jpg"
  alt="A cryptic motionless bush
stone-curlew snuggled into wood chips"
/>
```

With the `Image` component you might get a funky error like:

```
Error: Cannot access Image.propTypes on the server. You cannot dot into a client module from a server component. You can only pass the imported name through.
```

Let's address that.

One workaround described by Daniel Cornelison for this error is to move the image to its own component, which we'll place in `src/app/blog/_components/Figure.tsx`:

```
import Image from "next/image";
import type { ImageProps } from "next/
image";

const Figure = (props: ImageProps) =>
{
  return <Image {...props}
alt={props.alt ?? ""} />;
};

export default Figure;
```

With this component in place, you can import the `Figure` component in your MDX file `src/blog/first-mdx-post.mdx` and use the component like this:

```
import Figure from "@/app/blog/
_components/Figure";

export const metadata = {
  title: "My first MDX blog post",
  description: "A short MDX blog
post.",
};

<Figure
  src="/assets/images/curlew.jpg"
  alt="A cryptic motionless bush
stone-curlew snuggled into wood chips"
/>
```

There's a small improvement we can make on top of that. We can set up a custom MDX component in `mdx-components.tsx`:

```
import Figure from "@/app/blog/
_components/Figure";
import type { MDXComponents } from
"mdx/types";

export function
useMDXComponents(components:
MDXComponents): MDXComponents {
  return {
    Figure: (props) => <Figure
{...props} />,
    ...components,
  };
}
```

With that change to `mdx-components.tsx`, we can then use the `Figure` component in our posts without needing to import it in each post. Now you can change the MDX file to look like this, removing the previous `Figure` import:

```
export const metadata = {
  title: "My first MDX blog post",
  description: "A short MDX blog
post.",
};

## A h2 heading

Let's add a longer first sentence here
to see what happens when the text is
longer than 1 line.

<Figure
  src="/assets/images/curlew.jpg"
  alt="A cryptic motionless bush
stone-curlew snuggled into wood chips"
  width={1600}
  height={900}
/>
```

But even then, while using a static export for your site you might still get this error in dev:

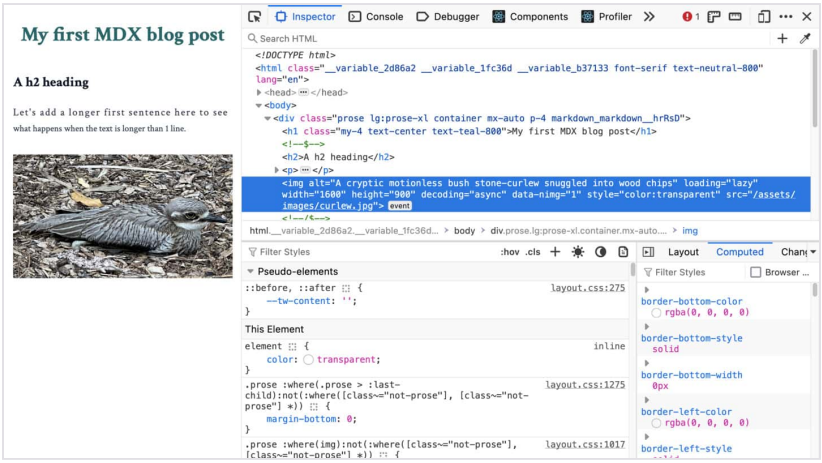
> Error: Image Optimization using the default loader is not compatible with `{ output: 'export' }`. Possible solutions:
>
> - Remove `{ output: 'export' }` and run "next start" to run server mode including the Image Optimization API.
> - Configure `{ images: { unoptimized: true } }` in `next.config.js` to disable the Image Optimization API.
>
> Read more: https://nextjs.org/docs/messages/export-image-api

And if you ran `npm run build`, it would build but the images would be mysteriously absent.

Next.js Image Optimization doesn't work out of the box with static exports. We can turn it off by amending `next.config.mjs` to include `images: { unoptimized: true }`:

```
const nextConfig = {
  output: "export",
  images: {
    unoptimized: true,
  },
  // …
};
```

Now when we run `npm run dev` there should be no error and we should be able to see our image:



Our post has inline images

The blog should also show images when built with `npm run build`.

To actually optimise images with a static export, you can define a custom image loader. That would let you use a service like Cloudinary for image optimisation. Alternatively, you could optimise images at build time using a package like `next-optimized-images`.

Otherwise, if you have no automatic optimisation, you might want to resize and optimise images manually before you commit them to your project repository. For example, you could use a tool like ImageOptim or ImageOptim-CLI.

Chapter 6: Blog Metadata and Navigation

Blog index listing page

Let's create a blog index listing page that shows a list of all of your blog posts.

Create a file, `src/app/blog/_lib/getAllBlogPostsData.ts`, to house our logic to get all the blog posts metadata. For convenience, we'll create some local helper functions to filter directory entries down to MDX files and to get a slug from a filename:

```
import type { Dirent } from "fs";

const isMDXFile = (dirent: Dirent) =>
  !dirent.isDirectory() &&
dirent.name.endsWith(".mdx");

const getSlugFromFilename = (dirent:
Dirent) =>
  dirent.name.substring(0,
dirent.name.lastIndexOf("."));
```

Using those helpers, we'll create the function to get all the blog post metadata:

```
import { readdir } from "fs/promises";
import { getBlogPostMetadata } from
"@/app/blog/_lib/getBlogPostData";
import type { BlogPostData } from "@/
app/blog/_lib/getBlogPostData";

export async function
getAllBlogPostsData():
Promise<BlogPostData[]> {
```

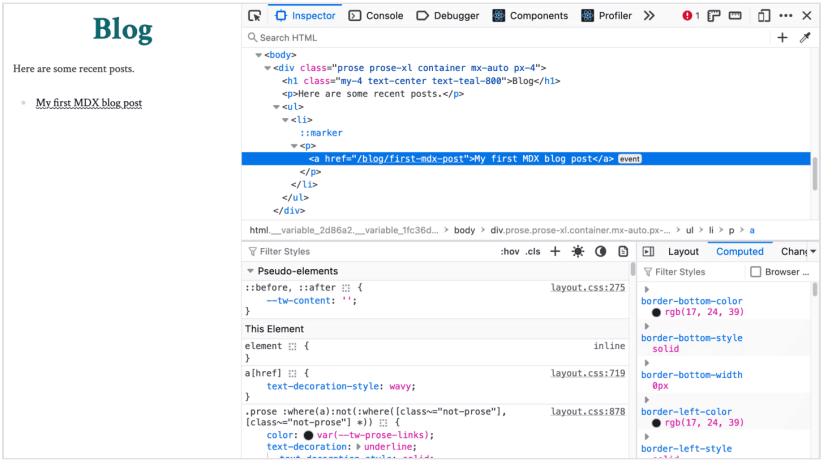
```
  try {
    const dirents = await readdir("./
src/blog/", {
      withFileTypes: true,
    });

    const slugs =
dirents.filter(isMDXFile).map(getSlugFromFilenam

    const result = await Promise.all(
      slugs.map((slug) => {
        return
getBlogPostMetadata(slug);
      }),
    );

    return result;
  } catch (error) {
    console.error(error);
    return [];
  }
}
```

After that we can create a new file, `src/app/blog/page.tsx`, for the blog index listing page to use our new `getAllBlogPostsData` function:

```
import Link from "next/link";
import { getAllBlogPostsData } from
"@/app/blog/_lib/getAllBlogPostsData";

export default async function Blogs()
{
  const blogs = await
getAllBlogPostsData();

  return (
    <div className="prose prose-xl
dark:prose-invert container mx-auto
px-4">
      <h1 className="my-4 text-center
text-teal-800">Blog</h1>
      <p>Here are some recent posts.</
p>
      <ul>
        {blogs.map(({ slug, metadata:
{ title } }) => (
          <li key={slug}>
            <p>
              <Link prefetch={false} h
ref={`/blog/${slug}`}>
                {`${title}`}
              </Link>
```

```
              </p>
            </li>
          ))}
        </ul>
      </div>
    );
}
```

Now when you visit `http://localhost:3000/blog`, you should see a list of blog posts:



A list of blog posts with a working link to the first MDX post

Now that we have a new `getAllBlogPostsData` function, we can use it to generate the static paths for the blog posts in `src/app/blog/[slug]/page.tsx` and fix our previously hard-coded blog post slug with the `FIXME`:

```
import { getAllBlogPostsData } from
"@/app/blog/_lib/getAllBlogPostsData";
//
export async function
generateStaticParams() {
  const blogPosts = await
getAllBlogPostsData();
  const blogStaticParams =
blogPosts.map((post) => ({
    slug: post.slug,
  }));

  return blogStaticParams;
}
```

Now you can make as many posts as you like.

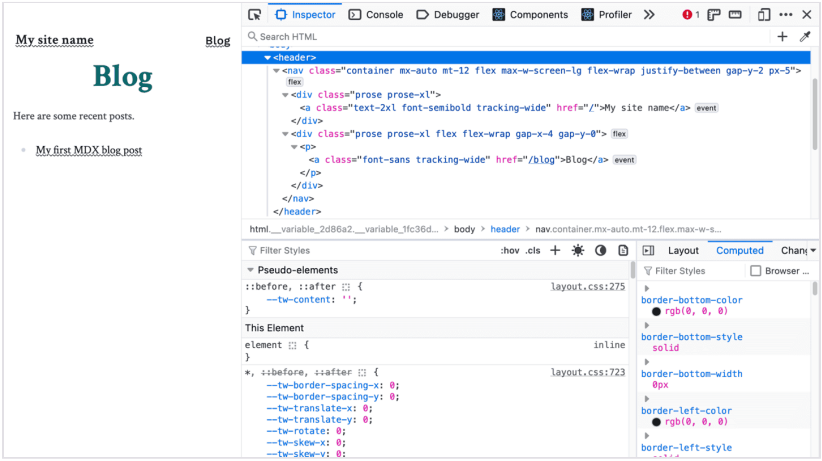
Navigation

Readers need to move about. Add a header containing `nav` to the existing root `src/app/layout.tsx` file inside the `<body>`:

```
import Link from "next/link";
//
export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html
      lang="en"
      className={`${crimson.variable}
${workSans.variable} ${overpassMono.va
riable} font-serif text-neutral-800`}
```

```
    >
      <body>
        <header>
          <nav className="container
mx-auto mt-12 flex max-w-screen-lg
flex-wrap justify-between gap-y-2
px-5">
            <div className="prose
prose-xl dark:prose-invert">
              <Link href="/"
className="text-2xl font-semibold
tracking-wide">
                My site name
              </Link>
            </div>
            <div className="prose
prose-xl dark:prose-invert flex flex-
wrap gap-x-4 gap-y-0">
              <p>
                <Link className="font-
sans tracking-wide" href="/blog">
                  Blog
                </Link>
              </p>
            </div>
          </nav>
        </header>
```

```
            {children}
          </body>
        </html>
    );
}
```



Navigation

Metadata title template

Now that we have multiple pages we can navigate between, we should fix up the metadata.

We want each page to have a unique page title, as well as include the website name. The page title is used in browser tabs, search engine results, social media previews, and Next.js's router to announce the page title to screen readers.

We can use the Next.js template object to generate metadata for each page. In the `src/app/layout.tsx` file, add a `metadata` object:

```
export const metadata: Metadata = {
  title: {
    template: `%s | My site name`,
    default: "My site name",
  },
  description: "My site is about…",
  authors: [{ name: "My name" }],
  openGraph: {
    locale: "en",
    type: "website",
  },
};
```

You can use the `%s` placeholder so that individual pages using the metadata title will have their title appended with the site name.

We can also add a blog layout in `src/app/blog/layout.tsx`:

```
import { Metadata } from "next";

export const metadata: Metadata = {
  title: {
    template: `%s | Blog | My site
name`,
    default: `Blog`,
  },
  description: "This blog is about…",
  openGraph: {
    locale: "en",
    type: "article",
  },
};

export default function
Layout({ children }: { children:
React.ReactNode }) {
  return <main className="mx-auto
px-4">{children}</main>;
}
```

Now the blog index page will have the title “Blog | My site name” and the individual blog post pages will have a title like “My first MDX blog post | Blog | My site name”.

Chapter 7: Syntax Highlighting

Add syntax highlighting

Using syntax highlighting in code blocks can make them easier to scan and read.

There are lots of options for syntax highlighting code blocks. Some add markup to the code blocks at build time and some add JavaScript to highlight code blocks at run time. For our static blog

purposes, we can use an option that runs at build time. To do this, we can add MDX plugins to our Next.js config.

> Rehype is a popular tool that transforms HTML with plugins. Remark is a tool that transforms Markdown with plugins. There are many plugins available for both tools. It's worth paying close attention to the names of plugins mentioned online so you don't mix up tools like `remark-highlight.js` and `rehype-highlight`.

Here are 3 popular choices for syntax highlighting HTML:

- `rehype-highlight`
- `rehype-prism-plus`
- `rehype-pretty-code`

All 3 packages have reasonably active development, are popular on NPM, and have no direct vulnerabilities in the Snyk Vulnerability Database. Each of them provide logical attributes and let you then style the attributes using your own theme stylesheet. They offer different features, such as highlighting line numbers or diffs. `rehype-highlight` has the fewest dependencies of the 3 and provides a solid foundation so we'll use that.

Add `rehype-highlight`

Install the package:

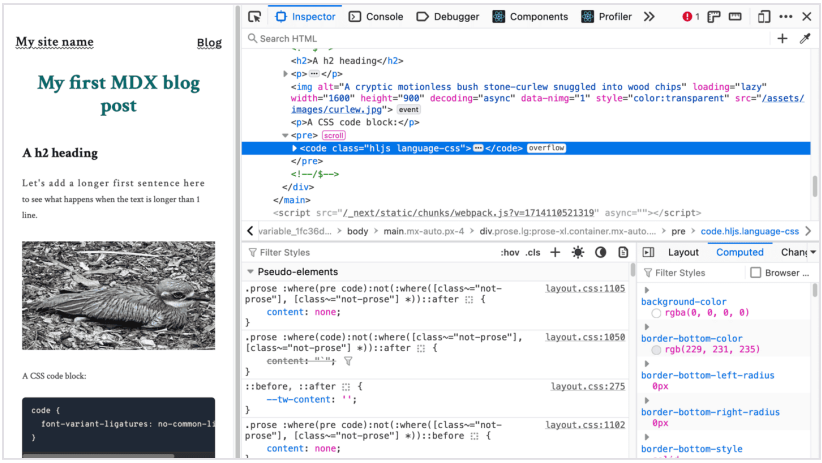
```
npm install rehype-highlight
```

Commit the `package.json` and `package-lock.json` changes.

In `next.config.mjs`, import the plugin package and add the plugin to an array of `rehypePlugins` in the MDX options:

```
import createMDX from "@next/mdx";
import rehypeHighlight from "rehype-
highlight";

//

const withMDX = createMDX({
  options: {
    rehypePlugins: [rehypeHighlight],
  },
});

export default withMDX(nextConfig);
```

Now, when you add a code block to your MDX content, it should have the `hljs` class added:



The hljs class is applied and the code block has no syntax highlighting yet.

Given a CSS code block, rehype highlight will add a `language-css` class too. By default, `rehype-highlight` includes 37 common languages.

Next, we need to style the added classes so that we have syntax highlighting.

Let's choose a `highlight.js` theme. You can find a list of themes on the highlight.js website. Set the language dropdown to "Common" to see examples just for the default included languages.

There are a lot of themes to choose from, but we can quickly cut down the list to a shorter list of candidates by browsing the accessible themes noted in this highlight.js issue.

The themes that have an accessible contrast ratio for text of 4.5:1 or greater are:

- `a11y-dark.css`: 7.1
- `a11y-light.css`: 4.5
- `devibeans.css`: 5.1
- `gml.css`: 4.7
- `ir-black.css`: 5
- `qtcreator-dark.css`: 6.7
- `stackoverflow-dark.css`: 5.5
- `stackoverflow-light.css`: 4.5
- `sunburst.css`: 4.8
- `tomorrow-night-bright.css`: 5

You can import external stylesheets in Next.js anywhere in the `app`. For example, you could import the `stackoverflow-dark.css` theme at the top of the `src/app/blog/[slug]/page.tsx` file:

```
import "highlight.js/styles/
stackoverflow-dark.css";
```

This approach assumes you have highlight.js package installed, which we do via `rehype-highlight`'s dependency on `lowlight`, which in turn depends on `highlight.js`.

With both the Tailwind typography plugin and the `highlight.js` theme providing padding, you might want to remove the `.prose pre` padding in `tailwind.config.ts`:

```
const config: Config = {
  //
  theme: {
    //
    extend: {
      //
```

```
      typography: () => ({
        DEFAULT: {
          css: {
            pre: {
              paddingTop: 0,
              paddingInlineEnd: 0,
              paddingBottom: 0,
              paddingInlineStart: 0,
            },
          },
        },
        xl: {
          css: {
            pre: {
              paddingTop: 0,
              paddingInlineEnd: 0,
              paddingBottom: 0,
              paddingInlineStart: 0,
            },
          },
        },
      }),
    },
  },
  plugins: [require("@tailwindcss/
typography")],
};
```

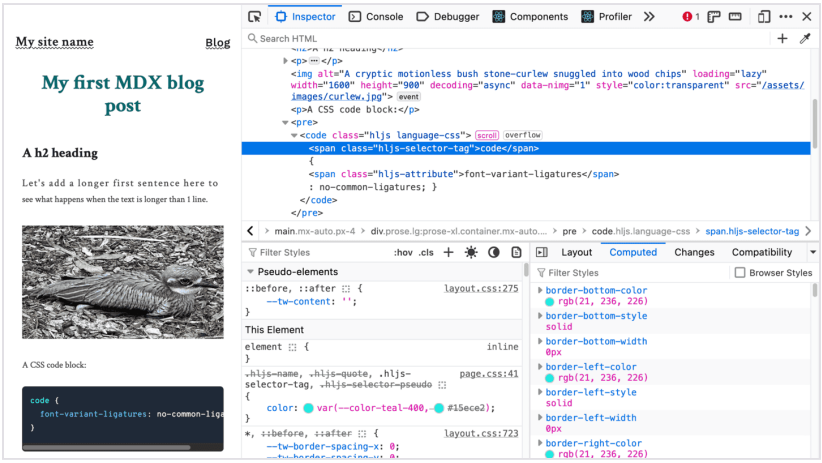
If you want to design your own highlight.js theme, you can make a small CSS file with the right classes for the stylable scopes. For example:

```
.hljs {
  display: block;
  overflow-x: auto;
  padding: 1em;
}

.hljs,
.hljs-subst {
  color: #fff;
}

.hljs-comment {
  color: var(--color-neutral-700,
#4f4f4f);
}

.hljs-attribute,
.hljs-literal,
.hljs-meta,
.hljs-number,
```

```
.hljs-operator,
.hljs-variable,
.hljs-selector-attr,
.hljs-selector-class,
.hljs-selector-id {
  color: var(--color-blue-400,
#3cc1f4);
}

.hljs-name,
.hljs-quote,
.hljs-selector-tag,
.hljs-selector-pseudo {
  color: var(--color-teal-400,
#15ece2);
}
```

You can then import that CSS file at the top of the `src/app/blog/[slug]/page.tsx` file:

```
import "./highlightjs-example.css";
```



The CSS code block now has custom syntax highlighting

If you wanted to pass options to the `rehype-highlight` plugin in `next.config.mjs`, you can use an array of arrays:

```
const withMDX = createMDX({
  options: {
    rehypePlugins: [[rehypeHighlight,
{ aliases: { markdown: "mdx" } }]],
  },
});
```

This lets you treat MDX code blocks as Markdown code blocks because MDX is not a registered language.

Check the output of `npm run build` to check that syntax highlighting works there without JavaScript.

Chapter 8: Heading IDs and Links

Add IDs to headings

As a reader or an author, it can be useful to link to specific sections in a post, which requires headings to have IDs. We can use `rehype-slug` to automatically add IDs to headings.

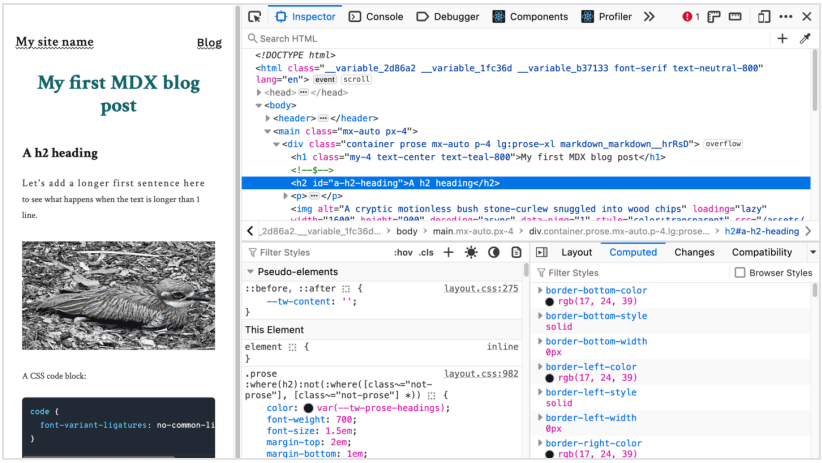
Install the package and commit the changes:

```
npm install rehype-slug
```

In `next.config.mjs`, import the plugin package and add the plugin to the array of `rehypePlugins`:

```
import rehypeSlug from "rehype-slug";

//

const withMDX = createMDX({
  options: {
    rehypePlugins: [
      rehypeSlug,
      [rehypeHighlight, { aliases: {
markdown: "mdx" } }],
    ],
  },
});
```

Run `npm run dev` and you should now see IDs added to headings in your MDX content:



The heading now has an automatic ID attribute

Now it is possible to link to a specific heading in a post.

Add links to headings

Once your headings have IDs, you can use `rehype-autolink-headings` to create links on headings so that each heading links to itself. This makes it easy to click on a heading to get a link to that section.

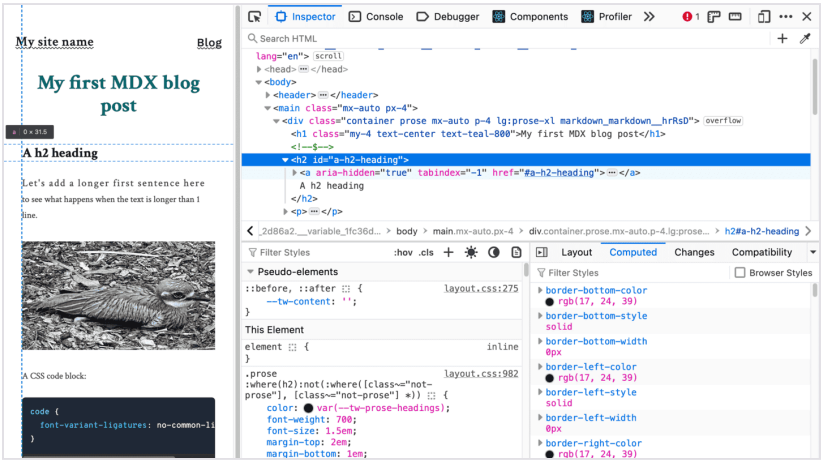
Install the package and commit the changes:

```
npm install rehype-autolink-headings
```

In `next.config.mjs`, we'll import the plugin package and add the plugin to the array of `rehypePlugins`. This plugin depends on headings already having IDs, so it needs to come after `rehype-slug` in the array:

```
import rehypeAutolinkHeadings from
"rehype-autolink-headings";

//

const withMDX = createMDX({
  options: {
    rehypePlugins: [
      rehypeSlug,
      rehypeAutolinkHeadings,
      [rehypeHighlight, { aliases: {
markdown: "mdx" } }],
    ],
  },
});
```

Now when you inspect the headings in your MDX content, you should see that they contain an anchor tag:



The heading now contains an automatic link

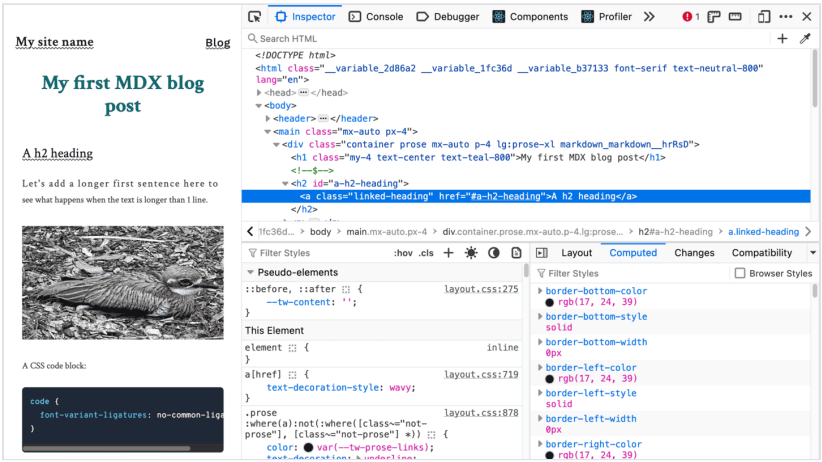
Technically it's there but the reader can't do much with it yet!

If we inspect the `rehype-autolink-headings` API docs, we can see there are options to wrap the heading text with the link and to add a class to the link. We can add these options to the plugin in `next.config.mjs`:

```
const withMDX = createMDX({
  options: {
    rehypePlugins: [
      rehypeSlug,
      [
        rehypeAutolinkHeadings,
        {
          behavior: "wrap",
          properties: {
            className: "linked-
heading",
          },
        },
      ],
      [rehypeHighlight, { aliases: {
markdown: "mdx" } }],
    ],
  },
});
```

Now it should be possible to click the heading text to navigate to that section. Once clicked, the URL should update to include the heading ID as the URL's fragment e.g. `http://localhost:3000/blog/first-mdx-post#a-h2-heading`.

But the styling for a link inside a heading is not great.



The heading has a wavy underline

There are a few ways we could improve that. Let's tweak the `tailwind.config.ts`:

```
const config: Config = {
  //
  theme: {
    //
    extend: {
      //
      typography: () => ({
        DEFAULT: {
          css: {
            ":is(h1, h2, h3, h4, h5)
a": {
              "font-weight":
"inherit",
              "text-decoration":
"inherit",
            },
            //
          },
        },
        xl: {
          css: {
```

```
            ":is(h1, h2, h3, h4, h5)
a": {
              "font-weight":
"inherit",
              "text-decoration":
"inherit",
            },
            //
          },
        },
      }),
    },
  },
  //
};
```

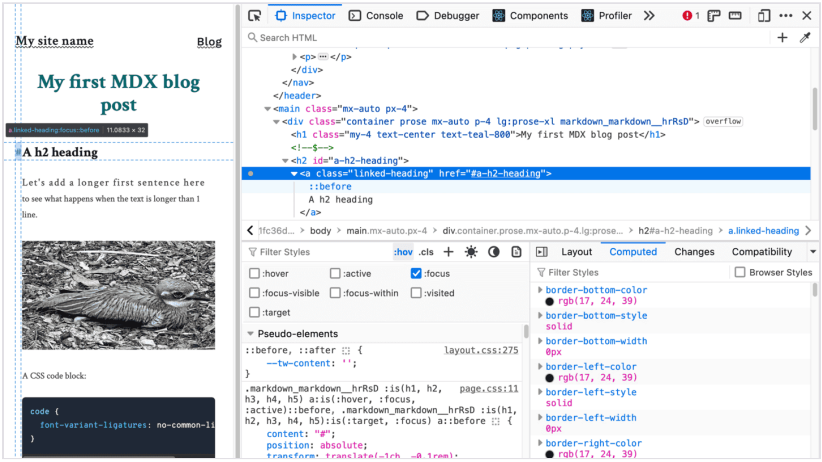
Now the linked headings should look like they did before the link was added.

When a heading link is hovered over, has focus, or is active, we can add a `#` symbol before the heading text. Once the heading link is clicked and becomes

the "target element" (with an `id` matching the URL's fragment), we can also display the `#` symbol to show that it is the selected section. We can achieve both of these styling goals using the `:is()` pseudo-class function and the `::before` pseudo-element in the `src/app/blog/_components/markdown/markdown.module.css` file:

```
.markdown :is(h1, h2, h3, h4, h5)
a:is(:hover, :focus, :active)::before,
.markdown :is(h1, h2, h3, h4, h5):is(:
target, :focus) a::before {
  content: "#";
  position: absolute;
  transform: translate(-1ch, -0.1rem);
}
```

Now when we hover over a heading link, the `#` symbol should appear before the heading text. When we click the heading link and navigate to `http://localhost:3000/blog/first-mdx-post#a-h2-heading`, the `#` symbol should remain before the heading text, even when we move focus away from the heading:



The heading has a # symbol in front of it

Now you and your readers can easily access links to specific sections to share with other people.

Chapter 9: RSS

Add RSS

RSS (RDF Site Summary or Really Simple Syndication) is a web feed that lets people subscribe to updates from your site. People can follow your blog using an RSS feed reader.

To create an RSS feed, you can use a Route Handler to return a non-UI response.

The RSS example from the Next.js docs includes `export const dynamic = 'force-dynamic' // defaults to auto`, which will throw an error in projects with static exports:

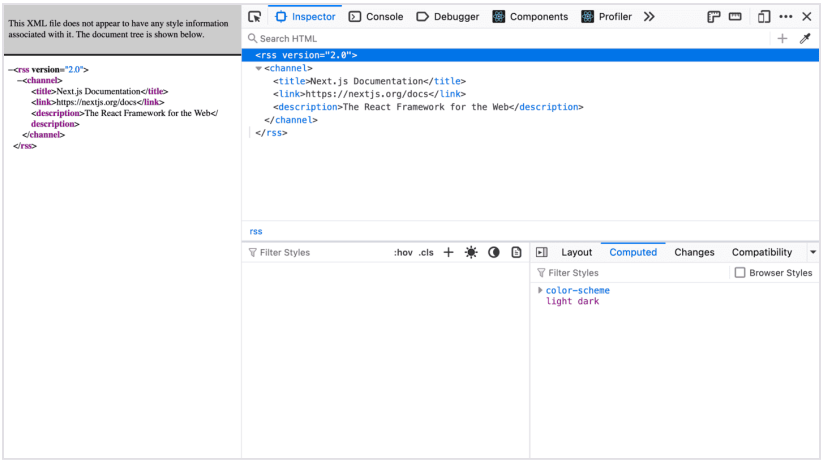
```
Error: export const dynamic = "force-
dynamic" on page "/blog/rss.xml" cannot
be used with "output: export". See more
info here: https://nextjs.org/docs/
advanced-features/static-html-export
```

But we can use the rest of the example as a starting point in a new file, `src/app/blog/rss.xml/route.ts`:

```
export async function GET() {
  return new Response(
    `<?xml version="1.0"
encoding="UTF-8" ?>
<rss version="2.0">

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

</rss>`,
    {
      headers: {
        "Content-Type": "text/xml",
      },
    },
  );
}
```

Whenever you find yourself writing a custom route handler (`route.ts`) in Next.js, it's a good time consider custom route handler security practices, such as any Cross-site Request Forgery (CSRF) risks. This plain little RSS feed example in our static site is safe enough because we're not handling sensitive data or connecting to a database or anything like that, but it's a good habit to consider security when writing custom route handlers.

When you visit `http://localhost:3000/blog/rss.xml` you should see the RSS feed XML:



An unstyled XML RSS feed

But using a template string to create the XML is chaotic. We can use an RSS feed package, such as `rss` or `feed` to generate the XML. Neither package has had a new release published in recent years, but RSS itself is stable so they probably wouldn't need a lot of updates. Neither package has a history of direct vulnerabilities in Snyk. Let's go with the slightly newer option, `feed`.

Install the package and commit the changes:

```
npm install feed
```

In `src/app/blog/rss.xml/route.ts`, import the package and generate a feed:

```
import { Feed } from "feed";

const feed = new Feed({
  title: "My Blog RSS Feed",
  description: "This is my personal
feed!",
  id: "https://example.com/blog",
  link: "https://example.com/blog/
index.xml",
  language: "en",
  copyright: "All rights reserved
2024, My Name",
});
```

In that same route handler, add a `GET` function that returns the feed in RSS 2.0 format as a response:

```
export async function GET() {
  return new Response(feed.rss2(), {
    headers: {
      "Content-Type": "application/
rss+xml",
    },
  });
}
```

The `feed` object is created with the required properties (and an optional one) and then the `rss2()` method is called to generate the feed. The `GET` function returns the response as `application/rss+xml` now instead of `text/xml`.

Now if you visit `http://localhost:3000/blog/rss.xml` your browser might download the file instead of viewing it. You can inspect the output in your editor to confirm the content looks correct and use the W3C feed validator to check the syntax. To learn more about the possible values to use in an RSS feed, check out the RSS Specification.

To add posts to our RSS feed, we can reuse our `getAllBlogPostsData` function in `src/app/blog/rss.xml/route.ts`:

```
import { Feed } from "feed";
import { getAllBlogPostsData } from
"@/app/blog/_lib/getAllBlogPostsData";

const feed = new Feed({
  //
});
```

```
export async function GET() {
  const posts = await
getAllBlogPostsData();

  posts.forEach((post) => {
    feed.addItem({
      title: `$
{post.metadata.title ?? ""}`,
      link: `https://example.com/blog/
${post.slug}`,
      description: `$
{post.metadata.description ?? ""}`,
      date: new Date(), // TODO: set
this to the post's publish date
    });
  });

  return new Response(feed.rss2(), {
    headers: {
      "Content-Type": "application/
rss+xml",
    },
  });
}
```

This should fetch the blog post metadata and add each post to the feed.

But using `new Date()` to set a new date for each post every time the feed is built is misleading. We can add a publish date to our posts and use that instead. While we could add it directly to our existing `metadata` object, Next.js has a lot of behaviour around the `metadata` object, such as automatically creating `<meta>` tags, and we might want to keep our post metadata separate for clarity. Instead, we can add a `publishDate` property to a `customMetadata` object.

In `src/app/blog/_lib/getBlogPostData.ts`, we can add a `publishDate` property to a `customMetadata` object on our `BlogPostData` type and return that data from `getBlogPostMetadata`:

```
export type PostMetadata = Metadata &
{
  title: string;
  description: string;
};

export type CustomMetadata = {
  publishDate: string;
};

export type BlogPostData = {
  slug: string;
  metadata: PostMetadata;
  customMetadata: CustomMetadata;
};

export async function
getBlogPostMetadata(slug: string):
Promise<BlogPostData> {
  try {
    const file = await import("@/
blog/" + slug + ".mdx");

    if (file?.metadata &&
file?.customMetadata) {
      if (!file.metadata.title || !
```

```
file.metadata.description) {
        throw new Error(`Missing some
required metadata fields in: ${slug}
`);
      }

      if (!
file.customMetadata.publishDate) {
        throw new Error(`Missing
required custom metadata field,
publishDate, in: ${slug}`);
      }

      return {
        slug,
        metadata: file.metadata,
        customMetadata:
file.customMetadata,
      };
    //
    }
  //
  }
}
```

In our post, `src/blog/first-mdx-post.mdx`, add a publish date using a JavaScript `Date` friendly format in an exported `customMetadata` object:

```
export const metadata = {
  title: "My first MDX blog post",
  description: "A short MDX blog
post.",
};

export const customMetadata = {
  publishDate:
"2024-03-27T09:00:00+10:00",
};
```

Finally, we can update our RSS feed in `src/app/blog/rss.xml/route.ts` to use this new date:

```
posts.forEach((post) => {
  feed.addItem({
    title: `${post.metadata.title ??
""}`,
    link: `https://example.com/blog/$
{post.slug}`,
    description: `$
{post.metadata.description ?? ""}`,
    date: new
Date(post.customMetadata.publishDate),
  });
});
```

When you visit `http://localhost:3000/blog/rss.xml` you should be able to download the new RSS feed with the post including its publish date.

Now that we have a publish date, we can also sort blog posts by this date in `src/app/blog/_lib/getAllBlogPostsData.ts`:

```
const result = await Promise.all(
  slugs.map((slug) => {
    return getBlogPostMetadata(slug);
  }),
);

result.sort(
  (a, b) =>
    +new
Date(b.customMetadata.publishDate) -
    +new
Date(a.customMetadata.publishDate),
);
```

This sort function converts the publish date strings to `Date` objects, coerces them to numbers in milliseconds since the epoch using the + unary plus operator, and then subtracts them to get the difference. The positive, negative, or zero result

decides whether post `a` or post `b` should come first, where `b - a` gives us descending order—newest post first.

It's also a good time to test that this works in a static export. Run `npm run build` and check that the RSS feed is generated.

If you're happy with the results, you can add the RSS feed to the metadata of your site. In `src/app/layout.tsx`, you can add an `alternates` object to the `metadata` object with your feed's title and URL:

```
export const metadata: Metadata = {
  //
  alternates: {
    types: {
      "application/rss+xml": [
        {
          title: "My Blog RSS Feed",
          url: "https://example.com/
blog/index.xml",
        },
      ],
    },
  },
};
```

Next.js docs on Metadata object alternates don't mention the "title" property, but the code supports arrays of `AlternateLinkDescriptor`s with an optional title string.

Note: If you're using `trailingSlash: true` in your `next.config.mjs` and a Next.js version from v14.1.1 to v14.2.3, you might encounter a bug where the trailing slash is added to the RSS feed URL. You can downgrade to v14.1.0 or upgrade to v14.2.4 or above to fix this issue.

RSS feeds with dynamic routes

We have only 1 RSS feed in our site and it uses a static route. If we were using a dynamic route, such as producing an RSS feed for each "category" or "tag" on our blog, we wouldn't be able to use this `route.ts` approach.

In a dynamic route like `src/app/blog/[category]/rss.xml/route.ts`, we wouldn't know at build time which category like "tech" or "design" needs to be built and which category of posts to fetch. At compile time, there's no web request to get the dynamic category from the URL. There's also no current way to access the dynamic route segment in a static export because that dynamic logic falls into unsupported features in static exports.

We won't do this but if you wanted to make that work, you could call a `generateRssFeed` method in a file like `src/category/[category]/page.tsx`:

```
export async function
generateStaticParams() {
  for (const category of categories) {
    await generateRssFeed(category);
  }

  const result =
categories.map((category) => ({
    category,
  }));

  return result;
}
```

And then in the `generateRssFeed` function, you could use the `category` to fetch the posts for that category and generate the RSS feeds into the `public/category/` directory:

```
export default async function
generateRssFeed(category: CategoryId)
```

```
{
  const categoryData =
getCategoryData(category);

  const feed = new RSS({
    title: `My ${categoryData.title}
RSS Feed`,
    id: `https://example.com/category/
${category.id}`,
    link: `https://example.com/
category/${category.id}/rss.xml`,
    copyright: "All rights reserved
2024, My Name",
  });

  const posts = await
getPostsByCategory(category);
  posts.forEach((post) => {
    feed.addItem({
      title: `$
{post.metadata.title ?? ""}`,
      link: `https://example.com/blog/
${post.slug}`,
      description: `$
{post.metadata.description ?? ""}`,
      date: new
Date(post.customMetadata.publishDate),
```

```
    });
  });

  const categoryRSSDir = `./public/
category/${category.id}`;

  if (!(await stat(categoryRSSDir))) {
    await mkdir(categoryRSSDir, {
recursive: true });
  }

  await writeFile(`${categoryRSSDir}/
rss.xml`, feed.rss2(), "utf-8");
}
```

Chapter 10: Favicons and Dark Mode

Favicon

A favicon is a small icon representing your site that can appear in a browser tab next to the page title, in bookmarks, in browser history, or on desktops or mobile home screens. You can use a custom favicon to help your blog stand out.

Next.js docs on favicons tells us that there are 2 ways to set icons in a Next.js app:

- Using image files (.ico, .jpg, .png)
- Using code to generate an icon (.js, .ts, .tsx)

We'll go with the first option.

Take an image file that's over 260x260px and convert it to a favicon using a tool like RealFaviconGenerator. RealFaviconGenerator will produce a zip file containing the favicon in multiple formats for wide coverage of devices.

Extract the favicon files into a directory called `public/favicon`. For extra coverage, you can copy the `favicon.ico` file to the root of the `public` directory and copy the `favicon-32x32.png` file to `public/favicon.png`. This should make sure that

the browser shows a favicon when looking at non-HTML documents, such as RSS feeds or PDFs that don't have `<link>` tags to tell them where to find the favicon file.

Delete any default favicons files provided by the Next.js CLI, such as `src/app/favicon.ico`.

Your files should look like this:

```
public/
└── favicon/
    └── android-chrome-192x192.png
    └── android-chrome-256x256.png
    └── apple-touch-icon.png
    └── browserconfig.xml
    └── favicon-16x16.png
    └── favicon-32x32.png
    └── favicon.ico
    └── mstile-150x150.png
    └── safari-pinned-tab.svg
    └── site.webmanifest
└── favicon.ico
└── favicon.png
```

Now in `src/app/layout.tsx`, add a `head` section to the `RootLayout` and some `<link>` tags to include the favicon:

```
export default function RootLayout({
  children,
}: Readonly<{
```

```
  children: React.ReactNode;
}>) {
  return (
    <html
      lang="en"
      className={`${crimson.variable}
${workSans.variable} ${overpassMono.va
riable} font-serif text-neutral-800`}
    >
      <head>
        <link
          rel="apple-touch-icon"
          sizes="180x180"
          href="/favicon/apple-touch-
icon.png"
        />
        <link
          rel="icon"
          type="image/png"
          sizes="32x32"
          href="/favicon/
favicon-32x32.png"
        />
        <link
          rel="icon"
          type="image/png"
          sizes="16x16"
```

```
          href="/favicon/
favicon-16x16.png"
        />
        <link rel="manifest" href="/
favicon/site.webmanifest" />
        <link
          rel="mask-icon"
          href="/favicon/safari-
pinned-tab.svg"
          color="#008080"
        />
        <link rel="shortcut icon"
href="/favicon/favicon.ico" />
        <meta name="msapplication-
TileColor" content="#000000" />
        <meta
          name="msapplication-config"
          content="/favicon/
browserconfig.xml"
        />
        <meta name="theme-color" conte
nt="#ffffff" />
      </head>
      <body>
        <header>{/* */}</header>
        {children}
      </body>
```

```
      </html>
    );
  }
```

Depending on how you've generated your favicon, you might need to adjust the paths in the linked `site.webmanifest` and `browserconfig.xml` files e.g. change `"src": "/android-chrome-192x192.png",` to `"src": "/favicon/android-chrome-192x192.png",`.

Now when you visit `http://localhost:3000`, you should see the favicon in the browser tab. I won't share a picture here because you will need your own favicon for this.

Dark mode

Dark mode is a feature to change the interface to a dark color scheme. It is popular to provide light and dark themes to let people read your blog the way they like.

To support a dark color scheme, we'll take a few steps.

First, we'll replace any instances of `prose` with `prose dark:prose-invert`. For your convenience, I already included `dark:prose-invert` in the previous code examples but if you added any of your own `prose` classes, be sure to add `dark:prose-invert` with them, e.g.:

```
<div
  className={`prose lg:prose-xl
dark:prose-invert container mx-auto
p-4 ${markdownStyles["markdown"]}`}
>
  <h1 className="my-4 text-center
text-teal-800">{title}</h1>
  <BlogMarkdown />
</div>
```

Next, we'll set the site background color in `src/app/layout.tsx`:

```
<html
  lang="en"
  className={`${crimson.variable} $
{workSans.variable} $
{overpassMono.variable} bg-white font-
serif text-neutral-800 dark:bg-
neutral-950`}
>
  {/* */}
</html>
```

We can now remove the default `body` styles that the Next.js CLI gave us in `src/app/globals.css` and rely on Tailwind classes instead. That is, remove these lines:

```
@media (prefers-color-scheme: dark) {
  :root {
    --foreground-rgb: 255, 255, 255;
  }
}

body {
  color: rgb(var(--foreground-rgb));
  background: rgb(var(--background-
rgb));
}
```

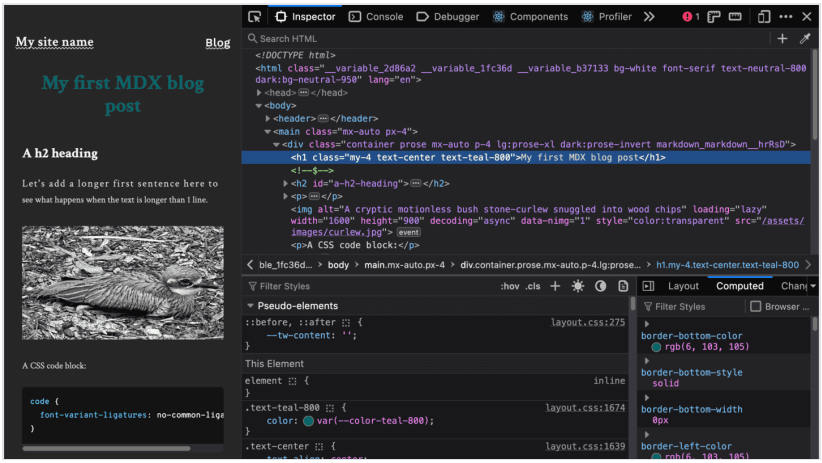
We can also export a Viewport object from the `src/app/layout.tsx` file to set the color scheme to `light dark` using a meta name to show light and dark schemes are supported by the site. While we're at it, we'll also add theme colors for light and dark mode:

```
import type { Viewport } from "next";

//

export const viewport: Viewport = {
  colorScheme: "light dark",
  themeColor: [
    { media: "(prefers-color-scheme:
light)", color: "#c5fffa" },
    { media: "(prefers-color-scheme:
dark)", color: "#003235" },
  ],
};
```

If you haven't already, you'll need to turn on dark mode. You can simulate dark mode using Firefox dev tools to view `prefers-color-scheme` as shown in this StackOverflow post, using Chrome to emulate CSS media feature `prefers-color-scheme`, or using Safari to override user preferences in the

Elements Tab of the Web Inspector. You can also set dark mode for macOS and change colors to dark for Windows.

Altogether now, here's how it should look:



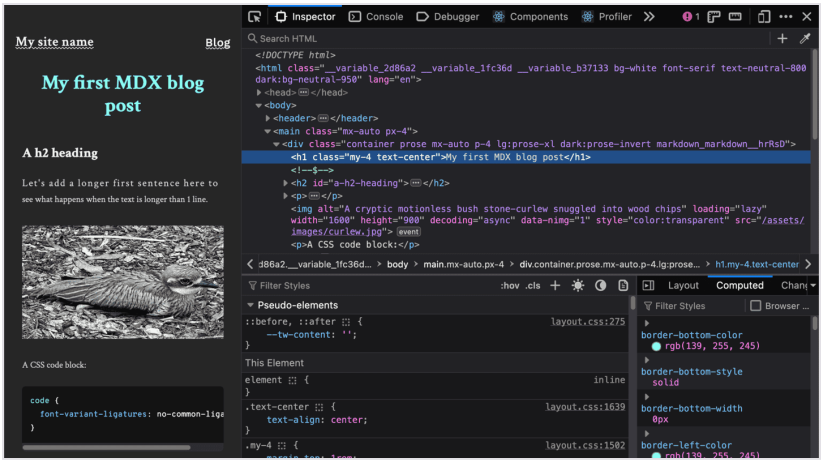
An initial dark mode attempt with a dark teal heading

If you want to give readers the ability to switch between light and dark mode on your site, check out the Tailwind docs on Toggling dark mode manually.

The first pass isn't too bad but let's update the heading color. We *could* update the `text-teal-800` class to use `text-teal-800 dark:text-teal-200` but there's another way. We can remove that class and update our `tailwind.config.ts` typography plugin settings to add a custom color theme:

```
import type { PluginAPI } from
"tailwindcss/types/config";

const config: Config = {
  //
  theme: {
    //
    extend: {
      //
      typography: ({ theme }:
```

```
PluginAPI) => ({
        DEFAULT: {
          css: {
            "--tw-prose-headings":
theme("colors.teal[800]"),
            "--tw-prose-invert-
headings": theme("colors.teal[200]"),
            //
          },
        },
        //
      }),
    },
  },
  plugins: [require("@tailwindcss/
typography")],
};
```



A vibrant teal heading without classes

Now you can go forth and style your site in dark mode.

Chapter 11: Google Analytics and Sitemap

Add Google Analytics

Adding web analytics to your site can be useful to help you understand reader behaviour and improve their experience.

To add some popular third-party packages like Google Analytics, you can use Next's experimental package for third-party libraries.

First, we'll look at how to use Google Analytics without Google Tag Manager.

Install the latest third-parties package and update Next.js to match:

```
npm install @next/third-parties@latest
next@latest
```

Commit the changes.

In your `src/app/layout.tsx` file, you can import the `GoogleAnalytics` component from `@next/third-parties` and use the `<GoogleAnalytics />` component after the closing `</body>` tag in the `RootLayout`:

```
import { GoogleAnalytics } from
"@next/third-parties/google";

//

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html
      lang="en"
      className={`${crimson.variable}
${workSans.variable} ${overpassMono.va
riable} bg-white font-serif text-
neutral-800 dark:bg-neutral-950`}
    >
      {/* */}
      <body>{/* */}</body>

<GoogleAnalytics gaId={"G-TODO"} />
    </html>
  );
}
```

Replace `G-TODO` with your Google Analytics ID.

If you want to use Google Analytics via Google Tag Manager, you can change the import to `import { GoogleTagManager } from '@next/third-parties/google'` and place `<GoogleTagManager gtmId="GTM-XYZ" />` with your GTM container ID *before* the opening `<body>` tag.

This is a good time to consider any relevant privacy regulations. You might add a privacy policy that outlines your data collection practices. You may also need to add cookie consent.

Add a sitemap

A sitemap lets you tell web crawlers about new pages on your site and how often they are updated. This can help some search engines index your site more effectively and share your latest posts.

We are going to generate a sitemap in Next.js to help search engines index our site with its pages and posts.

Create a file called `src/app/sitemap.ts`:

```
import { MetadataRoute } from "next";

export default async function sitemap():
Promise<MetadataRoute.Sitemap> {
  return [
    {
      url: `https://example.com/
index.html`,
      lastModified: new
Date("2024-04-15T12:00:00+10:00"),
      changeFrequency: "monthly",
      priority: 1,
    },
  ];
}
```

This is an extremely simple example showing just the home page. You can add more pages and posts to the sitemap by adding them to the array.

We'll reuse our `getAllBlogPostsData` function in a `getBlogsSitemap()` function:

```
import { getAllBlogPostsData } from
"@/app/blog/_lib/getAllBlogPostsData";

export async function getBlogsSitemap():
Promise<MetadataRoute.Sitemap> {
  const posts = await
getAllBlogPostsData();
  const blogPostEntries:
MetadataRoute.Sitemap =
posts.map((post) => ({
    url: `https://example.com/blog/$
{post.slug}/index.html`,
    lastModified:
post.customMetadata.publishDate,
    changeFrequency: "yearly",
    priority: 0.9,
  }));

  const newestBlogDate =
posts[0].customMetadata.publishDate;

  const blogIndexEntry:
MetadataRoute.Sitemap[0] = {
    url: `https://example.com/blog/
index.html`,
    lastModified: newestBlogDate,
```

```
    changeFrequency: "weekly",
    priority: 0.5,
  };

  const result =
[blogIndexEntry, ...blogPostEntries];

  return result;
}

//
```

In the sitemap, we'll use that new function to add blog posts to the sitemap. We'll also add the blog index page:

```
//

export default async function sitemap():
Promise<MetadataRoute.Sitemap> {
  const blogsSitemap = await
getBlogsSitemap();
```

```
  return [
    {
      url: `https://example.com/
index.html`,
      lastModified: new Date(),
      changeFrequency: "monthly",
      priority: 1,
    },
    ...blogsSitemap,
  ];
}
```

Be sure to update the URLs to match your site's URLs.

At the time of writing, a project with a static export using the supported `sitemap.xml` file convention will cause an error (here's the issue) when you run `npm run dev` like:

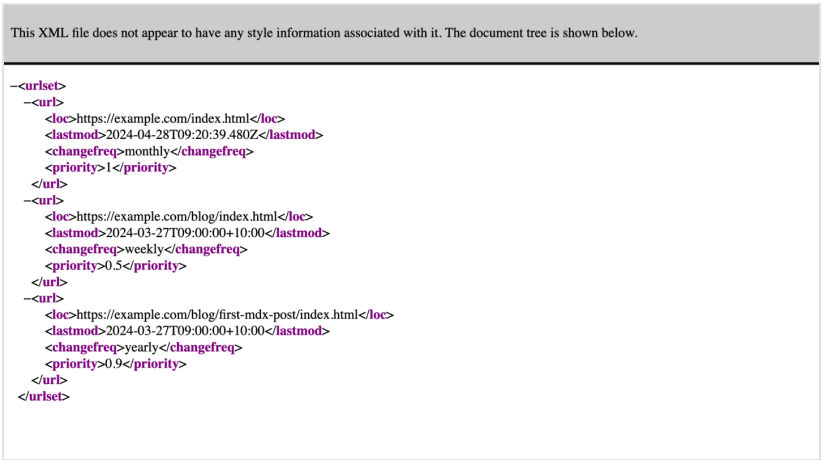
```
Error: Page "/sitemap.xml/
[[..__metadata_id__]]/route" is missing
exported function
"generateStaticParams()", which is
required with "output: export" config.
```

But it will work if you run `npm run build`!

If you don't want to wait for a build while iterating on your code, you can tweak the `next.config.mjs` file to set the output to `export` only in production:

```
const nextConfig = {
  // output: "export",
  output: process.env.NODE_ENV ===
"production" ? "export" : undefined,
  //
};
```

If you tweak the config like that and visit `http://localhost:3000/sitemap.xml` or run `npm run build && npx http-server ./out/` and visit `http://localhost:8080/sitemap.xml` you should see the sitemap XML:



A small, generated sitemap

Chapter 12: Robots file, 404s, and Open Graph images

`robots.txt`

Let's add a `robots.txt` file to tell search engines and other crawlers which pages they can access.

We can make a static `robots.txt` in Next.js by adding a file called `app/robots.txt`:

```
User-Agent: *
Allow: /
Disallow: /private/

Sitemap: https://example.com/
sitemap.xml
```

This tells search engines where to find our sitemap.

You can validate your `robots.txt` contents to check if a specific URL would be accessible by a specific user agent using a tool like TechnicalSeo.com's robots.txt Validator and Testing Tool.

AI scrapers

If you want to block crawlers that train Large Language Models (LLMs) with your content, you can add `User-Agent` directives to block specific bots. Here's an example:

```
Sitemap: https://example.com/
sitemap.xml

User-agent: *
Disallow:

User-agent: AdsBot-Google
User-agent: Amazonbot
User-agent: Applebot
User-agent: AwarioRssBot
User-agent: AwarioSmartBot
User-agent: Bytespider
User-agent: CCBot
User-agent: ChatGPT-User
User-agent: Claude-Web
User-agent: ClaudeBot
```

```
User-agent: DataForSeoBot
User-agent: FacebookBot
User-agent: FriendlyCrawler
User-agent: GPTBot
User-agent: Google-Extended
User-agent: GoogleOther
User-agent: ImagesiftBot
User-agent: Meltwater
User-agent: PerplexityBot
User-agent: PiplBot
User-agent: Seekr
User-agent: YouBot
User-agent: anthropic-ai
User-agent: cohere-ai
User-agent: img2dataset
User-agent: magpie-crawler
User-agent: omgili
User-agent: omgilibot
User-agent: peer39_crawler
User-agent: peer39_crawler/1.0
Disallow: /
```

Here's a list of AI agents and robots to block.

Add a custom 404 page and error pages

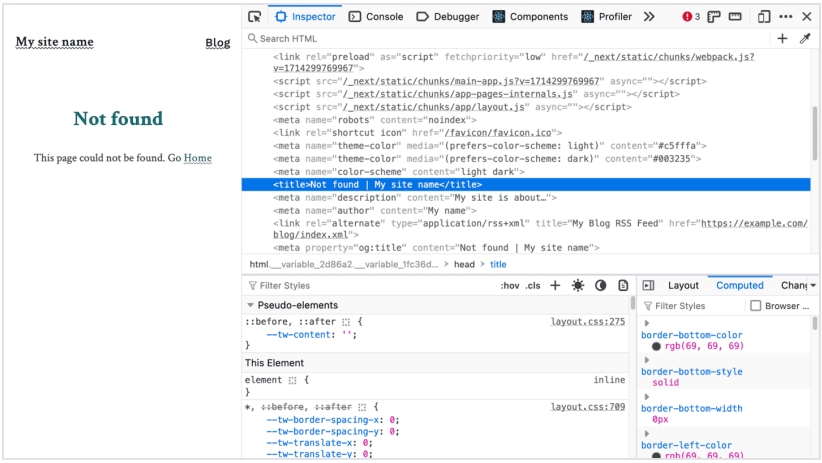
Whenever a reader gets lost on your site or bumps into an error, we want to help them recover and minimise the disruption.

To add a not-found page, you can create a file called `src/app/not-found.tsx`:

```
import Link from "next/link";
import type { Metadata } from "next/
types";

export const metadata: Metadata = {
  title: "Not found",
};

export default function NotFound() {
  return (
    <div className="my-24 flex items-
```

```
center justify-center text-center">
      <div className="prose prose-xl
dark:prose-invert">
        <h1 className="mr-4
text-4xl">Not found</h1>
        <p className="text-md">
          This page could not be
found. Go{" "}
          <Link className="tracking-
wide text-teal-700" href="/">
            Home
          </Link>
        </p>
      </div>
    </div>
  );
}
```

When you visit a misspelled URL or otherwise missing page like `http://localhost:3000/misspelled-url`, you should see a page like this:



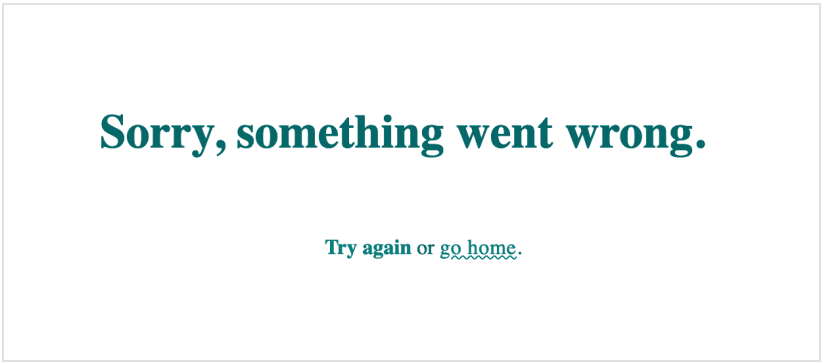
An unoriginal 404 page

We'll also add a global-error page to catch all other errors. Create a file called `src/app/global-error.tsx`:

```
"use client";

export default function GlobalError({
  error,
  reset,
}: {
```

```
  error: Error & { digest?: string };
  reset: () => void;
}) {
  console.error(error);

  return (
    <html
      lang="en"
      className="bg-white font-serif
text-neutral-800 dark:bg-neutral-950"
    >
      <body>
        <div className="prose
dark:prose-invert mx-auto flex min-h-
screen flex-col items-center justify-
center text-center">
          <h1 className="mr-4 inline-
block pr-4 text-4xl font-semibold">
            Sorry, something went
wrong.
          </h1>
          <div className="inline-
block">
            <p className="text-md">
              <button
                className="font-
semibold text-teal-700"
```

```
                onClick={() =>
reset()}
              >
                Try again
              </button>{" "}
              or{" "}
              <a className="tracking-
wide text-teal-700" href="/">
                go home
              </a>
              .
            </p>
          </div>
        </div>
      </body>
    </html>
  );
}
```



A bland error page

It can be a little tricky to test the global error page. With a static export, you might be able to test it by running `npm run build && npx http-server ./out/` and visiting `http://localhost:8080//bad-route` (with a double slash).

Open graph images

Open graph images are used by social media platforms to show a preview of a link.

To specify an open graph image, we can either:

- use `opengraph-image` file convention of naming a preview image `opengraph-image.png` (or with other extensions), or
- add an `openGraph` object to the `metadata` object.

In both cases, we can set a default at the site level and override it at each route segment or in each page.

We'll use a metadata object and set a default image in `src/app/layout.tsx`:

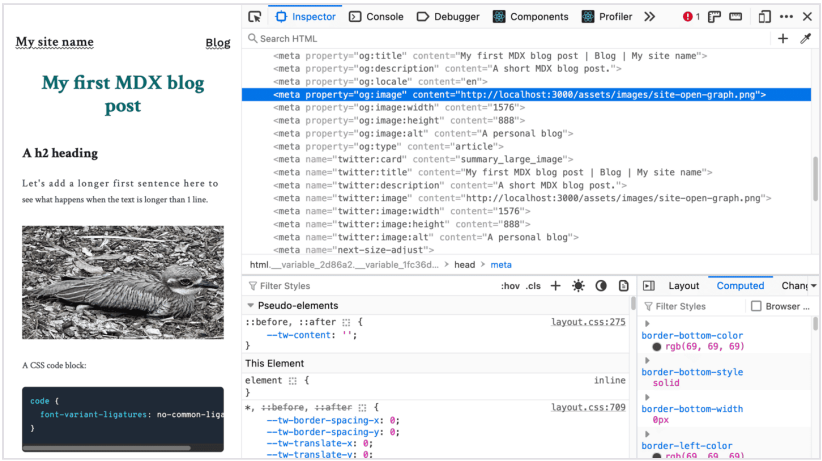
```
export const metadata: Metadata = {
  //
  metadataBase: new URL("https://
example.com"),
  openGraph: {
    images: [
      {
        url: "/assets/images/site-
open-graph.png", // Must be an
absolute URL or use metadataBase with
a relative URL
        width: 1576,
        height: 888,
        alt: "A personal blog",
      },
    ],
    locale: "en",
    type: "website",
  },
};
```

Note that we set a `metadataBase` in order to conveniently use a relative path for the image. In this case, the image is stored at `public/assets/images/site-open-graph.png`.

Each `openGraph` object set in a page lower in the site hierarchy overrides the parent object, so if you set other `openGraph` details, you'll need to include the default image again e.g. in `src/app/blog/layout.tsx`:

```
export const metadata: Metadata = {
  title: {
    template: `%s | Blog | My site
name`,
    default: `Blog`,
  },
  description: "This blog is about…",
  openGraph: {
    images: [
      {
        url: "/assets/images/site-
open-graph.png", // Must be an
absolute URL or use metadataBase with
a relative URL
        width: 1576,
        height: 888,
        alt: "A personal blog",
      },
    ],
    locale: "en",
    type: "article",
  },
};
```

Check the `<meta>` tags in the `<head>` of your site to see if the open graph image is set correctly:



Specifying the openGraph images set all the og:image and twitter:image meta tags

You now have a working default open graph image.

If you want to specify a particular image for a given blog post, you can add an `openGraph` object to the `metadata` object in that post's `.mdx` file:

```
export const metadata = {
  title: "My first MDX blog post",
  description: "A short MDX blog
post.",
  openGraph: {
    images: [
      {
        url: "/assets/images/
curlew.jpg",
        width: 1600,
        height: 900,
        alt: "A cryptic motionless
bush stone-curlew snuggled into wood
chips",
      },
    ],
    locale: "en",
    type: "article",
  },
};
```

Conclusion: Production

Production

When your site is nearly ready, review the Next.js production checklist.

Spell check

Before sharing your posts with the world, use a spell checker or even a fancier grammar checker. You can use a tool like Grammarly or Hemingway. In your editor, you can use a spell checker like Code Spell Checker for VS Code.

Lint and format

Once you're satisfied, lint and format your code:

```
npm run lint --fix && npm run format
```

Review any warnings or errors and commit any fixes.

Build

Build your site and review it locally:

```
npm run build && npx http-server ./
out/
```

Deployment

Given the static export, you can deploy your static Next.js site on a variety of platforms.

You could use Vercel from the creators of Next.js, Netlify, Cloudflare pages, GitHub Pages, AWS S3, Google Cloud Storage, Azure Blob Storage, or others.

Alternatively, you could use Virtual Private Server (VPS) options, such as DigitalOcean, or shared hosting options, such as NameCheap. In those cases, you can upload (e.g. using `rsync`) the `out` directory.

After deploying the static export of your site from the `out` directory, you can visit your site on the Internet and see your hard work in action. Check that the 404 page is configured correctly and subscribe to your RSS feed.

Celebrate!

Great work! You've built a blog with Next.js!

You've created a static blog using the latest Next.js and web features, including an RSS feed, a favicon, dark mode, Google Analytics, a sitemap, and Open Graph images. You should be proud of your efforts!

If you've found this book useful, please consider sharing the book, supporting me on Ko-fi, or following me on social media.

Next steps

Using what you've learned, you can continue to build out any new features that you like, such as an About or Contact page. You could add client React components to make your site and blog posts more interactive. Maybe you'd like to link to your social media profiles or add a newsletter subscription form. You can style your site any way you like to make it your very own.

To learn more Next.js from Vercel, check out Learn Next.js. To keep up to date with Next.js, you can follow the Next.js blog or newsletter, the Next.js GitHub repository, or Next.js on social media.