OCTOBER 31, 2019 / #GATSBY

How to Build Your Coding Blog From Scratch Using Gatsby and MDX

Scott Spence

I have been a Gatsby user since around version 0 back in May 2017.

Back then, I was using a template called Lumen. It was just what I needed at the time. Since then I have gone from using a template to creating my blog.

Over the years I have built my own Progressive Disclosure of Complexity with Gatsby to where I am now.

What does that mean?

It means that although there are an awesome amount of Gatsby starters and themes out there to get you up and running in minutes, this post is going to focus on what you need to do to build your own blog. Starting with the most basic “Hello World!” to deploying your code to production.

React components in Markdown goodness), so you will be able to add your own React components into your Markdown posts.

There'll be:

- Adding a Layout
- Basic styling with styled-components
- Code blocks with syntax highlighting
- Copy code snippet to clipboard
- Cover images for the posts
- Configuring an SEO component
- Deploying it to Netlify

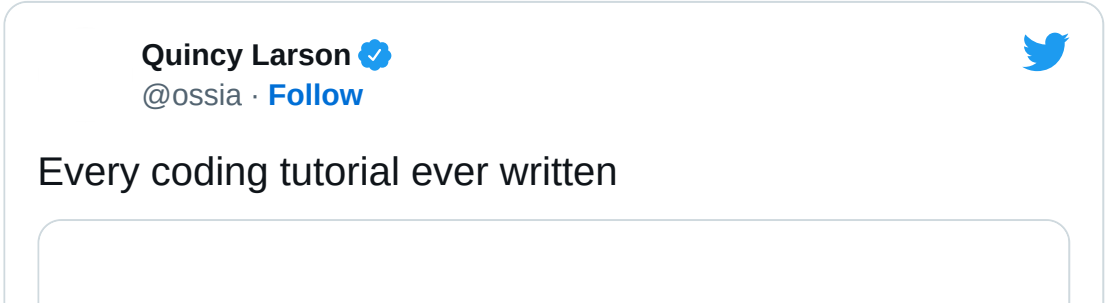
Who is this how-to for?

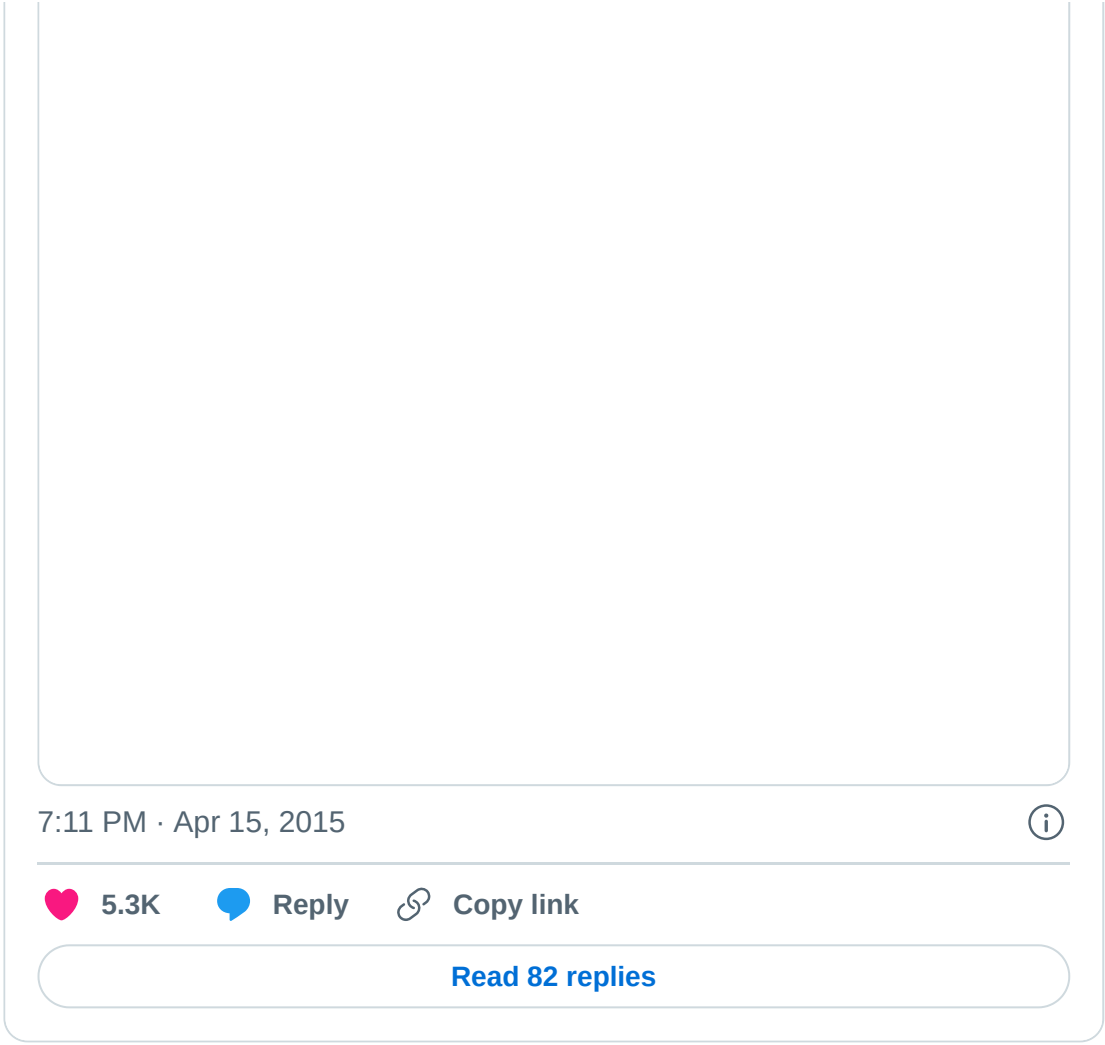
People that may have used Gatsby before as a template and now want to get more involved in how to make changes.

If you want to have code syntax highlighting.

If you want to use styled-components in an app.

I really want to avoid this!





Requirements

You're going to need a basic web development setup: node, terminal (bash, zsh or fish) and a text editor.

I do like to use codesandbox.io for these sort of guides to reduce the barrier to entry but in this case I have found there are some limitations with starting out from scratch on codesandbox.io which doesn't make this possible.

I have made a guide on getting set up for web development with Windows Web-Dev Bootstrap and covered the same process in

Hello World

To kick this off with the Gatsby 'hello world', you'll need to initialise the project with:

```
npm init -y
git init
```

I suggest that you commit this code to a git repository, so you should start with a `.gitignore` file.

```
touch .gitignore

echo "# Project dependencies
.cache
node_modules

# Build directory
public

# Other
.DS_Store
yarn-error.log" > .gitignore
```

Ok now is a good time to do a `git init` and if you're using VSCode you'll see the changes reflected in the sidebar.

basic hello world

Ok a Gatsby hello world, get started with the bare minimum! Install

```
yarn add gatsby react react-dom
```

You're going to need to create a pages directory and add an index file. You can do that in the terminal by typing the following:

```
# -p is to create parent directories too if needed
mkdir -p src/pages
touch src/pages/index.js
```

Now you can commence the hello word incantation! In the newly created `index.js` enter the following:

```
import React from 'react';

export default () => {
  return <h1>Hello World!</h1>;
};
```

Now you need to add the Gatsby develop script to the `package.json` file, `-p` specifies what port you want to run the project on and `-o` opens a new tab on your default browser, so in this case `localhost:9988` :

```
"dev": "gatsby develop -p 9988 -o"
```

And now it's time to run the code! From the terminal type the npm

```
yarn dev
```

> Note I'm using Yarn for installing all my dependencies and running scripts. If you prefer you can use npm, just bear in mind that the content on here uses yarn, so swap out commands where needed.

And with that the "Hello World" incantation is complete ?!

Add content

Now you have the base for your blog you're going to want to add some content. First up we're going to get the convention out of the way. For this how-to, the date format will be logical – the most logical way for a date format is **YYYYMMDD**, fight me!

So you're going to structure your posts content in years. In each one of those you're going to have another folder relating to the post

You could drill into this further if you like by separating out months and days. Depending on the volume of posts you've got going this may be a good approach. In this case and in the examples provided the convention detailed will be used.

```
# create multiple directories using curly braces
mkdir -p posts/2019/{2019-06-01-hello-world,2019-06-10-second-po
touch posts/2019/2019-06-01-hello-world/index.mdx
touch posts/2019/2019-06-10-second-post/index.mdx
touch posts/2019/2019-06-20-third-post/index.mdx
```

Ok that's how to set up your posts. Now you need to add some content to them, so each file you have in here should have frontmatter. Frontmatter is a way to assign properties to the contents, in this case a `title`, published `date` and a `published` flag (`true` or `false`).

```
---
title: Hello World - from mdx!
date: 2019-06-01
published: true
---

# h1 Heading

My first post!!

## h2 Heading

### h3 Heading
```

```
published: true
---

This is my second post!

#### h4 Heading

##### h5 Heading

###### h6 Heading
```

```
---
title: Third Post!
date: 2019-06-20
published: true
---

This is my third post!

> with a block quote!
```

Gatsby config API

Next, you're going to configure Gatsby so that it can read your super awesome content you just created. First up you need to create a the `gatsby-config.js` file. In the terminal create the file:

```
touch gatsby-config.js
```

Plugins

And now you can add the plugins Gatsby needs to use for sourcing

Gatsby source filesystem

The gatsby-source-filesystem collects the files on the local filesystem for use in Gatsby once configured.

Gatsby plugin MDX

The gatsby-plugin-mdx is what will be allowing us to write JSX in our Markdown documents and the heart of how the content is displayed in the blog.

Now is a good time to also add in dependent packages for the Gatsby plugin MDX which are `@mdx-js/mdx` and `@mdx-js/react`.

In the terminal install the dependencies:

```
yarn add gatsby-plugin-mdx @mdx-js/mdx @mdx-js/react gatsby-sour
```

Now it's time to configure `gatsby-config.js`:

```
module.exports = {
  siteMetadata: {
    title: `The Localhost Blog`,
    description: `This is my coding blog where I write about my
  },
  plugins: [
    {
      resolve: `gatsby-plugin-mdx`,
      options: {
        extensions: [`.mdx`, `.md`],
      },
    },
    {
```

```
      },
    },
  ],
};
```

Query data from GraphQL

Now you can see what the gatsby-source-filesystem and gatsby-plugin-mdx have done for us. You can now go to the Gatsby GraphQL GraphiQL explorer and check out the data:

```
{
  allMdx {
    nodes {
      frontmatter {
        title
        date
      }
    }
  }
}
```

Site Metadata

When you want to reuse common pieces of data across the site (for example, your site title), you can store that data in `siteMetadata`. You touched on this when defining the `gatsby-config.js`, and now you're going to separate this out from the `module.exports`. Why? It will be nicer to reason about once the config is filled with plugins.

At the top of `gatsby-config.js` add a new object variable for the site metadata:

```
const siteMetadata = {
  title: `The Localhost Blog`,
  description: `This is my coding blog where I write about my co
};
```

Now query the Site Metadata with GraphQL.

```
{
  site {
    siteMetadata {
      title
      description
    }
  }
}
```

Site metadata hook

Create a folder to keep all your hooks in and create a file for our hook. In the terminal do:

```
mkdir src/hooks
touch src/hooks/useSiteMetadata.js
```

Ok, and in your newly created file were going to use the Gatsby `useStaticQuery` hook to make your own hook:

```
import { graphql, useStaticQuery } from 'gatsby';

export const useSiteMetadata = () => {
  const { site } = useStaticQuery(
    graphql`
      query SITE_METADATA_QUERY {
        site {
          siteMetadata {
            title
            description
          }
        }
      }
    `
  );
  return site.siteMetadata;
};
```

Now you can use this hook anywhere in your site, so do that now in `src/pages/index.js`:

```
import React from 'react';
```

```
  return (
    <>
      <h1>{title}</h1>
      <p>{description}</p>
    </>
  );
};
```

Styling

You're going to use styled-components for styling. Styled-components (for me) help with scoping styles in your components. Time to go over the basics now.

install styled-components

```
yarn add gatsby-plugin-styled-components styled-components babel
```

The babel plugin is for automatic naming of components to help with debugging.

The Gatsby plugin is for built-in server-side rendering support.

Configure

Ok, with that detailed explanation out of the way, configure them in `gatsby-config.js`:

```
const siteMetadata = {
  title: `The Localhost Blog`,
  description: `This is my coding blog where I write about my co
};

module.exports = {
  siteMetadata: siteMetadata,
  plugins: [
    `gatsby-plugin-styled-components`,
    {
      resolve: `gatsby-plugin-mdx`,
      options: {
        extensions: [`.mdx`, `.md`],
      },
    },
    {
      resolve: `gatsby-source-filesystem`,
      options: { path: `${__dirname}/posts`, name: `posts` },
    },
  ],
};
```

Time to go over a styled component. In `index.js` you're going to `import styled from 'styled-components'` and create a `StyledH1` variable.

previously.

For this example make it the now iconic Gatsby `rebeccapurple` .

```
import React from 'react';
import styled from 'styled-components';
import { useSiteMetadata } from '../hooks/useSiteMetadata';

const StyledH1 = styled.h1`
  color: rebeccapurple;
`;

export default () => {
  const { title, description } = useSiteMetadata();
  return (
    <>
      <StyledH1>{title}</StyledH1>
      <p>{description}</p>
    </>
  );
};
```

That is styled-components on a very basic level. Basically create the styling you want for your page elements you're creating in the JSX.

Create a Developer Blog with Gatsby and MDX - styling wit...

Layout

Gatsby doesn't apply any layouts by default but instead uses the way you can compose React components for the layout. This means it's up to you how you want to layout what you're building with Gatsby.

In this guide we're going to initially create a basic layout component that you'll add to as you go along. For more detail on layout components take a look at the Gatsby layout components page.

Now you're going to refactor the home page (`src/pages/index.js`) a little and make some components for your blog layout and header. In the terminal create a components directory and a `Header` and `Layout` component:

```
mkdir src/components
touch src/components/Header.js src/components/Layout.js
```

Now to move the title and description from `src/pages/index.js` to the newly created `src/components/Header.js` component, destructuring props for the `siteTitle` and `siteDescription` , you'll pass these from the `Layout` component to here. You're going to add Gatsby Link to this so users can click on the header to go back to the home page.

```
import { Link } from 'gatsby';
import React from 'react';
```

```
  </Link>
);
```

Now to the Layout component: this is going to be a basic wrapper component for now. You're going to use your site metadata hook for the title and description and pass them to the header component and return the children of the wrapper (`Layout`).

```
import React from 'react';
import { useSiteMetadata } from '../hooks/useSiteMetadata';
import { Header } from './Header';

export const Layout = ({ children }) => {
  const { title, description } = useSiteMetadata();
  return (
    <>
      <Header siteTitle={title} siteDescription={description} />
      {children}
    </>
  );
};
```

Now to add the slightest of styles for some alignment for `src/components/Layout.js` , create an `AppStyles` styled component and make it the main wrapper of your `Layout` .

```
import React from 'react';
import styled from 'styled-components';
import { useSiteMetadata } from '../hooks/useSiteMetadata';
import { Header } from './Header';

const AppStyles = styled.main`
  width: 800px;
  margin: 0 auto;
```

```
  return (
    <AppStyles>
      <Header siteTitle={title} siteDescription={description} />
      {children}
    </AppStyles>
  );
};
```

Ok, now refactor your homepage (`src/pages/index.js`) with `Layout` .

```
import React from 'react';
import { Layout } from '../components/Layout';

export default () => {
  return (
    <>
      <Layout />
    </>
  );
};
```

Index page posts query

Now you can take a look at getting some of the posts you've created added to the index page of your blog. You're going to do that by creating a GraphQL query to list out the posts by title, order by date, and add an excerpt of the post.

The query will look something like this:

```
{
  allMdx {
    nodes {
      id
      excerpt(pruneLength: 250)
      frontmatter {
        title
        date
      }
    }
  }
}
```

If you put that into the GraphiQL GUI, though, you'll notice that the posts aren't in any given order. So now add a sort to this - and you'll also add in a filter for posts that are marked as published or not.

```
{
  allMdx(
    sort: { fields: [frontmatter___date], order: DESC }
    filter: { frontmatter: { published: { eq: true } } }
  ) {
    nodes {
      id
      excerpt(pruneLength: 250)
      frontmatter {
```

```
  }
}
```

On the homepage (`src/pages/index.js`) you're going to use the query we just put together to get a list of published posts in date order; add the following to the `index.js` file:

```
import { graphql } from 'gatsby';
import React from 'react';
import { Layout } from '../components/Layout';

export default ({ data }) => {
  return (
    <>
      <Layout>
        {data.allMdx.nodes.map(({ excerpt, frontmatter }) => (
          <>
            <h1>{frontmatter.title}</h1>
            <p>{frontmatter.date}</p>
            <p>{excerpt}</p>
          </>
        ))}
      </Layout>
    </>
  );
};

export const query = graphql`
  query SITE_INDEX_QUERY {
    allMdx(
      sort: { fields: [frontmatter___date], order: DESC }
      filter: { frontmatter: { published: { eq: true } } }
    ) {
      nodes {
        id
        excerpt(pruneLength: 250)
        frontmatter {
          title
          date
        }
```

Woah! WTF was all that yo!?

Ok, you're looping through the data passed into the component via the GraphQL query. Gatsby `graphql` runs the query (`SITE_INDEX_QUERY`) at runtime and gives us the results as props to your component via the `data` prop.

Slugs and Paths

Gatsby source filesystem will help with the creation of slugs (URL paths for the posts you're creating). In Gatsby node you're going to create the slugs for your posts.

First up you're going to need to create a `gatsby-node.js` file:

This will create the file path (URL) for each of the blog posts.

You're going to be using the Gatsby Node API `onCreateNode` and destructuring out `node`, `actions` and `getNode` for use in creating the file locations and associated value.

```
const { createFilePath } = require(`gatsby-source-filesystem`);

exports.onCreateNode = ({ node, actions, getNode }) => {
  const { createNodeField } = actions;
  if (node.internal.type === `Mdx`) {
    const value = createFilePath({ node, getNode });
    createNodeField({
      name: `slug`,
      node,
      value,
    });
  }
};
```

Now to help visualise some of the data being passed into the components you're going to use Dump.js for debugging the data. Thanks to Wes Bos for the super handy Dump.js component.

To get the component set up, create a `Dump.js` file in your `src\components` folder and copypasta the code from the linked GitHub page.

```
touch /src/components/Dump.js
```

```
  <div
    style={{
      fontSize: 20,
      border: '1px solid #efefef',
      padding: 10,
      background: 'white',
    }}>
    {Object.entries(props).map(([key, val]) => (
      <pre key={key}>
        <strong style={{ color: 'white', background: 'red' }}>
          {key} ?
        </strong>
        {JSON.stringify(val, '', ' ')}
      </pre>
    ))}
  </div>
);

export default Dump;
```

Now you can use the `Dump` component anywhere in your project. To demonstrate, use it with the index page `data` to see the output.

So in the `src/pages/index.js` you're going to import the Dump component and pass in the `data` prop and see what the output looks like.

```
import { graphql } from 'gatsby';
import React from 'react';
import Dump from '../components/Dump';
import { Layout } from '../components/Layout';

export default ({ data }) => {
  return (
    <>
      <Layout>
        <Dump data={data} />
        {data.allMdx.nodes.map(({ excerpt, frontmatter }) => (
```

```
            </>
          ))}
        </Layout>
      </>
    );
};

export const query = graphql`
  query SITE_INDEX_QUERY {
    allMdx(
      sort: { fields: [frontmatter___date], order: DESC }
      filter: { frontmatter: { published: { eq: true } } }
    ) {
      nodes {
        id
        excerpt(pruneLength: 250)
        frontmatter {
          title
          date
        }
      }
    }
  }
`;
```

Now you've created the paths you can link to them with Gatsby Link. First you'll need to add the slug to your `SITE_INDEX_QUERY` Then you can add gatsby `Link` to `src/pages/index.js`.

You're also going to create some styled-components for wrapping the list of posts and each individual post as well.

```
import { graphql, Link } from 'gatsby';
import React from 'react';
import styled from 'styled-components';
import { Layout } from '../components/Layout';

const IndexWrapper = styled.main``;

const PostWrapper = styled.div``;

export default ({ data }) => {
  return (
    <Layout>
      <IndexWrapper>
        {data.allMdx.nodes.map(
          ({ id, excerpt, frontmatter, fields }) => (
            <PostWrapper key={id}>
              <Link to={fields.slug}>
                <h1>{frontmatter.title}</h1>
                <p>{frontmatter.date}</p>
                <p>{excerpt}</p>
              </Link>
            </PostWrapper>
          )
        )}
      </IndexWrapper>
    </Layout>
  );
};

export const query = graphql`
  query SITE_INDEX_QUERY {
    allMdx(
      sort: { fields: [frontmatter___date], order: DESC }
```

```
        excerpt(pruneLength: 250)
        frontmatter {
          title
          date
        }
        fields {
          slug
        }
      }
    }
  }
`;
```

Adding a Blog Post Template

Now you have the links pointing to the blog posts you currently have no file associated with the path. This means that clicking a link will give you a 404 and the built-in gatsby 404 will list all the pages available in the project, currently only the `/` index/homepage.

So, for each one of your blog posts you're going to use a template that will contain the information you need to make up your blog post. To start, create a `templates` directory and template file for that with:

```
mkdir -p src/templates
touch src/templates/blogPostTemplate.js
```

For now you're going to scaffold out a basic template (you'll be adding data to this shortly):

```
      <>
        <p>post here</p>
      </>
    );
  };
```

To populate the template you'll need to use Gatsby node to create your pages.

Gatsby Node has many internal APIs available to us. For this example you're going to be using the `createPages` API.

More info on Gatsby `createPages` API can be found on the Gatsby docs, details here: https://www.gatsbyjs.org/docs/node-apis/#createPages

In your `gatsby-node.js` file you're going to add in the following in addition to the `onCreateNode` export you did earlier.

```
const { createFilePath } = require(`gatsby-source-filesystem`);
const path = require(`path`);

exports.createPages = ({ actions, graphql }) => {
  const { createPage } = actions;
  const blogPostTemplate = path.resolve(
    'src/templates/blogPostTemplate.js'
  );

  return graphql(`
    {
      allMdx {
        nodes {
          fields {
            slug
          }
          frontmatter {
```

```
    }
  `).then(result => {
    if (result.errors) {
      throw result.errors;
    }

    const posts = result.data.allMdx.nodes;

    // create page for each mdx file
    posts.forEach(post => {
      createPage({
        path: post.fields.slug,
        component: blogPostTemplate,
        context: {
          slug: post.fields.slug,
        },
      });
    });
  });
};

exports.onCreateNode = ({ node, actions, getNode }) => {
  const { createNodeField } = actions;
  if (node.internal.type === `Mdx`) {
    const value = createFilePath({ node, getNode });
    createNodeField({
      name: `slug`,
      node,
      value,
    });
  }
};
```

The part that you need to pay particular attention to right now is the `.forEach` loop where you're using the `createPage` function we destructured from the `actions` object.

This is where you pass the data needed by `blogPostTemplate` you defined earlier. You're going to be adding more to the `context` for

```
// create page for each mdx node
posts.forEach(post => {
  createPage({
    path: post.fields.slug,
    component: blogPostTemplate,
    context: {
      slug: post.fields.slug,
    },
  });
});
```

Build out Blog Post Template

Now you're going to take the context information passed to the `blogPostTemplate.js` to make the blog post page.

This is similar to the `index.js` homepage, whereas there's GraphQL data used to create the page. But in this instance the template uses a variable (also known as a parameter or an identifier) so you can query data specific to that given variable.

```
query PostBySlug($slug: String!) {
  mdx(fields: { slug: { eq: $slug } }) {
    frontmatter {
      title
      date(formatString: "YYYY MMMM Do")
    }
  }
}
```

Here you're defining the variable as slug with the `$` denoting that it's a variable. You also need to define the variable type as (in this case) `String!`. The exclamation after the type means that it has to be a string being passed into the query.

Using `mdx` you're going to filter on `fields` where the `slug` matches the variable being passed into the query.

Running the query now will show an error as there's no variable being fed into the query. If you look to the bottom of the query pane you should notice `QUERY VARIABLES`. Click on that to bring up the variables pane.

This is where you can add in one of the post paths you created earlier. If you have your dev server up and running, go to one of the posts and take the path and paste it into the quotes `""` and try running the query again.

```
{
  "slug": "/2019/2019-06-20-third-post/"
}
```

the query and have that at the bottom of your page file.

Right now you're going to create a simple react component that will display the data you have queried.

Destructuring the `frontmatter` and `body` from the GraphQL query, you'll get the Title and the Data from the frontmatter object and wrap the `body` in the `MDXRenderer` .

```
import { graphql } from 'gatsby';
import { MDXRenderer } from 'gatsby-plugin-mdx';
import React from 'react';
import { Layout } from '../components/Layout';

export default ({ data }) => {
  const { frontmatter, body } = data.mdx;
  return (
    <Layout>
      <h1>{frontmatter.title}</h1>
      <p>{frontmatter.date}</p>
      <MDXRenderer>{body}</MDXRenderer>
    </Layout>
  );
};

export const query = graphql`
  query PostsBySlug($slug: String!) {
    mdx(fields: { slug: { eq: $slug } }) {
      body
      frontmatter {
        title
        date(formatString: "YYYY MMMM Do")
      }
    }
  }
`;
```

If you haven't done so already now would be a good time to restart

template in all it's basic glory!

Previous and Next navigation

Coolio! Now you have your basic blog where you can list available posts and click a link to see the full post in a predefined template. Once you're in a post you have to navigate back to the home page to pick out a new post to read. In this section you're going to work on adding in some previous and next navigation.

Remember the `.forEach` snippet you looked at earlier? That's where you're going to pass some additional context to the page by selecting out the previous and next posts.

```
// create page for each mdx node
posts.forEach((post, index) => {
  const previous =
    index === posts.length - 1 ? null : posts[index + 1];
  const next = index === 0 ? null : posts[index - 1];
```

```
    context: {
      slug: post.fields.slug,
      previous,
      next,
    },
  });
});
```

So this should now match up with the query you have on the homepage (`src/pages/index.js`) except you currently have no filter or sort applied here. So do that now in `gatsby-node.js` and apply the same filters as on the homepage query:

```
const { createFilePath } = require(`gatsby-source-filesystem`);
const path = require(`path`);

exports.createPages = ({ actions, graphql }) => {
  const { createPage } = actions;
  const blogPostTemplate = path.resolve(
    'src/templates/blogPostTemplate.js'
  );

  return graphql(`
    {
      allMdx(
        sort: { fields: [frontmatter___date], order: DESC }
        filter: { frontmatter: { published: { eq: true } } }
      ) {
        nodes {
          fields {
            slug
          }
          frontmatter {
            title
          }
        }
      }
    }
  `).then(result => {
```

```
    const posts = result.data.allMdx.nodes;

    // create page for each mdx node
    posts.forEach((post, index) => {
      const previous =
        index === posts.length - 1 ? null : posts[index + 1];
      const next = index === 0 ? null : posts[index - 1];

      createPage({
        path: post.fields.slug,
        component: blogPostTemplate,
        context: {
          slug: post.fields.slug,
          previous,
          next,
        },
      });
    });
  });
};

exports.onCreateNode = ({ node, actions, getNode }) => {
  const { createNodeField } = actions;
  if (node.internal.type === `Mdx`) {
    const value = createFilePath({ node, getNode });
    createNodeField({
      name: `slug`,
      node,
      value,
    });
  }
};
```

Now you will be able to expose the `previous` and `next` objects passed in as context from Gatsby node.

You can destructure `previous` and `next` from `pageContext` and for now pop them into your super handy `Dump` component to take a look at their contents.

```
import React from 'react';
import Dump from '../components/Dump';
import { Layout } from '../components/Layout';

export default ({ data, pageContext }) => {
  const { frontmatter, body } = data.mdx;
  const { previous, next } = pageContext;
  return (
    <Layout>
      <Dump previous={previous} />
      <Dump next={next} />
      <h1>{frontmatter.title}</h1>
      <p>{frontmatter.date}</p>
      <MDXRenderer>{body}</MDXRenderer>
    </Layout>
  );
};

export const query = graphql`
  query PostsBySlug($slug: String!) {
    mdx(fields: { slug: { eq: $slug } }) {
      body
      frontmatter {
        title
        date(formatString: "YYYY MMMM Do")
      }
    }
  }
`;
```

Add in previous and next navigation, this is a couple of ternary operations. If the variable is empty then return `null` else render a Gatsby `Link` component with the page slug and the frontmatter title:

```
import { graphql, Link } from 'gatsby';
import { MDXRenderer } from 'gatsby-plugin-mdx';
import React from 'react';
import Dump from '../components/Dump';
import { Layout } from '../components/Layout';
```

```
  return (
    <Layout>
      <Dump previous={previous} />
      <Dump next={next} />
      <h1>{frontmatter.title}</h1>
      <p>{frontmatter.date}</p>
      <MDXRenderer>{body}</MDXRenderer>
      {previous === false ? null : (
        <>
          {previous && (
            <Link to={previous.fields.slug}>
              <p>{previous.frontmatter.title}</p>
            </Link>
          )}
        </>
      )}
      {next === false ? null : (
        <>
          {next && (
            <Link to={next.fields.slug}>
              <p>{next.frontmatter.title}</p>
            </Link>
          )}
        </>
      )}
    </Layout>
  );
};

export const query = graphql`
  query PostsBySlug($slug: String!) {
    mdx(fields: { slug: { eq: $slug } }) {
      body
      frontmatter {
        title
        date(formatString: "YYYY MMMM Do")
      }
    }
  }
`;
```

Code Blocks

Now to add some syntax highlighting for adding code blocks to your blog pages. To do that you're going to add dependencies for prism-react-renderer and react-live. You'll also create the files you're going to need to use:

```
yarn add prism-react-renderer react-live
touch root-wrapper.js gatsby-ssr.js gatsby-browser.js
```

You'll come onto `react-live` soon. For now, you're going to get `prism-react-render` up and running for syntax highlighting for any code you're going to add to the blog. But before that you're going to go over the root wrapper concept.

So, to change the rendering of a page element, such as a heading or a code block, you're going to need to use the `MDXProvider` . The `MDXProvider` is a component you can use anywhere higher in the React component tree than the MDX content you want to render.

Gatsby browser and a Gatsby SSR both have `wrapRootElement`

`browser.js` and `gatsby-ssr.js` so you're not duplicating code.

Before you go any further I want to add that there is a top quality egghead.io playlist resource for using MDX with Gatsby by Chris Chris Biscardi. There's a ton of useful information in there on MDX in Gatsby.

Ok, first up you're going to import the `root-wrapper.js` file into both `gatsby-browser.js` and `gatsby-ssr.js`. Into both code modules paste the following:

```
import { wrapRootElement as wrap } from './root-wrapper';

export const wrapRootElement = wrap;
```

Ok, now you can work on the code that will be used in both modules. MDX allows you to control the rendering of page elements in your markdown. `MDXProvider` is used to give React components to override the markdown page elements.

Quick demonstration, in `root-wrapper.js` add the following:

```
import { MDXProvider } from '@mdx-js/react';
import React from 'react';

const components = {
  h2: ({ children }) => (
    <h2 style={{ color: 'rebeccapurple' }}>{children}</h2>
  ),
  'p.inlineCode': props => (
    <code style={{ backgroundColor: 'lightgray' }} {...props} />
  ),
```

```
);
```

You're now overriding any `h2` in your rendered markdown along with any `code` blocks (that's words wrapped in `` `backticks` ``).

Ok, now for the syntax highlighting, create a post with a block of code in it:

```
mkdir posts/2019-07-01-code-blocks
touch posts/2019-07-01-code-blocks/index.mdx
```

Paste in some content:

```
---
```

````
## Yes! Some code!

Here is the `Dump` component!

```jsx
import React from 'react';

const Dump = props => (
  <div
    style={{
      fontSize: 20,
      border: '1px solid #efefef',
      padding: 10,
      background: 'white',
    }}>
    {Object.entries(props).map(([key, val]) => (
      <pre key={key}>
        <strong style={{ color: 'white', background: 'red' }}>
          {key} ?
        </strong>
        {JSON.stringify(val, '', ' ')}
      </pre>
    ))}
  </div>
);

export default Dump;
```
````

Go to the prism-react-renderer GitHub page and copy the example code into `root-wrapper.js` for the `pre` element.

You're going to copy the provided code for highlighting to validate that it works.

```
import { MDXProvider } from '@mdx-js/react';
import Highlight, { defaultProps } from 'prism-react-renderer';
```

```
    <h2 style={{ color: 'rebeccapurple' }}>{children}</h2>
  ),
  'p.inlineCode': props => (
    <code style={{ backgroundColor: 'lightgray' }} {...props} />
  ),
  pre: props => (
    <Highlight
      {...defaultProps}
      code={`
        (function someDemo() {
          var test = "Hello World!";
          console.log(test);
        })();

        return () => <App />;
      `}
      language="jsx">
      {({
        className,
        style,
        tokens,
        getLineProps,
        getTokenProps,
      }) => (
        <pre className={className} style={style}>
          {tokens.map((line, i) => (
            <div {...getLineProps({ line, key: i })}>
              {line.map((token, key) => (
                <span {...getTokenProps({ token, key })} />
              ))}
            </div>
          ))}
        </pre>
      )}
    </Highlight>
  ),
};

export const wrapRootElement = ({ element }) => (
  <MDXProvider components={components}>{element}</MDXProvider>
);
```

```
import { MDXProvider } from '@mdx-js/react';
import Highlight, { defaultProps } from 'prism-react-renderer';
import React from 'react';

const components = {
  pre: props => (
    <Highlight
      {...defaultProps}
      code={props.children.props.children.trim()}
      language="jsx">
      {({
        className,
        style,
        tokens,
        getLineProps,
        getTokenProps,
      }) => (
        <pre className={className} style={style}>
          {tokens.map((line, i) => (
            <div {...getLineProps({ line, key: i })}>
              {line.map((token, key) => (
                <span {...getTokenProps({ token, key })} />
              ))}
            </div>
          ))}
        </pre>
      )}
    </Highlight>
  ),
};

export const wrapRootElement = ({ element }) => (
  <MDXProvider components={components}>{element}</MDXProvider>
);
```

Then to match the language, for now you're going to add in a `matches` function to match the language class assigned to the code block.

```
import React from 'react';

const components = {
  h2: ({ children }) => (
    <h2 style={{ color: 'rebeccapurple' }}>{children}</h2>
  ),
  'p.inlineCode': props => (
    <code style={{ backgroundColor: 'lightgray' }} {...props} />
  ),
  pre: props => {
    const className = props.children.props.className || '';
    const matches = className.match(/language-(?<lang>.*)/);
    return (
      <Highlight
        {...defaultProps}
        code={props.children.props.children.trim()}
        language={
          matches && matches.groups && matches.groups.lang
            ? matches.groups.lang
            : ''
        }>
        {({
          className,
          style,
          tokens,
          getLineProps,
          getTokenProps,
        }) => (
          <pre className={className} style={style}>
            {tokens.map((line, i) => (
              <div {...getLineProps({ line, key: i })}>
                {line.map((token, key) => (
                  <span {...getTokenProps({ token, key })} />
                ))}
              </div>
            ))}
          </pre>
        )}
      </Highlight>
    );
  },
};

export const wrapRootElement = ({ element }) => (
```

prism-react-renderer comes with additional themes over the default theme which is duotoneDark. You're going to use nightOwl in this example, but feel free to take a look at the other examples if you like.

Import the `theme` then use it in the props of the `Highlight` component.

```
import { MDXProvider } from '@mdx-js/react';
import Highlight, { defaultProps } from 'prism-react-renderer';
import theme from 'prism-react-renderer/themes/nightOwl';
import React from 'react';

const components = {
  pre: props => {
    const className = props.children.props.className || '';
    const matches = className.match(/language-(?<lang>.*)/);

    return (
      <Highlight
        {...defaultProps}
        code={props.children.props.children.trim()}
        language={
          matches && matches.groups && matches.groups.lang
            ? matches.groups.lang
            : ''
        }
        theme={theme}>
        {({
          className,
          style,
          tokens,
          getLineProps,
          getTokenProps,
        }) => (
          <pre className={className} style={style}>
            {tokens.map((line, i) => (
              <div {...getLineProps({ line, key: i })}>
                {line.map((token, key) => (
```

```
          </pre>
        )}
      </Highlight>
    );
  },
};

export const wrapRootElement = ({ element }) => (
  <MDXProvider components={components}>{element}</MDXProvider>
);
```

Ok, now time to abstract this out into its own component so your `root-wrapper.js` isn't so crowded.

Make a `Code.js` component, and move the code from `root-wrapper.js` into there:

```
touch src/components/Code.js
```

> Cool, cool! Now you want to replace the pasted in code example with the props of the child component of the pre component. You can do that with `props.children.props.children.trim()` ?.

If that ☝ makes no real amount of sense for you (I've had to read it many, many times myself), don't worry: now you're going to dig into that a bit more for the creation of the code block component.

So, for now in the `components` you're adding into the `MDXProvider`, take a look at the `props` coming into the `pre` element.

Comment out the code you added earlier and add in a `console.log`:

```
pre: props => {
  console.log('======================');
  console.log(props);
  console.log('======================');
  return <pre />;
};
```

Now if you pop open the developer tools of your browser you can see the output.

```
{children: {…}}
  children:
    $$typeof: Symbol(react.element)
    key: null
    props: {parentName: "pre", className: "language-jsx", origin
    ref: null
    type: ƒ (re....
```

those props. If you take a look at the contents of that you will see that it is the code string for your code block. This is what you're going to be passing into the `Code` component you're about to create. Other properties to note here are the `className` and `mdxType`.

So, take the code you used earlier for `Highlight`, everything inside and including the `return` statement, and paste it into the `Code.js` module you created earlier.

`Highlight` requires several props:

```
<Highlight
  {...defaultProps}
  code={codeString}
  language={language}
  theme={theme}
>
```

The `Code` module should look something like this now:

```
import Highlight, { defaultProps } from 'prism-react-renderer';
import theme from 'prism-react-renderer/themes/nightOwl';
import React from 'react';

const Code = ({ codeString, language }) => {
  return (
    <Highlight
      {...defaultProps}
      code={codeString}
      language={language}
      theme={theme}>
      {({
        className,
```

```
        }) => (
          <pre className={className} style={style}>
            {tokens.map((line, i) => (
              <div {...getLineProps({ line, key: i })}>
                {line.map((token, key) => (
                  <span {...getTokenProps({ token, key })} />
                ))}
              </div>
            ))}
          </pre>
        )}
      </Highlight>
    );
  };

  export default Code;
```

Back to the `root-wrapper` where you're going to pass the `props` needed to the `Code` component.

The first check you're going to do is if the `mdxType` is `code` then you can get the additional props you need to pass to your `Code` component.

You're going to get `defaultProps` and the `theme` from `prism-react-renderer` so all that's needed is the `code` and `language`.

The `codeString` you can get from the `props`, and the `children` by destructuring from the `props` being passed into the `pre` element. The `language` can either be the tag assigned to the meta property of the backticks, like `js`, `jsx` or equally empty. So you check for that with some JavaScript and also remove the `language-` prefix, then pass in the elements `{...props}`:

```
          codeString={props.children.trim()}
          language={
            props.className && props.className.replace('language-'
          }
          {...props}
        />
      );
    }
  };
```

Ok, now you're back to where you were before abstracting out the `Highlight` component to it's own module. Add some additional styles with `styled-components` and replace the `pre` with a styled `Pre`. You can also add in some line numbers with a styled span and style that as well.

```
import Highlight, { defaultProps } from 'prism-react-renderer';
import theme from 'prism-react-renderer/themes/nightOwl';
import React from 'react';
```

```
  margin: 1em 0;
  padding: 0.5em;
  overflow-x: auto;
  border-radius: 3px;

  & .token-line {
    line-height: 1.3em;
    height: 1.3em;
  }
  font-family: 'Courier New', Courier, monospace;
`;

export const LineNo = styled.span`
  display: inline-block;
  width: 2em;
  user-select: none;
  opacity: 0.3;
`;

const Code = ({ codeString, language, ...props }) => {
  return (
    <Highlight
      {...defaultProps}
      code={codeString}
      language={language}
      theme={theme}>
      {({
        className,
        style,
        tokens,
        getLineProps,
        getTokenProps,
      }) => (
        <Pre className={className} style={style}>
          {tokens.map((line, i) => (
            <div {...getLineProps({ line, key: i })}>
              <LineNo>{i + 1}</LineNo>
              {line.map((token, key) => (
                <span {...getTokenProps({ token, key })} />
              ))}
            </div>
          ))}
        </Pre>
      )}
    </Highlight>
```

Copy code to clipboard

What if you had some way of getting that props code string into the clipboard?

I had a look around and found the majority of the components available for this sort of thing expected an input until this in the Gatsby source code. Which is creating the input for you ?

So, create a `utils` directory and the `copy-to-clipboard.js` file and add in the code from the Gatsby source code.

```
mkdir src/utils
touch src/utils/copy-to-clipboard.js
```

```
export const copyToClipboard = str => {
  const clipboard = window.navigator.clipboard;
  /*
   * fallback to older browsers (including Safari)
   * if clipboard API not supported
   */
  if (!clipboard || typeof clipboard.writeText !== `function`) {
    const textarea = document.createElement(`textarea`);
    textarea.value = str;
    textarea.setAttribute(`readonly`, true);
    textarea.setAttribute(`contenteditable`, true);
    textarea.style.position = `absolute`;
    textarea.style.left = `-9999px`;
    document.body.appendChild(textarea);
    textarea.select();
    const range = document.createRange();
    const sel = window.getSelection();
    sel.removeAllRanges();
    sel.addRange(range);
    textarea.setSelectionRange(0, textarea.value.length);
    document.execCommand(`copy`);
    document.body.removeChild(textarea);

    return Promise.resolve(true);
  }

  return clipboard.writeText(str);
};
```

Now you're going to want a way to trigger copying the code to the clipboard.

Let's create a styled button. But first add a `position: relative;` to the `Pre` component which will let us position the styled button:

```
const CopyCode = styled.button`
  position: absolute;
  right: 0.25rem;
```

```
  &:hover {
    opacity: 1;
  }
`;
```

And now you need to use the `copyToClipboard` function in the `onClick` of the button:

```
import Highlight, { defaultProps } from 'prism-react-renderer';
import theme from 'prism-react-renderer/themes/nightOwl';
import React from 'react';
import styled from 'styled-components';
import { copyToClipboard } from '../utils/copy-to-clipboard';

export const Pre = styled.pre`
  text-align: left;
  margin: 1rem 0;
  padding: 0.5rem;
  overflow-x: auto;
  border-radius: 3px;

  & .token-line {
    line-height: 1.3rem;
    height: 1.3rem;
  }
  font-family: 'Courier New', Courier, monospace;
  position: relative;
`;

export const LineNo = styled.span`
  display: inline-block;
  width: 2rem;
  user-select: none;
  opacity: 0.3;
`;

const CopyCode = styled.button`
  position: absolute;
  right: 0.25rem;
  border: 0;
```

```
    opacity: 1;
  }
`;

const Code = ({ codeString, language }) => {
  const handleClick = () => {
    copyToClipboard(codeString);
  };

  return (
    <Highlight
      {...defaultProps}
      code={codeString}
      language={language}
      theme={theme}>
      {({
        className,
        style,
        tokens,
        getLineProps,
        getTokenProps,
      }) => (
        <Pre className={className} style={style}>
          <CopyCode onClick={handleClick}>Copy</CopyCode>
          {tokens.map((line, i) => (
            <div {...getLineProps({ line, key: i })}>
              <LineNo>{i + 1}</LineNo>
              {line.map((token, key) => (
                <span {...getTokenProps({ token, key })} />
              ))}
            </div>
          ))}
        </Pre>
      )}
    </Highlight>
  );
};

export default Code;
```

React live

So with React Live you need to add two snippets to your `Code.js` component.

You're going to import the components:

```
import {
  LiveEditor,
  LiveError,
  LivePreview,
  LiveProvider,
} from 'react-live';
```

Then you're going to check if `react-live` has been added to the language tag on your mdx file via the props:

```
if (props['react-live']) {
  return (
    <LiveProvider code={codeString} noInline={true} theme={theme
      <LiveEditor />
      <LiveError />
      <LivePreview />
```

Here's the full component:

```
import Highlight, { defaultProps } from 'prism-react-renderer';
import theme from 'prism-react-renderer/themes/nightOwl';
import React from 'react';
import {
  LiveEditor,
  LiveError,
  LivePreview,
  LiveProvider,
} from 'react-live';
import styled from 'styled-components';
import { copyToClipboard } from '../../utils/copy-to-clipboard';

const Pre = styled.pre`
  position: relative;
  text-align: left;
  margin: 1em 0;
  padding: 0.5em;
  overflow-x: auto;
  border-radius: 3px;

  & .token-lline {
    line-height: 1.3em;
    height: 1.3em;
  }
  font-family: 'Courier New', Courier, monospace;
`;

const LineNo = styled.span`
  display: inline-block;
  width: 2em;
  user-select: none;
  opacity: 0.3;
`;

const CopyCode = styled.button`
  position: absolute;
  right: 0.25rem;
  border: 0;
  border-radius: 3px;
```

```
  }
`;

export const Code = ({ codeString, language, ...props }) => {
  if (props['react-live']) {
    return (
      <LiveProvider code={codeString} noInline={true} theme={the
        <LiveEditor />
        <LiveError />
        <LivePreview />
      </LiveProvider>
    );
  }

  const handleClick = () => {
    copyToClipboard(codeString);
  };

  return (
    <Highlight
      {...defaultProps}
      code={codeString}
      language={language}
      theme={theme}>
      {({
        className,
        style,
        tokens,
        getLineProps,
        getTokenProps,
      }) => (
        <Pre className={className} style={style}>
          <CopyCode onClick={handleClick}>Copy</CopyCode>
          {tokens.map((line, i) => (
            <div {...getLineProps({ line, key: i })}>
              <LineNo>{i + 1}</LineNo>
              {line.map((token, key) => (
                <span {...getTokenProps({ token, key })} />
              ))}
            </div>
          ))}
        </Pre>
      )}
    </Highlight>
```

To test this, add `react-live` next to the language on your `Dump` component, so you have added to the blog post you made:

````
```jsx react-live
````

Now you can edit the code directly. Try changing a few things like this:

```jsx react-live
const Dump = props => (
  <div
    style={{
      fontSize: 20,
      border: '1px solid #efefef',
      padding: 10,
      background: 'white',
    }}>
    {Object.entries(props).map(([key, val]) => (
      <pre key={key}>
        <strong style={{ color: 'white', background: 'red' }}>
          {key} ?
        </strong>
        {JSON.stringify(val, '', ' ')}
      </pre>
    ))}
  </div>
);

render(<Dump props={['One', 'Two', 'Three', 'Four']} />);
```
```

Cover Image

Now to add a cover image to go with each post, you'll need to install a couple of packages to manage images in Gatsby.

install:

```
yarn add gatsby-transformer-sharp gatsby-plugin-sharp gatsby-rem
```

Now you should config `gatsby-config.js` to include the newly added packages. Remember to add `gatsby-remark-images` to `gatsby-plugin-mdx` as both a `gatsbyRemarkPlugins` option and as a `plugins` option.

config:

```
module.exports = {
  siteMetadata: siteMetadata,
  plugins: [
    `gatsby-plugin-styled-components`,
    `gatsby-transformer-sharp`,
    `gatsby-plugin-sharp`,
```

```
        gatsbyRemarkPlugins: [
          {
            resolve: `gatsby-remark-images`,
            options: {
              maxWidth: 590,
            },
          },
        ],
        plugins: [
          {
            resolve: `gatsby-remark-images`,
            options: {
              maxWidth: 590,
            },
          },
        ],
      },
    },
    {
      resolve: `gatsby-source-filesystem`,
      options: { path: `${__dirname}/posts`, name: `posts` },
    },
  ],
};
```

Add image to index query in `src/pages.index.js` :

```
cover {
  publicURL
  childImageSharp {
    sizes(
      maxWidth: 2000
      traceSVG: { color: "#639" }
    ) {
      ...GatsbyImageSharpSizes_tracedSVG
    }
  }
}
```

```
date(formatString: "YYYY MMMM Do")
```

This will show the date as full year, full month and the day as a 'st', 'nd', 'rd' and 'th'. So if today's date were 1970/01/01 it would read 1970 January 1st.

Add `gatsby-image` use that in a styled component:

```
const Image = styled(Img)`
  border-radius: 5px;
`;
```

Add some JavaScript to determine if there's anything to render:

```
{
  !!frontmatter.cover ? (
    <Image sizes={frontmatter.cover.childImageSharp.sizes} />
  ) : null;
}
```

Here's what the full module should look like now:

```
import { Link } from 'gatsby';
import Img from 'gatsby-image';
import React from 'react';
import styled from 'styled-components';
import { Layout } from '../components/Layout';

const IndexWrapper = styled.main``;
```

```
  border-radius: 5px;
`;

export default ({ data }) => {
  return (
    <Layout>
      <IndexWrapper>
        {/* <Dump data={data}></Dump> */}
        {data.allMdx.nodes.map(
          ({ id, excerpt, frontmatter, fields }) => (
            <PostWrapper key={id}>
              <Link to={fields.slug}>
                {!!frontmatter.cover ? (
                  <Image
                    sizes={frontmatter.cover.childImageSharp.siz
                  />
                ) : null}
                <h1>{frontmatter.title}</h1>
                <p>{frontmatter.date}</p>
                <p>{excerpt}</p>
              </Link>
            </PostWrapper>
          )
        )}
      </IndexWrapper>
    </Layout>
  );
};

export const query = graphql`
  query SITE_INDEX_QUERY {
    allMdx(
      sort: { fields: [frontmatter___date], order: DESC }
      filter: { frontmatter: { published: { eq: true } } }
    ) {
      nodes {
        id
        excerpt(pruneLength: 250)
        frontmatter {
          title
          date(formatString: "YYYY MMMM Do")
          cover {
            publicURL
            childImageSharp {
              sizes(maxWidth: 2000, traceSVG: { color: "#639" })
```

```
        }
        fields {
          slug
        }
      }
    }
  }
`;
```

Additional resources:

- this helped me for my own blog: https://juliangaramendy.dev/custom-open-graph-images-in-gatsby-blog/
- and the Gatsby docs: https://www.gatsbyjs.org/docs/working-with-images/

Andrew Welch on SEO and a link to a presentation he did back in 2017.

Crafting Modern SEO with Andrew Welch:

Crafting Modern SEO with Andrew Welch:

Crafting Modern SEO with Andrew Welch

Craft CMS

own implementation which I have implemented as a React component. You're going to be configuring that now in this how-to.

First up, install and configure `gatsby-plugin-react-helmet`. This is used for server rendering data added with React Helmet.

```
yarn add gatsby-plugin-react-helmet
```

You'll need to add the plugin to your `gatsby-config.js`. If you haven't done so already now is a good time to also configure the `gatsby-plugin-styled-components` as well.

Configure SEO Component for Homepage

To visualise the data you're going to need to get into the SEO component use the `Dump` component to begin with to validate the data.

The majority of the information needed for `src/pages/index.js` can be first added to the `gatsby-config.js`, `siteMetadata` object then queried with the `useSiteMetadata` hook. Some of the data added here can then be used in `src/templates/blogPostTemplate.js` – more on that in the next section.

For now add the following:

```
const siteMetadata = {
  title: `The Localhost Blog`,
```

```
    siteLocale: `en_gb`,
    twitterUsername: `@spences10`,
    authorName: `Scott Spence`,
  }

  module.exports = {
    siteMetadata: siteMetadata,
    plugins: [
      ...
```

You don't have to abstract out the `siteMetadata` into its own component here. It's only a suggestion on how to manage it.

The `image` is going to be the default image for your site. You should create a `static` folder at the root of the project and add in an image you want to be shown when the homepage of your site is shared on social media.

For `siteUrl` at this stage it doesn't necessarily have to be valid. You can add a dummy url for now and change it later.

The `siteLanguage` is your language of choice for the site. Take a look at w3 language tags for more info.

Facebook OpenGraph is the only place the `siteLocale` is used and it is different from language tags.

Add your `twitterUsername` and your `authorName`.

Update the `useSiteMetadata` hook now to reflect the newly added properties:

```
    graphql`
      query SITE_METADATA_QUERY {
        site {
          siteMetadata {
            description
            title
            image
            siteUrl
            siteLanguage
            siteLocale
            twitterUsername
            authorName
          }
        }
      }
    `
  );
  return site.siteMetadata;
};
```

Begin with importing the `Dump` component in `src/pages/index.js` then plug in the props as they are detailed in the docs of the `react-seo-component` .

```
import Dump from '../components/Dump'
import { useSiteMetadata } from '../hooks/useSiteMetadata'

export default ({ data }) => {
  const {
    description,
    title,
    image,
    siteUrl,
    siteLanguage,
    siteLocale,
    twitterUsername,
  } = useSiteMetadata()
  return (
    <Layout>
      <Dump
```

```
        siteLanguage={siteLanguage}
        siteLocale={siteLocale}
        twitterUsername={twitterUsername}
      />
      <IndexWrapper>
        {data.allMdx.nodes.map(
          ...
```

Check that all the props are displaying valid values. Then you can swap out the `Dump` component with the `SEO` component.

The complete `src/pages/index.js` should look like this now:

```
import { graphql, Link } from 'gatsby';
import Img from 'gatsby-image';
import React from 'react';
import SEO from 'react-seo-component';
import styled from 'styled-components';
import { Layout } from '../components/Layout';
import { useSiteMetadata } from '../hooks/useSiteMetadata';

const IndexWrapper = styled.main``;

const PostWrapper = styled.div``;

const Image = styled(Img)`
  border-radius: 5px;
`;

export default ({ data }) => {
  const {
    description,
    title,
    image,
    siteUrl,
    siteLanguage,
    siteLocale,
    twitterUsername,
  } = useSiteMetadata();
  return (
```

```
          image={`${siteUrl}${image}`}
          pathname={siteUrl}
          siteLanguage={siteLanguage}
          siteLocale={siteLocale}
          twitterUsername={twitterUsername}
        />
        <IndexWrapper>
          {/* <Dump data={data}></Dump> */}
          {data.allMdx.nodes.map(
            ({ id, excerpt, frontmatter, fields }) => (
              <PostWrapper key={id}>
                <Link to={fields.slug}>
                  {!!frontmatter.cover ? (
                    <Image
                      sizes={frontmatter.cover.childImageSharp.siz
                    />
                  ) : null}
                  <h1>{frontmatter.title}</h1>
                  <p>{frontmatter.date}</p>
                  <p>{excerpt}</p>
                </Link>
              </PostWrapper>
            )
          )}
        </IndexWrapper>
      </Layout>
    );
};

export const query = graphql`
  query SITE_INDEX_QUERY {
    allMdx(
      sort: { fields: [frontmatter___date], order: DESC }
      filter: { frontmatter: { published: { eq: true } } }
    ) {
      nodes {
        id
        excerpt(pruneLength: 250)
        frontmatter {
          title
          date(formatString: "YYYY MMMM Do")
          cover {
            publicURL
            childImageSharp {
              sizes(maxWidth: 2000, traceSVG: { color: "#639" })
```

```
        }
        fields {
          slug
        }
      }
    }
  }
`;
```

Configure SEO Component for Blog Posts

This will be the same approach as with the homepage. Import the `Dump` component and validate the props before swapping out the `Dump` component with the `SEO` component.

```
import Dump from '../components/Dump'
import { useSiteMetadata } from '../hooks/useSiteMetadata'

export default ({ data, pageContext }) => {
  const {
```

```
  twitterUsername,
  authorName,
} = useSiteMetadata()
const { frontmatter, body, fields, excerpt } = data.mdx
const { title, date, cover } = frontmatter
const { previous, next } = pageContext
return (
  <Layout>
    <Dump
      title={title}
      description={excerpt}
      image={
        cover === null
          ? `${siteUrl}${image}`
          : `${siteUrl}${cover.publicURL}`
      }
      pathname={`${siteUrl}${fields.slug}`}
      siteLanguage={siteLanguage}
      siteLocale={siteLocale}
      twitterUsername={twitterUsername}
      author={authorName}
      article={true}
      publishedDate={date}
      modifiedDate={new Date(Date.now()).toISOString()}
    />
    <h1>{frontmatter.title}</h1>
    ...
```

Add `fields.slug`, `excerpt` and `cover.publicURL` to the `PostsBySlug` query and destructure them from `data.mdx` and `frontmatter`, respectively.

For the image you'll need to do some logic as to whether the `cover` exists and default to the default site image if it doesn't.

The complete `src/templates/blogPostTemplate.js` should look like this now:

```
import SEO from 'react-seo-component';
import { Layout } from '../components/Layout';
import { useSiteMetadata } from '../hooks/useSiteMetadata';

export default ({ data, pageContext }) => {
  const {
    image,
    siteUrl,
    siteLanguage,
    siteLocale,
    twitterUsername,
    authorName,
  } = useSiteMetadata();
  const { frontmatter, body, fields, excerpt } = data.mdx;
  const { title, date, cover } = frontmatter;
  const { previous, next } = pageContext;
  return (
    <Layout>
      <SEO
        title={title}
        description={excerpt}
        image={
          cover === null
            ? `${siteUrl}${image}`
            : `${siteUrl}${cover.publicURL}`
        }
        pathname={`${siteUrl}${fields.slug}`}
        siteLanguage={siteLanguage}
        siteLocale={siteLocale}
        twitterUsername={twitterUsername}
        author={authorName}
        article={true}
        publishedDate={date}
        modifiedDate={new Date(Date.now()).toISOString()}
      />
      <h1>{frontmatter.title}</h1>
      <p>{frontmatter.date}</p>
      <MDXRenderer>{body}</MDXRenderer>
      {previous === false ? null : (
        <>
          {previous && (
            <Link to={previous.fields.slug}>
              <p>{previous.frontmatter.title}</p>
            </Link>
          )}
```

```
            {next && (
              <Link to={next.fields.slug}>
                <p>{next.frontmatter.title}</p>
              </Link>
            )}
          </>
        )}
      </Layout>
    );
};

export const query = graphql`
  query PostBySlug($slug: String!) {
    mdx(fields: { slug: { eq: $slug } }) {
      frontmatter {
        title
        date(formatString: "YYYY MMMM Do")
        cover {
          publicURL
        }
      }
      body
      excerpt
      fields {
        slug
      }
    }
  }
`;
```

Create a Developer Blog with Gatsby and MDX - configure ...

Build Site and Validate Meta Tags

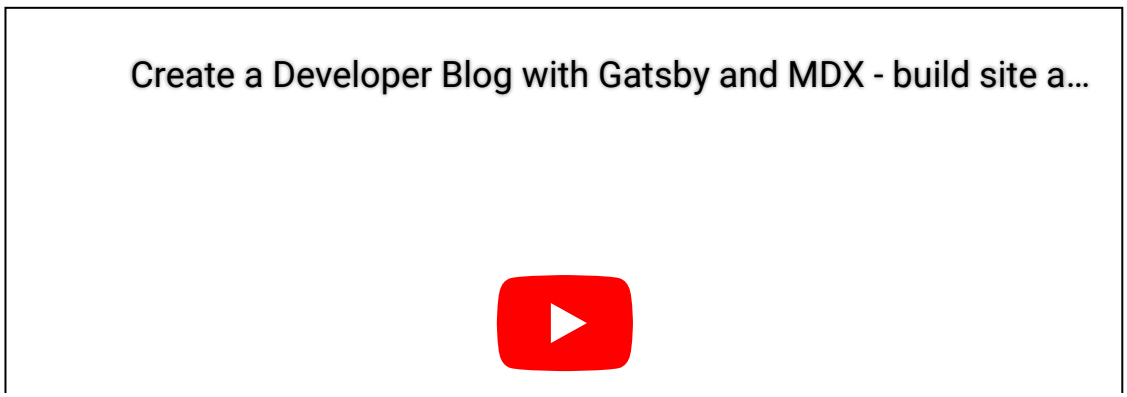
Add in the build script to `package.json` and also a script for serving the built site locally.

```
"scripts": {
  "dev": "gatsby develop -p 9988 -o",
  "build": "gatsby build",
  "serve": "gatsby serve -p 9500 -o"
},
```

Now it's time to run:

```
yarn build && yarn serve
```

This will build the site and open a browser tab so you can see the site as it will appear when it is on the internet. Validate meta tags have been added to the build by selecting "View page source" (Crtl+u in Windows and Linux) on the page. You can do a Ctrl+f to find them.



Adding the Project to GitHub

Add your code to GitHub by either selecting the plus (+) icon next to your avatar on GitHub or by going directly to https://github.com/new

Name your repository and click create repository. Then you will be given the instructions to link your local code to the repository you created via the command line.

How you authenticate with GitHub will depend on what the command looks like.

Some good resources for authenticating with GitHub via SSH are Kent Dodds Egghead.io video and also a how-to on CheatSheets.xyz.

Deploy to Netlify

To deploy your site to Netlify, if you haven't done so already, you'll need to add the GitHub integration to your GitHub profile. If you got to app.netlify.com the wizard will walk you through the process.

From here you can add your built site's `public` folder drag 'n drop style directly to the Netlify global CDNs.

You, however, are going to load your site via the Netlify CLI! In your terminal, if you haven't already got the CLI installed, run:

```
yarn global add netlify-cli
```

Then once the CLI is installed:

```
# authenticate via the CLI
netlify login
# initialise the site
netlify init
```

Enter the details for your team: the site name is optional, the build command will be `yarn build`, and directory to deploy is `public`.

You will be prompted to commit the changes and push them to GitHub (with `git push`). Once you have done that your site will be published and ready for all to see!

Validate Metadata with Heymeta

Last up is validating the metadata for the OpenGraph fields. To do that you'll need to make sure that the `siteUrl` reflects what you have in your Netlify dashboard.

If you needed to change the url you'll need to commit and push the changes to GitHub again.

Once your site is built with a valid url you can then test the homepage and a blog page for the correct meta tags with heymeta.com.

OpenGraph checking tools:

- https://www.heymeta.com/
- https://opengraphcheck.com/
- https://cards-dev.twitter.com/validator
- https://developers.facebook.com/tools/debug/sharing
- https://www.linkedin.com/post-inspector/

Additional resources:

- The Essential Meta Tags for Social Media

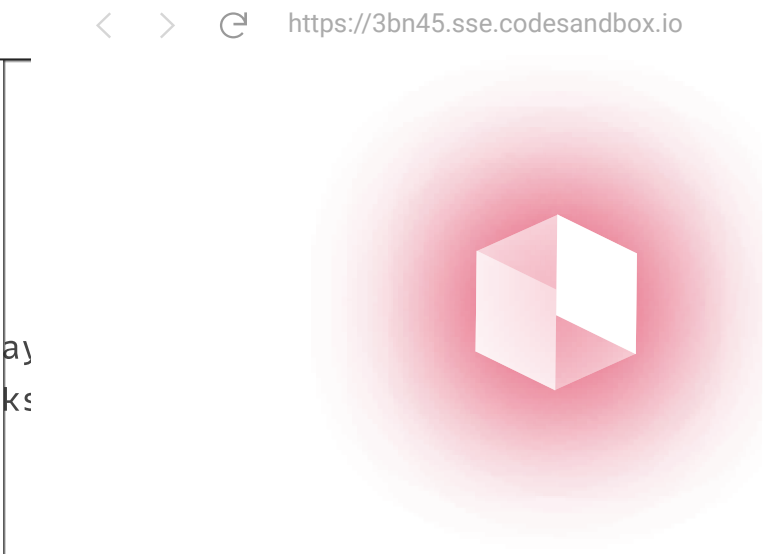
Example Code for this Blog can be found here:

```
 { graphql, Link } from 'gatsby'
 Img from 'gatsby-image'
 React from 'react'
 SEO from 'react-seo-component'
 styled from 'styled-components'
 { Layout } from '../components/La
 { useSiteMetadata } from '../hooks

IndexWrapper = styled.main``

PostWrapper = styled.div``

Image = styled(Img)`
er-radius: 5px;
```



Or here.

Thanks for reading ?

That's all folks! If there is anything I have missed, or if there is a better way to do something, then please let me know.

Follow me on Twitter or Ask Me Anything on GitHub.

> **You can read other articles like this on my digital garden.**

Scott Spence

Developer Advocate @GraphCMS • Learner of things • Svelte • SvelteKit • Jamstack • JavaScript • 🔥 Prev @Accenture, @Deloitte, @Fidelity, @Barclays

---

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

Tweet a thanks

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

You can make a tax-deductible donation here.

Trending Guides

Learn JavaScript
Linux ln Example
JS document.ready()
Delete a Row in SQL
Python Round to Int
What is msmpeng.exe?
Queue Data Structure
Learn Web Development
Install Node on Windows
Remove Char from String
Open Task Manager on Mac
parseInt() in JavaScript
Print statement in Python
Remove Directory in Linux
Python str.lower() Example
Rust Lang
Python Sets
C++ Strings
Python map()
Python .pop()
Python arrays
npm Uninstall
Insertion Sort
Python If-Else
All Caps in CSS
Second Monitor Not Detected
How to Declare Strings in C
How to Use .len() in Python
Python Convert String to Int
How to create a free website

Our Nonprofit