Build A SvelteKit Markdown Blog

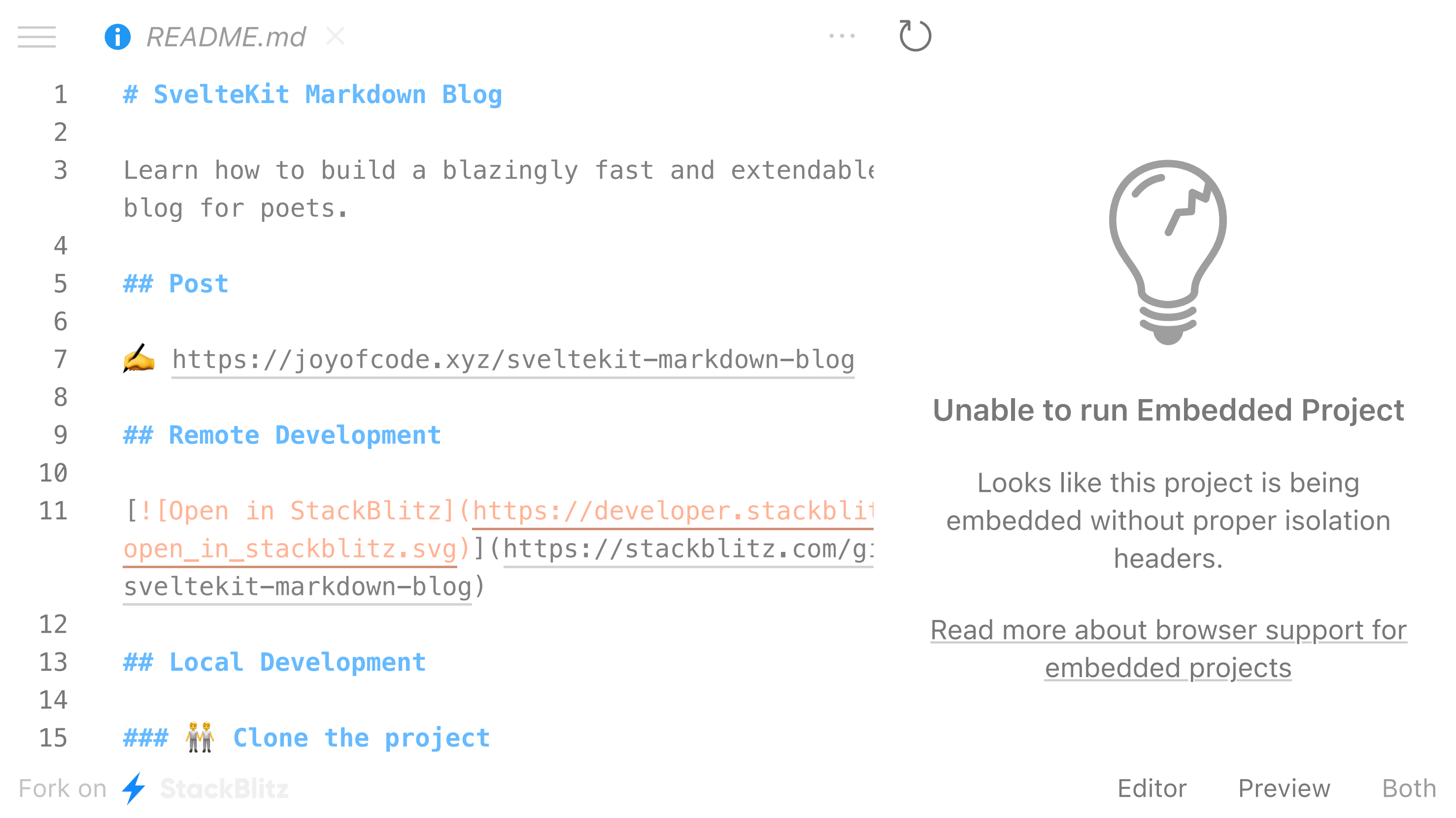
Published Apr 28, 2023

Table Of Contents

- 🔥 Layout And Styles
- 🔥 Setting Up Mdsvex
- 🔥 Posts API Endpoint
- 🔥 Showing Posts
- 🔥 Showing A Single Post
- 🔥 Syntax Highlighting
- 🔥 Using Components Inside Markdown
- 🔥 Using Markdown Plugins
- 🔥 Light And Dark Mode Toggle
- 🔥 Page Transitions
- 🔥 RSS Feed
- 🔥 Custom Error Page
- 🔥 Deployment

Project Setup

You’re going to make a blazingly fast and extendable SvelteKit Markdown blog you can be proud of and deploy it to Vercel at no cost.



README.md
Unable to run Embedded Project
Looks like this project is being embedded without proper isolation headers.
Read more about browser support for embedded projects
Fork on StackBlitz
Editor
Preview
Both

```
# SvelteKit Markdown Blog

Learn how to build a blazingly fast and extendable
blog for poets.

## Post

✍️ https://joyofcode.xyz/sveltekit-markdown-blog

## Remote Development

[![Open in StackBlitz](https://developer.stackbli
open_in_stackblitz.svg)](https://stackblitz.com/g
sveltekit-markdown-blog)

## Local Development

### 👬 Clone the project
```

You can find the finished project **on GitHub**.

🔥 If you want to learn SvelteKit you can watch **The Complete SvelteKit Course For Building Modern Web Apps** on YouTube.

Start by creating a new SvelteKit project.

terminal

```
npm create svelte@latest
```

I'm using **TypeScript** but **types are optional** and can be ignored, so use

JavaScript if you prefer it. Use the **spacebar** to select **ESLint** to find problems in the code and **Prettier** to format the code.

terminal

```
┌  Welcome to SvelteKit!
│
◆ Where should we create your project?
│  sveltekit-blog
│
◇  Which Svelte app template?
│  Skeleton project
│
◇  Add type checking with TypeScript?
│  Yes, using TypeScript syntax
│
◆  Select additional options (use arrow keys/space bar)
│  ■ Add ESLint for code linting
│  ■ Add Prettier for code formatting
│  □ Add Playwright for browser testing
│  □ Add Vitest for unit testing
└
```

Install the dependencies and run the development server at **http://localhost:5173/**.

terminal

```
# install dependencies
npm i

# run the development server
npm run dev
```

Layout And Styles

For styling I’m using **Open Props** which provides design tokens as CSS variables — it’s like Tailwind CSS but instead of utility classes you get CSS variables.

I also want beautiful and consistent icons and **Lucide** is my favorite choice.

For the fonts I’m going to use **Manrope** as the sans serif font for the entire site and **JetBrains Mono** as the monospace font for code blocks.

terminal

```
npm i open-props lucide-svelte @fontsource/manrope @fontsource/jetbrai
```

Update the favicon inside `app.html` so everyone knows your site is blazingly fast.

src/app.html

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <link rel="icon" href="https://fav.farm/🔥" />
    %sveltekit.head%
  </head>
  <body data-sveltekit-preload-data="hover">
    <div style="display: contents">%sveltekit.body%</div>
  </body>
</html>
```

I'm going to add a **config** file for the site which is going to make it easy to update it in the future since everything is in one place.

src/lib/config.ts

```ts
import { dev } from '$app/environment'

export const title = 'Shakespeare'
export const description = 'SvelteKit blog for poets'
export const url = dev ? 'http://localhost:5173/' : 'https://joyofcode
```

🐿️ If SvelteKit doesn't detect the `$lib` alias after you added it restart the development server.

Add a root layout inside `src/routes/+layout.svelte` which is going to include the **header**, **footer** and **styles** for the site.

src/routes/+layout.svelte

```
<script lang="ts">
  import Footer from './footer.svelte'
  import Header from './header.svelte'

  import 'open-props/style'
  import 'open-props/normalize'
  import 'open-props/buttons'

  import '../app.css'
</script>

<div class="layout">
  <!-- Header -->
  <Header />

  <main>
    <!-- Black hole for other content -->
    <slot />
  </main>

  <!-- Footer -->
  <Footer />
</div>

<style>
  .layout {
    height: 100%;
    max-inline-size: 1440px;
    display: grid;
    grid-template-rows: auto 1fr auto;
    margin-inline: auto;
    padding-inline: var(--size-7);
  }

  main {
    padding-block: var(--size-9);
  }

  @media (min-width: 1440px) {
    .layout {
      padding-inline: 0;
    }
  }
</style>
```

src/routes/header.svelte

```svelte
<script lang="ts">
  import * as config from '$lib/config'
</script>

<nav>
  <!-- Title -->
  <a href="/" class="title">
    <b>{config.title}</b>
  </a>

  <!-- Navigation -->
  <ul class="links">
    <li>
      <a href="/about">About</a>
    </li>
    <li>
      <a href="/contact">Contact</a>
    </li>
    <li>
      <a href="/rss.xml" target="_blank">RSS</a>
    </li>
  </ul>

  <!-- Theme -->
  <button>Toggle</button>
</nav>
```

```
<style>
  nav {
    padding-block: var(--size-7);
  }

  .links {
    margin-block: var(--size-7);
  }

  a {
    color: inherit;
    text-decoration: none;
  }

  @media (min-width: 768px) {
    nav {
      display: flex;
      justify-content: space-between;
    }

    .links {
      display: flex;
      gap: var(--size-7);
      margin-block: 0;
    }
  }
</style>
```

src/routes/footer.svelte

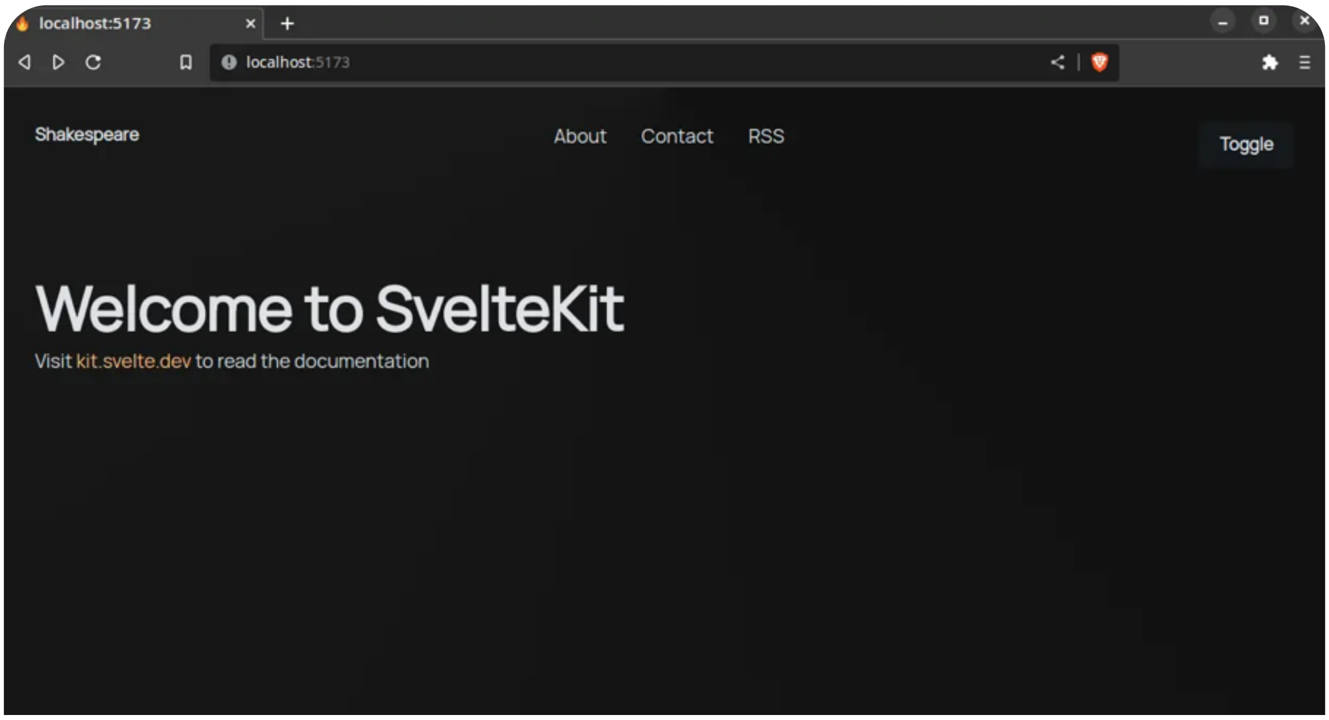
```svelte
<script lang="ts">
  import * as config from '$lib/config'
</script>

<!-- Footer -->
<footer>
  <p>{config.title} &copy {new Date().getFullYear()}</p>
</footer>

<style>
  footer {
    padding-block: var(--size-7);
    border-top: 1px solid var(--border);
  }

  p {
    color: var(--text-2);
  }
</style>
```

src/app.css

```css
@import '@fontsource/manrope';
@import '@fontsource/jetbrains-mono';

html {
  /* font */
  /* font */
  --font-sans: 'Manrope', sans-serif;
  --font-mono: 'JetBrains Mono', monospace;

  /* dark */
  --brand-dark: var(--orange-3);
  --text-1-dark: var(--gray-3);
  --text-2-dark: var(--gray-5);
  --surface-1-dark: var(--gray-12);
  --surface-2-dark: var(--gray-11);
  --surface-3-dark: var(--gray-10);
  --surface-4-dark: var(--gray-9);
  --background-dark: var(--gradient-8);
  --border-dark: var(--gray-9);

  /* light */
  --brand-light: var(--orange-10);
  --text-1-light: var(--gray-8);
  --text-2-light: var(--gray-7);
  --surface-1-light: var(--gray-0);
  --surface-2-light: var(--gray-1);
  --surface-3-light: var(--gray-2);
  --surface-4-light: var(--gray-3);
  --background-light: none;
  --border-light: var(--gray-4);
}

:root {

  color-scheme: dark;
  color-scheme: dark;

  --brand: var(--brand-dark);
  --text-1: var(--text-1-dark);
  --text-2: var(--text-2-dark);
  --surface-1: var(--surface-1-dark);
  --surface-2: var(--surface-2-dark);
  --surface-3: var(--surface-3-dark);
  --surface-4: var(--surface-4-dark);
  --background: var(--background-dark);
  --border: var(--border-dark);
}

@media (prefers-color-scheme: light) {
  :root {
    color-scheme: light;

    --brand: var(--brand-light);
    --text-1: var(--text-1-light);
    --text-2: var(--text-2-light);
    --surface-1: var(--surface-1-light);
    --surface-2: var(--surface-2-light);
    --surface-3: var(--surface-3-light);
    --surface-4: var(--surface-4-light);
    --background: var(--background-light);
    --border: var(--border-light);
  }
}
```

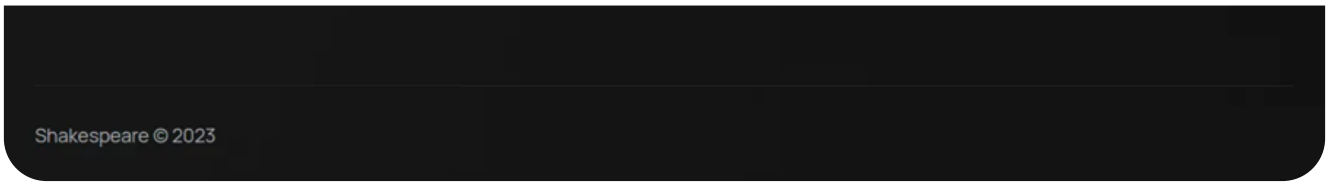
```css
[color-scheme='dark'] {
  color-scheme: dark;

  --brand: var(--brand-dark);
  --text-1: var(--text-1-dark);
  --text-2: var(--text-2-dark);
  --surface-1: var(--surface-1-dark);
  --surface-2: var(--surface-2-dark);
  --surface-3: var(--surface-3-dark);
  --surface-4: var(--surface-4-dark);
  --background: var(--background-dark);
  --border: var(--border-dark);
}

[color-scheme='light'] {
  color-scheme: light;

  --brand: var(--brand-light);
  --text-1: var(--text-1-light);
  --text-2: var(--text-2-light);
  --surface-1: var(--surface-1-light);
  --surface-2: var(--surface-2-light);
  --surface-3: var(--surface-3-light);
  --surface-4: var(--surface-4-light);
  --background: var(--background-light);
  --border: var(--border-light);
}
```

```css
html,
body {
  height: 100%;
}

html {
  color: var(--text-1);
  accent-color: var(--link);
  background-image: var(--background);
  background-attachment: fixed;
}

img {
  border-radius: var(--radius-3);
}

ul,
ol {
  list-style: none;
  padding: 0;
}

li {
  padding-inline-start: 0;
}

.surface-1 {

  background-color: var(--surface-1);
  background-color: var(--surface-1);
  color: var(--text-2);
}

.surface-2 {
  background-color: var(--surface-2);
  color: var(--text-2);
}

.surface-3 {
  background-color: var(--surface-3);
  color: var(--text-1);
}

.surface-4 {
  background-color: var(--surface-4);
  color: var(--text-1);
}
```



Welcome to SvelteKit

Visit kit.svelte.dev to read the documentation



Shakespeare © 2023

💪 As an exercise try adding the `/about` and `/contact` routes yourself since they're mostly used as placeholders.

This sets us up nicely for the rest of the post and I'm going to show you how simple it's going to be to implement a theme switcher because we already set everything up with CSS variables.

Right now even if you don't have a theme toggle it's going to respect the user preference because of the `prefers-color-scheme` media query.

Setting Up Mdsvex

mdsvex is like **MDX** for React but it's a preprocessor for Svelte which lets you have interactive Svelte components inside Markdown and has extra options and extensions for using Markdown plugins.

To get started install mdsvex as a development dependency.

terminal

```
npm i -D mdsvex
```

Add mdsvex as a preprocessor inside `svelte.config.js` .

svelte.config.js

```
import adapter from '@sveltejs/adapter-auto'
import { vitePreprocess } from '@sveltejs/vite-plugin-svelte'

import { mdsvex } from 'mdsvex'

/** @type {import('mdsvex').MdsvexOptions} */
const mdsvexOptions = {
  extensions: ['.md'],
}

/** @type {import('@sveltejs/kit').Config} */
const config = {
  extensions: ['.svelte', '.md'],
  preprocess: [vitePreprocess(), mdsvex(mdsvexOptions)],
  kit: {
    adapter: adapter()
  }
}

export default config
```

Svelte by default handles `.svelte` files but adding `.md` to `extensions` inside `config` lets you treat `+page.md` as a page alongside `+page.svelte`.

Svelte is able to show `+page.md` as a page but **you need mdsvex to preprocess Markdown**.

You have to specify the extension inside the `extensions` array for `mdsvexOptions` which you name name whatever you want like `.md`, `.svx` or `.banana`.

Instead of using `+page.md` files you can import a Markdown post as a module and render it as a regular Svelte component with **svelte:component** which is what I'm going to do later.

I'm going to add some posts in `/src/posts`.

src/posts/first-post.md

```
---
title: First post
description: First post.
date: '2023-4-14'
categories:
  - sveltekit
  - svelte
```

~~~
published: true
---

## Markdown

Hey friends! 👋

```ts
function greet(name: string) {
  console.log(`Hey ${name}! 👋`)
}
```
~~~

src/posts/second-post.md

````
---
title: Second
description: Second post.
date: '2023-4-16'
categories:
  - sveltekit
  - svelte
published: true
---

## Svelte

Media inside the **static** folder is served from `/`.
````

```
Media inside the **static** folder is served from `/`.

![Svelte](favicon.png)
```

Posts API Endpoint

You could write the logic to get the posts data for each page in their respective `+page.ts` or `+page.server.ts` file but you would end up duplicating the logic.

I'm going to create a `routes/api/posts/+server.ts` endpoint instead where I'm going to write the logic once I can use anywhere in the app.

src/routes/api/posts/+server.ts

```ts
import { json } from '@sveltejs/kit'
import type { Post } from '$lib/types'

async function getPosts() {
  let posts: Post[] = []

  const paths = import.meta.glob('/src/posts/*.md', { eager: true })

  for (const path in paths) {
    const file = paths[path]
    const slug = path.split('/').at(-1)?.replace('.md', '')
    if (file && typeof file === 'object' && 'metadata' in file && slug
      const metadata = file.metadata as Omit<Post, 'slug'>
      const post = { ...metadata, slug } satisfies Post
      post.published && posts.push(post)
    }
  }

  posts = posts.sort((first, second) =>
    new Date(second.date).getTime() - new Date(first.date).getTime()
  )

  return posts
}

export async function GET() {
  const posts = await getPosts()
  return json(posts)
}
```

🔥 `import.meta.glob` is a useful Vite feature to get all the posts using a glob (`eager` reads the contents of the file avoiding `await paths[path]()`)

🔥 I loop over the `paths` and get the slug `post.md` but replace `.md` since we only want the slug

🔥 I'm checking if `file` has a `metadata` property inside of it and if the

🔥 I'm checking if `file` has a `metadata` property inside of it and if the `slug` exists to be safe and then I'm going to get the `metadata` or **frontmatter** from the post

🔥 I'm creating a `post` that includes `metadata` and the `slug`

🔥 I only want to add the post if `published` is set to `true`

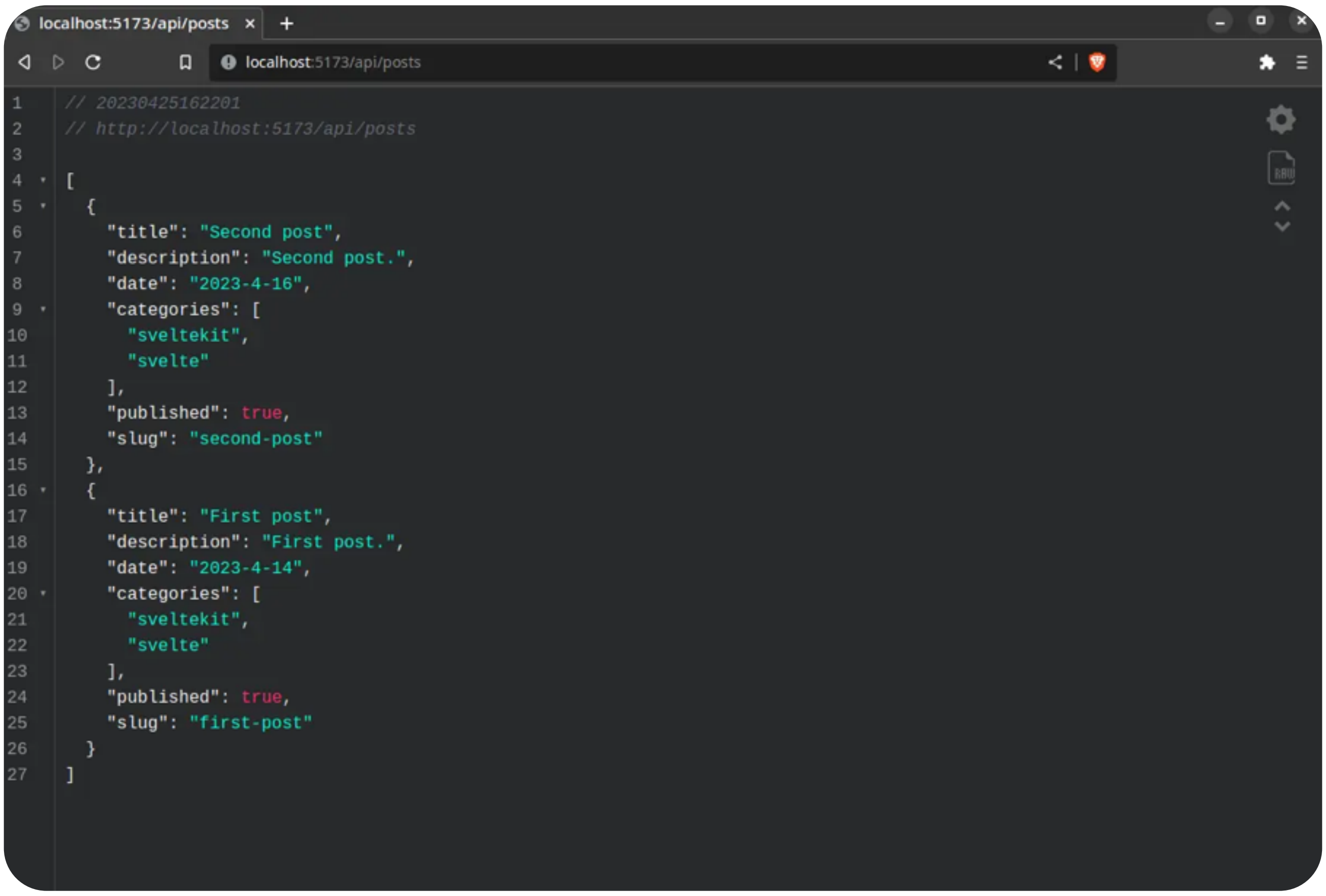
🔥 sort `posts` by date and return them

If you're using TypeScript here are the types.

src/lib/types.ts

```ts
export type Categories = 'sveltekit' | 'svelte'

export type Post = {
  title: string
  slug: string
  description: string
  date: string
  categories: Categories[]
  published: boolean
}
```

I often see developers split everything into separate files but I prefer to keep the logic where it's used unless it's used in multiple places in which case I would put the `getPosts` function inside `lib/posts.ts` .



```
// 20230425162201
// http://localhost:5173/api/posts

[
  {
    "title": "Second post",
    "description": "Second post.",
    "date": "2023-4-16",
    "categories": [
      "sveltekit",
      "svelte"
    ],
    "published": true,
    "slug": "second-post"
  },
  {
    "title": "First post",
    "description": "First post.",
    "date": "2023-4-14",
    "categories": [
      "sveltekit",
      "svelte"
    ],
    "published": true,
    "slug": "first-post"
  }
]
```

🐿️ You can use the **JSON Viewer** Chrome extension for the highlighting.

Awesome! You created an API endpoint for posts you can reuse across your app (you can even make it public for others to consume). 🔥

Showing Posts

Now you can use the posts endpoint you just created to server-side render the posts for the page.

src/routes/+page.server.ts

```ts
import type { Post } from '$lib/types'

export async function load({ fetch }) {
  const response = await fetch('api/posts')
  const posts: Post[] = await response.json()
  return { posts }
}
```

🐿️ The `fetch` function from `load` has superpowers like being able to resolve the relative URL `api/posts` which would not work using regular `fetch` .

At this point you just need to loop over the posts and that's it.

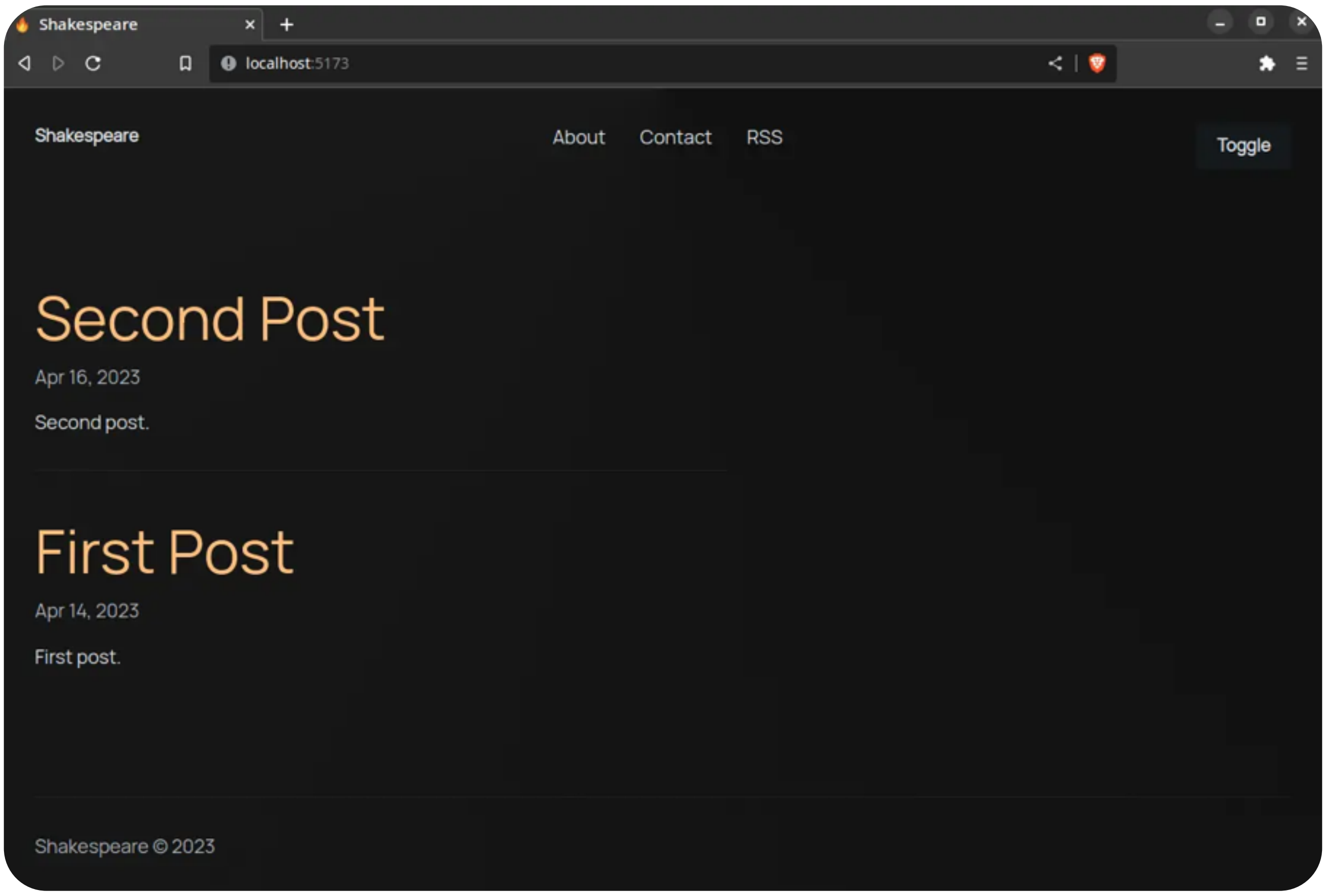
src/routes/+page.svelte

```svelte
<script lang="ts">
  import { formatDate } from '$lib/utils'
  import * as config from '$lib/config'
  export let data
</script>

<svelte:head>
  <title>{config.title}</title>
</svelte:head>

<!-- Posts -->
<section>
  <ul class="posts">
    {#each data.posts as post}
      <li class="post">
        <a href={post.slug} class="title">{post.title}</a>
        <p class="date">{formatDate(post.date)}</p>
        <p class="description">{post.description}</p>
      </li>
    {/each}
  </ul>
</section>

<style>
  .posts {
    display: grid;
    gap: 2rem;
  }

  .post {

    max-inline-size: var(--size-content-3);
  }
  }

  .post:not(:last-child) {
    border-bottom: 1px solid var(--border);
    padding-bottom: var(--size-7);
  }

  .title {
    font-size: var(--font-size-fluid-3);
    text-transform: capitalize;
  }

  .date {
    color: var(--text-2);
  }

  .description {
    margin-top: var(--size-3);
  }
</style>
```

I love using this function which uses the built-in browser API to format dates.

src/lib/utils.ts

```
type DateStyle = Intl.DateTimeFormatOptions['dateStyle']

export function formatDate(date: string, dateStyle: DateStyle = 'mediu
  // Safari is mad about dashes in the date
  const dateToFormat = new Date(date.replaceAll('-', '/'))
  const dateFormatter = new Intl.DateTimeFormat(locales, { dateStyle }
  return dateFormatter.format(dateToFormat)
}
```



Second Post

Apr 16, 2023

Second post.

First Post

Apr 14, 2023

First post.

Shakespeare © 2023

Things are coming together! 💪

Showing A Single Post

Instead of doing `routes/post/+page.md|svelte` for every post I'm going to use a dynamic route `routes/[slug]/+page.svelte` that's going to slurp up the post based on the slug.

Here is where picking mdsvex shines because I'm going to use a dynamic import to get the post which is going to have the content and metadata.

src/routes/[slug]/+page.ts

```
import { error } from '@sveltejs/kit'

export async function load({ params }) {
  try {
    const post = await import(`../../posts/${params.slug}.md`)

    return {
      content: post.default,
      meta: post.metadata
    }
  } catch (e) {
    error(404, `Could not find ${params.slug}`)
  }
}
```

Because the Markdown file is imported as a module and processed by mdsvex you can pass `data.content` as a Svelte component to `<svelte:component this={data.content} />` .

src/routes/[slug]/+page.svelte

```svelte
<script lang="ts">
  import { formatDate } from '$lib/utils'

  export let data
</script>

<!-- SEO -->
<svelte:head>
  <title>{data.meta.title}</title>
  <meta property="og:type" content="article" />
  <meta property="og:title" content={data.meta.title} />
</svelte:head>

<article>
  <!-- Title -->
  <hgroup>
    <h1>{data.meta.title}</h1>
    <p>Published at {formatDate(data.meta.date)}</p>
  </hgroup>

  <!-- Tags -->
  <div class="tags">
    {#each data.meta.categories as category}
      <span class="surface-4">&num;{category}</span>
    {/each}
  </div>
```

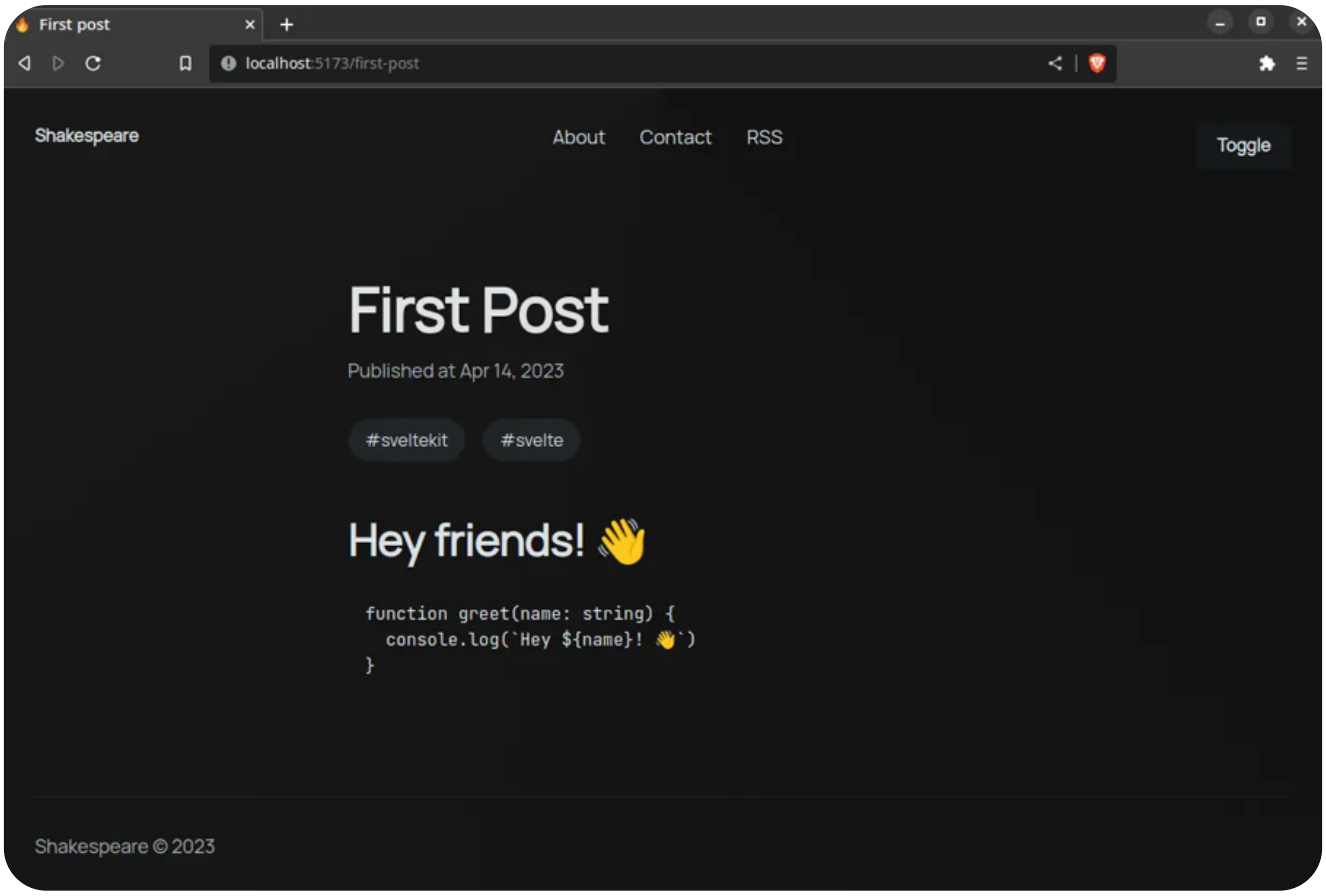
```
    <!-- Post -->
    <div class="prose">
      <svelte:component this={data.content} />
    </div>
  </article>

  <style>
    article {
      max-inline-size: var(--size-content-3);
      margin-inline: auto;
    }

    h1 {
      text-transform: capitalize;
    }

    h1 + p {
      margin-top: var(--size-2);
      color: var(--text-2);
    }

    .tags {
      display: flex;
      gap: var(--size-3);
      margin-top: var(--size-7);
    }


    .tags > * {
```

```
    padding: var(--size-2) var(--size-3);
    border-radius: var(--radius-round);
  }
</style>
```

Let’s add some styles for the post which I’m going to do inside `app.css` because we don’t have control over the markup.

src/app.css

```css
/* ... */

.prose :is(h2, h3, h4, h5, h6) {
  margin-top: var(--size-8);
  margin-bottom: var(--size-3);
}

.prose p:not(:is(h2, h3, h4, h5, h6) + p) {
  margin-top: var(--size-7);
}

.prose :is(ul, ol) {
  list-style-type: '🔥';
  padding-left: var(--size-5);
}

.prose :is(ul, ol) li {
  margin-block: var(--size-2);
  padding-inline-start: var(--size-2);
}


.prose pre {
  max-inline-size: 100%;
  padding: 1rem;
  border-radius: 8px;
  tab-size: 2;
}
```



First Post

Published at Apr 14, 2023

#sveltekit #svelte

Hey friends! 👋

```
function greet(name: string) {
  console.log(`Hey ${name}! 👋`)
}
```

Shakespeare © 2023

💪 Right now the tags are just using a `<span>` but can you turn

them into links `<a href="/category/{category}>{category}`

`</a>` so they point to a `/category/[category]` page that shows the posts based on the name of the category?

Everything looks great!

Syntax Highlighting

mdsvex uses **Prism** by default for syntax highlighting and you can have a look at **Prism themes** which you can include in your styles.

You can use a Prism theme and be done but I want to use a modern syntax highlighter like **Shiki** that uses the same highlighter as VS Code which means you can use real themes.

To use Shiki I'm going to create a custom highlighter which is only going to be a couple of lines of code.

Install Shiki.

terminal

```
npm i shiki
```

Create a custom highlighter.

svelte.config.js

```
// ...
import { mdsvex, escapeSvelte } from 'mdsvex'
import { getHighlighter } from 'shiki'

/** @type {import('mdsvex').MdsvexOptions} */
const mdsvexOptions = {
  extensions: ['.md'],
  highlight: {
    highlighter: async (code, lang = 'text') => {
      const highlighter = await getHighlighter({
        themes: ['poimandres'],
        langs: ['javascript', 'typescript']
      })
      await highlighter.loadLanguage('javascript', 'typescript')
      const html = escapeSvelte(highlighter.codeToHtml(code, { lang, t
      return `{@html \`${html}\` }`
    }
  },
}

// ...
```

🔥 first I create the highlighter using `shiki.getHighlighter` and pass one

of the **VS Code themes** you want (you can also give it a path to your theme) of the **VS Code themes** you want (you can also give it a path to your theme)

🔥 you have to use `escapeSvelte` to escape some characters like `{` that are going to cause a problem in Svelte

🔥 Shiki is going to generate HTML that looks like your code in VS Code using the `code` and `lang` you passed

🔥 I want to insert `{@html html}` in the Svelte component to output the code block but we need to escape the backticks with `\`

I would love to take credit as a genius but I figured everything out by **reading this response** inside an mdsvex GitHub issue.

You can do a lot more with Shiki by reading their docs but if you want line numbers and highligthing you're going to have to **look through the Shiki issues on GitHub**.

Using Components Inside Markdown

You can import regular Svelte components inside Markdown from interactive data visualizations to working code examples.

src/posts/counter.svelte

```
<script lang="ts">
```

```
  let count = 0

  const increment = () => (count += 1)
</script>

<button on:click={increment}>
  {count}
</button>
```

src/posts/example.md

```
<!-- ... -->
<script>
  import Counter from './counter.svelte'
</script>

## Counter

The counter is rendered inside Markdown.

<Counter />
```

Another great mdsvex feature is being able to replace elements with **custom components**.

One example where this is useful is for the `<img>` or `<iframe>` element where you might want set `loading="lazy"` to only load it when it’s in view but you can't set attributes on an image like `![Text](image.webp)` inside Markdown.

First you need to create a default **mdsvex layout** that's going to wrap the Markdown files and you can name it anything but I'm going to name it `mdsvex.svelte` to avoid confusion with `+layout.svelte` .

svelte.config.js

```
/** @type {import('mdsvex').MdsvexOptions} */
const mdsvexOptions = {
  extensions: ['.md'],
  layout: {
    _: './src/mdsvex.svelte'
  },
  // ...
}
```

I'm going to create a custom `img.svelte` component but I'm going to use `index.ts` to export every custom component from `lib/components/custom` making it easier to use when you add more components in the future.

src/lib/components/custom/img.svelte

```
<script lang="ts">
```

```
  export let src: string
  export let alt: string
</script>

<img {src} {alt} loading="lazy" />
```

src/lib/components/custom/index.ts

```
import img from './img.svelte'
export { img }
```

The custom component receives the attributes of the element you want to replace like `src` and `alt` as props.

Inside the layout you have to import and export the custom component **with the same name** as the element you want to replace.

src/mdsvex.svelte

```
<script lang="ts" context="module">
  import { img } from '$lib/components/custom'
  export { img }
</script>

<slot />
```

Images and other media should be placed inside the `static` folder at the root of your project.

You can use images inside Markdown with `![Text](image.webp)` and you should see that it was replaced with the custom component.

You can **explore more of the options** mdsvex offers like **smartypants** that replaces quotes with **“real typographic punctuation”**.

Using Markdown Plugins

There’s an entire world of **abstract syntax trees** (ASTs) which is what HTML or Markdown get turned into to be easily manipulated instead of using regular expressions.

For **transforming HTML** with plugins you can use **rehype** and for **transforming Markdown** with plugins you use **remark**.

I’m going to refer to them as **Markdown plugins** even if they’re general plugins for transforming HTML and Markdown.

mdsvex first parses the Markdown into a Markdown AST (MDAST) where **remark** plugins run and then it converts it into a HTML AST (HAST) where **rehype** plugins run.

You don’t have to understand ASTs but I recommend reading **How to Modify**

You don't have to understand ASTs but I recommend reading **How to Modify Nodes in an Abstract Syntax Tree** if you want to learn the fundamentals and write your own plugin which is just a JavaScript function.

mdsvex makes it easy to use these plugins and you only have to install the desired plugin then pass it into the `remarkPlugins` or `rehypePlugins` options array.

You can find these plugins on **npm** and in the **repository for remark plugins** or **repository for rehype plugins**.

I'm going to show you a couple of plugins, so you get an idea how simple it is to extend your Markdown blog:

🔥 I'm going to use `remark-unwrap-images` to remove `<p>` tags around images because they're annoying for styling

🔥 I want to add slugs to headings like `<h2 id="section">` to make it linkable as `example.com/post#section` using `rehype-slug`

🔥 I want to generate a table of contents based on the headings using `remark-toc`

Install the Markdown plugins.

terminal

```
npm i remark-unwrap-images remark-toc rehype-slug
npm i remark-unwrap-images remark-toc rehype-slug
```

Add them to the config.

svelte.config.js

```js
// ...
import remarkUnwrapImages from 'remark-unwrap-images'
import remarkToc from 'remark-toc'
import rehypeSlug from 'rehype-slug'

/** @type {import('mdsvex').MdsvexOptions} */
const mdsvexOptions = {
  // ...
  remarkPlugins: [remarkUnwrapImages, [remarkToc, { tight: true }]],
  rehypePlugins: [rehypeSlug]
}
```

🐿️ You can pass options for the plugin like `[plugin, { options }]` which you can find in their docs. In this case `{ tight: true }` removes `<p>` tags around `<li>` .

That's how easy that was! 😄

Light And Dark Mode Toggle

Earlier we set ourselves up for success by using CSS variables which respects the users preference by using the `prefers-color-scheme` media query but we also included selectors when `<html>` has a `color-scheme="dark"` and `color-scheme="light"` attribute.

To make a theme toggle we need to write some JavaScript that's going to check when the page loads if the user has a theme set in **localStorage** and set the attribute on `<html>` based on their preference and if not default to using the dark theme.

I'm going to write the code that checks for the theme in `app.html` because it's going to load first and prevent issues like flashing.

Flashing happens when JavaScript is not loaded on the page and because of it when your component mounts it only then goes to `localStorage` to check if you set a theme.

src/app.html

```html
<!DOCTYPE html>
<html lang="en">
  <head>

    <!-- ... -->
    <script type="module">
      const theme = localStorage.getItem('color-scheme')

      theme
        ? document.documentElement.setAttribute('color-scheme', theme)
        : localStorage.setItem('color-scheme', 'dark')
    </script>
  </head>
  <!-- ... -->
</html>
```

I'm going to make a simple **Svelte store** that's going to export the active `theme` so you can subscribe to it anywhere and get notified when it changes including `toggleTheme` to update it and `setTheme` to set the theme in case you want to use it.

src/lib/theme.ts

```
import { writable } from 'svelte/store'
import { browser } from '$app/environment'

type Theme = 'light' | 'dark'

// we set the theme in `app.html` to prevent flashing
const userTheme = browser && localStorage.getItem('color-scheme')

// create the store
export const theme = writable(userTheme ?? 'dark')
export const theme = writable(userTheme ?? 'dark')

// update the theme
export function toggleTheme() {
  theme.update((currentTheme) => {
    const newTheme = currentTheme === 'dark' ? 'light' : 'dark'

    document.documentElement.setAttribute('color-scheme', newTheme)
    localStorage.setItem('color-scheme', newTheme)

    return newTheme
  })
}

// set the theme
export function setTheme(newTheme: Theme) {
  theme.set(newTheme)
}
```

💪 If you want to improve this I would use JavaScript to check the user preference with `const preference = window.matchMedia('(prefers-color-scheme: dark)').matches` and then use an event listener `preference.addEventListener('change', (mediaQuery) => ...)` to update the store value if it changes

I'm going to create a `toggle.svelte` component and use it inside `header.svelte` .

src/routes/toggle.svelte

```svelte
<script lang="ts">
  import { fly } from 'svelte/transition'
  import { Moon, Sun } from 'lucide-svelte'
  import { theme, toggleTheme } from '$lib/theme'
</script>

<button on:click={toggleTheme} aria-label="Toggle theme">
  {#if $theme === 'dark'}
    <div in:fly={{ y: 10 }}>
      <Sun />
      <span>Light</span>
    </div>
  {:else}
    <div in:fly={{ y: -10 }}>
```

```svelte
    </div>
  {/if}
</button>

<style>

  button {
    padding: 0;
    font-weight: inherit;
    background: none;
    border: none;
    box-shadow: none;
    overflow: hidden;
  }

  button > * {
    display: flex;
    gap: var(--size-2);
  }
</style>
```

header.svelte

```
<script lang="ts">
  import Toggle from './toggle.svelte'
  // ...
</script>

<nav>
  <!-- ... -->
  <Toggle />
</nav>
```

That’s it! 😎

Page Transitions

Adding some simple page transitions is going to give your site an air of whimsy and sophistication.

I have an entire post on **SvelteKit Page Transitions** but the gist is that we need to know when the url changed to destroy and recreate the page which is going to play the transition.

> 🐿️ You could use the `$navigates` store from SvelteKit as the key for the transition but it's going to cause problems on slower connections if the page isn't ready to load and play the transition for the current page since the navigation occured.

src/routes/+layout.ts.

```
export async function load({ url }) {
  return {
    url: url.pathname
  }
}
```

src/routes/transition.svelte

src/routes/transition.svelte

```svelte
<script lang="ts">
  import { fade } from 'svelte/transition'

  export let url: string
</script>

{#key url}
  <div class="transition" in:fade>
    <slot />
  </div>
{/key}

<style>
  .transition {
    height: 100%;
  }
</style>
```

src/routes/+layout.svelte

```svelte
<script lang="ts">
  import Footer from './footer.svelte'
  import Header from './header.svelte'
  import PageTransition from './transition.svelte'

  // ...
  export let data
```

```
</script>

<div class="layout">
  <Header />

  <main>
    <PageTransition url={data.url}>
      <slot />
    </PageTransition>
  </main>

  <Footer />
</div>
```

The site is a smooth operator! 🎷

RSS Feed

A lot of people prefer to get notified about updates in their RSS reader.

Creating an RSS feed in SvelteKit is simple as creating an API endpoint that returns XML.

src/routes/rss.xml/+server.ts

```
import * as config from '$lib/config'
import type { Post } from '$lib/types'

export async function GET({ fetch }) {
  const response = await fetch('api/posts')
  const posts: Post[] = await response.json()

  const headers = { 'Content-Type': 'application/xml' }

  const xml = `
    <rss xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">
      <channel>
        <title>${config.title}</title>
        <description>${config.description}</description>
        <link>${config.url}</link>
        <atom:link href="${config.url}/rss.xml" rel="self" type="appli
        ${posts
          .map(
            (post) => `
            <item>
              <title>${post.title}</title>
              <description>${post.description}</description>
              <link>${config.url}/${post.slug}</link>
              <guid isPermaLink="true">${config.url}/${post.slug}</gui
              <pubDate>${new Date(post.date).toUTCString()}</pubDate>
            </item>
          `
          )
          .join('')}
      </channel>
    </rss>
  `.trim()

  return new Response(xml, { headers })
}
```

🐿️ The `rss.xml` extension is optional. You could also name the posts endpoint `posts.json` if you wanted which is preferred if you use prerendering and have clashing route names.

This is the least XML required for an RSS feed which you can validate with **W3C Feed Validation Service** when you deploy the site.

You can include this markup in `app.html` so when people input your website URL in their RSS reader it's going to pick it up.

src/app.html

```html
<!DOCTYPE html>
<html lang="en">

  <head>
    <link rel="alternate" type="application/atom+xml" href="/rss.xml"
    <!-- ... -->
  </head>
  <!-- ... -->
</html>
```

Custom Error Page

Let’s customize the error page before we deploy the site by adding `+error.svelte` in `routes` .

🐿️ You can add `+error.svelte` inside other routes if you want but since we don’t have a lot of nested layouts this is fine.

src/routes/+error.svelte

```
<script>
  import { page } from '$app/stores'
</script>

<div class="error">
  <h1>{$page.status}: {$page.error?.message}</h1>
</div>

<style>
```

```
  .error {
    height: 100%;
    display: grid;
    place-content: center;
  }
</style>
```

Deployment

For deployment I’m going to use **Vercel** and prerender the content ahead of time before deploying it which means it’s going to be blazingly fast. 🔥

Prerendering means creating the HTML files at build time (when you run `npm run build` or `vite build`).

Prerendering your site takes one line of code.

src/routes/+layout.ts

```
export const prerender = true
// ...
```

You also need to prerender the RSS feed.

src/routes/rss.xml/+server.ts

```
export const prerender = true
// ...
```

If you use the `prerender` SvelteKit page option in the root layout SvelteKit is going to crawl the links in your site and prerender the pages (`+page.svelte`) and server routes (`+server.ts`).

If you have pages with **form actions** they can't be prerendered because you need a server but SvelteKit is flexible and you can disable prerendering for that page.

I'm going to use the **Vercel adapter** and update the config.

terminal

```
# remove the default adapter
npm remove @sveltejs/adapter-auto

# add Vercel adapter
npm i -D @sveltejs/adapter-vercel
```

svelte.config.js

```
import adapter from '@sveltejs/adapter-vercel'
// ...
```

🐿️ Before deploying run `npm run build` and `npm run preview` to check for any obvious errors instead of finding out about it during deployment. This is going to create a `.vercel` folder which should be added to `.gitignore` .

Create a new project on GitHub and name it anything you want then push the code to GitHub.

terminal

```
# initialize Git repository
git init

# stage changes for every file
git add .

# add commit
git commmit -m "Add project"

# add remote repository

git remote add origin https://github.com/you/sveltekit-blog.git

# rename current branch to main
git branch -M main
```

```
# push the main branch to origin
git push -u origin main
```

Add a new project on Vercel and **import** your repository. You can leave the default options and press **deploy** which should take a minute.

Look at the **Building** output in case you run into errors in which case just stay calm and read the error like you would in local development because you probably forgot something or made a typo.

You should be greeted by a celebratory screen 🎉 where you can tap **Continue to Dashboard** and then tap **Visit** to see the site.

If you don't like the name of the random URL Vercel assigned to your project go to **Settings** > **Domains** and tap **Edit** and give it a new domain name that ends with **.vercel.app** like **shakespeare.vercel.app** which is a great name already without having to use a custom domain name.

Each time you push to the GitHub repository Vercel is going to **redeploy** and **run the build** and **bust the cache** because it's integrated with GitHub.

That's it! 🎉

Support

You can subscribe on YouTube, or consider becoming a patron if you want to support my work.

Patreon →

Found a mistake?

Every post is a Markdown file so contributing is simple as following the link below and pressing the pencil icon inside GitHub to edit it.

Edit on GitHub →