

# 1 - Svelte

In this chapter, we will set up the skeleton for our Svelte application, creating the basic UI, routing, and page content.

## Chapter Overview

### Learning goals

At the end of this chapter, you will be able to:

- Understand why we will use Svelte to build our application.
- Understand and explain the structure of a Svelte application.
- Add content, typescript, and styles (using Tailwind CSS) to a Svelte application.
- Understand how basic routing works in a Svelte application using SvelteKit.
- Understand what `+page.svelte` files are, their purpose, and how to use them to create pages for your application.
- Understand what `+layout.svelte` files are, their purpose, and how to use them to create layouts for your application.
- Understand how to create Components in Svelte to modularize your UI for reusability.
- Understand how to use Typescript in Svelte components.
- Understand how the `$state` and `$derived` runes work in svelte to create reactive variables and derived state.
- Understand how to use the `$props()` rune to pass data between components.
- Understand how re-use strings across components.

At the end of this chapter, you will have created:

- A Svelte application with basic layout and routing.
- A site with the following structure:
  - `/` : A home page with a welcome message, and a reactive counter.
  - `/about` : An about page with a message about the application.
  - `/dashboard` : A dashboard page with links to user and sessions pages.
    - `/dashboard/user` : A user page with some sample content.
    - `/dashboard/sessions` : A sessions page with some sample content.
- A reusable, styled, and modular `NavBar.svelte` component.

### Learning Resources

- [HTML \(W3 schools\)](#)
- [CSS \(W3 schools\)](#)
- [JavaScript \(W3 schools\)](#)
- Typescript:
  - [Typescript \(Mozilla\)](#)
  - [Typescript \(W3 schools\)](#)
  - [Typescript \(Official Documentation\)](#)
- Svelte:
  - [Svelte \(Official Documentation\)](#)
  - [Svelte \(Interactive Tutorial\)](#)
  - [SvelteKit \(Official Documentation\)](#)
- Tailwind CSS:
  - [Tailwind CSS \(Official Documentation\)](#)

## Why Svelte?

Svelte and SvelteKit are awesome frameworks for building web applications. There are many other frameworks available, such as [React](#), [Vue](#), and [Angular](#), but Svelte stands out for its ease of use and simple yet thorough documentation. Svelte also has an awesome [interactive tutorial](#) that is a great way to learn the basics of the framework.

## Popularity and community

Though not [as popular](#) as other frameworks as of 2024, Svelte is [gaining traction](#) and is, according to its homepage, consistently ranked as the most-loved framework. The Stack Overflow survey showed that [73% of developers](#) who use Svelte want to keep working with Svelte.

It's fun, easy to learn, and has a helpful community (e.g. their [Discord server](#) has lots of active discussions and members asking - and answering - questions).

## But is it a good career choice?

Yes! But maybe for different reasons than you might hope or think.

- **Svelte is *not* as widely used in the industry as some other frameworks** (although we already showed you that it is gaining traction), so you might not find as many internships, or job postings specifically asking for experience with Svelte.
- **Svelte is easy to learn and accessible for beginners**, which means it's a great way to get into thinking about web development and building applications, without all of the complexity that other frameworks might introduce.

- **You can translate many of the skills you learn in Svelte to other frameworks**, and many companies are looking for developers who can learn new technologies quickly. So, while you might not find a job specifically asking for Svelte experience, you will find that the skills you learn in this course will help you land a job in web development.
- **You can always branch out into other frameworks later should you so desire.** Many of them share many concepts with Svelte (components, state management, routing, etc.), and you will find that your knowledge of Svelte will help you learn other frameworks more easily. (You will also probably find that you wish they did things like Svelte does).

## The most important reason to use Svelte

**Svelte is just nice to work with. Which means you will be more likely to finish this course.** It has simple and intuitive syntax, and uses a compiler to abstract away much of the complex logic you might find in other frameworks. This means you can focus on building your application without getting too bogged down in the details of the framework itself.

This is especially important for a course like this, where we want you to be able to focus on building your application and learning the concepts, rather than getting stuck on the framework itself.

## Svelte and SvelteKit

These two terms may seem similar, but they refer to different things:


- **Svelte** is a component-based framework that allows you to build reusable UI components.
  - It's a way of writing components that compile into highly optimized JavaScript code that runs in the browser, resulting in fast and efficient applications.
  - Svelte is designed to be easy to learn and use, with a simple and intuitive syntax that allows you to focus on building your application rather than the framework itself.
- **SvelteKit** is a framework for building web applications with Svelte that provides routing, server-side rendering, and other features.
  - It is built for applications using Svelte and provides a set of tools and conventions for building web applications.
  - SvelteKit is the recommended way to build Svelte applications, as it provides a lot of features out of the box that you would otherwise have to implement yourself.

Here's a simple analogy: We can use Svelte to create a web page. SvelteKit allows us to create an entire web application.

Refer to the [SvelteKit documentation](#) for more information on the differences between Svelte and SvelteKit.

# Basic Svelte

In the previous section, we have created our Svelte application using the 'Skeleton' template. Now, we will set up our basic application structure.

 Before we go any further, please make sure that you have completed at least part of the [Svelte interactive tutorial](#) to familiarize yourself with Svelte. It perfectly explains the basics of Svelte.

We will still cover some of the basics of Svelte here, but it is highly recommended to follow the interactive tutorial first, and then come back to this section to reinforce your newly acquired knowledge.

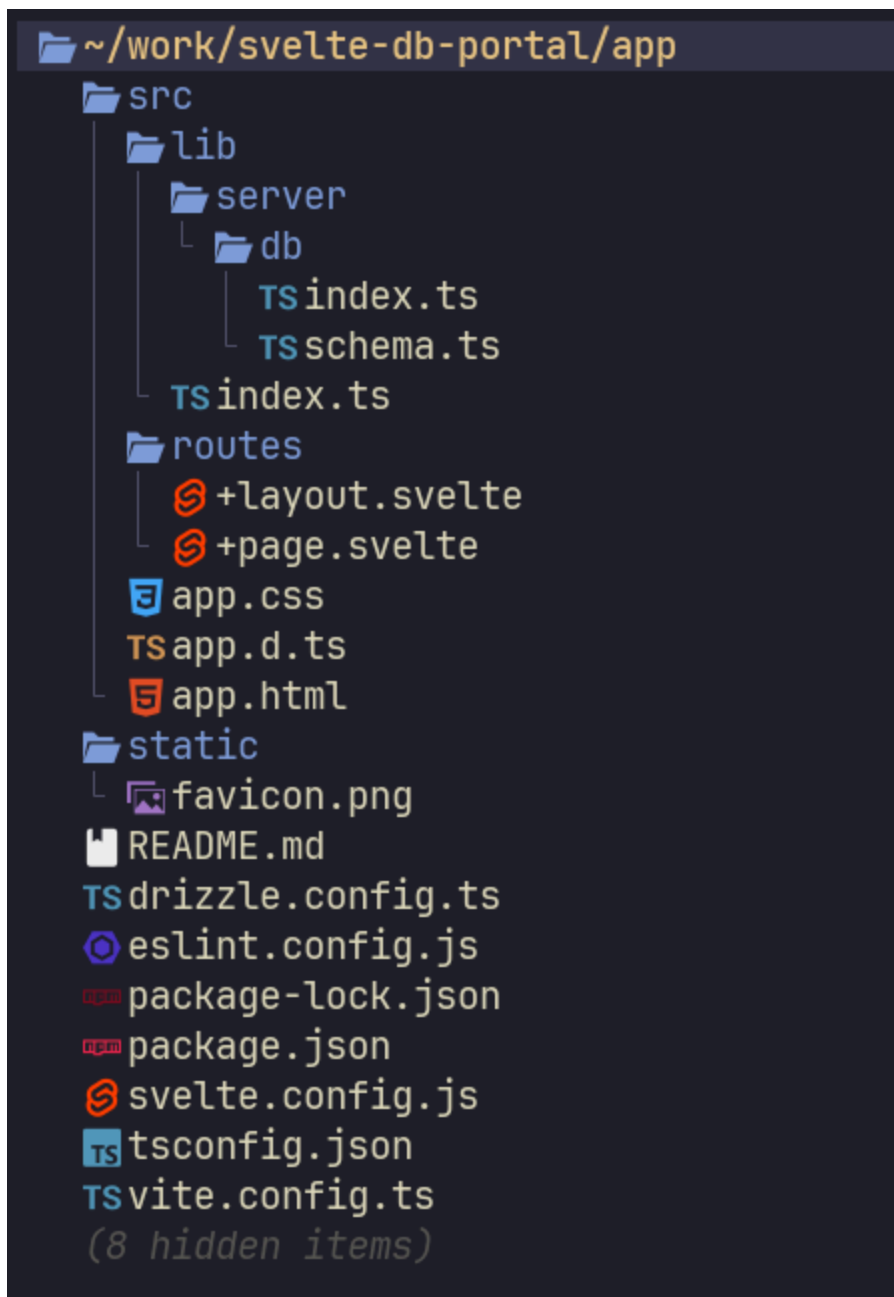
Try to cover at least these topics before continuing (or, at the very least, complete them as you encounter them in this chapter):

- ☐ Basic Svelte
  - ☐ Introduction
  - ☐ Reactivity
  - ☐ Props
    - ☐ Declaring Props
    - ☐ Default Values
  - ☐ Logic
  - ☐ Events
    - ☐ Dom Events
    - ☐ Inline Handlers
  - ☐ Bindings
    - ☐ Text inputs
    - ☐ Numeric inputs
  - ☐ Classes and Styles
- ☐ Basic SvelteKit
  - ☐ Introduction
  - ☐ Routing

It's understandable if you don't understand everything in the interactive tutorial yet, but this should give you a good starting point to follow along with the rest of this chapter.

## Understanding the application structure

The project structure of a Svelte application is described in the [SvelteKit documentation](#) quite well, but let's take a look at the structure of our generated Svelte application (using the `sv` CLI) to understand how it works and what files are important for us to know about.



Let's break down the relevant parts of this structure for now:

- `src/` : This is the main source directory for your Svelte application. All your Svelte components, routes, and styles will be placed here.
  - `src/lib/` : This directory is used for reusable components and utilities. You can create subdirectories here to organize your code better.
  - `src/routes` : This directory contains the routing structure of your application. Each file in this directory corresponds to a route in your application.
    - `+layout.svelte` : This file defines the layout for your application. It is used to wrap all the pages in your application with a common layout (e.g., a navigation bar, footer, etc.).

- `+page.svelte` : This file defines the content for the root route ( / ) of your application. You can create additional `+page.svelte` files in subdirectories to define content for other routes.
- `app.css` : This file contains the global styles for your application. You can import Tailwind CSS here to style your application.
- `app.d.ts` : This file is used to define TypeScript types for your application. It can be used to declare global types or interfaces that are used throughout your application.
- `app.html` : This is the main HTML file for your application. It is used to define the structure of your application and can be modified to include additional meta tags, links, or scripts.
- `static/` : This directory is used to store static assets for your application, such as images, fonts, or other files that do not change.
  - `favicon.png` : This is the favicon for your application. It defines the little icon that appears in a browser tab.
- `README.md` : This file contains information about your application, such as how to run it, how to contribute, and other relevant details.
- `drizzle.config.ts` : This file is used to configure Drizzle ORM, which is a TypeScript ORM for working with databases.
- `package.json` : This file contains the metadata for your application, such as the name, version, dependencies, and scripts.

### Files we won't touch, but are important to know about:

- `package-lock.json` : This file is automatically generated by npm and contains the exact versions of the dependencies installed in your application.
- `eslint.config.js` : This file is used to configure ESLint, which is a tool for identifying and fixing problems in your JavaScript/TypeScript code.
- `svelte.config.js` : This file is used to configure Svelte and its plugins. It can be used to customize the behavior of Svelte and add additional features.
- `tsconfig.json` : This file is used to configure TypeScript for your application. It defines the compiler options and the files that should be included in the compilation.
- `vite.config.ts` : This file is used to configure Vite, which is a build tool for modern web applications. It defines how your application should be built and served.

## Editing our first page

Let's take a look at `src/routes/+page.svelte` :

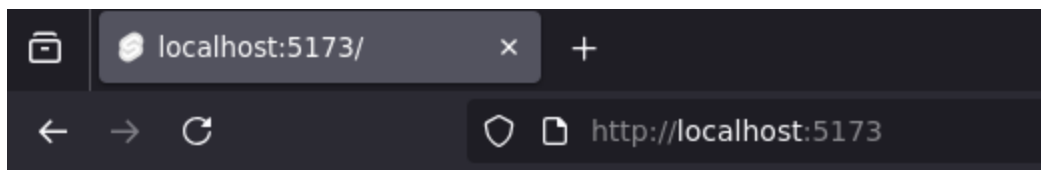
```
<h1>Welcome to SvelteKit</h1>
<p>Visit <a href="https://svelte.dev/docs/kit">svelte.dev/docs/kit</a> to
```

read the documentation</p>

As mentioned above, the `src/routes` directory defines the routing structure of our application. We can see it as the root directory of our application. The `+page.svelte` defines the content for the root route ( / )

We can see what the `+page.svelte` file looks like in the browser by running the application. If you haven't done so already, run `npm run dev` in the terminal to start the development server. (Ensure you run this in the svelte application's root directory, e.g. `~/work/svelte-db-portal/app` , not in the root of the git repository, e.g. `~/work/svelte-db-portal` ). It's a good idea to keep this terminal open in the background while you work on your application, as it will hot-reload your changes automatically.

With the development server running, open your browser and navigate to <http://localhost:5173/> as prompted. You should see the content of the `+page.svelte` file rendered in the browser:



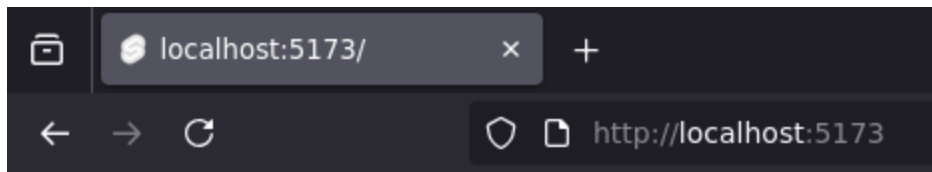
Welcome to SvelteKit  
Visit [svelte.dev/docs/kit](https://svelte.dev/docs/kit) to read the documentation

Let's edit this file to include our own welcome message.

```
<!-- /src/routes/+page.svelte -->

<h1>Welcome to the Web Portal</h1>
<p>We hope you enjoy your stay!</p>
```

And see it update as soon as we save, without having to reload the browser.



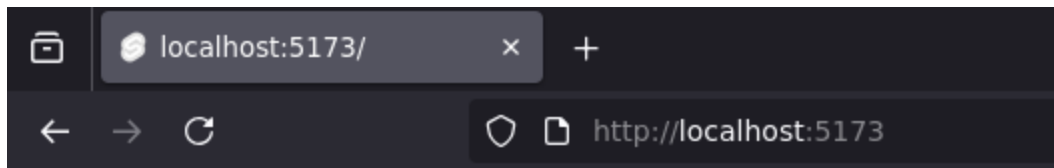
Welcome to the Web Portal  
We hope you enjoy your stay!

Now, we *could* add some styling with the `style` tag and some CSS, and see that update as well:

```
<!-- /src/routes/+page.svelte -->

<h1>Welcome to the Web Portal</h1>
<p>We hope you enjoy your stay!</p>

<style>
  h1 {
    font-size: 2em;
    font-weight: bold;
  }
</style>
```



# Welcome to the Web Portal

We hope you enjoy your stay!

But since we've installed [Tailwind CSS](#), we won't be using those. Instead we'll be using Tailwind's class-based styling instead:

```
<!-- /src/routes/+page.svelte -->

<h1 class="text-3xl font-bold">Welcome to the Web Portal</h1>
<p>We hope you enjoy your stay!</p>
```



Okay, so we know we can do some pretty basic stuff now. Let's look at something a little more advanced.

## Adding our first reactive element (\$state and \$derived)

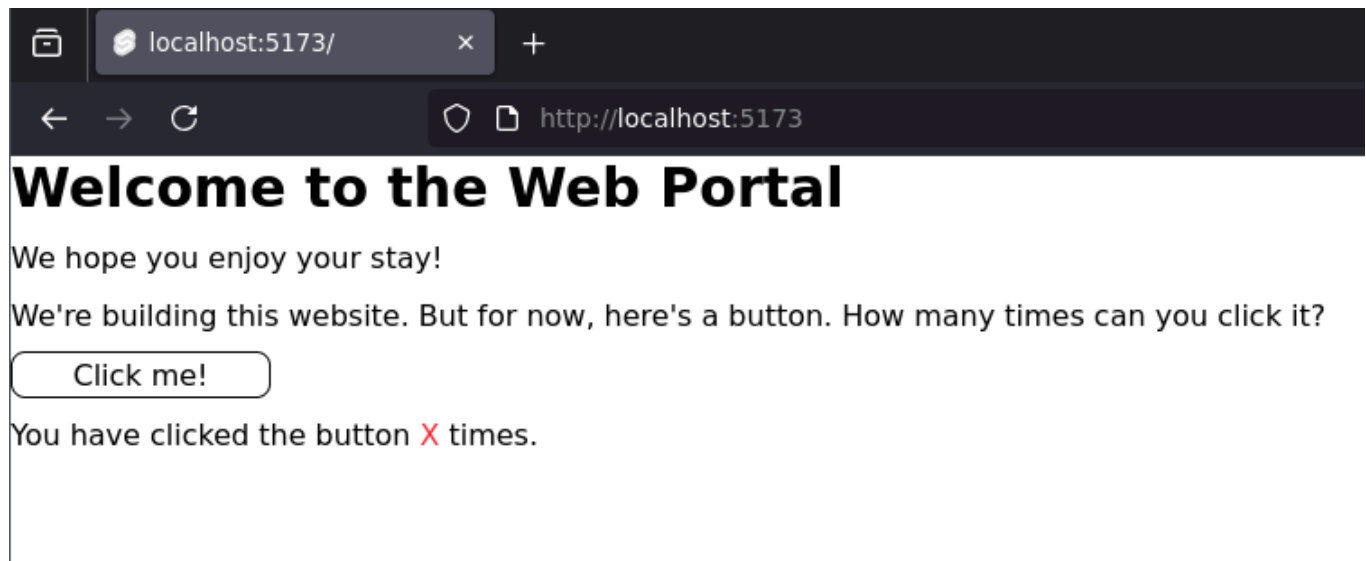
Just to get a feel of how Svelte works, let's add a little button and some text that displays the number of times that we clicked on that button.

```
<!-- /src/routes/+page.svelte -->

<div class="flex flex-col gap-2">
  <h1 class="text-3xl font-bold">Welcome to the Web Portal</h1>
  <p>We hope you enjoy your stay!</p>

  <p>We're building this website. But for now, here's a button. How many
times can you click it?</p>

  <button class="max-w-36 rounded-lg border px-4">Click me!</button>
  <p>You have clicked the button <span class="text-red-500">X</span>
times.</p>
</div>
```



This obviously doesn't do much yet, but let's add that functionality. For this, we need to understand two of the runes that Svelte provides: [\\$state](#) and [\\$derived](#) (click on the links to see the official documentation).

- `$state` : allows you to create reactive state, which means that your UI reacts when it changes.
- `$derived` : allows you to create derived state, which means you can create a variable that *depends* on another variable. It will automatically update when the variable it depends on

changes.

1. We add a `script` tag directly in the `.svelte` page, keeping the logic close to the UI that it affects.
2. We define a variable `buttonClicks` to hold the number of times the button has been clicked.
3. We use the `$state` rune to make this variable reactive, so that when it changes, the UI in which it is referenced will update automatically.
4. We add an `onclick` event handler to the button that increments the `buttonClicks` variable when the button is clicked.  
(We can use an inline handler or refer to a function that we define in the script tag.)
5. We use the `$derived` rune to create a derivative `timeOrTimes` variable that handles pluralization based on the number of clicks, so that it will display "time" for 1 click and "times" for 0 or more than 1 clicks.
6. We replace the `X` and `times` text in the paragraph with references to our reactive variables using the `{}` syntax.

```
<!-- /src/routes/+page.svelte -->

<script lang="ts">
  // The $state rune is used to create a reactive state variable.
  let buttonClicks: number = $state(0);
  // The $derived rune is used to create a derived variable that updates
  when its dependencies change.
  let timeOrTimes: string = $derived(buttonClicks === 1 ? 'time' :
  'times');
</script>

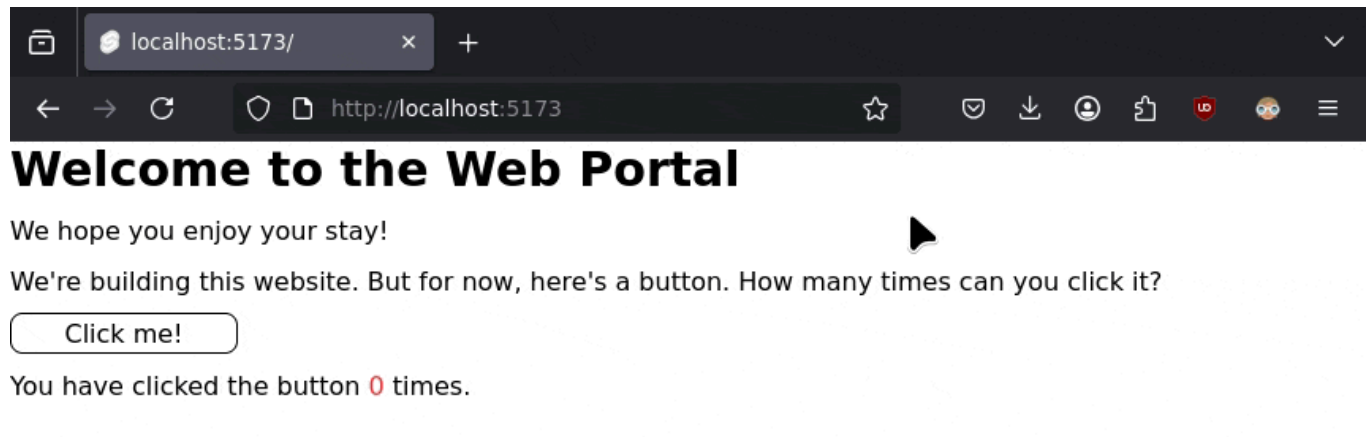
<div class="flex flex-col gap-2">
  <h1 class="text-3xl font-bold">Welcome to the Web Portal</h1>
  <p>We hope you enjoy your stay!</p>

  <p>We're building this website. But for now, here's a button. How many
  times can you click it?</p>

  <!-- The button will increment the buttonClicks state variable when
  clicked. -->
  <button class="max-w-36 rounded-lg border px-4" onclick={() =>
  buttonClicks++}>Click me!</button>
  <!-- The text below will update reactively based on the buttonClicks
  state variable. -->
  <p>You have clicked the button <span class="text-red-500">{buttonClicks}
```

```
</span> {timeOrTimes}.</p>
</div>
```

Let's see it in action:



Cool! It's a simple example, but gives a good idea of how Svelte's reactivity works. Now let's see some of SvelteKit's routing features in action.

## Adding pages and routing

To add a new page to our application, we first need to create a new directory in the `src/routes` directory.

The name of that directory will be the name of the route, so to create an "About" page, we would create a new directory called `about` in the `src/routes` directory, and then add a `+page.svelte` file inside that directory:

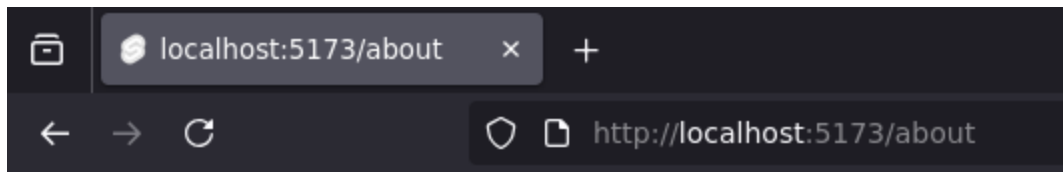
```
mkdir src/routes/about
touch src/routes/about/+page.svelte
```

Now, let's add some content to it:

```
<!-- /src/routes/about/+page.svelte -->

<h1 class="text-3xl font-bold">About us.</h1>
<p>This is some very important information about us.</p>
```

If we then navigate to <http://localhost:5173/about> in our browser, we should see the content of the `about` page:



## About us.

This is some very important information about us.

So what if we wanted to add some sub-pages to, like a dashboard page at `/dashboard`, a dashboard sub-page user at `/dashboard/user` and another dashboard sub-page sessions at `/dashboard/session`?

Well, we would just create the `src/routes/dashboard` directory as we did above, add subdirectories inside the `dashboard` directory, and add `+page.svelte` files inside those directories as well:

```
<!-- /src/routes/dashbaord/+page.svelte -->

<h1 class="text-3xl font-bold">Dashboard.</h1>
<p>This is where the dashboard will live.</p>

<!-- /src/routes/dashboard/user/+page.svelte -->

<h2 class="text-xl font-bold">User Page</h1>
<p>This is the user page.</p>

<!-- /src/routes/dashboard/session/+page.svelte -->

<h2 class="text-xl font-bold">Session Page</h1>
<p>This is the session page.</p>
```

And again,

- given the route `/dashboard`, we can navigate to <http://localhost:5173/dashboard> to see the dashboard page.
- given the route `/dashboard/user`, we can navigate to <http://localhost:5173/dashboard/user> to see the user sub-page.
- given the route `/dashboard/session`, we can navigate to <http://localhost:5173/dashboard/session> to see the sessions sub-page.

Currently, the dashboard pages are all separate pages, so the Dashboard header is not visible on the user and session pages. We can however consolidate the dashboard pages to share a

common layout, which we will do in the next section.

## Using `+layout.svelte` and `$props`

The `+layout.svelte` file is used to define a layout for your application. It allows you to wrap all the pages in your application with a common layout, such as a navigation bar, footer, or other components that should be present on every page.

Each `+layout.svelte` file applies to all the pages in its directory and any subdirectories. For example, a `+layout.svelte` file in the `src/routes` directory will apply to all the pages in the application, while a `+layout.svelte` file in the `src/routes/dashboard` directory will only apply to the pages in the `dashboard` directory and its subdirectories.

Let's take a look at the `src/routes/+layout.svelte` file that was generated for us:

```
<!-- /src/routes/+layout.svelte -->
<script lang="ts">
  import '../app.css';

  let { children } = $props();
</script>

{@render children()}
```

We can see that it exposes a `children` prop using the `$props()` [rune](#), which is used for passing data between pages (and components). This is a special property that will automatically contain the content of the sub layouts and sub pages that are rendered inside this layout.

The `{@render children()}` syntax is used to render the content of the `children` prop. We can write markup around this to define the layout of our application.

For example, let's add a simple navigation bar to our layout.

```
<!-- /src/routes/+layout.svelte -->

<script lang="ts">
  import '../app.css';
  import type { Page } from '$lib/types/page';

  let { children } = $props();
</script>

<div class="flex gap-x-4">
```

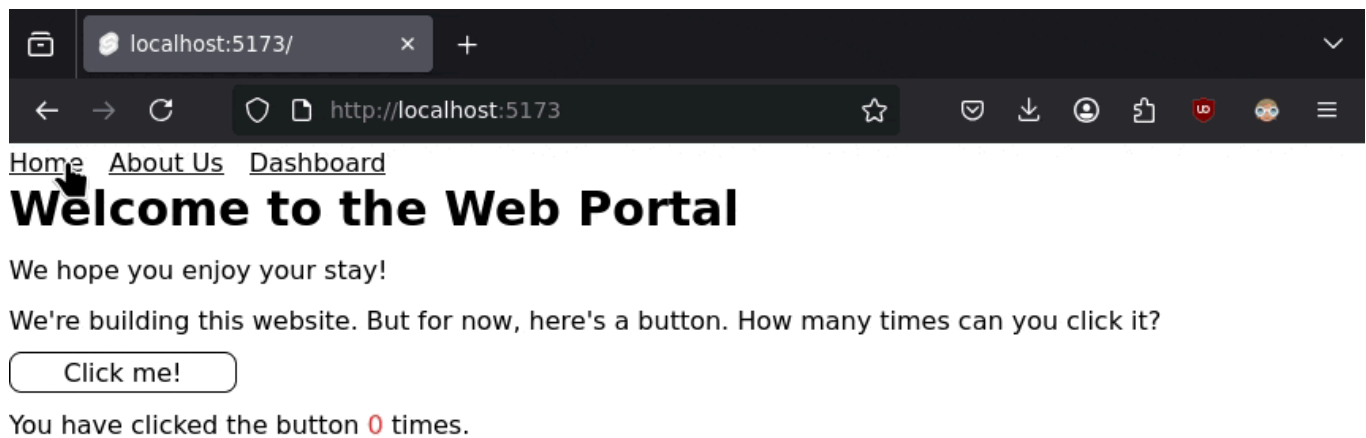
```

    <a href="/" class="underline">
      Home
    </a>
    <a href="/about" class="underline">
      About Us
    </a>
    <a href="/dashboard" class="underline">
      Dashboard
    </a>
  </div>

  {@render children()}

```

We should see the navigation links appear on any page, before the content:



Okay, cool! But we have yet to consolidate our dashboard paages to share a common layout. Let's see how we can do that now.

As said, the `+layout.svelte` files applies to all the pages in its directory *and any subdirectories*. So, to create a layout for the dashboard pages, we can create a new `+layout.svelte` file in the `src/routes/dashboard` directory:

```

<!-- /src/routes/dashboard/+layout.svelte -->

<script lang="ts">
  let { children } = $props();
</script>

<h1 class="text-3xl font-bold">Dashboard.</h1>

<div class="flex gap-x-4">

```

```

    <a href="/dashboard/user" class="underline">
      User
    </a>
    <a href="/dashboard/session" class="underline">
      Sessions
    </a>
  </div>

  <!-- Here, the content from the sub-routes renders -->
  {@render children()}

```

We can now alter the contents of the `src/routes/dashboard/+page.svelte` file to make the header a sub-header, as we already have a header in the layout:

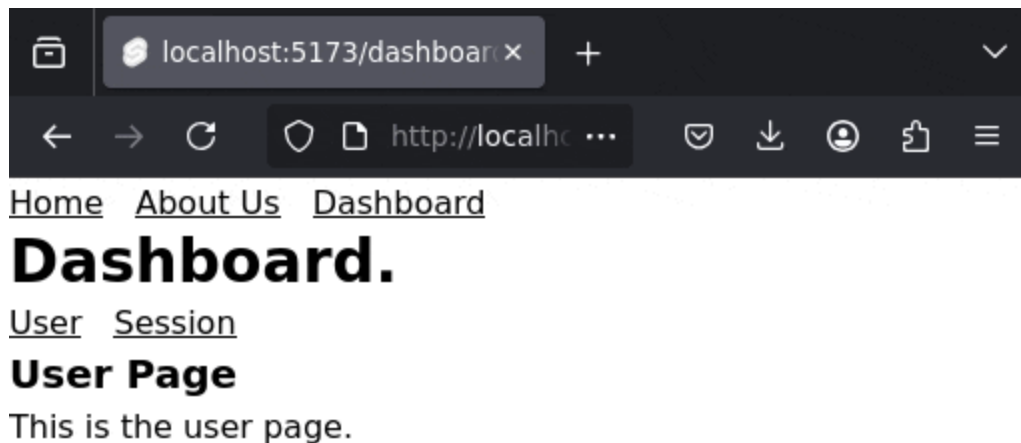
```

<!-- /src/routes/dashboard/+page.svelte -->

<h2 class="text-xl font-bold">Dashboard home.</h2>
<p>This is where the dashboard will live.</p>

```

Now we can see that the dashboard pages share a common layout:



Cool!

However, we now have some duplicate markup for our navigation in both our `src/routes/+layout.svelte` file as well as our `src/routes/dashboard/+layout.svelte` file that doesn't really *directly relate* to defining a layout. Besides, the urls are hardcoded. We can do much better. Applying the [Single Responsibility Principle](#), we should extract this

navigation logic into a separate component, and then use that component in our layout. Luckily, Svelte makes this easy to do.

## Creating your first Svelte component

A Svelte component is a self-contained piece of UI, and can contain its own logic, styles, and markup. A component can also accept props, which are used to pass data from the parent component to the child component.

Creating a new component is as simple as creating a new `.svelte` file. The recommended location for reusable components is the `src/lib` directory. We can create subdirectories to organize our components better.

Let's move some of our navigation markup from `src/routes/+layout.svelte` to a new `NavBar.svelte` component in the `src/lib/ui/nav` directory:

```
<!-- /src/lib/ui/nav/NavBar.svelte -->

<div class="flex gap-x-4">
  <a href="/" class="underline">
    Home
  </a>
  <a href="/about" class="underline">
    About Us
  </a>
  <a href="/dashboard" class="underline">
    Dashboard
  </a>
</div>
```

And now, let's import this component in our `/src/routes/+layout.svelte` file. (`$lib` is a special alias that points to the `src/lib` directory, so we can import our component like this):

```
<!-- /src/routes/+layout.svelte -->

<script lang="ts">
  import './app.css';
  import NavBar from '$lib/ui/nav/NavBar.svelte';

  let { children } = $props();
</script>
```



```
<NavBar />
{@render children()}
```

That already looks much better, but this component isn't very reusable just yet.

We should look into re-using this component in our `src/routes/dashboard/+layout.svelte` file as well, but to do that we first need to understand how to pass props to components, and how to use the `#each` block to iterate over arrays in Svelte.

## Using the `#each` block

To make our navigation bar more easy to maintain, we can make use of the `#each` block to iterate over an array of pages and render the links dynamically. This way, if we want to add or remove pages, we only need to update the array, rather than the markup.

Let's modify our `NavBar.svelte` component to use the `#each` block:

```
<!-- /src/lib/ui/nav/NavBar.svelte -->

<script lang="ts">
  let pages = [
    { name: 'Home', url: '/', description: 'The home page.' },
    { name: 'About Us', url: '/about', description: 'Learn more about us.' },
    { name: 'Dashboard', url: '/dashboard', description: 'View the dashboard.' }
  ]
</script>

<div class="flex gap-x-4">
  {#each pages as page (page.url)}
    <a
      href={page.url}
      class="underline"
      title={page.description}
    >
      {page.name}
    </a>
  {/each}
</div>
```

Hopefully you can already see where this is going. We have an array of `pages`, and we have already learned we can use `$props()` to pass data to components. So eventually, we'll be removing any information about the pages from our `NavBar.svelte` component, and instead pass it from the parent component.

Before we get into that, we must take another quick detour to understand how to add type safety to our components using TypeScript, so we know we'll be passing the correct data around.

## Using Typescript in Svelte components

Svelte supports TypeScript out of the box, which allows us to define types for our variables and props. This can help us catch errors early and improve the maintainability of our code.

For type safety, let's extrapolate the `PageInfo` type into an `interface` in a separate file, so that we can easily reuse it in other components or files, should we need to.

Create a new file at `src/lib/types/page.ts` and define the `PageInfo` interface:

```
// src/lib/types/pageInfo.ts

export interface PageInfo {
  name: string;
  description: string;
  url: string;
}
```

Now we can import and use this `PageInfo` interface in our `NavBar.svelte` component:

```
<!-- /src/lib/ui/nav/NavBar.svelte -->

<script lang="ts">
  import type { PageInfo } from '$lib/types/pageInfo';

  let pages: PageInfo[] = [
    { name: 'Home', url: '/', description: 'The home page.' },
    { name: 'Dashboard', url: '/dashboard', description: 'View the
dashboard.' },
    { name: 'About Us', url: '/about', description: 'Learn more about
us.' }
  ];
</script>

<div class="flex gap-x-4">
  {#each pages as page (page.url)}
    <a href={page.url} class="underline" title={page.description}>
      {page.name}
    </a>
```

```
    {/each}  
</div>
```

It may look like we haven't changed much, but we have now added type safety to our `pages` array, which will help us catch errors early should we try to add a property with an incompatible type, or forget to add a required property:

```
<!-- /src/lib/nav/NavBar.svelte -->  
  
<script lang="ts">  
  import type { PageInfo } from '$lib/types/pageInfo';  
  
  let pages: PageInfo[] = [  
    { name: 'Home', url: '/', description: 'The home page.' },  
    { name: 'About Us', url: '/about', description: 'Learn more about us.' },  
    { name: 'Dashboard', url: '/dashboard', description: 'View the dashboard.' },  
    { name: 'Contact', link: '/contact' }  
  ];  
  // 2353: Object literal may only specify known properties, and 'link' does not exist in type 'PageInfo'.  
</script>  
  
<div class="flex gap-x-4">  
  {#each pages as page (page.url)}  
    <a href={page.url} class="underline" title={page.description}>  
      {page.name}  
    </a>  
  {/each}  
</div>
```

Awesome! This will definitely help us catch errors later as we increase the complexity of our application.

Now, as promised, we can start making the `NavBar.svelte` component more reusable by exposing the `pages` array as a prop, and passing the data from the parent component instead of hard-coding it in the component itself.

## Reusing Svelte components with \$props()

Currently, our `NavBar.svelte` component has the `pages` array hard-coded inside the component. We want to make it more reusable by passing the `pages` array as a prop from the parent component. This way, we can re-use the `NavBar` component to display the links to Home, About Us, and Dashboard, but also to display the links to the Dashboard sub-pages.

To do this, let's expose the `pages` array as a prop in `NavBar.svelte`:

```
<!-- /src/lib/ui/nav/NavBar.svelte -->  
  
<script lang="ts">  
  import type { PageInfo } from '$lib/types/pageInfo';  
  
  let { pages }: { pages: PageInfo[] } = $props();  
</script>  
  
<div class="flex gap-x-4">
```

```

    {#each pages as page (page.url)}
      <a href={page.url} class="underline" title={page.description}>
        {page.name}
      </a>
    {/each}
  </div>

```

And pass the `pages` array in `+src/routes/+layout.svelte` instead:

```

<!-- src/routes/+layout.svelte -->

<script lang="ts">
  import '../app.css';
  import NavBar from '$lib/ui/nav/NavBar.svelte';

  let pages: PageInfo[] = [
    { name: 'Home', url: '/', description: 'The home page.' },
    { name: 'About Us', url: '/about', description: 'Learn more about us.' },
    { name: 'Dashboard', url: '/dashboard', description: 'View the dashboard.' }
  ];

  let { children } = $props();
</script>

<NavBar pages={pages} />
{@render children}

```

Let's do the same for our `src/routes/dashboard/+layout.svelte` file, so that we can re-use the `NavBar` component there as well:

```

<!-- /src/routes/dashboard/+layout.svelte -->

<script lang="ts">
  import NavBar from '$lib/ui/nav/NavBar.svelte';

  let pages: PageInfo[] = [
    { name: 'User', url: '/dashboard/user', description: 'Query user information' },
    { name: 'Session', url: '/dashboard/session', description: 'View session details.' }
  ];

  let { children } = $props();

```

```
</script>

<h1 class="text-3xl font-bold">Dashboard.</h1>

<NavBar pages={pages} />

<!-- Here, the content from the sub-routes renders -->
{@render children()}
```

Okay, that looks much better. But the keen eyed among you may have noticed that we have now re-introduced an earlier problem. The `pages` array is now stored in two different places, and neither of them should really be responsible for storing the navigation data anyway. Let's fix that using constants.

## Reusing immutable values across components

We have a few immutable values in our application, such as the `pages` array and the `dashboardPages` array. We can store these in a separate file and import them into our components, so that we can re-use them across our application without duplicating the data.

Let's create a new file at `src/lib/constants/navigation.ts` and define our `PAGES` and `DASHBOARD_PAGES` constants there:

```
// /src/lib/constants/navigation.ts

import { PanelsTopLeft, User, Dumbbell } from 'lucide-svelte';
import type { PageInfo } from '$lib/types/pageInfo';

export const PAGES: PageInfo[] = [
  { name: 'Home', url: '/', description: 'The home page.' },
  { name: 'Dashboard', url: '/dashboard', description: 'View the dashboard.' },
  { name: 'About Us', url: '/about', description: 'Learn more about us.' }
];

export const DASHBOARD_PAGES: PageInfo[] = [
  {
    name: 'Overview',
    url: '/dashboard/',
    description: 'Dashboard overview.',
    icon: PanelsTopLeft
  },
  {
    name: 'User',
    url: '/dashboard/user',
```

```

        description: 'Query user information',
        icon: User
    },
    {
        name: 'Session',
        url: '/dashboard/session',
        description: 'View session details.',
        icon: Dumbbell
    }
];

```

And use the stores in our `+layout.svelte` files instead (note the `$` prefix to access the store's value):

```

<!-- src/routes/+layout.svelte -->

<script lang="ts">
    import '../app.css';
    import NavBar from '$lib/ui/nav/NavBar.svelte';
    import { PAGES } from '$lib/constants/navigation';

    let { children } = $props();
</script>

<NavBar pages={PAGES} />
{@render children()}

<!-- /src/routes/dashboard/+layout.svelte -->

<script lang="ts">
    import NavBar from '$lib/ui/nav/NavBar.svelte';
    import { DASHBOARD_PAGES } from '$lib/constants/navigation';

    let { children } = $props();
</script>

<h1 class="text-3xl font-bold">Dashboard.</h1>

<NavBar pages={DASHBOARD_PAGES} />

<!-- Here, the content from the sub-routes renders -->
{@render children()}

```

Now we have a single source for our navigation data. We can easily add or remove pages from the `navigation.ts` file, and re-use the `NavBar` component across our root and dashboard `+layout.svelte` files without duplicating the navigation data.

## Wrapping up

We now know how to:

- Create a Svelte application using SvelteKit.
- Navigate the structure of a Svelte application.
- Add content, typescript, and styles to a Svelte application.
- Create pages and layouts using `+page.svelte` and `+layout.svelte` files.
- Create reusable components in Svelte.
- Use TypeScript in Svelte components to add type safety.
- Use the `$state` and `$derived` runes to create reactive variables and derived state.
- Use the `$props()` rune to pass data between components.
- Re-use immutable data across components using constants.

We have also built a simple web portal with a home page, an about page, and a dashboard with user and session pages, all using SvelteKit and Tailwind CSS.

Now that we have some experience with Svelte as a web development framework, let's try to create something a little more pretty. In the next chapter, we will learn how to add some pretty UI to our application.

See [2.0 - UI](#) for more information.