2 - UI

Until now, we have been focusing on the functionality of our application, creating a small website with a couple of pages and an interactive button. In this chapter, we will start building out the user interface (UI) for our application. We will make the website more visually appealing and user-friendly, and ensure it works well on different devices and screen sizes. We will use Skeleton, an apdative design system powerd by Tailwind CSS, to help us create a responsive and modern UI.

Chapter overview

Learning goals

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

- Install and use lucide-svelte to render icons in Svelte components.
- Set up the Skeleton UI framework with Tailwind CSS in a SvelteKit project.
- Apply Skeleton utility classes to style components.
- Use Skeleton components to build a responsive, mobile-first, themeable layout.
- Build a reusable theme switcher component that stored user preferences using localStorage and Svelte state.
- Implement a dynamic NavBar that switches between modal and inline navigation.
- Use Skeleton's Navigation.Rail and Navigation.Bar components for section-based navigation.
- Create a flexible Card component using Svelte snippets and Tailwind styling.

Learning Resources

- Lucide Svelte installation
- Lucide icons database
- Skeleton installation
- Skeleton themes
- Responsive design with Tailwind
- Dark mode with Tailwind
- Theme variables with Tailwind

Lucide

First, let's talk about Lucide. It is a community-based collection of beautiful icons that can be used in web applications. We'll be using `lucide-svelte` throughout our UI, but first we need to install it. (Skeleton also uses Lucide for some icons.)

Installing Lucide

Always refer to the latest instructions on the Lucide website for installation, as the process may change over time. All we had to do was run `npm install lucide-svelte`.

Using Lucide

For future reference, to use Lucide we simply import the icons we want to use in our Svelte components. For example, to use the `sun` and `moon` icons, we can do the following:

```
<script>
    import { Sun, Moon } from 'lucide-svelte';
</script>

<Sun />
<Moon />
```

Cool. If we now need icons later, we have a good source for them.

Skeleton

Skeleton is a design system powered by Tailwind CSS that provides a set of pre-designed components and utilities to help you build responsive and modern user interfaces quickly. Using a design system like Skeleton allows us to spend less time on creating components, and more time on the functionality of our application while still ensuring a consistent and visually appealing design. Skeleton works nicely with Svelte, allowing us to create reusable components that can be easily styled and customized.

Installing Skeleton

To get started with Skeleton, we first need to install it in our Svelte project. Follow the latest instructions on the Skeleton website for installation, as the process may change over time. We'll still walk through the steps we took here, but they might be outdated.

1. We install Skeleton by running the following command at the Svelte app's root directory:

   ```
   npm i -D @skeletonlabs/skeleton @skeletonlabs/skeleton-svelte
   ```

2. Next, we need to import Skeleton's styles into our Svelte app. Open `src/app.css` and add Skeleton's imports, (note it imports the `cerberus` theme by default):

   ```
   /* /src/app.css */

   @import 'tailwindcss';

   @import '@skeletonlabs/skeleton';
   @import '@skeletonlabs/skeleton/optional/presets';
   @import '@skeletonlabs/skeleton/themes/cerberus';

   @source '../node_modules/@skeletonlabs/skeleton-svelte/dist';
   ```

3. Now, we add the `data-theme` attribute to the `<html>` tag in `src/app.html` to apply the cerberus theme:

   ```
   <!doctype html>
   <html lang="en" data-theme="cerberus">
       <head>
           <meta charset="utf-8" />
           <link rel="icon" href="%sveltekit.assets%/favicon.png" />
           <meta name="viewport" content="width=device-width, initial-scale=1" />
   ```

```
            %sveltekit.head%
        </head>
        <body data-sveltekit-preload-data="hover">
            <div style="display: contents">%sveltekit.body%</div>
        </body>
    </html>
```

4. (Re)run the development server and see that our theming has now changed: install-theme.png

Basic Skeleton components

Now that we have Skeleton installed, let's try changing our earlier button to a themed button using Skeleton's components (powered by Tailwind CSS).

After a quick look at the Skeleton documentation, we can just copy the example code for a button into our `src/routes/+page.svelte` file:

```
<!-- src/routes/+page.svelte -->
...
<button class="btn preset-filled-primary-500 max-w-36" onclick={() => buttonClicks++}>
    Click me!
</button>
...
```

And we'll see that our button has now changed to a themed button with a color according to the Skeleton Color System :

themed-button.png

Now, this is a nice looking button, but Skeleton can do more than just buttons. First though, let's take a look at how we can change the theme of our website, and then we'll explore some more advanced Skeleton components to switch between those themes.

Skeleton Themes and Dark Mode

Before we start adding more complex UI to our application, it's a good idea to set up some themes and dark mode support, so we can design our UI with those in mind. This is definitely not a requirement, but it will make our application look more polished, and the best time to implement this is now.

First, let's add some themes of your choice and a dark mode toggle to our `app.css` file.

```
/* /src/app.css */

@import 'tailwindcss';

/* Skeleton imports */
@import '@skeletonlabs/skeleton';
@import '@skeletonlabs/skeleton/optional/presets';
@source '../node_modules/@skeletonlabs/skeleton-svelte/dist';
```

```
/* Skeleton themes */
@import '@skeletonlabs/skeleton/themes/catppuccin';
@import '@skeletonlabs/skeleton/themes/cerberus';
@import '@skeletonlabs/skeleton/themes/vintage';
@import '@skeletonlabs/skeleton/themes/modern';

/* Tailwind dark mode toggle */
@custom-variant dark (&:where([data-mode=dark], [data-mode=dark] *));
```

You can try switching to one of the new themes by changing the `data-theme` attribute in the `<html>` tag in `src/app.html`.

You can also try enabling the dark mode by adding a `data-mode="dark"` attribute to the `<html>` tag in `src/app.html`.

```
<!-- /src/app.html -->

<html lang="en" data-theme="catppuccin" data-mode="dark">
```

As you can see, it is pretty easy to switch the theme of our website. It would be nice if we could give the user the ability to switch between themes and dark mode themselves, though. Let's create a theme switcher using some more advanced Skeleton components.

Switching themes and storing user preferences

We now know how to change the theme of our website using `data` attributes, but we want to give the user the ability to switch between them, and store their preference between sessions.

Looking at the Skeleton documentation we can see that there's a component `Popover` which could be a useful component for creating a Theme Switch.

Let's take a look at the example code for the `Popover` component:

```
<script lang="ts">
  import { Popover } from '@skeletonlabs/skeleton-svelte';
  import IconX from '@lucide/svelte/icons/x';

  let openState = $state(false);

  function popoverClose() {
    openState = false;
  }
</script>

<Popover
  open={openState}
  onOpenChange={(e) => (openState = e.open)}
  positioning={{ placement: 'top' }}
  triggerBase="btn preset-tonal"
  contentBase="card bg-surface-200-800 p-4 space-y-4 max-w-[320px]"
```

```
    arrow
    arrowBackground="!bg-surface-200 dark:!bg-surface-800"
  >
    {#snippet trigger()}Click Me{/snippet}
    {#snippet content()}
      <header class="flex justify-between">
        <p class="font-bold text-xl">Popover Example</p>
        <button class="btn-icon hover:preset-tonal" onclick={popoverClose}><IconX /></button>
      </header>
      <article>
        <p class="opacity-60">
          This will display a basic popover with a header and body. This also includes a title,
        </p>
      </article>
    {/snippet}
  </Popover>
```

Let's walk through this code:

- It imports the `Popover` component from Skeleton, and an icon from Lucide.
- It creates a state variable `openState` that manages whether the popover is open or closed.
    - The `onOpenChange` event handler updates the `openState` variable when the popover is opened.
- The `Popover` component is used with several props:
    - `open`: controls whether the popover is open or closed.
    - `onOpenChange`: updates the `openState` when the popover's open state changes.
    - `positioning`: sets the placement of the popover (in this case, at the top).
    - `triggerBase`: defines the base class for the trigger button.
    - `contentBase`: defines the base class for the content of the popover.
    - `arrow`: adds an arrow/speechbubble indicator to the popover.
    - `arrowBackground`: sets the background color of the arrow.
- We see the first use of a Svelte `snippet` here, which allows us to pass a chunk of markup to the `Popover` component. Inside the `Popover` source code (which can be found here) we can see that the `trigger` and `content` snippets are rendered using the `{@render snippet}` syntax. This way we can re-use components and alter sections of the component without having to rewrite the entire component.
- The button calls `popoverClose` when clicked, which sets `openState` to `false`, closing the popover.
- There's a `header` section with a title and a close button, and an `article` section for the contents of the popover.

Now, let's save this `Popover` code in a new file `src/lib/ui/control/ThemeSwitch.svelte`, so we can later alter it to suit our needs.

> ℹ️ You may notice that the import for the `IconX` throws an error. In our case, we had to replace it with `import { X } from 'lucide-svelte';`, and replace the `<IconX />` component with `<X />`.

We'll also import the `ThemeSwitch` into our `src/routes/+layout.svelte` file:

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

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

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

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

Theme Constants

First, for type safety, we will add a new Type Theme in `src/lib/types/theme.ts`

```
// /src/lib/types/theme.ts
export interface Theme {
        label: string;
        value: string;
}
```

Next, we will create a file that holds the available themes. This will allow us to easily access the themes throughout our application and ensure that we can update the list of themes in one place if needed.

```
// /src/lib/constants/themes.ts

import type { Theme } from '$lib/types/theme';

// The label is the display name of the theme,
// while the value is the actual theme name used
// in the `data-theme` attribute.
export const THEMES: Theme[] = [
        { label: '🐱 Catppuccin', value: 'catppuccin' },
        { label: '🐺 Cerberus', value: 'cerberus' },
        { label: '📺 Vintage', value: 'vintage' },
        { label: '🌸 Modern', value: 'modern' }
];
```

Theme Switcher Markup

Let's use this store in our `ThemeSwitch` component.

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

```
	import { THEMES } from '$lib/constants/themes';
	...
</script>

...

<!-- In the content snippet of the popover, we will create a form with a dropdown to select a
<article>
	<form>
		<label class="label">
			<span class="label-text">Select</span>
			<!-- The value of the select is bound to selectedTheme, a variable we
			<!-- We couple the onchange event to a function handleThemeSelect, wh
			<select class="select" bind:value={selectedTheme} onchange={handleThe
				<!-- Loop through the themes store to create an option for ea
				{#each THEMES as theme (theme.value)}
					<option value={theme.value}>{theme.label}</option>
				{/each}
			</select>
		</label>
	</form>
</article>
```

While we're at it, let's also replace the header of the popover content with a more descriptive title, and add some styling to the close button:

```
<header class="flex justify-between">
	<p class="text-xl font-bold">Choose Theme</p>
	<button class="btn-icon hover:preset-tonal" onclick={popoverClose}><X /></button>
</header>
```

And change the `trigger` snippet to use a Lucide icon for the trigger button:

```
<script lang="ts">
	import { X, Palette } from 'lucide-svelte';
</script>

...

{#snippet trigger()}<Palette />{/snippet}
```

Finally, we will add a Switch component to toggle dark mode. (Yes, we can use Skeleton components inside other Skeleton components!)

```
<script lang="ts">
	import { Popover, Switch } from '@skeletonlabs/skeleton-svelte';
</script>
```

```
...
<article> ... </article>
<span class="flex">
                        <!-- Switch to toggle dark mode -->
                        <span class="label">Dark Mode</span>
                        <!-- The checked state is set to whether darkMode is on or not. We cı
                        <!-- The onCheckedChange event is coupled to a function handleDarkMoc
                        <Switch name="darkMode" checked={darkMode} onCheckedChange={handleDaı
</span>
```

Now, that is the markup done for our basic `ThemeSwitch` component, but we still need to implement the logic so that it actually works.

Theme Switcher Logic

First, let's add some state variables to our `ThemeSwitch` component to keep track of the selected theme and dark mode state. We already refer to these in the markup.

```
// We'll leave them undefined for now
let selectedTheme: string | undefined = $state();
let darkMode: boolean | undefined = $state();
```

Next, we will use the onMount lifecycle hook to retrieve the stored theme and dark mode state from `localStorage` when the component is mounted. This function runs once the component is added to the DOM, allowing us to initialize the variables with the stored values. If no theme or mode is stored, we will retrieve the default values from the HTML document.

```
import { onMount } from 'svelte';

onMount(() => {
        // Retrieve the stored theme and mode (if any), or use the default theme given in the
        const storedTheme =
                localStorage.getItem('theme.name') || document.documentElement.dataset.themeI
        const storedMode =
                (localStorage.getItem('theme.mode') || document.documentElement.dataset.modeI
                'dark';
        // Set the initial values for selectedTheme and darkMode based on the stored values.
        selectedTheme = storedTheme;
        darkMode = storedMode;
});
```

We need to add these default values to the `<html>` tag in `src/app.html`:

```
<html lang="en" data-theme-default="catppuccin" data-mode-default="dark">
```

Back in our `ThemeSwitch` component, let's implement the `handleThemeSelect` function and `handleDarkModeSelect` functions to update the selected theme and dark mode state when the user interacts with the dropdown and switch, respectively. You can use the `svelte` LSP to generate the function signatures (alt+enter in

VSCode while your cursor is on the unimplemented function) and then fill in the logic.

```
// This function is called when the user selects a theme from the dropdown.
function handleThemeSelect(event: Event & { currentTarget: EventTarget & HTMLSelectElement }
        // When the user selects a theme, update the selectedTheme variable and store it in l
        selectedTheme = event.currentTarget.value;
        localStorage.setItem('theme.name', selectedTheme);
        document.documentElement.dataset.theme = selectedTheme;
}

// This function is called when the user toggles the dark mode switch.
function handleDarkModeSelect(details: { checked: boolean }): void {
        // When the user toggles dark mode, update the darkMode variable and store it in loca
        darkMode = details.checked;
        const mode = darkMode ? 'dark' : 'light';
        localStorage.setItem('theme.mode', mode);
        document.documentElement.dataset.mode = mode;
}
```

Finally, since we're now storing our choices in `localStorage`, we'll need to update our `app.html` one more time to load the theme and mode from `localStorage`. We can do this by adding a small script directly into the <head> of our `src/app.html` file:

```
<script>
        (() => {
                const html = document.querySelector('html');
                if (!html) return;
                // Set theme and mode from localStorage or data-default attributes
                html.dataset.mode = localStorage.getItem('theme.mode') || html.dataset.modeDe
                html.dataset.theme = localStorage.getItem('theme.name') || html.dataset.theme
        })();
</script>
```

Awesome. Now our `ThemeSwitch` component is fully functional. If you got lost along the way, here's the complete code:

```
<!-- src/app.html -->

<!doctype html>
<html lang="en" data-theme-default="catppuccin" data-mode-default="dark">
        <head>
                <meta charset="utf-8" />
                <link rel="icon" href="%sveltekit.assets%/favicon.png" />
                <meta name="viewport" content="width=device-width, initial-scale=1" />
                <script>
                        (() => {
                                const html = document.querySelector('html');
                                if (!html) return;
```

```
				// Set theme and mode from localStorage or data-default attr
				html.dataset.mode = localStorage.getItem('theme.mode') || htr
				html.dataset.theme = localStorage.getItem('theme.name') || ht
			})();
		</script>
		%sveltekit.head%
	</head>
	<body data-sveltekit-preload-data="hover">
		<div style="display: contents">%sveltekit.body%</div>
	</body>
</html>
```

```
<!-- src/lib/ui/control/ThemeSwitch.svelte -->

<script lang="ts">
	import { THEMES } from '$lib/constants/themes';
	import { Popover, Switch } from '@skeletonlabs/skeleton-svelte';
	import { X, Palette } from 'lucide-svelte';
	import { onMount } from 'svelte';

	// Stores the currently selected theme and dark mode state.
	let selectedTheme: string | undefined = $state();
	let darkMode: boolean | undefined = $state();

	// openState determines whether the popover is open or closed.
	let openState = $state(false);

	// The onMount function is a Svelte lifecycle hook that runs as soon as the componen
	onMount(() => {
		// Retrieve the stored theme and mode (if any), or use the default theme give
		const storedTheme =
			localStorage.getItem('theme.name') || document.documentElement.datase
		const storedMode =
			(localStorage.getItem('theme.mode') || document.documentElement.datas
			'dark';
		// Set the initial values for selectedTheme and darkMode based on the stored
		selectedTheme = storedTheme;
		darkMode = storedMode;
	});

	function popoverClose() {
		openState = false;
	}

	// This function is called when the user selects a theme from the dropdown.
	function handleThemeSelect(event: Event & { currentTarget: EventTarget & HTMLSelectE
		// When the user selects a theme, update the selectedTheme variable and store
		selectedTheme = event.currentTarget.value;
		localStorage.setItem('theme.name', selectedTheme);
```

```
			document.documentElement.dataset.theme = selectedTheme;
		}

		// This function is called when the user toggles the dark mode switch.
		function handleDarkModeSelect(details: { checked: boolean }): void {
			// When the user toggles dark mode, update the darkMode variable and store it
			darkMode = details.checked;
			const mode = darkMode ? 'dark' : 'light';
			localStorage.setItem('theme.mode', mode);
			document.documentElement.dataset.mode = mode;
		}
	</script>

	<Popover
		open={openState}
		onOpenChange={(e) => (openState = e.open)}
		positioning={{ placement: 'top' }}
		triggerBase="btn hover:preset-tonal"
		contentBase="card bg-surface-200-800 p-4 space-y-4 max-w-[320px]"
		arrow
		arrowBackground="!bg-surface-200 dark:!bg-surface-800"
	>
		{#snippet trigger()}<Palette />{/snippet}
		{#snippet content()}
			<header class="flex justify-between">
				<p class="text-xl font-bold">Choose Theme</p>
				<button class="btn-icon hover:preset-tonal" onclick={popoverClose}><X
			</header>
			<article>
				<form>
					<label class="label">
						<span class="label-text">Select</span>
						<!-- Dropdown to select a theme from the available th
						<!-- The value of the select is bound to selectedThem
						<select class="select" bind:value={selectedTheme} onc
							<!-- Loop through the themes to create an opt
							{#each THEMES as theme (theme.value)}
								<option value={theme.value}>{theme.la
							{/each}
						</select>
					</label>
				</form>
			</article>
			<span class="flex">
				<!-- Switch to toggle dark mode -->
				<span class="label">Dark Mode</span>
				<!-- The checked state is set to whether darkMode is on or not -->
				<Switch name="darkMode" checked={darkMode} onCheckedChange={handleDar
			</span>
		{/snippet}
```

```
</Popover>
```

Let's try it out (Note that the drop down is not rendered inside the screen recording software used for the .gif below, but it should appear on your screen!):

theme-switcher.gif

A better layout

Okay, that theme switcher was a bit of work, but now that we have our theming set up, we can start making out application look better.

First, let's tackle the main `+layout.svelte` file. We'll base our website on the One Column layout example from Skeleton.

```
<!-- 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>

<div class="grid min-h-screen grid-rows-[auto_1fr_auto]">
	<!-- Header -->
	<header>
		<NavBar pages={PAGES} />
	</header>
	<!-- Page -->
	<main class="space-y-4">
		{@render children()}
	</main>
	<!-- Footer -->
	<footer></footer>
</div>
```

This layout uses a `grid` with three rows: one for the header, one for the main content, and one for the footer.

The `grid-rows-[auto_1fr_auto]` sets the header and footer to take up only as much space as they need, while the main content area takes up the remaining space (`1fr`).

The `min-h-screen` class ensures that the grid takes up at least the full height of the screen, so the footer will always be at the bottom of the page, even if there is not enough content to fill the screen.

(You may have noticed we have removed the `ThemeSwitch` component from the `+layout.svelte` file, as we will add it to the `NavBar` component later.)

We won't actually use a footer in this application, but let's leave the footer in the layout for now, so we can always add one later if we want to.

We won't see much change just yet, but this sets us up nicely for a responsive layout.

A better NavBar

Next up is the `NavBar` component.

We had a simple `NavBar` component before, but let's expand it with a logo and our `ThemeSwitch` component, while making it prettier and more responsive.

Let's look at an example of the `NavBar` we'll be creating.

navbar.png

That looks pretty, but on smaller screens, these buttons might get a little too cramped. we will responsively switch over to side-bar navigation instead:

modal-navbar.gif

Okay, so how do we create this?

First, let's think about the structure of the `NavBar` component.

It's one big site-wide navigation bar that contains:

- A left aligned Logo section (or a hamburger menu on smaller screens)
- A center navigation section
- A right aligned theme switcher section

Let's design that NavBar.

Responsive Design (Mobile-first)

Generally, it's a good idea to design a website mobile-first. Aka, design for small screens first, then add style overrides for larger screens. This is easier than the other way around, as it is easier to expand a design to fit more information than to shrink it down to fit the same amount of information on a small screen.

With tailwind, we can use screen size prefixes to apply styles conditionally based on the screen size.

For example, if we add the `grid` class to an element, we could have it display as a 1-column grid on small screens, but as the screen grows larger, we can change it to a 2-column (or more) grid:

```
grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3
```

Or another example, we can use the `hidden` class to hide an element on small screens, but show it on larger screens:

```
hidden md:block
```

So generally, we write the css directives for small screens first, and then add the screen size prefixes to override them for larger screens.

NavBar Markup

Let's start with the markup for our small screen NavBar. We'll use the Modal component from Skeleton to create a hamburger menu that opens a side-bar navigation when clicked.

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

<script lang="ts">
	import { Modal } from '@skeletonlabs/skeleton-svelte';
	import { ArrowLeft, DatabaseZap, Menu } from 'lucide-svelte';
	import { goto } from '$app/navigation';
	import { PAGES } from '$lib/constants/navigation';
	import ThemeSwitch from '$lib/ui/control/ThemeSwitch.svelte';

	let drawerState = $state(false);

	function drawerClose() {
		drawerState = false;
	}

	function onNavItemClick(url: string) {
		drawerClose();
		goto(url);
	}
</script>

<div class="align-items-center bg-primary-200-800 grid h-[var(--h-navbar)] w-full grid-cols-
	<div class="col-span-1 flex items-center justify-start">
		<Modal
			classes="block"
			open={drawerState}
			onOpenChange={(e) => (drawerState = e.open)}
			triggerBase="btn hover:preset-tonal"
			contentBase="bg-surface-100-900 p-4 space-y-4 shadow-xl w-[480px] h-
			positionerJustify="justify-start"
			positionerAlign=""
			positionerPadding=""
			transitionsPositionerIn={{ x: -480, duration: 200 }}
			transitionsPositionerOut={{ x: -480, duration: 200 }}
		>
			{#snippet trigger()}
				<Menu></Menu>
			{/snippet}
			{#snippet content()}
				<header class="flex items-center gap-x-4">
					<button onclick={drawerClose} class="btn hover:prese
						<ArrowLeft />
					</button>
					<h3 class="h3">Pages</h3>
				</header>
				<hr class="hr my-2" />
				<article class="flex flex-col">
					{#each PAGES as page, index (page.name)}
						<button
```

```
                                                        class="btn hover:preset-tonal h5 jus
                                                        onclick={() => onNavItemClick(page.u
                                                >
                                                        <h5 class="h5">
                                                                {page.name}
                                                        </h5>
                                                </button>
                                                {#if index < PAGES.length - 1}{/if}
                                        {/each}
                                </article>
                        {/snippet}
                </Modal>
        </div>

        <div class="col-span-3 flex items-center justify-center">
                <button class="btn hover:preset-tonal" onclick={() => goto('/')}>
                        <span class="flex items-center justify-center gap-x-2">
                                <DatabaseZap />
                                <h3 class="h3">Web Portal</h3></span
                        >
                </button>
        </div>
        <div class="col-span-1 flex items-center justify-end">
                <ThemeSwitch />
        </div>
</div>
```

Key points to note here:

- We've removed the pages prop from the `NavBar` component, as we won't be reusing this component in other places, like we were earlier (in the dashboard). (Not shown here is that we removed the `pages` prop from the `NavBar` component in `src/routes/+layout.svelte` as well.)
- We use the `Modal` component from Skeleton to create a side-bar navigation that opens when the hamburger menu is clicked.
- The `trigger` snippet contains the hamburger menu icon, which opens the modal when clicked.
- The `content` snippet contains the contents of the side bar, in which we list the pages from our `navigation.ts`
- Clicking on a page link in the side bar will close the side bar before navigating to the requested page.
- We divide the `NavBar` into a five column grid, with the navigation section taking up three columns, and the logo and theme switch taking up one column each.
- We use Lucide icons for the website logo (`DatabaseZap`), the hamburger menu (`Menu`), and the back button (`ArrowLeft`).
- We wrap the logo in a button that navigates to the home page when clicked.
- We added a `--h-navbar` CSS variable to our `app.css` file to set the height of the NavBar, as we'll be using this variable for some calculation later:

```css
/* /src/app.css */

@theme {
    --h-navbar: calc(var(--spacing) * 24)
}
```

Responsive NavBar

Now, let's add some screen size prefixes to the `NavBar` to make it responsive as we discussed earlier.

```html
<!-- In the left section -->
<div class="col-span-1 flex items-center justify-start">
    <!-- Show the modal, but hide it on md and larger -->
    <Modal classes = "block md:hidden" ... />
    <!-- When it's hidden, we replace it with a logo section -->
    <div class="hidden md:flex md:items-center md:gap-x-2">
        ...
    </div>
</div>
<!-- Likewise, in the center section, we hide the logo on md and larger -->
<!-- Instead, we show the navigation links -->
<div class="col-span-3 flex items-center justify-center">
    <button class="btn hover:preset-tonal md:hidden" onclick={() => goto('/')}>
        ...
    </button>
    <span class="hidden md:flex md:justify-center">
        ...
    </span>
</div>

<!-- The theme switcher section can be left as is -->
```

Let's take a look at the complete `NavBar` component with responsive design:

responsive-navbar.gif

Again, here's the full code if you got a little lost along the way:

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

<script lang="ts">
        import { Modal } from '@skeletonlabs/skeleton-svelte';
        import { ArrowLeft, DatabaseZap, Menu } from 'lucide-svelte';
        import { goto } from '$app/navigation';
        import { PAGES } from '$lib/constants/navigation';
        import ThemeSwitch from '$lib/ui/control/ThemeSwitch.svelte';

        let drawerState = $state(false);
```

```
	function drawerClose() {
		drawerState = false;
	}

	function onNavItemClick(url: string) {
		drawerClose();
		goto(url);
	}
</script>

<div class="align-items-center bg-primary-200-800 grid h-[var(--h-navbar)] w-full grid-cols-
	<div class="col-span-1 flex items-center justify-start">
		<Modal
			classes="block md:hidden"
			open={drawerState}
			onOpenChange={(e) => (drawerState = e.open)}
			triggerBase="btn hover:preset-tonal"
			contentBase="bg-surface-100-900 p-4 space-y-4 shadow-xl w-[480px] h-
			positionerJustify="justify-start"
			positionerAlign=""
			positionerPadding=""
			transitionsPositionerIn={{ x: -480, duration: 200 }}
			transitionsPositionerOut={{ x: -480, duration: 200 }}
		>
			{#snippet trigger()}
				<Menu></Menu>
			{/snippet}
			{#snippet content()}
				<header class="flex items-center gap-x-4">
					<button onclick={drawerClose} class="btn hover:prese
						<ArrowLeft />
					</button>
					<h3 class="h3">Pages</h3>
				</header>
				<hr class="hr my-2" />
				<article class="flex flex-col">
					{#each PAGES as page, index (page.name)}
						<button
							class="btn hover:preset-tonal h5 jus
							onclick={() => onNavItemClick(page.u
						>
							<h5 class="h5">
								{page.name}
							</h5>
						</button>
						{#if index < PAGES.length - 1}{/if}
					{/each}
				</article>
			{/snippet}
```

```
				</Modal>
				<div class="hidden md:flex md:items-center md:gap-x-2">
					<button class="btn hover:preset-tonal" onclick={() => goto('/')}>
						<DatabaseZap />
						<h3 class="h3">Web Portal</h3>
					</button>
				</div>
			</div>

			<div class="col-span-3 flex items-center justify-center">
				<button class="btn hover:preset-tonal md:hidden" onclick={() => goto('/')}>
					<span class="flex items-center justify-center gap-x-2">
						<DatabaseZap />
						<h3 class="h3">Web Portal</h3></span
					>
				</button>
				<span class="hidden md:flex md:justify-center">
					{#each PAGES as page (page.name)}
						<button class="btn hover:preset-tonal h5 justify-start" oncl
							<h5 class="h5">
								{page.name}
							</h5>
						</button>
					{/each}
				</span>
			</div>
			<div class="col-span-1 flex items-center justify-end">
				<ThemeSwitch />
			</div>
	</div>
```

The dashboard layout

Now that we have a nice looking NavBar, let's improve the layout for our dashboard pages.

Here's an example of what we want to achieve for smaller screens:

bar-nav.png

Shown above is a content section that takes up most of the screen, with Bar navigation that shows navigation to the three sub-pages (we count the `/dashboard` page as the 'overview' page) at the bottom of the screen.

For larger screens, we want to switch to `Rail` navigation, which puts the navigation element on the left-hand side of the screen:

rail-nav.png

First, let's expand the `navigation.ts` file (and `PageInfo` interface) to include Lucide icons we can display for each page:

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

```
import type { Icon } from 'lucide-svelte';

export interface PageInfo {
        name: string;
        description: string;
        url: string;
        icon?: typeof Icon;
}


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

import { PanelsTopLeft, User, Dumbbell } from 'lucide-svelte';

...

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
        }
];
```

Now, let's alter `src/routes/dashboard/+layout.svelte` to use the `Rail` and `Bar` navigation components, hiding the `Rail` navigation on smaller screens, and showing the `Bar` navigation on larger screens.

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

<script lang="ts">
        import { Navigation } from '@skeletonlabs/skeleton-svelte';
        import { page } from '$app/state';
        import { DASHBOARD_PAGES } from '$lib/constants/navigation';

        let value = $state('overview');
        let { children } = $props();
        let currentPageUrl = $derived(page.url.pathname);
</script>

<div
        class="flex h-full max-h-[calc(100dvh-var(--h-navbar))] w-full flex-col overflow-y-sc
>
        <!-- This is the rail for larger screens -->
        <aside class="hidden lg:sticky lg:block">
                <Navigation.Rail {value} onValueChange={(newValue) => (value = newValue)}>
                        {#snippet tiles()}
                                {#each DASHBOARD_PAGES as dashboardPage (dashboardPage.name)
                                        <Navigation.Tile
                                                href={dashboardPage.url}
                                                label={dashboardPage.name}
                                                selected={currentPageUrl === dashboardPage.ur
                                        >
                                                {#if dashboardPage.icon}
                                                        <dashboardPage.icon />
                                                {/if}
                                        </Navigation.Tile>
                                {/each}
                        {/snippet}
                </Navigation.Rail>
        </aside>
        <div class="flex flex-grow flex-col overflow-y-scroll p-4">
                {@render children()}
        </div>
        <!-- This is the bar for smaller screens -->
        <aside class="block lg:hidden">
                <Navigation.Bar classes="max-h-24" {value} onValueChange={(newValue) => (valu
                        {#each DASHBOARD_PAGES as dashboardPage (dashboardPage.name)}
                                <Navigation.Tile
                                        href={dashboardPage.url}
                                        label={dashboardPage.name}
                                        selected={currentPageUrl === dashboardPage.url}
                                >
                                        {#if dashboardPage.icon}
                                                <dashboardPage.icon />
                                        {/if}
                                </Navigation.Tile>
                        {/each}
                </Navigation.Bar>
        </aside>
</div>
```

Key points to note here:

- We set the maximum height of the main content area to be the full height of the screen minus the height of

the NavBar using `max-h-[calc(100dvh-var(--h-navbar))]`.
    - Using `dvh` (dynamic viewport height) as opposed to `vh` (viewport height) ensures that the height is calculated correctly even when the browser's address bar is hidden on mobile devices.
    - We add `overflow-y-scroll` so if content overflows the available height, it will scroll, leaving plenty of space for the navigation elements.
    - This ensures that the navigation elements are always centered vertically, even if the content is longer than the screen height.
- We use the `Navigation` component from Skeleton to create the `Rail` and `Bar` navigation.
- We show and hide the large screen `Rail` navigation using `hidden lg:block`
- We show and hide the small screen `Bar` navigation using `block lg:hidden`
- We use the `value` state variable to keep track of the currently selected page, and update it when the user clicks on a navigation item.
- We use the `currentPageUrl` (which is derived from the 'page' state) to determine the page that is currently being viewed, and highlight the corresponding navigation item by setting the `selected` prop on the `Navigation.Tile` component to true.
- We directly use the `dashboardPage.icon` as a component in Svelte. This is a neat trick - and is possible because we stored the Icon Component directly into the `icon` field in the navigation constants (`src/lib/constants/navigation.ts`).

A custom Card component

Cards are a great way to display information in a visually appealing way.

Let's create a custom `Card` component, based on the Skeleton `Card` component, that we can use throughout our application.

We want to create a `Card` component that:

- Uses the Skeleton `Card` template as a base
- Allows us to pass in a header, article, and footer section.
- Allows us to pass in some `$props` to override or extend the card styling where needed.

Let's look at the code for our custom `Card` component and then walk through it:

```
<!-- /src/lib/ui/views/Card.svelte -->

<script lang="ts">
        import type { Snippet } from 'svelte';

        let {
                base = 'card preset-filled-surface-100-900 border-surface-200-800 card-hover
                baseExtension = '',
                headerBase = 'h3 p-8',
                headerExtension = '',
                articleBase = 'flex flex-col gap-4 p-8',
                articleExtension = '',
                footerBase = 'flex items-center gap-4 p-8',
                footerExtension = '',
                header,
                article,
                footer
```

```
	}: {
		base?: string;
		baseExtension?: string;
		headerBase?: string;
		headerExtension?: string;
		articleBase?: string;
		articleExtension?: string;
		footerBase?: string;
		footerExtension?: string;
		header?: Snippet;
		article?: Snippet;
		footer?: Snippet;
	} = $props();
</script>

<div class="{base} {baseExtension}">
	{#if header}
		<header class="{headerBase} {headerExtension}">
			{@render header()}
		</header>
	{/if}
	{#if article}
		<article class="{articleBase} {articleExtension}">
			{@render article()}
		</article>
	{/if}
	{#if footer}
		<footer class="{footerBase} {footerExtension}">
			{@render footer()}
		</footer>
	{/if}
</div>
```

Key points to note here: - We use the `Snippet` type from Svelte to allows us to pass in custom markup for the header, article, and footer sections of the card. - We create variables for the base styling of the card, as well as for the header, article, and footer sections.- We create variables for the extensions to the base styling. - We insert the custom markup into the `class` attributes where needed. - We use the `@render` directives to render the Snippets passed in for the header, article, and footer sections.

Now, let's use this `Card` component on our home page (at /) to replace our demo counter button and display a welcome message instead.

We'll create a new file `src/lib/constants/strings.ts` to hold the name of the game we are building the web portal for (we'll call it `DemoBots`), so we can later re-use it in other pages or components.

```
// /src/lib/constants/strings.ts

export const GAME_NAME = 'DemoBots';
```

We also add an image to the `static` folder at the root of our project named `header.jpg` that we can use as a header image for our card.

Then, we alter the `+page.svelte` file to use our new `Card` component and display a welcome message.

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

<script lang="ts">
	import { GAME_NAME } from '$lib/constants/strings';
	import Card from '$lib/ui/views/Card.svelte';
</script>

<div
	class="container mx-auto grid h-full w-full grid-cols-1 items-center justify-items-ce
>
	<Card footerBase="flex gap-x-4 p-8">
		{#snippet header()}
			<img src="/header.jpg" class="aspect-[21/9] w-full hue-rotate-90" alt
		{/snippet}
		{#snippet article()}
			<div>
				<h3 class="h3">Welcome!</h3>
			</div>
			<p class="opacity-60">
				This website functions as a web portal to a database for the
			</p>
		{/snippet}
	</Card>
</div>
```

That looks much better!

welcome-card.png

We now have all major UI components in place to build out the rest of our application.

We have a responsive layout with a navigation bar, a dashboard section with it's own navigation UI, a theme switcher, and a custom card component to display information. We'll be looking at a couple more new UI components in the future, such as text fields and tables, but most of these will live inside other `Card` components, so we'll handle them as we go along.

Wrapping up

In this chapter, you:

- Installed and used lucide-svelte to add icons to your components.
- Set up and configured the Skeleton design system for your Svelte app.
- Created a responsive, mobile-first layout using Skeleton components.
- Built a theme switcher that stores preferences using localStorage and Svelte state.
- Implemented a dynamic NavBar that adapts between modal and inline layouts.
- Used Skeleton’s Navigation.Rail and Navigation.Bar for contextual navigation.

- Built a reusable Card component with flexible layout slots using Svelte snippets.

You now have a strong foundation for building user-friendly, adaptable interfaces. In Chapter 3, we'll look at implementing Drizzle ORM to interact with our database, allowing us to fetch and display data in our application.