Starter Guide for Svelte & SvelteKit

Table of Contents

1 Introduction

1.1 Preface

The intention of this guide is to introduce you to the Svelte ecosystem. This guide is neither a comprehensive guide to Svelte or SvelteKit, nor a tutorial for basic JavaScript, TypeScript or programming in general, but rather a starting point to get you up and running with the basics and thus most of the more advanced topics of Svelte & SvelteKit will be omitted.

If you are looking for a more comprehensive guide, or for a tutorial to build up your skills or more detailed explanations for the following features (regarding Svelte or SvelteKit, not JavaScript or TypeScript), I recommend the official documentation for Svelte and SvelteKit.

⚠ This guide currently only talks about the features of Svelte but not SvelteKit. It will be updated in the future to include SvelteKit as well, however it is more than enough to get you started with Svelte. The most important information about SvelteKit is already included in the guide, whatsoever.

1.2 Prerequisites

This guides uses a starter project, accessible via a GitHub repository, that you can clone and then use throughout the guide try out and apply the features that you will learn about. This guide will also contain examples and has been enriched with additional information to help you understand the concepts better.

If you are not familiar with Git, you can create a starter project using the node package manager (npm) or any other package manager of your choice. However, we recommend using Git, so that you have full and free access to the starter project and the educational material that it contains.

1.2.1 Resources & Dependencies

In order to get started with Svelte and SvelteKit, you will need to have the following downloaded and installed on your machine:

- Node.js
- IDE that supports Node.js (see recommendations below)

The comprehensive list for the resources that you should have available on your machine are:

- Node.js (necessary)
- Svelte Documentation Chrome Extension (recommended)
- Jetbrains Toolbox (recommended)
- Jetbrains Webstorm (recommended, comprehensive, licensed), IntelliJ Idea Ultimate (comprehensive, licensed) / IntelliJ Idea Community (comprehensive, free) or Visual Studio Code (recommended, lightweight, free)

1.2.2 Access & Installation

To access the resources mentioned above, you can follow these steps:

1. Install Node.js:

   Visit https://nodejs.org/en/download/, download the latest (stable) version of Node.js and install it on your machine.

2. Install Javascript, Svelte & SvelteKit and verify Node.js:

   Open a terminal of your choice and run the following commands:

   ```
   npm install
   npm --version
   node --version
   ```

3. Install the Svelte Documentation Chrome Extension:

   Using Chrome, visit https://svelte.dev/ and then use the ↗ button at the right end of the search bar to install and open the extension.

4. Install Jetbrains Toolbox (recommended, optional):

   Visit https://www.jetbrains.com/toolbox-app/ download the latest (stable) version and install it on your machine.

5. Install an IDE:
    - Jetbrains Webstorm:

      Either visit https://www.jetbrains.com/webstorm/download/ or use the Jetbrains Toolbox download the latest (stable) version and install it on your machine.

    - Jetbrains IntelliJ Idea Ultimate or Community:

      Either visit https://www.jetbrains.com/idea/download/ for both editions or use the Jetbrains Toolbox download the latest (stable) version and install it on your machine.

    - Visual Studio Code:

      Visit https://code.visualstudio.com/ download the latest (stable) version and install it on your machine.

1.2.3 Setup & Configuration

Since this guide is theory only, you don't necessarily need to set up a Svelte Project however, if you want to try some things out and create a project you can follow the steps below:

1. Open a terminal of your choice and run the following commands:

   ```
   cd path/to/your/project
   npx sv create
   ```

2. Question "Where should we create your project?":

   Provide the path to your project, or press enter to create the project in the current directory, which you should have already provided in the first command.

3. Question "Which Svelte app template?":

   Choose "SvelteKit Minimal" and press enter.

4. Question "Add type checking with Typescript":

   Choose "Yes, using Typescript Syntax" and press enter.

5. Question "Add type checking with Typescript":

   Choose "Yes, using Typescript Syntax" and press enter.

6. Question "What would you like to add to your project?":

   I recommend the following options:

    - Code Quality: Prettier
    - Testing: Vitetest
    - CSS: Tailwind CSS (If you care about styling and know how to work with Vanilla CSS)

7. Question "Which package manager do you want to install dependencies with?":

   Choose "npm" and press enter.

You can now open the project in the IDE of your choice.

1.2.4 Verification & Execution

To verify the installation and setup, you can run the following command in the terminal (inside or outside your IDE):

```
cd path/to/your/project/SSG-SvelteStarterGuide
npm run dev
```

or

```
cd path/to/your/project/SSG-SvelteStarterGuide
npm start
```

You can use the commands above to start and test your application, whereas `npm run dev` will start a development server, and thus provide important features such as hot reload that will help you during development and `npm start` will start a production server, thus not providing important features for development, such as hot reload.

We generally recommend using `npm run dev` while using this guide to develop your application.

1.3 Framework

To understand what Svelte & SvelteKit are about, you will find a short summary below, however it is recommended to read both of the very short introductory sections in the official documentation for Svelte and SvelteKit, respectively. Both of these sections are only one page long and will give you a good overview of Svelte and SvelteKit.

1.3.1 Svelte

Svelte is a monolithic, reactive and component-based TypeScript framework that compiles your code to highly efficient vanilla JavaScript. It is a great choice for building web applications that are fast, lightweight and easy to maintain.

Svelte can be used as a standalone framework, or in combination with SvelteKit. (For more information on SvelteKit, see 1.3.2 SvelteKit)

For differences between Svelte and other frameworks, see 1.4.3 Similar frameworks.

1.3.2 SvelteKit

SvelteKit is a set of libraries and tools (or subframework) that complement the Svelte framework. It provides a more feature-rich development experience, including server-side rendering (SSR), static site generation (SSG), routing, and more.

SvelteKit can only be used in combination with Svelte.

1.4 Features

In order to give you a better understanding of the features of Svelte and SvelteKit, you will find short but concise descriptions of all the features that will be covered later on in this guide below, as well as a comparison of Svelte and SvelteKit with other frameworks.:

1.4.1 Svelte

- Project Structure:

  How the project is structured and organized, and which files are responsible for what. (2.1 Project Structure)

- Styling:

  How styling can be applied to components and pages, and how it is scoped through the application. (2.2 Styling)

- Components:

  What components are and how they can be used in contrast to traditional pages. (2.3 Components)

- Data:

  How the DOM and other parts of the application react to changes in the data. (2.4 Data)

- Bindings:

  How input elements react to changes in the data. (2.5 Bindings)

- Actions:

  What lifecycle hooks are and how they can be used to execute functions on elements of the DOM. (2.6 Actions)

- Logic:

  How logic can be embedded into the HTML code and reacts to changes in the data. (2.7 Logic)

- Events:

  How events can be used to handle user interactions and react to changes in the data. (2.8

Events)

1.4.2 SvelteKit

- Routing:

  How components can be navigated and what alternatives there are to components. (3.1 Routing)

- Data:

  How the DOM reacts to changes of data in an advanced way. (3.2 Data)

- Headers & Cookies:

  How data can be stored and accessed. (3.3 Headers & Cookies)

- Forms:

  How new and existing data can be submitted and processed. (3.4 Forms)

- Handlers:

  How REST-Based CRUD operations can be implemented(3.5 Handlers).

- Stores:

  How data can be stored and accessed globally, which doesn't belong to any component. (3.6 Stores)

1.4.3 Similar frameworks

Here you can see some basic functionalities for common reactive frameworks, and how they are implemented, in comparison to Svelte:

Functionality	**Angular**	**React**	**Vue**	**Svelte**
Component-based	Uses classes and decorators	Uses functions or classes	Uses an options API or Composition API	Uses compiled components
Reactive updates	Uses Zone.js for change detection	Uses a virtual DOM diffing algorithm	Uses a virtual DOM diffing algorithm and reactivity system	Compiles reactivity into vanilla JS
State management	Uses services and RxJS	Uses local component state and context API, often used with external libraries like Redux or MobX	Uses local component state and Vuex for global state	Uses local component state and stores for shared state

Functionality	Angular	React	Vue	Svelte
Routing	Uses Angular Router	Often used with React Router or Reach Router	Uses Vue Router	Uses page.js in SvelteKit
Server-side rendering (SSR)	Uses Angular Universal	Uses Next.js or other libraries	Uses Nuxt.js or other libraries	Built-in with SvelteKit
Static site generation (SSG)	Uses Scully	Uses Next.js or other libraries	Uses Nuxt.js or other libraries	Built-in with SvelteKit
TypeScript support	Built-in	Requires additional setup	Requires additional setup	Requires additional setup

2 Svelte

2.1 Project Structure

The project structure of a Svelte application is quite simple and easy to understand. It consists of a few folders and files that are used to organize your code and resources. Below you can see an overview of the folders and files that you will work with in a Svelte application:

- application (root):

  This folder contains the configuration files, which you should only modify, when you know what you are doing, and all other files off the project.

- .svelte-kit & node_modules:

  These folders contain the and dependencies for your application. You should not modify these files.

- src:

  This folder contains all the source code for your application. It also contains the `app.html` file, which is the entry point of your application.

- lib:

  This folder contains all the reusable parts for your application, mainly components.

- routes:

  This folder contains all the pages of your application. Every subfolder in this directory represents a route to page of your application.

- static:

  This folder contains all the static files (resources) for your application, such as images, fonts, and other assets.

2.1.1 Files

Files in Svelte are always a combination of HTML, CSS and Typescript, and thus are quite compact since they combine three files into. However, Svelte doesn't limit you in that regard, and you can still use external files for your styling and scripting.

If you create a new svelte file, it should generally look like this:

```
<style>
    <!-- Your CSS Code -->
</style>

<!-- Your HTML Code -->

<script>
    <!-- Your TypeScript Code -->
</script>
```

2.1.2 Base

Have a look at the `/src` folder inside the application where you will find the `app.html` file. This file is the entry point of your application and contains the basic HTML needed to build all the important components needed for your Web Application such as the DOM. In there you will find the following code:

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

The `%sveltekit.body%` attribute inside the body tag is responsible for rendering the components that you will create in the `/src/routes` folder. The `app.html` file is the main layout file for your application, and it is used to define the layout of your application.

Unless you have a specific reason to change the `app.html` file, for example, to apply Global Styling, which is not a recommended practice (If you want to know more about styling you should have a look at 2.2 Styling) or to add necessary Scripts, you should leave it as it is.

2.2 Styling

Styling in Svelte is done using CSS, and thus not majorly different from styling in vanilla JavaScript. However, Svelte provides some key features that make styling easier and more efficient.

2.2.1 Scope

Styles in Svelte can be scoped differently and are by default not applied to all elements that correspond to the selectors used in a style tag. In Svelte styles can thus be scoped globally or locally.

2.2.1.1 Global Styling

Global Styling in Svelte can be achieved in two ways:

- Adding a style tag to the `app.html` file (not recommended):

  Example 1A: Placing global styles in the style tag

  ```html
  <!doctype html>
  <html lang="en">
      <head>
          <meta charset="utf-8" />
          <meta name="viewport" content="width=device-width, initial-
  scale=1" />
          <link rel="icon" href="%sveltekit.assets%/favicon.png" />
          <title>Svelte  Application</title>
          %sveltekit.head%
      </head>
      <body data-sveltekit-preload-data="hover">
          <div style="display: contents">%sveltekit.body%</div>
      </body>
      <style>
          <!-- Your Global Styling -->
      </style>
  </html>
  ```

  The styles in the style tag will now affect every element of your application.

  This option is generally not recommended, since it is not best practice to add global styling to the `app.html` file.

- Adding a global CSS file to the `/src` folder (recommended):

  To add a global CSS file to your Svelte application, you can create a new CSS file in the `/src` folder and then import it into the `app.html` file. From there you have to possibilities to apply the styles to your files, by either importing the file to every svelte file or the much better option which is to apply the global property to all of your selectors:

  Example 1B: Placing global styles in a seperated file using the global directive

  ```
  <style>
      :global(body) {
          background-color: black;
      }

      :global(h1) {
          color: white;
      }
  ```

```
    /* Your other global styles */
</style>
```

The styles in the style tag will now affect every element of your application.

This option allows you to effectively apply global styling to your entire application from one central file, without having to import it every time.

2.2.1.2 Local Styling

When styling a Svelte Component, Page or Layout, you can either use an external stylesheet that you then import, or you can use the `<style>` tag inside the file:

Example 2: Applying local styles in the style tag

```
<style>
    h1 {
        color: white;
    }

    /* Your other  styles */
</style>

<h1>Hello World</h1>

<script>
    // Your script
</script>
```

Styles in Svelte are scoped by default, which means that styles defined in a Component, Page or Layout or imported to said file only apply to said file and its children.

When a Svelte application is initialized all elements that have styles applied to them, get added to classes which name consists of a unique hash to make sure that the styles are only applied to the elements they are supposed to be applied to.

2.2.2 Transitions

Transitions are animations that can be applied to elements in order to gracefully transform them from a visible to an invisible state and vice versa. You can either use built-in transitions or create your own custom transitions using CSS and JS.

2.2.2.1 Mono-Transitions

In order to apply a transition to an element, you can use the `transition` directive, which takes the transforming function and corresponding parameters as an argument.

Example 1: Applying a transition to an element.

```
<script>
    import { fade } from 'svelte/transition';

    let visible = $state(true);
</script>

<label>
    <input type="checkbox" bind:checked={visible} />
    visible
</label>

{#if visible}
    <p transition:fade>
        Fades in and out
    </p>
{/if}
```

Instead of disappearing and reappearing instantly, the element will now fade in and out.

2.2.2.2 Bi-Transitions

Transitions can also have different animations for appearing and disappearing. This can be done by using the `in:` and `out:` directives.

Example 2: Applying different transitions to an element for appearing and disappearing.

```
<script>
    import { fly } from 'svelte/transition';

    let visible = $state(true);
</script>

<label>
    <input type="checkbox" bind:checked={visible} />
    visible
</label>

{#if visible}
    <p transition:fly={{ y: 200, duration: 2000 }}>
        Flies in and out
    </p>
{/if}
```

Instead of disappearing and reappearing instantly, the element will now fly in and out.

2.2.2.3 Global Transitions

Transitions can also be synchronised with all other elements that have the same transition applied to them. This can be done by using the `transition` or `in` and `out` directives with the `global` pipe.

Example 3: Applying a global transition to an element.

```
<script>
    import { slide } from 'svelte/transition';

    let items = ['one', 'two', 'three', 'four', 'five', 'six', 'seven',
'eight','nine', 'ten'];

    let showItems = $state(true);
    let i = $state(5);
</script>

<label>
    <input type="checkbox" bind:checked={showItems} />
    show list
</label>

<label>
    <input type="range" bind:value={i} max="10" />
</label>

{#if showItems}
    {#each items.slice(0, i) as item}
        <div transition:slide>
                {item}
        </div>
    {/each}
{/if}

<style>
    div {
        padding: 0.5em 0;
        border-top: 1px solid #eee;
    }
</style>
```

Now all elements that have the transition `transition:slide` applied to them will slide in and out simultaneously.

2.2.2.4 Transition Events

With transitions, you can also use events to trigger functions when a transition starts or ends. This can be done by using the `on:` directive with the `introstart`, `introend`, `outrostart` or `outroend` event.

Example 4: Applying a transition to an element and triggering a function when the transition starts or ends.

```
<script>
    import { fly } from 'svelte/transition';

    let visible = $state(true);
    let status = $state('waiting...');
</script>

<p>status: {status}</p>

<label>
    <input type="checkbox" bind:checked={visible} />
    visible
</label>

{#if visible}
    <p
        transition:fly={{ y: 200, duration: 2000 }}
        onintrostart={() => status = 'intro started'}
        onoutrostart={() => status = 'outro started'}
        onintroend={() => status = 'intro ended'}
        onoutroend={() => status = 'outro ended'}
    >
        Flies in and out
    </p>
{/if}
```

When the transitions start or end, the status and the text will change accordingly.

2.3 Components

Components are files which are reusable pieces of code that can be used to create complex user interfaces. Components can be nested inside other components, can take arguments, and they can communicate with each other using props and events and more.

To create a new component click on the folder `lib`, then hover over new and select `Svelte Component`. Enter the name of your component, and you are done, make sure, however, that your file ends with `.svelte`.

Unlike it is the case in other frameworks such as Angular, you are not required to actually make use of components in your Application, in case you use SvelteKit.

2.3.1 Props

When passing parameters to an embedded component, you can use the `$props` rune to access these parameters from within the component.

Example 6: Accessing passed parameters by assigning the $props rune to a variable

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

If a parameter named `answer` is passed to a component, then its value will automatically be assigned to the variable `answer` inside the component.

One can also set default values for variables to make it safe for falsy values.

Example 7: Accessing passed parameters by assigning the $props rune to a variable

```
<script>
    let { answer = 'a mystery' } = $props();
</script>
```

In case you have an entire property that you want to pass to a component, you can use the spread operator `…name` to pass all objects of a property to the component, without having to pass them individually.

Example 8: Passing all objects of a property to a component

```
<script>
    import PackageInfo from './PackageInfo.svelte';

    const pkg = {
        name: 'svelte',
        version: 5,
        description: 'blazing fast',
        website: 'https://svelte.dev'
    };
</script>

<PackageInfo {...pkg} />
```

2.4 Data

Data in Svelte can be made reactive, which is a key functionality that allows you to create dynamic and interactive user interfaces. The core idea is that every time data in one of the scripts changes or the DOM changes, the respective opposite is automatically updated to reflect those changes.

Furthermore, variables can easily be embedded into the HTML.

2.4.1 Variable Injection

In Svelte, one can embed/inject variables into existing markup code in order to use the data from the respective variables in the DOM, without having to build some kind of bridge between the script and the markup code.

Example 1: Injecting a variable as text

```
<script>
    let name = 'Svelte';
</script>

<h1>Hello {name}!</h1>
```

The DOM will render the text "Hello Svelte!", and automatically rerender in case the value of the variable `name` changes.

Furthermore, HTML code can also be directly embedded into existing markup code by the `@html` in an injection in order to format the text correctly.

Example 2: Injecting formatted HTML code as text

```
<script>
    let string = `this string contains some <strong>HTML!</strong>`;
</script>

<p>{@html string}</p>
```

The DOM will render the text as HTML, and automatically rerender in case the value of the variable `string` changes.

Variables can not only be used as text, but also as values for HTML and Svelte attributes.

Example 3: Injecting a variable into HTML attributes

```
<button
    class="card {flipped ? 'flipped' : ''}"
    onclick={() => flipped = !flipped}
>
```

The functionality of the button will change depending on the value of the variable `flipped`.

Even though it might seem like this concept of reactivity could be applied to any kind of statement, there are actually limitations to the way reactivity is applied here and thus different rules for different kinds of statements apply, such as input fields, styling, conditions, loops, and more, which will be covered in the following sections, which mainly are 2.5 Bindings, 2.6 Actions and 2.7 Logic.

2.4.2. State

To make sure that an application is reactive towards changes of variables, one needs to use State. State in Svelte is an expression that can be applied to any variable and that will make any HTML Node be updated automatically as a part of the DOM if the data that the variable holds changes.

To apply state to a variable, encapsulate it with the `$state()` expression. This makes the DOM react to any change of the encapsulated variable.

Example 1: Creating a state from a variable to make the DOM reactive towards reassignments:

```
<script>
    let count = $state(0);

    function increment() {
        count += 1;
    }
</script>

<button onclick={increment}>
    Clicked {count}
    {count === 1 ? 'time' : 'times'}
</button>
```

The `$state()` expression is shorthand for the `writable()` Svelte function, which is used to create a writable store.

Expressions in Svelte that have the format `$expression()` are called runes. They are used to apply special features to variables, such as reactivity, and are generally used to make the code more readable and easier to understand.

When using state to make the DOM react to mutability changes instead of reassignments, as previously shown, one calls this deep state.

Example 2: Creating a deep state from a variable to make the DOM reactive towards mutations:

```
<script>
    let numbers = $state([1, 2, 3, 4]);

    function addNumber() {
        numbers.push(numbers.length + 1);
    }
</script>

<p>{numbers.join(' + ')} = ...</p>
<button onclick={addNumber}>Add a number</button>
```

2.4.1.1 Derived State

When extracting data from on state into another the `$derived()` rune is used. This creates another state which updates whenever the original state updates. Derived states are unlike normal states immutable.

Example 3: Creating a derived state from a state to make the DOM reactive towards state changes:

```
<script>
    let numbers = $state([1, 2, 3, 4]);
    let total = $derived(numbers.reduce((t, n) =>   t + n, 0));

    function addNumber() {
    numbers.push(numbers.length + 1);
    }
</script>

<p>{numbers.join(' + ')} = {total}</p>
<button onclick={addNumber}>Add a number</button>
```

2.4.1.2 Inspected State

States are an ever-changing mechanic. Svelte thus makes it possible to create a snapshot of a state at any given point in time. There are two ways to do this:

- Using `$state.snapshot()`:

Example 4: Creating and logging a snapshot of a state:

```
<script>
    let numbers = $state([1, 2, 3, 4]);
    let total = $derived(numbers.reduce((t, n) => t + n, 0));

    function addNumber() {
        numbers.push(numbers.length + 1);
        console.log($state.snapshot(numbers));
    }
</script>

<p>{numbers.join(' + ')} = {total}</p>
<button onclick={addNumber}>Add a number</button>
```

- Using `$inspect()`:

Example 5: Inspecting and logging a snapshot of a state:

```
<script>
    let numbers = $state([1, 2, 3, 4]);
    let total = $derived(numbers.reduce((t, n) => t + n, 0));

    function addNumber() {
        numbers.push(numbers.length + 1);
    }

    $inspect(numbers).with(console.trace);
</script>

<p>{numbers.join(' + ')} = {total}</p>
<button onclick={addNumber}>Add a number</button>
```

Everytime the state of the assigned variable changes, the `$inspect()` function handover the value to the supplied function, in this case `console.trace`. This means that you don't have to manually log the state every time it changes.

2.4.3 Key-Blocks

Key-Blocks in Svelte are used to forcefully recreate elements of the DOM, when the respective data changes, in contrast to the default behaviour of Svelte, which is to only rerender/update the according elements.

Example 6: Using a Key-Block to recreate elements of the DOM when the data changes:

```
{#key i}
    <p in:typewriter={{ speed: 10 }}>
        {messages[i] || ''}
    </p>
{/key}
```

The text will be recreated every time, if any of the values change.

2.5 Bindings

When using any type of input, select or textarea elements, you can use Binding, which is a common concept in reactive frameworks and works similarly in Svelte. Binding allows you to synchronize (bind together) the value of the aforementioned input, select & textarea elements, and the value of one or many variables. This means that every time the value of one side of the binding changes, the value of all the corresponding sides will automatically be updated.

Two-way binding is done by adding the `bind:` directive to before the value of the element and after that an assignment for the variable that is to be bound to said element.

Example 12: Binding the value of an input element to a variable:

```
<input bind:value={name}>
```

Example 13: Binding the value of number & range input elements to a variable:

```
<label>
    <input type="number" bind:value={a} min="0" max="10" />
    <input type="range" bind:value={a} min="0" max="10" />
</label>

<label>
    <input type="number" bind:value={b} min="0" max="10" />
    <input type="range" bind:value={b} min="0" max="10" />
</label>
```

Example 14: Binding the value of a select element to a variable:

```
<select bind:value={selected}>
```

When one wants to bind multiple radio or checkbox input elements to a variable, one has to bind to the `group` attribute and not the value to bind all of the elements to the same variable.

Example 15: Binding the value of a multiple radio input elements to a variable:

```
<input
    type="radio"
    name="scoops"
    value={number}
    bind:group={scoops}
/>
```

Example 16: Binding the value of multiple checkbox input elements to a variable:

```
<input
    type="checkbox"
    name="flavours"
    value={flavour}
    bind:group={flavours}
/>
```

Example 17: Binding the value of a textarea element to a variable:

```
<textarea bind:value={value}></textarea>
```

2.6 Actions

Actions in Svelte are used to execute a function on an Element of the DOM when it is rendered, such as focusing an input element, scrolling to a specific position, or animating an element. Actions are added to HTML elements using the `use:action` directive, where `action` is the name of the function you want to execute.

Example 18: Using the `use:action` directive to focus an input element:

```
<input use:focus />
```

The focus function will automatically be called on the Input Element when the page is rendered.

When one uses functions that require parameters, one can pass them to the function by adding a bracket with the required parameters to the `use:action` directive.

Example 19: Using the `use:action` with additional parameters to scroll to a specific position:

```
<div use:scrollTo={1000} />
```

The scrollTo function will automatically be called on the div Element with the parameter 1000 when the page is rendered, and the div Element will be scrolled to the position 1000.

2.7 Logic

One of the main features of Svelte is that you can directly write Logic inside your HTML code. This allows you to encapsulate certain parts of your UI with Logic to tailor the UX, without having to use the script tag, and to save a significant amount of code, which makes your files more simplistic, efficient and readable overall.

2.7.1 If Blocks

If Blocks in Svelte are used to conditionally render parts of your UI.

An If Statements starts with the opening tag `{#if condition}` and ends with the closing tag `{/if}`. The condition can be any expression that evaluates to a boolean value. One can add an Else Blocks by using the `{:else}` tag and/or an Else-If Blocks to an arbitrary depth by using the `{:else if condition}` tag in between the opening and closing tag of the If Block.

Example 6: Creating an If Block to conditionally render a part of the UI:

```
{#if count > 10}
    <p>{count} is greater than 10</p>
{:else if count < 5}
    <p>{count} is less than 5</p>
{:else}
    <p>{count} is between 5 and 10</p>
```

```
{/if}
```

2.7.2 Each Blocks

Each Blocks in Svelte are used to iterate over an array and render a part of your UI for each element in the array.

An Each Block starts with the opening tag `{#each array as element}` and ends with the closing tag `{/each}`. The array can be any expression that evaluates to an array. One can also add an Else Blocks by using the `{:else}` tag to render a part of the UI if the array is empty. Additionally, one can also add an index the end of the condition of the Each block `{#each array as element, index}` to access the index of the current element.

Example 7: Creating an Each Block to iterate over an array and render a part of the UI for each element:

```
<div>
    {#each colors as color, i}
        <button
            style="background: {color}"
            aria-label={color}
            aria-current={selected === color}
            onclick={() => selected = color}>
            {i + 1}
        </button>
    {/each}
</div>
```

2.7.3 Await Blocks

Await Blocks in Svelte are used to wait for a promise to resolve and render a part of your UI for each stage of the promise.

An Await Block starts with the opening tag `{#await promise}` and ends with the closing tag `{/await}`. The promise can be any expression that evaluates to a promise. One can also add an Then Block by using the `{:then value}` tag to render a part of the UI if the promise resolves successfully, and an Catch Blocks by using the `{:catch error}` tag to render a part of the UI if the promise rejects.

Example 8: Creating an Await Block to wait for a promise to resolve and render a part of the UI for each stage of the promise:

```
{#await promise}
    <p>Waiting...</p>
{:then value}
    <p>The value is {value}</p>
{:catch error}
    <p style="color: red">{error.message}</p>
```

```
{/await}
```

2.8 Events

Events in Svelte are used to handle user interactions, such as clicks, hovers, and key presses. Events are added to HTML elements using the `onevent` directive, where `event` is the name of the event you want to handle.

Example 9: Handling a pointer move event on a div element:

```
<div onpointermove={onpointermove}>
    The pointer is at {Math.round(m.x)} x {Math.round(m.y)}
</div>
```

Every time the pointer moves over the div element, the `onpointermove` function will be called.

Alternatively, one can also directly embed the event handler into the HTML element by using an inline Lambda function.

Example 10: Handling a pointer move event on a div element using an inline Lambda function:

```
<div onpointermove={(event) => {
        m.x = event.clientX;
        m.y = event.clientY;
    }}>
    The pointer is at {m.x} x {m.y}
</div>
```

Every time the pointer moves over the div element, the inline handler will be executed.

2.8.1 Event Bubbling & Capturing

Just like it is the case in Vanilla JavaScript, Events in Svelte bubble up in the DOM tree starting with the most nested element to the least nested element.

This means that if you have a nested element inside another element with EventListeners on both of them, and you click on the nested element, the event will first be handled by the nested element and then by the parent element.

To reverse the order of the event handling, one can add the word capture to the event directive `on:keydowncapture`, which will make the event bubble down the DOM tree starting with the least nested element to the most nested element.

Example 11: Bubbling down with EventListeners on a nested div:

```
<div on:keydowncapture={(e) => alert(`<div> ${e.key}`)}>
    <input on:keydowncapture={(e) => alert(`<input> ${e.key}`)} />
</div>
```

3 SvelteKit

3.1 Routing

Routing in SvelteKit is one of the most important features of the framework. Unlike it is the case in most other reactive frameworks, routing has the following advantages:

- Requires no setup.
- Automatically updates & adapts to your changes.
- Is implemented by the directories you create in the `/src/routes` folder.

Routing in Svelte is based on the file system that you use to manage your Pages & Layouts. Take a look at the `/src/routes`. In Svelte it contains all the Pages & Layouts that you will create for your application. Each page is a separate file that contains the HTML, CSS, and JavaScript code for that page.

Pages & Layouts will be discussed in more detail in the next 2 points <<S.

This also means that you simply start by creating a new page in the `/src/routes` folder. The name of the directory where the page is will then be the name of the route.

Example 3: Creating a route

If you create the directory `/src/routes/about` with a page inside, then the route to that page will be `http://localhost:5173/about`.

You can also create nested routes by creating a subdirectory of an already existing directory inside the `/src/routes` folder. You can generally create up to infinitely many and infinitely deep routes.

Example 4: Creating a nested route

If you create a directory `/src/routes/about/imprint` with a page inside, and then a then the route to that page will be `http://localhost:5173/about/imprint`.

Finally, you can also create routes that contain parameters. This is done by creating a directory where the name of the directory is the name of the parameter, encapsuled by two square brackets `[` and `]`.

Example 5: Creating a parameterized route

If you create the directories `/src/routes/[id]x[name]/account` with a page inside account, and then the route to that page will be for example `http://localhost:5173/12345xKlaus Unger/imprint`, whereas the route can have infinity many parameters, which are each to be seperated with one static character (e.g. x) and whereas the value of any parameter can be any string.

3.1.1 Pages

Pages in Svelte are the main building blocks of your application. They are the files that contain the HTML, CSS, and JavaScript code for a specific route in your application. Each page is a separate file that is located in the `/src/routes` folder.

To create a new page click on the folder you want to have your page in, then hover over new and select Svelte Component. Enter the name `+page.svelte` and you are done. The page will then be reachable under the directory you created it in.

Pages in svelte have to be named `+page.svelte` in order to be recognized as a page by the framework, since it would otherwise be recognized as a component. Do thus not name your pages differently.

Example 1: Creating a page

If you create the directory `/src/routes/about` with and then create a page inside with the name `+page.svelte`, then the route to that page will be `http://localhost:5173/about`.

3.1.2 Layouts

Layouts in Svelte are just like any other page, however like components they are reusable. Layouts are used to define the layout of your application including things such as a header and footer, and they can be used to create a consistent look and feel across all the pages in your application. Layouts apply to all pages that are in the same directory and all subdirectories of that directory.

To create a new layout click on the folder you want to have your page in, then hover over new and select Svelte Component. Enter the name `+layout.svelte` and you are done. To apply the UI that is created in a layout make sure to add the `<slot />` tag to your Layout, which is used to mark the spot where the UI of other pages is added. The layout will then be applied to all pages in the directory you created it in and all subdirectories.

Layouts in svelte have to be named `+layout.svelte` in order to be recognized as a layout by the framework, since it would otherwise be recognized as a component. Do thus not name your layouts differently.

Example 2: Creating a layout

Create the directory `/src/routes/about` and then create a page inside. Now create a page and layout in the `/src/routes` folder. Create some basic HTML code in the layout file:

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <meta name="viewport" content="width=device-width, initial-
scale=1"/>
        <title>SvelteKit Application</title>
    </head>
    <body>
        <!-- The Code you want to have on every page above the pages
content goes here-->
        <slot /> <!-- The content of every page that the layout is
applied to goes here-->
        <!-- The Code you want to have on every page below the pages
content goes here-->
    </body>
</html>
```

Now the code that you wrote into this file (this also includes any type of styling and scripting) will be applied to the page in `/src/routes` and the page `/src/routes/about`.

3.2 Data

3.3 Headers & Cookies

3.4 Forms

3.5 Handlers

3.6 Stores

3.7 Errors & Redirects