Svelte 5 & SvelteKit: From Zero to Production

> *A comprehensive, project-driven course covering every layer of modern Svelte development — from your very first component to a fully deployed, production-grade application.*

Table of Contents

Chapter 1 — Welcome to Svelte

1.1 What Is Svelte?

Svelte is a **compiler-first** UI framework. Unlike React or Vue, which ship a runtime to the browser, Svelte shifts the work to build time. The result is smaller bundles, faster startup, and less boilerplate.

1.2 Why Svelte 5?

Svelte 5 introduces **Runes** — a set of universal, explicit primitives for reactivity (`$state` , `$derived` , `$effect` , `$props`). Runes replace the implicit reactivity of Svelte 3/4, making code easier to reason about, refactor, and share across files.

1.3 Setting Up Your Environment

```bash
# Prerequisites: Node.js >= 18
node -v

# Create a new SvelteKit project
npx sv create my-app
cd my-app
npm install
npm run dev
```

1.4 Project Structure Overview

```
my-app/
├── src/
│   ├── routes/          # SvelteKit pages
│   │   └── +page.svelte
│   ├── lib/             # Shared components & utilities
│   └── app.html         # HTML shell
├── static/              # Public assets
├── svelte.config.js     # Svelte configuration
├── vite.config.ts       # Vite configuration
└── package.json
```

Exercise 1

1. Scaffold a brand-new SvelteKit project.
2. Start the dev server and open `http://localhost:5173`.
3. Edit `src/routes/+page.svelte` — change the heading text and confirm hot-reload works.

Chapter 2 — Your First Component

2.1 Anatomy of a `.svelte` File

Every `.svelte` file has up to three top-level sections:

```
<script>
  // JavaScript logic
</script>

<!-- Markup (HTML) -->
<h1>Hello!</h1>

<style>
  /* Scoped CSS */
  h1 { color: teal; }
</style>
```

2.2 Rendering Expressions

Use curly braces `{}` to embed JavaScript expressions inside markup:

```
<script>
  const name = 'World';
</script>

<h1>Hello, {name}!</h1>
<p>The answer is {6 * 7}.</p>
```

2.3 Creating a Reusable Component

```
<!-- src/lib/Greeting.svelte -->
<script>
  let { name } = $props();
</script>
```

```
<p>Welcome, {name}!</p>
```

Consume it in a page:

```
<!-- src/routes/+page.svelte -->
<script>
  import Greeting from '$lib/Greeting.svelte';
</script>

<Greeting name="Ada" />
```

Exercise 2

1. Create a `Card.svelte` component that accepts a `title` prop.
2. Import and render three `Card` instances on the home page with different titles.

Chapter 3 — Reactivity with Runes: `$state`

3.1 Declaring Reactive State

`$state` creates a deeply-reactive signal. Whenever the value changes, every place that reads it is automatically updated.

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

<button onclick={() => count++}>
  Clicks: {count}
</button>
```

3.2 Objects and Arrays

`$state` makes objects and arrays deeply reactive:

```
<script>
  let todos = $state([
    { text: 'Learn Svelte', done: false },
    { text: 'Build app', done: false }
  ]);

  function addTodo(text) {
    todos.push({ text, done: false });
  }
</script>

{#each todos as todo}
  <label>
    <input type="checkbox" bind:checked={todo.done} />
    {todo.text}
  </label>
{/each}
```

3.3 `$state` Outside Components

You can use `$state` in plain `.svelte.js` or `.svelte.ts` files for shared reactive stores:

```
// src/lib/counter.svelte.js
export function createCounter(initial = 0) {
  let count = $state(initial);

  return {
    get count() { return count; },
    increment() { count++; },
    reset() { count = initial; }
  };
}
```

Exercise 3

1. Build a counter with increment, decrement, and reset buttons.
2. Extract the counter logic into a shared `createCounter` function in `$lib`.
3. Use the shared counter in two different components and verify they share state when given the same instance.

Chapter 4 — Reactivity with Runes: `$derived`

4.1 Computed Values

`$derived` creates a value that is automatically recalculated whenever its dependencies change:

```
<script>
  let width = $state(10);
  let height = $state(5);
  let area = $derived(width * height);
</script>

<p>Area: {area}</p>
```

4.2 `$derived.by` for Complex Computations

When the derivation requires more than a single expression, use `$derived.by`:

```
<script>
  let items = $state([5, 3, 8, 1, 9]);

  let stats = $derived.by(() => {
    const sorted = [...items].sort((a, b) => a - b);
    return {
      min: sorted[0],
      max: sorted[sorted.length - 1],
      sum: sorted.reduce((a, b) => a + b, 0)
    };
  });
</script>

<p>Min: {stats.min}, Max: {stats.max}, Sum: {stats.sum}</p>
```

Exercise 4

1. Create a temperature converter: the user types Celsius, and Fahrenheit is `$derived`.

2. Add a list of numbers and display `$derived.by` statistics (mean, median, mode).

Chapter 5 — Reactivity with Runes: `$effect`

5.1 Side Effects

`$effect` runs **after** the DOM updates, whenever its dependencies change. It is the Svelte 5 equivalent of `onMount` + reactive statements combined:

```
<script>
  let query = $state('');

  $effect(() => {
    console.log('Search query changed:', query);
  });
</script>

<input bind:value={query} placeholder="Search..." />
```

5.2 Cleanup

Return a cleanup function for teardown (timers, event listeners, subscriptions):

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

  $effect(() => {
    const id = setInterval(() => seconds++, 1000);
    return () => clearInterval(id);
  });
</script>

<p>Elapsed: {seconds}s</p>
```

5.3 `$effect.pre`

Runs **before** the DOM updates — useful for things like scroll preservation:

```
<script>
  let messages = $state([]);

  $effect.pre(() => {
    // measure scroll position before DOM changes
  });
</script>
```

5.4 When NOT to Use `$effect`

Avoid using `$effect` to synchronise state — use `$derived` instead. Effects are for **side effects** (DOM manipulation, network requests, logging).

Exercise 5

1. Build a stopwatch using `$effect` with cleanup.
2. Create a component that logs every prop change to the console via `$effect`.

3. Implement a "debounced search" that waits 300 ms after the user stops typing before logging the query.

Chapter 6 — Reactivity with Runes: `$props`

6.1 Declaring Props

In Svelte 5, you declare component props with `$props()` :

```
<script>
  let { name, age = 25 } = $props();
</script>

<p>{name} is {age} years old.</p>
```

6.2 Rest Props

Spread remaining props onto an element:

```
<script>
  let { variant = 'primary', children, ...rest } = $props();
</script>

<button class="btn btn-{variant}" {...rest}>
  {@render children()}
</button>
```

6.3 Reactive Props

Props received via `$props` are reactive — when the parent changes the value, the child re-renders automatically.

Exercise 6

1. Create a `Button` component that accepts `variant` , `size` , and rest props.
2. Create an `Alert` component with a `type` prop (`info` , `warning` , `error`) that changes the background colour.

Chapter 7 — Template Syntax

7.1 Conditionals

```
{#if loggedIn}
  <p>Welcome back!</p>
{:else}
  <p>Please log in.</p>
{/if}
```

7.2 Each Blocks

```
{#each items as item, index (item.id)}
  <li>{index + 1}. {item.name}</li>
{/each}
```

7.3 Await Blocks

```
{#await fetchData()}
  <p>Loading...</p>
{:then data}
  <pre>{JSON.stringify(data, null, 2)}</pre>
{:catch error}
  <p>Error: {error.message}</p>
{/await}
```

7.4 Key Blocks

Force a component to re-mount when a value changes:

```
{#key selectedId}
  <Profile id={selectedId} />
{/key}
```

7.5 `{@html}` for Raw HTML

```
{@html '<em>Rendered as HTML</em>'}
```

> ***Warning:*** *Only use* `{@html}` *with trusted content to avoid XSS.*

Exercise 7

1. Build a user list with `{#each}` that shows a loading spinner via `{#await}` .
2. Add conditional rendering for empty states.

Chapter 8 — Events and Binding

8.1 Event Handlers

Svelte 5 uses standard HTML event attributes:

```
<button onclick={(e) => console.log('Clicked!', e)}>
  Click me
</button>
```

8.2 Two-Way Binding

```
<script>
  let name = $state('');
</script>

<input bind:value={name} />
<p>Hello, {name}!</p>
```

8.3 Binding to Component State

```
<input type="checkbox" bind:checked={agreed} />
<select bind:value={selected}>
  <option value="a">A</option>
  <option value="b">B</option>
</select>
```

8.4 Bind to DOM Elements

```
<script>
  let inputEl = $state();

  function focusInput() {
    inputEl.focus();
  }
</script>

<input bind:this={inputEl} />
<button onclick={focusInput}>Focus</button>
```

Exercise 8

1. Create a form with `bind:value` for text inputs, `bind:checked` for a checkbox, and `bind:group` for radio buttons.
2. Display the live form state below the form.

Chapter 9 — Component Props and Children Snippets

9.1 Snippets

Snippets are Svelte 5's replacement for slots. They allow you to pass renderable content to components:

```
<!-- Parent -->
<Card>
  {#snippet header()}
    <h2>My Card Title</h2>
  {/snippet}

  {#snippet footer()}
    <button>Close</button>
  {/snippet}

  <p>This is the default (children) content.</p>
</Card>
```

9.2 Rendering Snippets

Inside the component, render snippets with `{@render}`:

```
<!-- Card.svelte -->
<script>
  let { header, footer, children } = $props();
</script>
```

```
<div class="card">
  {#if header}
    <div class="card-header">{@render header()}</div>
  {/if}

  <div class="card-body">
    {@render children()}
  </div>

  {#if footer}
    <div class="card-footer">{@render footer()}</div>
  {/if}
</div>
```

9.3 Snippets with Parameters

Snippets can accept arguments, enabling render-prop patterns:

```
<!-- List.svelte -->
<script>
  let { items, row } = $props();
</script>

<ul>
  {#each items as item}
    <li>{@render row(item)}</li>
  {/each}
</ul>

<!-- Usage -->
<List {items}>
  {#snippet row(item)}
    <span>{item.name} — {item.email}</span>
  {/snippet}
</List>
```

Exercise 9

1. Build a reusable `Modal` component that accepts `header`, `children`, and `footer` snippets.
2. Create a `DataTable` that uses a snippet-with-parameters pattern for custom row rendering.

Chapter 10 — Styling in Svelte

10.1 Scoped Styles

All `<style>` blocks are scoped to the component by default:

```
<p>This is teal.</p>

<style>
  p { color: teal; }
</style>
```

10.2 Global Styles

Escape scoping when necessary:

```
<style>
  :global(body) {
    margin: 0;
    font-family: system-ui;
  }
</style>
```

10.3 CSS Custom Properties as Props

```
<Card --card-bg="salmon" --card-radius="12px" />

<!-- Card.svelte -->
<div class="card">
  {@render children()}
</div>

<style>
  .card {
    background: var(--card-bg, white);
    border-radius: var(--card-radius, 8px);
  }
</style>
```

10.4 Dynamic Classes

```
<div class:active={isActive} class:error={hasError}>
  Content
</div>
```

Exercise 10

1. Build a theme-able `Button` component using CSS custom properties.
2. Create a dark-mode toggle that swaps custom properties on `:root`.

Chapter 11 — Introduction to SvelteKit

11.1 What Is SvelteKit?

SvelteKit is the **full-stack application framework** for Svelte. It provides:

- File-based routing
- Server-side rendering (SSR)
- API routes (server endpoints)
- Code splitting
- Prerendering
- Adapter-based deployment

11.2 Architecture Overview

```
Request → SvelteKit Router → Load Function → Component → HTML Response
```

11.3 Key Configuration

```
// svelte.config.js
import adapter from '@sveltejs/adapter-auto';

export default {
  kit: {
    adapter: adapter()
  }
};
```

Exercise 11

1. Explore the default SvelteKit project — identify the route files, layout, and config.
2. Add a second page at `/about` and navigate between pages.

Chapter 12 — Routing

12.1 File-Based Routes

Each folder inside `src/routes/` becomes a URL path:

```
src/routes/
├── +page.svelte          →  /
├── about/
│   └── +page.svelte      →  /about
├── blog/
│   ├── +page.svelte      →  /blog
│   └── [slug]/
│       └── +page.svelte  →  /blog/:slug
```

12.2 Dynamic Parameters

```
<!-- src/routes/blog/[slug]/+page.svelte -->
<script>
  let { data } = $props();
</script>

<h1>{data.post.title}</h1>
```

12.3 Rest Parameters

```
src/routes/docs/[...path]/+page.svelte  →  /docs/any/nested/path
```

12.4 Route Groups

Use parentheses for layout grouping without affecting the URL:

```
src/routes/
├── (marketing)/
│   ├── +layout.svelte
│   ├── +page.svelte       →  /
│   └── pricing/
│       └── +page.svelte   →  /pricing
├── (app)/
│   ├── +layout.svelte
│   └── dashboard/
│       └── +page.svelte   →  /dashboard
```

12.5 Navigation

```
<a href="/about">About</a>

<!-- Programmatic navigation -->
<script>
  import { goto } from '$app/navigation';
  goto('/dashboard');
</script>
```

Exercise 12

1. Create a blog route with a dynamic `[slug]` parameter.
2. Display the slug on the page.
3. Add a "back to blog" link using an `<a>` tag.

Chapter 13 — Layouts

13.1 Root Layout

The root layout wraps every page:

```
<!-- src/routes/+layout.svelte -->
<script>
  let { children } = $props();
</script>

<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
</nav>

<main>
  {@render children()}
</main>

<footer>&copy; 2026</footer>
```

13.2 Nested Layouts

```
<!-- src/routes/dashboard/+layout.svelte -->
<script>
  let { children } = $props();
</script>

<aside>Sidebar</aside>
<div class="content">
```

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

13.3 Layout Data

```
// src/routes/+layout.server.js
export async function load() {
  return {
    siteName: 'My App'
  };
}
```

13.4 Error Pages

```
<!-- src/routes/+error.svelte -->
<script>
  import { page } from '$app/state';
</script>

<h1>{page.status}</h1>
<p>{page.error?.message}</p>
```

Exercise 13

1. Create a root layout with a navigation bar and a footer.
2. Add a nested layout for `/dashboard` that includes a sidebar.
3. Create a custom error page.

Chapter 14 — Loading Data

14.1 Server Load Functions

```
// src/routes/blog/+page.server.js
export async function load({ fetch }) {
  const res = await fetch('/api/posts');
  const posts = await res.json();

  return { posts };
}
```

14.2 Universal Load Functions

Run on both server and client:

```
// src/routes/blog/+page.js
export async function load({ fetch }) {
  const res = await fetch('/api/posts');
  return { posts: await res.json() };
}
```

14.3 Using Loaded Data

```
<!-- src/routes/blog/+page.svelte -->
<script>
  let { data } = $props();
</script>

{#each data.posts as post}
  <article>
    <h2><a href="/blog/{post.slug}">{post.title}</a></h2>
    <p>{post.excerpt}</p>
  </article>
{/each}
```

14.4 Params in Load Functions

```
// src/routes/blog/[slug]/+page.server.js
export async function load({ params }) {
  const post = await getPost(params.slug);

  if (!post) {
    throw error(404, 'Post not found');
  }

  return { post };
}
```

14.5 API Routes (Server Endpoints)

```
// src/routes/api/posts/+server.js
import { json } from '@sveltejs/kit';

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

Exercise 14

1. Create a `+page.server.js` that returns an array of mock blog posts.
2. Display them on the page using `data.posts`.
3. Add a dynamic route that loads a single post by slug.

Chapter 15 — Forms and Actions

15.1 Form Actions

SvelteKit provides a progressive-enhancement-first approach to forms:

```
// src/routes/login/+page.server.js
export const actions = {
  default: async ({ request }) => {
    const formData = await request.formData();
    const email = formData.get('email');
    const password = formData.get('password');
    // Validate & authenticate
    if (!email) {
      return fail(400, { email, missing: true });
    }

    // Redirect on success
    redirect(303, '/dashboard');
  }
};
```

15.2 The Form Component

```
<!-- src/routes/login/+page.svelte -->
<script>
  import { enhance } from '$app/forms';
  let { form } = $props();
</script>

<form method="POST" use:enhance>
  <input name="email" value={form?.email ?? ''} />
  {#if form?.missing}
    <p class="error">Email is required.</p>
  {/if}
  <input name="password" type="password" />
  <button>Log In</button>
</form>
```

15.3 Named Actions

```
export const actions = {
  login: async ({ request }) => { /* ... */ },
  register: async ({ request }) => { /* ... */ }
};
```

```
<form method="POST" action="?/register" use:enhance>
  <!-- ... -->
</form>
```

15.4 Progressive Enhancement with `use:enhance`

`use:enhance` upgrades the form to use `fetch` instead of a full page reload, while keeping it functional without JavaScript.

Exercise 15

1. Build a contact form with `name` , `email` , and `message` fields.
2. Add server-side validation in a form action.
3. Display validation errors inline using the `form` prop.
4. Add `use:enhance` for a no-reload experience.

Chapter 16 — TypeScript in SvelteKit

16.1 Enabling TypeScript

SvelteKit supports TypeScript out of the box. Use `<script lang="ts">` :

```
<script lang="ts">
  let count: number = $state(0);
</script>
```

16.2 Typing Props

```
<script lang="ts">
  interface Props {
    name: string;
    age?: number;
  }

  let { name, age = 25 }: Props = $props();
</script>
```

16.3 Typing Load Functions

```
// src/routes/blog/+page.server.ts
import type { PageServerLoad } from './$types';

export const load: PageServerLoad = async ({ fetch }) => {
  const res = await fetch('/api/posts');
  const posts: Post[] = await res.json();
  return { posts };
};
```

16.4 Typing Form Actions

```
import type { Actions } from './$types';
import { fail, redirect } from '@sveltejs/kit';

export const actions: Actions = {
  default: async ({ request }) => {
    const data = await request.formData();
    // ...
  }
};
```

Exercise 16

1. Convert an existing component to use `lang="ts"` .
2. Create typed interfaces for your data models.
3. Type a load function and a form action.

Chapter 17 — SEO and Prerendering

17.1 The `<svelte:head>` Element

```
<svelte:head>
  <title>My Page Title</title>
  <meta name="description" content="A description for search engines." />
  <meta property="og:title" content="My Page Title" />
  <meta property="og:image" content="/og-image.png" />
</svelte:head>
```

17.2 Dynamic SEO Data

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

<svelte:head>
  <title>{data.post.title} | My Blog</title>
  <meta name="description" content={data.post.excerpt} />
</svelte:head>
```

17.3 Prerendering

Prerender static pages at build time:

```
// src/routes/about/+page.js
export const prerender = true;
```

Prerender the entire site:

```
// svelte.config.js
export default {
  kit: {
    prerender: {
      entries: ['*']
    }
  }
};
```

17.4 Sitemap and robots.txt

```
// src/routes/sitemap.xml/+server.js
export async function GET() {
  const pages = await getPages();
  const xml = buildSitemap(pages);
  return new Response(xml, {
    headers: { 'Content-Type': 'application/xml' }
  });
}
```

Exercise 17

1. Add `<svelte:head>` to every page with appropriate title and meta tags.
2. Mark the `/about` page as prerendered.
3. Create a `/sitemap.xml` server endpoint.

Chapter 18 — GSAP Animation, Testing, and Deployment

18.1 GSAP in Svelte

```
npm install gsap
```

```
<script>
  import { gsap } from 'gsap';

  let box = $state();

  $effect(() => {
    gsap.from(box, {
      duration: 1,
      opacity: 0,
      y: 50,
      ease: 'power3.out'
    });
  });
</script>

<div bind:this={box} class="box">Animated!</div>
```

18.2 GSAP ScrollTrigger

```
<script>
  import { gsap } from 'gsap';
  import { ScrollTrigger } from 'gsap/ScrollTrigger';

  gsap.registerPlugin(ScrollTrigger);

  let section = $state();

  $effect(() => {
    gsap.from(section, {
      scrollTrigger: {
        trigger: section,
        start: 'top 80%'
      },
      opacity: 0,
      y: 100,
      duration: 1
    });
  });
</script>

<section bind:this={section}>
  <h2>Scroll to reveal</h2>
</section>
```

18.3 Testing with Vitest

```
npm install -D vitest @testing-library/svelte jsdom
```

```
// src/lib/Counter.test.ts
import { render, screen, fireEvent } from '@testing-library/svelte';
import { describe, it, expect } from 'vitest';
import Counter from './Counter.svelte';

describe('Counter', () => {
  it('increments on click', async () => {
    render(Counter);
    const button = screen.getByRole('button');
    expect(button.textContent).toBe('Clicks: 0');
    await fireEvent.click(button);
    expect(button.textContent).toBe('Clicks: 1');
  });
});
```

18.4 End-to-End Testing with Playwright

```
npm install -D @playwright/test
```

```
// tests/home.spec.ts
import { test, expect } from '@playwright/test';

test('home page has heading', async ({ page }) => {
  await page.goto('/');
  await expect(page.locator('h1')).toBeVisible();
});
```

18.5 Deployment

SvelteKit uses **adapters** for deployment:

Target	Adapter
Node.js server	`@sveltejs/adapter-node`
Static site	`@sveltejs/adapter-static`
Vercel	`@sveltejs/adapter-vercel`
Cloudflare Pages	`@sveltejs/adapter-cloudflare`
Netlify	`@sveltejs/adapter-netlify`
Auto-detect	`@sveltejs/adapter-auto`

```
# Example: deploy to Vercel
npm install -D @sveltejs/adapter-vercel

# Build
npm run build
```

```
# Preview locally
npm run preview
```

Exercise 18

1. Add a GSAP entrance animation to your home page hero section.
2. Write a Vitest unit test for a component.
3. Write a Playwright E2E test that navigates between two pages.
4. Choose an adapter, build, and deploy your project.

Appendix A — Svelte 5 Runes Quick Reference

Rune	Purpose	Example
`$state`	Reactive state	`let x = $state(0)`
`$derived`	Computed value	`let d = $derived(x * 2)`
`$derived.by`	Complex computed value	`let d = $derived.by(() => { ... })`
`$effect`	Side effect (runs after update)	`$effect(() => { ... })`
`$effect.pre`	Side effect (runs before update)	`$effect.pre(() => { ... })`
`$props`	Component props	`let { a, b } = $props()`
`$bindable`	Two-way bindable prop	`let { value = $bindable() } = $props()`
`$inspect`	Debug logging (dev only)	`$inspect(count)`

Appendix B — SvelteKit File Conventions

File	Purpose
`+page.svelte`	Page component
`+page.js`	Universal load function
`+page.server.js`	Server load function
`+layout.svelte`	Layout component
`+layout.js`	Layout universal loader
`+layout.server.js`	Layout server loader
`+error.svelte`	Error page
`+server.js`	API route / endpoint

Congratulations! You have completed the Svelte 5 & SvelteKit course. You now have the knowledge to build, test, and deploy modern web applications with Svelte.