Svelte

A Complete 12-Chapter Course

Topics covered:

Compiler model & Runes · $state / $derived / $effect · Bindings & Events

Logic blocks · Components & Props · Communication · Snippets · Lifecycle

Shared reactive modules · SvelteKit Routing · Data Loading & Form Actions

Exercises: 36 hands-on challenges with sample solutions

Format: A4 · Dark-theme code examples · React/Vue/Angular comparison tables

Table of Contents

CHAPTER 1 OF 12

What Svelte Is, the .svelte File, Vite Setup

CHAPTER 1

What Svelte Is, the .svelte File, and Vite Setup

A framework that's actually a compiler — it disappears at build time, leaving tiny vanilla JS behind

Having worked through React, Angular, and Vue, Svelte is the genuine outlier — and the reason it's the perfect contrast piece. React, Angular, and Vue all ship a **runtime** to the browser: a framework library that runs alongside your code, doing reconciliation (a virtual DOM, or change detection) to figure out what to update. Svelte takes a fundamentally different path: it's a **compiler**. Your components are compiled at build time into small, surgical vanilla JavaScript that updates the DOM directly — and the framework itself mostly *disappears*, never shipped to the browser at all.

No Virtual DOM, No Runtime

When data changes in React or Vue, the framework re-runs render logic and diffs the result against the previous one to find what changed. Svelte's compiler instead analyzes your component *ahead of time* and generates precise code that updates exactly the right DOM node when exactly the right value changes — no diffing, no virtual DOM, no reconciliation library along for the ride. The practical results: very small bundles (no framework runtime weight) and excellent performance, with code that often looks like less than the equivalent in the other three.

"SVELTE" THE LANGUAGE VS "SVELTEKIT" THE META-FRAMEWORK

Svelte is the component language/compiler — Chapters 1–10 of this course. **SvelteKit** (Chapters 11–12) is its official meta-framework, adding file-based routing, server-side rendering, and data loading on top — the rough equivalent of Next.js for React or Nuxt for Vue. They're often used together, but Svelte the component framework is the foundation, so we start there.

The .svelte File

```svelte
<!-- Greeting.svelte -->
<script>
  let name = 'Philip';
</script>

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

<style>
  h1 { color: #ff3e00; }
</style>
```

A Svelte component is a `.svelte` file with three optional parts: a `<script>` block for logic, the **markup** (written directly, not wrapped in a `<template>` tag), and a `<style>` block. This is close to Vue's Single-File Component, with one notable difference: the markup is just *there* at the top level of the file — no enclosing template element. `{name}` is interpolation, the same single-brace style as React's JSX rather than Vue/Angular's double braces.

Scoped Styles by Default

The `<style>` block's rules are automatically scoped to this component only — no `scoped` attribute needed (Vue) and no setup at all (unlike React). Two components can both style `h1` differently with zero collision. Svelte achieves this by adding a unique class to the component's elements at compile time, with no runtime cost.

Creating a Project with Vite

```bash
# scaffold a Svelte project (component-only, no SvelteKit)
npm create vite@latest my-app -- --template svelte

cd my-app
npm install
npm run dev
```

`npm create vite@latest -- --template svelte` scaffolds a plain Svelte + Vite project — ideal for learning the component framework on its own (we add SvelteKit later). `npm run dev` starts the dev server with live reload, exactly as it did for React and Vue. To start a full SvelteKit app instead, you'd run `npx sv create my-app` — but the bare Vite template keeps the early chapters focused.

- `src/main.js` — the entry point; mounts the root `App` component.
- `src/App.svelte` — the root component, the first one you'll edit.
- `index.html` — the single real HTML page everything mounts into.

Mounting the Root Component

```
// src/main.js
import { mount } from 'svelte';
import App from './App.svelte';

const app = mount(App, { target: document.getElementById('app') });

export default app;
```

`mount(App, { target })` attaches the root component to an element in `index.html` — the Svelte 5 equivalent of React's `createRoot(...).render()`, Angular's `bootstrapApplication()`, and Vue's `createApp(App).mount()`. (Older Svelte code used `new App({ target })` with the `new` keyword — replaced by the `mount` function in Svelte 5.)

THIS COURSE TEACHES SVELTE 5 WITH RUNES — OLDER CODE LOOKS DIFFERENT

Svelte 5 (late 2024) introduced **runes** (`$state`, `$derived`, `$effect`, etc.) — a new, more explicit reactivity model used throughout this course. Svelte 3/4 code looks meaningfully different: reactivity came from plain `let` declarations plus `$:` reactive statements, and props used `export let`. That older style still works in Svelte 5 for now but is being superseded — the same "modern vs legacy" split as Vue's Composition-vs-Options API or Angular's standalone-vs-NgModule. Recognize the old syntax in tutorials; this course uses runes.

CONCEPT	REACT	VUE	SVELTE
Runs in browser as	Library + your code	Library + your code	Compiled vanilla JS (no runtime)
Update mechanism	Virtual DOM diff	Virtual DOM diff	Compiled direct DOM updates
Component file	`.jsx`	`.vue` (with `<template>`)	`.svelte` (markup at top level)
Interpolation	`{value}`	`{{ value }}`	`{value}`
Scoped styles	CSS Modules / lib	`<style scoped>`	`<style>` (automatic)

Coding Challenges

CHALLENGE 1

Create a new Svelte + Vite project, and edit App.svelte so it shows "Welcome, [your name]!" by interpolating a variable declared in the script block.

View solution

CHALLENGE 2

Build a Greeting.svelte component with all three blocks — a name variable in the script, an interpolated heading in the markup, and a scoped style coloring that heading.

View solution

CHALLENGE 3

Build a second component (e.g. Footer.svelte) and use it inside App.svelte by importing it in the script block and placing its tag in the markup — confirming a child component renders inside a parent.

View solution

CHAPTER 1 QUICK REFERENCE

- **Svelte is a compiler** — components compile to vanilla JS; no virtual DOM, no runtime shipped
- **.svelte file** — `<script>`, markup (at the top level, no `<template>`), and `<style>`
- **{value}** — interpolation, single braces like React's JSX
- **<style>** — automatically scoped to the component, no extra syntax
- **npm create vite -- --template svelte** / **npm run dev** — scaffold and run
- **mount(App, { target })** — bootstraps the root component (Svelte 5)
- This course uses **Svelte 5 runes**; older Svelte 3/4 used `$:` and `export let`
- **Next chapter:** reactivity with the `$state` rune

CHAPTER 2 OF 12

Reactivity with $state

CHAPTER 2

Reactivity with $state

Making a value reactive with a rune — and why a counter just works with plain assignment

Chapter 1's interpolated values never changed. For a value that changes and updates the UI, Svelte 5 uses the `$state` **rune** — a compiler keyword (recognizable by the `$` prefix) that marks a variable as reactive. This is Svelte's answer to React's `useState`, Vue's `ref`, and Angular's `signal` — but it reads the most like ordinary JavaScript of any of them.

$state — A Reactive Value

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

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

<p>Count: {count}</p>
<button onclick={increment}>+1</button>
```

`let count = $state(0)` declares a reactive variable holding `0`. The remarkable part: you read it as plain `count` and update it with plain assignment — `count++` — and the UI updates automatically. No `.value` (Vue), no setter function (React's `setCount`), no `.update()` call (Angular). The `$state` rune tells the compiler "track this variable," and the compiler rewrites your ordinary `count++` into the precise DOM update behind the scenes.

RUNES ARE COMPILER MAGIC, NOT FUNCTION CALLS

Although `$state(0)` looks like a function call, it isn't one you import — it's a **rune**, a special keyword the Svelte compiler recognizes. You never `import { $state }`; it's just available, like a language built-in. The `$` prefix is reserved for runes, which is how you spot them at a glance.

Why Not a Plain let?

```
<!-- this does NOT update the UI in Svelte 5 -->
<script>
  let count = 0;   // plain variable, not reactive
  function increment() {
    count++;   // changes the variable, but the UI never re-renders
  }
</script>
```

A plain `let count = 0` increments fine in memory, but Svelte 5 doesn't treat it as reactive — nothing tells the compiler to wire it to the DOM, so the displayed number never changes. The same trap as a plain variable in React or Vue. The `$state` rune is the single thing that makes it reactive.

> **THIS IS THE BIG SVELTE 3/4 → 5 CHANGE**
>
> In older Svelte (3/4), a plain `let count = 0` at the top of a component *was* automatically reactive — the compiler tracked all top-level `let` declarations. Svelte 5 made reactivity **explicit** via `$state` instead, because the old implicit magic didn't work outside components and had confusing edge cases. So if you see a reactive-looking plain `let` in a tutorial with no `$state`, it's pre-Svelte-5 code.

Reactive Objects and Arrays

```
let user = $state({ name: 'Philip', age: 35 });

function birthday() {
  user.age++;   // mutating a property works — it's deeply reactive
}

let todos = $state([]);

function add(text) {
  todos.push({ text });   // even .push() triggers an update
}
```

`$state` makes objects and arrays **deeply reactive** — mutating a nested property (`user.age++`) or even calling `.push()` on an array triggers a UI update. This is a notable contrast with React, where you must create a *new* object/array (spread, `map`, `filter`) because React compares by reference and never mutates. Svelte lets you mutate directly, which often reads more naturally — under the hood it uses a proxy (like Vue's `reactive`) to detect those mutations.

Markup Expressions

```
<p>{count * 2}</p>
<p>{name.toUpperCase()}</p>
<p>{isActive ? 'On' : 'Off'}</p>
```

Interpolation isn't limited to a bare value — any JavaScript expression works inside `{ }`: arithmetic, method calls, ternaries. Identical to React's curly braces and Vue's interpolation: expressions only, no statements. (For a value reused in several places or with heavier logic, you'd reach for `$derived` — next chapter.)

	REACT	VUE	SVELTE 5
Create	`useState(0)`	`ref(0)`	`$state(0)`
Read	`count`	`count.value` (script)	`count`
Update	`setCount(count+1)`	`count.value++`	`count++`
Object mutation	Forbidden (new copy)	Allowed (proxy)	Allowed (proxy)

Coding Challenges

CHALLENGE 1

Build a counter with a $state value starting at 0 and a +1 button that increments it with plain count++, displaying the count via interpolation.

View solution

CHALLENGE 2

Build a component with a $state object holding firstName and lastName, plus a button that changes one property directly (e.g. user.firstName = '...'), displaying "Full name: [first] [last]" with both updating live.

View solution

CHALLENGE 3

Build a component with a $state array of strings and an "Add item" button that pushes a new string onto it directly, confirming the rendered list updates even though you used .push() rather than creating a new array.

View solution

CHAPTER 2 QUICK REFERENCE

- **$state(value)** — declares a reactive variable (= `useState` / `ref` / `signal`)
- Read it as a plain variable; update it with **plain assignment** (`count++`) — no setter, no `.value`
- A **rune** is a compiler keyword (`$` prefix) — never imported, just available
- A plain `let` is **not** reactive in Svelte 5 — only `$state` is tracked
- `$state` objects/arrays are **deeply reactive** — mutation (`.push()`, `obj.x++`) triggers updates
- **{ }** accepts any JS expression (math, method calls, ternaries)
- **Next chapter:** derived state (`$derived`) and effects (`$effect`)

CHAPTER 3 OF 12

Derived State and Effects

CHAPTER 3

Derived State and Effects

$derived for values computed from other state, and $effect for side effects

Two more runes complete Svelte's reactivity core. `$derived` computes a value from other reactive state and keeps it up to date automatically — the counterpart to Vue's `computed`, Angular's `computed` signal, and React's `useMemo`. `$effect` runs a side effect whenever the state it reads changes — the counterpart to `watch` / `useEffect`. Both, like `$state`, track their dependencies automatically.

$derived — A Computed Value

```
<script>
  let price = $state(100);
  let quantity = $state(2);

  let total = $derived(price * quantity);
</script>

<p>Total: {total}</p>
```

`$derived(price * quantity)` creates a value that automatically recalculates whenever `price` or `quantity` changes. Notice the syntax is even lighter than the others: you pass the *expression itself*, not a function returning it (no `() =>`). It's read as a plain variable (`{total}`), just like `$state`. Like Vue's `computed` and Angular's `computed`, it's **cached** (only recomputes when a dependency changes) and its dependencies are **auto-tracked** — no dependency array like React's `useMemo`.

$derived.by — For Multi-Line Logic

```
let todos = $state([/* ... */]);
let showDone = $state(false);

let visible = $derived.by(() => {
  if (showDone) return todos;
  return todos.filter((t) => !t.done);
});
```

When the derivation needs more than a single expression, `$derived.by(() => {...})` takes a function with a `return` — useful for filtering, conditionals, or any multi-step computation. This is the natural home for "filter a list before rendering it," the same pattern seen in Vue's computed filter and React's filter-before-map.

$effect — Running a Side Effect

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

  $effect(() => {
    console.log(`count is now ${count}`);
  });
</script>
```

`$effect(() => {...})` runs its function after the component mounts, then again whenever any reactive value it reads changes — automatically tracked, no dependency list. This is for genuine **side effects**: logging, syncing to `localStorage`, manually touching the DOM or a third-party library. It's the equivalent of React's `useEffect` with auto-detected dependencies, or Vue's `watchEffect`.

Effect Cleanup

```
$effect(() => {
  const id = setInterval(() => console.log('tick'), 1000);

  return () => clearInterval(id);   // cleanup: runs before re-run, and on unmount
});
```

Returning a function from an `$effect` gives a **cleanup function** — run before the effect re-runs, and when the component is destroyed. This is exactly React's `useEffect` cleanup return, used the same way: anything that "starts" something ongoing (a timer, a listener, a subscription) returns its matching "stop." This is also how you handle teardown that the other frameworks put in a separate lifecycle hook (Vue's `onUnmounted`, Angular's `ngOnDestroy`).

DON'T REACH FOR $EFFECT TO COMPUTE A VALUE — USE $DERIVED

The most common misuse is using an `$effect` to set one piece of state from another (`$effect(() => { doubled = count * 2 })`). That's almost always a `$derived` in disguise, and the `$derived` version is simpler, cached, and avoids extra re-render cycles. The same rule as Vue's watch-vs-computed: **deriving a value → `$derived`; doing something with a side effect → `$effect`**. Svelte will even warn you about some of these cases.

THIS REPLACED SVELTE 4'S $: REACTIVE STATEMENTS

Pre-Svelte-5 code used a label syntax for both jobs: `$: total = price * quantity` for derivations and `$: console.log(count)` for effects — the same `$:` prefix doing double duty, which was a frequent source of confusion. Svelte 5 split them into two clearly-named runes (`$derived` and `$effect`). If you see `$:` in a tutorial, that's the old combined form.

SVELTE 5	VUE	REACT	ANGULAR
`$derived(expr)`	`computed(() => ...)`	`useMemo` (dep array)	`computed()` signal
`$derived.by(() => {...})`	`computed(() => {...})`	`useMemo`	`computed()`
`$effect(() => {...})`	`watchEffect`	`useEffect` (auto)	`effect()` signal
`return cleanup from $effect`	`onUnmounted` / watch stop	cleanup return	`ngOnDestroy`

Coding Challenges

CHALLENGE 1

Build a component with firstName and lastName $state values and a fullName $derived combining them, with inputs to edit each — confirming fullName updates automatically.

View solution

CHALLENGE 2

Build a todo list with a "show completed" checkbox, using $derived.by to compute the visible todos based on the checkbox, then render that filtered list.

View solution

CHALLENGE 3

Build a component with a count $state and an $effect that persists count to localStorage on every change, reading the saved value back as the initial value so it survives a refresh — plus a separate $effect with a setInterval and a cleanup

return.

View solution

CHAPTER 3 QUICK REFERENCE

- **$derived(expr)** — a cached, auto-tracked computed value; pass the expression, not a function
- **$derived.by(() => {...})** — the function form for multi-line / conditional logic
- **$effect(() => {...})** — runs a side effect on mount and whenever read state changes (auto-tracked)
- Return a function from `$effect` for **cleanup** (= React's useEffect cleanup)
- Rule of thumb: **deriving a value → $derived; a side effect → $effect**
- These split Svelte 4's combined `$:` reactive statements into two clear runes
- **Next chapter:** bindings and events — bind:value, event handlers, class:/style: directives

CHAPTER 4 OF 12

Bindings and Events

CHAPTER 4

Bindings and Events

Two-way binding with bind:, event handlers, and the class:/style: directives

With reactivity covered, this chapter connects it to the DOM: handling events, two-way-binding form inputs, and toggling classes/styles conditionally. Svelte leans toward standard HTML here — event handlers are plain DOM attributes — with a few concise directives layered on top.

Event Handlers

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

<button onclick={() => count++}>+1</button>
<input oninput={handleInput} />
```

In Svelte 5, events are just standard lowercase DOM attributes — `onclick`, `oninput`, `onsubmit` — set to a function. `onclick={() => count++}` uses an inline arrow; `oninput={handleInput}` passes a function reference. This is closer to plain HTML than any of the other frameworks (which all use their own syntax — `onClick` in React, `(click)` in Angular, `@click` in Vue). The native event object is the handler's argument, exactly as in vanilla JS.

SVELTE 4 USED ON:CLICK — SVELTE 5 USES ONCLICK

Pre-Svelte-5 code used a directive form with a colon: `on:click={handler}`. Svelte 5 replaced it with the plain DOM-attribute form `onclick={handler}` (no colon), aligning with standard HTML. Both may appear during the transition, but `onclick` is the modern style this course uses. The `on:` version is the giveaway for older code.

bind: — Two-Way Binding

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

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

`bind:value={name}` keeps a form input and a `$state` variable synchronized in both directions — typing updates `name`, and changing `name` in code updates the input. This is Svelte's equivalent of Vue's `v-model` and Angular's `[(ngModel)]`, and it collapses React's manual `value` + `onChange` controlled-input pattern into one directive. The `bind:` prefix marks a two-way binding (vs a one-way attribute).

bind: on Different Input Types

```svelte
<input type="checkbox" bind:checked={agreed} />   <!-- boolean -->
<input type="number" bind:value={age} />      <!-- auto-coerced to a number -->
<select bind:value={country}>                  <!-- string -->
  <option value="ie">Ireland</option>
  <option value="hu">Hungary</option>
</select>
```

Like Vue's `v-model`, Svelte's `bind:` adapts to the input type — checkboxes use `bind:checked` (a boolean), a `<select>` binds to the chosen option's value, and a `type="number"` input even **auto-coerces** the value to a number for you (a small nicety React makes you do by hand). Same simplicity benefit as Vue over React's per-type handling.

class: — Toggling a Class

```svelte
<div class:active={isActive}>...</div>

<!-- shorthand when the variable name matches the class name -->
<div class:active>...</div>
```

`class:active={isActive}` adds the `active` class when `isActive` is truthy and removes it otherwise — a clean, declarative toggle, the equivalent of Angular's `[class.active]` and Vue's class binding. When the variable name matches the class name, you can shorten it to just `class:active`. Multiple `class:` directives can sit on one element.

style: — Binding a Style Property

```
<p style:color={isError ? 'red' : 'black'}>...</p>
<div style:width={`${progress}%`}>...</div>
```

`style:color={...}` binds a single inline style property to a reactive expression — handy for values that change dynamically, like a progress bar's width or a conditional color. The parallel of Angular's `[style.color]`, expressed as a directive.

Form Submission

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

  function handleSubmit(e) {
    e.preventDefault();
    console.log(value);
    value = '';
  }
</script>

<form onsubmit={handleSubmit}>
  <input bind:value={value} />
  <button>Submit</button>
</form>
```

Svelte uses the standard `onsubmit` handler and standard `e.preventDefault()` — no special "prevent" modifier like Vue's `@submit.prevent`. Because `bind:value` keeps `value` in sync, the handler reads the current value directly and clears it by reassigning, with reactivity handling the rest.

SVELTE	VUE	ANGULAR	REACT
onclick={fn}	`@click="fn"`	`(click)="fn()"`	`onClick={fn}`
bind:value={x}	`v-model="x"`	`[(ngModel)]="x"`	`value + onChange`
class:active={x}	class binding	`[class.active]`	conditional className
style:color={x}	`:style`	`[style.color]`	`style={{}}`

Coding Challenges

CHALLENGE 1

Build a counter with +1/-1/reset buttons using onclick handlers (inline arrows are fine), displaying the $state count.

View solution

CHALLENGE 2

Build a form with a text input (bind:value), a checkbox (bind:checked), and a select (bind:value), each bound to its own $state variable, displaying all three values live below the form.

View solution

CHALLENGE 3

Build a "toggle" component: a button that flips an isOn $state boolean, a div whose active class is toggled with class:active={isOn}, and a style:background bound to a color that changes based on isOn.

View solution

CHAPTER 4 QUICK REFERENCE

- **onclick={fn}** — standard lowercase DOM event attributes (Svelte 5; was `on:click` in v4)
- **bind:value={x}** — two-way binding (= `v-model` / `[(ngModel)]`); no import needed
- `bind:` adapts to type: **bind:checked** (boolean), `type="number"` auto-coerces
- **class:active={x}** — toggle a class; shorthand `class:active` when names match
- **style:prop={x}** — bind a single inline style property
- Forms use standard `onsubmit` + `e.preventDefault()` — no special modifier
- **Next chapter:** conditional and list rendering — {#if}, {#each}, {#await}

CHAPTER 5 OF 12

Conditional and List Rendering

CHAPTER 5

Conditional and List Rendering

Logic blocks in the markup — {#if}, {#each}, and the rather neat {#await}

Svelte handles conditionals and lists with **logic blocks** — special `{#...}` markers in the markup, closed with `{/...}`. This is a distinct style from all three other frameworks: not JavaScript-in-JSX (React), not directives-on-elements (Vue/Angular), but dedicated block syntax that reads almost like a templating language. Svelte even has a block for handling promises directly, which the others lack.

{#if} / {:else if} / {:else}

```
{#if status === 'loading'}
  <p>Loading...</p>
{:else if status === 'error'}
  <p>Something went wrong.</p>
{:else}
  <p>Ready!</p>
{/if}
```

An `{#if}` block conditionally renders its contents, with optional `{:else if}` and `{:else}` branches. The opening tag uses `#`, continuation branches use `:`, and the close uses `/` — a consistent convention across all Svelte blocks. Like Angular's `@if` and Vue's `v-if`, a false branch genuinely removes its elements from the DOM. The whole `{#if}/{:else if}/{:else}` chain cleanly handles the loading/error/success pattern.

{#each} — Rendering a List

```
<ul>
  {#each fruits as fruit}
    <li>{fruit}</li>
  {/each}
</ul>
```

`{#each items as item}` repeats its contents once per array entry — Svelte's `.map()` equivalent. For lists that change (items added, removed, reordered), you should provide a **key** in parentheses so Svelte can track each

item efficiently — the same role as React's `key`, Vue's `:key`, and Angular's `track`:

```
{#each todos as todo (todo.id)}
  <li>{todo.text}</li>
{/each}
```

The `(todo.id)` after the item is the key. Unlike Angular's mandatory `track`, it's optional in Svelte — but strongly recommended for any list that mutates, for exactly the same correctness reasons.

{#each} with Index and an Empty Fallback

```
{#each todos as todo, index (todo.id)}
  <li>{index + 1}. {todo.text}</li>
{:else}
  <li>No todos yet.</li>
{/each}
```

A second variable after the item gives the index — `todo, index`. And neatly, `{#each}` supports its own `{:else}` branch, rendered when the array is empty — handling the "empty list" case inline, the same convenience as Angular's `@empty` but built right into the each-block.

{#await} — Rendering a Promise Directly

```
{#await promise}
  <p>Loading...</p>
{:then data}
  <p>Got: {data.title}</p>
{:catch error}
  <p>Failed: {error.message}</p>
{/await}
```

This block has no equivalent in the other frameworks. `{#await promise}` renders the three states of a Promise directly in the markup: pending (before `{:then}`), resolved (`{:then data}` with the result), and rejected (`{:catch error}`). The loading/error/success pattern that took manual `status` state in every other framework's data-fetching is handled declaratively here, with the promise itself as the source of truth. (The `{:then}` and `{:catch}` branches are both optional if you only care about some states.)

> **{#AWAIT} IS GENUINELY DISTINCTIVE**
>
> Every other framework makes you manage `isLoading` / `error` / `data` state by hand (or reach for a library like React Query). Svelte bakes the three-state promise lifecycle into a template block. Pass a fetch promise straight in —

`{#await fetch(url).then(r => r.json())}` — and the markup handles all three states with no extra state variables at all.

NO V-IF + V-FOR PROBLEM HERE — BUT KEY YOUR DYNAMIC LISTS

Svelte doesn't have Vue's "don't combine v-if and v-for" issue (the blocks nest cleanly). The one thing to stay disciplined about is keying: an unkeyed `{#each}` over a list that gets reordered or filtered can attach the wrong state to the wrong item — the same class of bug a missing React `key` causes. Add `(item.id)` to any list that changes.

SVELTE	VUE	ANGULAR	REACT
`{#if} / {:else if} / {:else}`	`v-if / v-else-if / v-else`	`@if / @else if`	ternary / `&&`
`{#each x as item (id)}`	`v-for + :key`	`@for + track`	`.map()` + `key`
`{#each ...}{:else}`	separate check	`@empty`	separate check
`{#await} / {:then} / {:catch}`	—	—	— (or React Query/Suspense)

Coding Challenges

CHALLENGE 1

Build a component with a status $state ("loading"/"error"/"success") cycled by a button, using {#if}/{:else if}/{:else} to show different content per status.

View solution

CHALLENGE 2

Given a $state array of objects ({ id, name }), render them with {#each} (keyed on id) in a numbered list using the index, and include a {:else} branch showing "No items" when the array is cleared by a button.

View solution

CHALLENGE 3

Build a component that fetches from any free public API into a promise and renders it with {#await}/{:then}/{:catch} — showing a loading message, the result on success, and an error message on failure, with no manual status state.

View solution

CHAPTER 5 QUICK REFERENCE

- **{#if} / {:else if} / {:else} / {/if}** — conditional rendering; `#` opens, `:` continues, `/` closes
- **{#each items as item (id)}** — list rendering; the `(id)` key is optional but advised for dynamic lists
- **{#each items as item, index}** — also exposes the index
- **{#each …}{:else}** — built-in empty-list fallback (= Angular's `@empty`)
- **{#await promise}{:then data}{:catch error}** — render a Promise's three states inline (unique to Svelte)
- Key any list that gets reordered/filtered, the same discipline as React's `key`
- **Next chapter:** components and props ($props)

CHAPTER 6 OF 12

Components and Props

CHAPTER 6

Components and Props

Passing data into a child with the $props rune — destructuring with defaults

Chapter 1's child components were static. **Props** make a component reusable by letting a parent pass data in — the same concept as React props, Vue's `defineProps`, and Angular's `@Input`. Svelte 5 declares them with the `$props` rune, and the result reads like plain JavaScript object destructuring.

Declaring Props with $props

```
<!-- UserCard.svelte -->
<script>
  let { name, age } = $props();
</script>

<h3>{name}</h3>
<p>Age: {age}</p>
```

`let { name, age } = $props()` declares the props this component accepts by destructuring them from the `$props()` rune. The destructured variables are then used directly in the markup (`{name}`), and — because `$props` is reactive — they automatically update if the parent passes new values. This is strikingly close to React's `function UserCard({ name, age })` destructuring, just pulled from a rune instead of function parameters.

Passing Props from a Parent

```
<!-- parent -->
<script>
  import UserCard from './UserCard.svelte';
</script>

<UserCard name="Philip" age={35} />
<UserCard name="Sam" age={28} />
```

Import the child, then place its tag with props as attributes. The familiar string-vs-expression rule applies, exactly as in JSX: `name="Philip"` passes the literal string, but `age={35}` (with braces) passes the real number. This is identical to React's quoting rule and the same idea as Vue's `:age` vs `age` — Svelte just uses JSX-style braces rather than a binding prefix.

Default Values

```
let { name, age = 0, isAdmin = false } = $props();
```

Because props are destructured, plain JavaScript default-value syntax gives a fallback when the parent omits a prop — `age = 0`, `isAdmin = false`. No special "default" option object like Vue's or Angular's; it's just destructuring defaults, the same as React's. Clean and familiar.

Rest Props and Spreading

```
<!-- collect any extra props into `rest` -->
let { label, ...rest } = $props();
```

```
<!-- forward them onto a real element -->
<button {...rest}>{label}</button>
```

The rest pattern (`...rest`) collects any props you didn't name explicitly, and `{...rest}` spreads them onto an element — the standard way to build a wrapper component that forwards arbitrary attributes (e.g. a custom `<Button>` that still accepts `disabled`, `type`, etc.). This is exactly React's `{...rest}` spread, working the same way.

> **ADDING TYPESCRIPT TYPES IS STRAIGHTFORWARD**
>
> With `<script lang="ts">`, props get typed by annotating the destructure: `let { name, age = 0 }: { name: string; age?: number } = $props()`. This gives the same compile-time prop checking that Vue's typed `defineProps` and Angular's typed `@Input` provide — and that React needs PropTypes or TS for. The course stays in plain JS, but the TS path is a one-line annotation away.

Passing Objects and Arrays

```
<ProductCard
  title="Keyboard"           <!-- string -->
  price={49.99}              <!-- number -->
  inStock={true}             <!-- boolean -->
  tags={['new', 'sale']}     <!-- array -->
/>
```

Anything that isn't a plain string goes in braces — numbers, booleans, arrays, objects, functions. Note Svelte uses the JS-native `inStock` (camelCase) attribute name directly, with no kebab-case conversion (unlike Vue's `:in-stock` convention) — props are passed exactly as named.

PROPS ARE READ-ONLY — DON'T REASSIGN THEM

A child should treat props as read-only and never reassign a destructured prop — Svelte warns if you do. Props flow one way, parent to child, as in every framework. If a child needs a local editable copy, derive it (`let local = $derived(name)` , or a separate `$state` initialized from the prop); to change the parent's data, use a callback prop or event — next chapter. (There's a `$bindable()` rune for genuine two-way prop binding, covered alongside component `bind:` in Chapter 7.)

SVELTE	REACT	VUE	ANGULAR
`let { name } = $props()`	params `{ name }`	`defineProps(['name'])`	`@Input() name`
`name="x" (string)`	`name="x"`	`name="x"`	`name="x"`
`age={35} (real value)`	`age={35}`	`:age="35"`	`[age]="35"`
`{ age = 0 } default`	destructure default	`{ default: 0 }`	typed input + default
`...rest + {...rest}`	`...rest` spread	`v-bind="$attrs"`	—

Coding Challenges

CHALLENGE 1

Build a MovieCard component with title and year props (via $props), rendering "Title (Year)". Use it three times in a parent with three different movies, passing year as a real number.

View solution

CHALLENGE 2

Build a Badge component with a text prop and a color prop defaulting to "gray". Render it once with a custom color and once without, confirming the default applies.

View solution

CHALLENGE 3

Build a Button wrapper component that takes a label prop and collects all other props with ...rest, spreading them onto a real button element — then use it passing label plus extra attributes like disabled and a title.

View solution

CHAPTER 6 QUICK REFERENCE

- **let { name, age } = $props()** — declares props by destructuring the rune (= React params)
- Used directly in markup; reactive — updates when the parent passes new values
- **name="x"** passes a string; **prop={expr}** passes a real value (braces, like JSX)
- Defaults are plain destructuring defaults: `{ age = 0 }`
- **...rest** collects extra props; **{...rest}** forwards them onto an element
- Props are **read-only** — change parent data via a callback/event (next chapter)
- **Next chapter:** component communication — callback props and events

CHAPTER 7 OF 12

Component Communication

CHAPTER 7

Component Communication

Callback props for child-to-parent, and $bindable for two-way component binding

Props (Chapter 6) send data *down*. To send something back *up* — a click, a deletion, a chosen value — Svelte 5's primary approach is a **callback prop**: the parent passes a function down as a prop, and the child calls it. This is exactly React's model (Project 1's todo delete), and a notable shift from Svelte 4's event system.

Callback Props — Child Calls a Function

```
<!-- TodoItem.svelte -->
<script>
  let { id, text, onDelete } = $props();
</script>

<li>
  {text}
  <button onclick={() => onDelete(id)}>Delete</button>
</li>
```

```
<!-- parent -->
<TodoItem id={todo.id} text={todo.text} onDelete={removeTodo} />
```

`onDelete` is just another prop — a function. The child calls `onDelete(id)` when its button is clicked, and the parent's `removeTodo` receives the `id`. There's no special "emit" mechanism (unlike Angular's `@Output` or Vue's `defineEmits`) — it's a plain function passed as a prop, identical to how React does it. Naming it `onSomething` is the convention, matching the event-handler naming.

SVELTE 4'S CREATEEVENTDISPATCHER IS GONE IN V5

Pre-Svelte-5 code used `createEventDispatcher()` plus `dispatch('delete', id)` in the child and `on:delete={handler}` in the parent — a custom-event system much like Angular's `@Output`. Svelte 5 **removed** this in favor of callback props, simplifying the model to match React's. If you see `createEventDispatcher` or `on:customEvent` on a component, that's Svelte 4 code.

Multiple Callbacks

```
<!-- Dialog.svelte -->
let { onSave, onCancel } = $props();
```

```
<button onclick={() => onSave(formData)}>Save</button>
<button onclick={onCancel}>Cancel</button>
```

A component takes as many callback props as it needs, each a separate function. This scales cleanly — a dialog with `onSave` and `onCancel`, a form with `onSubmit` and `onReset` — all just functions passed down, with payloads passed as arguments when calling them.

$bindable — Two-Way Component Binding

Sometimes you want genuine two-way binding on a custom component — like a reusable input where the parent uses `bind:value` directly on it. The `$bindable()` rune marks a prop as bindable from the parent:

```
<!-- CustomInput.svelte -->
<script>
  let { value = $bindable('') } = $props();
</script>

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

```
<!-- parent: bind: works directly on the component -->
<CustomInput bind:value={name} />
```

`$bindable('')` declares `value` as a prop the parent can two-way bind (with a default of `''`). The child binds it to its real input; the parent uses `bind:value={name}` on the component itself, and changes flow both ways automatically. This is Svelte's equivalent of Vue's `defineModel` / `v-model` on a component, and Angular's `value` / `valueChange` two-way convention.

CALLBACK PROPS VS $BINDABLE — WHICH TO USE

Reach for a **callback prop** when the child reports an event the parent handles however it likes (delete, save, select) — the common case. Reach for **$bindable** only when you specifically want `bind:` ergonomics on a component, like a reusable form control. Most communication is callback props; `$bindable` is the more specialized tool, and Svelte deliberately requires opting in (a prop isn't bindable unless declared so).

A CHILD STILL DOESN'T REASSIGN A NORMAL PROP

Outside of an explicit `$bindable`, the one-way rule from Chapter 6 holds: the child reports upward via a callback and lets the parent own the source of truth. `$bindable` is the deliberate exception, used sparingly — not a license to mutate ordinary props.

SVELTE 5	REACT	VUE	ANGULAR
`onDelete prop (a function)`	callback prop `onDelete`	`defineEmits` + `emit`	`@Output` + `emit`
`onDelete(id)`	`onDelete(id)`	`emit('delete', id)`	`this.delete.emit(id)`
`$bindable + bind:value`	value + onChange props	`defineModel` / `v-model`	`[(value)]`

Coding Challenges

CHALLENGE 1

Build a TodoItem component with id/text props and an onDelete callback prop. The parent holds a $state array of todos, renders one TodoItem each (keyed), and removes the matching todo when onDelete fires.

View solution

CHALLENGE 2

Build a ConfirmBar component with onConfirm and onCancel callback props (two buttons), and use it in a parent that logs which action was taken.

View solution

CHALLENGE 3

Build a CustomInput component exposing a $bindable value prop, then use bind:value on it in a parent and display the bound value live.

View solution

CHAPTER 7 QUICK REFERENCE

- **Callback prop** — pass a function down (`onDelete`); the child calls it (= React's model)
- No special emit mechanism — it's a plain function prop, unlike Angular's `@Output` / Vue's `defineEmits`
- Name callbacks `onSomething` ; pass payloads as arguments when calling
- **$bindable(default)** — marks a prop as two-way bindable, so the parent can use `bind:` on the component

- Use callbacks for events (the common case); `$bindable` only for genuine two-way control bindings
- Svelte 5 dropped Svelte 4's `createEventDispatcher` in favor of callback props
- **Next chapter:** snippets and slots — Svelte's content-projection model

CHAPTER 8 OF 12

Snippets and Content

CHAPTER 8

Snippets and Content

Passing markup into a child — children, named snippets, and parameterised snippets

Props pass *data* into a child; **content** passes *markup*. Svelte 5's mechanism is the `children` snippet plus the `{#snippet}` block — its replacement for the old `<slot>` system, covering React's `children`, Vue's slots, and Angular's `<ng-content>` all at once. It's the most distinctive of the four frameworks' approaches, and worth taking slowly.

The Default children Snippet

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

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

```
<!-- parent -->
<Card>
  <h2>Hello!</h2>
  <p>Markup passed in from outside.</p>
</Card>
```

Whatever a parent writes between a component's tags becomes a special prop called `children` — a **snippet** (a reusable chunk of markup). The child renders it with `{@render children()}`. So `children` is React's `{children}`, but instead of just embedding it, you *call* it via `{@render}` — because a snippet is technically a renderable function. `Card` doesn't know what's inside; it just renders whatever it's handed.

{@RENDER} AND {#SNIPPET} ARE THE TWO HALVES

> Svelte 5 replaced slots with two pieces: `{#snippet name()}...{/snippet}` *defines* a chunk of markup, and `{@render name()}` *renders* one. The default `children` snippet (the content between tags) is just the most common case. Everything in this chapter is built from these two primitives.

Named Snippets — Multiple Insertion Points

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

<div class="panel">
  <header>{@render header()}</header>
  <main>{@render children()}</main>
  <footer>{@render footer()}</footer>
</div>
```

```
<!-- parent: define named snippets inside the component -->
<Panel>
  {#snippet header()}<h2>Title</h2>{/snippet}
  <p>Body (the default children).</p>
  {#snippet footer()}<small>Footer</small>{/snippet}
</Panel>
```

For several insertion points, the parent defines named snippets with `{#snippet header()}...{/snippet}` inside the component tags — and they arrive as props of those names. Anything *not* in a named snippet becomes the default `children`. This is the equivalent of Vue's named slots and React's named-JSX-props approach, expressed through the snippet system.

Fallback Content

```
{#if children}
  {@render children()}
{:else}
  <p>No content provided.</p>
{/if}
```

Because snippets are just props (a value, or `undefined` if not passed), fallback content is a plain `{#if}` check — render the snippet if it exists, otherwise show a default. More explicit than Vue's "put fallback inside the slot tag," but using only tools you already know.

Parameterised Snippets — Passing Data Back

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

<ul>
  {#each items as item (item.id)}
    <li>{@render row(item)}</li>   <!-- pass each item to the snippet -->
  {/each}
</ul>
```

```svelte
<!-- parent: the snippet receives the item -->
<List items={people}>
  {#snippet row(person)}
    <strong>{person.name}</strong>
  {/snippet}
</List>
```

The most powerful case: a snippet can take **parameters**. The child calls `{@render row(item)}`, passing data *out* to the markup the parent provided; the parent's `{#snippet row(person)}` receives it. `List` owns the looping and data; the parent decides exactly how each row looks. This is the direct equivalent of React's render props and Vue's scoped slots — and notably, it's the *same* snippet syntax as the simpler cases, not a separate feature.

ONE MECHANISM FOR EVERYTHING REACT SPLIT THREE WAYS

React used `children` for simple content, named props for multi-region layouts, and render props for "pass data to the caller's markup." Svelte 5's snippets do all three with one consistent primitive — and snippets can even be defined and reused *within* a single component (a chunk of repeated markup rendered in several places), which the others can't do as cleanly.

{@RENDER}, NOT JUST EMBEDDING — AND <SLOT> IS GONE

Two things to watch coming from the old model or other frameworks: a snippet must be *rendered* with `{@render name()}` (with parentheses — it's a call), not embedded like a value. And Svelte 4's `<slot>` / `<slot name="x">` / `let:` system is **removed** in Svelte 5 — snippets replace all of it. If you see `<slot>` in a tutorial, that's pre-v5 code.

SVELTE 5	REACT	VUE	ANGULAR
children + {@render}	{children}	default <slot>	<ng-content>
named snippets	named JSX props	named slots	<ng-content select>

SVELTE 5	REACT	VUE	ANGULAR
`parameterised snippet`	render props	scoped slots	template context
`reuse a snippet in-component`	— (extract a component)	—	—

Coding Challenges

CHALLENGE 1

Build a Card component that renders its children snippet inside a styled div, then use it twice with completely different inner markup. Add a {#if children} fallback shown when no content is passed.

View solution

CHALLENGE 2

Build a Panel component with header and footer named snippets plus the default children for the body, and fill all three from a parent using {#snippet header()} / default content / {#snippet footer()}.

View solution

CHALLENGE 3

Build a List component that takes an items array and a row snippet, looping over the items and calling {@render row(item)} for each. In the parent, define {#snippet row(person)} to render each item your own way (e.g. bold name + muted id).

View solution

CHAPTER 8 QUICK REFERENCE

- **children prop + {@render children()}** — content between tags (= React's `{children}`)
- **{#snippet name()}...{/snippet}** defines markup; **{@render name()}** renders it
- **Named snippets** arrive as same-named props — for multi-region layouts (= named slots)
- **Fallback** — a plain `{#if children}...{:else}...{/if}`
- **Parameterised snippets** (`{@render row(item)}` → `{#snippet row(x)}`) = render props / scoped slots
- One mechanism covers children, named regions, and data-passing; Svelte 4's `<slot>` is removed

- **Next chapter:** lifecycle — onMount, onDestroy, and $effect for teardown

CHAPTER 9 OF 12

Lifecycle

CHAPTER 9

Lifecycle

onMount and onDestroy — and why $effect often replaces both

A component is created, mounted to the DOM, and eventually destroyed. Svelte provides lifecycle functions for the key moments — but with a twist: in Svelte 5, the `$effect` rune (Chapter 3) handles many cases that would be lifecycle hooks in other frameworks, because an effect with a cleanup return already covers "run on mount, clean up on destroy."

onMount — After the Component Is in the DOM

```
<script>
  import { onMount } from 'svelte';

  let users = $state([]);

  onMount(async () => {
    const res = await fetch('/api/users');
    users = await res.json();
  });
</script>
```

`onMount` runs a callback once, after the component is first inserted into the DOM — the standard place for an initial data fetch, the equivalent of React's `useEffect(() => {...}, [])`, Vue's `onMounted`, and Angular's `ngOnInit`. Unlike the others, `onMount` is **imported** from `'svelte'` (it's a function, not a rune). By the time it runs, the DOM exists, so it's also safe to measure or access elements here.

TOP-LEVEL SCRIPT CODE RUNS AT "CREATION" — BEFORE MOUNT

Much of what you'd put in `onMount` elsewhere can just live at the top of `<script>`: `$state`, `$derived`, and plain setup run once when the component is created, before it mounts. Reach for `onMount` specifically when you need the DOM to exist first (measuring an element, initializing a canvas, a third-party widget) or want to defer work until after the first render — and note `onMount` does **not** run during server-side rendering, which makes it the right place for browser-only code.

onDestroy — Cleanup Before Removal

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

let id;
onMount(() => {
  id = setInterval(() => console.log('tick'), 1000);
});

onDestroy(() => {
  clearInterval(id);   // stop the timer when the component is removed
});
```

`onDestroy` runs just before the component is removed — the place for cleanup, mirroring Vue's `onUnmounted` and Angular's `ngOnDestroy`. The universal rule applies: anything that "starts" something ongoing (a timer, a listener, a subscription) needs a matching "stop" here, or it leaks.

The Cleaner Way — onMount Returning Cleanup

```
onMount(() => {
  const id = setInterval(() => console.log('tick'), 1000);

  return () => clearInterval(id);   // returned function runs on destroy
});
```

A neat shortcut: if `onMount` *returns a function*, Svelte calls it on destroy automatically — so setup and teardown live together in one place, without a separate `onDestroy`. This is exactly React's `useEffect` cleanup-return pattern, and it keeps the timer's start and stop side by side.

$effect Often Replaces Lifecycle Entirely

```
<!-- no onMount/onDestroy needed -->
$effect(() => {
  const id = setInterval(() => console.log('tick'), 1000);
  return () => clearInterval(id);
});
```

Because `$effect` (Chapter 3) runs after mount and its returned cleanup runs on destroy, an effect frequently does the job of `onMount` + `onDestroy` together — and re-runs if its dependencies change, which a one-time `onMount` doesn't. The practical guidance for Svelte 5: **use `$effect` for reactive setup/teardown** (anything

that should respond to changing state); use `onMount` for one-time, mount-only, browser-only setup (DOM measurement, SSR-sensitive code). You'll reach for `onDestroy` on its own fairly rarely now.

SVELTE 4 HAD MORE LIFECYCLE HOOKS — MOST ARE NOW $EFFECT

Svelte 4 also exposed `beforeUpdate` and `afterUpdate` hooks (run around each re-render). Svelte 5 deprecated them in favor of `$effect` (and `$effect.pre` for "before DOM update"), since reactive effects express the same intent more precisely. `onMount` and `onDestroy` remain; `beforeUpdate` / `afterUpdate` are the ones to replace with effects in modern code.

SVELTE 5	REACT	VUE	ANGULAR
top of <script>	component body	top of `<script setup>`	constructor
onMount	`useEffect(.., [])`	`onMounted`	`ngOnInit`
onDestroy / onMount cleanup return	cleanup return	`onUnmounted`	`ngOnDestroy`
$effect (+ cleanup)	`useEffect` (reactive)	`watchEffect`	`effect()`

Coding Challenges

CHALLENGE 1

Build a component that fetches a list from any free public API in onMount, storing it in a $state for display, with a loading state shown until the data arrives.

View solution

CHALLENGE 2

Build a Clock component that starts a setInterval in onMount (updating a time $state every second) and clears it by returning a cleanup function from onMount. Toggle the component with {#if} in a parent to confirm the ticking stops when removed.

View solution

CHALLENGE 3

Build a component that tracks the window's width using a resize listener — implemented two ways for comparison: once with onMount + onDestroy, and once with a single $effect that returns its cleanup. Display the live width.

View solution

CHAPTER 9 QUICK REFERENCE

- Top-of-`<script>` code runs at **creation**, before mount
- **onMount** (imported from `'svelte'`) — runs once after mount; browser-only, skipped during SSR
- **onDestroy** — runs before removal; or return a cleanup function from `onMount` instead
- **$effect** with a cleanup return often replaces `onMount` + `onDestroy`, and re-runs reactively
- Guidance: `$effect` for reactive setup/teardown; `onMount` for one-time, mount-only, browser-only work
- Svelte 4's `beforeUpdate` / `afterUpdate` are replaced by `$effect` / `$effect.pre`
- **Next chapter:** shared reactive logic in `.svelte.js` modules

CHAPTER 10 OF 12

Shared Reactive Logic

CHAPTER 10

Shared Reactive Logic

Moving runes into .svelte.js modules — Svelte's composable and store story in one

So far, runes have lived inside `.svelte` components. Svelte 5 lets you use them in plain JavaScript modules too — files named `.svelte.js` (or `.svelte.ts`) — which is how you share reactive logic and state across components. This single mechanism covers what was two separate things in the other frameworks: reusable logic (React's custom hooks, Vue's composables) *and* global state (Svelte 4's stores, Vue's Pinia).

The .svelte.js Module

A regular `.js` file can't use runes — they're only enabled in files with the `.svelte.js` extension. That naming tells the compiler "process runes in here too." Once you've done that, `$state`, `$derived`, and `$effect` all work exactly as they do in a component.

A Reusable Factory — Like a Composable

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

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

```
<!-- usage in a component -->
<script>
  import { createCounter } from './counter.svelte.js';
  const counter = createCounter();
</script>

<p>{counter.count}</p>
<button onclick={counter.increment}>+1</button>
```

A factory function returning reactive state plus its behavior is Svelte's equivalent of a React custom hook or a Vue composable — call it in a component and each call gets its own independent state. The notable detail: to expose the reactive `count` while keeping it readable from outside, the returned object uses a **getter** (`get count()`). That's because a destructured `$state` value would lose its reactive connection — the getter re-reads the live value each access. (Vue's composables solve the same problem by returning the `ref` itself.)

> **THE GETTER PATTERN IS THE ONE WRINKLE TO LEARN**
>
> Reactive values can't be "handed out" by plain return — pass the *object* that owns them and access through it (`counter.count`), or expose a `get` accessor as above. This is the Svelte equivalent of Vue's "return the ref, not `ref.value`" rule and React's "return state, not a snapshot." Same underlying concern, slightly different shape.

Shared Global State — A Single Instance

```
// cart.svelte.js
export const cart = $state({ items: [] });

export function addToCart(product) {
  const existing = cart.items.find((i) => i.id === product.id);
  if (existing) existing.quantity++;
  else cart.items.push({ ...product, quantity: 1 });
}
```

```
<!-- any component imports the same shared cart -->
<script>
  import { cart, addToCart } from './cart.svelte.js';
</script>

<p>Items: {cart.items.length}</p>
```

For genuinely global state, export a `$state` object at module level — there's only one instance, so every component importing it shares the same reactive data. This is Svelte's whole answer to global state management: no Pinia, no Redux, no Context provider — just an exported reactive object. Because objects are

deeply reactive (Chapter 2), mutating `cart.items.push(...)` updates every component using it. This is dramatically lighter than the equivalent in any of the other three.

EXPORT THE OBJECT, NOT A BARE REACTIVE VARIABLE

Export a reactive *object* (`export const cart = $state({...})`) and access its properties, rather than exporting a bare primitive. A directly-exported reactive primitive can't keep its reactive binding across the module boundary — the same getter/object-access concern from earlier. Wrapping shared state in an object (or using a getter) is the reliable pattern.

Svelte Stores Still Exist (and the contrast)

```
// the older store API — still supported
import { writable } from 'svelte/store';
export const count = writable(0);
// in a component, read with the $ prefix: {$count}
```

Svelte 4's **stores** (`writable` / `readable`, accessed in components with a `$` prefix like `{$count}`) still work in Svelte 5 and remain useful — especially for RxJS-style reactive streams. But for most shared state, the rune-based `.svelte.js` approach above is now the recommended default: it's the same mental model as in-component reactivity, with no separate store API to learn. Recognize `writable` and the `$store` syntax in existing code; reach for `$state` modules in new code.

NEED	SVELTE 5	VUE	REACT
Reusable stateful logic	factory in `.svelte.js`	composable	custom hook
Global shared state	exported `$state` object	Pinia store	Zustand / Context
Each call = own state	factory returns new state	composable	hook
Reactive streams (legacy)	`svelte/store` (`$store`)	—	—

Coding Challenges

CHALLENGE 1

Write a createCounter factory in a counter.svelte.js module returning { count (getter), increment, decrement, reset }, and use it in two separate components — confirming each gets its own independent count.

View solution

CHALLENGE 2

Build a shared cart in a cart.svelte.js module: an exported $state object with items, plus an addToCart function with the duplicate-quantity check. Use it from a product list and a separate cart-display component, confirming both reflect the same shared state.

View solution

CHALLENGE 3

Write a createLocalStore(key, initial) factory in a .svelte.js module that returns a getter/setter-style reactive value initialized from localStorage and persisted via an $effect on change, then use it to persist a counter so it survives a refresh.

View solution

CHAPTER 10 QUICK REFERENCE

- **.svelte.js** (or `.svelte.ts`) — modules where runes work outside components
- **Factory function** returning reactive state + behavior = a composable / custom hook; each call is independent
- Expose reactive values via a **getter** (`get count()`) or by returning the owning object — never a bare value
- **Exported** `$state` **object** = global shared state — one instance, no Pinia/Redux/Context needed
- Deep reactivity means mutating the shared object (`.push()`) updates every consumer
- Svelte 4 **stores** (`writable`, `$store`) still work; rune modules are the modern default
- **Next chapter:** SvelteKit routing (the meta-framework layer)

CHAPTER 11 OF 12

SvelteKit Routing

CHAPTER 11

SvelteKit Routing

File-based routing — the folder structure IS the route map

Routing in the Svelte world is provided by **SvelteKit**, its official meta-framework — the equivalent of Next.js for React or Nuxt for Vue (and parallel to the SSR that Angular ships built-in). The standout feature: routing is **file-based**. You don't write a route config array (React Router, Vue Router) — instead, the folder structure under `src/routes` *is* the route map. This is the same model as Next.js's App Router, which Project 7 of the React course touched on.

STARTING A SVELTEKIT PROJECT

The bare Vite template used so far is component-only. A SvelteKit app is created with `npx sv create my-app` (choosing the SvelteKit option), which scaffolds the `src/routes` directory and the dev server. Everything you've learned about Svelte components applies unchanged — SvelteKit just adds the routing, data-loading, and SSR layer around them.

Pages Are +page.svelte Files

```
src/routes/
├── +page.svelte            // the "/" route
├── about/
│   └── +page.svelte        // the "/about" route
└── contact/
    └── +page.svelte        // the "/contact" route
```

Each folder under `src/routes` is a URL segment, and a `+page.svelte` file inside it is the page rendered for that URL. The root `+page.svelte` is `/`, `about/+page.svelte` is `/about`, and so on. The `+` prefix marks SvelteKit's special files (distinguishing them from your own components). No route config to maintain — adding a page is creating a file.

Layouts — Shared UI Across Routes

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

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

{@render children()}   <!-- the current page renders here -->
```

A `+layout.svelte` file wraps every page in its folder (and subfolders) — perfect for a shared nav, header, or footer. The current page renders where you call `{@render children()}` (the snippet mechanism from Chapter 8). This is SvelteKit's equivalent of a layout route with `<Outlet>` (React Router / Vue Router's `<RouterView>`), but defined by file convention rather than configuration.

Navigation — Just Anchor Tags

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

A genuinely nice surprise: SvelteKit navigates with **plain `<a>` tags**. There's no `<Link>` (React Router) or `<RouterLink>` (Vue) component — SvelteKit intercepts ordinary anchor clicks and turns them into client-side navigation automatically (no full page reload), while still working as a real link if JS is unavailable. The warning every other framework needed ("don't use a plain `<a>`!") is simply inverted here: plain anchors are the *correct* way.

Dynamic Routes — [param] Folders

```
src/routes/
└── product/
    └── [id]/
        └── +page.svelte        // matches /product/anything
```

A folder named in square brackets — `[id]` — is a dynamic segment, matching any value in that URL position (`/product/1`, `/product/42`). The param is read inside the page from the `page` state:

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

  const id = $derived(page.params.id);
</script>

<p>Showing product {id}</p>
```

`page.params.id` reads the `[id]` segment — the equivalent of React Router's `useParams()`, Vue Router's `route.params`, and Angular's `ActivatedRoute.paramMap`. Wrapping it in `$derived` keeps it reactive, so navigating from `/product/1` to `/product/2` updates it. (The `$app/state` import is SvelteKit-provided.)

Programmatic Navigation

```
import { goto } from '$app/navigation';

function openProduct(id) {
  goto(`/product/${id}`);
}
```

`goto(url)` navigates from code — after a form submits, a login succeeds, an action completes — the equivalent of React Router's `useNavigate()`, Vue's `router.push()`, and Angular's `Router.navigate()`. Imported from `$app/navigation`.

OTHER SPECIAL ROUTE FILES EXIST — AND A CATCH-ALL

SvelteKit has more `+` files than just `+page.svelte`: `+layout.svelte` (covered), `+error.svelte` (an error boundary for a route), and `+page.js` / `+page.server.js` for data loading (next chapter). A catch-all dynamic segment uses `[...rest]` (e.g. `[...path]/+page.svelte`) — the equivalent of a wildcard route. The file-based system trades a config array for a set of naming conventions to learn.

SVELTEKIT	REACT ROUTER	VUE ROUTER
folder + +page.svelte	routes config / file-based (Next)	routes array
+layout.svelte + {@render children()}	`<Outlet />`	`<RouterView />`
<a href> (auto-intercepted)	`<Link to>`	`<RouterLink to>`
[id] folder + page.params	`:id` + `useParams()`	`:id` + `route.params`
goto()	`useNavigate()`	`router.push()`

Coding Challenges

CHALLENGE 1

Set up a SvelteKit app with three pages — Home (/), About (/about), Contact (/contact) — as +page.svelte files, plus a +layout.svelte with a nav (plain anchor tags) wrapping all of them via {@render children()}.

View solution

CHALLENGE 2

Add a dynamic product/[id] route. From a product list page, link to /product/[id] for each product with plain anchors, and in the [id] page read page.params.id (via $derived) to display the matching product.

View solution

CHALLENGE 3

Build a page with a button that uses goto() to navigate programmatically to another route (e.g. after a simulated action), confirming it does client-side navigation without a full reload.

View solution

CHAPTER 11 QUICK REFERENCE

- **SvelteKit** — Svelte's meta-framework (routing, SSR, data loading); `npx sv create`
- **File-based routing** — folders under `src/routes` are URL segments; `+page.svelte` is the page
- **+layout.svelte** + `{@render children()}` — shared UI across routes (= `<Outlet>` / `<RouterView>`)
- Navigate with **plain** `<a href>` — auto-intercepted; no `<Link>` / `<RouterLink>` component
- **[id] folder** — dynamic segment; read with `page.params.id` from `$app/state` (wrap in `$derived`)
- **goto(url)** from `$app/navigation` — programmatic navigation
- Other `+` files: `+error.svelte`, `+page.js` (next chapter); `[...rest]` is the catch-all
- **Next chapter:** data loading and global state patterns (the final chapter)

CHAPTER 12 OF 12

SvelteKit Data Loading & Final Patterns

CHAPTER 12

Data Loading & Final Patterns

load functions, form actions, SSR, and deploying a SvelteKit app

The final chapter covers how SvelteKit gets data *into* pages and data back *out* via forms — the pieces that make it a full-stack framework rather than just a router. This is where SvelteKit's server-rendering story (its big advantage over the plain Vite SPA from Chapter 1) really shows, and where it lines up directly against Next.js and Nuxt.

The load Function — Data Before the Page Renders

```
src/routes/blog/
├── +page.svelte        // the page (UI)
└── +page.js            // the load function (data)
```

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

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

<ul>
  {#each data.posts as post (post.id)}
    <li>{post.title}</li>
  {/each}
</ul>
```

A `+page.js` beside a page exports a `load` function that runs *before* the page renders; whatever it returns arrives as the page's `data` prop. This is a real shift from the Chapter 9 pattern of fetching in `onMount` — `load` runs ahead of render (on the server for the first load), so the page arrives with its data already present, no loading flash, and it's SEO-friendly. It's the direct equivalent of Next.js server components / `getServerSideProps` and Nuxt's `useAsyncData`.

> **+PAGE.JS VS +PAGE.SERVER.JS**
> A `load` in `+page.js` can run on both server and client (a "universal" load — use it for public API calls). A `load` in `+page.server.js` runs **only on the server** — the place for database queries, secret API keys, and anything that must never reach the browser. Choosing the right file is how you keep secrets server-side: code in `.server.js` is never bundled to the client.

Dynamic Routes Get Their Param in load

```
// src/routes/product/[id]/+page.js
export async function load({ params, fetch }) {
  const res = await fetch(`/api/products/${params.id}`);
  return { product: await res.json() };
}
```

The `load` function receives `params`, so a dynamic `[id]` route (Chapter 11) fetches exactly the right record server-side — `params.id` here is the same value `page.params.id` gave you in the component, but available before render so the data ships with the page.

Form Actions — Data Back to the Server

```
// src/routes/contact/+page.server.js
export const actions = {
  default: async ({ request }) => {
    const data = await request.formData();
    const email = data.get('email');
    // ...save it server-side...
    return { success: true };
  },
};
```

```
<!-- +page.svelte: a real form that POSTs to the action -->
<form method="POST">
  <input name="email" type="email" />
  <button>Submit</button>
</form>
```

Form actions handle submissions on the server. A `+page.server.js` exports an `actions` object, and a standard HTML `<form method="POST">` posts to it — no `onsubmit` handler, no manual `fetch`, and crucially **it works without JavaScript** (progressive enhancement). This is SvelteKit's signature data-out pattern, paralleling Next.js Server Actions and Remix's form actions — the framework leaning into web-standard forms rather than client-side-only handlers.

> **USE:ENHANCE — PROGRESSIVE ENHANCEMENT, UPGRADED**
>
> Adding the `use:enhance` action to the form (`<form method="POST" use:enhance>`, imported from `$app/forms`) keeps the no-JS behavior but, when JS *is* available, submits via fetch with no full-page reload and updates the page in place. You get the resilient baseline and the SPA-smooth experience from the same markup — the best of both, which is hard to achieve in a purely client-side framework.

SSR, CSR, and Prerendering

SvelteKit renders on the **server** by default (SSR) — the first page arrives as real HTML (fast first paint, SEO-friendly), then "hydrates" into an interactive client app. You can tune this per route with page options: `export const prerender = true` turns a route into a static file at build time (ideal for content that rarely changes — a marketing or docs page), and `export const ssr = false` makes a route client-only (for something that can't run on the server). This per-route control over SSR / static / client-only is exactly the spectrum Next.js and Nuxt offer.

Deployment — Adapters

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

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

SvelteKit deploys via **adapters** — small plugins that package the build for a target platform. `adapter-auto` detects common hosts (Vercel, Netlify, Cloudflare); there's `adapter-node` for a plain Node server and `adapter-static` for a fully static site. You write the same app and swap the adapter for the destination — the same "deploy anywhere" promise as Nuxt's presets and Next.js's hosting flexibility.

DON'T FETCH IN ONMOUNT WHEN LOAD WILL DO — AND WATCH THE SERVER/CLIENT LINE

Two habits to carry forward. First: in a SvelteKit app, prefer a `load` function over an `onMount` fetch for a page's primary data — you get SSR, no loading flash, and better SEO. Reserve `onMount` fetching for genuinely client-only, after-render needs. Second: be deliberate about the `+page.js` vs `+page.server.js` split — secrets, database access, and private keys belong in `.server.js` so they never ship to the browser.

SVELTEKIT	NEXT.JS	NUXT
`load in +page.js / +page.server.js`	server components / `getServerSideProps`	`useAsyncData` / `useFetch`
`form actions + use:enhance`	Server Actions	server routes + `useFetch`
`SSR default + prerender / ssr opts`	SSG / SSR / RSC per route	SSR / SSG / hybrid
`adapters (auto / node / static)`	built-in + hosting targets	nitro presets

Coding Challenges

CHALLENGE 1

Build a /posts route with a +page.js load function fetching from any free public API and returning { posts }, and a +page.svelte that reads data via $props and lists them — with no onMount and no loading state needed.

View solution

CHALLENGE 2

Build a /posts/[id] route whose +page.js load uses params.id to fetch a single post and return it, with the +page.svelte displaying the post's title and body from data.

View solution

CHALLENGE 3

Build a /contact route with a +page.server.js exposing a default form action that reads email/message from formData and returns { success: true }, plus a +page.svelte with a method="POST" form using use:enhance, showing a thank-you message from the action's result.

View solution

CHAPTER 12 QUICK REFERENCE

- **load** in `+page.js` — runs before render; its return becomes the page's `data` prop (= `getServerSideProps`)
- **+page.js** = universal (server + client); **+page.server.js** = server-only (secrets, DB)
- `load({ params })` — dynamic-route data fetched server-side before render
- **Form actions** (`actions` in `+page.server.js` + `<form method="POST">`) — work without JS
- **use:enhance** — keeps no-JS fallback, adds fetch-based no-reload submit when JS is present
- **SSR by default**; per-route `prerender` / `ssr` options; deploy via **adapters**
- Prefer `load` over `onMount` for a page's primary data (SSR, no flash, SEO)

★ Svelte Course Complete — 12 / 12 chapters

From the compiler model and runes through components, snippets, shared reactive modules, and the full SvelteKit stack. You now have the same conceptual map across React, Angular, Vue, and Svelte — and can read the differences as variations on shared ideas rather than four separate worlds.