SVELTE Docs - English

Table of contents

Overview

- Short intro to what Svelte is and why it's the best ever
- A few code examples to have a very rough understanding of how Svelte code looks like
- Jump off points to tutorial, SvelteKit etc

Svelte is a web UI framework that uses a compiler to turn declarative component code like this...

```
<! file: App.svelte --->
<script lang="ts">
    let count = $state(0);

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

<button onclick={increment}>
    clicks: {count}
</button>
```

...into tightly optimized JavaScript that updates the document when state like count changes. Because the compiler can 'see' where count is referenced, the generated code is highly efficient, and because we're hijacking syntax like `$state(...)` and `=` instead of using cumbersome APIs, you can write less code.

Besides being fun to work with, Svelte offers a lot of features built-in, such as animations and transitions. Once you've written your first components you can reach for our batteries included metaframework SvelteKit which provides you with an opinionated router, data loading and more.

If you're new to Svelte, visit the interactive tutorial before consulting this documentation. You can try Svelte online using the REPL. Alternatively, if you'd like a more fully-featured environment, you can try Svelte on StackBlitz.

Go to TOC

Getting started

- `npm create svelte@latest`, describe that it scaffolds SvelteKit project
- `npm create vite@latest`, describe that it scaffolds Svelte SPA powered by Vite
- mention `svelte-add`
- Jump off points to tutorial, SvelteKit etc

Start a new project

We recommend using SvelteKit, the official application framework from the Svelte team:

```
npm create svelte@latest myapp
cd myapp
npm install
npm run dev
```

SvelteKit will handle calling the Svelte compiler to convert your `.svelte` files into `.js` files that create the DOM and `.css` files that style it. It also provides all the other pieces you need to build a web application such as a development server, routing, deployment, and SSR support. SvelteKit uses Vite to build your code.

Don't worry if you don't know Svelte yet! You can ignore all the nice features SvelteKit brings on top for now and dive into it later.

Alternatives to SvelteKit

If you don't want to use SvelteKit for some reason, you can also use Svelte with Vite (but without SvelteKit) by running `npm create vite@latest` and selecting the `svelte` option. With this, `npm run build` will generate HTML, JS and CSS files inside the `dist` directory thanks using vite-plugin-svelte. In most cases, you will probably need to choose a routing library as well.

Alternatively, there are plugins for Rollup, Webpack and a few others to handle Svelte compilation — which will output `.js` and `.css` that you can insert into your HTML — but setting up SSR with them requires more manual work.

Editor tooling

The Svelte team maintains a VS Code extension and there are integrations with various other editors and tools as well.

You can also check your code from the command line using svelte-check (using the Svelte or Vite CLI setup will install this for you).

Getting help

Don't be shy about asking for help in the Discord chatroom! You can also find answers on Stack Overflow.

Go to TOC

Reactivity fundamentals

Reactivity is at the heart of interactive UIs. When you click a button, you expect some kind of response. It's your job as a developer to make this happen. It's Svelte's job to make your job as intuitive as possible, by providing a good API to express reactive systems.

Runes

Svelte 5 uses *runes*, a powerful set of primitives for controlling reactivity inside your Svelte components and inside `.svelte.js` and `.svelte.ts` modules.

Runes are function-like symbols that provide instructions to the Svelte compiler. You don't need to import them from anywhere — when you use Svelte, they're part of the language.

The following sections introduce the most important runes for declare state, derived state and side effects at a high level. For more details refer to the later sections on state and side effects.

`$state`

Reactive state is declared with the `$state` rune:

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

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

You can also use `$state` in class fields (whether public or private):

```
// @errors: 7006 2554
class Todo {
    done = $state(false);
    text = $state();

    constructor(text) {
        this.text = text;
    }
}
```

`$derived`

Derived state is declared with the `$derived` rune:

```
<script>
    let count = $state(0);
    let doubled = $derived(count * 2);
</script>
```

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

<p>{count} doubled is {doubled}</p>
```

The expression inside `$derived(...)` should be free of side-effects. Svelte will disallow state changes (e.g. `count++`) inside derived expressions.

As with `$state`, you can mark class fields as `$derived`.

`$effect`

To run *side-effects* when the component is mounted to the DOM, and when values change, we can use the `$effect` rune (demo):

```
<script>
    let size = $state(50);
    let color = $state('#ff3e00');

    let canvas;

    $effect(() => {
        const context = canvas.getContext('2d');
        context.clearRect(0, 0, canvas.width, canvas.height);

        // this will re-run whenever `color` or `size` change
        context.fillStyle = color;
        context.fillRect(0, 0, size, size);
    });
</script>

<canvas bind:this={canvas} width="100" height="100" />
```

The function passed to `$effect` will run when the component mounts, and will re-run after any changes to the values it reads that were declared with `$state` or `$derived` (including those passed in with `$props`). Re-runs are batched (i.e. changing `color` and `size` in the same moment won't cause two separate runs), and happen after any DOM updates have been applied.

Go to TOC

Introduction

Go to TOC

Component fundamentals

- script (module) / template / style (rough overview)
- `$props` / `$state` (in the context of components)

Components are the building blocks of Svelte applications. They are written into `.svelte` files, using a superset of HTML.

All three sections — script, styles and markup — are optional.

```
<script>
    // logic goes here
</script>

<!-- markup (zero or more items) goes here -->

<style>
    /* styles go here */
</style>
```

<script>

A `<script>` block contains JavaScript (or TypeScript, when adding the `lang="ts"` attribute) that runs when a component instance is created. Variables declared (or imported) at the top level are 'visible' from the component's markup.

Public API of a component

Svelte uses the `$props` rune to declare *properties* or *props*, which means describing the public interface of the component which becomes accessible to consumers of the component.

> `$props` is one of several runes, which are special hints for Svelte's compiler to make things reactive.

```
<script>
    let { foo, bar, baz } = $props();

    // Values that are passed in as props
    // are immediately available
    console.log({ foo, bar, baz });
</script>
```

You can specify a fallback value for a prop. It will be used if the component's consumer doesn't specify the prop on the component when instantiating the component, or if the passed value is `undefined` at some point.

```
<script>
    let { foo = 'optional default initial value' } = $props();
</script>
```

To get all properties, use rest syntax:

```
<script>
    let { a, b, c, ...everythingElse } = $props();
</script>
```

You can use reserved words as prop names.

```
<script>
    // creates a `class` property, even
    // though it is a reserved word
    let { class: className } = $props();
</script>
```

If you're using TypeScript, you can declare the prop types:

```
<script lang="ts">
    interface Props {
        required: string;
        optional?: number;
        [key: string]: unknown;
    }

    let { required, optional, ...everythingElse }: Props = $props();
</script>
```

If you're using JavaScript, you can declare the prop types using JSDoc:

```
<script>
    /** @type {{ x: string }} */
    let { x } = $props();

    // or use @typedef if you want to document the properties:

    /**
     * @typedef {Object} MyProps
     * @property {string} y Some documentation
     */

    /** @type {MyProps} */
    let { y } = $props();
</script>
```

If you export a `const`, `class` or `function`, it is readonly from outside the component.

```
<script>
    export const thisIs = 'readonly';

    export function greet(name) {
        alert(`hello ${name}!`);
    }
</script>
```

Readonly props can be accessed as properties on the element, tied to the component using `bind:this` syntax.

Reactive variables

To change component state and trigger a re-render, just assign to a locally declared variable that was declared using the `$state` rune.

Update expressions (`count += 1`) and property assignments (`obj.x = y`) have the same effect.

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

    function handleClick() {
        // calling this function will trigger an
        // update if the markup references `count`
        count = count + 1;
    }
</script>
```

Svelte's `<script>` blocks are run only when the component is created, so assignments within a `<script>` block are not automatically run again when a prop updates.

```
<script>
    let { person } = $props();
    // this will only set `name` on component creation
    // it will not update when `person` does
    let { name } = person;
</script>
```

If you'd like to react to changes to a prop, use the `$derived` or `$effect` runes instead.

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

    let double = $derived(count * 2);

    $effect(() => {
        if (count > 10) {
            alert('Too high!');
        }
    });
</script>
```

For more information on reactivity, read the documentation around runes.

<script context="module">

A `<script>` tag with a `context="module"` attribute runs once when the module first evaluates, rather than for each component instance. Values declared in this block are accessible from a regular `<script>` (and the component markup) but not vice versa.

You can `export` bindings from this block, and they will become exports of the compiled module.

You cannot `export default`, since the default export is the component itself.

```
<script context="module">
    let totalComponents = 0;

    // the export keyword allows this function to imported with e.g.
    // `import Example, { alertTotal } from './Example.svelte'`
    export function alertTotal() {
        alert(totalComponents);
    }
</script>

<script>
    totalComponents += 1;
    console.log(`total number of times this component has been created:
${totalComponents}`);
</script>
```

<style>

CSS inside a `<style>` block will be scoped to that component.

```
<style>
    p {
        /* this will only affect <p> elements in this component */
        color: burlywood;
    }
</style>
```

For more information regarding styling, read the documentation around styles and classes.

Go to TOC

Basic markup

- basically what we have in the Svelte docs today

Tags

A lowercase tag, like `<div>`, denotes a regular HTML element. A capitalised tag, such as `<Widget>` or `<Namespace.Widget>`, indicates a *component*.

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

<div>
    <Widget />
</div>
```

Attributes and props

By default, attributes work exactly like their HTML counterparts.

```
<div class="foo">
    <button disabled>can't touch this</button>
</div>
```

As in HTML, values may be unquoted.

```
<input type=checkbox />
```

Attribute values can contain JavaScript expressions.

```
<a href="page/{p}">page {p}</a>
```

Or they can *be* JavaScript expressions.

```
<button disabled={!clickable}>...</button>
```

Boolean attributes are included on the element if their value is truthy and excluded if it's falsy.

All other attributes are included unless their value is nullish (`null` or `undefined`).

```
<input required={false} placeholder="This input field is not required" />
<div title={null}>This div has no title attribute</div>
```

Quoting a singular expression does not affect how the value is parsed yet, but in Svelte 6 it will:

```
<button disabled="{number !== 42}">...</button>
```

When the attribute name and value match (`name={name}`), they can be replaced with `{name}`.

```
<button {disabled}>...</button>
<!-- equivalent to
<button disabled={disabled}>...</button>
-->
```

By convention, values passed to components are referred to as *properties* or *props* rather than *attributes*, which are a feature of the DOM.

As with elements, `name={name}` can be replaced with the `{name}` shorthand.

```
<Widget foo={bar} answer={42} text="hello" />
```

Spread attributes allow many attributes or properties to be passed to an element or component at once.

An element or component can have multiple spread attributes, interspersed with regular ones.

```
<Widget {...things} />
```

The `value` attribute of an `input` element or its children `option` elements must not be set with spread attributes when using `bind:group` or `bind:checked`. Svelte needs to be able to see the element's `value` directly in the markup in these cases so that it can link it to the bound variable.

Sometimes, the attribute order matters as Svelte sets attributes sequentially in JavaScript. For example, `<input type="range" min="0" max="1" value={0.5} step="0.1"/>`, Svelte will attempt to set the value to `1` (rounding up from 0.5 as the step by default is 1), and then set the step to `0.1`. To fix this, change it to `<input type="range" min="0" max="1" step="0.1" value={0.5}/>`.

Another example is `<img src="..." loading="lazy" />`. Svelte will set the img `src` before making the img element `loading="lazy"`, which is probably too late. Change this to `<img loading="lazy" src="...">` to make the image lazily loaded.

Events

Listening to DOM events is possible by adding attributes to the element that start with `on`. For example, to listen to the `click` event, add the `onclick` attribute to a button:

```
<button onclick={() => console.log('clicked')}>click me</button>
```

Event attributes are case sensitive. `onclick` listens to the `click` event, `onClick` listens to the `Click` event, which is different. This ensures you can listen to custom events that have uppercase characters in them.

Because events are just attributes, the same rules as for attributes apply:

- you can use the shorthand form: `<button {onclick}>click me</button>`
- you can spread them: `<button {...thisSpreadContainsEventAttributes}>click me</button>`
- component events are just (callback) properties and don't need a separate concept

Timing-wise, event attributes always fire after events from bindings (e.g. `oninput` always fires after an update to `bind:value`). Under the hood, some event handlers are attached directly with `addEventListener`, while others are *delegated*.

Event delegation

To reduce memory footprint and increase performance, Svelte uses a technique called event delegation. This means that for certain events — see the list below — a single event listener at the application root takes responsibility for running any handlers on the event's path.

There are a few gotchas to be aware of:

- when you manually dispatch an event with a delegated listener, make sure to set the `{ bubbles: true }` option or it won't reach the application root
- when using `addEventListener` directly, avoid calling `stopPropagation` or the event won't reach the application root and handlers won't be invoked. Similarly, handlers added manually inside the application root will run *before* handlers added declaratively deeper in the DOM (with e.g. `onclick={...}`), in both capturing and bubbling phases. For these reasons it's better to use the `on` function imported from `svelte/events` rather than `addEventListener`, as it will ensure that order is preserved and `stopPropagation` is handled correctly.

The following event handlers are delegated:

- `beforeinput`
- `click`
- `change`
- `dblclick`
- `contextmenu`
- `focusin`
- `focusout`
- `input`
- `keydown`
- `keyup`
- `mousedown`
- `mousemove`
- `mouseout`
- `mouseover`
- `mouseup`
- `pointerdown`

- `pointermove`
- `pointerout`
- `pointerover`
- `pointerup`
- `touchend`
- `touchmove`
- `touchstart`

Text expressions

A JavaScript expression can be included as text by surrounding it with curly braces.

```
{expression}
```

Curly braces can be included in a Svelte template by using their HTML entity strings: `&lbrace;`, `&lcub;`, or `&#123;` for `{` and `&rbrace;`, `&rcub;`, or `&#125;` for `}`.

If you're using a regular expression (`RegExp`) literal notation, you'll need to wrap it in parentheses.

```
<h1>Hello {name}!</h1>
<p>{a} + {b} = {a + b}.</p>

<div>{(/^[A-Za-z ]+$/).test(value) ? x : y}</div>
```

The expression will be stringified and escaped to prevent code injections. If you want to render HTML, use the `{@html}` tag instead.

```
{@html potentiallyUnsafeHtmlString}
```

Make sure that you either escape the passed string or only populate it with values that are under your control in order to prevent XSS attacks

Comments

You can use HTML comments inside components.

```
<!-- this is a comment! --><h1>Hello world</h1>
```

Comments beginning with `svelte-ignore` disable warnings for the next block of markup. Usually, these are accessibility warnings; make sure that you're disabling them for a good reason.

```
<!-- svelte-ignore a11y-autofocus -->
<input bind:value={name} autofocus />
```

You can add a special comment starting with `@component` that will show up when hovering over the component name in other files.

~~~
<!--
@component
- You can use markdown here.
- You can also use code blocks here.
- Usage:
  ```html
  <Main name="Arethra">
  ```
-->
<script>
    let { name } = $props();
</script>

<main>
    <h1>
        Hello, {name}
    </h1>
</main>
~~~

Go to TOC

Control flow

- if
- each
- await (or move that into some kind of data loading section?)
- NOT: key (move into transition section, because that's the common use case)

Svelte augments HTML with control flow blocks to be able to express conditionally rendered content or lists.

The syntax between these blocks is the same:

- `{#` denotes the start of a block
- `{:` denotes a different branch part of the block. Depending on the block, there can be multiple of these
- `{/` denotes the end of a block

{#if ...}

```
<! copy: false  --->
{#if expression}...{/if}
```

```
<! copy: false  --->
{#if expression}...{:else if expression}...{/if}
```

```
<! copy: false  --->
{#if expression}...{:else}...{/if}
```

Content that is conditionally rendered can be wrapped in an if block.

```
{#if answer === 42}
    <p>what was the question?</p>
{/if}
```

Additional conditions can be added with `{:else if expression}`, optionally ending in an `{:else}` clause.

```
{#if porridge.temperature > 100}
    <p>too hot!</p>
{:else if 80 > porridge.temperature}
    <p>too cold!</p>
{:else}
    <p>just right!</p>
{/if}
```

(Blocks don't have to wrap elements, they can also wrap text within elements!)

{#each ...}

```
<! copy: false  --->
{#each expression as name}...{/each}
```

```
<! copy: false  --->
{#each expression as name, index}...{/each}
```

```
<! copy: false  --->
{#each expression as name (key)}...{/each}
```

```
<! copy: false  --->
{#each expression as name, index (key)}...{/each}
```

```
<! copy: false  --->
{#each expression as name}...{:else}...{/each}
```

Iterating over lists of values can be done with an each block.

```
<h1>Shopping list</h1>
<ul>
    {#each items as item}
        <li>{item.name} x {item.qty}</li>
    {/each}
</ul>
```

You can use each blocks to iterate over any array or array-like value — that is, any object with a `length` property.

An each block can also specify an *index*, equivalent to the second argument in an `array.map(...)` callback:

```
{#each items as item, i}
    <li>{i + 1}: {item.name} x {item.qty}</li>
{/each}
```

If a *key* expression is provided — which must uniquely identify each list item — Svelte will use it to diff the list when data changes, rather than adding or removing items at the end. The key can be any object, but strings and numbers are recommended since they allow identity to persist when the objects themselves change.

```
{#each items as item (item.id)}
    <li>{item.name} x {item.qty}</li>
{/each}

<!-- or with additional index value -->
{#each items as item, i (item.id)}
    <li>{i + 1}: {item.name} x {item.qty}</li>
{/each}
```

You can freely use destructuring and rest patterns in each blocks.

```
{#each items as { id, name, qty }, i (id)}
    <li>{i + 1}: {name} x {qty}</li>
{/each}

{#each objects as { id, ...rest }}
    <li><span>{id}</span><MyComponent {...rest} /></li>
{/each}
```

```
{#each items as [id, ...rest]}
    <li><span>{id}</span><MyComponent values={rest} /></li>
{/each}
```

An each block can also have an `{:else}` clause, which is rendered if the list is empty.

```
{#each todos as todo}
    <p>{todo.text}</p>
{:else}
    <p>No tasks today!</p>
{/each}
```

It is possible to iterate over iterables like `Map` or `Set`. Iterables need to be finite and static (they shouldn't change while being iterated over). Under the hood, they are transformed to an array using `Array.from` before being passed off to rendering. If you're writing performance-sensitive code, try to avoid iterables and use regular arrays as they are more performant.

Other block types

Svelte also provides `#snippet`, `#key` and `#await` blocks. You can find out more about them in their respective sections.

Go to TOC

Snippets

Better title needed?

- `#snippet`
- `@render`
- how they can be used to reuse markup
- how they can be used to pass UI content to components

Snippets, and *render tags*, are a way to create reusable chunks of markup inside your components. Instead of writing duplicative code like this...

```
{#each images as image}
    {#if image.href}
        <a href={image.href}>
            <figure>
                <img src={image.src} alt={image.caption} width={image.width}
height={image.height} />
                <figcaption>{image.caption}</figcaption>
            </figure>
        </a>
    {:else}
        <figure>
            <img src={image.src} alt={image.caption} width={image.width} height=
{image.height} />
            <figcaption>{image.caption}</figcaption>
        </figure>
    {/if}
{/each}
```

...you can write this:

```
{#snippet figure(image)}
    <figure>
        <img
            src={image.src}
            alt={image.caption}
            width={image.width}
            height={image.height}
        />
        <figcaption>{image.caption}</figcaption>
    </figure>
{/snippet}

{#each images as image}
    {#if image.href}
        <a href={image.href}>
            {@render figure(image)}
        </a>
    {:else}
        {@render figure(image)}
    {/if}
{/each}
```

Like function declarations, snippets can have an arbitrary number of parameters, which can have default values, and you can destructure each parameter. You cannot use rest parameters however.

Snippet scope

Snippets can be declared anywhere inside your component. They can reference values declared outside themselves, for example in the `<script>` tag or in `{#each ...}` blocks (demo)...

```
<script>
    let { message = `it's great to see you!` } = $props();
</script>

{#snippet hello(name)}
    <p>hello {name}! {message}!</p>
{/snippet}

{@render hello('alice')}
{@render hello('bob')}
```

...and they are 'visible' to everything in the same lexical scope (i.e. siblings, and children of those siblings):

```
<div>
    {#snippet x()}
        {#snippet y()}...{/snippet}

        <!-- this is fine -->
        {@render y()}
    {/snippet}

    <!-- this will error, as `y` is not in scope -->
    {@render y()}
</div>

<!-- this will also error, as `x` is not in scope -->
{@render x()}
```

Snippets can reference themselves and each other (demo):

```
{#snippet blastoff()}
    <span>🚀</span>
{/snippet}

{#snippet countdown(n)}
    {#if n > 0}
        <span>{n}...</span>
        {@render countdown(n - 1)}
    {:else}
        {@render blastoff()}
    {/if}
{/snippet}

{@render countdown(10)}
```

Passing snippets to components

Within the template, snippets are values just like any other. As such, they can be passed to components as props (demo):

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

    const fruits = [
        { name: 'apples', qty: 5, price: 2 },
        { name: 'bananas', qty: 10, price: 1 },
        { name: 'cherries', qty: 20, price: 0.5 }
    ];
</script>

{#snippet header()}
    <th>fruit</th>
    <th>qty</th>
    <th>price</th>
    <th>total</th>
{/snippet}

{#snippet row(d)}
    <td>{d.name}</td>
    <td>{d.qty}</td>
    <td>{d.price}</td>
    <td>{d.qty * d.price}</td>
{/snippet}

<Table data={fruits} {header} {row} />
```

Think about it like passing content instead of data to a component. The concept is similar to slots in web components.

As an authoring convenience, snippets declared directly *inside* a component implicitly become props *on* the component (demo):

```
<!-- this is semantically the same as the above -->
<Table data={fruits}>
    {#snippet header()}
        <th>fruit</th>
        <th>qty</th>
        <th>price</th>
        <th>total</th>
    {/snippet}

    {#snippet row(d)}
        <td>{d.name}</td>
        <td>{d.qty}</td>
        <td>{d.price}</td>
        <td>{d.qty * d.price}</td>
    {/snippet}
</Table>
```

Any content inside the component tags that is *not* a snippet declaration implicitly becomes part of the `children` snippet (demo):

```
<! file: App.svelte --->
<Button>click me<Button>
```

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

<!-- result will be <button>click me</button> -->
<button>{@render children()}</button>
```

Note that you cannot have a prop called `children` if you also have content inside the component — for this reason, you should avoid having props with that name

You can declare snippet props as being optional. You can either use optional chaining to not render anything if the snippet isn't set...

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

{@render children?.()}
```

...or use an `#if` block to render fallback content:

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

{#if children}
    {@render children()}
{:else}
    fallback content
{/if}
```

Typing snippets

Snippets implement the `Snippet` interface imported from `'svelte'`:

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

    interface Props {
        data: any[];
        children: Snippet;
        row: Snippet<[any]>;
    }

    let { data, children, row }: Props = $props();
</script>
```

With this change, red squigglies will appear if you try and use the component without providing a `data` prop and a `row` snippet. Notice that the type argument provided to `Snippet` is a tuple, since snippets can have multiple parameters.

We can tighten things up further by declaring a generic, so that `data` and `row` refer to the same type:

```
<script lang="ts" generics="T">
    import type { Snippet } from 'svelte';

    let {
        data,
        children,
        row
    }: {
        data: T[];
        children: Snippet;
        row: Snippet<[T]>;
    } = $props();
</script>
```

Snippets and slots

In Svelte 4, content can be passed to components using slots. Snippets are more powerful and flexible, and as such slots are deprecated in Svelte 5.

Go to TOC

Styles & Classes

- style scoping
- `:global`
- `style:`
- `class:`
- `--css` props

Styling is a fundamental part of UI components. Svelte helps you style your components with ease, providing useful features out of the box.

Scoped by default

By default CSS inside a `<style>` block will be scoped to that component.

This works by adding a class to affected elements, which is based on a hash of the component styles (e.g. `svelte-123xyz`).

```
<style>
    p {
        /* this will only affect <p> elements in this component */
        color: burlywood;
    }
</style>
```

:global(...)

To apply styles to a single selector globally, use the `:global(...)` modifier:

```
<style>
    :global(body) {
        /* applies to <body> */
        margin: 0;
    }

    div :global(strong) {
        /* applies to all <strong> elements, in any component,
           that are inside <div> elements belonging
           to this component */
        color: goldenrod;
    }

    p:global(.big.red) {
        /* applies to all <p> elements belonging to this component
           with `class="big red"`, even if it is applied
           programmatically (for example by a library) */
    }
</style>
```

If you want to make @keyframes that are accessible globally, you need to prepend your keyframe names with `-global-`.

The `-global-` part will be removed when compiled, and the keyframe will then be referenced using just `my-animation-name` elsewhere in your code.

```
<style>
    @keyframes -global-my-animation-name {
        /* code goes here */
    }
</style>
```

:global

To apply styles to a group of selectors globally, create a `:global {...}` block:

```
<style>
    :global {
        /* applies to every <div> in your application */
        div { ... }

        /* applies to every <p> in your application */
        p { ... }
    }

    .a :global {
        /* applies to every `.b .c .d` element, in any component,
           that is inside an `.a` element in this component */
        .b .c .d {...}
    }
</style>
```

> The second example above could also be written as an equivalent `.a :global .b .c .d` selector, where everything after the `:global` is unscoped, though the nested form is preferred.

Nested style tags

There should only be 1 top-level `<style>` tag per component.

However, it is possible to have a `<style>` tag nested inside other elements or logic blocks.

In that case, the `<style>` tag will be inserted as-is into the DOM; no scoping or processing will be done on the `<style>` tag.

```
<div>
    <style>
        /* this style tag will be inserted as-is */
        div {
            /* this will apply to all `<div>` elements in the DOM */
            color: red;
        }
    </style>
</div>
```

class:*name*

```
<! copy: false --->
class:name={value}
```

```
<! copy: false --->
class:name
```

A `class:` directive provides a shorter way of toggling a class on an element.

```
<!-- These are equivalent -->
<div class={isActive ? 'active' : ''}>...</div>
<div class:active={isActive}>...</div>

<!-- Shorthand, for when name and value match -->
<div class:active>...</div>

<!-- Multiple class toggles can be included -->
<div class:active class:inactive={!active} class:isAdmin>...</div>
```

style:*property*

```
<! copy: false --->
style:property={value}
```

```
<! copy: false --->
style:property="value"
```

```
<! copy: false --->
style:property
```

The `style:` directive provides a shorthand for setting multiple styles on an element.

```
<!-- These are equivalent -->
<div style:color="red">...</div>
<div style="color: red;">...</div>

<!-- Variables can be used -->
<div style:color={myColor}>...</div>

<!-- Shorthand, for when property and variable name match -->
<div style:color>...</div>

<!-- Multiple styles can be included -->
<div style:color style:width="12rem" style:background-color={darkMode ? 'black' :
'white'}>...</div>

<!-- Styles can be marked as important -->
<div style:color|important="red">...</div>
```

When `style:` directives are combined with `style` attributes, the directives will take precedence:

```
<div style="color: blue;" style:color="red">This will be red</div>
```

--style-props

```
<! copy: false --->
--style-props="anycssvalue"
```

You can also pass styles as props to components for the purposes of theming, using CSS custom properties.

Svelte's implementation is essentially syntactic sugar for adding a wrapper element. This example:

```
<Slider bind:value min={0} --rail-color="black" --track-color="rgb(0, 0, 255)" />
```

Desugars to this:

```
<div style="display: contents; --rail-color: black; --track-color: rgb(0, 0,
255)">
    <Slider bind:value min={0} max={100} />
</div>
```

For SVG namespace, the example above desugars into using `<g>` instead:

```
<g style="--rail-color: black; --track-color: rgb(0, 0, 255)">
    <Slider bind:value min={0} max={100} />
</g>
```

> Since this is an extra `<div>` (or `<g>`), beware that your CSS structure might accidentally target this. Be mindful of this added wrapper element when using this feature.

Svelte's CSS Variables support allows for easily themeable components:

```
<style>
    .potato-slider-rail {
        background-color: var(--rail-color, var(--theme-color, 'purple'));
    }
</style>
```

So you can set a high-level theme color:

```
/* global.css */
html {
    --theme-color: black;
}
```

Or override it at the consumer level:

```
<Slider --rail-color="goldenrod" />
```

Go to TOC

Transitions & Animations

- how to use (template syntax)
- when to use
- global vs local
- easing & motion
- mention imports
- key block

Svelte provides different techniques and syntax for incorporating motion into your Svelte projects.

transition:*fn*

```
<! copy: false --->
transition:fn
```

```
<! copy: false --->
transition:fn={params}
```

```
<! copy: false --->
transition:fn|global
```

```
<! copy: false --->
transition:fn|global={params}
```

```
<! copy: false --->
transition:fn|local
```

```
<! copy: false --->
transition:fn|local={params}
```

```
/// copy: false
// @noErrors
transition = (node: HTMLElement, params: any, options: { direction: 'in' | 'out' |
'both' }) => {
    delay?: number,
    duration?: number,
    easing?: (t: number) => number,
    css?: (t: number, u: number) => string,
    tick?: (t: number, u: number) => void
}
```

A transition is triggered by an element entering or leaving the DOM as a result of a state change.

When a block is transitioning out, all elements inside the block, including those that do not have their own transitions, are kept in the DOM until every transition in the block has been completed.

The `transition:` directive indicates a *bidirectional* transition, which means it can be smoothly reversed while the transition is in progress.

```
{#if visible}
    <div transition:fade>fades in and out</div>
{/if}
```

Transitions are local by default. Local transitions only play when the block they belong to is created or destroyed, *not* when parent blocks are created or destroyed.

```
{#if x}
    {#if y}
        <p transition:fade>fades in and out only when y changes</p>

        <p transition:fade|global>fades in and out when x or y change</p>
    {/if}
{/if}
```

By default intro transitions will not play on first render. You can modify this behaviour by setting `intro: true` when you create a component and marking the transition as `global`.

Transition parameters

Transitions can have parameters.

(The double `{{curlies}}` aren't a special syntax; this is an object literal inside an expression tag.)

```
{#if visible}
    <div transition:fade={{ duration: 2000 }}>fades in and out over two
seconds</div>
{/if}
```

Custom transition functions

Transitions can use custom functions. If the returned object has a `css` function, Svelte will create a CSS animation that plays on the element.

The `t` argument passed to `css` is a value between `0` and `1` after the `easing` function has been applied. *In* transitions run from `0` to `1`, *out* transitions run from `1` to `0` — in other words, `1` is the element's natural state, as though no transition had been applied. The `u` argument is equal to `1 - t`.

The function is called repeatedly *before* the transition begins, with different `t` and `u` arguments.

```
<script>
    import { elasticOut } from 'svelte/easing';

    /** @type {boolean} */
    export let visible;

    /**
     * @param {HTMLElement} node
     * @param {{ delay?: number, duration?: number, easing?: (t: number) => number
}} params
     */
    function whoosh(node, params) {
        const existingTransform = getComputedStyle(node).transform.replace('none',
'');

        return {
            delay: params.delay || 0,
            duration: params.duration || 400,
            easing: params.easing || elasticOut,
            css: (t, u) => `transform: ${existingTransform} scale(${t})`
        };
    }
</script>

{#if visible}
    <div in:whoosh>whooshes in</div>
{/if}
```

A custom transition function can also return a `tick` function, which is called *during* the transition with the same `t` and `u` arguments.

> If it's possible to use `css` instead of `tick`, do so — CSS animations can run off the main thread, preventing jank on slower devices.

```
<! file: App.svelte --->
<script>
    export let visible = false;

    /**
     * @param {HTMLElement} node
     * @param {{ speed?: number }} params
     */
    function typewriter(node, { speed = 1 }) {
        const valid = node.childNodes.length === 1 && node.childNodes[0].nodeType
=== Node.TEXT_NODE;

        if (!valid) {
            throw new Error(`This transition only works on elements with a single
text node child`);
        }

        const text = node.textContent;
        const duration = text.length / (speed * 0.01);

        return {
            duration,
            tick: (t) => {
                const i = ~~(text.length * t);
                node.textContent = text.slice(0, i);
            }
        };
    }
</script>

{#if visible}
    <p in:typewriter={{ speed: 1 }}>The quick brown fox jumps over the lazy
dog</p>
{/if}
```

If a transition returns a function instead of a transition object, the function will be called in the next micro-task. This allows multiple transitions to coordinate, making crossfade effects possible.

Transition functions also receive a third argument, `options`, which contains information about the transition.

Available values in the `options` object are:

- `direction` - one of `in`, `out`, or `both` depending on the type of transition

Transition events

An element with transitions will dispatch the following events in addition to any standard DOM events:

- `introstart`
- `introend`
- `outrostart`
- `outroend`

```
{#if visible}
    <p
        transition:fly={{ y: 200, duration: 2000 }}
        on:introstart={() => (status = 'intro started')}
        on:outrostart={() => (status = 'outro started')}
        on:introend={() => (status = 'intro ended')}
        on:outroend={() => (status = 'outro ended')}
    >
        Flies in and out
    </p>
{/if}
```

in:*fn*/out:*fn*

```
<! copy: false --->
in:fn
```

```
<! copy: false --->
in:fn={params}
```

```
<! copy: false --->
in:fn|global
```

```
<! copy: false --->
in:fn|global={params}
```

```
<! copy: false --->
in:fn|local
```

```
<! copy: false --->
in:fn|local={params}
```

```
<! copy: false --->
out:fn
```

```
<! copy: false --->
out:fn={params}
```

```
<! copy: false --->
out:fn|global
```

```
<! copy: false --->
out:fn|global={params}
```

```
<! copy: false --->
out:fn|local
```

```
<! copy: false --->
out:fn|local={params}
```

Similar to `transition:`, but only applies to elements entering (`in:`) or leaving (`out:`) the DOM.

Unlike with `transition:`, transitions applied with `in:` and `out:` are not bidirectional — an in transition will continue to 'play' alongside the out transition, rather than reversing, if the block is outroed while the transition is in progress. If an out transition is aborted, transitions will restart from scratch.

```
{#if visible}
    <div in:fly out:fade>flies in, fades out</div>
{/if}
```

animate:*fn*

```
<! copy: false --->
animate:name
```

```
<! copy: false --->
animate:name={params}
```

```
/// copy: false
// @noErrors
animation = (node: HTMLElement, { from: DOMRect, to: DOMRect } , params: any) => {
    delay?: number,
    duration?: number,
    easing?: (t: number) => number,
    css?: (t: number, u: number) => string,
    tick?: (t: number, u: number) => void
}
```

```
/// copy: false
// @noErrors
DOMRect {
    bottom: number,
    height: number,
    left: number,
    right: number,
    top: number,
    width: number,
    x: number,
    y: number
}
```

An animation is triggered when the contents of a keyed each block are re-ordered. Animations do not run when an element is added or removed, only when the index of an existing data item within the each block changes. Animate directives must be on an element that is an *immediate* child of a keyed each block.

Animations can be used with Svelte's built-in animation functions or custom animation functions.

```
<!-- When `list` is reordered the animation will run-->
{#each list as item, index (item)}
    <li animate:flip>{item}</li>
{/each}
```

Animation Parameters

As with actions and transitions, animations can have parameters.

(The double `{{curlies}}` aren't a special syntax; this is an object literal inside an expression tag.)

```
{#each list as item, index (item)}
    <li animate:flip={{ delay: 500 }}>{item}</li>
{/each}
```

Custom animation functions

Animations can use custom functions that provide the `node`, an `animation` object and any `parameters` as arguments. The `animation` parameter is an object containing `from` and `to` properties each containing a DOMRect describing the geometry of the element in its `start` and `end` positions. The `from` property is the DOMRect of the element in its starting position, and the `to` property is the DOMRect of the element in its final position after the list has been reordered and the DOM updated.

If the returned object has a `css` method, Svelte will create a CSS animation that plays on the element.

The `t` argument passed to `css` is a value that goes from `0` and `1` after the `easing` function has been applied. The `u` argument is equal to `1 - t`.

The function is called repeatedly *before* the animation begins, with different `t` and `u` arguments.

```
<script>
    import { cubicOut } from 'svelte/easing';

    /**
     * @param {HTMLElement} node
     * @param {{ from: DOMRect; to: DOMRect }} states
     * @param {any} params
     */
    function whizz(node, { from, to }, params) {
        const dx = from.left - to.left;
        const dy = from.top - to.top;

        const d = Math.sqrt(dx * dx + dy * dy);

        return {
            delay: 0,
            duration: Math.sqrt(d) * 120,
            easing: cubicOut,
            css: (t, u) => `transform: translate(${u * dx}px, ${u * dy}px)
rotate(${t * 360}deg);`
        };
    }
</script>

{#each list as item, index (item)}
    <div animate:whizz>{item}</div>
{/each}
```

A custom animation function can also return a `tick` function, which is called *during* the animation with the same `t` and `u` arguments.

> If it's possible to use `css` instead of `tick`, do so — CSS animations can run off the main thread, preventing jank on slower devices.

```
<script>
    import { cubicOut } from 'svelte/easing';

    /**
     * @param {HTMLElement} node
     * @param {{ from: DOMRect; to: DOMRect }} states
     * @param {any} params
     */
    function whizz(node, { from, to }, params) {
        const dx = from.left - to.left;
        const dy = from.top - to.top;

        const d = Math.sqrt(dx * dx + dy * dy);

        return {
            delay: 0,
            duration: Math.sqrt(d) * 120,
            easing: cubicOut,
            tick: (t, u) => Object.assign(node.style, { color: t > 0.5 ? 'Pink' :
'Blue' })
        };
    }
</script>

{#each list as item, index (item)}
    <div animate:whizz>{item}</div>
{/each}
```

{#key ...}

```
<! copy: false  --->
{#key expression}...{/key}
```

Key blocks destroy and recreate their contents when the value of an expression changes.

This is useful if you want an element to play its transition whenever a value changes.

```
{#key value}
    <div transition:fade>{value}</div>
{/key}
```

When used around components, this will cause them to be reinstantiated and reinitialised.

```
{#key value}
    <Component />
{/key}
```

Actions

- template syntax
- how to write
- typings
- adjust so that `$effect` is used instead of update/destroy?

```
<! copy: false --->
use:action
```

```
<! copy: false --->
use:action={parameters}
```

```
/// copy: false
// @noErrors
action = (node: HTMLElement, parameters: any) => {
    update?: (parameters: any) => void,
    destroy?: () => void
}
```

Actions are functions that are called when an element is created. They can return an object with a `destroy` method that is called after the element is unmounted:

```
<! file: App.svelte --->
<script>
    /** @type {import('svelte/action').Action}  */
    function foo(node) {
        // the node has been mounted in the DOM

        return {
            destroy() {
                // the node has been removed from the DOM
            }
        };
    }
</script>

<div use:foo />
```

An action can have a parameter. If the returned value has an `update` method, it will be called immediately after Svelte has applied updates to the markup whenever that parameter changes.

> Don't worry that we're redeclaring the `foo` function for every component instance — Svelte will hoist any functions that don't depend on local state out of the component definition.

```
<! file: App.svelte --->
<script>
    /** @type {string} */
    export let bar;
    /** @type {import('svelte/action').Action<HTMLElement, string>}  */
    function foo(node, bar) {
        // the node has been mounted in the DOM

        return {
            update(bar) {
                // the value of `bar` has changed
            },

            destroy() {
                // the node has been removed from the DOM
            }
        };
    }
</script>

<div use:foo={bar} />
```

Attributes

Sometimes actions emit custom events and apply custom attributes to the element they are applied to. To support this, actions typed with `Action` or `ActionReturn` type can have a last parameter, `Attributes` :

```
<! file: App.svelte --->
<script>
    /**
     * @type {import('svelte/action').Action<HTMLDivElement, { prop: any }, {
'on:emit': (e: CustomEvent<string>) => void }>}
     */
    function foo(node, { prop }) {
        // the node has been mounted in the DOM

        //...LOGIC
        node.dispatchEvent(new CustomEvent('emit', { detail: 'hello' }));

        return {
            destroy() {
                // the node has been removed from the DOM
            }
        };
    }
</script>

<div use:foo={{ prop: 'someValue' }} onemit={handleEmit} />
```

Go to TOC

Bindings

- how for dom elements
- list of all bindings
- how for components

Most of the time a clear separation between data flowing down and events going up is worthwhile and results in more robust apps. But in some cases - especially when interacting with form elements - it's more ergonomic to declare a two way binding. Svelte provides many element bindings out of the box, and also allows component bindings.

bind:*property* for elements

```
<! copy: false --->
bind:property={variable}
```

Data ordinarily flows down, from parent to child. The `bind:` directive allows data to flow the other way, from child to parent. Most bindings are specific to particular elements.

The simplest bindings reflect the value of a property, such as `input.value`.

```
<input bind:value={name} />
<textarea bind:value={text} />

<input type="checkbox" bind:checked={yes} />
```

If the name matches the value, you can use a shorthand.

```
<input bind:value />
<!-- equivalent to
<input bind:value={value} />
-->
```

Numeric input values are coerced; even though `input.value` is a string as far as the DOM is concerned, Svelte will treat it as a number. If the input is empty or invalid (in the case of `type="number"`), the value is `undefined`.

```
<input type="number" bind:value={num} />
<input type="range" bind:value={num} />
```

On `<input>` elements with `type="file"`, you can use `bind:files` to get the `FileList` of selected files. It is readonly.

```
<label for="avatar">Upload a picture:</label>
<input accept="image/png, image/jpeg" bind:files id="avatar" name="avatar"
type="file" />
```

If you're using `bind:` directives together with `on:` directives, the order that they're defined in affects the value of the bound variable when the event handler is called.

```
<script>
    let value = 'Hello World';
</script>

<input
    on:input={() => console.log('Old value:', value)}
    bind:value
    on:input={() => console.log('New value:', value)}
/>
```

Here we were binding to the value of a text input, which uses the `input` event. Bindings on other elements may use different events such as `change`.

Binding `<select>` value

A `<select>` value binding corresponds to the `value` property on the selected `<option>`, which can be any value (not just strings, as is normally the case in the DOM).

```
<select bind:value={selected}>
    <option value={a}>a</option>
    <option value={b}>b</option>
    <option value={c}>c</option>
</select>
```

A `<select multiple>` element behaves similarly to a checkbox group. The bound variable is an array with an entry corresponding to the `value` property of each selected `<option>`.

```
<select multiple bind:value={fillings}>
    <option value="Rice">Rice</option>
    <option value="Beans">Beans</option>
    <option value="Cheese">Cheese</option>
    <option value="Guac (extra)">Guac (extra)</option>
</select>
```

When the value of an `<option>` matches its text content, the attribute can be omitted.

```
<select multiple bind:value={fillings}>
    <option>Rice</option>
    <option>Beans</option>
    <option>Cheese</option>
    <option>Guac (extra)</option>
</select>
```

Elements with the `contenteditable` attribute support the following bindings:

- `innerHTML`
- `innerText`
- `textContent`

There are slight differences between each of these, read more about them here.

```
<div contenteditable="true" bind:innerHTML={html} />
```

`<details>` elements support binding to the `open` property.

```
<details bind:open={isOpen}>
    <summary>Details</summary>
    <p>Something small enough to escape casual notice.</p>
</details>
```

Media element bindings

Media elements (`<audio>` and `<video>`) have their own set of bindings — seven *readonly* ones...

- `duration` (readonly) — the total duration of the video, in seconds
- `buffered` (readonly) — an array of `{start, end}` objects
- `played` (readonly) — ditto
- `seekable` (readonly) — ditto
- `seeking` (readonly) — boolean
- `ended` (readonly) — boolean
- `readyState` (readonly) — number between (and including) 0 and 4

...and five *two-way* bindings:

- `currentTime` — the current playback time in the video, in seconds
- `playbackRate` — how fast or slow to play the video, where 1 is 'normal'
- `paused` — this one should be self-explanatory
- `volume` — a value between 0 and 1
- `muted` — a boolean value indicating whether the player is muted

Videos additionally have readonly `videoWidth` and `videoHeight` bindings.

```
<video
    src={clip}
    bind:duration
    bind:buffered
    bind:played
    bind:seekable
    bind:seeking
    bind:ended
    bind:readyState
    bind:currentTime
    bind:playbackRate
    bind:paused
    bind:volume
    bind:muted
    bind:videoWidth
    bind:videoHeight
/>
```

Image element bindings

Image elements (`<img>`) have two readonly bindings:

- `naturalWidth` (readonly) — the original width of the image, available after the image has loaded
- `naturalHeight` (readonly) — the original height of the image, available after the image has loaded

```
<img
    bind:naturalWidth
    bind:naturalHeight
></img>
```

Block-level element bindings

Block-level elements have 4 read-only bindings, measured using a technique similar to this one:

- `clientWidth`
- `clientHeight`
- `offsetWidth`
- `offsetHeight`

```
<div bind:offsetWidth={width} bind:offsetHeight={height}>
    <Chart {width} {height} />
</div>
```

bind:group

```
<! copy: false --->
bind:group={variable}
```

Inputs that work together can use `bind:group`.

```
<script>
    let tortilla = 'Plain';

    /** @type {Array<string>} */
    let fillings = [];
</script>

<!-- grouped radio inputs are mutually exclusive -->
<input type="radio" bind:group={tortilla} value="Plain" />
<input type="radio" bind:group={tortilla} value="Whole wheat" />
<input type="radio" bind:group={tortilla} value="Spinach" />

<!-- grouped checkbox inputs populate an array -->
<input type="checkbox" bind:group={fillings} value="Rice" />
<input type="checkbox" bind:group={fillings} value="Beans" />
<input type="checkbox" bind:group={fillings} value="Cheese" />
<input type="checkbox" bind:group={fillings} value="Guac (extra)" />
```

`bind:group` only works if the inputs are in the same Svelte component.

bind:this

```
<! copy: false --->
bind:this={dom_node}
```

To get a reference to a DOM node, use `bind:this`.

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

    /** @type {HTMLCanvasElement} */
    let canvasElement;

    onMount(() => {
        const ctx = canvasElement.getContext('2d');
        drawStuff(ctx);
    });
</script>

<canvas bind:this={canvasElement} />
```

Components also support `bind:this`, allowing you to interact with component instances programmatically.

```
<! App.svelte --->
<ShoppingCart bind:this={cart} />

<button onclick={() => cart.empty()}> Empty shopping cart </button>
```

```
<! ShoppingCart.svelte --->
<script>
    // All instance exports are available on the instance object
    export function empty() {
        // ...
    }
</script>
```

Note that we can't do `{cart.empty}` since `cart` is `undefined` when the button is first rendered and throws an error.

bind:*property* for components

```
bind:property={variable}
```

You can bind to component props using the same syntax as for elements.

```
<Keypad bind:value={pin} />
```

While Svelte props are reactive without binding, that reactivity only flows downward into the component by default. Using `bind:property` allows changes to the property from within the component to flow back up out of the component.

To mark a property as bindable, use the `$bindable` rune:

```
<script>
    let { readonlyProperty, bindableProperty = $bindable() } = $props();
</script>
```

Declaring a property as bindable means it *can* be used using `bind:`, not that it *must* be used using `bind:`.

Bindable properties can have a fallback value:

```
<script>
    let { bindableProperty = $bindable('fallback value') } = $props();
</script>
```

This fallback value *only* applies when the property is *not* bound. When the property is bound and a fallback value is present, the parent is expected to provide a value other than `undefined`, else a runtime error is thrown. This prevents hard-to-reason-about situations where it's unclear which value should apply.

Go to TOC

Special elements

- basically what we have in the docs today

Some of Svelte's concepts need special elements. Those are prefixed with `svelte:` and listed here.

`<svelte:self>`

The `<svelte:self>` element allows a component to include itself, recursively.

It cannot appear at the top level of your markup; it must be inside an if or each block or passed to a component's slot to prevent an infinite loop.

```
<script>
    /** @type {number} */
    export let count;
</script>

{#if count > 0}
    <p>counting down... {count}</p>
    <svelte:self count={count - 1} />
{:else}
    <p>lift-off!</p>
{/if}
```

`<svelte:component>`

```
<svelte:component this={expression} />
```

The `<svelte:component>` element renders a component dynamically, using the component constructor specified as the `this` property. When the property changes, the component is destroyed and recreated.

If `this` is falsy, no component is rendered.

```
<svelte:component this={currentSelection.component} foo={bar} />
```

`<svelte:element>`

```
<svelte:element this={expression} />
```

The `<svelte:element>` element lets you render an element of a dynamically specified type. This is useful for example when displaying rich text content from a CMS. Any properties and event listeners present will be applied to the element.

The only supported binding is `bind:this`, since the element type-specific bindings that Svelte does at build time (e.g. `bind:value` for input elements) do not work with a dynamic tag type.

If `this` has a nullish value, the element and its children will not be rendered.

If `this` is the name of a void element (e.g., `br`) and `<svelte:element>` has child elements, a runtime error will be thrown in development mode.

```
<script>
    let tag = 'div';

    export let handler;
</script>

<svelte:element this={tag} on:click={handler}>Foo</svelte:element>
```

Svelte tries its best to infer the correct namespace from the element's surroundings, but it's not always possible. You can make it explicit with an `xmlns` attribute:

```
<svelte:element this={tag} xmlns="http://www.w3.org/2000/svg" />
```

`<svelte:window>`

```
<svelte:window on:event={handler} />
```

```
<svelte:window bind:prop={value} />
```

The `<svelte:window>` element allows you to add event listeners to the `window` object without worrying about removing them when the component is destroyed, or checking for the existence of `window` when server-side rendering.

Unlike `<svelte:self>`, this element may only appear at the top level of your component and must never be inside a block or element.

```
<script>
    /** @param {KeyboardEvent} event */
    function handleKeydown(event) {
        alert(`pressed the ${event.key} key`);
    }
</script>

<svelte:window on:keydown={handleKeydown} />
```

You can also bind to the following properties:

- `innerWidth`
- `innerHeight`
- `outerWidth`
- `outerHeight`
- `scrollX`
- `scrollY`
- `online` — an alias for `window.navigator.onLine`
- `devicePixelRatio`

All except `scrollX` and `scrollY` are readonly.

```
<svelte:window bind:scrollY={y} />
```

> Note that the page will not be scrolled to the initial value to avoid accessibility issues. Only subsequent changes to the bound variable of `scrollX` and `scrollY` will cause scrolling. However, if the scrolling behaviour is desired, call `scrollTo()` in `onMount()` .

`<svelte:document>`

```
<svelte:document on:event={handler} />
```

```
<svelte:document bind:prop={value} />
```

Similarly to `<svelte:window>`, this element allows you to add listeners to events on `document`, such as `visibilitychange`, which don't fire on `window`. It also lets you use actions on `document`.

As with `<svelte:window>`, this element may only appear the top level of your component and must never be inside a block or element.

```
<svelte:document on:visibilitychange={handleVisibilityChange} use:someAction />
```

You can also bind to the following properties:

- `activeElement`
- `fullscreenElement`
- `pointerLockElement`
- `visibilityState`

All are readonly.

`<svelte:body>`

```
<svelte:body on:event={handler} />
```

Similarly to `<svelte:window>`, this element allows you to add listeners to events on `document.body`, such as `mouseenter` and `mouseleave`, which don't fire on `window`. It also lets you use actions on the `<body>` element.

As with `<svelte:window>` and `<svelte:document>`, this element may only appear the top level of your component and must never be inside a block or element.

```
<svelte:body on:mouseenter={handleMouseenter} on:mouseleave={handleMouseleave}
use:someAction />
```

`<svelte:head>`

```
<svelte:head>...</svelte:head>
```

This element makes it possible to insert elements into `document.head`. During server-side rendering, `head` content is exposed separately to the main `html` content.

As with `<svelte:window>`, `<svelte:document>` and `<svelte:body>`, this element may only appear at the top level of your component and must never be inside a block or element.

```
<svelte:head>
    <title>Hello world!</title>
    <meta name="description" content="This is where the description goes for SEO"
/>
</svelte:head>
```

`<svelte:options>`

```
<svelte:options option={value} />
```

The `<svelte:options>` element provides a place to specify per-component compiler options, which are detailed in the compiler section. The possible options are:

- `immutable={true}` — you never use mutable data, so the compiler can do simple referential equality checks to determine if values have changed
- `immutable={false}` — the default. Svelte will be more conservative about whether or not mutable objects have changed
- `accessors={true}` — adds getters and setters for the component's props
- `accessors={false}` — the default
- `namespace="..."` — the namespace where this component will be used, most commonly "svg"; use the "foreign" namespace to opt out of case-insensitive attribute names and HTML-specific warnings
- `customElement="..."` — the name to use when compiling this component as a custom element

```
<svelte:options customElement="my-custom-element" />
```

Go to TOC

Data fetching

Fetching data is a fundamental part of apps interacting with the outside world. Svelte is unopinionated with how you fetch your data. The simplest way would be using the built-in `fetch` method:

```
<script>
    let response = $state();
    fetch('/api/data').then(async (r) => (response = r.json()));
</script>
```

While this works, it makes working with promises somewhat unergonomic. Svelte alleviates this problem using the `#await` block.

{#await ...}

```
<! copy: false  --->
{#await expression}...{:then name}...{:catch name}...{/await}
```

```
<! copy: false  --->
{#await expression}...{:then name}...{/await}
```

```
<! copy: false  --->
{#await expression then name}...{/await}
```

```
<! copy: false  --->
{#await expression catch name}...{/await}
```

Await blocks allow you to branch on the three possible states of a Promise — pending, fulfilled or rejected. In SSR mode, only the pending branch will be rendered on the server. If the provided expression is not a Promise only the fulfilled branch will be rendered, including in SSR mode.

```
{#await promise}
    <!-- promise is pending -->
    <p>waiting for the promise to resolve...</p>
{:then value}
    <!-- promise was fulfilled or not a Promise -->
    <p>The value is {value}</p>
{:catch error}
    <!-- promise was rejected -->
    <p>Something went wrong: {error.message}</p>
{/await}
```

The `catch` block can be omitted if you don't need to render anything when the promise rejects (or no error is possible).

```
{#await promise}
    <!-- promise is pending -->
    <p>waiting for the promise to resolve...</p>
{:then value}
    <!-- promise was fulfilled -->
    <p>The value is {value}</p>
{/await}
```

If you don't care about the pending state, you can also omit the initial block.

```
{#await promise then value}
    <p>The value is {value}</p>
{/await}
```

Similarly, if you only want to show the error state, you can omit the `then` block.

```
{#await promise catch error}
    <p>The error is {error}</p>
{/await}
```

SvelteKit loaders

Fetching inside your components is great for simple use cases, but it's prone to data loading waterfalls and makes code harder to work with because of the promise handling. SvelteKit solves this problem by providing a opinionated data loading story that is coupled to its router. Learn more about it in the docs.

Go to TOC

Template syntax

Go to TOC

State

- `$state` (.frozen)
- `$derived` (.by)
- using classes
- getters/setters (what to do to keep reactivity "alive")
- universal reactivity

Svelte 5 uses *runes*, a powerful set of primitives for controlling reactivity inside your Svelte components and inside `.svelte.js` and `.svelte.ts` modules.

Runes are function-like symbols that provide instructions to the Svelte compiler. You don't need to import them from anywhere — when you use Svelte, they're part of the language. This page describes the runes that are concerned with managing state in your application.

`$state`

The `$state` rune is the at the heart of the runes API. It is used to declare reactive state:

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

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

Variables declared with `$state` are the variable *itself*, in other words there's no wrapper around the value that it contains. This is possible thanks to the compiler-nature of Svelte. As such, updating state is done through simple reassignment.

You can also use `$state` in class fields (whether public or private):

```
// @errors: 7006 2554
class Todo {
    done = $state(false);
    text = $state();

    constructor(text) {
        this.text = text;
    }

    reset() {
        this.text = '';
        this.done = false;
    }
}
```

> In this example, the compiler transforms `done` and `text` into `get` / `set` methods on the class prototype referencing private fields

Objects and arrays are made deeply reactive by wrapping them with `Proxies`. What that means is that in the following example, we can mutate the `entries` object and the UI will still update - but only the list item that is actually changed will rerender:

```
<script>
    let entries = $state([
        { id: 1, text: 'foo' },
        { id: 2, text: 'bar' }
    ]);
</script>

{#each entries as entry (entry.id)}
    {entry.text}
{/each}

<button onclick={() => (entries[1].text = 'baz')}>change second entry
text</button>
```

> Only POJOs (plain old JavaScript objects) are made deeply reactive. Reactivity will stop at class boundaries and leave those alone

`$state.frozen`

State declared with `$state.frozen` cannot be mutated; it can only be *reassigned*. In other words, rather than assigning to a property of an object, or using an array method like `push`, replace the object or array altogether if you'd like to update it:

```
let person = $state.frozen({
    name: 'Heraclitus',
    age: 49
});

// this will have no effect (and will throw an error in dev)
person.age += 1;

// this will work, because we're creating a new person
person = {
    name: 'Heraclitus',
    age: 50
};
```

This can improve performance with large arrays and objects that you weren't planning to mutate anyway, since it avoids the cost of making them reactive. Note that frozen state can *contain* reactive state (for example, a frozen array of reactive objects).

In development mode, the argument to `$state.frozen` will be shallowly frozen with `Object.freeze()`, to make it obvious if you accidentally mutate it.

> Objects and arrays passed to `$state.frozen` will have a `Symbol` property added to them to signal to Svelte that they are frozen. If you don't want this, pass in a clone of the object or array instead. The argument cannot be an existing state proxy created with `$state(...)`.

`$state.snapshot`

To take a static snapshot of a deeply reactive `$state` proxy, use `$state.snapshot`:

```
<script>
    let counter = $state({ count: 0 });

    function onclick() {
        // Will log `{ count: ... }` rather than `Proxy { ... }`
        console.log($state.snapshot(counter));
    }
</script>
```

This is handy when you want to pass some state to an external library or API that doesn't expect a proxy, such as `structuredClone`.

`$state.is`

Sometimes you might need to compare two values, one of which is a reactive `$state(...)` proxy but the other is not. For this you can use `$state.is(a, b)`:

```
<script>
    let foo = $state({});
    let bar = {};

    foo.bar = bar;

    console.log(foo.bar === bar); // false — `foo.bar` is a reactive proxy
    console.log($state.is(foo.bar, bar)); // true
</script>
```

This is handy when you might want to check if the object exists within a deeply reactive object/array.

Under the hood, `$state.is` uses `Object.is` for comparing the values.

> Use this as an escape hatch - most of the time you don't need this. Svelte will warn you at dev time if you happen to run into this problem

`$derived`

Derived state is declared with the `$derived` rune:

```
<script>
    let count = $state(0);
    let doubled = $derived(count * 2);
</script>

<button onclick={() => count++}>
    {doubled}
</button>

<p>{count} doubled is {doubled}</p>
```

The expression inside `$derived(...)` should be free of side-effects. Svelte will disallow state changes (e.g. `count++`) inside derived expressions.

As with `$state`, you can mark class fields as `$derived`.

`$derived.by`

Sometimes you need to create complex derivations that don't fit inside a short expression. In these cases, you can use `$derived.by` which accepts a function as its argument.

```
<script>
    let numbers = $state([1, 2, 3]);
    let total = $derived.by(() => {
        let total = 0;
        for (const n of numbers) {
            total += n;
        }
        return total;
    });
</script>

<button onclick={() => numbers.push(numbers.length + 1)}>
    {numbers.join(' + ')} = {total}
</button>
```

In essence, `$derived(expression)` is equivalent to `$derived.by(() => expression)`.

Universal reactivity

In the examples above, `$state` and `$derived` only appear at the top level of components. You can also use them within functions or even outside Svelte components inside `.svelte.js` or `.svelte.ts` modules.

```
/// file: counter.svelte.ts
export function createCounter(initial: number) {
    let count = $state(initial);
    let double = $derived(count * 2);
    return {
        get count() {
            return count;
```

```
		},
		get double() {
			return double;
		},
		increment: () => count++
	};
}
```

```
<! file: App.svelte --->
<script>
	import { createCounter } from './counter.svelte';

	const counter = createCounter();
</script>

<button onclick={counter.increment}>{counter.count} / {counter.double}</button>
```

There are a few things to note in the above example:

- We're using getters to transport reactivity across the function boundary. This way we keep reactivity "alive". If we were to return the value itself, it would be fixed to the value at that point in time. This is no different to how regular JavaScript variables behave.
- We're not destructuring the counter at the usage site. Because we're using getters, destructuring would fix `count` and `double` to the value at that point in time. To keep the getters "alive", we're not using destructuring. Again, this is how regular JavaScript works.

If you have shared state you want to manipulate from various places, you don't need to resort to getters. Instead, you can take advantage of `$state` being deeply reactive and only update its properties, not the value itself:

```
/// file: app-state.svelte.ts
export const appState = $state({
	loggedIn: true
});
```

```
<! file: App.svelte --->
<script>
	import { appState } from './app-state.svelte';
</script>

<button onclick={() => (appState.loggedIn = false)}>Log out</button>
```

Go to TOC

Side effects

- `$effect` (.pre)
- when not to use it, better patterns for what to do instead

Side effects play a crucial role in applications. They are triggered by state changes and can then interact with external systems, like logging something, setting up a server connection or synchronize with a third-party library that has no knowledge of Svelte's reactivity model.

`$effect` fundamentals

To run *side-effects* when the component is mounted to the DOM, and when values change, we can use the `$effect` rune (demo):

```
<script>
    let size = $state(50);
    let color = $state('#ff3e00');

    let canvas;

    $effect(() => {
        const context = canvas.getContext('2d');
        context.clearRect(0, 0, canvas.width, canvas.height);

        // this will re-run whenever `color` or `size` change
        context.fillStyle = color;
        context.fillRect(0, 0, size, size);
    });
</script>

<canvas bind:this={canvas} width="100" height="100" />
```

The function passed to `$effect` will run when the component mounts, and will re-run after any changes to the values it reads that were declared with `$state` or `$derived` (including those passed in with `$props`). Re-runs are batched (i.e. changing `color` and `size` in the same moment won't cause two separate runs), and happen after any DOM updates have been applied.

You can return a function from `$effect`, which will run immediately before the effect re-runs, and before it is destroyed (demo).

```
<script>
    let count = $state(0);
    let milliseconds = $state(1000);

    $effect(() => {
        // This will be recreated whenever `milliseconds` changes
        const interval = setInterval(() => {
            count += 1;
        }, milliseconds);

        return () => {
            // if a callback is provided, it will run
            // a) immediately before the effect re-runs
            // b) when the component is destroyed
            clearInterval(interval);
        };
    });
</script>

<h1>{count}</h1>

<button onclick={() => (milliseconds *= 2)}>slower</button>
<button onclick={() => (milliseconds /= 2)}>faster</button>
```

`$effect` dependencies

`$effect` automatically picks up any reactivy values (`$state`, `$derived`, `$props`) that are *synchronously* read inside its function body and registers them as dependencies. When those dependencies change, the `$effect` schedules a rerun.

Values that are read asynchronously — after an `await` or inside a `setTimeout`, for example — will *not* be tracked. Here, the canvas will be repainted when `color` changes, but not when `size` changes (demo):

```
// @filename: index.ts
declare let canvas: {
    width: number;
    height: number;
    getContext(type: '2d', options?: CanvasRenderingContext2DSettings):
CanvasRenderingContext2D;
};
declare let color: string;
declare let size: number;

// cut---
$effect(() => {
    const context = canvas.getContext('2d');
    context.clearRect(0, 0, canvas.width, canvas.height);

    // this will re-run whenever `color` changes...
    context.fillStyle = color;

    setTimeout(() => {
        // ...but not when `size` changes
        context.fillRect(0, 0, size, size);
    }, 0);
});
```

An effect only reruns when the object it reads changes, not when a property inside it changes. (If you want to observe changes *inside* an object at dev time, you can use `$inspect`.)

```
<script>
    let state = $state({ value: 0 });
    let derived = $derived({ value: state.value * 2 });

    // this will run once, because `state` is never reassigned (only mutated)
    $effect(() => {
        state;
    });
    // this will run whenever `state.value` changes...
    $effect(() => {
        state.value;
    });

    // ...and so will this, because `derived` is a new object each time
    $effect(() => {
        derived;
    });
</script>

<button onclick={() => (state.value += 1)}>
    {state.value}
</button>

<p>{state.value} doubled is {derived.value}</p>
```

When not to use `$effect`

In general, `$effect` is best considered something of an escape hatch — useful for things like analytics and direct DOM manipulation — rather than a tool you should use frequently. In particular, avoid using it to synchronise state. Instead of this...

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

    // don't do this!
    $effect(() => {
        doubled = count * 2;
    });
</script>
```

...do this:

```
<script>
    let count = $state(0);
    let doubled = $derived(count * 2);
</script>
```

For things that are more complicated than a simple expression like `count * 2`, you can also use `$derived.by`.

You might be tempted to do something convoluted with effects to link one value to another. The following example shows two inputs for "money spent" and "money left" that are connected to each other. If you update one, the other should update accordingly. Don't use effects for this (demo):

```
<script>
    let total = 100;
    let spent = $state(0);
    let left = $state(total);

    $effect(() => {
        left = total - spent;
    });

    $effect(() => {
        spent = total - left;
    });
</script>

<label>
    <input type="range" bind:value={spent} max={total} />
    {spent}/{total} spent
</label>

<label>
    <input type="range" bind:value={left} max={total} />
    {left}/{total} left
</label>
```

Instead, use callbacks where possible (demo):

```
<script>
    let total = 100;
    let spent = $state(0);
    let left = $state(total);

    function updateSpent(e) {
        spent = +e.target.value;
        left = total - spent;
    }

    function updateLeft(e) {
        left = +e.target.value;
        spent = total - left;
    }
</script>

<label>
    <input type="range" value={spent} oninput={updateSpent} max={total} />
    {spent}/{total} spent
</label>

<label>
    <input type="range" value={left} oninput={updateLeft} max={total} />
    {left}/{total} left
</label>
```

If you need to use bindings, for whatever reason (for example when you want some kind of "writable `$derived`"), consider using getters and setters to synchronise state (demo):

```
<script>
    let total = 100;
    let spent = $state(0);

    let left = {
        get value() {
            return total - spent;
        },
        set value(v) {
            spent = total - v;
        }
    };
</script>

<label>
    <input type="range" bind:value={spent} max={total} />
    {spent}/{total} spent
</label>

<label>
    <input type="range" bind:value={left.value} max={total} />
    {left.value}/{total} left
</label>
```

If you absolutely have to update `$state` within an effect and run into an infinite loop because you read and write to the same `$state`, use untrack.

`$effect.pre`

In rare cases, you may need to run code *before* the DOM updates. For this we can use the `$effect.pre` rune:

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

    let div = $state();
    let messages = $state([]);

    // ...

    $effect.pre(() => {
        if (!div) return; // not yet mounted

        // reference `messages` array length so that this code re-runs whenever it
changes
        messages.length;

        // autoscroll when new messages are added
        if (div.offsetHeight + div.scrollTop > div.scrollHeight - 20) {
            tick().then(() => {
                div.scrollTo(0, div.scrollHeight);
            });
        }
    });
</script>

<div bind:this={div}>
    {#each messages as message}
        <p>{message}</p>
    {/each}
</div>
```

Apart from the timing, `$effect.pre` works exactly like `$effect` — refer to its documentation for more info.

`$effect.tracking`

The `$effect.tracking` rune is an advanced feature that tells you whether or not the code is running inside a tracking context, such as an effect or inside your template (demo):

```
<script>
    console.log('in component setup:', $effect.tracking()); // false

    $effect(() => {
        console.log('in effect:', $effect.tracking()); // true
    });
</script>

<p>in template: {$effect.tracking()}</p> <!-- true -->
```

This allows you to (for example) add things like subscriptions without causing memory leaks, by putting them in child effects. Here's a `readable` function that listens to changes from a callback function as long as it's inside a tracking context:

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

export default function readable<T>(
    initial_value: T,
    start: (callback: (update: (v: T) => T) => T) => () => void
) {
    let value = $state(initial_value);

    let subscribers = 0;
    let stop: null | (() => void) = null;

    return {
        get value() {
            // If in a tracking context ...
            if ($effect.tracking()) {
                $effect(() => {
                    // ...and there's no subscribers yet...
                    if (subscribers === 0) {
                        // ...invoke the function and listen to changes to update
state
                        stop = start((fn) => (value = fn(value)));
                    }

                    subscribers++;

                    // The return callback is called once a listener unlistens
                    return () => {
                        tick().then(() => {
                            subscribers--;
                            // If it was the last subscriber...
                            if (subscribers === 0) {
                                // ...stop listening to changes
                                stop?.();
                                stop = null;
                            }
                        });
                    };
                });
            }
            return value;
        }
    };
}
```

`$effect.root`

The `$effect.root` rune is an advanced feature that creates a non-tracked scope that doesn't auto-cleanup. This is useful for nested effects that you want to manually control. This rune also allows for creation of effects outside of the component initialisation phase.

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

    const cleanup = $effect.root(() => {
        $effect(() => {
            console.log(count);
        });

        return () => {
            console.log('effect root cleanup');
        };
    });
</script>
```

Go to TOC

Runes

Go to TOC

Stores

- how to use
- how to write
- TODO should the details for the store methods belong to the reference section?

A *store* is an object that allows reactive access to a value via a simple *store contract*. The `svelte/store` module contains minimal store implementations which fulfil this contract.

Any time you have a reference to a store, you can access its value inside a component by prefixing it with the `$` character. This causes Svelte to declare the prefixed variable, subscribe to the store at component initialisation and unsubscribe when appropriate.

Assignments to `$`-prefixed variables require that the variable be a writable store, and will result in a call to the store's `.set` method.

Note that the store must be declared at the top level of the component — not inside an `if` block or a function, for example.

Local variables (that do not represent store values) must *not* have a `$` prefix.

```
<script>
    import { writable } from 'svelte/store';

    const count = writable(0);
    console.log($count); // logs 0

    count.set(1);
    console.log($count); // logs 1

    $count = 2;
    console.log($count); // logs 2
</script>
```

When to use stores

Prior to Svelte 5, stores were the go-to solution for creating cross-component reactive states or extracting logic. With runes, these use cases have greatly diminished.

- when extracting logic, it's better to take advantage of runes' universal reactivity: You can use runes outside the top level of components and even place them into JavaScript or TypeScript files (using a `.svelte.js` or `.svelte.ts` file ending)
- when creating shared state, you can create a `$state` object containing the values you need and manipulating said state

Stores are still a good solution when you have complex asynchronous data streams or it's important to have more manual control over updating values or listening to changes. If you're familiar with RxJs and want to reuse that knowledge, the `$` also comes in handy for you.

svelte/store

The `svelte/store` module contains a minimal store implementation which fulfil the store contract. It provides methods for creating stores that you can update from the outside, stores you can only update from the inside, and for combining and deriving stores.

`writable`

Function that creates a store which has values that can be set from 'outside' components. It gets created as an object with additional `set` and `update` methods.

`set` is a method that takes one argument which is the value to be set. The store value gets set to the value of the argument if the store value is not already equal to it.

`update` is a method that takes one argument which is a callback. The callback takes the existing store value as its argument and returns the new value to be set to the store.

```js
/// file: store.js
import { writable } from 'svelte/store';

const count = writable(0);

count.subscribe((value) => {
    console.log(value);
}); // logs '0'

count.set(1); // logs '1'

count.update((n) => n + 1); // logs '2'
```

If a function is passed as the second argument, it will be called when the number of subscribers goes from zero to one (but not from one to two, etc). That function will be passed a `set` function which changes the value of the store, and an `update` function which works like the `update` method on the store, taking a callback to calculate the store's new value from its old value. It must return a `stop` function that is called when the subscriber count goes from one to zero.

```js
/// file: store.js
import { writable } from 'svelte/store';

const count = writable(0, () => {
    console.log('got a subscriber');
    return () => console.log('no more subscribers');
});

count.set(1); // does nothing

const unsubscribe = count.subscribe((value) => {
    console.log(value);
}); // logs 'got a subscriber', then '1'

unsubscribe(); // logs 'no more subscribers'
```

Note that the value of a `writable` is lost when it is destroyed, for example when the page is refreshed. However, you can write your own logic to sync the value to for example the `localStorage`.

`readable`

Creates a store whose value cannot be set from 'outside', the first argument is the store's initial value, and the second argument to `readable` is the same as the second argument to `writable`.

```
import { readable } from 'svelte/store';

const time = readable(new Date(), (set) => {
    set(new Date());

    const interval = setInterval(() => {
        set(new Date());
    }, 1000);

    return () => clearInterval(interval);
});

const ticktock = readable('tick', (set, update) => {
    const interval = setInterval(() => {
        update((sound) => (sound === 'tick' ? 'tock' : 'tick'));
    }, 1000);

    return () => clearInterval(interval);
});
```

`derived`

Derives a store from one or more other stores. The callback runs initially when the first subscriber subscribes and then whenever the store dependencies change.

In the simplest version, `derived` takes a single store, and the callback returns a derived value.

```
// @filename: ambient.d.ts
import { type Writable } from 'svelte/store';

declare global {
    const a: Writable<number>;
}

export {};

// @filename: index.ts
// cut---
import { derived } from 'svelte/store';

const doubled = derived(a, ($a) => $a * 2);
```

The callback can set a value asynchronously by accepting a second argument, `set`, and an optional third argument, `update`, calling either or both of them when appropriate.

In this case, you can also pass a third argument to `derived` — the initial value of the derived store before `set` or `update` is first called. If no initial value is specified, the store's initial value will be `undefined`.

```
// @filename: ambient.d.ts
import { type Writable } from 'svelte/store';

declare global {
    const a: Writable<number>;
}

export {};

// @filename: index.ts
// @errors: 18046 2769 7006
// cut---
import { derived } from 'svelte/store';

const delayed = derived(
    a,
    ($a, set) => {
        setTimeout(() => set($a), 1000);
    },
    2000
);

const delayedIncrement = derived(a, ($a, set, update) => {
    set($a);
    setTimeout(() => update((x) => x + 1), 1000);
    // every time $a produces a value, this produces two
    // values, $a immediately and then $a + 1 a second later
});
```

If you return a function from the callback, it will be called when a) the callback runs again, or b) the last subscriber unsubscribes.

```
// @filename: ambient.d.ts
import { type Writable } from 'svelte/store';

declare global {
    const frequency: Writable<number>;
}

export {};

// @filename: index.ts
// cut---
import { derived } from 'svelte/store';

const tick = derived(
    frequency,
    ($frequency, set) => {
        const interval = setInterval(() => {
            set(Date.now());
        }, 1000 / $frequency);

        return () => {
            clearInterval(interval);
        };
```

```
	},
	2000
);
```

In both cases, an array of arguments can be passed as the first argument instead of a single store.

```
// @filename: ambient.d.ts
import { type Writable } from 'svelte/store';

declare global {
	const a: Writable<number>;
	const b: Writable<number>;
}

export {};

// @filename: index.ts

// cut---
import { derived } from 'svelte/store';

const summed = derived([a, b], ([$a, $b]) => $a + $b);

const delayed = derived([a, b], ([$a, $b], set) => {
	setTimeout(() => set($a + $b), 1000);
});
```

`readonly`

This simple helper function makes a store readonly. You can still subscribe to the changes from the original one using this new readable store.

```
import { readonly, writable } from 'svelte/store';

const writableStore = writable(1);
const readableStore = readonly(writableStore);

readableStore.subscribe(console.log);

writableStore.set(2); // console: 2
// @errors: 2339
readableStore.set(2); // ERROR
```

`get`

Generally, you should read the value of a store by subscribing to it and using the value as it changes over time. Occasionally, you may need to retrieve the value of a store to which you're not subscribed. `get` allows you to do so.

> This works by creating a subscription, reading the value, then unsubscribing. It's therefore not recommended in hot code paths.

```
// @filename: ambient.d.ts
import { type Writable } from 'svelte/store';

declare global {
    const store: Writable<string>;
}

export {};

// @filename: index.ts
// cut---
import { get } from 'svelte/store';

const value = get(store);
```

Store contract

```
// @noErrors
store = { subscribe: (subscription: (value: any) => void) => (() => void), set?:
(value: any) => void }
```

You can create your own stores without relying on `svelte/store`, by implementing the *store contract*:

1. A store must contain a `.subscribe` method, which must accept as its argument a subscription function. This subscription function must be immediately and synchronously called with the store's current value upon calling `.subscribe`. All of a store's active subscription functions must later be synchronously called whenever the store's value changes.
2. The `.subscribe` method must return an unsubscribe function. Calling an unsubscribe function must stop its subscription, and its corresponding subscription function must not be called again by the store.
3. A store may *optionally* contain a `.set` method, which must accept as its argument a new value for the store, and which synchronously calls all of the store's active subscription functions. Such a store is called a *writable store*.

For interoperability with RxJS Observables, the `.subscribe` method is also allowed to return an object with an `.unsubscribe` method, rather than return the unsubscription function directly. Note however that unless `.subscribe` synchronously calls the subscription (which is not required by the Observable spec), Svelte will see the value of the store as `undefined` until it does.

Go to TOC

Context

- get/set/hasContext
- how to use, best practises (like encapsulating them)

Most state is component-level state that lives as long as its component lives. There's also section-wide or app-wide state however, which also needs to be handled somehow.

The easiest way to do that is to create global state and just import that.

```
/// file: state.svelte.ts

export const myGlobalState = $state({
    user: {
        /* ... */
    }
    /* ... */
});
```

```
<! file: App.svelte --->
<script>
    import { myGlobalState } from './state.svelte';
    // ...
</script>
```

This has a few drawbacks though:

- it only safely works when your global state is only used client-side - for example, when you're building a single page application that does not render any of your components on the server. If your state ends up being managed and updated on the server, it could end up being shared between sessions and/or users, causing bugs
- it may give the false impression that certain state is global when in reality it should only used in a certain part of your app

To solve these drawbacks, Svelte provides a few `context` primitives which alleviate these problems.

Setting and getting context

To associate an arbitrary object with the current component, use `setContext`.

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

    setContext('key', value);
</script>
```

The context is then available to children of the component (including slotted content) with `getContext`.

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

    const value = getContext('key');
</script>
```

`setContext` and `getContext` solve the above problems:

- the state is not global, it's scoped to the component. That way it's safe to render your components on the server and not leak state
- it's clear that the state is not global but rather scoped to a specific component tree and therefore can't be used in other parts of your app

> `setContext` / `getContext` must be called during component initialisation.

Context is not inherently reactive. If you need reactive values in context then you can pass a `$state` object into context, whos properties *will* be reactive.

```
<! file: Parent.svelte --->
<script>
    import { setContext } from 'svelte';

    let value = $state({ count: 0 });
    setContext('counter', value);
</script>

<button onclick={() => value.count++}>increment</button>
```

```
<! file: Child.svelte --->
<script>
    import { setContext } from 'svelte';

    const value = setContext('counter');
</script>

<p>Count is {value.count}</p>
```

To check whether a given `key` has been set in the context of a parent component, use `hasContext`.

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

    if (hasContext('key')) {
        // do something
    }
</script>
```

You can also retrieve the whole context map that belongs to the closest parent component using `getAllContexts`. This is useful, for example, if you programmatically create a component and want to pass the existing context to it.

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

    const contexts = getAllContexts();
</script>
```

Encapsulating context interactions

The above methods are very unopionated about how to use them. When your app grows in scale, it's worthwhile to encapsulate setting and getting the context into functions and properly type them.

```
// @errors: 2304
import { getContext, setContext } from 'svelte';

let userKey = Symbol('user');

export function setUserContext(user: User) {
    setContext(userKey, user);
}

export function getUserContext(): User {
    return getContext(userKey) as User;
}
```

Go to TOC

Lifecycle hooks

- onMount/onDestroy
- mention that `$effect` might be better for your use case
- beforeUpdate/afterUpdate with deprecation notice?
- or skip this entirely and only have it in the reference docs?

In Svelte 5, the component lifecycle consists of only two parts: Its creation and its destruction. Everything in-between - when certain state is updated - is not related to the component as a whole, only the parts that need to react to the state change are notified. This is because under the hood the smallest unit of change is actually not a component, it's the (render) effects that the component sets up upon component initialization. Consequently, there's no such thing as a "before update"/"after update" hook.

`onMount`

The `onMount` function schedules a callback to run as soon as the component has been mounted to the DOM. It must be called during the component's initialisation (but doesn't need to live *inside* the component; it can be called from an external module).

`onMount` does not run inside a component that is rendered on the server.

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

    onMount(() => {
        console.log('the component has mounted');
    });
</script>
```

If a function is returned from `onMount`, it will be called when the component is unmounted.

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

    onMount(() => {
        const interval = setInterval(() => {
            console.log('beep');
        }, 1000);

        return () => clearInterval(interval);
    });
</script>
```

> This behaviour will only work when the function passed to `onMount` *synchronously* returns a value. `async` functions always return a `Promise`, and as such cannot *synchronously* return a function.

`onDestroy`

```
EXPORT_SNIPPET: svelte#onDestroy
```

Schedules a callback to run immediately before the component is unmounted.

Out of `onMount`, `beforeUpdate`, `afterUpdate` and `onDestroy`, this is the only one that runs inside a server-side component.

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

    onDestroy(() => {
        console.log('the component is being destroyed');
    });
</script>
```

`tick`

While there's no "after update" hook, you can use `tick` to ensure that the UI is updated before continuing. `tick` returns a promise that resolves once any pending state changes have been applied, or in the next microtask if there are none.

```
<script>
    import { beforeUpdate, tick } from 'svelte';

    beforeUpdate(async () => {
        console.log('the component is about to update');
        await tick();
        console.log('the component just updated');
    });
</script>
```

Deprecated: `beforeUpdate` / `afterUpdate`

Svelte 4 contained hooks that ran before and after the component as a whole was updated. For backwards compatibility, these hooks were shimmed in Svelte 5 but not available inside components that use runes.

```
<script>
    import { beforeUpdate, afterUpdate } from 'svelte';

    beforeUpdate(() => {
        console.log('the component is about to update');
    });

    afterUpdate(() => {
        console.log('the component just updated');
    });
</script>
```

Instead of `beforeUpdate` use `$effect.pre` and instead of `afterUpdate` use `$effect` instead - these runes offer more granular control and only react to the changes you're actually interested in.

Chat window example

To implement a chat window that autoscrolls to the bottom when new messages appear (but only if you were *already* scrolled to the bottom), we need to measure the DOM before we update it.

In Svelte 4, we do this with `beforeUpdate`, but this is a flawed approach — it fires before *every* update, whether it's relevant or not. In the example below, we need to introduce checks like `updatingMessages` to make sure we don't mess with the scroll position when someone toggles dark mode.

With runes, we can use `$effect.pre`, which behaves the same as `$effect` but runs before the DOM is updated. As long as we explicitly reference `messages` inside the effect body, it will run whenever `messages` changes, but *not* when `theme` changes.

`beforeUpdate`, and its equally troublesome counterpart `afterUpdate`, are therefore deprecated in Svelte 5.

- Before
- After

```
<script>
-   import { beforeUpdate, afterUpdate, tick } from 'svelte';
+   import { tick } from 'svelte';

-   let updatingMessages = false;
-   let theme = 'dark';
-   let messages = [];
+   let theme = $state('dark');
+   let messages = $state([]);

    let viewport;

-   beforeUpdate(() => {
+   $effect.pre(() => {
-       if (!updatingMessages) return;
+       messages;
        const autoscroll = viewport && viewport.offsetHeight + viewport.scrollTop
> viewport.scrollHeight - 50;

        if (autoscroll) {
            tick().then(() => {
                viewport.scrollTo(0, viewport.scrollHeight);
            });
        }

-       updatingMessages = false;
    });

    function handleKeydown(event) {
        if (event.key === 'Enter') {
            const text = event.target.value;
            if (!text) return;
-           updatingMessages = true;
            messages = [...messages, text];
            event.target.value = '';
        }
    }

    function toggle() {
        toggleValue = !toggleValue;
    }
</script>

<div class:dark={theme === 'dark'}>
    <div bind:this={viewport}>
        {#each messages as message}
            <p>{message}</p>
        {/each}
    </div>

-   <input on:keydown={handleKeydown} />
+   <input onkeydown={handleKeydown} />

-   <button on:click={toggle}>
+   <button onclick={toggle}>
        Toggle dark mode
    </button>
</div>
```

Go to TOC

Imperative component API

better title needed?

- mount
- unmount
- render
- hydrate
- how they interact with each other

Every Svelte application starts by imperatively creating a root component. On the client this component is mounted to a specific element. On the server, you want to get back a string of HTML instead which you can render. The following functions help you achieve those tasks.

`mount`

Instantiates a component and mounts it to the given target:

```
// @errors: 2322
import { mount } from 'svelte';
import App from './App.svelte';

const app = mount(App, {
	target: document.querySelector('#app'),
	props: { some: 'property' }
});
```

You can mount multiple components per page, and you can also mount from within your application, for example when creating a tooltip component and attaching it to the hovered element.

Note that unlike calling `new App(...)` in Svelte 4, things like effects (including `onMount` callbacks, and action functions) will not run during `mount`. If you need to force pending effects to run (in the context of a test, for example) you can do so with `flushSync()`.

`unmount`

Unmounts a component created with `mount` or `hydrate`:

```
// @errors: 1109
import { mount, unmount } from 'svelte';
import App from './App.svelte';

const app = mount(App, {...});

// later
unmount(app);
```

`render`

Only available on the server and when compiling with the `server` option. Takes a component and returns an object with `body` and `head` properties on it, which you can use to populate the HTML when server-rendering your app:

```
// @errors: 2724 2305 2307
import { render } from 'svelte/server';
import App from './App.svelte';

const result = render(App, {
    props: { some: 'property' }
});
result.body; // HTML for somewhere in this <body> tag
result.head; // HTML for somewhere in this <head> tag
```

`hydrate`

Like `mount`, but will reuse up any HTML rendered by Svelte's SSR output (from the `render` function) inside the target and make it interactive:

```
// @errors: 2322
import { hydrate } from 'svelte';
import App from './App.svelte';

const app = hydrate(App, {
    target: document.querySelector('#app'),
    props: { some: 'property' }
});
```

As with `mount`, effects will not run during `hydrate` — use `flushSync()` immediately afterwards if you need them to.

Go to TOC

Runtime

Go to TOC

Debugging

- `@debug`
- `$inspect`

Svelte provides two built-in ways to debug your application.

`$inspect`

The `$inspect` rune is roughly equivalent to `console.log`, with the exception that it will re-run whenever its argument changes. `$inspect` tracks reactive state deeply, meaning that updating something inside an object or array using fine-grained reactivity will cause it to re-fire. (Demo:)

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

    $inspect(count, message); // will console.log when `count` or `message` change
</script>

<button onclick={() => count++}>Increment</button>
<input bind:value={message} />
```

`$inspect` returns a property `with`, which you can invoke with a callback, which will then be invoked instead of `console.log`. The first argument to the callback is either `"init"` or `"update"`, all following arguments are the values passed to `$inspect`. Demo:

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

    $inspect(count).with((type, count) => {
        if (type === 'update') {
            debugger; // or `console.trace`, or whatever you want
        }
    });
</script>

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

A convenient way to find the origin of some change is to pass `console.trace` to `with`:

```
// @errors: 2304
$inspect(stuff).with(console.trace);
```

> `$inspect` only works during development. In a production build it becomes a noop.

@debug

```
<! copy: false --->
{@debug}
```

```
<! copy: false --->
{@debug var1, var2, ..., varN}
```

The `{@debug ...}` tag offers an alternative to `console.log(...)`. It logs the values of specific variables whenever they change, and pauses code execution if you have devtools open.

```
<script>
    let user = {
        firstname: 'Ada',
        lastname: 'Lovelace'
    };
</script>

{@debug user}

<h1>Hello {user.firstname}!</h1>
```

`{@debug ...}` accepts a comma-separated list of variable names (not arbitrary expressions).

```
<!-- Compiles -->
{@debug user}
{@debug user1, user2, user3}

<!-- WON'T compile -->
{@debug user.firstname}
{@debug myArray[0]}
{@debug !isReady}
{@debug typeof user === 'object'}
```

The `{@debug}` tag without any arguments will insert a `debugger` statement that gets triggered when *any* state changes, as opposed to the specified variables.

Go to TOC

Testing

Testing helps you write and maintain your code and guard against regressions. Testing frameworks help you with that, allowing you to describe assertions or expectations about how your code should behave. Svelte is unopinionated about which testing framework you use — you can write unit tests, integration tests, and end-to-end tests using solutions like Vitest, Jasmine, Cypress and Playwright.

Unit and integration testing using Vitest

Unit tests allow you to test small isolated parts of your code. Integration tests allow you to test parts of your application to see if they work together. If you're using Vite (including via SvelteKit), we recommend using Vitest.

To get started, install Vitest:

```
npm install -D vitest
```

Then adjust your `vite.config.js`:

```
/// file: vite.config.js
- import { defineConfig } from 'vite';
+ import { defineConfig } from 'vitest/config';

export default defineConfig({ /* ... */ })
```

You can now write unit tests for code inside your `.js/.ts` files:

```
/// file: multiplier.svelte.test.js
import { flushSync } from 'svelte';
import { expect, test } from 'vitest';
import { multiplier } from './multiplier.js';

test('Multiplier', () => {
    let double = multiplier(0, 2);

    expect(double.value).toEqual(0);

    double.set(5);

    expect(double.value).toEqual(10);
});
```

Using runes inside your test files

It is possible to use runes inside your test files. First ensure your bundler knows to route the file through the Svelte compiler before running the test by adding `.svelte` to the filename (e.g `multiplier.svelte.test.js`). After that, you can use runes inside your tests.

```js
/// file: multiplier.svelte.test.js
import { flushSync } from 'svelte';
import { expect, test } from 'vitest';
import { multiplier } from './multiplier.svelte.js';

test('Multiplier', () => {
    let count = $state(0);
    let double = multiplier(() => count, 2);

    expect(double.value).toEqual(0);

    count = 5;

    expect(double.value).toEqual(10);
});
```

If the code being tested uses effects, you need to wrap the test inside `$effect.root`:

```js
/// file: logger.svelte.test.js
import { flushSync } from 'svelte';
import { expect, test } from 'vitest';
import { logger } from './logger.svelte.js';

test('Effect', () => {
    const cleanup = $effect.root(() => {
        let count = $state(0);

        // logger uses an $effect to log updates of its input
        let log = logger(() => count);

        // effects normally run after a microtask,
        // use flushSync to execute all pending effects synchronously
        flushSync();
        expect(log.value).toEqual([0]);

        count = 1;
        flushSync();

        expect(log.value).toEqual([0, 1]);
    });

    cleanup();
});
```

Component testing

It is possible to test your components in isolation using Vitest.

> Before writing component tests, think about whether you actually need to test the component, or if it's more about the logic *inside* the component. If so, consider extracting out that logic to test it in isolation, without the overhead of a component

To get started, install jsdom (a library that shims DOM APIs):

```
npm install -D jsdom
```

Then adjust your `vite.config.js`:

```js
/// file: vite.config.js
import { defineConfig } from 'vitest/config';

export default defineConfig({
    plugins: [
        /* ... */
    ],
    test: {
        // If you are testing components client-side, you need to setup a DOM
environment.
        // If not all your files should have this environment, you can use a
        // `// @vitest-environment jsdom` comment at the top of the test files
instead.
        environment: 'jsdom'
    },
    // Tell Vitest to use the `browser` entry points in `package.json` files, even
though it's running in Node
    resolve: process.env.VITEST
        ? {
                conditions: ['browser']
          }
        : undefined
});
```

After that, you can create a test file in which you import the component to test, interact with it programmatically and write expectations about the results:

```js
/// file: component.test.js
import { flushSync, mount, unmount } from 'svelte';
import { expect, test } from 'vitest';
import Component from './Component.svelte';

test('Component', () => {
    // Instantiate the component using Svelte's `mount` API
    const component = mount(Component, {
        target: document.body, // `document` exists because of jsdom
        props: { initial: 0 }
    });

    expect(document.body.innerHTML).toBe('<button>0</button>');

    // Click the button, then flush the changes so you can synchronously write
expectations
    document.body.querySelector('button').click();
    flushSync();

    expect(document.body.innerHTML).toBe('<button>1</button>');

    // Remove the component from the DOM
    unmount(component);
});
```

While the process is very straightforward, it is also low level and somewhat brittle, as the precise structure of your component may change frequently. Tools like @testing-library/svelte can help streamline your tests. The above test could be rewritten like this:

```
/// file: component.test.js
import { render, screen } from '@testing-library/svelte';
import userEvent from '@testing-library/user-event';
import { expect, test } from 'vitest';
import Component from './Component.svelte';

test('Component', async () => {
    const user = userEvent.setup();
    render(Component);

    const button = screen.getByRole('button');
    expect(button).toHaveTextContent(0);

    await user.click(button);
    expect(button).toHaveTextContent(1);
});
```

When writing component tests that involve two-way bindings, context or snippet props, it's best to create a wrapper component for your specific test and interact with that. `@testing-library/svelte` contains some examples.

E2E tests using Playwright

E2E (short for 'end to end') tests allow you to test your full application through the eyes of the user. This section uses Playwright as an example, but you can also use other solutions like Cypress or NightwatchJS.

To get start with Playwright, either let you guide by their VS Code extension, or install it from the command line using `npm init playwright`. It is also part of the setup CLI when you run `npm create svelte`.

After you've done that, you should have a `tests` folder and a Playwright config. You may need to adjust that config to tell Playwright what to do before running the tests - mainly starting your application at a certain port:

```
/// file: playwright.config.js
const config = {
    webServer: {
        command: 'npm run build && npm run preview',
        port: 4173
    },
    testDir: 'tests',
    testMatch: /(.+\.)?(test|spec)\.[jt]s/
};

export default config;
```

You can now start writing tests. These are totally unaware of Svelte as a framework, so you mainly interact with the DOM and write assertions.

```
/// file: tests/hello-world.spec.js
import { expect, test } from '@playwright/test';

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

TypeScript

- basically what we have today
- built-in support, but only for type-only features
- generics
- using `Component` and the other helper types
- using `svelte-check`

You can use TypeScript within Svelte components. IDE extensions like the Svelte VS Code extension will help you catch errors right in your editor, and `svelte-check` does the same on the command line, which you can integrate into your CI.

`<script lang="ts">`

To use TypeScript inside your Svelte components, add `lang="ts"` to your `script` tags:

```
<script lang="ts">
    let name: string = 'world';

    function greet(name: string) {
        alert(`Hello, ${name}!`);
    }
</script>

<button onclick={(e: Event) => greet(e.target.innerText)}>
    {name as string}
</button>
```

Doing so allows you to use TypeScript's *type-only* features. That is, all features that just disappear when transpiling to JavaScript, such as type annotations or interface declarations. Features that require the TypeScript compiler to output actual code are not supported. This includes:

- using enums
- using `private`, `protected` or `public` modifiers in constructor functions together with initializers
- using features that are not yet part of the ECMAScript standard (i.e. not level 4 in the TC39 process) and therefore not implemented yet within Acorn, the parser we use for parsing JavaScript

If you want to use one of these features, you need to setup up a `script` preprocessor.

Preprocessor setup

To use non-type-only TypeScript features within Svelte components, you need to add a preprocessor that will turn TypeScript into JavaScript.

Using SvelteKit or Vite

The easiest way to get started is scaffolding a new SvelteKit project by typing `npm create svelte@latest`, following the prompts and choosing the TypeScript option.

```js
/// file: svelte.config.js
// @noErrors
import { vitePreprocess } from '@sveltejs/kit/vite';

const config = {
    preprocess: vitePreprocess()
};

export default config;
```

If you don't need or want all the features SvelteKit has to offer, you can scaffold a Svelte-flavoured Vite project instead by typing `npm create vite@latest` and selecting the `svelte-ts` option.

```js
/// file: svelte.config.js
import { vitePreprocess } from '@sveltejs/vite-plugin-svelte';

const config = {
    preprocess: vitePreprocess()
};

export default config;
```

In both cases, a `svelte.config.js` with `vitePreprocess` will be added. Vite/SvelteKit will read from this config file.

Other build tools

If you're using tools like Rollup or Webpack instead, install their respective Svelte plugins. For Rollup that's rollup-plugin-svelte and for Webpack that's svelte-loader. For both, you need to install `typescript` and `svelte-preprocess` and add the preprocessor to the plugin config (see the respective READMEs for more info). If you're starting a new project, you can also use the rollup or webpack template to scaffold the setup from a script.

> If you're starting a new project, we recommend using SvelteKit or Vite instead

Typing `$props`

Type `$props` just like a regular object with certain properties.

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

    interface Props {
        requiredProperty: number;
        optionalProperty?: boolean;
        snippetWithStringArgument: Snippet<[string]>;
        eventHandler: (arg: string) => void;
        [key: string]: unknown;
    }

    let {
        requiredProperty,
        optionalProperty,
        snippetWithStringArgument,
        eventHandler,
        ...everythingElse
    }: Props = $props();
</script>

<button onclick={() => eventHandler('clicked button')}>
    {@render snippetWithStringArgument('hello')}
</button>
```

Generic `$props`

Components can declare a generic relationship between their properties. One example is a generic list component that receives a list of items and a callback property that reveives an item from the list. To declare that the `items` property and the `select` callback operate on the same types, add the `generics` attribute to the `script` tag:

```
<script lang="ts" generics="Item extends { text: string }">
    interface Props {
        items: Item[];
        select: Item;
    }

    let { items, select } = $props();
</script>

{#each items as item}
    <button onclick={() => select(item)}>
        {item.text}
    </button>
{/each}
```

The content of `generics` is what you would put between the `<...>` tags of a generic function. In other words, you can use multiple generics, `extends` and fallback types.

Typing `$state`

You can type `$state` like any other variable.

```
let count: number = $state(0);
```

If you don't give `$state` an initial value, part of its types will be `undefined`.

```
// @noErrors
// Error: Type 'number | undefined' is not assignable to type 'number'
let count: number = $state();
```

If you know that the variable *will* be defined before you first use it, use an `as` casting. This is especially useful in the context of classes:

```
class Counter {
    count = $state() as number;
    constructor(initial: number) {
        this.count = initial;
    }
}
```

The `Component` type

Svelte components or of type `Component`. You can use it and its related types to express a variety of constraints.

Using it together with `<svelte:component>` to restrict what kinds of component can be passed to it:

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

    interface Props {
        // only components that have at most the "prop"
        // property required can be passed
        component: Component<{ prop: string }>
    }

    let { component }: Props = $props();
</script>

<svelte:component this={component} prop="foo" />
```

Closely related to the `Component` type is the `ComponentProps` type which extracts the properties a component expects.

```
import type { Component, ComponentProps } from 'svelte';
import MyComponent from './MyComponent.svelte';

function withProps<TComponent extends Component<any>>(
    component: TComponent,
    props: ComponentProps<TComponent>
) {}

// Errors if the second argument is not the correct props expected
// by the component in the first argument.
withProps(MyComponent, { foo: 'bar' });
```

Enhancing built-in DOM types

Svelte provides a best effort of all the HTML DOM types that exist. Sometimes you may want to use experimental attributes or custom events coming from an action. In these cases, TypeScript will throw a type error, saying that it does not know these types. If it's a non-experimental standard attribute/event, this may very well be a missing typing from our HTML typings. In that case, you are welcome to open an issue and/or a PR fixing it.

In case this is a custom or experimental attribute/event, you can enhance the typings like this:

```
/// file: additional-svelte-typings.d.ts
declare namespace svelteHTML {
    // enhance elements
    interface IntrinsicElements {
        'my-custom-element': { someattribute: string; 'on:event': (e:
CustomEvent<any>) => void };
    }
    // enhance attributes
    interface HTMLAttributes<T> {
        // If you want to use on:beforeinstallprompt
        'on:beforeinstallprompt'?: (event: any) => any;
        // If you want to use myCustomAttribute={..} (note: all lowercase)
        mycustomattribute?: any; // You can replace any with something more
specific if you like
    }
}
```

Then make sure that `d.ts` file is referenced in your `tsconfig.json`. If it reads something like `"include": ["src/**/*"]` and your `d.ts` file is inside `src`, it should work. You may need to reload for the changes to take effect.

You can also declare the typings by augmenting the `svelte/elements` module like this:

```
/// file: additional-svelte-typings.d.ts
import { HTMLButtonAttributes } from 'svelte/elements';

declare module 'svelte/elements' {
    export interface SvelteHTMLElements {
        'custom-button': HTMLButtonAttributes;
    }

    // allows for more granular control over what element to add the typings to
    export interface HTMLButtonAttributes {
        veryexperimentalattribute?: string;
    }
}

export {}; // ensure this is not an ambient module, else types will be overridden
instead of augmented
```

Go to TOC

Custom elements

- basically what we have today

Svelte components can also be compiled to custom elements (aka web components) using the `customElement: true` compiler option. You should specify a tag name for the component using the `<svelte:options>` element.

```
<svelte:options customElement="my-element" />

<script>
    let { name = 'world' } = $props();
</script>

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

You can leave out the tag name for any of your inner components which you don't want to expose and use them like regular Svelte components. Consumers of the component can still name it afterwards if needed, using the static `element` property which contains the custom element constructor and which is available when the `customElement` compiler option is `true`.

```
// @noErrors
import MyElement from './MyElement.svelte';

customElements.define('my-element', MyElement.element);
```

Once a custom element has been defined, it can be used as a regular DOM element:

```
document.body.innerHTML = `
    <my-element>
        <p>This is some slotted content</p>
    </my-element>
`;
```

Any props are exposed as properties of the DOM element (as well as being readable/writable as attributes, where possible).

```
// @noErrors
const el = document.querySelector('my-element');

// get the current value of the 'name' prop
console.log(el.name);

// set a new value, updating the shadow DOM
el.name = 'everybody';
```

Component lifecycle

Custom elements are created from Svelte components using a wrapper approach. This means the inner Svelte component has no knowledge that it is a custom element. The custom element wrapper takes care of handling its lifecycle appropriately.

When a custom element is created, the Svelte component it wraps is *not* created right away. It is only created in the next tick after the `connectedCallback` is invoked. Properties assigned to the custom element before it is inserted into the DOM are temporarily saved and then set on component creation, so their values are not lost. The same does not work for invoking exported functions on the custom element though, they are only available after the element has mounted. If you need to invoke functions before component creation, you can work around it by using the `extend` option.

When a custom element written with Svelte is created or updated, the shadow DOM will reflect the value in the next tick, not immediately. This way updates can be batched, and DOM moves which temporarily (but synchronously) detach the element from the DOM don't lead to unmounting the inner component.

The inner Svelte component is destroyed in the next tick after the `disconnectedCallback` is invoked.

Component options

When constructing a custom element, you can tailor several aspects by defining `customElement` as an object within `<svelte:options>` since Svelte 4. This object may contain the following properties:

- `tag`: the mandatory `tag` property for the custom element's name
- `shadow`: an optional property that can be set to `"none"` to forgo shadow root creation. Note that styles are then no longer encapsulated, and you can't use slots
- `props`: an optional property to modify certain details and behaviors of your component's properties. It offers the following settings:
  - `attribute: string`: To update a custom element's prop, you have two alternatives: either set the property on the custom element's reference as illustrated above or use an HTML attribute. For the latter, the default attribute name is the lowercase property name. Modify this by assigning `attribute: "<desired name>"`.
  - `reflect: boolean`: By default, updated prop values do not reflect back to the DOM. To enable this behavior, set `reflect: true`.
  - `type: 'String' | 'Boolean' | 'Number' | 'Array' | 'Object'`: While converting an attribute value to a prop value and reflecting it back, the prop value is assumed to be a `String` by default. This may not always be accurate. For instance, for a number type, define it using `type: "Number"` You don't need to list all properties, those not listed will use the default settings.
- `extend`: an optional property which expects a function as its argument. It is passed the custom element class generated by Svelte and expects you to return a custom element class. This comes in handy if you have very specific requirements to the life cycle of the custom element or want to enhance the class to for example use ElementInternals for better HTML form integration.

```
<svelte:options
    customElement={{
        tag: 'custom-element',
        shadow: 'none',
        props: {
            name: { reflect: true, type: 'Number', attribute: 'element-index' }
        },
        extend: (customElementConstructor) => {
            // Extend the class so we can let it participate in HTML forms
            return class extends customElementConstructor {
                static formAssociated = true;

                constructor() {
                    super();
                    this.attachedInternals = this.attachInternals();
                }

                // Add the function here, not below in the component so that
                // it's always available, not just when the inner Svelte component
                // is mounted
                randomIndex() {
                    this.elementIndex = Math.random();
                }
            };
        }
    }}
/>

<script>
    let { elementIndex, attachedInternals } = $props();
    // ...
    function check() {
        attachedInternals.checkValidity();
    }
</script>

...
```

Caveats and limitations

Custom elements can be a useful way to package components for consumption in a non-Svelte app, as they will work with vanilla HTML and JavaScript as well as most frameworks. There are, however, some important differences to be aware of:

- Styles are *encapsulated*, rather than merely *scoped* (unless you set `shadow: "none"`). This means that any non-component styles (such as you might have in a `global.css` file) will not apply to the custom element, including styles with the `:global(...)` modifier
- Instead of being extracted out as a separate .css file, styles are inlined into the component as a JavaScript string
- Custom elements are not generally suitable for server-side rendering, as the shadow DOM is invisible until JavaScript loads
- In Svelte, slotted content renders *lazily*. In the DOM, it renders *eagerly*. In other words, it will always be created even if the component's `<slot>` element is inside an `{#if ...}` block. Similarly, including a `<slot>` in an `{#each ...}` block will not cause the slotted content to be rendered multiple times

- The deprecated `let:` directive has no effect, because custom elements do not have a way to pass data to the parent component that fills the slot
- Polyfills are required to support older browsers
- You can use Svelte's context feature between regular Svelte components within a custom element, but you can't use them across custom elements. In other words, you can't use `setContext` on a parent custom element and read that with `getContext` in a child custom element.

Go to TOC

Reactivity indepth

- how to think about Runes ("just JavaScript" with added reactivity, what this means for keeping reactivity alive across boundaries)
- signals

Go to TOC

Svelte 5 migration guide

- the stuff from the preview docs and possibly more

Misc

Go to TOC

$state

TODO

- add other pages
- figure out a way to get separator titles in here, so we can have a 'runes' section and an 'imports' section and an 'errors/warnings' section without introducing another layer of nesting
- figure out a good way to import reference docs from other repos that works locally and in prod

svelte

Go to TOC

svelte/action

Go to TOC

svelte/animate

Go to TOC

svelte/compiler

Go to TOC

svelte/easing

Go to TOC

svelte/events

Go to TOC

svelte/legacy

Go to TOC

svelte/motion

Go to TOC

svelte/reactivity

Go to TOC

svelte/server

Go to TOC

svelte/store

Go to TOC

svelte/transition

Go to TOC

Reference

Go to TOC

Colophon

This book is created by using the following sources:

- Svelte - English
- GitHub source: sveltejs/svelte/documentation/docs
- Created: 2024-07-28
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- Viewer: https://read-html-download-pdf.vercel.app/
- GitHub repo: https://github.com/shinokada/markdown-docs-as-pdf
- Viewer repo: https://github.com/shinokada/read-html-download-pdf