

SVELTE Docs - English



Table of contents

• Introduction	3
• Svelte components	4
• Basic markup	11
• Logic blocks	14
• Special tags	18
• Element directives	20
• Component directives	34
• Special elements	37
• Svelte	44
• Svelte store	49
• Svelte motion	54
• Svelte transition	58
• Svelte animate	64
• Svelte easing	66
• Svelte action	67
• Svelte compiler	69
• Client side component api	75
• Server side component api	80
• Custom elements api	81
• Faq	85
• Accessibility warnings	89
• Typescript	96
• V4 migration guide	101
• Svelte register	107

Introduction

Welcome to the Svelte reference documentation! This is intended as a resource for people who already have some familiarity with Svelte and want to learn more about using it.

If that's not you (yet), you may prefer to visit the [interactive tutorial](#) or the [examples](#) before consulting this reference. You can try Svelte online using the [REPL](#). Alternatively, if you'd like a more fully-featured environment, you can try Svelte on [StackBlitz](#).

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.

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 init vite` and selecting the `svelte` option. With this, `npm run build` will generate HTML, JS and CSS files inside the `dist` directory. In most cases, you will probably need to [choose a routing library](#) as well.

Alternatively, there are [plugins for all the major web bundlers](#) to handle Svelte compilation — which will output `.js` and `.css` that you can insert into your HTML — but most others won't handle SSR.

Editor tooling

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

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](#)

Svelte 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 that runs when a component instance is created. Variables declared (or imported) at the top level are 'visible' from the component's markup. There are four additional rules:

1. `export` creates a component prop

Svelte uses the `export` keyword to mark a variable declaration as a *property* or *prop*, which means it becomes accessible to consumers of the component (see the section on [attributes and props](#) for more information).

```
<script>
  export let foo;

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

You can specify a default initial value for a prop. It will be used if the component's consumer doesn't specify the prop on the component (or if its initial value is `undefined`) when instantiating the component. Note that if the values of props are subsequently updated, then any prop whose value is not specified will be set to `undefined` (rather than its initial value).

In development mode (see the [compiler options](#)), a warning will be printed if no default initial value is provided and the consumer does not specify a value. To squelch this warning, ensure that a default initial value is specified, even if it is `undefined`.

```
<script>
  export let bar = 'optional default initial value';
  export let baz = undefined;
</script>
```

If you export a `const`, `class` or `function`, it is readonly from outside the component. Functions are valid prop values, however, as shown below.

```
<! file: App.svelte --->
<script>
  // these are readonly
  export const thisIs = 'readonly';

  /** @param {string} name */
  export function greet(name) {
    alert(`hello ${name}!`);
  }

  // this is a prop
  export let format = (n) => n.toFixed(2);
</script>
```

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

You can use reserved words as prop names.

```
<! file: App.svelte --->
<script>
  /** @type {string} */
  let className;

  // creates a `class` property, even
  // though it is a reserved word
  export { className as class };
</script>
```

2. Assignments are 'reactive'

To change component state and trigger a re-render, just assign to a locally declared variable.

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

```
<script>
  let count = 0;

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

Because Svelte's reactivity is based on assignments, using array methods like `.push()` and `.splice()` won't automatically trigger updates. A subsequent assignment is required to trigger the update. This and more details can also be found in the [tutorial](#).

```
<script>
  let arr = [0, 1];
```

```
function handleClick() {
  // this method call does not trigger an update
  arr.push(2);
  // this assignment will trigger an update
  // if the markup references `arr`
  arr = arr;
}
</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. If you'd like to track changes to a prop, see the next example in the following section.

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

3. `$:` marks a statement as reactive

Any top-level statement (i.e. not inside a block or a function) can be made reactive by prefixing it with the `$:` [JS label syntax](#). Reactive statements run after other script code and before the component markup is rendered, whenever the values that they depend on have changed.

```
<script>
  export let title;
  export let person;

  // this will update `document.title` whenever
  // the `title` prop changes
  $: document.title = title;

  $: {
    console.log(`multiple statements can be combined`);
    console.log(`the current title is ${title}`);
  }

  // this will update `name` when `person` changes
  $: ({ name } = person);

  // don't do this. it will run before the previous line
  let name2 = name;
</script>
```

Only values which directly appear within the `$:` block will become dependencies of the reactive statement. For example, in the code below `total` will only update when `x` changes, but not `y`.

```
<! file: App.svelte --->
<script>
  let x = 0;
  let y = 0;

  /** @param {number} value */
  function yPlusAValue(value) {
```

```

        return value + y;
    }

    $: total = yPlusAValue(x);
</script>

Total: {total}
<button on:click={() => x++}> Increment X </button>

<button on:click={() => y++}> Increment Y </button>

```

It is important to note that the reactive blocks are ordered via simple static analysis at compile time, and all the compiler looks at are the variables that are assigned to and used within the block itself, not in any functions called by them. This means that `yDependent` will not be updated when `x` is updated in the following example:

```

<script>
    let x = 0;
    let y = 0;

    /** @param {number} value */
    function setY(value) {
        y = value;
    }

    $: yDependent = y;
    $: setY(x);
</script>

```

Moving the line `$. yDependent = y` below `$. setY(x)` will cause `yDependent` to be updated when `x` is updated.

If a statement consists entirely of an assignment to an undeclared variable, Svelte will inject a `let` declaration on your behalf.

```

<! file: App.svelte --->
<script>
    /** @type {number} */
    export let num;

    // we don't need to declare `squared` and `cubed`
    // - Svelte does it for us
    $: squared = num * num;
    $: cubed = squared * num;
</script>

```

4. Prefix stores with `$` to access their values

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 initialization 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>
```

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 unsubscribe 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.

<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.

Variables defined in `module` scripts are not reactive — reassigning them will not trigger a rerender even though the variable itself will update. For values shared between multiple components, consider using a [store](#).

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

  // the export keyword allows this function to be 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.

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>
```

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

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

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

  p:global(.red) {
```

```

        /* this will apply to all <p> elements belonging to this
           component with a class of red, even if class="red" does
           not initially appear in the markup, and is instead
           added at runtime. This is useful when the class
           of the element is dynamically applied, for instance
           when updating the element's classList property directly. */
    }
</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 then be referenced using just `my-animation-name` elsewhere in your code.

```

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

```

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

However, it is possible to have `<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>

```

Basic markup

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>
```

An expression might include characters that would cause syntax highlighting to fail in regular HTML, so quoting the value is permitted. The quotes do not affect how the value is parsed:

```
<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} />
```

`$$props` references all props that are passed to a component, including ones that are not declared with `export`. Using `$$props` will not perform as well as references to a specific prop because changes to any prop will cause Svelte to recheck all usages of `$$props`. But it can be useful in some cases – for example, when you don't know at compile time what props might be passed to a component.

```
<Widget {...$$props} />
```

`$$restProps` contains only the props which are *not* declared with `export`. It can be used to pass down other unknown attributes to an element in a component. It shares the same performance characteristics compared to specific property access as `$$props`.

```
<input {...$$restProps} />
```

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

Text expressions

```
{expression}
```

Text can also contain JavaScript expressions:

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>
```

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 />
```

Logic blocks

{#if ...}

```
{#if expression}...{/if}
```

```
{#if expression}...{:else if expression}...{/if}
```

```
{#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 ...}

```
{#each expression as name}...{/each}
```

```
{#each expression as name, index}...{/each}
```

```
{#each expression as name (key)}...{/each}
```

```
{#each expression as name, index (key)}...{/each}
```

```
{#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}
```

Since Svelte 4 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.

{#await ...}

```
{#await expression}...{:then name}...{:catch name}...{/await}
```

```
{#await expression}...{:then name}...{/await}
```

```
{#await expression then name}...{/await}
```

```
{#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 state will be rendered on the server.

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

{#key ...}

```
{#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}
```

Special tags

{@html ...}

```
{@html expression}
```

In a text expression, characters like `<` and `>` are escaped; however, with HTML expressions, they're not.

The expression should be valid standalone HTML — `{@html "<div>"}content{@html "</div>"}` will *not* work, because `</div>` is not valid HTML. It also will *not* compile Svelte code.

Svelte does not sanitize expressions before injecting HTML. If the data comes from an untrusted source, you must sanitize it, or you are exposing your users to an XSS vulnerability.

```
<div class="blog-post">
  <h1>{post.title}</h1>
  {@html post.content}
</div>
```

{@debug ...}

```
{@debug}
```

```
{@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.

`{@const ...}`

```
{@const assignment}
```

The `{@const ...}` tag defines a local constant.

```
<script>
  export let boxes;
</script>

{#each boxes as box}
  {@const area = box.width * box.height}
  {box.width} * {box.height} = {area}
{/each}
```

`{@const}` is only allowed as direct child of `{#if}`, `{:else if}`, `{:else}`, `{#each}`, `{:then}`, `{:catch}`, `<Component />` or `<svelte:fragment />`.

Element directives

As well as attributes, elements can have *directives*, which control the element's behaviour in some way.

on:eventname

```
on:eventname={handler}
```

```
on:eventname|modifiers={handler}
```

Use the `on:` directive to listen to DOM events.

```
<!-- file: App.svelte --->
<script>
  let count = 0;

  /** @param {MouseEvent} event */
  function handleClick(event) {
    count += 1;
  }
</script>

<button on:click={handleClick}>
  count: {count}
</button>
```

Handlers can be declared inline with no performance penalty. As with attributes, directive values may be quoted for the sake of syntax highlighters.

```
<button on:click={() => (count += 1)}>
  count: {count}
</button>
```

Add *modifiers* to DOM events with the `|` character.

```
<form on:submit|preventDefault={handleSubmit}>
  <!-- the `submit` event's default is prevented,
        so the page won't reload -->
</form>
```

The following modifiers are available:

- `preventDefault` — calls `event.preventDefault()` before running the handler
- `stopPropagation` — calls `event.stopPropagation()`, preventing the event reaching the next element
- `stopImmediatePropagation` — calls `event.stopImmediatePropagation()`, preventing other listeners of the same event from being fired.
- `passive` — improves scrolling performance on touch/wheel events (Svelte will add it automatically where it's safe to do so)
- `nonpassive` — explicitly set `passive: false`
- `capture` — fires the handler during the *capture* phase instead of the *bubbling* phase

- `once` — remove the handler after the first time it runs
- `self` — only trigger handler if `event.target` is the element itself
- `trusted` — only trigger handler if `event.isTrusted` is `true`. I.e. if the event is triggered by a user action.

Modifiers can be chained together, e.g. `on:click|once|capture={...}`.

If the `on:` directive is used without a value, the component will *forward* the event, meaning that a consumer of the component can listen for it.

```
<button on:click> The component itself will emit the click event </button>
```

It's possible to have multiple event listeners for the same event:

```
<!-- file: App.svelte --->
<script>
  let counter = 0;
  function increment() {
    counter = counter + 1;
  }

  /** @param {MouseEvent} event */
  function track(event) {
    trackEvent(event);
  }
</script>

<button on:click={increment} on:click={track}>Click me!</button>
```

bind:property

```
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 (``) 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

```
bind:group={variable}
```

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

```

<!-- file: App.svelte --->
<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

```
bind:this={dom_node}
```

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

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

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

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

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

class:name

```
class:name={value}
```

```
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

```
style:property={value}
```

```
style:property="value"
```

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>

```

use:action

use:action

`use:action={parameters}`

```

// @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 whenever that parameter changes, immediately after Svelte has applied updates to the markup.

Don't worry about the fact 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>
  export let bar;

  /** @type {import('svelte/action').Action} */
  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} />
```

Read more in the [svelte/action](#) page.

transition:fn

transition:fn

transition:fn={params}

transition:fn|global

transition:fn|global={params}

transition:fn|local

transition:fn|local={params}

```
// @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 (in Svelte 3, they were global 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}
    <!-- Svelte 3: <p transition:fadelocal> -->
    <p transition:fade>fades in and out only when y changes</p>

    <!-- Svelte 3: <p transition:fade> -->
    <p transition:fadelglobal>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

Like actions, 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.

```

<! file: App.svelte --->
<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`

`in:fn`

`in:fn={params}`

`in:fn|global`

`in:fn|global={params}`

`in:fn|local``in:fn|local={params}``out:fn``out:fn={params}``out:fn|global``out:fn|global={params}``out:fn|local``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 outtroed 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

`animate:name``animate:name={params}`

```
// @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
}
```

```
// @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.

```
<! file: App.svelte --->
<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.

```

<! file: App.svelte --->
<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}

```

Component directives

on:eventname

```
on:eventname={handler}
```

Components can emit events using `createEventDispatcher` or by forwarding DOM events.

```
<!-- SomeComponent.svelte -->
<script>
  import { createEventDispatcher } from 'svelte';

  const dispatch = createEventDispatcher();
</script>

<!-- programmatic dispatching -->
<button on:click={() => dispatch('hello')}>
  one
</button>

<!-- declarative event forwarding -->
<button on:click>
  two
</button>
```

Listening for component events looks the same as listening for DOM events:

```
<SomeComponent on:whatever={handler} />
```

As with DOM events, if the `on:` directive is used without a value, the event will be forwarded, meaning that a consumer can listen for it.

```
<SomeComponent on:whatever />
```

--style-props

```
--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>
```

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

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>
```

Note: Since this is an extra `<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:

```
<!-- Slider.svelte -->
<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" />
```

bind:property

```
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.

bind:this

```
bind:this={component_instance}
```

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

```
<ShoppingCart bind:this={cart} />  
<button on:click={() => cart.empty()}> Empty shopping cart </button>
```

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

Special elements

`<slot>`

```
<slot><!-- optional fallback --></slot>
```

```
<slot name="x"><!-- optional fallback --></slot>
```

```
<slot prop={value} />
```

Components can have child content, in the same way that elements can.

The content is exposed in the child component using the `<slot>` element, which can contain fallback content that is rendered if no children are provided.

```
<!-- Widget.svelte -->
<div>
  <slot>
    this fallback content will be rendered when no content is provided, like
    in the first example
  </slot>
</div>

<!-- App.svelte -->
<Widget />
<!-- this component will render the default content -->

<Widget>
  <p>this is some child content that will overwrite the default slot content</p>
</Widget>
```

Note: If you want to render regular `<slot>` element, You can use `<svelte:element this="slot" />`.

`<slot name="name">`

Named slots allow consumers to target specific areas. They can also have fallback content.

```
<!-- Widget.svelte -->
<div>
  <slot name="header">No header was provided</slot>
  <p>Some content between header and footer</p>
  <slot name="footer" />
</div>

<!-- App.svelte -->
<Widget>
  <h1 slot="header">Hello</h1>
  <p slot="footer">Copyright (c) 2019 Svelte Industries</p>
</Widget>
```

Components can be placed in a named slot using the syntax `<Component slot="name" />`. In order to place content in a slot without using a wrapper element, you can use the special element `<svelte:fragment>`.

```
<!-- Widget.svelte -->
<div>
  <slot name="header">No header was provided</slot>
  <p>Some content between header and footer</p>
  <slot name="footer" />
</div>

<!-- App.svelte -->
<Widget>
  <HeaderComponent slot="header" />
  <svelte:fragment slot="footer">
    <p>All rights reserved.</p>
    <p>Copyright (c) 2019 Svelte Industries</p>
  </svelte:fragment>
</Widget>
```

\$\$slots

`$$slots` is an object whose keys are the names of the slots passed into the component by the parent. If the parent does not pass in a slot with a particular name, that name will not be present in `$$slots`. This allows components to render a slot (and other elements, like wrappers for styling) only if the parent provides it.

Note that explicitly passing in an empty named slot will add that slot's name to `$$slots`. For example, if a parent passes `<div slot="title" />` to a child component, `$$slots.title` will be truthy within the child.

```
<!-- Card.svelte -->
<div>
  <slot name="title" />
  {#if $$slots.description}
    <!-- This <hr> and slot will render only if a slot named "description" is
provided. -->
    <hr />
    <slot name="description" />
  {/if}
</div>

<!-- App.svelte -->
<Card>
  <h1 slot="title">Blog Post Title</h1>
  <!-- No slot named "description" was provided so the optional slot will not be
rendered. -->
</Card>
```

`<slot key={value}>`

Slots can be rendered zero or more times and can pass values *back* to the parent using props. The parent exposes the values to the slot template using the `let:` directive.

The usual shorthand rules apply — `let:item` is equivalent to `let:item={item}`, and `<slot {item}>` is equivalent to `<slot item={item}>`.

```
<!-- FancyList.svelte -->
<ul>
  {#each items as item}
    <li class="fancy">
      <slot prop={item} />
    </li>
  {/each}
</ul>

<!-- App.svelte -->
<FancyList {items} let:prop={thing}>
  <div>{thing.text}</div>
</FancyList>
```

Named slots can also expose values. The `let:` directive goes on the element with the `slot` attribute.

```
<!-- FancyList.svelte -->
<ul>
  {#each items as item}
    <li class="fancy">
      <slot name="item" {item} />
    </li>
  {/each}
</ul>

<slot name="footer" />

<!-- App.svelte -->
<FancyList {items}>
  <div slot="item" let:item>{item.text}</div>
  <p slot="footer">Copyright (c) 2019 Svelte Industries</p>
</FancyList>
```

<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';

  /** @type {(e: MouseEvent) => void} */
  export let handler;
</script>

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

<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:

- `fullscreenElement`
- `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" />
```

<svelte:fragment>

The `<svelte:fragment>` element allows you to place content in a [named slot](#) without wrapping it in a container DOM element. This keeps the flow layout of your document intact.

```
<!-- Widget.svelte -->
<div>
  <slot name="header">No header was provided</slot>
  <p>Some content between header and footer</p>
  <slot name="footer" />
</div>

<!-- App.svelte -->
<Widget>
  <h1 slot="header">Hello</h1>
  <svelte:fragment slot="footer">
    <p>All rights reserved.</p>
    <p>Copyright (c) 2019 Svelte Industries</p>
  </svelte:fragment>
</Widget>
```

svelte

The `svelte` package exposes [lifecycle functions](#) and the [context API](#).

onMount

EXPORT_SNIPPET: svelte#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 [server-side component](#).

```
<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. a `sync` functions always return a `Promise`, and as such cannot *synchronously* return a function.

beforeUpdate

EXPORT_SNIPPET: svelte#beforeUpdate

Schedules a callback to run immediately before the component is updated after any state change.

The first time the callback runs will be before the initial `onMount`

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

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

afterUpdate

EXPORT_SNIPPET: svelte#afterUpdate

Schedules a callback to run immediately after the component has been updated.

The first time the callback runs will be after the initial `onMount`

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

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

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

EXPORT_SNIPPET: svelte#tick

Returns a promise that resolves once any pending state changes have been applied, or in the next micro-task 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>

```

setContext

EXPORT_SNIPPET: svelte#setContext

Associates an arbitrary `context` object with the current component and the specified `key` and returns that object. The context is then available to children of the component (including slotted content) with `getContext`.

Like lifecycle functions, this must be called during component initialisation.

```

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

  setContext('answer', 42);
</script>

```

Context is not inherently reactive. If you need reactive values in context then you can pass a store into context, which *will* be reactive.

getContext

EXPORT_SNIPPET: svelte#getContext

Retrieves the context that belongs to the closest parent component with the specified `key`. Must be called during component initialisation.

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

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

hasContext

EXPORT_SNIPPET: svelte#hasContext

Checks whether a given `key` has been set in the context of a parent component. Must be called during component initialisation.

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

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

getAllContexts

EXPORT_SNIPPET: svelte#getAllContexts

Retrieves the whole context map that belongs to the closest parent component. Must be called during component initialisation. 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>
```

createEventDispatcher

EXPORT_SNIPPET: svelte#createEventDispatcher

Creates an event dispatcher that can be used to dispatch [component events](#). Event dispatchers are functions that can take two arguments: `name` and `detail`.

Component events created with `createEventDispatcher` create a `CustomEvent`. These events do not [bubble](#). The `detail` argument corresponds to the `CustomEvent.detail` property and can contain any type of data.

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

  const dispatch = createEventDispatcher();
</script>

<button on:click={() => dispatch('notify', 'detail value')}>Fire Event</button>
```

Events dispatched from child components can be listened to in their parent. Any data provided when the event was dispatched is available on the `detail` property of the event object.

```
<script>
  function callbackFunction(event) {
    console.log(`Notify fired! Detail: ${event.detail}`);
  }
</script>

<Child on:notify={callbackFunction} />
```

Events can be cancelable by passing a third parameter to the dispatch function. The function returns `false` if the event is cancelled with `event.preventDefault()`, otherwise it returns `true`.

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

  const dispatch = createEventDispatcher();

  function notify() {
    const shouldContinue = dispatch('notify', 'detail value', { cancelable:
true });
    if (shouldContinue) {
      // no one called preventDefault
    } else {
      // a listener called preventDefault
    }
  }
</script>
```

You can type the event dispatcher to define which events it can receive. This will make your code more type safe both within the component (wrong calls are flagged) and when using the component (types of the events are now narrowed). See [here](#) how to do it.

Types

TYPES: svelte

[Go to TOC](#)

'svelte/store'

The `svelte/store` module exports functions for creating [readable](#), [writable](#) and [derived](#) stores.

Keep in mind that you don't *have* to use these functions to enjoy the [reactive](#) `$store` [syntax](#) in your components. Any object that correctly implements `.subscribe`, `unsubscribe`, and (optionally) `.set` is a valid store, and will work both with the special syntax, and with Svelte's built-in [derived](#) stores.

This makes it possible to wrap almost any other reactive state handling library for use in Svelte. Read more about the [store contract](#) to see what a correct implementation looks like.

writable

EXPORT_SNIPPET: svelte/store#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.

```
/// 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.

```
/// 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

EXPORT_SNIPPET: svelte/store#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`.

```

<!-- file: App.svelte --->
// cut---
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

EXPORT_SNIPPET: svelte/store#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

EXPORT_SNIPPET: svelte/store#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

EXPORT_SNIPPET: svelte/store#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);
```

Types

TYPES: svelte/store

'svelte/motion'

The `svelte/motion` module exports two functions, `tweened` and `spring`, for creating writable stores whose values change over time after `set` and `update`, rather than immediately.

tweened

EXPORT_SNIPPET: svelte/motion#tweened

Tweened stores update their values over a fixed duration. The following options are available:

- `delay` (`number`, default 0) — milliseconds before starting
- `duration` (`number` | `function`, default 400) — milliseconds the tween lasts
- `easing` (`function`, default `t => t`) — an [easing function](#)
- `interpolate` (`function`) — see below

`store.set` and `store.update` can accept a second `options` argument that will override the options passed in upon instantiation.

Both functions return a Promise that resolves when the tween completes. If the tween is interrupted, the promise will never resolve.

Out of the box, Svelte will interpolate between two numbers, two arrays or two objects (as long as the arrays and objects are the same 'shape', and their 'leaf' properties are also numbers).

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

  const size = tweened(1, {
    duration: 300,
    easing: cubicOut
  });

  function handleClick() {
    // this is equivalent to size.update(n => n + 1)
    $size += 1;
  }
</script>

<button on:click={handleClick} style="transform: scale({$size}); transform-origin:
0 0">
  embiggen
</button>
```

If the initial value is `undefined` or `null`, the first value change will take effect immediately. This is useful when you have tweened values that are based on props, and don't want any motion when the component first renders.

```
// @filename: ambient.d.ts
declare global {
  var $size: number;
  var big: number;
}

export {};
// @filename: motion.ts
// cut---
import { tweened } from 'svelte/motion';
import { cubicOut } from 'svelte/easing';

const size = tweened(undefined, {
  duration: 300,
  easing: cubicOut
});

$: $size = big ? 100 : 10;
```

The `interpolate` option allows you to tween between *any* arbitrary values. It must be an `(a, b) => t => value` function, where `a` is the starting value, `b` is the target value, `t` is a number between 0 and 1, and `value` is the result. For example, we can use the [d3-interpolate](#) package to smoothly interpolate between two colours.

```
<script>
  import { interpolateLab } from 'd3-interpolate';
  import { tweened } from 'svelte/motion';

  const colors = ['rgb(255, 62, 0)', 'rgb(64, 179, 255)', 'rgb(103, 103, 120)'];

  const color = tweened(colors[0], {
    duration: 800,
    interpolate: interpolateLab
  });
</script>

{#each colors as c}
  <button style="background-color: {c}; color: white; border: none;" on:click=
  {(e) => color.set(c)}>
    {c}
  </button>
{/each}

<h1 style="color: {$color}">{$color}</h1>
```

spring

EXPORT_SNIPPET: svelte/motion#spring

A `spring` store gradually changes to its target value based on its `stiffness` and `damping` parameters. Whereas `tweened` stores change their values over a fixed duration, `spring` stores change over a duration that is determined by their existing velocity, allowing for more natural-seeming motion in many situations. The following options are available:

- `stiffness` (`number`, default `0.15`) — a value between 0 and 1 where higher means a 'tighter' spring
- `damping` (`number`, default `0.8`) — a value between 0 and 1 where lower means a 'springier' spring
- `precision` (`number`, default `0.01`) — determines the threshold at which the spring is considered to have 'settled', where lower means more precise

All of the options above can be changed while the spring is in motion, and will take immediate effect.

```
import { spring } from 'svelte/motion';

const size = spring(100);
size.stiffness = 0.3;
size.damping = 0.4;
size.precision = 0.005;
```

As with `tweened` stores, `set` and `update` return a Promise that resolves if the spring settles.

Both `set` and `update` can take a second argument — an object with `hard` or `soft` properties. `{ hard: true }` sets the target value immediately; `{ soft: n }` preserves existing momentum for `n` seconds before settling. `{ soft: true }` is equivalent to `{ soft: 0.5 }`.

```
import { spring } from 'svelte/motion';

const coords = spring({ x: 50, y: 50 });
// updates the value immediately
coords.set({ x: 100, y: 200 }, { hard: true });
// preserves existing momentum for 1s
coords.update(
  (target_coords, coords) => {
    return { x: target_coords.x, y: coords.y };
  },
  { soft: 1 }
);
```

See a full example on the spring tutorial.

```
<script>
  import { spring } from 'svelte/motion';

  const coords = spring(
    { x: 50, y: 50 },
    {
      stiffness: 0.1,
      damping: 0.25
    }
  );
</script>
```

If the initial value is `undefined` or `null`, the first value change will take effect immediately, just as with `tweened` values (see above).

```
// @filename: ambient.d.ts
declare global {
  var $size: number;
  var big: number;
}
```



```
export {};  
  
// @filename: motion.ts  
// cut---  
import { spring } from 'svelte/motion';  
  
const size = spring();  
$: $size = big ? 100 : 10;
```

Types

TYPES: svelte/motion

'svelte/transition'

The `svelte/transition` module exports seven functions: `fade`, `blur`, `fly`, `slide`, `scale`, `draw` and `crossfade`. They are for use with Svelte `transitions`.

fade

EXPORT_SNIPPET: svelte/transition#fade

```
transition:fade={params}
```

```
in:fade={params}
```

```
out:fade={params}
```

Animates the opacity of an element from 0 to the current opacity for `in` transitions and from the current opacity to 0 for `out` transitions.

`fade` accepts the following parameters:

- `delay` (`number`, default 0) — milliseconds before starting
- `duration` (`number`, default 400) — milliseconds the transition lasts
- `easing` (`function`, default `linear`) — an [easing function](#)

You can see the `fade` transition in action in the [transition tutorial](#).

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

{#if condition}
  <div transition:fade={{ delay: 250, duration: 300 }}>fades in and out</div>
{/if}
```

blur

EXPORT_SNIPPET: svelte/transition#blur

```
transition:blur={params}
```

```
in:blur={params}
```

```
out:blur={params}
```

Animates a `blur` filter alongside an element's opacity.

`blur` accepts the following parameters:

- `delay` (`number` , default 0) — milliseconds before starting
- `duration` (`number` , default 400) — milliseconds the transition lasts
- `easing` (`function` , default `cubicInOut`) — an [easing function](#)
- `opacity` (`number` , default 0) - the opacity value to animate out to and in from
- `amount` (`number` | `string` , default 5) - the size of the blur. Supports css units (for example: `"4rem"`). The default unit is `px`

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

{#if condition}
  <div transition:blur={{ amount: 10 }}>fades in and out</div>
{/if}
```

fly

EXPORT_SNIPPET: svelte/transition#fly

```
transition:fly={params}
```

```
in:fly={params}
```

```
out:fly={params}
```

Animates the x and y positions and the opacity of an element. `in` transitions animate from the provided values, passed as parameters to the element's default values. `out` transitions animate from the element's default values to the provided values.

`fly` accepts the following parameters:

- `delay` (`number` , default 0) — milliseconds before starting
- `duration` (`number` , default 400) — milliseconds the transition lasts
- `easing` (`function` , default `cubicOut`) — an [easing function](#)
- `x` (`number` | `string` , default 0) - the x offset to animate out to and in from
- `y` (`number` | `string` , default 0) - the y offset to animate out to and in from
- `opacity` (`number` , default 0) - the opacity value to animate out to and in from

x and y use `px` by default but support css units, for example `x: '100vw'` or `y: '50%'`. You can see the `fly` transition in action in the [transition tutorial](#).

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

```

</script>

{#if condition}
  <div
    transition:fly={{ delay: 250, duration: 300, x: 100, y: 500, opacity: 0.5,
    easing: quintOut }}
  >
    flies in and out
  </div>
{/if}

```

slide

EXPORT_SNIPPET: svelte/transition#slide

```
transition:slide={params}
```

```
in:slide={params}
```

```
out:slide={params}
```

Slides an element in and out.

`slide` accepts the following parameters:

- `delay` (`number` , default 0) — milliseconds before starting
- `duration` (`number` , default 400) — milliseconds the transition lasts
- `easing` (`function` , default `cubicOut`) — an [easing function](#)
- `axis` (`x` | `y` , default `y`) — the axis of motion along which the transition occurs

```

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

{#if condition}
  <div transition:slide={{ delay: 250, duration: 300, easing: quintOut, axis:
  'x' }}>
    slides in and out horizontally
  </div>
{/if}

```

scale

EXPORT_SNIPPET: svelte/transition#scale

```
transition:scale={params}
```

```
in:scale={params}
```

```
out:scale={params}
```

Animates the opacity and scale of an element. `in` transitions animate from an element's current (default) values to the provided values, passed as parameters. `out` transitions animate from the provided values to an element's default values.

`scale` accepts the following parameters:

- `delay` (`number` , default 0) — milliseconds before starting
- `duration` (`number` , default 400) — milliseconds the transition lasts
- `easing` (`function` , default `cubicOut`) — an [easing function](#)
- `start` (`number` , default 0) - the scale value to animate out to and in from
- `opacity` (`number` , default 0) - the opacity value to animate out to and in from

```
<script>
  import { scale } from 'svelte/transition';
  import { quintOut } from 'svelte/easing';
</script>

{#if condition}
  <div transition:scale={{ duration: 500, delay: 500, opacity: 0.5, start: 0.5,
    easing: quintOut }}>
    scales in and out
  </div>
{/if}
```

draw

EXPORT_SNIPPET: svelte/transition#draw

```
transition:draw={params}
```

```
in:draw={params}
```

```
out:draw={params}
```

Animates the stroke of an SVG element, like a snake in a tube. `in` transitions begin with the path invisible and draw the path to the screen over time. `out` transitions start in a visible state and gradually erase the path. `draw` only works with elements that have a `getTotalLength` method, like `<path>` and `<polyline>`.

`draw` accepts the following parameters:

- `delay` (`number` , default 0) — milliseconds before starting
- `speed` (`number` , default undefined) - the speed of the animation, see below.
- `duration` (`number` | `function` , default 800) — milliseconds the transition lasts
- `easing` (`function` , default `cubicInOut`) — an [easing function](#)

The `speed` parameter is a means of setting the duration of the transition relative to the path's length. It is a modifier that is applied to the length of the path: $\text{duration} = \text{length} / \text{speed}$. A path that is 1000 pixels with a speed of 1 will have a duration of `1000ms`, setting the speed to `0.5` will double that duration and setting it to `2` will halve it.

```
<script>
  import { draw } from 'svelte/transition';
  import { quintOut } from 'svelte/easing';
</script>

<svg viewBox="0 0 5 5" xmlns="http://www.w3.org/2000/svg">
  {#if condition}
    <path
      transition:draw={{ duration: 5000, delay: 500, easing: quintOut }}
      d="M2 1 h1 v1 h-1 v1 h-1 v-1 h-1 v-1 h1 z"
      fill="none"
      stroke="cornflowerblue"
      stroke-width="0.1px"
      stroke-linejoin="round"
    />
  {/if}
</svg>
```

crossfade

EXPORT_SNIPPET: svelte/transition#crossfade

The `crossfade` function creates a pair of [transitions](#) called `send` and `receive`. When an element is 'sent', it looks for a corresponding element being 'received', and generates a transition that transforms the element to its counterpart's position and fades it out. When an element is 'received', the reverse happens. If there is no counterpart, the `fallback` transition is used.

`crossfade` accepts the following parameters:

- `delay` (`number`, default 0) — milliseconds before starting
- `duration` (`number` | `function`, default 800) — milliseconds the transition lasts
- `easing` (`function`, default `cubicOut`) — an [easing function](#)
- `fallback` (`function`) — A fallback [transition](#) to use for send when there is no matching element being received, and for receive when there is no element being sent.

```
<script>
  import { crossfade } from 'svelte/transition';
  import { quintOut } from 'svelte/easing';

  const [send, receive] = crossfade({
    duration: 1500,
    easing: quintOut
  });
</script>

{#if condition}
```

```
<h1 in:send={{ key }} out:receive={{ key }}>BIG ELEM</h1>
{:else}
<small in:send={{ key }} out:receive={{ key }}>small elem</small>
{/if}
```

Types

TYPES: svelte/transition

'svelte/animate'

The `svelte/animate` module exports one function for use with Svelte [animations](#).

flip

EXPORT_SNIPPET: svelte/animate#flip

```
animate:flip={params}
```

The `flip` function calculates the start and end position of an element and animates between them, translating the `x` and `y` values. `flip` stands for [First](#), [Last](#), [Invert](#), [Play](#).

`flip` accepts the following parameters:

- `delay` (`number` , default 0) — milliseconds before starting
- `duration` (`number` | `function` , default `d => Math.sqrt(d) * 120`) — see below
- `easing` (`function` , default `cubicOut`) — an [easing function](#)

`duration` can be provided as either:

- a `number` , in milliseconds.
- a function, `distance: number => duration: number` , receiving the distance the element will travel in pixels and returning the duration in milliseconds. This allows you to assign a duration that is relative to the distance travelled by each element.

You can see a full example on the [animations tutorial](#).

```
<script>
  import { flip } from 'svelte/animate';
  import { quintOut } from 'svelte/easing';

  let list = [1, 2, 3];
</script>

{#each list as n (n)}
  <div animate:flip={{ delay: 250, duration: 250, easing: quintOut }}>
    {n}
  </div>
{/each}
```

Types

TYPES: svelte/animate

'svelte/easing'

Easing functions specify the rate of change over time and are useful when working with Svelte's built-in transitions and animations as well as the tweened and spring utilities. `svelte/easing` contains 31 named exports, a `linear` ease and 3 variants of 10 different easing functions: `in`, `out` and `inOut`.

You can explore the various eases using the [ease visualiser](#) in the [examples section](#).

ease	in	out	inOut
back	<code>backIn</code>	<code>backOut</code>	<code>backInOut</code>
bounce	<code>bounceIn</code>	<code>bounceOut</code>	<code>bounceInOut</code>
circ	<code>circIn</code>	<code>circOut</code>	<code>circInOut</code>
cubic	<code>cubicIn</code>	<code>cubicOut</code>	<code>cubicInOut</code>
elastic	<code>elasticIn</code>	<code>elasticOut</code>	<code>elasticInOut</code>
expo	<code>expoIn</code>	<code>expoOut</code>	<code>expoInOut</code>
quad	<code>quadIn</code>	<code>quadOut</code>	<code>quadInOut</code>
quart	<code>quartIn</code>	<code>quartOut</code>	<code>quartInOut</code>
quint	<code>quintIn</code>	<code>quintOut</code>	<code>quintInOut</code>
sine	<code>sineIn</code>	<code>sineOut</code>	<code>sineInOut</code>

[Go to TOC](#)

svelte/action

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' }} on:emit={handleEmit} />
```

Types

TYPES: svelte/action

'svelte/compiler'

Typically, you won't interact with the Svelte compiler directly, but will instead integrate it into your build system using a bundler plugin. The bundler plugin that the Svelte team most recommends and invests in is [vite-plugin-svelte](#). The [SvelteKit](#) framework provides a setup leveraging `vite-plugin-svelte` to build applications as well as a [tool for packaging Svelte component libraries](#). Svelte Society maintains a list of [other bundler plugins](#) for additional tools like Rollup and Webpack.

Nonetheless, it's useful to understand how to use the compiler, since bundler plugins generally expose compiler options to you.

compile

EXPORT_SNIPPET: svelte/compiler#compile

This is where the magic happens. `svelte.compile` takes your component source code, and turns it into a JavaScript module that exports a class.

```
// @filename: ambient.d.ts
declare global {
  var source: string
}

export {}

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

const result = compile(source, {
  // options
});
```

Refer to [CompileOptions](#) for all the available options.

The returned `result` object contains the code for your component, along with useful bits of metadata.

```
// @filename: ambient.d.ts
declare global {
  const source: string;
}

export {};

// @filename: main.ts
import { compile } from 'svelte/compiler';
// cut---
const { js, css, ast, warnings, vars, stats } = compile(source);
```

Refer to [CompileResult](#) for a full description of the compile result.

parse

```
EXPORT_SNIPPET: svelte/compiler#parse
```

The `parse` function parses a component, returning only its abstract syntax tree. Unlike compiling with the `generate: false` option, this will not perform any validation or other analysis of the component beyond parsing it. Note that the returned AST is not considered public API, so breaking changes could occur at any point in time.

```
// @filename: ambient.d.ts
declare global {
  var source: string;
}

export {};

// @filename: main.ts
// cut---
import { parse } from 'svelte/compiler';

const ast = parse(source, { filename: 'App.svelte' });
```

preprocess

```
EXPORT_SNIPPET: svelte/compiler#preprocess
```

A number of [official and community-maintained preprocessing plugins](#) are available to allow you to use Svelte with tools like TypeScript, PostCSS, SCSS, and Less.

You can write your own preprocessor using the `svelte.preprocess` API.

The `preprocess` function provides convenient hooks for arbitrarily transforming component source code. For example, it can be used to convert a `<style lang="sass">` block into vanilla CSS.

The first argument is the component source code. The second is an array of *preprocessors* (or a single preprocessor, if you only have one), where a preprocessor is an object with a `name` which is required, and `markup`, `script` and `style` functions, each of which is optional.

The `markup` function receives the entire component source text, along with the component's `filename` if it was specified in the third argument.

The `script` and `style` functions receive the contents of `<script>` and `<style>` elements respectively (`content`) as well as the entire component source text (`markup`). In addition to `filename`, they get an object of the element's attributes.

Each `markup`, `script` or `style` function must return an object (or a Promise that resolves to an object) with a `code` property, representing the transformed source code. Optionally they can return an array of `dependencies` which represents files to watch for changes, and a `map` object which is a sourcemap mapping back the transformation to the original code. `script` and `style` preprocessors can optionally return a record of attributes which represent the updated attributes on the script/style tag.

Preprocessor functions should return a `map` object whenever possible or else debugging becomes harder as stack traces can't link to the original code correctly.

```
// @filename: ambient.d.ts
declare global {
  var source: string;
}

export {};

// @filename: main.ts
// cut---
import { preprocess } from 'svelte/compiler';
import MagicString from 'magic-string';

const { code } = await preprocess(
  source,
  {
    markup: ({ content, filename }) => {
      const pos = content.indexOf('foo');
      if (pos < 0) {
        return { code: content };
      }
      const s = new MagicString(content, { filename });
      s.overwrite(pos, pos + 3, 'bar', { storeName: true });
      return {
        code: s.toString(),
        map: s.generateMap()
      };
    },
    {
      filename: 'App.svelte'
    }
  }
);
```

If a `dependencies` array is returned, it will be included in the result object. This is used by packages like `vite-plugin-svelte` and `rollup-plugin-svelte` to watch additional files for changes, in the case where your `<style>` tag has an `@import` (for example).

```
// @filename: ambient.d.ts
declare global {
  var source: string;
}

export {};

// @filename: main.ts
```

```

// @errors: 2322 2345 2339
/// <reference types="@types/node" />
// cut---
import { preprocess } from 'svelte/compiler';
import MagicString from 'magic-string';
import sass from 'sass';
import { dirname } from 'path';

const { code } = await preprocess(
  source,
  {
    name: 'my-fancy-preprocessor',
    markup: ({ content, filename }) => {
      // Return code as is when no foo string present
      const pos = content.indexOf('foo');
      if (pos < 0) {
        return;
      }

      // Replace foo with bar using MagicString which provides
      // a source map along with the changed code
      const s = new MagicString(content, { filename });
      s.overwrite(pos, pos + 3, 'bar', { storeName: true });

      return {
        code: s.toString(),
        map: s.generateMap({ hires: true, file: filename })
      };
    },
    style: async ({ content, attributes, filename }) => {
      // only process <style lang="sass">
      if (attributes.lang !== 'sass') return;

      const { css, stats } = await new Promise((resolve, reject) => {
        sass.render(
          {
            file: filename,
            data: content,
            includePaths: [dirname(filename)]
          },
          (err, result) => {
            if (err) reject(err);
            else resolve(result);
          }
        );
      });

      // remove lang attribute from style tag
      delete attributes.lang;

      return {
        code: css.toString(),
        dependencies: stats.includedFiles,
        attributes
      };
    }
  },
  {
    filename: 'App.svelte'
  }
);

```


Multiple preprocessors can be used together. The output of the first becomes the input to the second. Within one preprocessor, `markup` runs first, then `script` and `style`.

In Svelte 3, all `markup` functions ran first, then all `script` and then all `style` preprocessors. This order was changed in Svelte 4.

```
// @errors: 2322
// @filename: ambient.d.ts
declare global {
  var source: string;
}

export {};

// @filename: main.ts
// cut---
import { preprocess } from 'svelte/compiler';

const { code } = await preprocess(source, [
  {
    name: 'first preprocessor',
    markup: () => {
      console.log('this runs first');
    },
    script: () => {
      console.log('this runs second');
    },
    style: () => {
      console.log('this runs third');
    }
  },
  {
    name: 'second preprocessor',
    markup: () => {
      console.log('this runs fourth');
    },
    script: () => {
      console.log('this runs fifth');
    },
    style: () => {
      console.log('this runs sixth');
    }
  }
], {
  filename: 'App.svelte'
});
```

walk

EXPORT_SNIPPET: svelte/compiler#walk

The `walk` function provides a way to walk the abstract syntax trees generated by the parser, using the compiler's own built-in instance of `estree-walker`.

The walker takes an abstract syntax tree to walk and an object with two optional methods: `enter` and `leave`. For each node, `enter` is called (if present). Then, unless `this.skip()` is called during `enter`, each of the children are traversed, and then `leave` is called on the node.

```
// @filename: ambient.d.ts
declare global {
  var ast: import('estree').Node;
  function do_something(node: import('estree').Node): void;
  function do_something_else(node: import('estree').Node): void;
  function should_skip_children(node: import('estree').Node): boolean;
}

export {};

// @filename: main.ts
// @errors: 7006
// cut---
import { walk } from 'svelte/compiler';

walk(ast, {
  enter(node, parent, prop, index) {
    do_something(node);
    if (should_skip_children(node)) {
      this.skip();
    }
  },
  leave(node, parent, prop, index) {
    do_something_else(node);
  }
});
```

VERSION

EXPORT_SNIPPET: svelte/compiler#VERSION

The current version, as set in package.json.

```
import { VERSION } from 'svelte/compiler';
console.log(`running svelte version ${VERSION}`);
```

Types

TYPES: svelte/compiler

[Go to TOC](#)

'Client-side component API'

Creating a component

```
// @errors: 2554
// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare global {
  class Component extends SvelteComponent {}
  var options: ComponentConstructorOptions<Record<string, any>>;
}

// @filename: index.ts
// @errors: 2554
// cut---
const component = new Component(options);
```

A client-side component — that is, a component compiled with `generate: 'dom'` (or the `generate` option left unspecified) is a JavaScript class.

```
// @errors: 2554
// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare module './App.svelte' {
  class Component extends SvelteComponent {}
  export default Component;
}

// @filename: index.ts
// @errors: 2554
// cut---
import App from './App.svelte';

const app = new App({
  target: document.body,
  props: {
    // assuming App.svelte contains something like
    // `export let answer`:
    answer: 42
  }
});
```

The following initialisation options can be provided:

option	default	description
<code>target</code>	<code>none</code>	An <code>HTMLElement</code> or <code>ShadowRoot</code> to render to. This option is required
<code>anchor</code>	<code>null</code>	A child of <code>target</code> to render the component immediately before
<code>props</code>	<code>{}</code>	An object of properties to supply to the component

option	default	description
con- text	new Map()	A <code>Map</code> of root-level context key-value pairs to supply to the component
hy- drate	false	See below
intro	false	If <code>true</code> , will play transitions on initial render, rather than waiting for subsequent state changes

Existing children of `target` are left where they are.

The `hydrate` option instructs Svelte to upgrade existing DOM (usually from server-side rendering) rather than creating new elements. It will only work if the component was compiled with the `hydratable: true` option. Hydration of `<head>` elements only works properly if the server-side rendering code was also compiled with `hydratable: true`, which adds a marker to each element in the `<head>` so that the component knows which elements it's responsible for removing during hydration.

Whereas children of `target` are normally left alone, `hydrate: true` will cause any children to be removed. For that reason, the `anchor` option cannot be used alongside `hydrate: true`.

The existing DOM doesn't need to match the component — Svelte will 'repair' the DOM as it goes.

```
// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare module './App.svelte' {
  class Component extends SvelteComponent {}
  export default Component;
}

// @filename: index.ts
// @errors: 2322 2554
// cut---
import App from './App.svelte';

const app = new App({
  target: document.querySelector('#server-rendered-html'),
  hydrate: true
});
```

\$set

```
// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare global {
  class Component extends SvelteComponent {}
  var component: Component;
  var props: Record<string, any>;
}

export {};
```

```
// @filename: index.ts
// cut---
component.$set(props);
```

Programmatically sets props on an instance. `component.$set({ x: 1 })` is equivalent to `x = 1` inside the component's `<script>` block.

Calling this method schedules an update for the next microtask — the DOM is *not* updated synchronously.

```
// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare global {
  class Component extends SvelteComponent {}
  var component: Component;
}

export {};

// @filename: index.ts
// cut---
component.$set({ answer: 42 });
```

\$on

```
// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare global {
  class Component extends SvelteComponent {}
  var component: Component;
  var ev: string;
  var callback: (event: CustomEvent) => void;
}

export {};

// @filename: index.ts
// cut---
component.$on(ev, callback);
```

Causes the `callback` function to be called whenever the component dispatches an `event`.

A function is returned that will remove the event listener when called.

```
// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare global {
  class Component extends SvelteComponent {}
  var component: Component;
}

export {};

// @filename: index.ts
// cut---
const off = component.$on('selected', (event) => {
```

```

    console.log(event.detail.selection);
  });
  off();

```

\$destroy

```

// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare global {
  class Component extends SvelteComponent {}
  var component: Component;
}

export {}

// @filename: index.ts
// cut---
component.$destroy();

```

Removes a component from the DOM and triggers any `onDestroy` handlers.

Component props

```

// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare global {
  class Component extends SvelteComponent {}
  var component: Component;
}

export {}

// @filename: index.ts
// @errors: 2339
// cut---
component.prop;

```

```

// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare global {
  class Component extends SvelteComponent {}
  var component: Component;
  var value: unknown;
}

export {}

// @filename: index.ts
// @errors: 2339
// cut---
component.prop = value;

```

If a component is compiled with `accessors: true`, each instance will have getters and setters corresponding to each of the component's props. Setting a value will cause a *synchronous* update, rather than the default async update caused by `component.$set(...)`.

By default, `accessors` is `false`, unless you're compiling as a custom element.

```
// @filename: ambient.d.ts
import { SvelteComponent, ComponentConstructorOptions } from 'svelte';

declare global {
  class Component extends SvelteComponent {}
  var component: Component;
  var props: Record<string, any>;
}

export {}

// @filename: index.ts
// @errors: 2339
// cut---
console.log(component.count);
component.count += 1;
```

'Server-side component API'

```
// @noErrors
const result = Component.render(...)
```

Unlike client-side components, server-side components don't have a lifespan after you render them — their whole job is to create some HTML and CSS. For that reason, the API is somewhat different.

A server-side component exposes a `render` method that can be called with optional props. It returns an object with `head`, `html`, and `css` properties, where `head` contains the contents of any `<svelte:head>` elements encountered.

You can import a Svelte component directly into Node using `svelte/register`.

```
// @noErrors
require('svelte/register');

const App = require('./App.svelte').default;

const { head, html, css } = App.render({
  answer: 42
});
```

The `.render()` method accepts the following parameters:

parameter	default	description
<code>props</code>	<code>{}</code>	An object of properties to supply to the component
<code>options</code>	<code>{}</code>	An object of options

The `options` object takes in the following options:

option	default	description
<code>context</code>	<code>new Map()</code>	A <code>Map</code> of root-level context key-value pairs to supply to the component

```
// @noErrors
const { head, html, css } = App.render(
  // props
  { answer: 42 },
  // options
  {
    context: new Map([['context-key', 'context-value']])
  }
);
```

[Go to TOC](#)

'Custom elements API'

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" />

<!-- in Svelte 3, do this instead:
<svelte:options tag="my-element" />
-->

<script>
  export let name = 'world';
</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);
// In Svelte 3, do this instead:
// customElements.define('my-element', MyElement);
```

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>
`;
```

By default, custom elements are compiled with `accessors: true`, which means that any `props` are exposed as properties of the DOM element (as well as being readable/writable as attributes, where possible).

To prevent this, add `accessors={false}` to `<svelte:options>`.

```
// @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>
  export let elementIndex;
  export let attachedInternals;
  // ...
  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 `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.

Frequently asked questions

I'm new to Svelte. Where should I start?

We think the best way to get started is playing through the interactive [tutorial](#). Each step there is mainly focused on one specific aspect and is easy to follow. You'll be editing and running real Svelte components right in your browser.

Five to ten minutes should be enough to get you up and running. An hour and a half should get you through the entire tutorial.

Where can I get support?

If your question is about certain syntax, the [API page](#) is a good place to start.

Stack Overflow is a popular forum to ask code-level questions or if you're stuck with a specific error. Read through the existing questions tagged with [Svelte](#) or [ask your own!](#)

There are online forums and chats which are a great place for discussion about best practices, application architecture or just to get to know fellow Svelte users. [Our Discord](#) or [the Reddit channel](#) are examples of that. If you have an answerable code-level question, Stack Overflow is usually a better fit.

Are there any third-party resources?

Svelte Society maintains a [list of books and videos](#).

How can I get VS Code to syntax-highlight my .svelte files?

There is an [official VS Code extension for Svelte](#).

Is there a tool to automatically format my .svelte files?

You can use prettier with the [prettier-plugin-svelte](#) plugin.

How do I document my components?

In editors which use the Svelte Language Server you can document Components, functions and exports using specially formatted comments.

```
<script>
  /** What should we call the user? */
  export let name = 'world';
</script>
```

```

<!--
@Component
Here's some documentation for this component.
It will show up on hover.

- You can use markdown here.
- You can also use code blocks here.
- Usage:
  ```tsx
 <main name="Arethra">
  ```
-->
<main>
  <h1>
    Hello, {name}
  </h1>
</main>

```

Note: The `@component` is necessary in the HTML comment which describes your component.

What about TypeScript support?

You need to install a preprocessor such as [svelte-preprocess](#). You can run type checking from the command line with [svelte-check](#).

To declare the type of a reactive variable in a Svelte template, you should use the following syntax:

```

const count: number = 100;

// cut---
let x: number;
$: x = count + 1;

```

To import a type or interface make sure to use TypeScript's `type` modifier:

```

// @filename: SomeFile.ts
export interface SomeInterface {
  foo: string;
}

// @filename: index.ts
// cut---
import type { SomeInterface } from './SomeFile';

```

You must use the `type` modifier because [svelte-preprocess](#) doesn't know whether an import is a type or a value — it only transpiles one file at a time without knowledge of the other files and therefore can't safely erase imports which only contain types without this modifier present.

Does Svelte scale?

There will be a blog post about this eventually, but in the meantime, check out [this issue](#).

Is there a UI component library?

There are several UI component libraries as well as standalone components. Find them under the [components section](#) of the Svelte Society website.

How do I test Svelte apps?

How your application is structured and where logic is defined will determine the best way to ensure it is properly tested. It is important to note that not all logic belongs within a component - this includes concerns such as data transformation, cross-component state management, and logging, among others. Remember that the Svelte library has its own test suite, so you do not need to write tests to validate implementation details provided by Svelte.

A Svelte application will typically have three different types of tests: Unit, Component, and End-to-End (E2E).

Unit Tests: Focus on testing business logic in isolation. Often this is validating individual functions and edge cases. By minimizing the surface area of these tests they can be kept lean and fast, and by extracting as much logic as possible from your Svelte components more of your application can be covered using them. When creating a new SvelteKit project, you will be asked whether you would like to setup [Vitest](#) for unit testing. There are a number of other test runners that could be used as well.

Component Tests: Validating that a Svelte component mounts and interacts as expected throughout its life-cycle requires a tool that provides a Document Object Model (DOM). Components can be compiled (since Svelte is a compiler and not a normal library) and mounted to allow asserting against element structure, listeners, state, and all the other capabilities provided by a Svelte component. Tools for component testing range from an in-memory implementation like jsdom paired with a test runner like [Vitest](#) to solutions that leverage an actual browser to provide a visual testing capability such as [Playwright](#) or [Cypress](#).

End-to-End Tests: To ensure your users are able to interact with your application it is necessary to test it as a whole in a manner as close to production as possible. This is done by writing end-to-end (E2E) tests which load and interact with a deployed version of your application in order to simulate how the user will interact with your application. When creating a new SvelteKit project, you will be asked whether you would like to setup [Playwright](#) for end-to-end testing. There are many other E2E test libraries available for use as well.

Some resources for getting started with testing:

- [Svelte Testing Library](#)
- [Svelte Component Testing in Cypress](#)
- [Example using vitest](#)
- [Example using uvu test runner with JSDOM](#)
- [Test Svelte components using Vitest & Playwright](#)
- [Component testing with WebdriverIO](#)

Is there a router?

The official routing library is [SvelteKit](#). SvelteKit provides a filesystem router, server-side rendering (SSR), and hot module reloading (HMR) in one easy-to-use package. It shares similarities with Next.js for React.

However, you can use any router library. A lot of people use [page.js](#). There's also [navaid](#), which is very similar. And [universal-router](#), which is isomorphic with child routes, but without built-in history support.

If you prefer a declarative HTML approach, there's the isomorphic [svelte-routing](#) library and a fork of it called [svelte-navigator](#) containing some additional functionality.

If you need hash-based routing on the client side, check out [svelte-spa-router](#) or [abstract-state-router](#).

[Routify](#) is another filesystem-based router, similar to SvelteKit's router. Version 3 supports Svelte's native SSR.

You can see a [community-maintained list of routers on sveltesociety.dev](#).

Is Svelte v2 still available?

New features aren't being added to it, and bugs will probably only be fixed if they are extremely nasty or present some sort of security vulnerability.

The documentation is still available [here](#).

How do I do hot module reloading?

We recommend using [SvelteKit](#), which supports HMR out of the box and is built on top of [Vite](#) and [svelte-hmr](#). There are also community plugins for [rollup](#) and [webpack](#).

'Accessibility warnings'

Accessibility (shortened to a11y) isn't always easy to get right, but Svelte will help by warning you at compile time if you write inaccessible markup. However, keep in mind that many accessibility issues can only be identified at runtime using other automated tools and by manually testing your application.

Some warnings may be incorrect in your concrete use case. You can disable such false positives by placing a `<!-- svelte-ignore a11y-<code> -->` comment above the line that causes the warning. Example:

```
<!-- svelte-ignore a11y-autofocus -->


```

Here is a list of accessibility checks Svelte will do for you.

a11y-accesskey

Enforce no `accesskey` on element. Access keys are HTML attributes that allow web developers to assign keyboard shortcuts to elements. Inconsistencies between keyboard shortcuts and keyboard commands used by screen reader and keyboard-only users create accessibility complications. To avoid complications, access keys should not be used.

```
<!-- A11y: Avoid using accesskey -->
<div accessKey="z" />
```

a11y-aria-activedescendant-has-tabindex

An element with `aria-activedescendant` must be tabbable, so it must either have an inherent `tabindex` or declare `tabindex` as an attribute.

```
<!-- A11y: Elements with attribute aria-activedescendant should have tabindex
value -->
<div aria-activedescendant="some-id" />
```

a11y-aria-attributes

Certain reserved DOM elements do not support ARIA roles, states and properties. This is often because they are not visible, for example `meta`, `html`, `script`, `style`. This rule enforces that these DOM elements do not contain the `aria-*` props.

```
<!-- A11y: <meta> should not have aria-* attributes -->
<meta aria-hidden="false" />
```

a11y-autofocus

Enforce that `autofocus` is not used on elements. Autofocusing elements can cause usability issues for sighted and non-sighted users alike.

```
<!-- A11y: Avoid using autofocus -->
<input autofocus />
```

ally-click-events-have-key-events

Enforce `on:click` is accompanied by at least one of the following: `on:keyup`, `on:keydown`, `on:key-press`. Coding for the keyboard is important for users with physical disabilities who cannot use a mouse, AT compatibility, and screenreader users.

This does not apply for interactive or hidden elements.

```
<!-- A11y: visible, non-interactive elements with an on:click event must be
accompanied by an on:keydown, on:keyup, or on:keypress event. -->
<div on:click={() => {}} />
```

Note that the `keypress` event is now deprecated, so it is officially recommended to use either the `keyup` or `keydown` event instead, accordingly.

ally-distracting-elements

Enforces that no distracting elements are used. Elements that can be visually distracting can cause accessibility issues with visually impaired users. Such elements are most likely deprecated, and should be avoided.

The following elements are visually distracting: `<marquee>` and `<blink>`.

```
<!-- A11y: Avoid <marquee> elements -->
<marquee />
```

ally-hidden

Certain DOM elements are useful for screen reader navigation and should not be hidden.

```
<!-- A11y: <h2> element should not be hidden -->
<h2 aria-hidden="true">invisible header</h2>
```

ally-img-redundant-alt

Enforce `img alt` attribute does not contain the word image, picture, or photo. Screen readers already announce `img` elements as an image. There is no need to use words such as *image*, *photo*, and/or *picture*.

```


<!-- aria-hidden, won't be announced by screen reader -->


<!-- A11y: Screen readers already announce <img> elements as an image. -->


<!-- A11y: Screen readers already announce <img> elements as an image. -->

```

```
<!-- A11y: Screen readers already announce <img> elements as an image. -->

```

ally-incorrect-aria-attribute-type

Enforce that only the correct type of value is used for aria attributes. For example, `aria-hidden` should only receive a boolean.

```
<!-- A11y: The value of 'aria-hidden' must be exactly one of true or false -->
<div aria-hidden="yes" />
```

ally-invalid-attribute

Enforce that attributes important for accessibility have a valid value. For example, `href` should not be empty, `'#'`, or `javascript:.`

```
<!-- A11y: '' is not a valid href attribute -->
<a href="">invalid</a>
```

ally-interactive-supports-focus

Enforce that elements with an interactive role and interactive handlers (mouse or key press) must be focusable or tabbable.

```
<!-- A11y: Elements with the 'button' interactive role must have a tabindex value. -->
<div role="button" on:keypress={() => {}} />
```

ally-label-has-associated-control

Enforce that a label tag has a text label and an associated control.

There are two supported ways to associate a label with a control:

- Wrapping a control in a label tag.
- Adding `for` to a label and assigning it the ID of an input on the page.

```
<label for="id">B</label>
<label>C <input type="text" /></label>
<!-- A11y: A form label must be associated with a control. -->
<label>A</label>
```

ally-media-has-caption

Providing captions for media is essential for deaf users to follow along. Captions should be a transcription or translation of the dialogue, sound effects, relevant musical cues, and other relevant audio information. Not only is this important for accessibility, but can also be useful for all users in the case that the media is unavailable (similar to `alt` text on an image when an image is unable to load).

The captions should contain all important and relevant information to understand the corresponding media. This may mean that the captions are not a 1:1 mapping of the dialogue in the media content. However, captions are not necessary for video components with the `muted` attribute.

```
<video><track kind="captions" /></video>

<audio muted />

<!-- A11y: Media elements must have a <track kind=\"captions\"> -->
<video />

<!-- A11y: Media elements must have a <track kind=\"captions\"> -->
<video><track /></video>
```

ally-misplaced-role

Certain reserved DOM elements do not support ARIA roles, states and properties. This is often because they are not visible, for example `meta`, `html`, `script`, `style`. This rule enforces that these DOM elements do not contain the `role` props.

```
<!-- A11y: <meta> should not have role attribute -->
<meta role="tooltip" />
```

ally-misplaced-scope

The scope attribute should only be used on `<th>` elements.

```
<!-- A11y: The scope attribute should only be used with <th> elements -->
<div scope="row" />
```

ally-missing-attribute

Enforce that attributes required for accessibility are present on an element. This includes the following checks:

- `<a>` should have an href (unless it's a [fragment-defining tag](#))
- `<area>` should have alt, aria-label, or aria-labelledby
- `<html>` should have lang
- `<iframe>` should have title
- `` should have alt
- `<object>` should have title, aria-label, or aria-labelledby

- `<input type="image">` should have alt, aria-label, or aria-labelledby

```
<!-- A11y: <input type=\"image\"> element should have an alt, aria-label or aria-  
labelledby attribute -->  
<input type="image" />  
  
<!-- A11y: <html> element should have a lang attribute -->  
<html />  
  
<!-- A11y: <a> element should have an href attribute -->  
<a>text</a>
```

ally-missing-content

Enforce that heading elements (`h1`, `h2`, etc.) and anchors have content and that the content is accessible to screen readers

```
<!-- A11y: <a> element should have child content -->  
<a href="/foo" />  
  
<!-- A11y: <h1> element should have child content -->  
<h1 />
```

ally-mouse-events-have-key-events

Enforce that `on:mouseover` and `on:mouseout` are accompanied by `on:focus` and `on:blur`, respectively. This helps to ensure that any functionality triggered by these mouse events is also accessible to keyboard users.

```
<!-- A11y: on:mouseover must be accompanied by on:focus -->  
<div on:mouseover={handleMouseover} />  
  
<!-- A11y: on:mouseout must be accompanied by on:blur -->  
<div on:mouseout={handleMouseout} />
```

ally-no-redundant-roles

Some HTML elements have default ARIA roles. Giving these elements an ARIA role that is already set by the browser [has no effect](#) and is redundant.

```
<!-- A11y: Redundant role 'button' -->  
<button role="button" />  
  
<!-- A11y: Redundant role 'img' -->  

```

ally-no-interactive-element-to-noninteractive-role

[WAI-ARIA](#) roles should not be used to convert an interactive element to a non-interactive element. Non-interactive ARIA roles include `article`, `banner`, `complementary`, `img`, `listitem`, `main`, `region` and `tooltip`.

```
<!-- A11y: <textarea> cannot have role 'listitem' -->
<textarea role="listitem" />
```

a11y-no-noninteractive-element-interactions

A non-interactive element does not support event handlers (mouse and key handlers). Non-interactive elements include `<main>`, `<area>`, `<h1>` (, `<h2>`, etc), `<p>`, ``, ``, `` and ``. Non-interactive [WAI-ARIA roles](#) include `article`, `banner`, `complementary`, `img`, `listitem`, `main`, `region` and `tooltip`.

```
<!-- `A11y: Non-interactive element <li> should not be assigned mouse or keyboard
event listeners.` -->
<li on:click={() => {}} />

<!-- `A11y: Non-interactive element <div> should not be assigned mouse or keyboard
event listeners.` -->
<div role="listitem" on:click={() => {}} />
```

a11y-no-noninteractive-element-to-interactive-role

[WAI-ARIA](#) roles should not be used to convert a non-interactive element to an interactive element. Interactive ARIA roles include `button`, `link`, `checkbox`, `menuitem`, `menuitemcheckbox`, `menuitemradio`, `option`, `radio`, `searchbox`, `switch` and `textbox`.

```
<!-- A11y: Non-interactive element <h3> cannot have interactive role 'searchbox' -
->
<h3 role="searchbox">Button</h3>
```

a11y-no-noninteractive-tabindex

Tab key navigation should be limited to elements on the page that can be interacted with.

```
<!-- A11y: noninteractive element cannot have nonnegative tabIndex value -->
<div tabindex="0" />
```

a11y-no-static-element-interactions

Elements like `<div>` with interactive handlers like `click` must have an ARIA role.

```
<!-- A11y: <div> with click handler must have an ARIA role -->
<div on:click={() => ''} />
```

a11y-positive-tabindex

Avoid positive `tabindex` property values. This will move elements out of the expected tab order, creating a confusing experience for keyboard users.

```
<!-- A11y: avoid tabindex values above zero -->
<div tabindex="1" />
```

ally-role-has-required-aria-props

Elements with ARIA roles must have all required attributes for that role.

```
<!-- A11y: A11y: Elements with the ARIA role "checkbox" must have the following
attributes defined: "aria-checked" -->
<span role="checkbox" aria-labelledby="foo" tabindex="0" />
```

ally-role-supports-aria-props

Elements with explicit or implicit roles defined contain only `aria-*` properties supported by that role.

```
<!-- A11y: The attribute 'aria-multiline' is not supported by the role 'link'. -->
<div role="link" aria-multiline />

<!-- A11y: The attribute 'aria-required' is not supported by the role 'listitem'.
This role is implicit on the element <li>. -->
<li aria-required />
```

ally-structure

Enforce that certain DOM elements have the correct structure.

```
<!-- A11y: <figcaption> must be an immediate child of <figure> -->
<div>
  <figcaption>Image caption</figcaption>
</div>
```

ally-unknown-aria-attribute

Enforce that only known ARIA attributes are used. This is based on the [WAI-ARIA States and Properties spec](#).

```
<!-- A11y: Unknown aria attribute 'aria-labeledby' (did you mean 'labelledby'?) --
>
<input type="image" aria-labeledby="foo" />
```

ally-unknown-role

Elements with ARIA roles must use a valid, non-abstract ARIA role. A reference to role definitions can be found at [WAI-ARIA](#) site.

```
<!-- A11y: Unknown role 'toooltip' (did you mean 'tooltip'?) -->
<div role="toooltip" />
```

TypeScript

You can use TypeScript within Svelte components. IDE extensions like the [Svelte VSCode 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.

Setup

To use TypeScript 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.

```
/// 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.

```
/// 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

`<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>
```

Props

Props can be typed directly on the `export let` statement:

```
<script lang="ts">
  export let name: string;
</script>
```

Slots

Slot and slot prop types are inferred from the types of the slot props passed to them:

```
<script lang="ts">
  export let name: string;
</script>

<slot {name} />

<!-- Later -->
<Comp let:name>
  <!-- ^ Inferred as string -->
  {name}
</Comp>
```

Events

Events can be typed with `createEventDispatcher`:

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

  const dispatch = createEventDispatcher<{
    event: null; // does not accept a payload
    click: string; // has a required string payload
    type: string | null; // has an optional string payload
 }>();

  function handleClick() {
```

```

    dispatch('event');
    dispatch('click', 'hello');
  }

  function handleType() {
    dispatch('event');
    dispatch('type', Math.random() > 0.5 ? 'world' : null);
  }
</script>

<button on:click={handleClick} on:keydown={handleType}>Click</button>

```

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/**/*.ts"]` and your `d.ts` file is inside `src`, it should work. You may need to reload for the changes to take effect.

Since Svelte version 4.2 / `svelte-check` version 3.5 / VS Code extension version 107.10.0 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

```

Experimental advanced typings

A few features are missing from taking full advantage of TypeScript in more advanced use cases like typing that a component implements a certain interface, explicitly typing slots, or using generics. These things are possible using experimental advanced type capabilities. See [this RFC](#) for more information on how to make use of them.

The API is experimental and may change at any point

Limitations

No TS in markup

You cannot use TypeScript in your template's markup. For example, the following does not work:

```

<script lang="ts">
  let count = 10;
</script>

<h1>Count as string: {count as string}!</h1> <!-- ✗ Does not work -->
{#if count > 4}
  {@const countString: string = count} <!-- ✗ Does not work -->
  {countString}
{/if}

```

Reactive Declarations

You cannot type your reactive declarations with TypeScript in the way you type a variable. For example, the following does not work:

```

<script lang="ts">
  let count = 0;

  $: doubled: number = count * 2; // ✗ Does not work
</script>

```

You cannot add a `: TYPE` because it's invalid syntax in this position. Instead, you can move the definition to a `let` statement just above:

```

<script lang="ts">
  let count = 0;

```

```
    let doubled: number;  
    $: doubled = count * 2;  
</script>
```

Types

TYPES: svelte

Svelte 4 migration guide

This migration guide provides an overview of how to migrate from Svelte version 3 to 4. See the linked PRs for more details about each change. Use the migration script to migrate some of these automatically: `npx`

`svelte-migrate@latest svelte-4`

If you're a library author, consider whether to only support Svelte 4 or if it's possible to support Svelte 3 too. Since most of the breaking changes don't affect many people, this may be easily possible. Also remember to update the version range in your `peerDependencies`.

Minimum version requirements

- Upgrade to Node 16 or higher. Earlier versions are no longer supported. ([#8566](#))
- If you are using SvelteKit, upgrade to 1.20.4 or newer ([sveltejs/k10172](#))
- If you are using Vite without SvelteKit, upgrade to `vite-plugin-svelte` 2.4.1 or newer ([#8516](#))
- If you are using webpack, upgrade to webpack 5 or higher and `svelte-loader` 3.1.8 or higher. Earlier versions are no longer supported. ([#8515](#), [198dbcf](#))
- If you are using Rollup, upgrade to `rollup-plugin-svelte` 7.1.5 or higher ([198dbcf](#))
- If you are using TypeScript, upgrade to TypeScript 5 or higher. Lower versions might still work, but no guarantees are made about that. ([#8488](#))

Browser conditions for bundlers

Bundlers must now specify the `browser` condition when building a frontend bundle for the browser. SvelteKit and Vite will handle this automatically for you. If you're using any others, you may observe lifecycle callbacks such as `onMount` not get called and you'll need to update the module resolution configuration.

- For Rollup this is done within the `@rollup/plugin-node-resolve` plugin by setting `browser: true` in its options. See the `rollup-plugin-svelte` documentation for more details
- For webpack this is done by adding `"browser"` to the `conditionNames` array. You may also have to update your `alias` config, if you have set it. See the `svelte-loader` documentation for more details

([#8516](#))

Removal of CJS related output

Svelte no longer supports the CommonJS (CJS) format for compiler output and has also removed the `svelte/register` hook and the CJS runtime version. If you need to stay on the CJS output format, consider using a bundler to convert Svelte's ESM output to CJS in a post-build step. ([#8613](#))

Stricter types for Svelte functions

There are now stricter types for `createEventDispatcher`, `Action`, `ActionReturn`, and `onMount`:

- `createEventDispatcher` now supports specifying that a payload is optional, required, or non-existent, and the call sites are checked accordingly ([#7224](#))

```
// @errors: 2554 2345
import { createEventDispatcher } from 'svelte';

const dispatch = createEventDispatcher<{
  optional: number | null;
  required: string;
  noArgument: null;
}>();

// Svelte version 3:
dispatch('optional');
dispatch('required'); // I can still omit the detail argument
dispatch('noArgument', 'surprise'); // I can still add a detail argument

// Svelte version 4 using TypeScript strict mode:
dispatch('optional');
dispatch('required'); // error, missing argument
dispatch('noArgument', 'surprise'); // error, cannot pass an argument
```

- `Action` and `ActionReturn` have a default parameter type of `undefined` now, which means you need to type the generic if you want to specify that this action receives a parameter. The migration script will migrate this automatically ([#7442](#))

```
-const action: Action = (node, params) => { .. } // this is now an error if you
  use params in any way
+const action: Action<HTMLElement, string> = (node, params) => { .. } // params is
  of type string
```

- `onMount` now shows a type error if you return a function asynchronously from it, because this is likely a bug in your code where you expect the callback to be called on destroy, which it will only do for synchronously returned functions ([#8136](#))

```
// Example where this change reveals an actual bug
onMount(
  - // someCleanup() not called because function handed to onMount is async
  - async () => {
  -   const something = await foo();
  + // someCleanup() is called because function handed to onMount is sync
  + () => {
  +   foo().then(something => ..
    // ..
    return () => someCleanup();
  }
);
```

Custom Elements with Svelte

The creation of custom elements with Svelte has been overhauled and significantly improved. The `tag` option is deprecated in favor of the new `customElement` option:

```
-<svelte:options tag="my-component" />
+<svelte:options customElement="my-component" />
```

This change was made to allow [more configurability](#) for advanced use cases. The migration script will adjust your code automatically. The update timing of properties has changed slightly as well. ([#8457](#))

SvelteComponentTyped is deprecated

`SvelteComponentTyped` is deprecated, as `SvelteComponent` now has all its typing capabilities. Replace all instances of `SvelteComponentTyped` with `SvelteComponent`.

```
- import { SvelteComponentTyped } from 'svelte';
+ import { SvelteComponent } from 'svelte';

- export class Foo extends SvelteComponentTyped<{ aProp: string }> {}
+ export class Foo extends SvelteComponent<{ aProp: string }> {}
```

If you have used `SvelteComponent` as the component instance type previously, you may see a somewhat opaque type error now, which is solved by changing `: typeof SvelteComponent` to `: typeof SvelteComponent<any>`.

```
<script>
  import ComponentA from './ComponentA.svelte';
  import ComponentB from './ComponentB.svelte';
  import { SvelteComponent } from 'svelte';

  - let component: typeof SvelteComponent;
  + let component: typeof SvelteComponent<any>;

  function choseRandomly() {
    component = Math.random() > 0.5 ? ComponentA : ComponentB;
  }
</script>

<button on:click={choseRandomly}>random</button>
<svelte:element this={component} />
```

The migration script will do both automatically for you. ([#8512](#))

Transitions are local by default

Transitions are now local by default to prevent confusion around page navigations. "local" means that a transition will not play if it's within a nested control flow block (`each/if/await/key`) and not the direct parent block but a block above it is created/destroyed. In the following example, the `slide` intro animation will only play when `success` goes from `false` to `true`, but it will *not* play when `show` goes from `false` to `true`:

```
{#if show}
  ...
  {#if success}
    <p in:slide>Success</p>
  {/each}
{/if}
```

To make transitions global, add the `|global` modifier - then they will play when *any* control flow block above is created/destroyed. The migration script will do this automatically for you. ([#6686](#))

Default slot bindings

Default slot bindings are no longer exposed to named slots and vice versa:

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

<Nested let:count>
  <p>
    count in default slot - is available: {count}
  </p>
  <p slot="bar">
    count in bar slot - is not available: {count}
  </p>
</Nested>
```

This makes slot bindings more consistent as the behavior is undefined when for example the default slot is from a list and the named slot is not. ([#6049](#))

Preprocessors

The order in which preprocessors are applied has changed. Now, preprocessors are executed in order, and within one group, the order is markup, script, style.

```
// @errors: 2304
import { preprocess } from 'svelte/compiler';

const { code } = await preprocess(
  source,
  [
    {
      markup: () => {
        console.log('markup-1');
      },
      script: () => {
        console.log('script-1');
      },
      style: () => {
        console.log('style-1');
      }
    },
    {
      markup: () => {
        console.log('markup-2');
      },
      script: () => {
        console.log('script-2');
      },
      style: () => {
        console.log('style-2');
      }
    }
  ]
);
```



```

    ],
    {
      filename: 'App.svelte'
    }
  );

  // Svelte 3 logs:
  // markup-1
  // markup-2
  // script-1
  // script-2
  // style-1
  // style-2

  // Svelte 4 logs:
  // markup-1
  // script-1
  // style-1
  // markup-2
  // script-2
  // style-2

```

This could affect you for example if you are using `MDsvex` - in which case you should make sure it comes before any script or style preprocessor.

```

preprocess: [
  - vitePreprocess(),
  - mdsvex(mdsvexConfig)
  + mdsvex(mdsvexConfig),
  + vitePreprocess()
]

```

Each preprocessor must also have a name. ([#8618](#))

New eslint package

`eslint-plugin-svelte3` is deprecated. It may still work with Svelte 4 but we make no guarantees about that. We recommend switching to our new package `eslint-plugin-svelte`. See [this Github post](#) for an instruction how to migrate. Alternatively, you can create a new project using `npm create svelte@latest`, select the eslint (and possibly TypeScript) option and then copy over the related files into your existing project.

Other breaking changes

- the `inert` attribute is now applied to outgoing elements to make them invisible to assistive technology and prevent interaction. ([#8628](#))
- the runtime now uses `classList.toggle(name, boolean)` which may not work in very old browsers. Consider using a [polyfill](#) if you need to support these browsers. ([#8629](#))
- the runtime now uses the `CustomEvent` constructor which may not work in very old browsers. Consider using a [polyfill](#) if you need to support these browsers. ([#8775](#))

- people implementing their own stores from scratch using the `StartStopNotifier` interface (which is passed to the create function of `writable` etc) from `svelte/store` now need to pass an update function in addition to the set function. This has no effect on people using stores or creating stores using the existing Svelte stores. ([#6750](#))
- `derived` will now throw an error on falsy values instead of stores passed to it. ([#7947](#))
- type definitions for `svelte/internal` were removed to further discourage usage of those internal methods which are not public API. Most of these will likely change for Svelte 5
- Removal of DOM nodes is now batched which slightly changes its order, which might affect the order of events fired if you're using a `MutationObserver` on these elements ([#8763](#))
- if you enhanced the global typings through the `svelte.JSX` namespace before, you need to migrate this to use the `svelteHTML` namespace. Similarly if you used the `svelte.JSX` namespace to use type definitions from it, you need to migrate those to use the types from `svelte/elements` instead. You can find more information about what to do [here](#)

'svelte/register'

This API is removed in Svelte 4. `require` hooks are deprecated and current Node versions understand ESM. Use a bundler like Vite or our full-stack framework [SvelteKit](#) instead to create JavaScript modules from Svelte components.

To render Svelte components in Node.js without bundling, use `require('svelte/register')`. After that, you can use `require` to include any `.svelte` file.

```
// @noErrors
require('svelte/register');

const App = require('./App.svelte').default;

// ...

const { html, css, head } = App.render({ answer: 42 });
```

The `.default` is necessary because we're converting from native JavaScript modules to the CommonJS modules recognised by Node. Note that if your component imports JavaScript modules, they will fail to load in Node and you will need to use a bundler instead.

To set compile options, or to use a custom file extension, call the `register` hook as a function:

```
// @noErrors
require('svelte/register')({
  extensions: ['.customextension'], // defaults to ['.html', '.svelte']
  preserveComments: true
});
```

Colophon

This book is created by using the following sources:

- Svelte - English
- GitHub source: [sveltejs/svelte/documentation/docs](https://github.com/sveltejs/svelte/documentation/docs)
- Created: 2023-08-25
- 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>