SVELTE Docs - English

Table of contents

Before we begin

This page contains detailed API reference documentation. It's intended to be a resource for people who already have some familiarity with Svelte.

If that's not you (yet), you may prefer to visit the interactive tutorial or the examples before consulting this reference.

Don't be shy about asking for help in the Discord chatroom.

Using an older version of Svelte? Have a look at the v2 docs.

Go to TOC

Getting started

To try Svelte in an interactive online environment you can try the REPL or StackBlitz.

To create a project locally, run:

```
npm create vite@latest myapp -- --template svelte
cd myapp
npm install
npm run dev
```

Or use SvelteKit, the official application framework from the Svelte team (currently in release candidate status).

The Svelte team maintains a VS Code extension and the Svelte community has created a list of integrations with various other tooling and editors.

If you're having trouble, get help on Discord or StackOverflow.

Go to TOC

Component format

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 whenever a prop is removed by the consumer, its value is set to `undefined` rather than the 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.

```
<script>
    // these are readonly
    export const thisIs = 'readonly';

    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.

```
<script>
    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`.

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

    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;

    const 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.

```
<script>
    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

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

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

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

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

<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;

    // this allows an importer to do 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 {...}
</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>
```

Go to TOC

Template syntax

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}`.

```
<!-- These are equivalent -->
<button disabled={disabled}>...</button>
<button {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`. It is not generally recommended, as it is difficult for Svelte to optimise. But it can be useful in rare cases – for example, when you don't know at compile time what props might be passed to a component.

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

`$$restProps` contains only the props which are *not* declared with `export`. It can be used to pass down other unknown attributes to an element in a component. It shares the same optimisation problems as `$$props`, and is likewise not recommended.

```
<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.

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>
```

{#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}
```

{#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}
```

{#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}
```

{@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.

```
<script>
    let count = 0;

    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
- `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:

```
<script>
    let counter = 0;
    function increment() {
        counter = counter + 1;
    }

    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}></textarea>

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

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

```
<!-- These are equivalent -->
<input bind:value={value}>
<input bind: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.

```
<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 `innerHTML` and `textContent` bindings.

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

`<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 — six *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

...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:currentTime
    bind:playbackRate
    bind:paused
    bind:volume
    bind:muted
    bind:videoWidth
    bind:videoHeight
></video>
```

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`.

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

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

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

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

    let canvasElement;

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

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

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="{active ? 'active' : ''}">...</div>
<div class:active={active}>...</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}
```

```
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:

```
<script>
    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></div>
```

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.

```
<script>
    export let bar;

    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}></div>
```

transition:*fn*

```
transition:fn
```

```
transition:fn={params}
```

```
transition:fn|local
```

```
transition:fn|local={params}
```

```
transition = (node: HTMLElement, params: any) => {
    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}
```

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

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.

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

    export let visible;

    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.

```
<script>
    export let visible = false;

    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 microtask. This allows multiple transitions to coordinate, making crossfade effects possible.

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}
```

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

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

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

in:*fn*/out:*fn*

```
in:fn
```

```
in:fn={params}
```

```
in:fn|local
```

```
in:fn|local={params}
```

```
out:fn
```

```
out:fn={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 outroed while the transition is in progress. If an out transition is aborted, transitions will restart from scratch.

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

animate:*fn*

```
animate:name
```

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

```
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
}
```

```
DOMRect {
    bottom: number,
    height: number,
    left: number,
    right: number,
    top: number,
    width: number,
    x: number,
    y: number
}
```

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

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

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

Animation Parameters

As with actions and transitions, animations can have parameters.

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

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

Custom animation functions

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

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

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

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

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

    function whizz(node, { from, to }, params) {

        const dx = from.left - to.left;
        const dy = from.top - to.top;

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

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

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

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

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

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

    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. 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 component will *forward* the event, meaning that a consumer of the component 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}/>
```

bind:this

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

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

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

```
<ShoppingCart bind:this={cart}/>

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

`<slot>`

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

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

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

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

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

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

<slot name="footer"></slot>

<!-- 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>
    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 tag (e.g., `br`) and `<svelte:element>` has child elements, a runtime error will be thrown in development mode.

```
<script>
    let tag = 'div';
    export let handler;
</script>

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

`<svelte: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>
    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

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: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>`, 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>` 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>
    <link rel="stylesheet" href="/tutorial/dark-theme.css">
</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
- `tag="..."` — the name to use when compiling this component as a custom element

```
<svelte:options tag="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"></slot>
</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>
```

Go to TOC

Run time

`svelte`

The `svelte` package exposes lifecycle functions and the context API.

`onMount`

```
onMount(callback: () => void)
```

```
onMount(callback: () => () => void)
```

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

`beforeUpdate`

```
beforeUpdate(callback: () => void)
```

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`

```
afterUpdate(callback: () => void)
```

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`

```
onDestroy(callback: () => void)
```

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`

```
promise: Promise = tick()
```

Returns a promise that resolves once any pending state changes have been applied, or in the next microtask if there are none.

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

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

`setContext`

```
setContext(key: any, context: any)
```

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`

```
context: any = getContext(key: any)
```

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`

```
hasContext: boolean = hasContext(key: any)
```

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

```
contexts: Map<any, any> = 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

```
dispatch: ((name: string, detail?: any, options?: DispatchOptions) => boolean) =
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>
```

`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`

```
store = writable(value?: any)
```

```
store = writable(value?: any, start?: (set: (value: any) => void) => () => void)
```

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.

```
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. It must return a `stop` function that is called when the subscriber count goes from one to zero.

```
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`

```
store = readable(value?: any, start?: (set: (value: any) => void) => () => void)
```

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

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

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

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

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

`derived`

```
store = derived(a, callback: (a: any) => any)
```

```
store = derived(a, callback: (a: any, set: (value: any) => void) => void | () =>
void, initial_value: any)
```

```
store = derived([a, ...b], callback: ([a: any, ...b: any[]]) => any)
store = derived([a, ...b], callback: ([a: any, ...b: any[]], set: (value: any) =>
void) => void | () => void, initial_value: any)
```

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.

```
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 calling it when appropriate.

In this case, you can also pass a third argument to `derived` — the initial value of the derived store before `set` is first called.

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

const delayed = derived(a, ($a, set) => {
    setTimeout(() => set($a), 1000);
}, 'one moment...');
```

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

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

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

    return () => {
        clearInterval(interval);
    };
}, 'one moment...');
```

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

```
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);
});
```

`get`

```
value: any = get(store)
```

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.

```js
import { get } from 'svelte/store';

const value = get(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`

```js
store = tweened(value: any, options)
```

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.

```
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`

```
store = spring(value: any, options)
```

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.

```
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 }` .

```
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).

```
const size = spring();
$: $size = big ? 100 : 10;
```

`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`

```
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`

```
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` , default 5) - the size of the blur in pixels

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

{#if condition}
    <div transition:blur="{{amount: 10}}">
```

```
        fades in and out
    </div>
{/if}
```

`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 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.

`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` , default 0) - the x offset to animate out to and in from
- `y` (`number` , 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

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`

```
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

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

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

`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`

```
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: `duration = length / 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 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`

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}
```

`svelte/animate`

The `svelte/animate` module exports one function for use with Svelte animations.

`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}
```

`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	`backIn`	`backOut`	`backInOut`
bounce	`bounceIn`	`bounceOut`	`bounceInOut`
circ	`circIn`	`circOut`	`circInOut`
cubic	`cubicIn`	`cubicOut`	`cubicInOut`
elastic	`elasticIn`	`elasticOut`	`elasticInOut`
expo	`expoIn`	`expoOut`	`expoInOut`
quad	`quadIn`	`quadOut`	`quadInOut`
quart	`quartIn`	`quartOut`	`quartInOut`
quint	`quintIn`	`quintOut`	`quintInOut`
sine	`sineIn`	`sineOut`	`sineInOut`

`svelte/register`

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

```
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:

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

Client-side component API

Creating a component

```
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.

```js
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
`target`	**none**	An `HTMLElement` or `ShadowRoot` to render to. This option is required
`anchor`	`null`	A child of `target` to render the component immediately before
`props`	`{}`	An object of properties to supply to the component
`context`	`new Map()`	A `Map` of root-level context key-value pairs to supply to the component
`hydrate`	`false`	See below
`intro`	`false`	If `true`, 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.

```js
import App from './App.svelte';

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

`$set`

```js
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.

```js
component.$set({ answer: 42 });
```

`$on`

```
component.$on(event, 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.

```
const off = app.$on('selected', event => {
    console.log(event.detail.selection);
});

off();
```

`$destroy`

```
component.$destroy()
```

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

Component props

```
component.prop
```

```
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.

```
console.log(app.count);
app.count += 1;
```

Custom element 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 tag="my-element" />

<script>
    export let name = 'world';
</script>

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

Alternatively, use `tag={null}` to indicate that the consumer of the custom element should name it.

```
import MyElement from './MyElement.svelte';

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>`.

```
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';
```

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*. 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
- Polyfills are required to support older browsers

Server-side component API

```
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`.

```
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 | | | --- | --- | | `props` | `{}` | An object of properties to supply to the component | `options` | `{}` | An object of options

The `options` object takes in the following options:

| option | default | description | | | --- | --- | | `context` | `new Map()` | A `Map` of root-level context key-value pairs to supply to the component

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

Go to TOC

Compile time

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.

`svelte.compile`

```
result: {
    js,
    css,
    ast,
    warnings,
    vars,
    stats
} = svelte.compile(source: string, options?: {...})
```

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

```
const svelte = require('svelte/compiler');

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

The following options can be passed to the compiler. None are required:

| option | default | description | | | --- | --- | | `filename` | `null` | `string` used for debugging hints and sourcemaps. Your bundler plugin will set it automatically. | `name` | `"Component"` | `string` that sets the name of the resulting JavaScript class (though the compiler will rename it if it would otherwise conflict with other variables in scope). It will normally be inferred from `filename`. | `format` | `"esm"` | If `"esm"`, creates a JavaScript module (with `import` and `export`). If `"cjs"`, creates a CommonJS module (with `require` and `module.exports`), which is useful in some server-side rendering situations or for testing. | `generate` | `"dom"` | If `"dom"`, Svelte emits a JavaScript class for mounting to the DOM. If `"ssr"`, Svelte emits an object with a `render` method suitable for server-side rendering. If `false`, no JavaScript or CSS is returned; just metadata. | `errorMode` | `"throw"` | If `"throw"`, Svelte throws when a compilation error occurred. If `"warn"`, Svelte will treat errors as warnings and add them to the warning report. | `varsReport` | `"strict"` | If `"strict"`, Svelte returns a variables report with only variables that are not globals nor internals. If `"full"`, Svelte returns a variables report with all detected variables. If `false`, no variables report is returned. | `dev` | `false` | If `true`, causes extra code to be added to components that will perform runtime checks and provide debugging information during development. | `immutable` | `false` | If

`true`, tells the compiler that you promise not to mutate any objects. This allows it to be less conservative about checking whether values have changed. | `hydratable` | `false` | If `true` when generating DOM code, enables the `hydrate: true` runtime option, which allows a component to upgrade existing DOM rather than creating new DOM from scratch. When generating SSR code, this adds markers to `<head>` elements so that hydration knows which to replace. | `legacy` | `false` | If `true`, generates code that will work in IE9 and IE10, which don't support things like `element.dataset`. | `accessors` | `false` | If `true`, getters and setters will be created for the component's props. If `false`, they will only be created for readonly exported values (i.e. those declared with `const`, `class` and `function`). If compiling with `customElement: true` this option defaults to `true`. | `customElement` | `false` | If `true`, tells the compiler to generate a custom element constructor instead of a regular Svelte component. | `tag` | `null` | A `string` that tells Svelte what tag name to register the custom element with. It must be a lowercase alphanumeric string with at least one hyphen, e.g. `"my-element"`. | `css` | `'injected'` | If `'injected'` (formerly `true`), styles will be included in the JavaScript class and injected at runtime for the components actually rendered. If `'external'` (formerly `false`), the CSS will be returned in the `css` field of the compilation result. Most Svelte bundler plugins will set this to `'external'` and use the CSS that is statically generated for better performance, as it will result in smaller JavaScript bundles and the output can be served as cacheable `.css` files. If `'none'`, styles are completely avoided and no CSS output is generated. | `cssHash` | See right | A function that takes a `{ hash, css, name, filename }` argument and returns the string that is used as a classname for scoped CSS. It defaults to returning `svelte-${hash(css)}` | `loopGuardTimeout` | 0 | A `number` that tells Svelte to break the loop if it blocks the thread for more than `loopGuardTimeout` ms. This is useful to prevent infinite loops. **Only available when `dev: true`** | `preserveComments` | `false` | If `true`, your HTML comments will be preserved during server-side rendering. By default, they are stripped out. | `preserveWhitespace` | `false` | If `true`, whitespace inside and between elements is kept as you typed it, rather than removed or collapsed to a single space where possible. | `sourcemap` | `object \| string` | An initial sourcemap that will be merged into the final output sourcemap. This is usually the preprocessor sourcemap. | `enableSourcemap` | `boolean \| { js: boolean; css: boolean; }` | If `true`, Svelte generate sourcemaps for components. Use an object with `js` or `css` for more granular control of sourcemap generation. By default, this is `true`. | `outputFilename` | `null` | A `string` used for your JavaScript sourcemap. | `cssOutputFilename` | `null` | A `string` used for your CSS sourcemap. | `sveltePath` | `"svelte"` | The location of the `svelte` package. Any imports from `svelte` or `svelte/[module]` will be modified accordingly. | `namespace` | `"html"` | The namespace of the element; e.g., `"mathml"`, `"svg"`, `"foreign"`.

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

```
const {
    js,
    css,
    ast,
    warnings,
    vars,
    stats
} = svelte.compile(source);
```

- `js` and `css` are objects with the following properties:
  - `code` is a JavaScript string

  - `map` is a sourcemap with additional `toString()` and `toUrl()` convenience methods
- `ast` is an abstract syntax tree representing the structure of your component.
- `warnings` is an array of warning objects that were generated during compilation. Each warning has several properties:
  - `code` is a string identifying the category of warning
  - `message` describes the issue in human-readable terms
  - `start` and `end`, if the warning relates to a specific location, are objects with `line`, `column` and `character` properties
  - `frame`, if applicable, is a string highlighting the offending code with line numbers
- `vars` is an array of the component's declarations, used by eslint-plugin-svelte3 for example. Each variable has several properties:
  - `name` is self-explanatory
  - `export_name` is the name the value is exported as, if it is exported (will match `name` unless you do `export...as`)
  - `injected` is `true` if the declaration is injected by Svelte, rather than in the code you wrote
  - `module` is `true` if the value is declared in a `context="module"` script
  - `mutated` is `true` if the value's properties are assigned to inside the component
  - `reassigned` is `true` if the value is reassigned inside the component
  - `referenced` is `true` if the value is used in the template
  - `referenced_from_script` is `true` if the value is used in the `<script>` outside the declaration
  - `writable` is `true` if the value was declared with `let` or `var` (but not `const`, `class` or `function`)
- `stats` is an object used by the Svelte developer team for diagnosing the compiler. Avoid relying on it to stay the same!

`svelte.parse`

```
ast: object = svelte.parse(
	source: string,
	options?: {
		filename?: string,
		customElement?: boolean
	}
)
```

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.

```
const svelte = require('svelte/compiler');

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

`svelte.preprocess`

A number of 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.

```
result: {
    code: string,
    dependencies: Array<string>
} = await svelte.preprocess(
    source: string,
    preprocessors: Array<{
        markup?: (input: { content: string, filename: string }) => Promise<{
            code: string,
            dependencies?: Array<string>
        }>,
        script?: (input: { content: string, markup: string, attributes:
Record<string, string>, filename: string }) => Promise<{
            code: string,
            dependencies?: Array<string>
        }>,
        style?: (input: { content: string, markup: string, attributes:
Record<string, string>, filename: string }) => Promise<{
            code: string,
            dependencies?: Array<string>
        }>
    }>,
    options?: {
        filename?: string
    }
)
```

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 `markup`, `script` and `style` functions, each of which is optional.

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, and an optional array of `dependencies`.

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

> Preprocessor functions should additionally return a `map` object alongside `code` and `dependencies`, where `map` is a sourcemap representing the transformation.

```
const svelte = require('svelte/compiler');
const MagicString = require('magic-string');
const { code } = await svelte.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'
});
```

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.

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

```
const svelte = require('svelte/compiler');
const sass = require('node-sass');
const { dirname } = require('path');

const { code, dependencies } = await svelte.preprocess(source, {
    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);
        }));

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

Multiple preprocessors can be used together. The output of the first becomes the input to the second. `markup` functions run first, then `script` and `style`.

```
const svelte = require('svelte/compiler');

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

`svelte.walk`

```
walk(ast: Node, {
    enter(node: Node, parent: Node, prop: string, index: number)?: void,
    leave(node: Node, parent: Node, prop: string, index: number)?: void
})
```

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.

```
const svelte = require('svelte/compiler');
svelte.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);
    }
});
```

`svelte.VERSION`

The current version, as set in package.json.

```
const svelte = require('svelte/compiler');
console.log(`running svelte version ${svelte.VERSION}`);
```

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.

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'></div>
```

`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>
```

`a11y-click-events-have-key-events`

Enforce `on:click` is accompanied by at least one of the following: `onKeyUp`, `onKeyDown`, `onKeyPress`. 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={() => {}} />
```

`a11y-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 />
```

`a11y-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>
```

`a11y-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*.

```
<img src="foo" alt="Foo eating a sandwich." />

<!-- aria-hidden, won't be announced by screen reader -->
<img src="bar" aria-hidden="true" alt="Picture of me taking a photo of an image"
/>

<!-- A11y: Screen readers already announce <img> elements as an image. -->
<img src="foo" alt="Photo of foo being weird." />

<!-- A11y: Screen readers already announce <img> elements as an image. -->
<img src="bar" alt="Image of me at a bar!" />

<!-- A11y: Screen readers already announce <img> elements as an image. -->
<img src="foo" alt="Picture of baz fixing a bug." />
```

`a11y-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"/>
```

`a11y-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>
```

`a11y-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>
```

`a11y-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></audio>

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

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

`a11y-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">
```

`a11y-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" />
```

`a11y-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
- `<img>` 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></html>

<!-- A11y: <a> element should have an href attribute -->
<a>text</a>
```

`a11y-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'></a>

<!-- A11y: <h1> element should have child content -->
<h1></h1>
```

`a11y-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} />
```

a11y-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' -->
<img role="img" src="foo.jpg" />
```

a11y-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-tabindex

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

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

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'/>
```

a11y-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"></span>
```

`a11y-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>
```

`a11y-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">
```

`a11y-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"></div>
```

Go to TOC

Colophon

This book is created by using the following sources:

- Svelte - English
- GitHub source: sveltejs/svelte/site/content/docs
- Created: 2022-11-29
- 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