`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`. 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) => void) = 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 and are not cancellable with `event.preventDefault()`. 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}"/>
```

`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.*

```
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`

```
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

---

As with [`tweened`] stores, `set` and `update` return a Promise that resolves if the spring settles. The `store.stiffness` and `store.damping` properties can be changed while the spring is in motion, and will take immediate effect.

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 }` .

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.

```
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.

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

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

`$set`

```
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.

```
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']])
    }
);
```