GETTING STARTED

Introduction

Edit this page on GitHub

ON THIS PAGE

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

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

Start a new project

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

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

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

Alternatives to SvelteKit

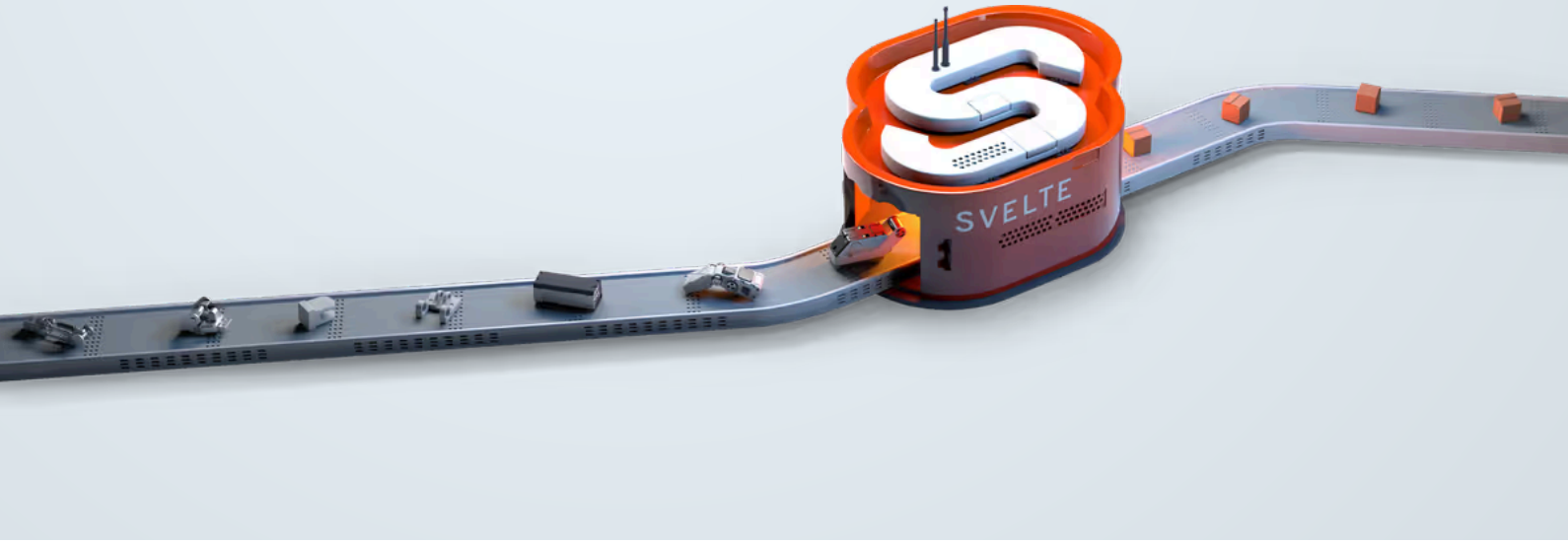
If you don't want to use SvelteKit for some reason, you can also use Svelte with Vite (but without SvelteKit) by running `npm create vite@latest` and selecting the `svelte` option. With this, `npm run build` will generate HTML, JS and CSS files inside the `dist` directory. In most cases, you will probably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

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

SVELTE

cybernetically enhanced
web apps

tutorial ↗ read the docs



compiled

Svelte shifts as much work as possible out of the browser and into your build step. No more manual optimisations — just faster, more efficient apps.

compact

Write breathtakingly concise components using languages you already know — HTML, CSS and JavaScript. Oh, and your application bundles will be tiny as well.

complete

Built-in scoped styling, state management, motion primitives, form bindings and more — don't waste time trawling npm for the bare essentials. It's all here.

GETTING STARTED

Introduction

Edit this page on GitHub

ON THIS PAGE

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

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

Start a new project

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

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

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

Alternatives to SvelteKit

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

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

INTRODUCTION

Dynamic attributes

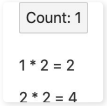
Styling

Nested components

HTML tags

REACTIVITY

Reactive assignments

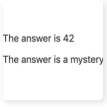


Reactive declarations

Reactive statements

PROPS

Declaring props



Default values

Spread props

LOGIC

If blocks

Else blocks

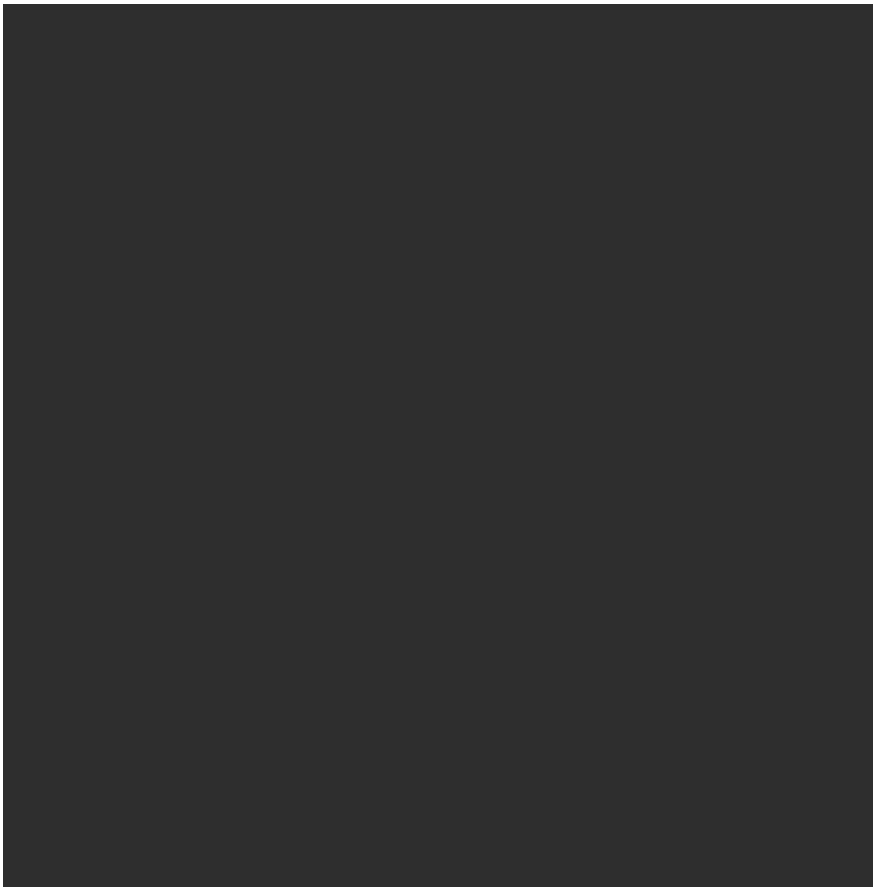
App.svelte +

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

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

```

Result JS output CSS output



Console CLEAR

App.svelte

Result JS output CSS output AST output

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

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

```

Hello world!

Console CLEAR

LATEST POST • 2024-07-01

What's new in Svelte: July 2024

svelte/events, simpler elements and more optional options

OLDER POSTS

What's new in Svelte: June 2024

Better `bind`s, migration tooling and a new comparison rune

What's new in Svelte: May 2024

Svelte 5 Release Candidate and all the other highlights from Svelte Summit Spring

Svelte 5 Release Candidate

We're almost there

What's new in Svelte: April 2024

Svelte Summit Spring on April 27! Plus: reactive `Map`, `Date` and `Set`

What's new in Svelte: March 2024

Nested CSS support and a much cleaner client-side API for Svelte 5

What's new in Svelte: February 2024

New in Kit: `reroute`, `emulate` and more!

Discord

You've been invited to join

Svelte

6,202 Online 63,638 Members

DISPLAY NAME

What should everyone call you?

This is how others see you. You can use special characters and emoji.

Continue

By registering, you agree to Discord's Terms of Service and Privacy Policy.

Already have an account?

GETTING STARTED

Introduction

Edit this page on GitHub

ON THIS PAGE

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

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

Start a new project

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

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

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

Alternatives to SvelteKit

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

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

TEMPLATE SYNTAX

Svelte components

Edit this page on GitHub

ON THIS PAGE

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
```

TEMPLATE SYNTAX

Basic markup

Edit this page on GitHub

ON THIS PAGE

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>
```

Strings can be JavaScript expressions.

TEMPLATE SYNTAX

Logic blocks

Edit this page on GitHub

ON THIS PAGE

{#if ...}

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

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

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

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

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

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

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

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

{#each ...}


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

TEMPLATE SYNTAX

Special tags

Edit this page on GitHub

ON THIS PAGE

{@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.

NOTE

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 = {
```

TEMPLATE SYNTAX

Element directives

Edit this page on GitHub

ON THIS PAGE

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.

App.svelte

```
<script>
  let count = 0;

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

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

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

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

Component directives

ON THIS PAGE

on:*eventname*

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

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

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

  const dispatch = createEventDispatcher();
</script>

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

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

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

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

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

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

--style-props

```
--style-props="anycssvalue"
```

TEMPLATE SYNTAX

Special elements

Edit this page on GitHub

ON THIS PAGE

<slot>

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

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

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

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

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

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

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

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

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

<slot name="*name*">

RUNTIME

svelte

Edit this page on GitHub

ON THIS PAGE

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

onMount

```
function onMount<T>(
  fn: () =>
    | NotFunction<T>
    | Promise<NotFunction<T>>
    | (() => any)
): 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');
```

RUNTIME

svelte/store

Edit this page on GitHub

ON THIS PAGE

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

```
function writable<T>(
  value?: T | undefined,
  start?: StartStopNotifier<T> | undefined
): Writable<T>;
```

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.

store.js

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

const count = writable(0);

count.subscribe((value) => {
  console.log(value);
```

RUNTIME

svelte/motion

Edit this page on GitHub

ON THIS PAGE

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

```
function tweened<T>(
  value?: T | undefined,
  defaults?: TweenedOptions<T> | undefined
): Tweened<T>;
```

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,
```

RUNTIME

svelte/transition

Edit this page on GitHub

ON THIS PAGE

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

fade

```
function fade(
  node: Element,
  { delay, duration, easing }?: FadeParams | undefined
): TransitionConfig;
```

```
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>
```

RUNTIME

svelte/animate

Edit this page on GitHub

ON THIS PAGE

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

flip

```
function flip(
  node: Element,
  {
    from,
    to
  }: {
    from: DOMRect;
    to: DOMRect;
  },
  params?: FlipParams
): AnimationConfig;
```

```
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

RUNTIME

svelte/easing

Edit this page on GitHub

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`

PREVIOUS
svelte/animate

NEXT
svelte/action

RUNTIME

svelte/action

Edit this page on GitHub

ON THIS PAGE

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:

App.svelte

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

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

<div use:foo />
```

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

NOTE

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

App.svelte

```
<script>
  /** @type {string} */
  export let bar;

  /** @type {import('svelte/action').Action<HTMLElement, string>}  */
  function foo(node, bar) {
```

COMPILER AND API

svelte/compiler

Edit this page on GitHub

ON THIS PAGE

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

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

compile

```
function compile(
  source: string,
  options?: CompileOptions
): CompileResult;
```

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

```
import { compile } from 'svelte/compiler';

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

Refer to CompileOptions for all the available options.

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

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

Refer to CompileResult for a full description of the compile result.

COMPILER AND API

Client-side component API

Edit this page on GitHub

ON THIS PAGE

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

COMPILER AND API

Server-side component API

Edit this page on GitHub

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

COMPILER AND API

Custom elements API

Edit this page on GitHub

ON THIS PAGE

Svelte components can also be compiled to custom elements (aka web components) using the `customElement: true` compiler option. You should specify a tag name for the component using the `<svelte:options>` element.

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

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

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

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

You can leave out the tag name for any of your inner components which you don't want to expose and use them like regular Svelte components. Consumers of the component can still name it afterwards if needed, using the static `element` property which contains the custom element constructor and which is available when the `customElement` compiler option is `true`.

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

customElements.define('my-element', MyElement.element);
// In Svelte 3, do this instead:
// customElements.define('my-element', MyElement);
```

Once a custom element has been defined, it can be used as a regular DOM element:

```
document.body.innerHTML = `
  <my-element>
    <p>This is some slotted content</p>
  </my-element>
`;
```

MISC

Frequently asked questions

Edit this page on GitHub

ON THIS PAGE

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

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

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

Where can I get support?

If your question is about certain syntax, the API page is a good place to start.

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

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

Are there any third-party resources?

Svelte Society maintains a list of books and videos.

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

There is an official VS Code extension for Svelte.

MISC

Accessibility warnings

Edit this page on GitHub

ON THIS PAGE

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

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

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

You can list multiple rules in a single comment, and add an explanatory note alongside them:

```
<!-- svelte-ignore a11y-click-events-have-key-events a11y-no-static-element-interactions
(because of reasons) -->
<div on:click>...</div>
```

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" />
```

MISC

TypeScript

Edit this page on GitHub

ON THIS PAGE

You can use TypeScript within Svelte components. IDE extensions like the Svelte VSCode extension will help you catch errors right in your editor, and svelte-check does the same on the command line, which you can integrate into your CI.

Setup

To use TypeScript within Svelte components, you need to add a preprocessor that will turn TypeScript into JavaScript.

Using SvelteKit or Vite

The easiest way to get started is scaffolding a new SvelteKit project by typing `npm create svelte@latest`, following the prompts and choosing the TypeScript option.

svelte.config.js

```
import { vitePreprocess } from '@sveltejs/kit/vite';

const config = {
  preprocess: vitePreprocess()
};

export default config;
```

If you don't need or want all the features SvelteKit has to offer, you can scaffold a Svelte-flavoured Vite project instead by typing `npm create vite@latest` and selecting the `svelte-ts` option.

svelte.config.js

```
import { vitePreprocess } from '@sveltejs/vite-plugin-svelte';
const config = {
```

MISC

Svelte 4 migration guide

Edit this page on GitHub

ON THIS PAGE

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

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

Minimum version requirements

- Upgrade to Node 16 or higher. Earlier versions are no longer supported. (#8566)
- If you are using SvelteKit, upgrade to 1.20.4 or newer (sveltejs/kit#10172)
- If you are using Vite without SvelteKit, upgrade to `vite-plugin-svelte` 2.4.1 or newer (#8516)
- If you are using webpack, upgrade to webpack 5 or higher and `svelte-loader` 3.1.8 or higher. Earlier versions are no longer supported. (#8515, 198dbcf)
- If you are using Rollup, upgrade to `rollup-plugin-svelte` 7.1.5 or higher (198dbcf)
- If you are using TypeScript, upgrade to TypeScript 5 or higher. Lower versions might still work, but no guarantees are made about that. (#8488)

Browser conditions for bundlers

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

For Rollup this is done within the `@rollup/plugin-node-resolve` plugin by setting `browser: true` in its options. See the rollup-plugin-svelte documentation for more details

LEGACY

svelte/register

Edit this page on GitHub

NOTE

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

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

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

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

// ...

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

NOTE

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
});
```

PREVIOUS
Svelte 4 migration guide

NEXT

INTRODUCTION

Dynamic attributes

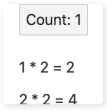
Styling

Nested components

HTML tags

REACTIVITY

Reactive assignments

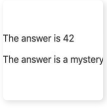


Reactive declarations

Reactive statements

PROPS

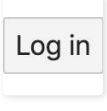
Declaring props



Default values

Spread props

LOGIC



If blocks

Else blocks

App.svelte +

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

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

```

Result JS output CSS output

Hello world!

Console CLEAR

SVELTE | Examples

App.svelte

Result JS output CSS output AST output

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

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

```

Hello world!

Console CLEAR

Start a new project

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

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

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

Alternatives to SvelteKit

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

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

Editor tooling

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

Getting help

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

Start a new project

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

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

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

Alternatives to SvelteKit

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

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

Editor tooling

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

Getting help

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

Is there a router?

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

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

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

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

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

You can see a community-maintained list of routers on sveltesociety.dev.

Can I tell Svelte not to remove my unused styles?

No. Svelte removes the styles from the component and warns you about them in order to prevent issues that would otherwise arise.

Svelte's component style scoping works by generating a class unique to the given component, adding it to the relevant elements in the component that are under Svelte's control, and then adding it to each of the selectors in that component's styles. When the compiler can't see what elements a style selector applies to, there would be two bad options for keeping it:

> If it keeps the selector and adds the scoping class to it, the selector will likely not match the expected elements in the component, and they definitely won't if they were created by a child component or `{@html ...}`.
>
> If it keeps the selector without adding the scoping class to it, the given style will become a global style, affecting your entire page.

If you need to style something that Svelte can't identify at compile time, you will need to explicitly opt into global styles by using `:global(...)`. But also keep in mind that you can wrap `:global(...)` around only part of a selector. `.foo :global(.bar) { ... }` will style any `.bar` elements that appear within the component's `.foo` elements. As long as there's some parent element in the current component to start from, partially global selectors like this will almost always be able to get you what you want.

Is Svelte v2 still available?

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

Start a new project

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

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

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

Alternatives to SvelteKit

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

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

Editor tooling

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

Getting help

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

Start a new project

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

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

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

Alternatives to SvelteKit

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

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

Editor tooling

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

Getting help

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

Discord

You've been invited to join

Svelte

6,202 Online 63,638 Members

DISPLAY NAME

What should everyone call you?

This is how others see you. You can use special characters and emoji.

Continue

By registering, you agree to Discord's Terms of Service and Privacy Policy.

Already have an account?

TEMPLATE SYNTAX

Svelte components

Edit this page on GitHub

ON THIS PAGE

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
```

GETTING STARTED

Introduction

Edit this page on GitHub

ON THIS PAGE

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

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

Start a new project

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

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

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

Alternatives to SvelteKit

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

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