SVELTE • MISC

Svelte 5 migration guide

ON THIS PAGE

Version 5 comes with an overhauled syntax and reactivity system. While it may look different at first, you'll soon notice many similarities. This guide goes over the changes in detail and shows you how to upgrade. Along with it, we also provide information on *why* we did these changes.

You don't have to migrate to the new syntax right away - Svelte 5 still supports the old Svelte 4 syntax, and you can mix and match components using the new syntax with components using the old and vice versa. We expect many people to be able to upgrade with only a few lines of code changed initially. There's also a migration script that helps you with many of these steps automatically.

Reactivity syntax changes

At the heart of Svelte 5 is the new runes API. Runes are basically compiler instructions that inform Svelte about reactivity. Syntactically, runes are functions starting with a dollar-sign.

let -> $state

In Svelte 4, a `let` declaration at the top level of a component was implicitly reactive. In Svelte 5, things are more explicit: a variable is reactive when created using the `$state` rune. Let's migrate the counter to runes mode by wrapping the counter in `$state`:

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

without a wrapper like `.value` or `getCount()`.

Why we did this

$: -> $derived/$effect

In Svelte 4, a `$:` statement at the top level of a component could be used to declare a derivation, i.e. state that is entirely defined through a computation of other state. In Svelte 5, this is achieved using the `$derived` rune:

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

As with `$state`, nothing else changes. `double` is still the number itself, and you read it directly, without a wrapper like `.value` or `getDouble()`.

A `$:` statement could also be used to create side effects. In Svelte 5, this is achieved using the `$effect` rune:

```
<script>
  let count = $state(0);
  $:$effect(() => {
    if (count > 5) {
      alert('Count is too high!');
    }
  });
</script>
```

Why we did this

export let -> $props

```
<script>
  export let optional = 'unset';
  export let required;
  let { optional = 'unset', required } = $props();
</script>
```

There are multiple cases where declaring properties becomes less straightforward than having a few `export let` declarations:

- you want to rename the property, for example because the name is a reserved identifier (e.g. `class`)
- you don't know which other properties to expect in advance
- you want to forward every property to another component

All these cases need special syntax in Svelte 4:

- renaming: `export { klass as class}`
- other properties: `$$restProps`
- all properties `$$props`

In Svelte 5, the `$props` rune makes this straightforward without any additional Svelte-specific syntax:

- renaming: use property renaming `let { class: klass } = $props();`
- other properties: use spreading `let { foo, bar, ...rest } = $props();`
- all properties: don't destructure `let props = $props();`

```
<script>
  let klass = '';
  export { klass as class};
  let { class: klass, ...rest } = $props();
</script>
<button class={klass} {...$$restPropsrest}>click me</button>
```

Docs

Event changes

Event handlers have been given a facelift in Svelte 5. Whereas in Svelte 4 we use the `on:` directive to attach an event listener to an element, in Svelte 5 they are properties like any other (in other words - remove the colon):

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

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

Since they're just properties, you can use the normal shorthand syntax...

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

  function onclick() {
    count++;
  }
</script>

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

...though when using a named event handler function it's usually better to use a more descriptive name.

Component events

In Svelte 4, components could emit events by creating a dispatcher with which means you then pass functions as properties to these components:

App.svelte

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

  let size = $state(15);
  let burst = $state(false);

  function reset() {
    size = 15;
    burst = false;
  }
</script>

<Pump
  on:inflate={(power) => {
    size += power.details;
    if (size > 75) burst = true;
  }}
  on:deflate={(power) => {
    if (size > 0) size -= power.details;
  }}
/>

{#if burst}
  <button onclick={reset}>new balloon</button>
  <span class="boom">💥</span>
{:else}
  <span class="balloon" style="scale: {0.01 * size}">
    🎈
  </span>
{/if}
```

Pump.svelte

```
<script>
  import { createEventDispatcher } from 'svelte';
  const dispatch = createEventDispatcher();

  let { inflate, deflate } = $props();
  inflate
</button>
<button onclick={() => dispatch('deflate', power)deflate(power)}>
  deflate
</button>
<button onclick={() => power--}>-</button>
Pump power: {power}
<button onclick={() => power++}>+</button>
```

Bubbling events

Instead of doing `<button on:click>` to 'forward' the event from the element to the component, the component should accept an `onclick` callback prop:

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

<button on:click {onclick}>
  click me
</button>
```

Note that this also means you can 'spread' event handlers onto the element along with other props instead of tediously forwarding each event separately:

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

<button {...$$props} on:click on:keydown on:all_the_other_stuff {...props}>
  click me
</button>
```

Event modifiers

In Svelte 4, you can add event modifiers to handlers:

Modifiers are specific to `on:` and as such do not work with modern event handlers. Adding things like `event.preventDefault()` inside the handler itself is preferable, since all the logic lives in one place rather than being split between handler and modifiers.

Since event handlers are just functions, you can create your own wrappers as necessary:

```
<script>
  function once(fn) {
    return function (event) {
      if (fn) fn.call(this, event);
      fn = null;
    };
  }

  function preventDefault(fn) {
    return function (event) {
      event.preventDefault();
      fn.call(this, event);
    };
  }
</script>

<button onclick={once(preventDefault(handler))}>...</button>
```

There are three modifiers — `capture`, `passive` and `nonpassive` — that can't be expressed as wrapper functions, since they need to be applied when the event handler is bound rather than when it runs.

For `capture`, we add the modifier to the event name:

```
<button onclickcapture={...}>...</button>
```

Changing the `passive` option of an event handler, meanwhile, is not something to be done lightly. If you have a use case for it — and you probably don't! — then you will need to use an action to apply the event handler yourself.

Docs

In Svelte 4, this is possible:

```
<button on:click={one} on:click={two}>...</button>
```

Duplicate attributes/properties on elements — which now includes event handlers — are not allowed. Instead, do this:

```
<button
  onclick={(e) => {
    one(e);
    two(e);
  }}
>
  ...
</button>
```

When spreading props, local event handlers must go *after* the spread, or they risk being overwritten:

```
<button
  {...props}
  onclick={(e) => {
    doStuff(e);
    props.onclick?.(e);
  }}
>
  ...
</button>
```

Why we did this

Snippets instead of slots

In Svelte 4, content can be passed to components using slots. Svelte 5 replaces them with components.

When using custom elements, you should still use `<slot />` like before. In a future version, when Svelte removes its internal version of slots, it will leave those slots as-is, i.e. output a regular DOM tag instead of transforming it.

Default content

In Svelte 4, the easiest way to pass a piece of UI to the child was using a `<slot />`. In Svelte 5, this is done using the `children` prop instead, which is then shown with `{@render children()}`:

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

<slot />
{@render children?.()}
```

Multiple content placeholders

If you wanted multiple UI placeholders, you had to use named slots. In Svelte 5, use props instead, name them however you like and `{@render ...}` them:

```
<script>
  let { header, main, footer } = $props();
</script>

<header>
  <slot name="header" />
  {@render header()}
</header>

<main>
  <slot name="main" />
  {@render main()}
  {@render footer()}
</footer>
```

Passing data back up

In Svelte 4, you would pass data to a `<slot />` and then retrieve it with `let:` in the parent component. In Svelte 5, snippets take on that responsibility:

App.svelte

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

<List items={['one', 'two', 'three']} let:item>
  {#snippet item(text)}
    <span>{text}</span>
  {/snippet}
  <span slot="empty">No items yet</span>
  {#snippet empty()}
    <span>No items yet</span>
  {/snippet}
</List>
```

List.svelte

```
<script>
  let { items, item, empty } = $props();
</script>

{#if items.length}
  <ul>
    {#each items as entry}
      <li>
        <slot item={entry} />
        {@render item(entry)}
      </li>
    {/each}
  </ul>
{:else}
```

Why we did this

Migration script

By now you should have a pretty good understanding of the before/after and how the old syntax relates to the new syntax. It probably also became clear that a lot of these migrations are rather technical and repetitive - something you don't want to do by hand.

We thought the same, which is why we provide a migration script to do most of the migration automatically. You can upgrade your project by using `npx sv migrate svelte-5`. This will do the following things:

- bump core dependencies in your `package.json`
- migrate to runes (`let -> $state` etc)
- migrate to event attributes for DOM elements (`on:click -> onclick`)
- migrate slot creations to render tags (`<slot /> -> {@render children()}`)
- migrate slot usages to snippets (`<div slot="x">...</div> -> {#snippet x()}<div>...</div>{/snippet}`)
- migrate obvious component creations (`new Component(...) -> mount(Component, ...)`)

You can also migrate a single component in VS Code through the `Migrate Component to Svelte 5 Syntax` command, or in our Playground through the `Migrate` button.

Not everything can be migrated automatically, and some migrations need manual cleanup afterwards. The following sections describe these in more detail.

run

You may see that the migration script converts some of your `$:` statements to a `run` instead. In other cases it may be right, but since `$:` statements also ran on the server but `$effect` does not, it isn't safe to transform it as such. Instead, `run` is used as a stopgap solution. `run` mimics most of the characteristics of `$:`, in that it runs on the server once, and runs as `$effect.pre` on the client (`$effect.pre` runs *before* changes are applied to the DOM; most likely you want to use `$effect` instead).

```
<script>
  import { run } from 'svelte/legacy';
  run(() => {
  $effect(() => {
    // some side effect code
  })
</script>
```

Event modifiers

Event modifiers are not applicable to event attributes (e.g. you can't do `onclick|preventDefault={...}`). Therefore, when migrating event directives to event attributes, we need a function-replacement for these modifiers. These are imported from `svelte/legacy`, and should be migrated away from in favor of e.g. just using `event.preventDefault()`.

```
<script>
  import { preventDefault } from 'svelte/legacy';
</script>

<button
  onclick={preventDefault((event) => {
    event.preventDefault();
    // ...
  })}
>
  click me
</button>
```

parts manually. It doesn't do it because it's too risky because it could result in breakage for users of the component, which the migration script cannot find out.

The migration script does not convert `beforeUpdate/afterUpdate`. It doesn't do it because it's impossible to determine the actual intent of the code. As a rule of thumb you can often go with a combination of `$effect.pre` (runs at the same time as `beforeUpdate` did) and `tick` (imported from `svelte`, allows you to wait until changes are applied to the DOM and then do some work).

Components are no longer classes

In Svelte 3 and 4, components are classes. In Svelte 5 they are functions and should be instantiated differently. If you need to manually instantiate components, you should use `mount` or `hydrate` (imported from `svelte`) instead. If you see this error using SvelteKit, try updating to the latest version of SvelteKit first, which adds support for Svelte 5. If you're using Svelte without SvelteKit, you'll likely have a `main.js` file (or similar) which you need to adjust:

```
import { mount } from 'svelte';
import App from './App.svelte'

const app = new App({ target: document.getElementById("app") });
const app = mount(App, { target: document.getElementById("app") });

export default app;
```

`mount` and `hydrate` have the exact same API. The difference is that `hydrate` will pick up the Svelte's server-rendered HTML inside its target and hydrate it. Both return an object with the exports of the component and potentially property accessors (if compiled with `accessors: true`). They do not come with the `$on`, `$set` and `$destroy` methods you may know from the class component API. These are its replacements:

For `$on`, instead of listening to events, pass them via the `events` property on the options

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

const app = new App({ target: document.getElementById("app") });
app.$on('event', callback);
const app = mount(App, { target: document.getElementById("app"), events: { event: callbac
```

> Note that using `events` is discouraged — instead, use callbacks

For `$set`, use `$state` instead to create a reactive property object and manipulate it. If you're doing this inside a `.js` or `.ts` file, adjust the ending to include `.svelte`, i.e. `.svelte.js` or `.svelte.ts`.

```
import { mount } from 'svelte';
import App from './App.svelte'

const app = new App({ target: document.getElementById("app"), props: { foo: 'bar' } });
app.$set({ foo: 'baz' });
const props = $state({ foo: 'bar' });
const app = mount(App, { target: document.getElementById("app"), props });
props.foo = 'baz';
```

For `$destroy`, use `unmount` instead.

```
import { mount, unmount } from 'svelte';
import App from './App.svelte'

const app = new App({ target: document.getElementById("app"), props: { foo: 'bar' } });
app.$destroy();
const app = mount(App, { target: document.getElementById("app") });
unmount(app);
```

As a stop-gap-solution, you can also use `createClassComponent` or `asClassComponent` (imported from `svelte/legacy`) instead to keep the same API known from Svelte 4 after instantiating.

Docs

```js
const app = createClassComponent({ component: App, target: document.getElementById("app")

export default app;
```

If this component is not under your control, you can use the `compatibility.componentApi` compiler option for auto-applied backwards compatibility, which means code using `new Component(...)` keeps working without adjustments (note that this adds a bit of overhead to each component). This will also add `$set` and `$on` methods for all component instances you get through `bind:this`.

```js
/// svelte.config.js
export default {
  compilerOptions: {
    compatibility: {
      componentApi: 4
    }
  }
};
```

Note that `mount` and `hydrate` are *not* synchronous, so things like `onMount` won't have been called by the time the function returns and the pending block of promises will not have been rendered yet (because `#await` waits a microtask to wait for a potentially immediately-resolved promise). If you need that guarantee, call `flushSync` (import from `'svelte'`) after calling `mount/hydrate`.

Server API changes

Similarly, components no longer have a `render` method when compiled for server side rendering. Instead, pass the function to `render` from `svelte/server`:

```js
import { render } from 'svelte/server';
import App from './App.svelte';

```

Svelte 5, this is no longer the case by default because most of the time you're using a tooling chain that takes care of it in other ways (like SvelteKit). If you need CSS to be returned from `render`, you can set the `css` compiler option to `'injected'` and it will add `<style>` elements to the `head`.

Component typing changes

The change from classes towards functions is also reflected in the typings: `SvelteComponent`, the base class from Svelte 4, is deprecated in favour of the new `Component` type which defines the function shape of a Svelte component. To manually define a component shape in a `d.ts` file:

```ts
import type { Component } from 'svelte';
export declare const MyComponent: Component<{
  foo: string;
}>;
```

To declare that a component of a certain type is required:

```svelte
<script lang="ts">
  import type { ~~SvelteComponent~~ Component } from 'svelte';
  import {
    ComponentA,
    ComponentB
  } from 'component-library';

  ~~let component: typeof SvelteComponent<{ foo: string }>~~
  let component: Component<{ foo: string }> = $state(
    Math.random() ? ComponentA : ComponentB
  );
</script>

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

The two utility types `ComponentEvents` and `ComponentType` are also deprecated.

Docs

`}>`).

bind:this changes

Because components are no longer classes, using `bind:this` no longer returns a class instance with `$set`, `$on` and `$destroy` methods on it. It only returns the instance exports (`export function/const`) and, if you're using the `accessors` option, a getter/setter-pair for each property.

Whitespace handling changed

Previously, Svelte employed a very complicated algorithm to determine if whitespace should be kept or not. Svelte 5 simplifies this which makes it easier to reason about as a developer. The rules are:

- Whitespace between nodes is collapsed to one whitespace
- Whitespace at the start and end of a tag is removed completely
- Certain exceptions apply such as keeping whitespace inside `pre` tags

As before, you can disable whitespace trimming by setting the `preserveWhitespace` option in your compiler settings or on a per-component basis in `<svelte:options>` .

Modern browser required

Svelte 5 requires a modern browser (in other words, not Internet Explorer) for various reasons:

- it uses `Proxies`
- elements with `clientWidth` / `clientHeight` / `offsetWidth` / `offsetHeight` bindings use a `ResizeObserver` rather than a convoluted `<iframe>` hack

Docs

than also listening for `change` events as a fallback

The `legacy` compiler option, which generated bulkier but IE-friendly code, no longer exists.

Changes to compiler options

- The `false` / `true` (already deprecated previously) and the `"none"` values were removed as valid values from the `css` option
- The `legacy` option was repurposed
- The `hydratable` option has been removed. Svelte components are always hydratable now
- The `enableSourcemap` option has been removed. Source maps are always generated now, tooling can choose to ignore it
- The `tag` option was removed. Use `<svelte:options customElement="tag-name" />` inside the component instead
- The `loopGuardTimeout` , `format` , `sveltePath` , `errorMode` and `varsReport` options were removed

The children prop is reserved

Content inside component tags becomes a snippet prop called `children` . You cannot have a separate prop by that name.

Dot notation indicates a component

In Svelte 4, `<foo.bar>` would create an element with a tag name of `"foo.bar"` . In Svelte 5, `foo.bar` is treated as a component instead. This is particularly useful inside `each` blocks:

Docs

```
{/each}
```

Breaking changes in runes mode

Some breaking changes only apply once your component is in runes mode.

Bindings to component exports are not allowed

Exports from runes mode components cannot be bound to directly. For example, having `export const foo = ...` in component `A` and then doing `<A bind:foo />` causes an error. Use `bind:this` instead — `<A bind:this={a} />` — and access the export as `a.foo`. This change makes things easier to reason about, as it enforces a clear separation between props and exports.

Bindings need to be explicitly defined using $bindable()

In Svelte 4 syntax, every property (declared via `export let`) is bindable, meaning you can `bind:` to it. In runes mode, properties are not bindable by default: you need to denote bindable props with the `$bindable` rune.

If a bindable property has a default value (e.g. `let { foo = $bindable('bar') } = $props();`), you need to pass a non-`undefined` value to that property if you're binding to it. This prevents ambiguous behavior — the parent and child must have the same value — and results in better performance (in Svelte 4, the default value was reflected back to the parent, resulting in wasteful additional render cycles).

accessors option is ignored

Setting the `accessors` option to `true` makes properties of a component directly accessible on the component instance. In runes mode, properties are never accessible on the component instance.

Setting the `immutable` option has no effect in runes mode. This concept is replaced by how `$state` and its variations work.

Classes are no longer “auto-reactive”

In Svelte 4, doing the following triggered reactivity:

```
<script>
  let foo = new Foo();
</script>

<button on:click={() => (foo.value = 1)}>{foo.value}</button
>
```

This is because the Svelte compiler treated the assignment to `foo.value` as an instruction to update anything that referenced `foo`. In Svelte 5, reactivity is determined at runtime rather than compile time, so you should define `value` as a reactive `$state` field on the `Foo` class. Wrapping `new Foo()` with `$state(...)` will have no effect — only vanilla objects and arrays are made deeply reactive.

<svelte:component> is no longer necessary

In Svelte 4, components are *static* — if you render `<Thing>`, and the value of `Thing` changes, nothing happens. To make it dynamic you must use `<svelte:component>`.

This is no longer true in Svelte 5:

```
<script>
  import A from './A.svelte';
  import B from './B.svelte';

  let Thing = $state();
</script>

<select bind:value={Thing}>
```

```
<!-- these are equivalent -->
<Thing />
<svelte:component this={Thing} />
```

Touch and wheel events are passive

When using `onwheel`, `onmousewheel`, `ontouchstart` and `ontouchmove` event attributes, the handlers are passive to align with browser defaults. This greatly improves responsiveness by allowing the browser to scroll the document immediately, rather than waiting to see if the event handler calls `event.preventDefault()`.

In the very rare cases that you need to prevent these event defaults, you should use `on` instead (for example inside an action).

Attribute/prop syntax is stricter

In Svelte 4, complex attribute values needn't be quoted:

```
<Component prop=this{is}valid />
```

This is a footgun. In runes mode, if you want to concatenate stuff you must wrap the value in quotes:

```
<Component prop="this{is}valid" />
```

Note that Svelte 5 will also warn if you have a single expression wrapped in quotes, like `answer="{42}"` — in Svelte 6, that will cause the value to be converted to a string, rather than passed as a number.

HTML structure is stricter

In Svelte 4, you were allowed to write HTML code that would be repaired by the browser

```
  <tr>
    <td>hi</td>
  </tr>
</table>
```

… and the browser would auto-insert a `<tbody>` element:

```
<table>
  <tbody>
    <tr>
      <td>hi</td>
    </tr>
  </tbody>
</table>
```

Svelte 5 is more strict about the HTML structure and will throw a compiler error in cases where the browser would repair the DOM.

Other breaking changes

Stricter @const assignment validation

Assignments to destructured parts of a `@const` declaration are no longer allowed. It was an oversight that this was ever allowed.

:is(...) and :where(...) are scoped

Previously, Svelte did not analyse selectors inside `:is(...)` and `:where(...)`, effectively treating them as global. Svelte 5 analyses them in the context of the current component. As such, some selectors may now be treated as unused if they were relying on this treatment. To fix this, use `:global(...)` inside the `:is(...)/:where(...)` selectors.

When using Tailwind's `@apply` directive, add a `:global` selector to preserve rules that use

```
  @apply bg-blue-100 dark:bg-blue-900;
}
```

CSS hash position no longer deterministic

Previously Svelte would always insert the CSS hash last. This is no longer guaranteed in Svelte 5. This is only breaking if you have very weird css selectors.

Scoped CSS uses :where(...)

To avoid issues caused by unpredictable specificity changes, scoped CSS selectors now use `:where(.svelte-xyz123)` selector modifiers alongside `.svelte-xyz123` (where `xyz123` is, as previously, a hash of the `<style>` contents). You can read more detail here.

In the event that you need to support ancient browsers that don't implement `:where`, you can manually alter the emitted CSS, at the cost of unpredictable specificity changes:

```
css = css.replace(/:where\((.+?)\)/, '$1');
```

Error/warning codes have been renamed

Error and warning codes have been renamed. Previously they used dashes to separate the words, they now use underscores (e.g. foo-bar becomes foo_bar). Additionally, a handful of codes have been reworded slightly.

Reduced number of namespaces

The number of valid namespaces you can pass to the compiler option `namespace` has been reduced to `html` (the default), `mathml` and `svg`.

The `foreign` namespace was only useful for Svelte Native, which we're planning to support differently in a 5.x minor.

Docs

`beforeUpdate` no longer runs twice on initial render if it modifies a variable referenced in the template.

`afterUpdate` callbacks in a parent component will now run after `afterUpdate` callbacks in any child components.

Both functions are disallowed in runes mode — use `$effect.pre(...)` and `$effect(...)` instead.

contenteditable behavior change

If you have a `contenteditable` node with a corresponding binding *and* a reactive value inside it (example: `<div contenteditable=true bind:textContent>count is {count}</div>`), then the value inside the contenteditable will not be updated by updates to `count` because the binding takes full control over the content immediately and it should only be updated through it.

oneventname attributes no longer accept string values

In Svelte 4, it was possible to specify event attributes on HTML elements as a string:

```
<button onclick="alert('hello')">...</button>
```

This is not recommended, and is no longer possible in Svelte 5, where properties like `onclick` replace `on:click` as the mechanism for adding event handlers.

null and undefined become the empty string

In Svelte 4, `null` and `undefined` were printed as the corresponding string. In 99 out of 100 cases you want this to become the empty string instead, which is also what most other frameworks out there do. Therefore, in Svelte 5, `null` and `undefined` become the empty string.

`bind:files` is now a two-way binding. As such, when setting a value, it needs to be either falsy (`null` or `undefined`) or of type `FileList` .

Bindings now react to form resets

Previously, bindings did not take into account `reset` event of forms, and therefore values could get out of sync with the DOM. Svelte 5 fixes this by placing a `reset` listener on the document and invoking bindings where necessary.

walk no longer exported

`svelte/compiler` reexported `walk` from `estree-walker` for convenience. This is no longer true in Svelte 5, import it directly from that package instead in case you need it.

Content inside svelte:options is forbidden

In Svelte 4 you could have content inside a `<svelte:options />` tag. It was ignored, but you could write something in there. In Svelte 5, content inside that tag is a compiler error.

<slot> elements in declarative shadow roots are preserved

Svelte 4 replaced the `<slot />` tag in all places with its own version of slots. Svelte 5 preserves them in the case they are a child of a `<template shadowrootmode="...">` element.

<svelte:element> tag must be an expression

In Svelte 4, `<svelte:element this="div">` is valid code. This makes little sense — you should just do `<div>` . In the vanishingly rare case that you *do* need to use a literal value for some reason, you can do this:

```
<svelte:element this={"div"}>
```

Docs

identically to `<input>` for the purposes of determining which `bind:` directives could be applied, Svelte 5 does not.

mount plays transitions by default

The `mount` function used to render a component tree plays transitions by default unless the `intro` option is set to `false`. This is different from legacy class components which, when manually instantiated, didn't play transitions by default.

<img src={...}> and {@html ...} hydration mismatches are not repaired

In Svelte 4, if the value of a `src` attribute or `{@html ...}` tag differ between server and client (a.k.a. a hydration mismatch), the mismatch is repaired. This is very costly: setting a `src` attribute (even if it evaluates to the same thing) causes images and iframes to be reloaded, and reinserting a large blob of HTML is slow.

Since these mismatches are extremely rare, Svelte 5 assumes that the values are unchanged, but in development will warn you if they are not. To force an update you can do something like this:

```
<script>
  let { markup, src } = $props();

  if (typeof window !== 'undefined') {
    // stash the values...
    const initial = { markup, src };

    // unset them...
    markup = src = undefined;

    $effect(() => {
      // ...and reset after we've mounted
      markup = initial.markup;
      src = initial.src;
    });
```

```
<img {src} />
```

Hydration works differently

Svelte 5 makes use of comments during server side rendering which are used for more robust and efficient hydration on the client. As such, you shouldn't remove comments from your HTML output if you intend to hydrate it, and if you manually authored HTML to be hydrated by a Svelte component, you need to adjust that HTML to include said comments at the correct positions.

onevent attributes are delegated

Event attributes replace event directives: Instead of `on:click={handler}` you write `onclick={handler}`. For backwards compatibility the `on:event` syntax is still supported and behaves the same as in Svelte 4. Some of the `onevent` attributes however are delegated, which means you need to take care to not stop event propagation on those manually, as they then might never reach the listener for this event type at the root.

--style-props uses a different element

Svelte 5 uses an extra `<svelte-css-wrapper>` element instead of a `<div>` to wrap the component when using CSS custom properties.

Edit this page on GitHub

PREVIOUS
Svelte 4 migration guide

NEXT
Frequently asked questions