SVELTE • TEMPLATE SYNTAX

bind:

ON THIS PAGE

Data ordinarily flows down, from parent to child. The `bind:` directive allows data to flow the other way, from child to parent.

The general syntax is `bind:property={expression}`, where `expression` is an *lvalue* (i.e. a variable or an object property). When the expression is an identifier with the same name as the property, we can omit the expression — in other words these are equivalent:

```
<input bind:value={value} />
<input bind:value />
```

Svelte creates an event listener that updates the bound value. If an element already has a listener for the same event, that listener will be fired before the bound value is updated.

Most bindings are *two-way*, meaning that changes to the value will affect the element and vice versa. A few bindings are *readonly*, meaning that changing their value will have no effect on the element.

<input bind:value>

A `bind:value` directive on an `<input>` element binds the input's `value` property:

```
<script>
  let message = $state('hello');
</script>
```

to a number (demo):

```
<script>
  let a = $state(1);
  let b = $state(2);
</script>

<label>
  <input type="number" bind:value={a} min="0" max="10" />
  <input type="range" bind:value={a} min="0" max="10" />
</label>

<label>
  <input type="number" bind:value={b} min="0" max="10" />
  <input type="range" bind:value={b} min="0" max="10" />
</label>

<p>{a} + {b} = {a + b}</p>
```

If the input is empty or invalid (in the case of `type="number"`), the value is `undefined`.

<input bind:checked>

Checkbox and radio inputs can be bound with `bind:checked`:

```
<label>
  <input type="checkbox" bind:checked={accepted} />
  Accept terms and conditions
</label>
```

<input bind:group>

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

```
  let fillings = [];
</script>

<!-- grouped radio inputs are mutually exclusive -->
<input type="radio" bind:group={tortilla} value="Plain" />
<input type="radio" bind:group={tortilla} value="Whole wheat" />
<input type="radio" bind:group={tortilla} value="Spinach" />

<!-- grouped checkbox inputs populate an array -->
<input type="checkbox" bind:group={fillings} value="Rice" />
<input type="checkbox" bind:group={fillings} value="Beans" />
<input type="checkbox" bind:group={fillings} value="Cheese" />
<input type="checkbox" bind:group={fillings} value="Guac (extra)" />
```

`bind:group` only works if the inputs are in the same Svelte component.

<input bind:files>

On `<input>` elements with `type="file"`, you can use `bind:files` to get the `FileList` of selected files. When you want to update the files programmatically, you always need to use a `FileList` object. Currently `FileList` objects cannot be constructed directly, so you need to create a new `DataTransfer` object and get `files` from there.

```
<script>
  let files = $state();

  function clear() {
    files = new DataTransfer().files; // null or undefined does not work
  }
</script>

<label for="avatar">Upload a picture:</label>
<input accept="image/png, image/jpeg" bind:files id="avatar" name="avatar" type="file" />
<button onclick={clear}>clear</button>
```

`FileList` objects also cannot be modified, so if you want to e.g. delete a single file from the

to `files` uninitialized prevents potential errors if components are server-side rendered.

<select bind:value>

A `<select>` value binding corresponds to the `value` property on the selected `<option>`, which can be any value (not just strings, as is normally the case in the DOM).

```
<select bind:value={selected}>
  <option value={a}>a</option>
  <option value={b}>b</option>
  <option value={c}>c</option>
</select>
```

A `<select multiple>` element behaves similarly to a checkbox group. The bound variable is an array with an entry corresponding to the `value` property of each selected `<option>`.

```
<select multiple bind:value={fillings}>
  <option value="Rice">Rice</option>
  <option value="Beans">Beans</option>
  <option value="Cheese">Cheese</option>
  <option value="Guac (extra)">Guac (extra)</option>
</select>
```

When the value of an `<option>` matches its text content, the attribute can be omitted.

```
<select multiple bind:value={fillings}>
  <option>Rice</option>
  <option>Beans</option>
  <option>Cheese</option>
  <option>Guac (extra)</option>
</select>
```

<audio>

- `currentTime`
- `playbackRate`
- `paused`
- `volume`
- `muted`

...and seven readonly ones:

- `duration`
- `buffered`
- `paused`
- `seekable`
- `seeking`
- `ended`
- `readyState`

```
<audio src={clip} bind:duration bind:currentTime bind:paused></audio>
```

<video>

`<video>` elements have all the same bindings as [#audio] elements, plus readonly `videoWidth` and `videoHeight` bindings.

`<img>` elements have two readonly bindings:

<details bind:open>

`<details>` elements support binding to the `open` property.

```
<details bind:open={isOpen}>
  <summary>How do you comfort a JavaScript bug?</summary>
  <p>You console it.</p>
</details>
```

Contenteditable bindings

Elements with the `contenteditable` attribute support the following bindings:

- `innerHTML`
- `innerText`
- `textContent`

There are subtle differences between `innerText` and `textContent`.

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

Dimensions

All visible elements have the following readonly bindings, measured with a `ResizeObserver`:

- `clientWidth`
- `clientHeight`

```
<div bind:offsetWidth={width} bind:offsetHeight={height}>
  <Chart {width} {height} />
</div>
```

> `display: inline` elements do not have a width or height (except for elements with 'intrinsic' dimensions, like `<img>` and `<canvas>`), and cannot be observed with a `ResizeObserver`. You will need to change the `display` style of these elements to something else, such as `inline-block`.

bind:this

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

To get a reference to a DOM node, use `bind:this`. The value will be `undefined` until the component is mounted — in other words, you should read it inside an effect or an event handler, but not during component initialisation:

```
<script>
  /** @type {HTMLCanvasElement} */
  let canvas;

  $effect(() => {
    const ctx = canvas.getContext('2d');
    drawStuff(ctx);
  });
</script>

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

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

```
<button onclick={() => cart.empty()}> Empty shopping cart </button>
```

ShoppingCart.svelte

```
<script>
  // All instance exports are available on the instance object
  export function empty() {
    // ...
  }
</script>
```

bind:*property* for components

```
bind:property={variable}
```

You can bind to component props using the same syntax as for elements.

```
<Keypad bind:value={pin} />
```

While Svelte props are reactive without binding, that reactivity only flows downward into the component by default. Using `bind:property` allows changes to the property from within the component to flow back up out of the component.

To mark a property as bindable, use the `$bindable` rune:

```
<script>
  let { readonlyProperty, bindableProperty = $bindable() } = $props();
</script>
```

Declaring a property as bindable means it *can* be used using `bind:`, not that it *must* be used using `bind:`.

```
  let { bindableProperty = $bindable('fallback value') } = $props();
</script>
```

This fallback value *only* applies when the property is *not* bound. When the property is bound and a fallback value is present, the parent is expected to provide a value other than `undefined`, else a runtime error is thrown. This prevents hard-to-reason-about situations where it's unclear which value should apply.

Edit this page on GitHub

PREVIOUS
{@debug ...}

NEXT
use: