SVELTE • MISC

Custom elements

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" />

<script>
  let { name = 'world' } = $props();
</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);
```

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

```
document.body.innerHTML = `
```

Any props are exposed as properties of the DOM element (as well as being readable/writable as attributes, where possible).

```
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';
```

Note that you need to list out all properties explicitly, i.e. doing `let props = $props()` without declaring `props` in the component options means that Svelte can't know which props to expose as properties on the DOM element.

Component lifecycle

Custom elements are created from Svelte components using a wrapper approach. This means the inner Svelte component has no knowledge that it is a custom element. The custom element wrapper takes care of handling its lifecycle appropriately.

When a custom element is created, the Svelte component it wraps is *not* created right away. It is only created in the next tick after the `connectedCallback` is invoked. Properties assigned to the custom element before it is inserted into the DOM are temporarily saved and then set on component creation, so their values are not lost. The same does not work for invoking exported functions on the custom element though, they are only available after the element has mounted. If you need to invoke functions before component creation, you can work around it by using the `extend` option.

When a custom element written with Svelte is created or updated, the shadow DOM will reflect the value in the next tick, not immediately. This way updates can be batched, and

invoked.

Component options

When constructing a custom element, you can tailor several aspects by defining `customElement` as an object within `<svelte:options>` since Svelte 4. This object may contain the following properties:

`tag: string` : an optional `tag` property for the custom element's name. If set, a custom element with this tag name will be defined with the document's `customElements` registry upon importing this component.

`shadow` : an optional property that can be set to `"none"` to forgo shadow root creation. Note that styles are then no longer encapsulated, and you can't use slots

`props` : an optional property to modify certain details and behaviors of your component's properties. It offers the following settings:

`attribute: string` : To update a custom element's prop, you have two alternatives: either set the property on the custom element's reference as illustrated above or use an HTML attribute. For the latter, the default attribute name is the lowercase property name. Modify this by assigning `attribute: "<desired name>"` .

`reflect: boolean` : By default, updated prop values do not reflect back to the DOM. To enable this behavior, set `reflect: true` .

`type: 'String' | 'Boolean' | 'Number' | 'Array' | 'Object'` : While converting an attribute value to a prop value and reflecting it back, the prop value is assumed to be a `String` by default. This may not always be accurate. For instance, for a number type, define it using `type: "Number"` You don't need to list all properties, those not listed will use the default settings.

`extend` : an optional property which expects a function as its argument. It is passed the custom element class generated by Svelte and expects you to return a custom element class. This comes in handy if you have very specific requirements to the life cycle of the

better HTML form integration.

```
<svelte:options
  customElement={{
    tag: 'custom-element',
    shadow: 'none',
    props: {
      name: { reflect: true, type: 'Number', attribute: 'element-index' }
    },
    extend: (customElementConstructor) => {
      // Extend the class so we can let it participate in HTML forms
      return class extends customElementConstructor {
        static formAssociated = true;

        constructor() {
          super();
          this.attachedInternals = this.attachInternals();
        }

        // Add the function here, not below in the component so that
        // it's always available, not just when the inner Svelte component
        // is mounted
        randomIndex() {
          this.elementIndex = Math.random();
        }
      };
    }
  }}
/>

<script>
  let { elementIndex, attachedInternals } = $props();
  // ...
  function check() {
    attachedInternals.checkValidity();
  }
</script>

...
```

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* (unless you set `shadow: "none"`). 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 deprecated `let:` directive has no effect, because custom elements do not have a way to pass data to the parent component that fills the slot
- Polyfills are required to support older browsers
- You can use Svelte's context feature between regular Svelte components within a custom element, but you can't use them across custom elements. In other words, you can't use `setContext` on a parent custom element and read that with `getContext` in a child custom element.

Edit this page on GitHub

PREVIOUS
TypeScript

NEXT
Svelte 4 migration guide

Docs