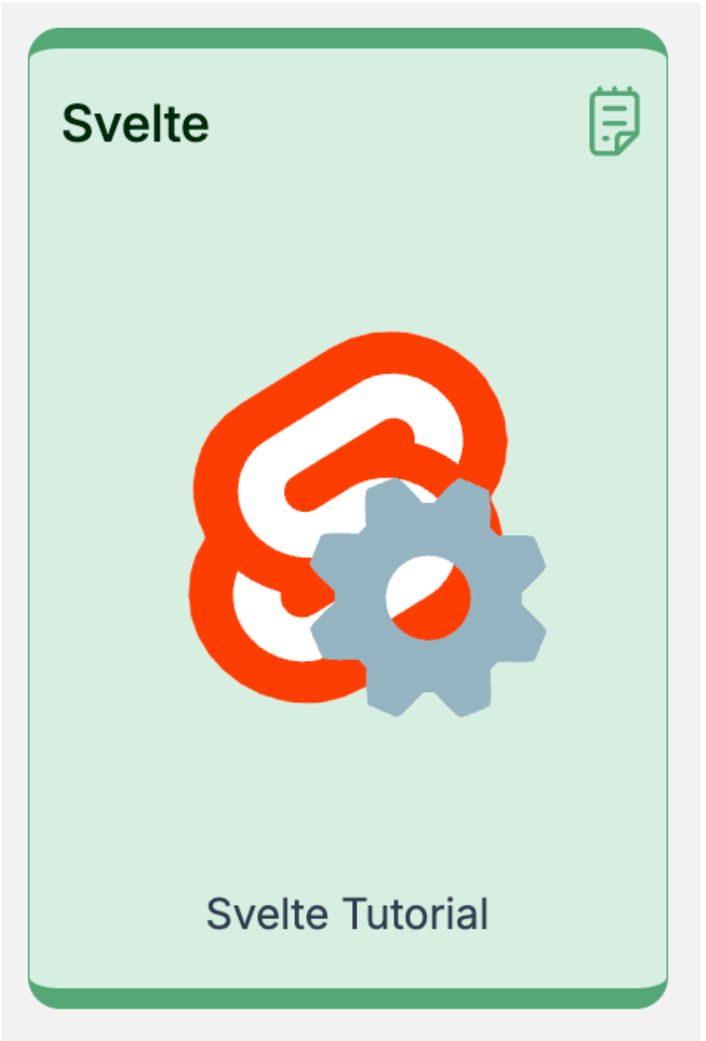
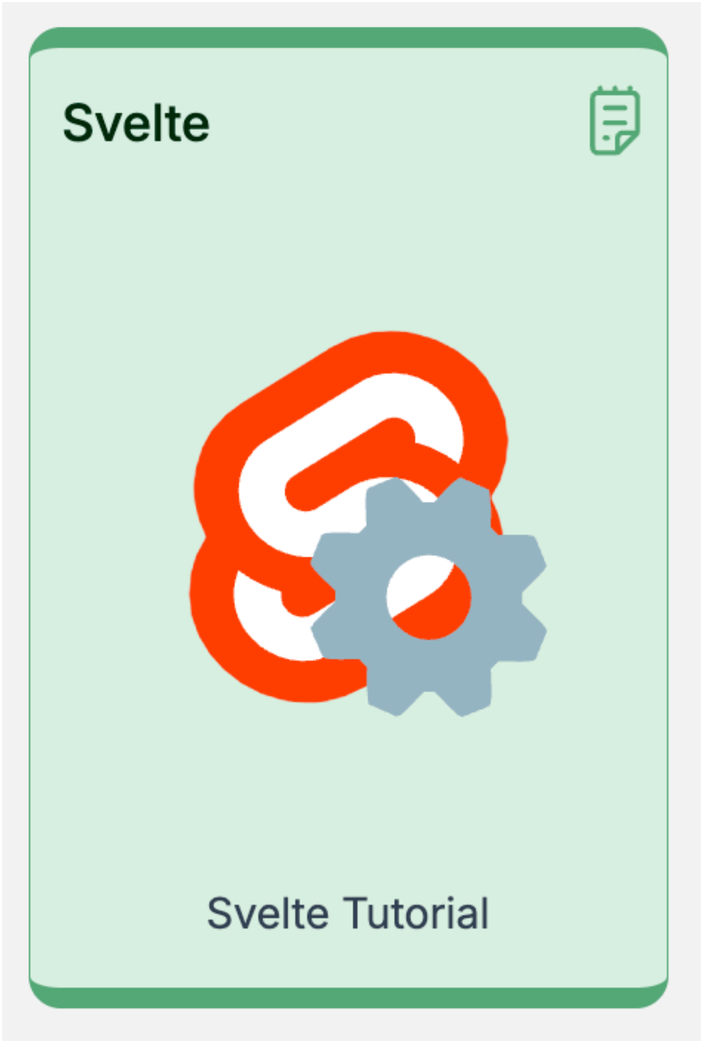
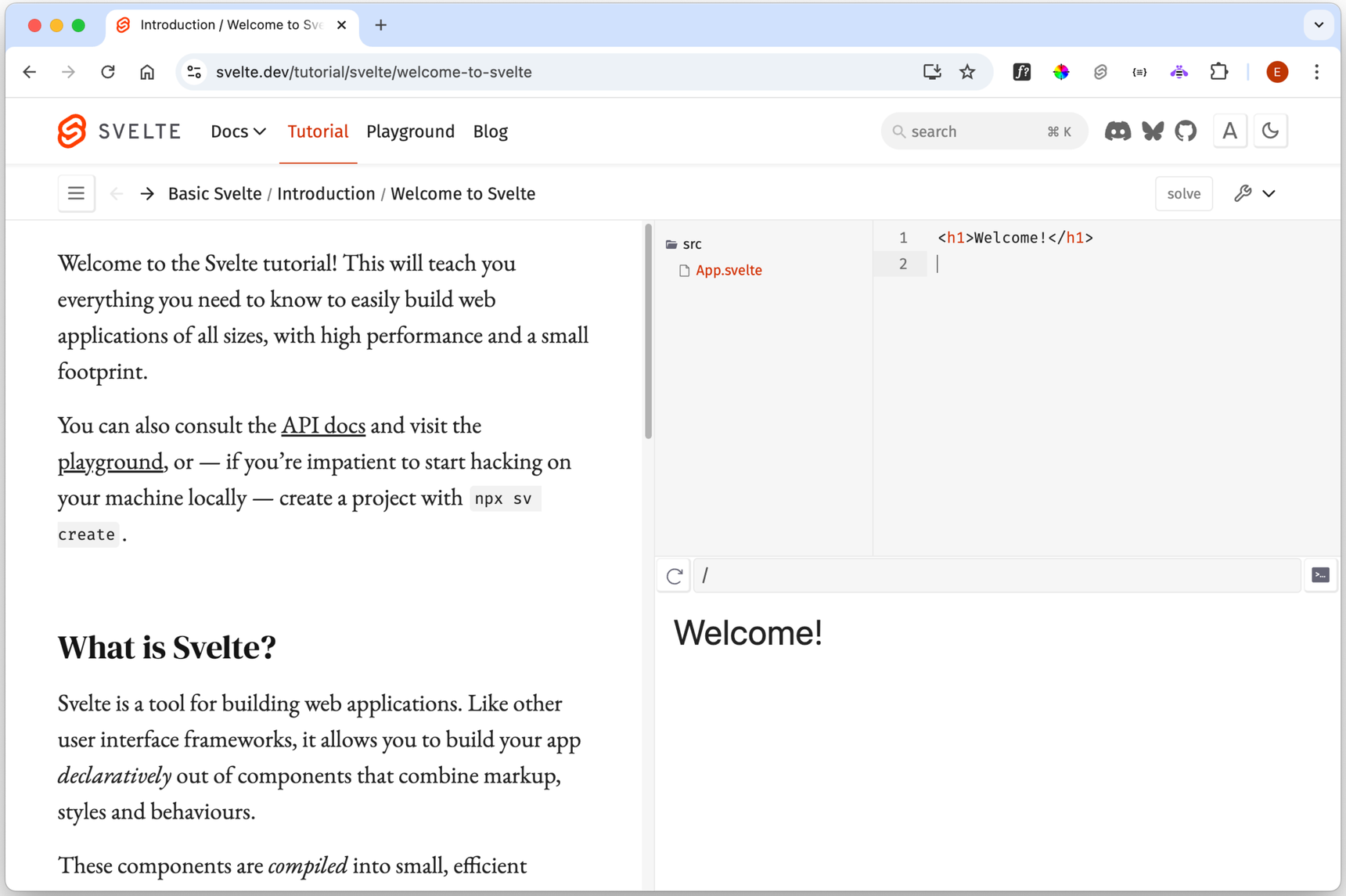
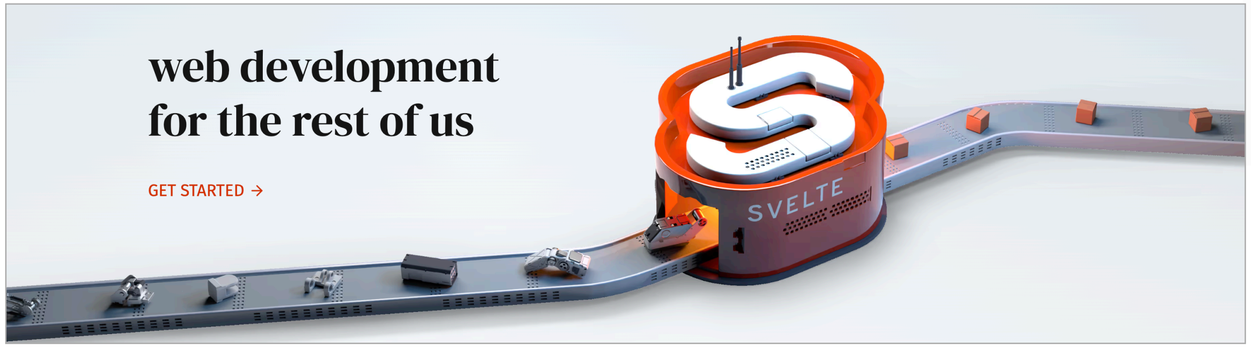

Svelte
Svelte Tutorial

https://svelte.dev/tutorial/svelte/welcome-to-svelte





- Basic Svelte
  - Introduction
  - Reactivity
  - Props
  - Logic
  - Events
  - Bindings
  - Classes and styles
  - Actions
  - Transitions
- Advanced Svelte
- Basic SvelteKit
- Advanced SvelteKit



- Introduction
  - Welcome to Svelte
  - Your first component
  - Dynamic attributes
  - Styling
  - Nested components
  - HTML tags
- Reactivity
  - State
  - Deep state
  - Derived state
  - Inspecting state
  - Effects
  - Universal reactivity
- Props
  - Declaring props
  - Default values
  - Spread props

- Logic
  - If blocks
  - Else blocks
  - Else-if blocks
  - Each blocks
  - Keyed each blocks
  - Await blocks
- Events
  - DOM events
  - Inline handlers
  - Capturing
  - Component events
  - Spreading events
- Bindings
  - Text inputs
  - Numeric inputs
  - Checkbox inputs
  - Select bindings
  - Group inputs
  - Select multiple
  - Textarea inputs

Key Svelte Concepts

- Components
- Reactivity
- Props
- Logic
- Events
- Bindings
- Classes & Styles
- Actions
- Transitions

- **Components**
- Reactivity
- Props
- Logic
- Events
- Bindings
- Classes & Styles
- Actions
- Transitions

Components

Components

- In Svelte, an application is composed from one or more components.
- A component is a reusable self-contained block of code that encapsulates HTML, CSS and JavaScript that belong together, written into a .svelte file

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

<h1>Hello {name.toUpperCase()}!</h1>
```

App.svelte

App.svelte

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

<h1>Hello world!</h1>
```

We can refer to name in the markup:

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

Inside the curly braces, we can put any JavaScript we want

```
<h1>Hello {name.toUpperCase()}!</h1>
```

Components : Dynamic Attributes

- Just like you can use curly braces to control text, you can use them to control element attributes.

- When building web apps, it's important to make sure that they're *accessible* to the broadest possible user base, including people with (for example) impaired vision or motion,

- Svelte will monitor Accessibility (shortened to a11y) and will help by warning you if you write inaccessible markup.

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

But if you hover over the **<img>** in the editor, Svelte is giving us a warning:

```
`<img>` element should have an alt attribute
```

We can use curly braces *inside* attributes.

App.svelte

```
<script>
  let src = '/tutorial/image.gif';
  let name = 'Rick Astley';
</script>

<img {src} alt="{name} dances." />
```

Components : Styling

- Just like in HTML, you can add a **<style>** tag to your component
- Importantly, these rules are *scoped to the component*.
- You won't accidentally change the style of **<p>** elements elsewhere in your app, as we'll see in the next step.

```
<p>This is a paragraph.</p>

<style>
  p {
    color: goldenrod;
    font-family: 'Comic Sans MS', cursive;
    font-size: 2em;
  }
</style>
```

App.svelte

Components : Nesting

- It would be impractical to put your entire app in a single component.
- Instead, we can import components from other files and include them in our markup
- Even though **Nested.svelte** has a **<p>** element, the styles from **App.svelte** don't leak in.

Nested.svelte

```
<p>This is another paragraph.</p>
```

App.svelte

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

<p>This is a paragraph.</p>
<Nested />

<style>
  p {
    color: goldenrod;
    font-family: 'Comic Sans MS', cursive;
    font-size: 2em;
  }
</style>
```

- Components
- **Reactivity**
- Props
- Logic
- Events
- Bindings
- Classes & Styles
- Actions
- Transitions

Reactivity

Reactivity

- At the heart of Svelte is a powerful system of *reactivity* for keeping the DOM in sync with your application state — for example, in response to an event.

- The count declaration is made reactive by wrapping the value with **$state(...)**

- This is called a rune, and it's how you tell Svelte that count isn't an ordinary variable.

App.svelte

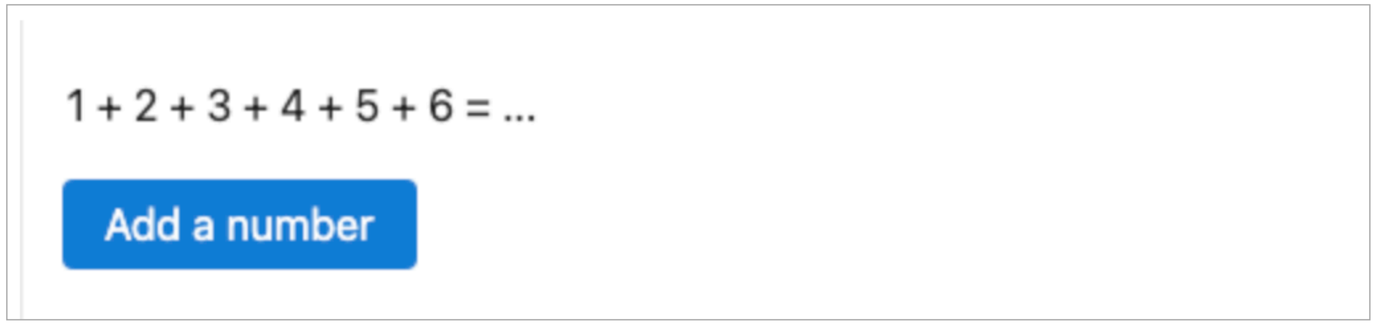
```
<script>
  let count = $state(0);

  function increment() {
    count += 1;
  }
</script>

<button onclick={increment}>
  Clicked {count}
  {count === 1 ? 'time' : 'times'}
</button>
```

Reactivity: Deep State

- State reacts to *reassignments*.
- But it also reacts to *mutations* — this is called *deep reactivity*.
- Now, when we change the array…
- …the component updates.

App.svelte

```svelte
<script>
  let numbers = $state([1, 2, 3, 4]);

  function addNumber() {
    numbers.push(numbers.length + 1);
  }
</script>

<p>{numbers.join(' + ')} = ...</p>

<button onclick={addNumber}>
  Add a number
</button>
```



Reactivity: Derived State

- Often, you will need to *derive* state from other state. For this, we have the **$derived** rune:
- The expression inside the **$derived** declaration will be re-evaluated whenever its dependencies (in this case, just **numbers**) are updated.
- Unlike normal state, derived state is read-only.

App.svelte

```
<script>
  let numbers = $state([1, 2, 3, 4]);
  let total = $derived(numbers.reduce((t, n) => t + n, 0));

  function addNumber() {
    numbers.push(numbers.length + 1);
  }
</script>

<p>{numbers.join(' + ')} = {total}</p>

<button onclick={addNumber}>
  Add a number
</button>
```

Reactivity: Inspecting State

- It's often useful to be able to track the value of a piece of state as it changes over time.
- the $inspect rune to automatically log a snapshot of the state whenever it changes.
- This code will automatically be stripped out of your production build:

App.svelte

```
<script>
  let numbers = $state([1, 2, 3, 4]);
  let total = $derived(numbers.reduce((t, n) => t + n, 0));

  function addNumber() {
    numbers.push(numbers.length + 1);
  }

  $inspect(numbers).with(console.trace);
</script>

<p>{numbers.join(' + ')} = {total}</p>

<button onclick={addNumber}>
  Add a number
</button>
```

Developer Console

Reactivity: Universal Reactivity

- We can also use runes outside components, for example to share some global state.
- The <Counter> components here are all importing the counter object from **shared.svelte.js.**
- When you click any button, all three update simultaneously
- For this to work, the counter object must have been called ".svelte.js"

shared.svelte.ts

```
export const counter = $state({
  count: 0
});
```

Counter.svelte

```
<script>
  import { counter } from './shared.svelte.js';
</script>

<button onclick={() => counter.count += 1}>
  clicks: {counter.count}
</button>
```

App.svelte

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

<Counter />
<Counter />
<Counter />
```

- Components
- Reactivity
- **Props**
- Logic
- Events
- Bindings
- Classes & Styles
- Actions
- Transitions

Props

Props

- In any real application, you'll need to pass data from one component down to its children.
- To do that, we need to declare *properties*, generally shortened to 'props'.
- In Svelte, we do that with the **$props** rune"

Nested.svelte

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

<p>The answer is {answer}</p>
```

App.svelte

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

<Nested answer={42} />
```

The answer is 42

Props: Default Values

- We can specify default values for props in **Nested.svelte**
- If we now add a second component *without* an **answer** prop, it will fall back to the default

Nested.svelte

```
<script>
  let { answer = 'a mystery' } = $props();
</script>

<p>The answer is {answer}</p>
```

App.svelte

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

<Nested answer={42} />
<Nested />
```

The answer is 42

The answer is a mystery

Props: Spread Props

- PackgteInfo props can be passed in the normal way:

```
<PackageInfo
  name={pkg.name}
  version={pkg.version}
  description={pkg.description}
  website={pkg.website}
/>
```

- … or they can be *spread*

Packageinfo.svelte

```
<script>
  let { name, version, description, website } = $props();
</script>

<p>
  The <code>{name}</code> package is {description}. Download version {version} from
  <a href="https://www.npmjs.com/package/{name}">npm</a> and <a href={website}>learn more here</a>
</p>
```

App.svelte

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

  const pkg = {
    name: 'svelte',
    version: 5,
    description: 'blazing fast',
    website: 'https://svelte.dev'
  };
</script>

<PackageInfo {...pkg} />
```

The `svelte` package is blazing fast. Download version 5 from npm and learn more here

- Components
- Reactivity
- Props
- **Logic**
- Events
- Bindings
- Classes & Styles
- Actions
- Transitions

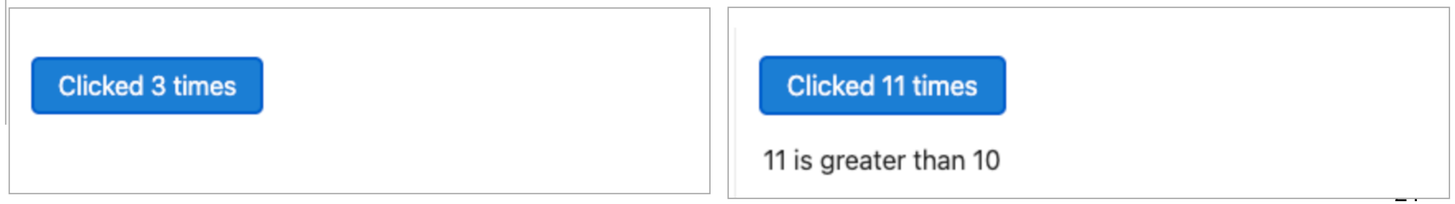
Logic

Logic: If Blocks

- HTML doesn't have a way of expressing *logic*, like conditionals and loops. Svelte does.
- To conditionally render some markup, we wrap it in an if block.
- The text that appears when count is greater than 10:

App.svelte

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

  function increment() {
    count += 1;
  }
</script>

<button onclick={increment}>
  Clicked {count}
  {count === 1 ? 'time' : 'times'}
</button>

{#if count > 10}
  <p>{count} is greater than 10</p>
{/if}
```

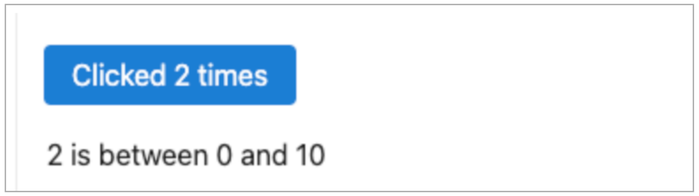


Logic: Else Blocks

- Just like in JavaScript, an **if** block can have an **else** block
- {#...} opens a block. {/...} closes a block. {:...} continues a block..

App.svelte

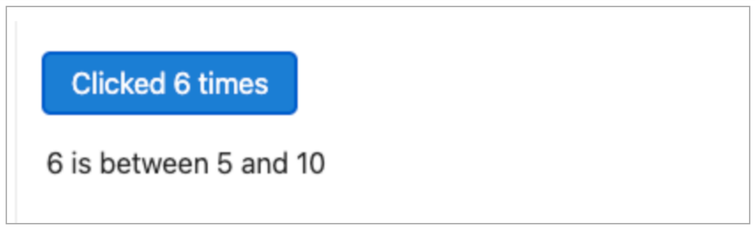
```
<script>
  let count = $state(0);

  function increment() {
    count += 1;
  }
</script>

<button onclick={increment}>
  Clicked {count}
  {count === 1 ? 'time' : 'times'}
</button>

{#if count > 10}
  <p>{count} is greater than 10</p>
{:else}
  <p>{count} is between 0 and 10</p>
{/if}
```



Logic: Else-if Blocks

- Multiple conditions can be ‘chained’ together with **else if**:.

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

  function increment() {
    count += 1;
  }
</script>

<button onclick={increment}>
  Clicked {count}
  {count === 1 ? 'time' : 'times'}
</button>

{#if count > 10}
  <p>{count} is greater than 10</p>
{:else if count < 5}
  <p>{count} is less than 5</p>
{:else}
  <p>{count} is between 5 and 10</p>
{/if}
```



Logic: Each Blocks

- When building user interfaces you'll often find yourself working with lists of data.
- Here, we repeated the <button> markup multiple times — changing the colour each time — but there's still more to add.
- Instead of laboriously copying, pasting and editing, we can get rid of all but the first button, then use an each block

App.svelte

```
<script>
  const colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];
  let selected = $state(colors[0]);
</script>

<h1 style="color: {selected}">Pick a colour</h1>

<div>
  {#each colors as color, i}
    <button
      style="background: {color}"
      aria-label={color}
      aria-current={selected === color}
      onclick={() => selected = color}
    >{i + 1}</button>
  {/each}
</div>
```

- Components
- Reactivity
- Props
- Logic
- **Events**
- Bindings
- Classes & Styles
- Actions
- Transitions

Events

Events: DOM Events

- You can listen to any DOM event on an element (such as click or pointermove) with an on<name> function

App.svelte

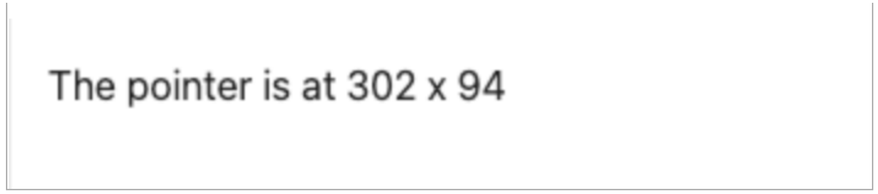
```svelte
<script>
  let m = $state({ x: 0, y: 0 });

  function onpointermove(event) {
    m.x = event.clientX;
    m.y = event.clientY;
  }
</script>

<div {onpointermove}>
  The pointer is at {Math.round(m.x)} x {Math.round(m.y)}
</div>

<style>
  div {
    position: fixed;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    padding: 1rem;
  }
</style>
```

Events: Inline Handlers

- You can also declare event handlers inline

App.svelte

```svelte
<script>
  let m = $state({ x: 0, y: 0 });
</script>

<div
  onpointermove={(event) => {
    m.x = event.clientX;
    m.y = event.clientY;
  }}
>
  The pointer is at {Math.round(m.x)} x {Math.round(m.y)}
</div>

<style>
  div {
    position: fixed;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    padding: 1rem;
  }
</style>
```



Events: Component Events

- You can pass event handlers to components like any other prop.
- In **Stepper.svelte** we have **increment** and **decrement** props…and wire them up.
- In **App.svelte**, define the handlers

Stepper.svelte

```svelte
<script>
  let { increment, decrement } = $props();
</script>

<button onclick={decrement}>-1</button>
<button onclick={increment}>+1</button>
```

App.svelte

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

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

<p>The current value is {value}</p>

<Stepper
  increment={() => value += 1}
  decrement={() => value -= 1}
/>
```

- Components
- Reactivity
- Props
- Logic
- Events
- **Bindings**
- Classes & Styles
- Actions
- Transitions

Bindings

Bindings

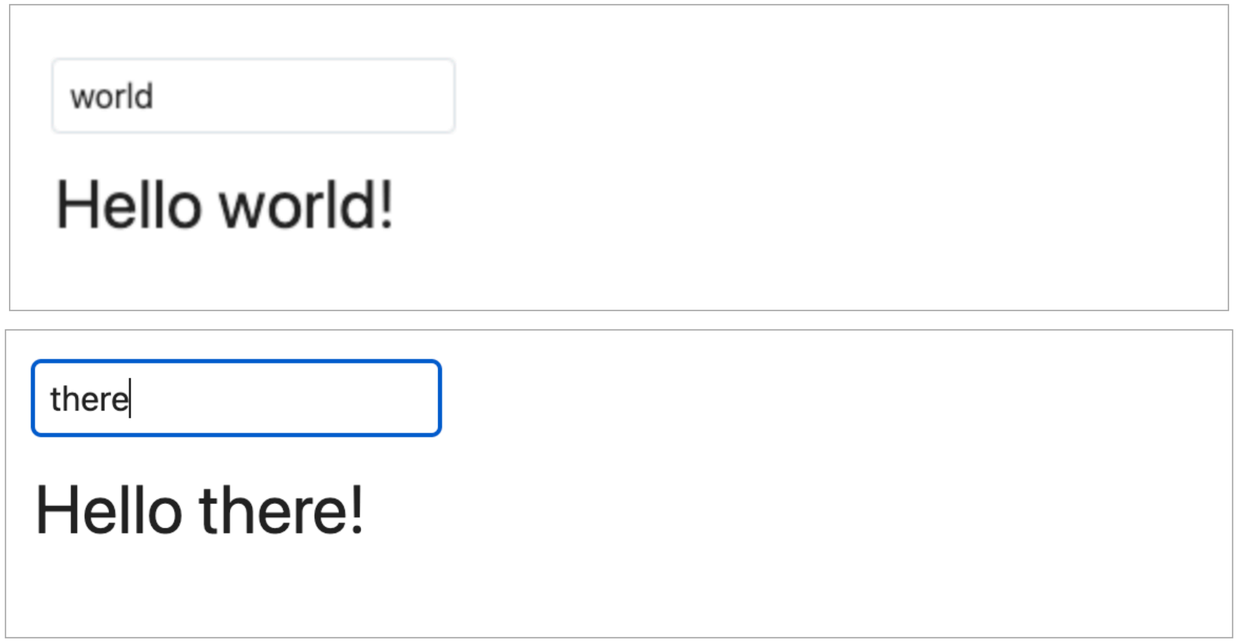
- Data flow in Svelte is top down — a parent component can set props on a child component, and a component can set attributes on an element, but not the other way around.

- Sometimes it's useful to break that rule with the bind:value directive,

- This means that not only will changes to the value of **name** update the input value, but changes to the input value will update **name**.

App.svelte

```
<script>
  let name = $state('world');
</script>

<input bind:value={name} />

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



Bindings: Numeric Inputs

- In the DOM, every input value is a string.
- That's unhelpful when you're dealing with numeric inputs — **type="number"** and **type =“range"**
- It means you have to remember to coerce **input.value** before using it.
- **bind:value** takes care of it for you

App.svelte

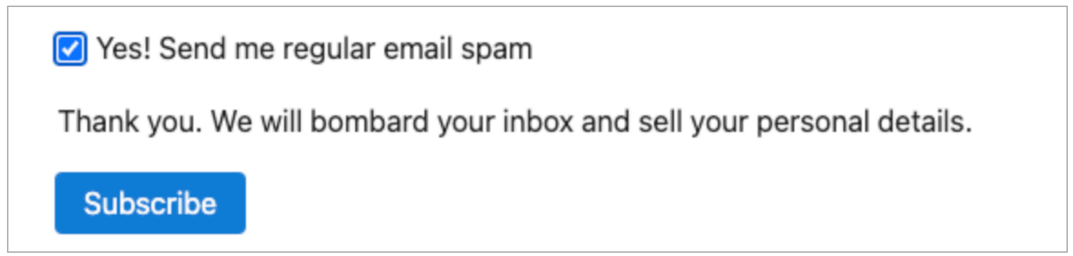
```svelte
<script>
  let a = $state(1);
  let b = $state(2);
</script>

<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>
```

Bindings: Checkbox inputs

- Checkboxes are used for toggling between states.
- Instead of binding to **input.value**, we bind to **input.checked**:

App.svelte

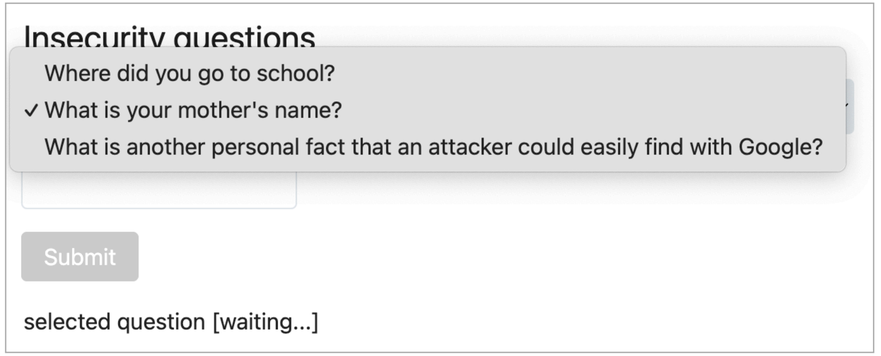
```
<script>
  let yes = $state(false);
</script>

<label>
  <input type="checkbox" bind:checked={yes} />
  Yes! Send me regular email spam
</label>

{#if yes}
  <p>
    Thank you. We will bombard your inbox and sell
    your personal details.
  </p>
{:else}
  <p>
    You must opt in to continue. If you're not
    paying, you're the product.
  </p>
{/if}

<button disabled={!yes}>Subscribe</button>
```



Bindings: Select bindings

```
<script>
  let questions = $state([
    {
      id: 1,
      text: `Where did you go to school?`
    },
    {
      id: 2,
      text: `What is your mother's name?`
    },
    {
      id: 3,
      text: `What is another personal fact that an attacker could easily find with Google?`
    }
  ]);

  let selected = $state();

  let answer = $state('');

  function handleSubmit(e) {
    e.preventDefault();

    alert(
      `answered question ${selected.id} (${selected.text}) with "${answer}"`
    );
  }
</script>
```

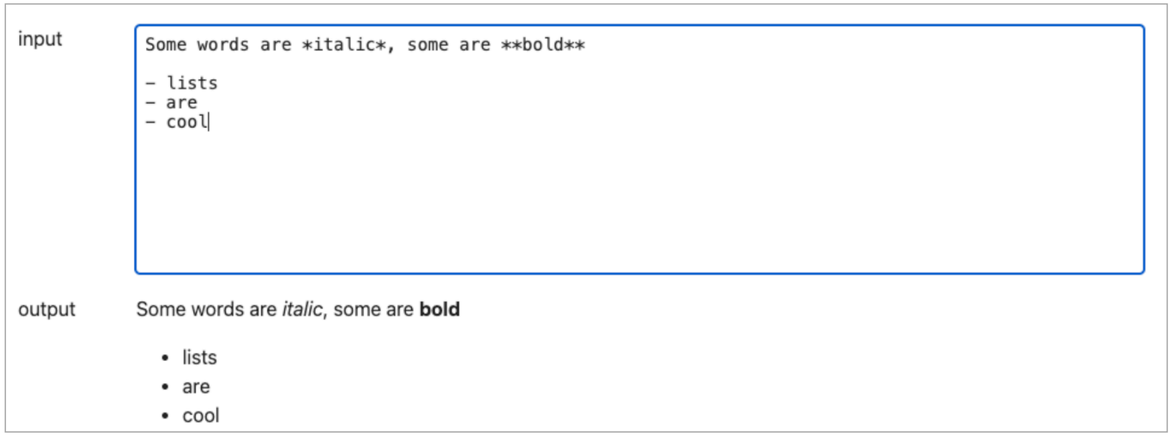
```
<h2>Insecurity questions</h2>

<form onsubmit={handleSubmit}>
  <select
    bind:value={selected}
    onchange={() => (answer = '')}
  >
    {#each questions as question}
      <option value={question}>
        {question.text}
      </option>
    {/each}
  </select>

  <input bind:value={answer} />

  <button disabled={!answer} type="submit">
    Submit
  </button>
</form>

<p>
  selected question {selected
    ? selected.id
    : '[waiting...]'}
</p>
```



- We can also use bind:value with <select> elements
- Note that the <option> values are objects rather than strings.

Bindings: Textarea inputs

- The **<textarea>** element behaves similarly to a text input in Svelte — use **bind:value**



App.svelte

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

  let value = $state(`Some words are *italic*, some are **bold**\n\n- lists\n- are\n- cool`);
</script>

<div class="grid">
  input
  <textarea bind:value></textarea>

  output
  <div>{@html marked(value)}</div>
</div>

<style>
  .grid {
    display: grid;
    grid-template-columns: 5em 1fr;
    grid-template-rows: 1fr 1fr;
    grid-gap: 1em;
    height: 100%;
  }

  textarea {
    flex: 1;
    resize: none;
  }
</style>
```

- Components
- Reactivity
- Props
- Logic
- Events
- Bindings
- **Classes & Styles**
- **Actions**
- **Transitions**

Classes & Styles

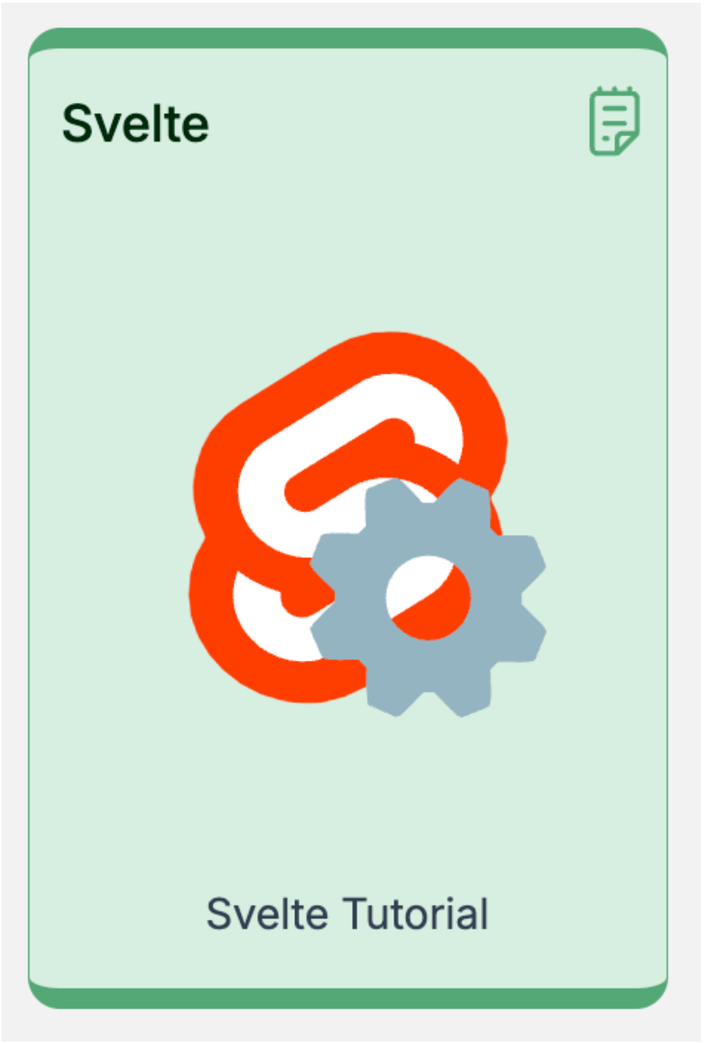
- Convenient techniques for dynamically changing element classes

Actions

- Element-level lifecycle functions, useful for interfacing with third-party libraries, lazy-loaded images, tooltip, custom event handlers

Transitions

- Gracefully transitioning elements into and out of the DOM


Svelte
Svelte Tutorial