LEARN.SVELTE.DEV

What is svelte? A framework for building apps declaratively, out of components that combine markup, styles and behaviours

Tutorial Overview

1. Basic svelte
2. Advanced svelte
3. Basic sveltekit
4. Advanced sveltekit

Part 1 =

PART - 1 BASIC SVELTE

Components

A component is a reusable block of code that encapsulates HTML, CSS and JavaScript that belong together, written into a .svelte file.

Eg. (Adding data)

App.svelte

```
<script>
    let name = 'svelte';
</script>
<h1>Hello {name.toUpperCase()}!</h1>
```

Will display

Hello svelte!

SVELTE!

Dynamic attributes

App.svelte

```
<script>
    let src = 'image.gif';
    let name = 'Rick Astley';
</script>
<img {src} alt="{name} dances">
```

Note

src={src} in shorthand {src}
attribute

Styling

App.svelte

```
<p> This is styled. </p>
<style>
    p {
        color: red
    }
</style>
```

Note

These rules are scoped to the component

Nested components

App.svelte

Nested.svelte

```
<p> Hello! </p>
```

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

Note:
The styles applied in Nested.svelte will not be effected by the styles applied in App.svelte

HTML Tags

App.svelte

```
<script>
  let string = 'This <strong> HTML </strong> in here';
</script>
<p> {@html string} </p>
```

... Will display ..

This **HTML** in here

Note:
Because of HTML Tags they can be used for
'Cross-site Scripting'
↓
Research this if you get time.

Reactivity

Is responsible for keeping the DOM in sync with the state of your application

Eg: using event handlers, note the assigment below

App.svelte

```
<script>
  let count = 0
  function increment () {
      count += 1      }
</script>
<button on:click={increment}>
  Clicked {count} {count===1 ? 'time' : 'times'}
</button>
```

Reactive Declaration

To declare a variable which will be updated when count changes: Add this to their corresponding tags.

App.svelte

```
<scrpt>
    $: doubled = count * 2
</scrpt>
<p> Doubled is {doubled} </p>
```

Without $:

The doubled var will not be updated on:click

Statements

You can also declare more using the $:

eg:
```
$: {console.log('somthn');
    console.log('natha');
   }

$: if (count >= 10) {
     alert('count is high!');
     count = 0;
   }
```

Updating Arrays and Objects

Svelte reactivity is triggered by assignments.
(Using .push and .splice won't automatically cause updates)

Consider the code below, without the numbers = numbers line the code is useless, this line is a simple fix)

App.svelte

```
<script> let numbers = [1,2,3,4];
  function addNumber() { numbers.push(numbers.length+1);
-> numbers = numbers; <- }

$: sum = numbers.reduce((total, currentNumber) => total + currentNumber, 0);
</script>
<p> {numbers.join(' + ')} = {sum} </p>
<button on:click={addNumber}>
    Add a number
</button>
```

There is a more idiomatic solution (Appropriate)
Change the code inside the Function

App.svelte

```
function addNumber() {
    numbers = [...numbers, numbers.length + 1];
}
```

PROPS

Declaring props

→ In order to pass data from one component down to its children.

→ To do that we declare properties in short 'prop'

→ Use the 'export' keyword

[App.svelte]

```
<script>
    import Nested from './Nested.svelte';
</script>
<Nested answer={42} />
```

[Nested.svelte]

```
<script>
    export let answer;
</script>
<p>The answer is {answer}</p>
```

To add a default value use:

Nested.svelte
```
export let answer = 'default value';
```

To view this add a second component after the code

App.svelte
```
<Nested />
```

Spread props

Consider the two components displayed in the App.svelte, the second is a short solution for the spread proper in case one forgets to declare them while displaying the child component

[App.svelte]

```
<script>
    import PackageInfo from './PackageInfo.svelte';
    const pkg = {
        name: 'svelte',
        speed: 'blazing',
        version: '4'
    }
</script>

① <PackageInfo
    name={pkg.name}
    speed={pkg.speed} />

② <PackageInfo {...pkg} />
```

[Package_Info.svelte]

```
<script>
    export let name;
    "      "  speed;
    "      "  version;

// $: href = `https://...`; //
</script>

<p> The {name} package is {speed} fast.
    Download {version}{version} now. </p>
```

LOGIC

Html does not have a way of expressing logic

If block {IF}

To conditionally render some mark up, we wrap it in an if block

Take the reactivity example in page (4)

Example

App.svelte

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

□ Else blocks.

Use the character : as in {:else} to indicate a block continuation tag within the if block

eg. {#if count > 10}
<P> ... <P>
{:else}
<P> ... <P>
{/if}

□ Else-if blocks

To chain multiple conditions, before the {:else} block use the {:else if } blocks for more conditions.

[EACH]

□ Each blocks.

Consider the example below where a list of colors can be used to change the text color

App.svelte

```
<script>
    const colors = ['red', 'orange', ...]
    let selected = colors[0];
</script>
<div>
    {#each colors as color, i}
<button style="background:{color}"
        on:click={() => selected = color}>
{i+1}</button>
    {/each}
</div>
```

□ keyed each block
For -> correct DOM updates
-> Improved performance
-> Better developer experience

A key which tells svelte how to figure out what to update when values change

[App.svelte]

```
<script> import Thing from './Thing.svelte';
  let things = [
    {id: 1, name: 'apple'},
    ...
  ];

  function handleClicked() {
    things = things.slice(1);
  }
</script>

<button on:click={handleClicked}>
  Remove first thing
</button>

{#each things as thing (thing.id)}
  <Thing name={thing.name} />
{/each}
```

[Thing.svelte]

```
<script>
  const emojis = { apple: '🍎' ... };

  export let name;
  const emoji = emojis[name];
</script>

<p>{emoji} = {name}</p>
```

AWAIT BLOCKS

Most web application have to deal with asynchronous data at some point

To await the value of a promise:

This is for a non-blocking architechure

[App.svelte]

```
<script>
    import { getRandomNumber } from './utils.js';

    let promise = getRandomNumber();

    function clicked() {
        promise = getRandomNumber();
    }
</script>

<button on:click={clicked}>generate random no</button>

{#await promise}<p>...waiting</p>
{:then number}<p>The number is {number}</p>
{:catch error}<p style="color:red">{error.message}</p>
{/await}
```

[utils.js]

```
export async function getRandomNumber() {
    const res = await fetch('/random-number');
    if (res.ok) { return await res.text(); }
    else { throw new Error('Request failed'); }
}
```