Lecture 5

Svelte

Frontend Web Development

Terminology

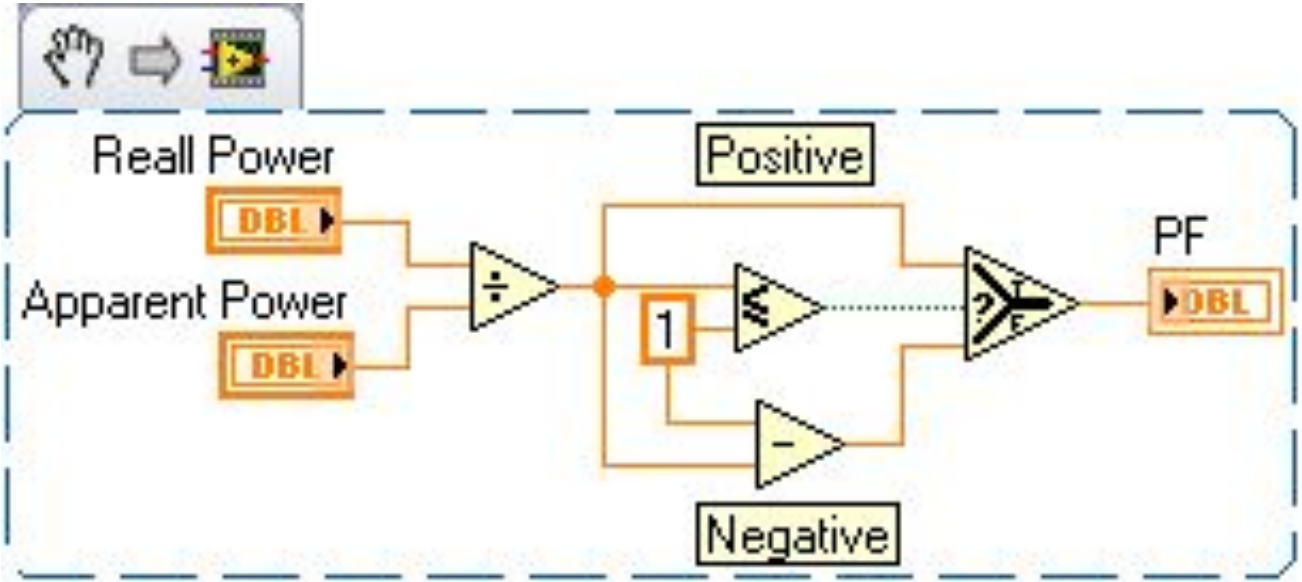
Reactive Programming

- Declarative
- Defines relationships rather than operations
- Updates variables when other variables they reference get updated

Example (pseudocode):

```
a := 1
b := 2
c = a + b
a := 10
print(c) # 12
```

Visual Example - LabVIEW



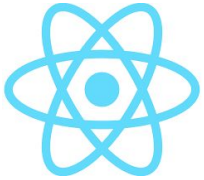
Compilation/Transpilation

Compilation: High-level code → Machine code

Transpilation: High-level code → A different high-level code

Practically speaking, the difference in terminology doesn't matter much.

Framework vs. Library



A **library** is a package that implements some particular functionality that you import and use.

Lo

A **framework** implements the **Inversion of Control** principle, where you fill in the blanks and it controls calling your code for you.

Framework	Library
Provides ready to use tools, standards, templates, and policies for fast application development	Provides reusable function for our code
The framework controls calling of libraries for our code	Our code controls when and where to call a library
To leverage the benefit of a framework, a fresh application can be developed following the framework's guideline	Library can be added to augment the features of an existing application
Easy to create and deploy an application	Facilitates program binding
Helps us to develop a software application quickly	Helps us to reuse a software function
Intent of a framework is to reduce the complexity of the software development process	Intent of a library is to provide reusable software functionality

https://www.baeldung.com/cs/framework-vs-library

Inversion of Control

We give up control over the lifecycle of the app to the framework. This helps:

- Decouple the execution of a task from its implementation
- Free modules from assumptions about the system
- Focus a module on the task it's designed for
- Prevent side-effects when replacing a module

Jokingly referred to as the "Hollywood Principle" →

Example

jQuery

```
<head>
   <script src="some-cdn/jquery.js"></script>
   <script src="./app.js"></script>
</head>
<body>
   <div id="app">
      <button id="myButton">Submit</button>
   </div>
</body>

// app.js
let error = false;
const errorMsg = 'An Error Occurred';
$('#myButton').on('click', () => {
  error = true; // pretend some error occured
  if (error) {
    $('#app')
      .append(`<p id="error">${errorMsg}</p>`);
  } else {
    $('#error').remove();
  }
});
```

Vue.js

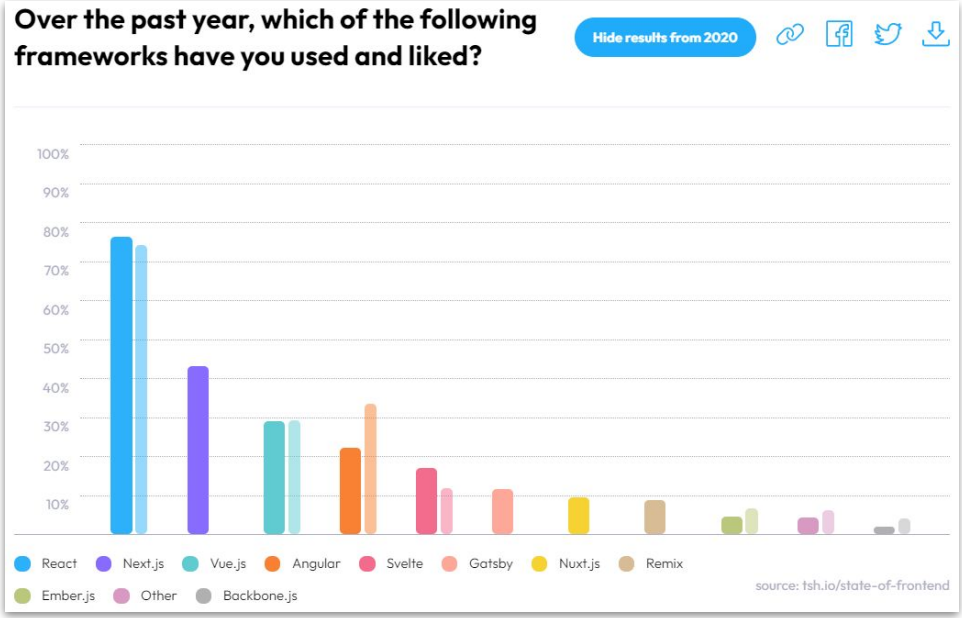
```
<head>
  <script src="some-cdn/vue.js"></script>
  <script src="./app.js"></script>
</head>
<body>
  <div id="app"></div>
</body>

// app.js
const vm = new Vue({
  template: `<div>
                <button
@click="checkForErrors">Submit</button>
              <p v-if="error">{{ errorMsg }}</p>
            </div>`,
  el: '#app',
  data: {
    error: false,
    errorMsg: 'An Error Occurred',
  },
  methods: {
    checkForErrors()  {
      this.error = !this.error;
    },
  },
});
```

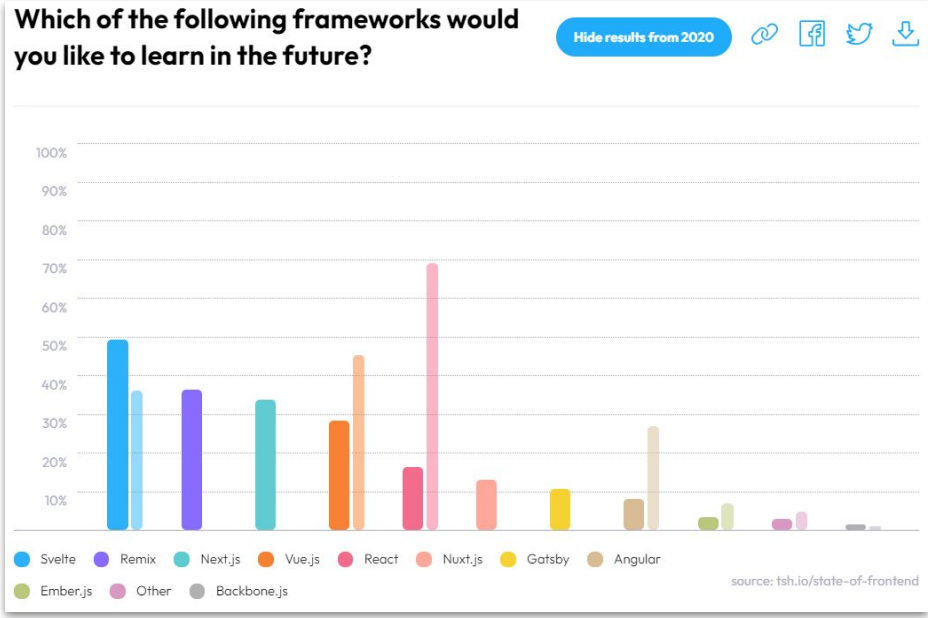
Single File Components

SFCs are a code organization style used by some JavaScript UI libraries/frameworks. An SFC encapsulates the related **template** (HTML), **logic** (JavaScript), and **style** (CSS) of a component in one file.

Components are modular and reusable, provide a more ergonomic syntax, aid with the single responsibility principle, and can provide some compile-time optimizations.

Industry State





[State of Frontend 2022 by The Software House]

Industry State

State of JS 2022

Svelte

Svelte

Svelte is one such JavaScript framework implementing the idea of SFCs that is:

- Free & open-source
- Compiled
- Shallow learning curve

Being compiled helps it avoid a lot of the overhead that comes with shipping a runtime with things like Virtual DOM (like Vue and React do)

SFC Anatomy

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

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

<style>
h1 {
    color: purple;
}
</style>
```

Logic

Markup (template)

Styles

Template interpolation

All values declared in the `script` tag are visible in the markup and can be used for interpolation:

```
<script>
    let src = 'https://media2.giphy.com/media/Ju7l5y9osyymQ/giphy.gif';
    let name = 'Rick Astley';
</script>

<p>Here is a gif of {name}:</p>
<!-- {src} is short for src={src} -->
<img {src} alt="{name} dancing" />
```

Style scoping

Styles written in one component only apply to that component and don't leak.

Example component code:

```
<p>Styled!</p>

<style>
    p {
        color: purple;
        font-family: Arial;
        font-size: 2em;
    }
</style>
```

Part of the JavaScript output:

```
p = element("p");
p.textContent = "Styled!";
attr(p, "class", "svelte-q3qk6d");
```

CSS output:

```
p.svelte-q3qk6d {
    color: purple;
    font-family: Arial;
    font-size: 2em;
}
```

Nested components

App.svelte:

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

<p>These styles...</p>
<Nested />

<style>
    p {
        color: purple;
        font-size: 2em;
    }
</style>
```

Nested.svelte:

```
<p>...don't affect this element</p>
```

Props and Reactivity

Props

Props are the way to define input to a component

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>
```

The answer is 42

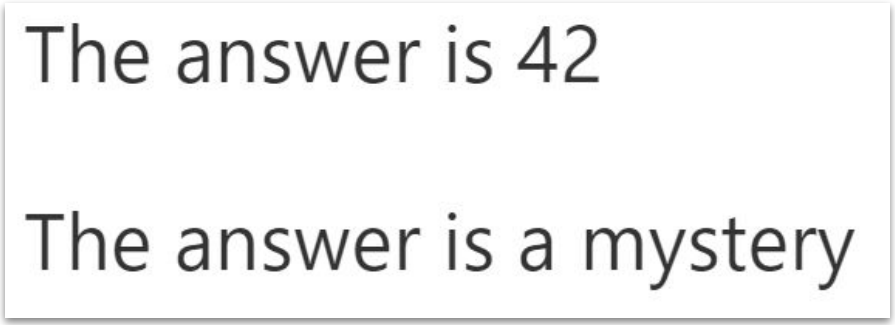
Props - Default values

App.svelte:

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

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

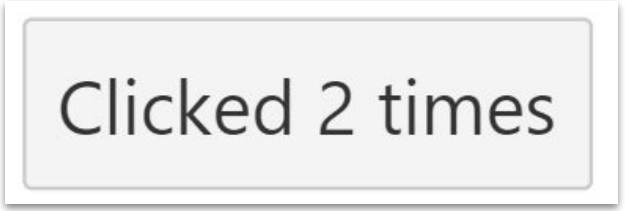
Nested.svelte:

```
<script>
  export let answer = 'a mystery';
</script>

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



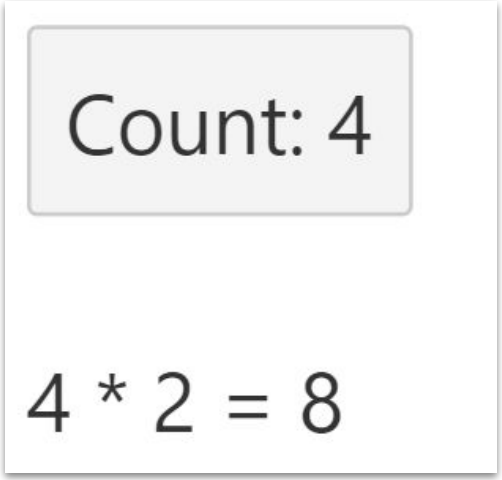
Reactivity

```
<script>
    let count = 0;

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

<button on:click={handleClick}>
    Clicked {count} times
</button>
```



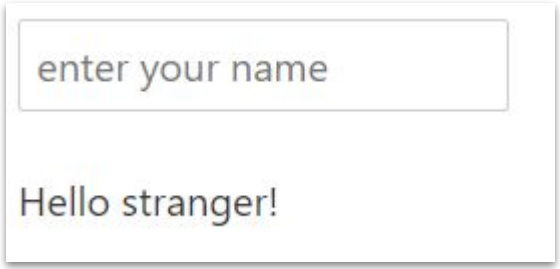
Reactive Declarations

```
<script>
    let count = 1;

    // the `$:` means 're-run whenever these values change'
    $: doubled = count * 2;

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

<button on:click={handleClick}>
    Count: {count}
</button>
<p>{count} * 2 = {doubled}</p>
```



2-way Data Binding

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

<input bind:value={name} placeholder="enter your name">
<p>Hello {name || 'stranger'}!</p>
```



Templating logic

Conditional Rendering

```
<script>
  export let porridge;
</script>

{#if porridge.temperature > 100}
  <p>too hot!</p>
{:else if porridge.temperature < 80}
  <p>too cold!</p>
{:else}
  <p>just right!</p>
{/if}
```

Looping

```
<h1>Shopping list</h1>
<ul>
  {#each items as item, i}
    <li>{i} - {item.name} x {item.qty}</li>
  {/each}
</ul>
```

Await blocks

```
<script>
    let promise = fetchSomeNumber();
</script>

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

Events

DOM Events

```
<script>
    function handleClick() {
        alert('first and last alert')
    }
</script>

<button on:click|once={handleClick}>
    Click me
</button>
```

Event Forwarding

App.svelte:

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

  function handleClick() {
    alert('clicked');
  }
</script>

<CustomButton on:click={handleClick}/>
```

CustomButton.svelte:

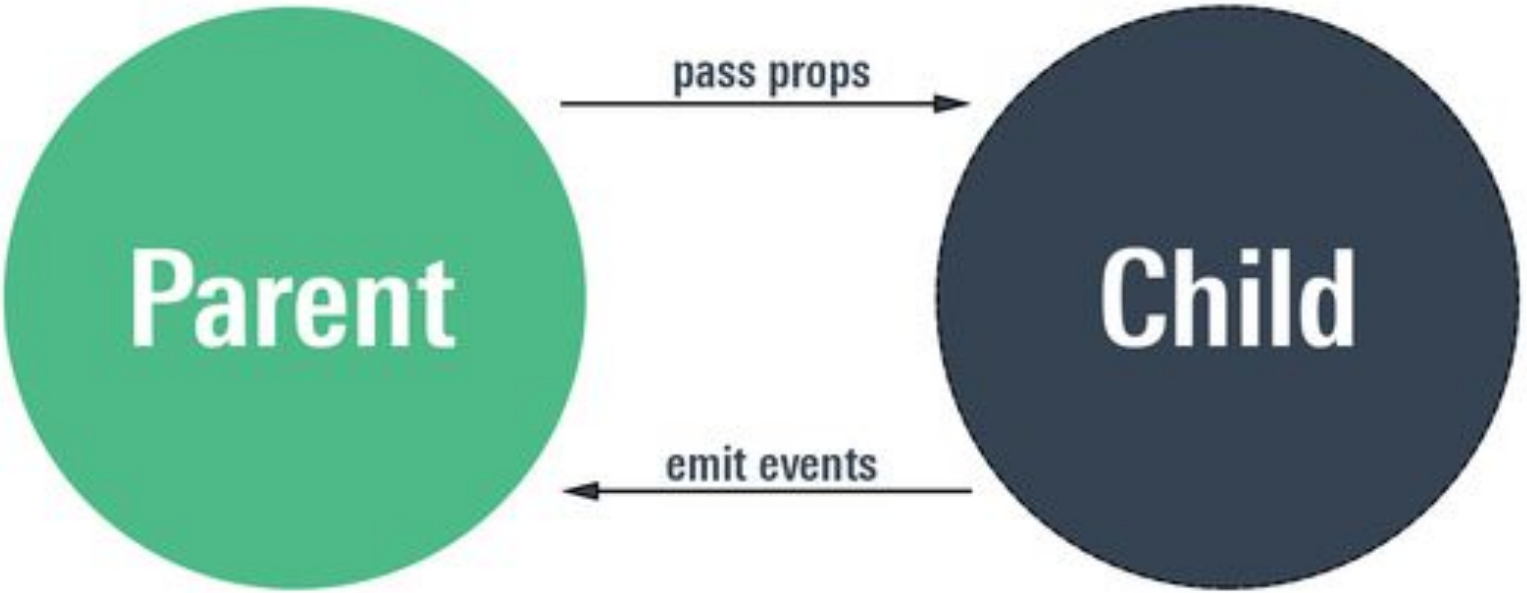
```
<button on:click>
    Click me
</button>
```

Dispatching custom events

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

  function handleMessage(event) {
    alert(event.detail.text);
  }
</script>

<Inner on:message={handleMessage}/>
```

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

  const dispatch = createEventDispatcher();

  function sayHello() {
    dispatch('message', {text: 'Hello!'});
  }
</script>

<button on:click={sayHello}>
    Click to say hello
</button>
```

Props + Events = Communication between components



Slots

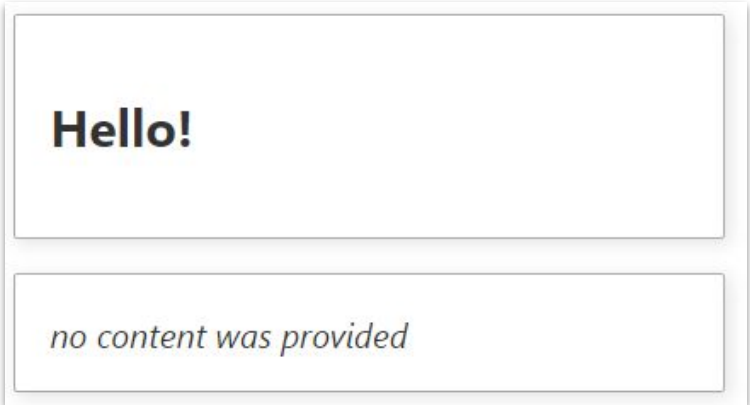
Slots

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

<Box>
  <h2>Hello!</h2>
  <p>This is a box. It can contain
anything.</p>
</Box>
```


```
<div class="box">
    <slot></slot>
</div>

<style>
    .box {
        width: 300px;
        border: 1px solid #aaa;
        /* … */
        padding: 1em;
        margin: 0 0 1em 0;
    }
</style>
```

Slot fallback

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

<Box>
  <h2>Hello!</h2>
</Box>

<Box />
```

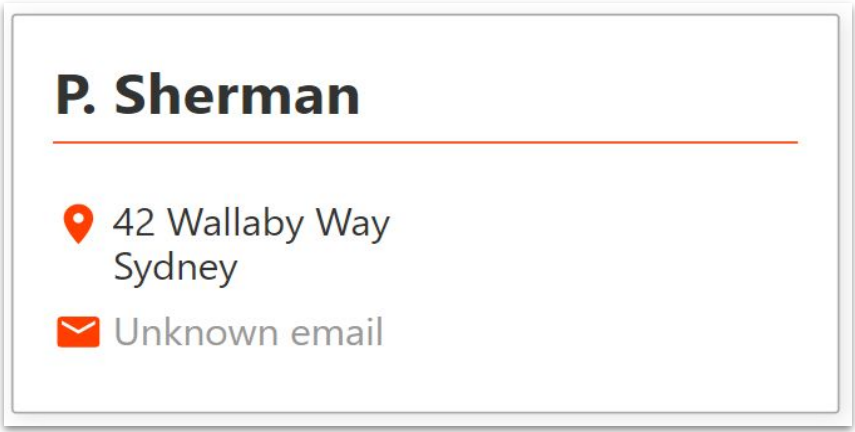



```
<div class="box">
  <slot>
    <em>no content was provided</em>
  </slot>
</div>

<style>
    .box {
        width: 300px;
        border: 1px solid #aaa;
        /* … */
        padding: 1em;
        margin: 0 0 1em 0;
    }
</style>
```

Named Slots

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

<ContactCard>
    <span slot="name">
        P. Sherman
    </span>

    <span slot="address">
        42 Wallaby Way<br>
        Sydney
    </span>
</ContactCard>
```

```
<article class="contact-card">
    <h2><slot name="name"></slot></h2>
    <div class="address">
        <slot name="address"></slot>
    </div>
    <div class="email">
        <slot name="email"></slot>
    </div>
</article>
<style> /* … */ </style>
```



Slot Props

App.svelte:

```
<FancyList {items} let:prop={thing}>
  <div>{thing.text}</div>
</FancyList>
```

FancyList.svelte:

```
<ul>
  {#each items as item}
    <li class="fancy">
      <slot prop={item}></slot>
    </li>
  {/each}
</ul>
```

Named Slot Props

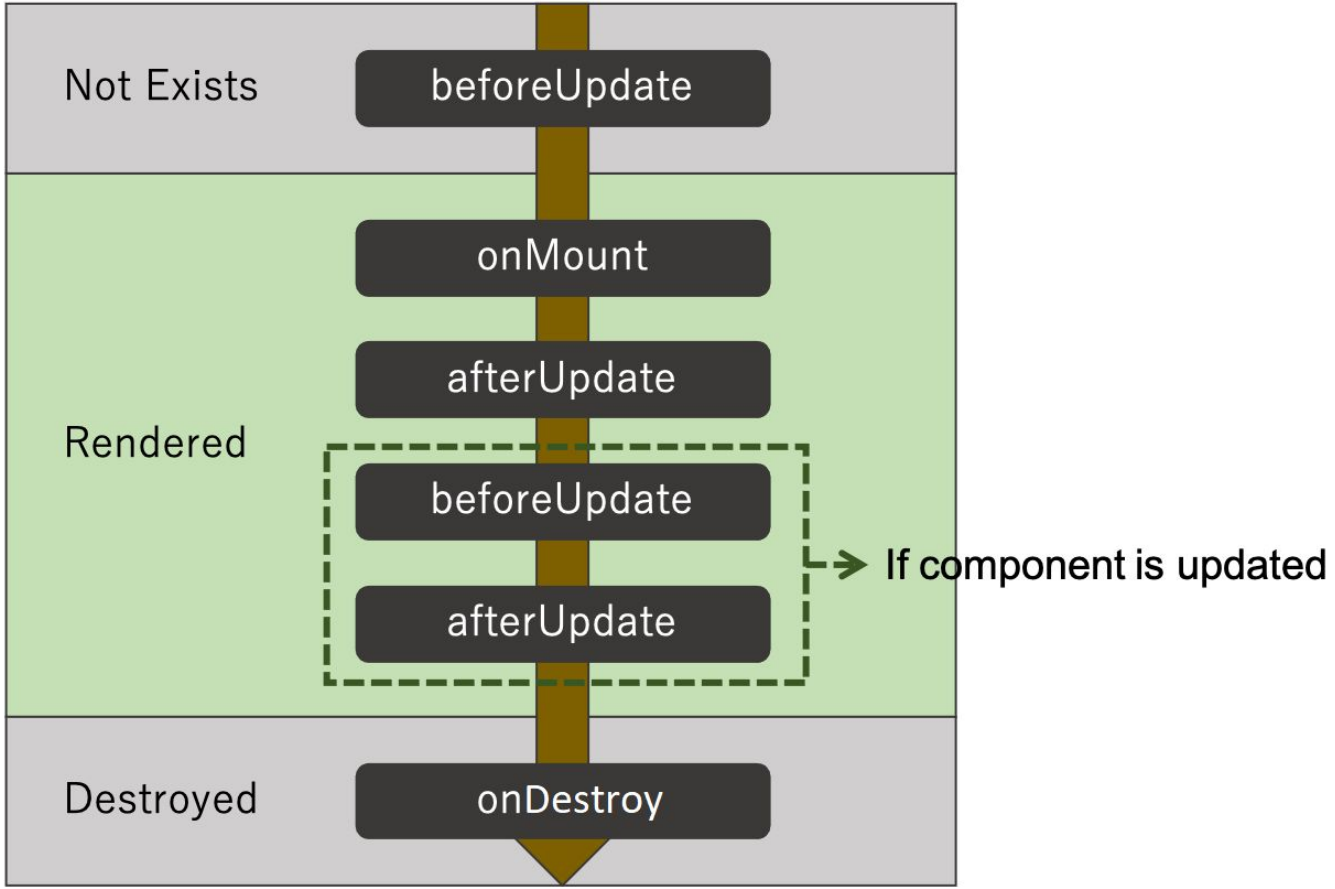
App.svelte:

```
<FancyList {items}>
  <div
    slot="item"
    let:item
  >
    {item.text}
  </div>
</FancyList>
```

FancyList.svelte:

```
<ul>
  {#each items as item}
    <li class="fancy">
      <slot
        name=“item”
        prop={item}
      ></slot>
    </li>
  {/each}
</ul>
```

Lifecycle hooks



Component Lifecycle

https://javascript.plainenglish.io/lifecycle-of-the-svelte-component-ef00c1969a4a

onMount

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

    let photos = [];

    onMount(async () => {
        const res = await fetch(`/tutorial/api/album`);
        photos = await res.json();
    });
</script>
```

onDestroy

```
import { onDestroy } from 'svelte';

export function onInterval(callback, milliseconds) {
    const interval = setInterval(callback, milliseconds);

    onDestroy(() => {
        clearInterval(interval);
    });
}
```

beforeUpdate

Schedules a callback to run immediately before the component is updated after any state change.

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

  beforeUpdate(() => {
    console.log('the component is about to update');
  });
</script>
```

afterUpdate

You try to guess this one...

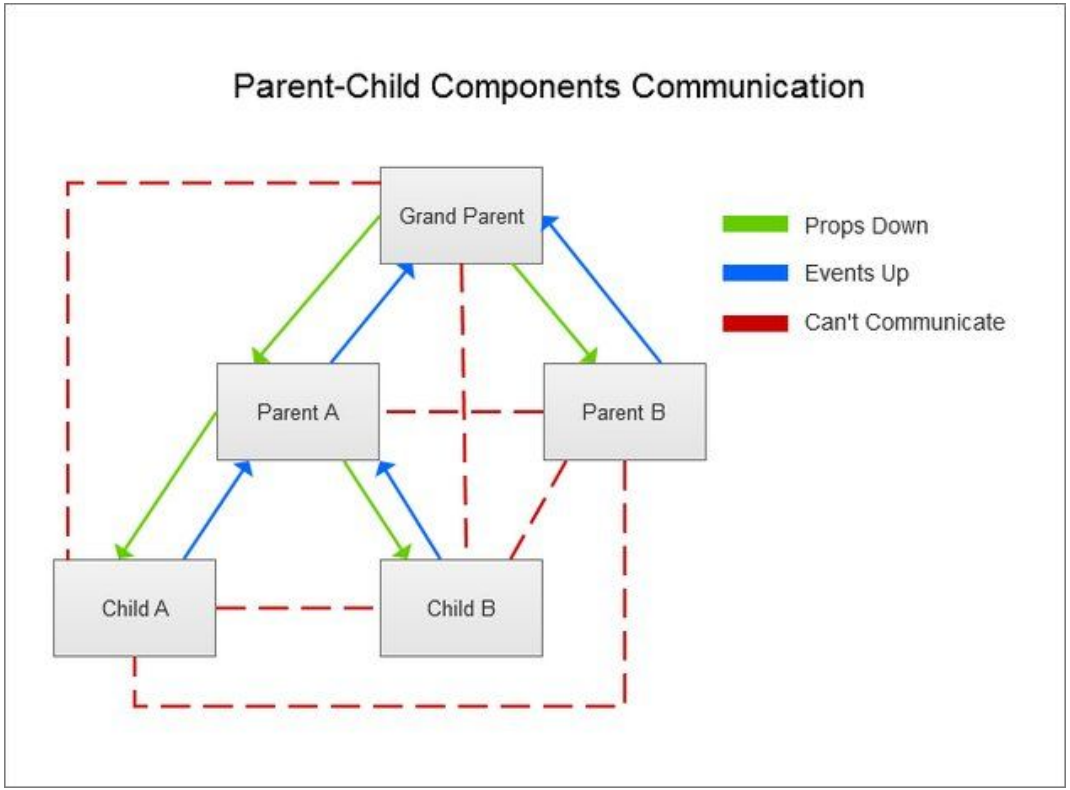
tick

Returns a promise that resolves once any pending state changes have been applied, or in the next microtask if there are none.

```
<script>
    import { beforeUpdate, tick } from 'svelte';

    beforeUpdate(async () => {
        console.log('the component is about to update');
        await tick();
        console.log('the component just updated');
    });
</script>
```

Data sharing

What if we want more complicated communication?



Stores

A way of sharing data with all components.

We can use them to implement reactivity outside Svelte components (in normal JS/TS files, where the "`$:`" syntax doesn't mean anything).

There are **readable**, **writable**, and **derived** stores.

Writable stores

```
import { writable } from 'svelte/store';

const count = writable(0);

count.subscribe(value => {
    console.log(value);
}); // logs '0' immediately

count.set(1); // logs '1'

count.update(n => n + 1); // logs '2'
```

Derived Stores

```
import { writable, derived } from 'svelte/store';

export const numbers = writable([1, 2, 3]);

export const total = derived(numbers, $nums =>
    $nums.reduce((a, b) => a + b, 0));
```

Store auto-subscription

```
<script>
    import { total } from './stores.js';
</script>

<p>The total is {$total}</p>
```

This syntax only works inside .svelte files

Context

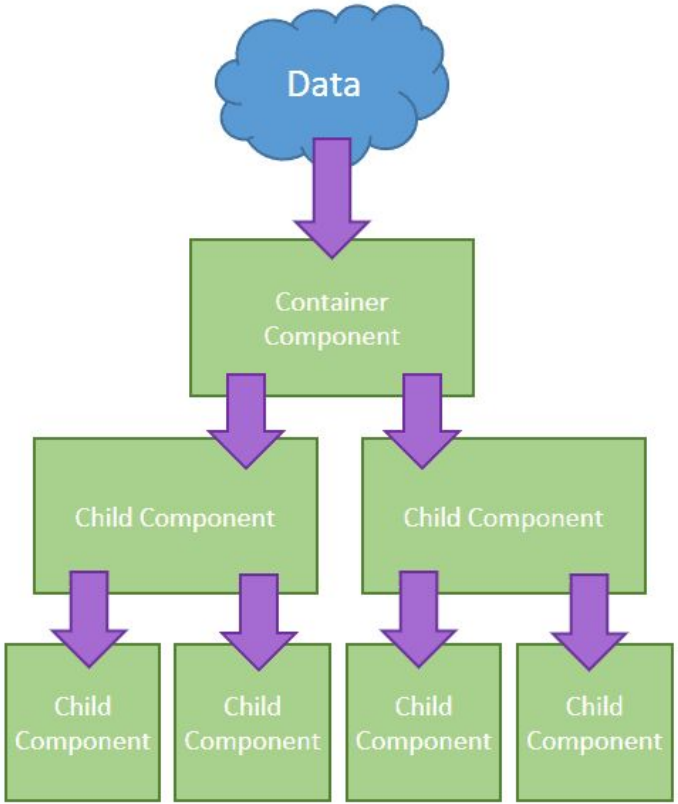
Another way of communicating across components but with descendants only. Context is **not** reactive, but you may use a store as its value for reactivity.

Parent.svelte:

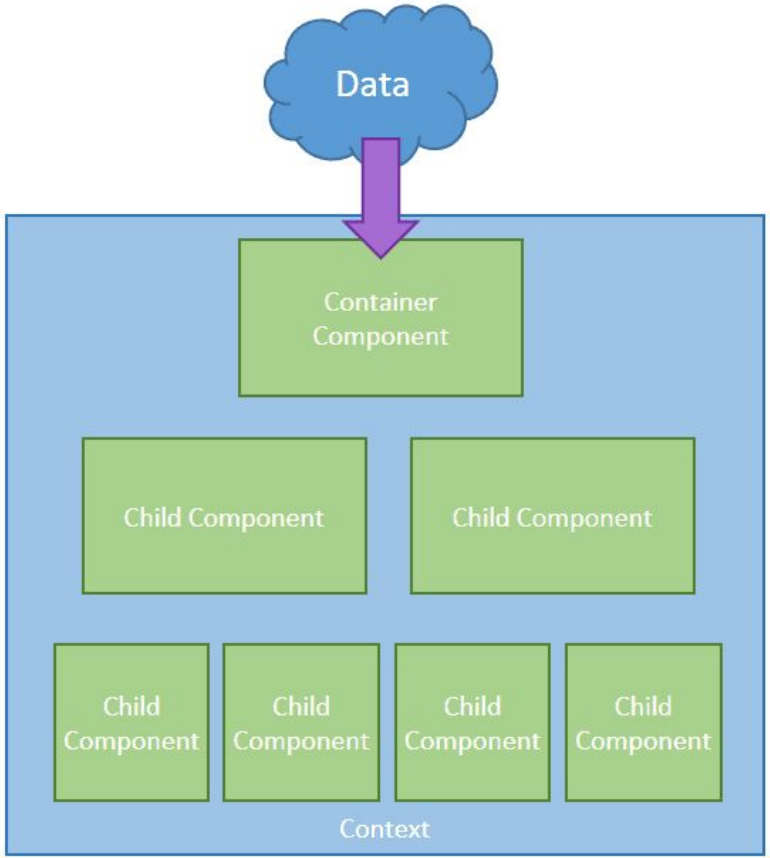
```
<script>
  import { setContext } from 'svelte';
  setContext('answer', 42);
</script>
```

GrandChild.svelte:

```
<script>
  import { getContext } from 'svelte';
  const answer = getContext('answer');
</script>

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



prop drilling



context API

https://www.carlrippon.com/playing-with-the-context-api-in-react-16-3/

Special elements

<svelte:self>

```
<script>
    export let count = 3;
</script>

{#if count > 0}
    <p>counting down... {count}</p>
    <svelte:self count={count - 1} />
{:else}
    <p>lift-off!</p>
{/if}
```

<svelte:head>

```
<svelte:head>
    <title>Home page</title>
    <link rel="stylesheet" href="/tutorial/dark-theme.css">
</svelte:head>

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

<svelte:window>

Allows adding event listeners to the `window` object, and binding to some properties on it.

```
<script>
  let y = 0;
  function handleKeydown(event) {
    alert(`pressed the ${event.key} key; scroll position=${y}`);
  }
</script>

<svelte:window on:keydown={handleKeydown} bind:scrollY={y} />
```

And more...

- `<svelte:component this={selectedComponent} />`
- `<svelte:element this={expression} />`
- `<svelte:body on:event={handler} />`
- `<svelte:options option={value} />`

100 SECONDS OF

References

https://svelte.dev/examples/

https://svelte.dev/tutorial

https://svelte.dev/repl (Playground for Svelte)