

SVELTE

¿Qué es?

Es un framework de **JavaScript** para construir **interfaces de usuario**.



¿Qué ventajas tiene?

- ¡Escribe menos código!
- No utiliza un Virtual DOM : compila nuestros componentes cuando realizamos un cambio
- Sintaxis sencilla y corta para crear aplicaciones basadas en componentes

Instalación

- Si quieras crear un proyecto desde cero, ejecuta este comando en la terminal :

```
npx degit sveltejs/template [nombre del proyecto]  
npm install  
npm run dev
```

estructura del proyecto



- `node_modules`: nos permite tener las dependencias instaladas.
- `/public`: donde se encuentran los archivos que se exponen al compilar el proyecto
 - = `favicon.png`
 - = `global.css`
 - = `index.html`
- `/src`: contiene dos archivos particulares
 - = `App.svelte`
 - = `main.js`
- `.gitignore`
- `package.json`
- `rollup.config.js`: herramienta que se encarga de compilar el proyecto.

Componentes

constan de → 3 elementos



- Etiqueta `<script>`: Para la lógica
- Etiqueta `<style>`: Para los estilos
- Escribir el `HTML`

Los componentes deben tener extensión `.svelte`
Por ejemplo: `About.svelte`

💡 · ¡Crea una carpeta para guardar tus componentes!

```
<script>  
Lógica  
</script>
```

```
<style>  
Estilos  
</style>
```

```
<h1>Hola</h1>
```



Es buena práctica crear un div asignándole el nombre del archivo para establecer todo lo que será parte del componente.

```
<div class = "About">  
  <h1>Hola</h1>  
</div>
```

Vamos con un ejemplo:

En About.svelte:

```
<script>  
  let someText = 'Frontend Developer';  
</script>
```

```
<style>  
  p {  
    color: #0084F6;  
  }  
</style>  
  
<div class = "About">  
  <p> {someText} </p>  
</div>
```



- * Este componente creado debe ser importado a App.svelte (o donde lo vayas a utilizar). Lo colocamos en el <script>

```
<script>  
  import About from "./components/About.svelte";  
  // carpeta donde se encuentra  
</script>
```
- ¡Ahora disponemos de **About** para utilizarlo!

REACTIVIDAD

Mantiene el DOM sincronizado con el estado de su aplicación.

Por ejemplo, con un controlador de eventos.

- Creamos una nueva variable:

`let count = 0;` → valor inicial de la variable

- Creamos un botón que aplicará los cambios:

`<button> Click {count} </button>`

- Creamos la función que manejará los clicks:

`function handleClick () {
 count += 1; → valor que queremos incrementar
}`

- Utilizamos la función dentro del botón creado:

`<button on:click="handleClick"> Click {count} </button>`



Cada vez que se hace click sobre el botón, la variable count sumará un nuevo valor.



“Declaraciones reactivas”

SVELTE Actualiza el DOM cuando cambia el estado de su componente.

Hay veces que partes del estado de un componente deben calcularse a partir de otras partes (como un **Fullname** derivado de un **Firstname** y un **Lastname**) y recalcularse cada vez que cambian.

Para esto, existen las declaraciones reactivas:

```
let count = 0;  
$: doubled = count * 2;
```

Luego, podemos utilizar **doubled**:

```
<p>{count} duplicado es {doubled}</p>
```



props

En cualquier aplicación real, deberá pasar datos de un componente a sus hijos.

Para esto, necesitamos declarar 'props'

En **SUELTE**, esto se realiza con la palabra clave
export

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

Podemos especificar valores predeterminados para los props:

```
<script>  
  export let answer = 'pugstagram';  
</script>
```



Si TIENES un objeto de PROPIEDADES, PUEDES 'EXTENDERLO' a un COMPONENTE en lugar de ESPECIFICAR cada uno:

<script>

```
import Person from '../Person.SUELTO';  
const data = {
```

name: 'Oscar',

lastName: 'Barajas',

age: 31

```
};
```

```
<Person name={data.name} lastName={data.lastName}  
age={data.age} />
```



```
<Person {...data} />
```



O.

condicionales

O.

Veamos con un **EJEMPLO** cómo **SVELTE** utiliza los condicionales:

<script>

```
let styles = {darkMode: false}
```

```
function toggle() {
```

```
    styles.darkMode = !styles.darkMode
```

```
}
```

} si el valor inicial
es **FALSE**, lo
convierte a
TRUE, y viceversa

¡RECUERDA aplicar los estilos para lograr el efecto **darkMode**!

Luego de estructurar los estilos, creamos un botón para complementar el efecto:

```
<button on:click={toggle}>Dark Mode</button>
```

Para aplicar las condicionales :

```
{#if !styles.darkMode}
```

Pasa ESTO

```
{:else}
```

Pasa ESTO

```
{/if}
```

→ No olvides CERRAR "if"



Estructuras de control

: EACH :



<SCRIPT>

```
let skills = [  
  { id: 1, name: 'HTML' },  
  { id: 2, name: 'CSS' },  
  { id: 3, name: 'JavaScript' }  
>
```

ARREGLO DE OBJETOS
QUE VAMOS A ITERAR
MEDIANTE EACH

</SCRIPT>

• <div class="skills">

```
{#each skills as {name}, i}
```

indice

ELEMENTO QUE QUEREMOS ITERAR → DESESTRUCTURAR EL ELEMENTO NECESARIO

```
<li>{name}</li>
```

```
{/each}
```

</div>

OTRA OPCIÓN:

• <div class="skills">

```
{#each skills as skill, i}
```

```
<li>{skill.name}</li>
```

```
{/each}
```

</div>

ES RECOMENDABLE UTILIZAR
ESTA CONVENCIÓN CUANDO
TENEMOS MÁS DE TRES
ELEMENTOS EN EL ARRAY

await blocks —

La mayoría de las aplicaciones web TIENEN que lidiar con datos asincrónos.

Svelte hace que sea fácil ESPERAR el valor:

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



Si ya sabemos que nuestra PROMESA no SE PUEDE RECHAZAR, podemos omitir el bloque **catch**.

También podemos omitir el primer bloque si no deseamos mostrar nada hasta que se resuelva la PROMESA:

```
{#await promise then value}  
<p>the value is {value}</p>  
{/await}
```

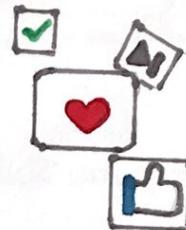


eventos del DOM

Con los cuales un usuario desencadena una actualización que se verá reflejada en nuestra aplicación.

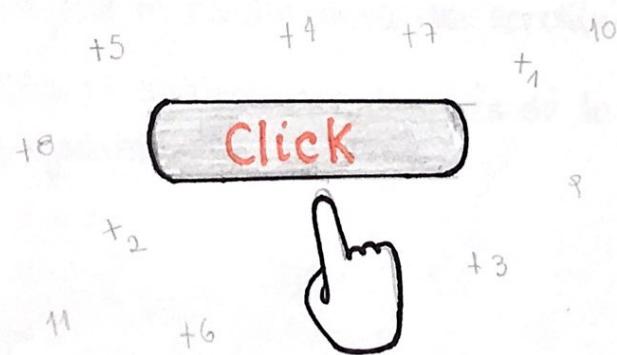
PUEDES ESCUCHAR CUALQUIER EVENTO EN UN ELEMENTO CON LA DIRECTIVA **ON:**

Por ejemplo → **ON: click**
al hacer click
desencadenará
una función.



```
function handleClick() {  
    count += 1  
}
```

```
<button text="Click" on:click={handleClick}></button>
```



modificadores de eventos.

Los controladores de eventos DOM pueden tener modificadores que alteren su comportamiento.

Por ejemplo, un controlador con un modificador **once**, sólo se ejecutará una vez.

```
<script>
```

```
  Function handleClick() {  
    alert('no more alerts')  
  }
```

```
</script>
```

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



Lista de modificadores:

preventDefault: llama a `event.preventDefault()` antes de ejecutar el controlador. Útil para el manejo de forms.

stopPropagation: llama a `event.stopPropagation()`, evitando que el evento llegue al siguiente elemento.

passive: mejora el rendimiento del scrolling.

once: quita el controlador después de la primera vez que se ejecuta.

= BINDINGS



CON SUELTE PUEDES MANEJAR DATOS BIDIRECCIONALES DENTRO DE TU COMPONENTE.

bind:value

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

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



Ahora, si `name` cambia, el campo de entrada (`input`) actualizará su valor.

`bind:value` FUNCIONA EN TODOS LOS TIPOS DE `input`: `type="number"`, `type="email"`. Pero también funciona para otros tipos de campos, como `textarea` y `select`.

checkboxes y radio buttons

`input` con `type="checkbox"` o `type="radio"`. Permiten 3 tipos de bindings:

`bind:checked` → nos permite 'bindear' un valor al estado 'checked' del elemento.

`bind:group` → es útil con checkboxes y radio buttons porque suelen usarse en grupos.

Puedes asociar un array a una lista de checkboxes y completarla en función a las elecciones realizadas por el usuario.

`bind:indeterminate` → nos permite vincular al estado indeterminado de un elemento.

CICLO DE VIDA

Cada componente tiene un ciclo de vida que comienza cuando se crea y termina cuando se destruye.



Existen funciones que permiten ejecutar código en momentos clave durante ese ciclo de vida.

El que utilizaremos con más frecuencia es `onMount` que se ejecutará después de que el componente se procese por primera vez en el DOM.

```
<script>  
import {onMount} from 'velte';  
  
let photos = [];  
  
onMount(async () => {  
    const res = await fetch('https://...')  
    photos = await res.json();  
});  
</script>
```

Las funciones del ciclo de vida deben llamarse mientras el componente se está inicializando para que la devolución (callback) esté vinculada a la instancia del componente.

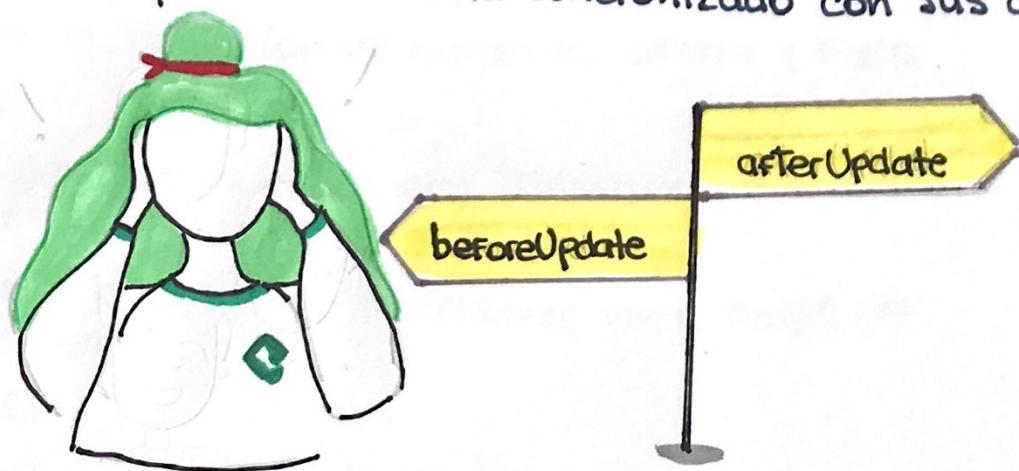
Si el callback de `onMount` devuelve una función, se llamará a esa función cuando se destruya el componente.

Para ejecutar el código cuando su componente se destruye, utilizaremos `onDestroy`.



La función `beforeUpdate` funciona inmediatamente antes de que se haya actualizado el DOM.

`afterUpdate` es su contraparte, se utiliza para ejecutar código una VEZ QUE EL DOM está sincronizado con sus datos.



La Función `tick` ES DIFERENTE a otras funciones del ciclo de vida, YA QUE PODEMOS LLAMARLA EN CUALQUIER MOMENTO, no solo cuando el componente SE INICIALIZA por primera VEZ. Devuelve una PROMESA que se RESUELVE tan PRONTO como se hayan aplicado los cambios de ESTADO PENDIENTES al DOM (o IMMEDIATAMENTE)

—transiciones—

¡Podemos crear interfaces de usuario más atractivas mediante la transición de elementos dentro y fuera del DOM!

Svelte hace esto muy fácil con transition

- Debemos importar las transiciones en el script del componente:

```
<script>  
import {blur} from 'svelte/transition';  
</script>
```

- Luego, activas las transiciones en los elementos

```
<p transition:blur> Hola blur </p>
```

...

Las funciones de transición pueden aceptar parámetros.

```
<script>  
import {fly} from 'svelte/transition';  
</script>
```

```
<p transition:fly = "{{y: 200, duration: 2000}}">  
Volando </p>
```



composición de componentes

Al igual que los elementos, los componentes pueden tener hijos. Sin embargo, antes de aceptarlos, debe saber dónde colocarlos.

ESTO SE REALIZA CON EL ELEMENTO `<slot>`

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

¡Ahora puedes colocar cosas en el box!

```
<box>  
  <h2>Hello</h2>  
</box>
```

o `<slot name = "name">`

o Los slot con nombre permiten apuntar a áreas específicas.

jNUNCA PARES
DE
APRENDER!

