

Descubre Svelte



Miguel Ángel Durán

Descubre Svelte

El framework más amado en el mundo del Frontend

Miguel Angel Durán García

Este libro está a la venta en <http://leanpub.com/descubre-svelte>

Esta versión se publicó en 2022-11-30



Este es un libro de [Leanpub](#). Leanpub anima a los autores y publicadoras con el proceso de publicación. [Lean Publishing](#) es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](#)

¡Tuitea sobre el libro!

Por favor ayuda a Miguel Angel Durán García hablando sobre el libro en [Twitter](#)!

El tuit sugerido para este libro es:

[¡Ya tengo el libro Descubre Svelte de @midudev! ¿Lo quieres? Lo puedes conseguir aquí:](#)

El hashtag sugerido para este libro es [#DescubreSvelte](#).

Descubre lo que otra gente dice sobre el libro haciendo clic en este enlace para buscar el hashtag en Twitter:

[#DescubreSvelte](#)

Índice general

Antes de empezar	1
Únete a la comunidad de Discord	1
Reportando erratas y sugerencias	2
Requerimientos	3
Agradecimientos	3
Introducción a Svelte	4
Cuando descubrí Svelte	4
¿Qué es Svelte?	5
Hola Mundo en Svelte	6
¿Y qué pasa con el estado?	8
Compilación de código	9
Componentes	11
Creando tu primer componente	11
Usando componentes	12
Props en Svelte	13
Spread de props en Svelte	14
Eventos	16
Eventos del DOM	16
Modificadores de eventos	17
Reenvío de eventos del DOM	18
Reactividad y Estado	19
Reactividad en Svelte	19
Valores derivados reactivos	20
Formularios	24
Trabajando con formularios	24
Lógica en tus componentes	29
Renderizado condicional	29
Renderizando listas	31

ÍNDICE GENERAL

Fetching de datos y resolviendo promesas	37
Resolviendo promesas con declaraciones reactivas y condicionales	37
Resolviendo promesas con la sintaxis #await	38
Usando la forma corta de la sintaxis	40
Resumiendo la sintaxis #await	40
Creamos una pequeña app	41
Creando el buscador	41
Obteniendo los resultados	42
Refactorización de la app	46
Crea un proyecto con Svelte	50
Creando el proyecto de Svelte.js usando Vite	50
Instala las dependencias y levanta el servidor	51
Estructura de carpetas del proyecto	53
Revisando el código fuente	54
Instala la extensión para tu editor favorito	54
Conclusiones	55
Despliegue de tu aplicación	56
Desplegando nuestra aplicación en Netlify	58
Configura el dominio de tu aplicación	59
Testing en Svelte	61
¿Qué es una prueba unitaria?	61
Las dependencias que necesitamos	62
Añadiendo scripts y configuración	62
Tu primer test	63
Probando el componente	64
Un caso más complejo	66
Mocking de los datos	68
Slots	71
¿Qué son los slots?	71
Slots nombrados	72
También funciona con componentes	73
Rellena slots sin usar elementos o componentes	73
Ciclo de vida de los componentes	75
Montaje	75
Antes y después de actualizar	76
Destrucción	77
Stores y estado global	80
Creando un store	80

ÍNDICE GENERAL

Usando un store	80
Limpiando la suscripción	82
Auto-suscripción, la magia de Svelte	82
Stores de sólo lectura	84
Animaciones y transiciones	86
Las transiciones	86
Svelte Kit	90
Próximamente	90
Routing con SvelteKit (enrutado)	91
Próximamente	91

Antes de empezar

¡Hola! Soy Miguel Ángel Durán y cuento con 15 años de experiencia en el mundo del desarrollo. Soy [creador de contenido sobre tecnologías web¹](#). He sido reconocido como GitHub Star y como Google Developer Expert en Web Technologies.

He escrito este libro para ayudar a las personas que les puede interesar aprender Svelte pero no se han atrevido por falta de tiempo, por no encontrar contenido o por no saber cómo empezar.

Es una guía rápida y muy práctica donde aprendemos desde cero Svelte, desplegamos a producción una pequeña app y hasta hacemos testing.

Ten en cuenta que se dan por sentados ciertos conocimientos de JavaScript, pero si ya tienes conocimientos de otro framework o biblioteca como **React** y **Vue** no tendrás ningún problema.

Este libro lo puedes [adquirir en LeanPub²](#). Si has disfrutado el libro o quieres apoyar mi contenido, [puedes pagar el precio que consideres](#). Aunque no es obligatorio, me ayudará a seguir escribiendo libros y a mejorar mi contenido.

Si quieres más contenido sobre Svelte, tengo un [curso completo gratuito en mi canal de YouTube³](#).

También en [Twitch⁴](#) hago streamings sobre desarrollo con JavaScript en general. ¡Y nos lo pasamos genial! **Te invito a pasarte**.

Puedes encontrarme en [Twitter como @midudev⁵](#), en [Instagram como @midu.dev⁶](#). También tengo [cuenta en TikTok⁷](#) pero te advierto que **no vas a encontrar ningún baile**.

Por último te dejo [mi página web⁸](#) donde puedes encontrar artículos, cursos, tutoriales y mucho más.

Únete a la comunidad de Discord

Hablando, compartiendo y ayudando a otras personas se aprende mucho. Por eso te invito a que te unas a [mi comunidad oficial de Discord⁹](#). Allí podrás conocer a personas interesadas en el mundo de la programación que comparten recursos de aprendizaje y otras cosas interesantes.

Por supuesto, **tenemos un canal sobre Svelte** que te puede servir para compartir este nuevo conocimiento con más gente.

¹<https://midu.dev/>

²<https://leanpub.com/descubre-svelte>

³https://www.youtube.com/watch?v=Xsxm8_BI63s&list=PLV8x_i1fqBw2QScggh0pw2ATSJg_WHqUN

⁴<https://www.twitch.tv/midudev>

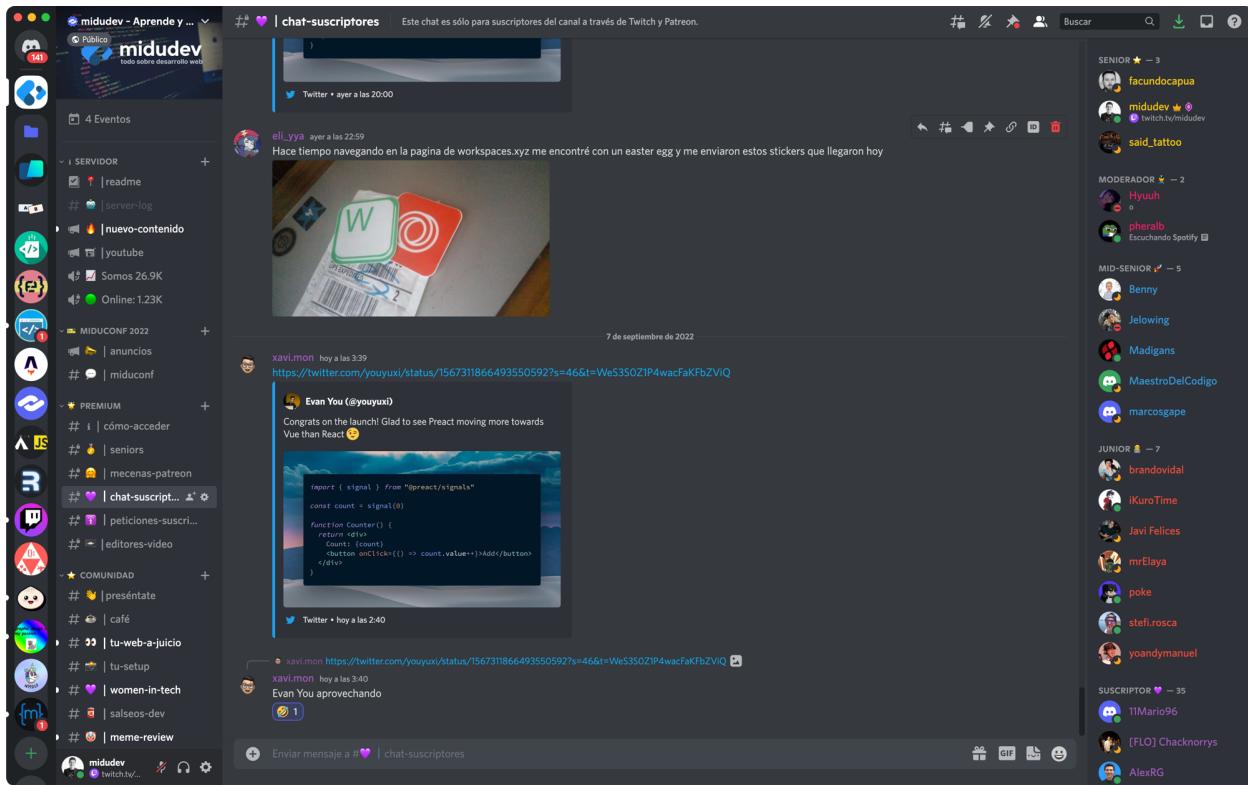
⁵<https://twitter.com/midudev>

⁶<https://www.instagram.com/midu.dev/>

⁷<https://www.tiktok.com/@midudev>

⁸<https://midu.dev/>

⁹<https://discord.gg/midudev>



En la comunidad de Discord encontrarás gente con tus mismos interés dispuestas a ayudarte y compartir conocimiento. ¡No te la pierdas!

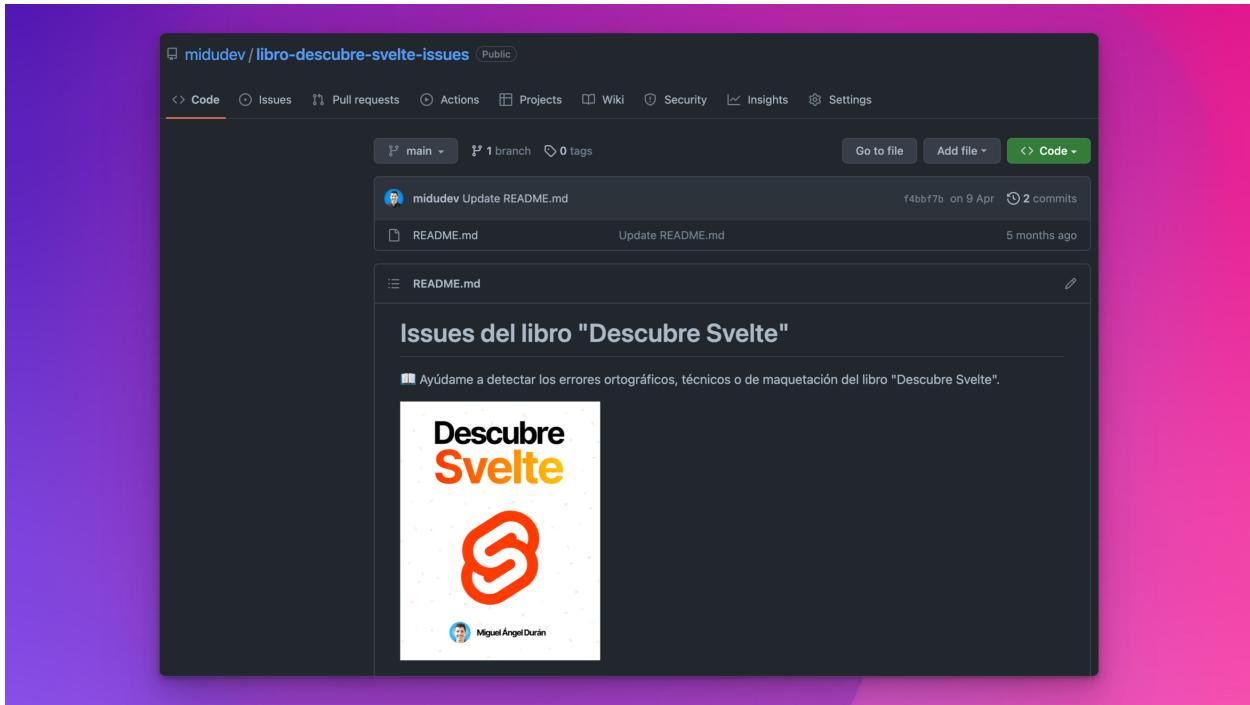
Reportando erratas y sugerencias

Creo que los libros los termina de escribir quien los lee. Por eso, he habilitado un repositorio de GitHub donde puedes abrir *issues* para avisar de errores que existan en el libro, ya sean ortográficos, técnicos o de maquetación. Intentaré solucionarlos lo antes posible y subir una versión actualizada con tus observaciones y correcciones.

Antes de reportar un problema, te recomiendo que revises la edición a la que te refieres. En la segunda página del libro encontrarás cuándo se publicó la versión. Por favor, usa esa referencia a la hora de crear la *issue* para que pueda revisar más fácilmente a qué versión te refieres.

Tienes el repositorio disponible¹⁰ para colaborar. ¡Cuento contigo para crear el mejor libro posible! ¡Gracias!

¹⁰<https://github.com/midudev/libro-descubre-svelte-issues>



En la sección de Issues del repositorio encontrarás una forma de reportar las erratas y problemas que te encuentres en el libro. ¡Ya hay gente que lo ha hecho!

Requerimientos

En el libro se da por sentado que sabes HTML, CSS y JavaScript. No hace falta que sea a nivel experto, pero desde luego un uso común es aconsejable.

Para muchos ejemplos se proporciona un enlace al [Svelte REPL](#), un editor de código que ejecuta tu código de Svelte¹¹ y te muestra el resultado y el output de los ficheros.

En capítulos posteriores se crea un proyecto desde cero y para ello se requiere tener instalado [Node.js +16.x](#) y [npm¹²](#).

Agradecimientos

Gracias a **Samar Jaffal**, que fue la primera en leerse el libro y corregir un montón de erratas que había cometido. Agradecimientos a **Mariano Álvarez** y **Ciprian Sauliuc** por sus correcciones.

¹¹<https://svelte.dev/repl>

¹²<https://nodejs.org/en/>

Introducción a Svelte

Cuando descubrí Svelte

En 2019 asistí a la conferencia JSCamp de Barcelona. Entre las charlas que vi, una me llamó poderosamente la atención. “[The Return of ‘Write Less, Do More’](#)¹³” de Rich Harris.

En ella, hablaba de cómo nos preocupamos de muchas métricas, como el tamaño de nuestro *bundle*, pero no le prestamos atención al número de líneas de código que escribimos porque a mayor número de líneas más potenciales bugs tendremos. ¿Y cómo lo solucionamos? Pues Rich Harris presentaba una solución en forma de framework: **Svelte**.



Rich Harris en la JSCamp de Barcelona explicando qué es Svelte

Hoy, años más tarde de aquella charla en la conferencia, *Svelte* se ha consolidado como uno de los frameworks más amados por la comunidad gracias a su experiencia de desarrollo y para el usuario.

¹³<https://svelte.dev/blog/write-less-code>

¿Qué es Svelte?

Svelte es un framework de frontend. Tiene conceptos parecidos a React, Vue y Angular pero, a la vez, viene con diferencias muy importantes.

Concretamente hay una diferencia importantísima y es que **Svelte tiene un paso previo de compilación**. ¿Por qué? Porque el código que escribimos con Svelte, aunque está basado en lenguajes que ya conoces como HTML, CSS y Javascript, es un código que no entendería el navegador sin tratar y tiene que ser compilado previamente.

Aunque muchos pueden ver esto como una desventaja, en realidad, es algo con lo que ya convivimos de alguna forma con el resto de frameworks por el uso JSX o para usar características de JavaScript que todavía no soportan los navegadores (Babel, SWC...). Aunque, de alguna forma, el resto de bibliotecas y frameworks sí podrían obviar este paso, **con Svelte es completamente necesario**.

Adiós Virtual DOM

No sólo la necesidad de compilar es la diferencia. También la total ausencia de Virtual DOM, o VDOM. El VDOM, que popularizó React en su día, guarda una foto del árbol de elementos del DOM en memoria de forma que, al tener que hacer actualizaciones, sabe los cambios mínimos a realizar. Esto lo hace porque manipular el DOM es lento y, de esta forma, se evita trabajo innecesario.



Pete Hunt en la JSConf 2013 hablando por qué decidieron añadir el Virtual DOM en React. El VDOM simplemente evita hacer trabajo innecesario en el DOM.

Entonces, **si es algo bueno, ¿por qué Svelte no lo usa?** Porque no es gratis. El *diffing* (cálculo de diferencias) que realiza el VDOM tiene su coste, y peor será cuanto más compleja sea nuestra app. ¿Eso quiere decir que Svelte es más lento que React y renderiza elementos de forma innecesaria? Nada de eso.

Como Svelte es un compilador, lo que hace, es aprovecharse del paso de compilación, para **envolver los cambios de estado y propiedades en métodos que, de forma quirúrgica, podrán actualizar el DOM**. Y eso, sin necesidad de un Virtual DOM.

Hola Mundo en Svelte

Svelte utiliza el principio de *Single File Component*. Esto es, que en un mismo archivo, tienes un componente con su marcado (HTML), su funcionalidad (JavaScript) y su estilo (CSS). Esto lo han popularizado frameworks como Angular y Vue pero en Svelte, como verás, es todavía más sencillo.

Entonces, **¿cómo escribiríamos un Hola Mundo con este framework?**

```

1 <script>
2   let name = "World"
3 </script>
4
5 <h1>Hello {name}!</h1>
```

Con esto ya podemos ver las primeras **diferencias y similitudes con otras alternativas como React**. De hecho, vamos a añadir un Hola Mundo, lo más parecido posible, en React para que puedas comparar con el código anterior de Svelte:

```

1 export default () => {
2   let name = "World"
3   return <h1>Hello {name}!</h1>
4 }
```

Diferencias:

- Svelte *parece* simplemente HTML, mientras que React es más JavaScript.
- Svelte separa el comportamiento y el renderizado en dos ámbitos claramente diferenciados. En React, el comportamiento y el renderizado están mezclados en un mismo ámbito.
- En React indicamos que exportamos un componente, mientras que en Svelte no es necesario.

Similitudes:

- La evaluación de expresiones en Svelte y React es igual. Se usan llaves para evaluar la variable y así poder renderizar el valor.

- En ambos *parece* que se renderiza HTML (en React en realidad es JSX y Svelte lo compilará internamente)

Svelte es tan parecido a HTML plano que puedes usar el coloreador de sintaxis de HTML en tu editor para trabajar con sus archivos.

Añadiendo estilos CSS

Ahora mismo nuestro ejemplo es un poco soso en cuanto a estilos, así que añadimos CSS para dejarlo más presentable. Lo haremos añadiendo una etiqueta `<style>`, indicando dentro los elementos que queremos estilar.

```
1 <script>
2   let name = 'World'
3 </script>
4
5 <h1>Hello {name}!</h1>
6
7 <style>
8   h1 {
9     color: #09f;
10    font-weight: 100;
11    text-transform: uppercase;
12  }
13 </style>
```

Playground con código¹⁴

Pero, espera... ¿estamos estilando directamente con elementos HTML? ¿Y qué pasa con la colisión de los estilos cuando nuestra aplicación crezca? Pues no tienes que preocuparte.

El ámbito de estilos de Svelte es **local** por defecto. Esto quiere decir que **los estilos que indicamos aquí no tendrán efecto en el resto de la aplicación**. De esta forma no tienes que pensar en clases o selectores y puedes simplificar mucho trabajar con ellos.

Por supuesto, **si lo necesitas, puedes seguir trabajando con clases**. Aunque lo ideal es que lo evites lo máximo posible para mantener tu código lo más limpio posible.

¹⁴<https://svelte.dev/repl/178233e02ea342398d076ef08e02dd05?version=3.50.0>

¿Y qué pasa con el estado?

Aquí es donde viene gran parte de la magia de Svelte. Una de las luchas de su creador es, justamente, acabar con el boilerplate y, si lo piensas fríamente, por poco boilerplate que tengan nuestros componentes de React o Vue, algo tienen.

El boilerplate hace referencia al código que se repite continuamente para poder inicializar o lograr algo en programación. Cuanto más boilerplate exista, más tedioso será el desarrollo.

Un boilerplate común en React es la creación de un estado:

1. Tienes que importar el hook `useState`.
2. Debes inicializar el estado y extraer la función para actualizarlo.
3. Llamas la función con el valor nuevo.

¿Y si pudiéramos crear variables y la propia librería pudiese saber si es el estado del componente o una simple variable? Como Svelte es un compilador, es capaz de determinar esto en tiempo de compilado. Así podemos escribir el mínimo código posible:

```
1 <script>
2   let name = 'World'
3   // a los 3 segundos, queremos que name pase a ser "Frontender"
4   setTimeout(function () {
5     name = 'Frontender'
6   }, 3000)
7 </script>
8
9 <h1>Hello {name}!</h1>
10
11 <style>
12   h1 {
13     font-weight: 100;
14     text-transform: uppercase;
15   }
16 </style>
```

Playground de código¹⁵

¹⁵<https://svelte.dev/repl/eb0ac6bf4a164e3893d8d371a1d4bb3d?version=3.50.0>

Lo que estamos haciendo es que, a los tres segundos, queremos cambiar el valor de `name` de forma que pase de `World` a `Frontender`. ¿Lo hará? Sí. ¿Pero cómo? Pues Svelte, al hacer la compilación, determinará que `name` es un estado y que, al reasignarle un valor, debe hacer un re-renderizado de nuestro componente.

Compilación de código

Para que entiendas el potencial de *Svelte* debes tener claro que cuando hablamos que tiene un paso de compilación es algo que va **mucho más allá de las transformaciones que se suelen hacer con Babel, SWC o similares**.

En el ejemplo anterior, que teníamos un estado con una simple variable, si revisamos el código que ha compilado veríamos algo así:

```
1  function instance($$self, $$props, $$invalidate) {
2    let name = 'World';
3
4    // a los 3 segundos, queremos que name pase a ser "Frontender"
5    setTimeout(
6      function () {
7        $$invalidate(0, name = 'Frontender');
8      },
9      3000
10    );
11
12    return [name];
13 }
```

Como ves, Svelte ha detectado gracias a la compilación que `name` no es una variable corriente y que la queremos usar como un estado. Así que llamará a la función `invalidate` que se encargará de actualizar el estado y cambiar el valor en el DOM por el nuevo.

El paso de compilación no sólo es útil para esto. También nos puede ayudar a encontrar errores en el código... ¡incluso a evitar código que no usamos! Tómalo como una especie de linter.

Por ejemplo, mira este código de Svelte:

```
1 <h2> ¡Hola, Svelte! </h2>
2
3 <style>
4   h1 {
5     color: #09f;
6   }
7 </style>
```

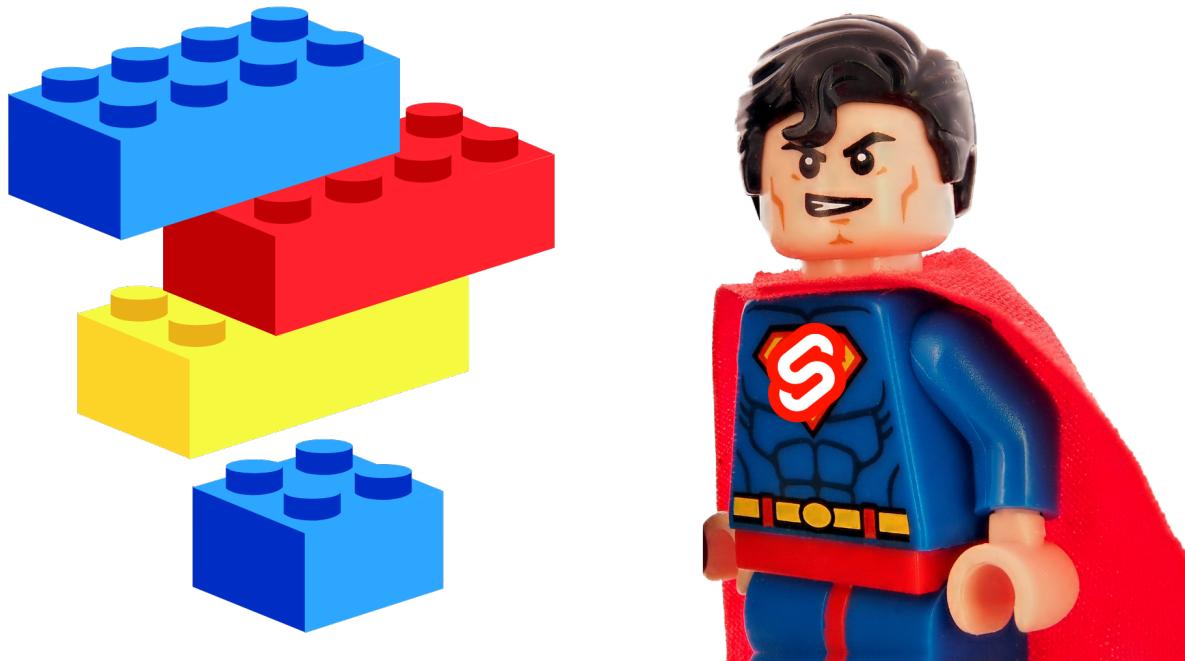
¿Notas algo raro? Sí, estamos intentando estilar el elemento `h1` pero en realidad estamos usando un `h2` en la presentación. Gracias a que Svelte es un compilador, nos indicará una advertencia:

```
1 Unused CSS selector "h1" (4:1)
```

Imagina las posibilidades de esto. **Gracias a esto no volverás a tener estilos CSS en tu aplicación que queden huérfanos de uso.**

Componentes

Hoy en día, todos los frameworks del frontend están basados en la componetización de la interfaz. Esto significa que la UI se divide en pequeños componentes parametrizables que se pueden reutilizar y componer para crear la interfaz final.



Que tu UI sea una colección de piezas de Lego es el super poder de frameworks como Svelte, React o Vue

Svelte no iba a ser menos. En este capítulo vamos a ver cómo se crean componentes en Svelte y cómo se usan.

Creando tu primer componente

Nuestro primer componente va a consistir en un botón que, más adelante, personalizaremos el contenido del mismo y, también, le añadiremos eventos.

```
1 <!-- src/components/Button.svelte -->
2 <button>Hello, button!</button>
3
4 <style>button { color: #09f }</style>
```

Con esto ya tendríamos nuestro primer componente. Como puedes ver el único nombre que debes darle es el del propio archivo. En este caso, `Button.svelte`.

Nota que los componentes en Svelte tienen la extensión `.svelte`. Esto no sólo es importante de cara a la sintaxis, sino que también es necesario para que Svelte pueda procesar los componentes correctamente.

Usando componentes

Para usar un componente, simplemente debemos importarlo y usarlo como si fuera un archivo de JavaScript.

```
1 <!-- src/App.svelte -->
2 <script>
3   import Button from './components/Button.svelte'
4 </script>
5
6 <Button/>
```

¿Te suena? Parece JSX, ¿verdad? Pero no es JSX... ¡es Svelte!

Primero, nota que podemos importar un componente pese a que no habíamos hecho un `export default` en el mismo. Esto es porque Svelte añade un `export default` por defecto a todos los componentes.

Por otro lado, el nombre que le ponemos al componente no importa. En este caso, lo hemos importado como `Button` pero podríamos usar el nombre que queramos:

```
1 <!-- src/App.svelte -->
2 <script>
3   import MyButton from './components/Button.svelte'
4 </script>
5
6 <MyButton/>
```

Aunque puedes hacerlo y lo explico para que lo sepas, lo ideal sería que siempre uses el mismo nombre del fichero para evitar tener diferentes nombres para el mismo componente a lo largo de tu proyecto.

Props en Svelte

Para que nuestros componentes realmente sean reutilizables, necesitamos poder cambiar su comportamiento desde fuera. Esto lo logramos con las *props*, un concepto que existe en otros frameworks y bibliotecas como React o Vue.

Imaginemos que a nuestro componente `Button` le queremos pasar un texto que se muestre en el botón. Para ello, podemos usar las props:

```
1 <!-- src/App.svelte -->
2 <script>
3   import Button from './components/Button.svelte'
4 </script>
5
6 <Button text="Click me!" />
```

Esta sería la forma de pasarle una prop `text` al botón para cambiar lo que renderiza. Pero... ¿Cómo recuperamos esta información dentro del componente? Para ello, Svelte nos ofrece una sintaxis especial para recuperar las props:

```
1 <!-- src/components/Button.svelte -->
2 <script>
3   export let text // Recuperamos la prop text
4 </script>
5
6 <button>{text}</button> <!-- Evaluamos la expresión -->
7
8 <style>button { color: #009f }</style>
```

Como puedes ver, Svelte nos ofrece una sintaxis especial para recuperar las props. En este caso, hemos usado `export let text` para recuperar la prop `text` que nos han pasado.

Después usamos las `{}` para evaluar la expresión y mostrar el texto que nos han pasado. Esto es muy similar a lo que hacemos en Vue o React.

Por supuesto también podríamos indicar que tiene un valor por defecto inicial, de esta forma, en el caso que no nos pasen una prop, nuestro componente podrá renderizar un valor por defecto:

```

1 <!-- src/components/Button.svelte -->
2 <script>
3   export let text = 'Hello, button!'
4 </script>
5
6 <button>{text}</button>

```

Ahora ya podemos usar nuestro componente tanto si le pasamos una prop como si no:

```

1 <!-- src/App.svelte -->
2 <script>
3   import Button from './components/Button.svelte'
4 </script>
5
6 <Button />
7 <Button text="Clic aquí" />
8 <Button text="Otro clic aquí" />

```

Playground con el código¹⁶

Spread de props en Svelte

A veces, nos encontramos con componentes que reciben muchas props. En estos casos, puede ser tedioso tener que pasar cada una de ellas. Por suerte, Svelte nos ofrece una sintaxis especial para pasar todas las props que nos pasen a un componente a otro componente.

Para ello podemos pasarle un objeto completo y cada propiedad se pasará como prop si usamos el spread operator ...

```

1 <!-- src/App.svelte -->
2 <script>
3   import UserInfo from './UserInfo.svelte';
4
5   const user = {
6     name: 'midudev',
7     twitch: 'https://twitch.tv/midudev',
8     age: 37,
9     website: 'https://midu.dev'
10    };
11 </script>
12
13 <UserInfo {...user} />

```

¹⁶<https://svelte.dev/repl/b64c8fd528784ea588b7d2f49d02d00c?version=3.50.0>

Cada propiedad del objeto `user` le llegará como prop por separado al componente y podremos acceder a ellas con `export let`:

```
1 <!-- src/UserInfo.svelte -->
2 <script>
3   export let name;
4   export let twitch;
5   export let age;
6   export let website;
7 </script>
8
9 <p>El usuario es {name}, su Twitch es {twitch}.
10 Tiene {age} años y puedes ir a su web en {website}</p>
```

Un comentario sobre la sintaxis

Es posible que pienses que la sintaxis de Svelte para las props no es natural. Aunque es verdad que el uso de `export` en este caso no es el que podríamos esperar, lo cierto es que Svelte ha elegido esta sintaxis para incorporar el mínimo número de sintaxis nueva posible.

Así la curva de aprendizaje es menor, la sintaxis es más familiar y el código es más legible. Una vez que te acostumbras, verás que no cuesta nada pensar que cada vez que exportas una variable desde un componente de Svelte, estás creando una prop.

Eventos

La UI de nuestra aplicación no sería nada sin la interacción del usuario. Ahora que ya sabemos cómo crear componentes, vamos a darles un poco de vida.

Eventos del DOM

En Svelte puedes definir eventos del DOM directamente en el elemento HTML. Para ello usamos la sintaxis `on:<evento>`. Donde evento sería, por ejemplo, `click` o `mousemove` o cualquiera de la inmensa [lista de eventos disponibles¹⁷](#).

A esta sintaxis, y otras que veremos más adelante, se le llaman **directivas**. Las directivas son una forma de extender la sintaxis del HTML que escribimos en Svelte para que podamos añadir funcionalidades extra.

Por ejemplo, si queremos que un botón muestre una alerta cuando el usuario hace clic haremos lo siguiente:

```
1 <!-- src/App.svelte -->
2 <script>
3   function showAlert() {
4     alert(' iAy ay ay! ')
5   }
6 </script>
7
8 <button on:click={showAlert}> iAlerta! </button>
```

[Playground con el código¹⁸](#)

Podemos notar unas cuantas cosas interesantes:

- Estamos escuchando el evento de click del elemento `<button>` añadiendo la directiva `on:click`.
- En el evento `on:click` hemos pasado una función. La evaluamos usando las llaves y le pasamos la referencia de la función.
- Hemos creado la función dentro de `<script>` y la podemos usar en cualquier parte del componente.

¹⁷<https://developer.mozilla.org/es/docs/Web/Events>

¹⁸<https://svelte.dev/repl/fde9966408364dd1806caf35690432b2?version=3.50.0>

Eventos en línea

Aunque en nuestro ejemplo hemos pasado una función como valor de la directiva `on:click`, también podemos pasar una expresión de JavaScript directamente. Por ejemplo, podemos pasar una función anónima:

```
1 <!-- src/App.svelte -->
2 <button on:click={() => alert(' iAy ay ay! ')>} iAlerta!</button>
```

Mi recomendación es que lo evites siempre que puedas, especialmente cuando el *callback* que le quieras pasar al elemento es algo complejo. Es mucho más fácil de leer y mantener el código si lo separas en una función.

Modificadores de eventos

Los modificadores de eventos son una forma de extender la funcionalidad de los eventos del DOM. Por ejemplo, podemos añadir el modificador `preventDefault` para evitar que el evento haga lo que hace por defecto. Por ejemplo, si tenemos un enlace que nos lleva a otra página, podemos evitar que se ejecute el evento de click y que se navegue a la página:

```
1 <!-- src/App.svelte -->
2 <a href="https://svelte.dev" on:click|preventDefault>Ir a Svelte</a>
```

También puedes hacer que el evento se ejecute una sola vez usando el modificador `once`:

```
1 <!-- src/App.svelte -->
2 <script>
3   function showAlert() {
4     alert(' iAy ay ay! ')
5   }
6 </script>
7
8 <button on:click|once={showAlert}>iAlerta sólo una vez!</button>
```

Playground con el código¹⁹

La lista completa es:

- `preventDefault`: Ejecuta `event.preventDefault()` antes de ejecutar el callback (si es que lo hay).

¹⁹<https://svelte.dev/repl/3d73268b8b124df1b3c525bad17f72f0?version=3.50.0>

- `stopPropagation`: Ejecuta `event.stopPropagation()` antes de ejecutar el callback (si es que lo hay).
- `passive`: Añade `{passive: true}` a las opciones del listener para mejorar el rendimiento de los eventos de scroll o touchmove.
- `nonpassive`: Añade `{passive: false}`.
- `capture`: Ejecuta el listener en la fase de captura en lugar de la fase de burbuja.
- `once`: Elimina el listener después de que se ejecute por primera vez.
- `self`: Ejecuta el callback sólo en el caso que el `event.target` sea el propio elemento. Útil para botones con svg, para evitar escuchar el evento del ícono en lugar del botón.
- `trusted`: Para ejecutar el callback sólo en el caso que el evento sea de confianza. Útil para evitar que se ejecute el callback cuando el evento se ha generado programáticamente.

Ten en cuenta, además, que los modificadores de eventos se pueden encadenar. Por ejemplo: `on:click|once|preventDefault`.

Reenvío de eventos del DOM

A veces tienes un componente, por ejemplo un botón, que no quieras que maneje el evento él mismo pero sí te gustaría que lo hiciera un componente padre. Para hacerlo, debes reenviar los eventos ya que, por defecto, los eventos no se propagan (como sí hacen los eventos nativos del DOM).

Por ejemplo, imagina que tienes esta aplicación:

```
1 <!-- src/App.svelte -->
2 <script>
3   import CustomButton from './CustomButton.svelte'
4 </script>
5
6 <CustomButton on:click={() => alert('iAy ay ay!')} />
```

Como puedes ver, estamos usando un componente `<CustomButton>`. Si ejecutamos la aplicación y hacemos clic en el botón, no veremos nada. Esto es porque el evento `click` no se está propagando. Para hacerlo, debemos reenviar el evento de esta forma.

```
1 <!-- src/CustomButton.svelte -->
2 <button on:click>iAlerta!</button>
```

Sólo tiene que añadir la directiva `on:click` al elemento que quieras que reenvíe el evento (sin pasarle ningún `callback`). En este caso, hemos añadido la directiva al elemento `<button>`. De esta forma el componente padre podrá escuchar el evento `click`.

Reactividad y Estado

Ya hemos visto cómo podemos pasar props a nuestros componentes y cómo manejar eventos.

Para que nuestra aplicación realmente sea usable, debemos poder cambiar la interfaz para responder a las interacciones del usuario. Para ello, necesitamos **cambiar el estado de nuestra aplicación y que se reflejen los cambios en la UI**.

Reactividad en Svelte

En Svelte, para declarar un estado sólo tenemos que declarar una variable usando `let` y asignarle el valor inicial que tiene. Más adelante, podremos cambiar el valor de esta variable y Svelte se encargará de actualizar la UI para reflejar los cambios.

La reactividad se basa en el concepto de **dependencias**. Cuando una variable cambia, Svelte detecta qué partes de la UI dependen de esa variable y actualiza sólo esas partes.

Para empezar con un ejemplo familiar, vamos a crear un contador con todos los conceptos que hemos visto hasta el momento y añadiendo un `count` que será el estado de nuestra aplicación:

```
1 <script>
2   let count = 0
3   const increment = () => count++
4 </script>
5
6 <p>{count}</p>
7 <button on:click={increment}>Incrementar</button>
```

[Playground con el código²⁰](#)

Como puedes ver, no estamos haciendo nada especial para que Svelte sepa que `count` es un estado. Simplemente hemos declarado una variable con `let` y le hemos asignado un valor inicial.

Después, hemos declarado una función que incrementa el valor de `count` y la hemos asignado al evento `click` del botón.

Y cada vez que hacemos clic, el contador se incrementa y la interfaz refleja este cambio.

²⁰<https://svelte.dev/repl/0c04a2b787d44ffaa96ce22f9cf66d15?version=3.50.0>

Valores derivados reactivos

En el ejemplo anterior, hemos visto que Svelte detecta automáticamente las dependencias de una variable y actualiza la UI cuando cambia. Pero, ¿qué pasa si queremos que una variable dependa de otra que es reactiva?

Por ejemplo, imagina que queremos indicar un mensaje que nos diga si el nuevo valor del contador es par o impar. Podríamos pensar que podríamos hacer algo como esto:

```

1 <script>
2   let count = 0
3   let message = count % 2 === 0 ? 'Es par' : 'Es impar'
4
5   const increment = () => count++
6 </script>
7
8 <p>{count}</p>
9 <span>{message}</span>
10 <button on:click={increment}>Incrementar</button>
```

[Playground con el código²¹](#)

Si lo pruebas verás que... ¡siempre aparece que es par!

Como counter se inicia a 0, al principio el mensaje es Es par pero, si hacemos click en el botón, pese a que el counter se incrementa en 1, el mensaje todavía es Es par, lo que significa que **no está mostrando el mensaje correcto**. ¿Por qué?

Lo que está ocurriendo es que la variable message sólo se está evaluando una vez. A diferencia de la variable count que *Svelte* sí está infiriendo que es un estado. Esto es un error muy común a la hora de trabajar con Svelte, ya que uno podría esperar que la variable message fuese a re-asignarse de forma automática cuando no es así.

Ahora que ya lo sabemos. **¿Cómo podemos arreglarlo?** Tenemos dos formas. La primera, **mover la evaluación al render**, en lugar de hacerlo dentro de las etiquetas `<script>`. Así, esto siempre se evaluará en cada renderizado y mostrará la información correcta.

²¹<https://svelte.dev/repl/4895295908294d4cae53b3e981e36292?version=3.50.0>

```

1 <script>
2   let count = 0
3
4   const increment = () => count++
5 </script>
6
7 <button on:click={increment}>Incrementar</button>
8 <span>{count}</span>
9 <span>{count % 2 === 0 ? 'Es par' : 'Es impar'}</span>

```

Playground con el código²²

Gracias a que podemos evaluar cualquier expresión entre llaves, podemos hacer que el mensaje se muestre correctamente, ya que Svelte detecta que count ha cambiado y la nueva expresión tendrá el valor correcto.

Esta opción es bastante sencilla, pero *Svelte* ofrece otra forma que nos desbloqueará un montón de posibilidades y es utilizar el símbolo especial \$ para indicar que la declaración es reactiva:

```

1 <script>
2   let count = 0;
3   let message;
4
5   $: message = count % 2 === 0 ? 'Es par' : 'Es impar'
6
7   const increment = () => count++
8 </script>
9
10 <button on:click={increment}>Incrementar</button>
11 <span>{count}</span>
12 <span>{message}</span>

```

Playground con el código²³

Con esto, si usamos el botón Incrementar, veremos que muestra el mensaje correcto. Lo que hace el símbolo \$ es avisar a *Svelte* que esta sentencia es reactiva y el framework detectará dentro de esa declaración qué variables se están usando (en este caso count) de forma que, cada vez que se actualice ese valor, pasará a ejecutar de nuevo la sentencia.

También es importante indicar que la declaración reactiva se ejecutará también nada más montar el componente, por eso vemos desde el principio un mensaje con la información correcta (ya que count es 0).

²²<https://svelte.dev/repl/296ae3599e184a078b98a3fe6c0e7c39?version=3.50.0>

²³<https://svelte.dev/repl/c9f455e42cb04a5e9fbb913e1e5c3283?version=3.50.0>

Además, podemos añadir tantas declaraciones reactivas como queramos. Por ejemplo, vamos a añadir una que, lo que va a hacer, es no permitir que el counter pase de un valor máximo y, para ello, vamos a ver cómo conseguirlo añadiendo una declaración multilínea:

```

1 <script>
2   let count = 0
3   let message
4
5   $: message = count % 2 === 0 ? 'Es par' : 'Es impar'
6
7   // Esta declaración reactiva
8   // no permite que el contador pase de 10
9   $: {
10     if (count > 9) {
11       count = 9 // reasigna a 9
12     }
13   }
14
15   const increment = () => count++
16 </script>
17
18 <button on:click={increment}>Incrementar</button>
19 <span>{count}</span>
20 <span>{message}</span>
```

Playground con el código²⁴

Las declaraciones reactivas pueden tener **cláusulas de entrada** por lo que podemos escribirlo de una manera todavía más limpia para conseguir el mismo resultado.

```

1   $: if (counter > 9) {
2     counter = 9
3   }
```

De esta forma, esta declaración reactiva sólo se ejecutará cuando el valor del counter sea mayor a nueve. Al ejecutarse, la declaración hará que el estado local counter no pueda nunca sobrepasar el valor de 9.

Las declaraciones reactivas son... similares a los efectos de React. La diferencia es que Svelte es capaz de detectar automáticamente las dependencias del efecto y, por tanto, no es necesario indicarlas

²⁴<https://svelte.dev/repl/668c989285914df4a627e66477821072?version=3.50.0>

explícitamente.

Formularios

Los formularios son una de las herramientas más importantes de cualquier sitio web ya que permiten que el usuario pueda introducir información. En este capítulo aprenderemos a crear formularios en Svelte, gestionar el estado de los campos y tratar sus eventos.

Trabajando con formularios

Con todo lo que sabemos hasta el momento, podemos crear un formulario muy sencillo. Vamos a crear un componente llamado `Input` que tendrá una caja de texto donde podremos añadir un texto a buscar.

Para poder gestionar el estado creamos una variable `value` que inicializamos a '' y que será la que guardará el valor de la caja de texto.

Para poder actualizar el estado cada vez que el usuario escriba en la caja de texto, vamos a escuchar el evento `input` gracias a la directiva `on:input` y ejecutaremos un método `handleInput` que actualiza el estado:

```
1 <script>
2   // donde guardamos el valor de la caja de texto
3   let value = ''
4   // escuchamos el evento `input` y ejecutamos
5   // este método para actualizar el estado
6   const handleInput = (event) => {
7     value = event.target.value
8   }
9 </script>
10
11 <input type="text" on:input={handleInput} />
12 <span>El valor es: {value}</span>
```

Playground de código²⁵

Aunque este código funciona, lo cierto es que hay una forma mucho más sencilla de lograrlo sin la necesidad de crear funciones.

²⁵<https://svelte.dev/repl/c3cc605f41b14ec5b714e199535747e7?version=3.50.0>

Enlazando el estado con el valor de la caja de texto

Como ya habrás notado, Svelte siempre intenta tener una forma que evite el boilerplate. El caso anterior es un buen ejemplo. En lugar de crear una función para actualizar el estado, podemos enlazar el estado con el valor de la caja de texto. Para ello, utilizaremos la directiva `bind:value`.

```

1 <script>
2   let value = ''
3 </script>
4
5 <input type="text" bind:value={value} />
6 <span>El valor es: {value}</span>
```

Playground de código²⁶

Ya ves lo mucho que se ha simplificado el código. Hemos quitado la función para actualizar el estado y ya no tenemos que escuchar el evento `input` para actualizar el estado. De esta forma, Svelte se encarga de actualizar el estado cada vez que el usuario escriba en la caja de texto.

Esto también funciona para otros tipos de inputs, como puedes ver en este ejemplo:

```

1 <script>
2   let number = 0
3   let range = 5
4 </script>
5
6 <input type="number" bind:value={number} max="99">
7 <input type="range" bind:value={range} min="0" max="10">
8 <p>{number + range}</p>
```

Playground de código²⁷

En el caso de los inputs de tipo `checkbox`, el valor del estado será un `boolean` que indica si el `checkbox` está seleccionado o no y deberás usar una directiva distinta para poder recuperar el valor.

²⁶<https://svelte.dev/repl/99c39789227a4e23b9b107c0fd9e9bd8?version=3.50.0>

²⁷<https://svelte.dev/repl/aac37f3245104493812e8e171c95f207?version=3.50.0>

```
1 <script>
2   let enabled = false
3 </script>
4
5 <label>
6   <input type="checkbox" bind:checked={enabled}>
7   ¿Activado?
8 </label>
```

Esto tiene sentido ya que un checkbox es seleccionado o no, no leemos el even.target.value como hacíamos con los inputs de tipo text. Leemos el valor de la propiedad checked del elemento. Por eso la directiva es otra.

Trabajando con un grupo de inputs

A veces quieras trabajar con un grupo de inputs del tipo checkbox o radio que se refieren a una misma propiedad. En este caso, Svelte nos permite enlazar el estado con un grupo de inputs y enlazarlo con un estado utilizando la directiva bind:group.

```
1 <script>
2   let selectedLanguage = ''
3 </script>
4
5 <label>
6   <input type="radio" bind:group={selectedLanguage} name="languages" value="JavaScri\
7 pt">
8   JavaScript
9 </label>
10
11 <label>
12   <input type="radio" bind:group={selectedLanguage} name="languages" value="TypeScri\
13 pt">
14   TypeScript
15 </label>
16
17 <label>
18   <input type="radio" bind:group={selectedLanguage} name="languages" value="Python">
19   Python
20 </label>
```

```

21
22 <p>
23   Selected: {selectedLanguage}
24 </p>
```

Playground de código²⁸

Accede al playground de código y prueba a cambiar el tipo de los inputs de radio a checkbox para ver qué ocurre.

Enviando formularios

Ya hemos visto cómo podemos trabajar con inputs y cómo podemos enlazar el estado con sus valores sin necesidad de escribir código adicional. Ahora vamos a ver cómo podemos hacer *submit* de un formulario.

Para ello, vamos a crear una pequeña aplicación que nos va a permitir añadir tareas a una lista. Para ello, vamos a crear un formulario con un input de tipo text y un botón de tipo submit.

```

1 <script>
2   let task
3   let tasks = []
4
5   const handleSubmit = () => {
6     tasks = [...tasks, task]
7   }
8 </script>
9
10 <form on:submit|preventDefault={handleSubmit}>
11   <input bind:value={task} />
12   <button type="submit">Añadir</button>
13 </form>
14
15 <p>Tareas pendientes: {tasks.length}</p>
```

Playground de código²⁹

Este pequeño código trae unas cuantas cosas interesantes de Svelte. ¡Vamos a repasarlas!

²⁸<https://svelte.dev/repl/f6e6b5b66c3a4f5c93f6d49d958cbc7f?version=3.50.0>

²⁹<https://svelte.dev/repl/a83897faa7c5414d982f7fd4beb80543?version=3.50.0>

Hemos añadido el modificador de evento `preventDefault` a la directiva `on:submit` para evitar que el formulario se envíe y se recargue la página (es su comportamiento por defecto).

También hemos añadido un botón de tipo `submit` para que el formulario se envíe cuando el usuario pulse el botón. Fíjate que no hace falta que añadamos el `on:click` ya que al ser del tipo `submit` este botón envía el formulario automáticamente.

Además, para actualizar el estado de la lista de tareas `task` estamos reasignando el valor de la lista. Recuerda que necesitamos hacer siempre una reasignación y trabajar con una copia del estado para que Svelte detecte los cambios y actualice la vista.

En el próximo capítulo vamos a ver cómo puedes mostrar la lista de tareas, no te preocupes.

Lógica en tus componentes

Svelte nos ofrece etiquetas especiales para que puedas realizar diferente lógica dentro del renderizado de tus componentes. De esta forma podrás renderizar diferentes elementos de forma condicional, iterar una lista de elementos o incluso realizar operaciones asíncronas y esperar los resultados.

Renderizado condicional

Un renderizado condicional es, simplemente, **renderizar diferentes elementos o componentes dependiendo de ciertas condiciones**. Dicho de otra forma: trasladar la funcionalidad de `if` al renderizado de nuestros componentes. Esto es muy típico en librerías como *React* o *Vue* y, por supuesto, Svelte no iba a ser una excepción.

La sintaxis del renderizado condicional en Svelte

Como ya vimos en un capítulo anterior, la forma más sencilla de lograr renderizar algo diferente dependiendo de una condición es **evaluando una ternaria** dentro de unas llaves.

```
1 <script>
2   let count = 2
3   const handleClick = () => count++
4 </script>
5
6 <button on:click={handleClick}>Incrementar</button>
7 <span>{count % 2 === 0 ? 'Es par' : 'Es impar'}</span>
```

En ese caso, estaríamos cambiando el mensaje que se renderiza dependiendo de si `count` es par o impar. Pero, **imaginemos que no sólo queremos cambiar el mensaje, si no que además queremos cambiar lo que renderiza**. Podríamos probar a hacer lo siguiente:

```
1 <script>
2   // ESTE CÓDIGO ES INCORRECTO
3   let count = 2
4   const handleClick = () => count++
5 </script>
6
7 <button on:click={handleClick}>Incrementar</button>
8 {count % 2 === 0
9   ? <strong>Is Even</strong>
10  : <small>Is Odd</small>
11 }
```

El código anterior no es correcto. Svelte nos dice que existen tokens que no esperaba al intentar compilarlo y es que esta no es la forma correcta de conseguir un renderizado condicional.

Ten cuidado porque en React y Vue sí que se podría realizar de esta forma. Pero ya ves que a Svelte no le gusta.

¿Cómo podemos lograr algo similar? Necesitamos usar la etiqueta `#if` y `#else` para lograrlo. La sintaxis es la siguiente:

```
1 <script>
2   let count = 0
3   const handleClick = () => count++
4 </script>
5
6 <button on:click={handleClick}>Incrementar</button>
7
8 {#if count % 2 === 0}
9   <strong> iEs par!</strong>
10 {/if}
```

También podemos controlar la condición contraria. Para ello, vamos a añadir a la plantilla la condición `:else` y dentro lo que queremos renderizar en el caso que el contador no sea par:

```

1 <script>
2   let count = 0
3   const handleClick = () => count++
4 </script>
5
6 <button on:click={handleClick}>Incrementar</button>
7
8 {#if count % 2 === 0}
9   <strong>iEs par!</strong>
10 {:else}
11   <small>iEs impar!</small>
12{/if}

```

Seguramente te lo estés preguntando, pero obviamente también permite utilizar una condición `else if` para simplificar el código. De esta forma, podríamos dejarlo de esta forma:

```

1 {#if count === 0}
2   <strong>Es 0</strong>
3 {:else if count % 2 === 0}
4   <h4>iEs par!</h4>
5 {:else}
6   <small>iEs impar!</small>
7{/if}

```

Al final, el renderizado condicional no es muy diferente a cómo trabajamos con las condiciones en nuestro código *JavaScript* pero *Svelte* tiene una sintaxis especial.

Dominar el renderizado condicional comienza a darle más sentido a los componentes de *Svelte*, ya que hace que se puedan adaptar a los diferentes estados y props que le llegan, y eso es clave para ser realmente reutilizable.

Renderizando listas

Otra de las funcionalidades que nos ofrece *Svelte* es la posibilidad de renderizar listas de elementos. Esto es muy útil cuando queremos mostrar una lista de resultados que provienen de una API, por ejemplo.

Mientras que en *Vue* usamos la directiva `v-for` y en *React* podemos usar directamente un `.map`, en *Svelte* también contamos con una sintaxis especial para poder iterar `Array` (o iterables como `String`, `Map` y `Set`) y así poder renderizar cada elemento en la interfaz.

La sintaxis para renderizar una lista de elementos en *Svelte*

Imaginemos que tenemos un `Array` de películas en la constante `movies`. Cada película es un objeto y tiene las propiedades `title` y `year`:

```
1 const movies = [
2   {title: "The Avengers", year: 2012},
3   {title: "Avengers: Infinity War", year: 2018},
4   {title: "Avengers: Age of Ultron", year: 2015},
5   {title: "Avengers: Endgame", year: 2019}
6 ]
```

Para conseguir **renderizar esta lista de elementos en Svelte**, tenemos que utilizar la sintaxis de plantillas `#each` de la siguiente forma:

```
1 {#each lista as elemento}...{/each}
```

Así que para iterar la lista de películas que hemos visto antes, haríamos lo siguiente:

```
1 <h1>Lista de películas</h1>
2 <ul>
3   {#each movies as movie}
4     <li>
5       <strong>{movie.Title}</strong>
6       <date>{movie.Year}</date>
7     </li>
8   {/each}
9 </ul>
```

Como hemos comentado antes, puedes iterar cualquier tipo de lista. No hace falta que sea un Array. **Cualquier objeto iterable, que tenga una propiedad `.length` funcionaría.** Por ejemplo, prueba a iterar un string y verás como funciona.

A veces, cuando queremos renderizar una lista, es interesante poder tener el índice del elemento. Lo podemos lograr añadiendo a la sintaxis un segundo parámetro.

```
1 {#each lista as elemento, indice}...{/each}
```

Por lo que nuestro ejemplo, podría quedar así:

```
1 <h1>Lista de películas</h1>
2 <ul>
3   {#each movies as movie, index}
4     <li>
5       <span>#{index}</span>
6       <strong>{movie.title}</strong>
7       <date>{movie.year}</date>
8     </li>
9   {/each}
10 </ul>
```

Desestructurando objetos al iterarlos

Ahora bien, es un poco molesto tener que utilizar las propiedades `title` y `year` desde el objeto `movie`. ¿Se podría mejorar de alguna forma para poder usar directamente las propiedades en nuestro código? Sí, usando la **desestructuración del objeto** directamente en la sintaxis del `{#each}`:

Para ello haríamos lo siguiente:

```
1 <h1>Lista de películas</h1>
2 <ul>
3   {#each movies as {title, year}, index}
4     <li>
5       <span>#{index}</span>
6       <strong>{title}</strong>
7       <date>{year}</date>
8     </li>
9   {/each}
10 </ul>
```

Siguiendo el ejemplo de la desestructuración, **podemos cambiarle el nombre a la variable al vuelo**. Sólo tenemos que definir el mismo nombre, de la misma forma que lo haríamos con la sintaxis de la desestructuración en JavaScript:

```

1 <h1>Lista de películas</h1>
2 <ul>
3   {#each movies as {title: movieTitle, year: movieYear}, index}
4     <li>
5       <span>#{index}</span>
6       <strong>{movieTitle}</strong>
7       <date>{movieYear}</date>
8     </li>
9   {/each}
10 </ul>

```

No solo puedes utilizar la desestructuración del objeto. Puedes utilizar también el operador `rest` y también desestructurar arrays, en el caso que sea eso lo que estás iterando. Simplemente, sigue la misma sintaxis que usas en JavaScript y funcionará.

Renderizado condicional DENTRO de una iteración de elementos

Imaginemos que si la película fue lanzada hace menos de cinco años, queremos decir que la película es nueva.

Para ello, podemos renderizar todos los elementos con `#each` y, dentro, usar la etiqueta `#if` para renderizar el texto de “nueva”, o no, dependiendo de una condición.

```

1 <h1>Lista de películas</h1>
2 <ul>
3   {#each movies as {title: movieTitle, year: movieYear}, index}
4     <li>
5       <span>#{index}</span>
6       <strong>{movieTitle}</strong>
7       <date>{movieYear}</date>
8       {#if (new Date().getFullYear() - movieYear < 5)}
9         <strong>nueva</strong>
10      {/if}
11    </li>
12  {/each}
13 </ul>

```

[¡Prueba la demo!](#)³⁰

Renderizado condicional FUERA de la iteración

También deberíamos controlar cuando la lista de elementos es vacía, de forma que le mostremos al usuario algún tipo de aviso para que lo sepa. Para ello, podríamos utilizar el renderizado condicional antes de iterar los elementos:

³⁰<https://svelte.dev/repl/70f796e68596411fb58e55960602a1d7?version=3.20.1>

```
1 <h1>Lista de películas</h1>
2 <ul>
3   {#if movies.length > 0}
4     {#each movies as {title: movieTitle, year: movieYear}, index}
5       <li>
6         <span>#{index}</span>
7         <strong>{movieTitle}</strong>
8         <date>{movieYear}</date>
9         {#if (new Date().getFullYear() - movieYear < 5)}
10           <strong>nueva</strong>
11         {/if}
12       </li>
13     {/each}
14   {:else}
15     <p>
16       No hay películas
17     </p>
18   {/if}
19 </ul>
```

De esta forma, cuando `movies` sea una lista vacía, entonces mostrará al usuario que *no hay películas* y, si hay elementos, entonces los listaremos como le hemos indicado. Esto, funcionará bien, pero **existe una forma mucho más sencilla de conseguir el mismo resultado utilizando sólo la sintaxis de `#each`**, lo podemos lograr fusionando las dos funcionalidades así:

```
1 <h1>Lista de películas</h1>
2 <ul>
3   {#each movies as {Title: movieTitle, Year: movieYear}, index}
4     <li>
5       <span>#{index}</span>
6       <strong>{movieTitle}</strong>
7       <date>{movieYear}</date>
8       {#if (2020 - movieYear < 5)}
9         <span role="img">□</span>
10        {/if}
11      </li>
12    {:else}
13      <p>
14        No hay películas
15      </p>
16    {/each}
17 </ul>
```

Como ves, la sintaxis de `#each` también acepta una cláusula `:else` que se ejecutará cuando la lista que queremos iterar esté vacía.

Así conseguimos controlar más fácilmente este caso, sin necesidad de tirar del `#if` para hacer un renderizado condicional, y facilitará mucho más el código de nuestros componentes en Svelte.

Resumen de la sintaxis de iteración

Recapitulando, hemos visto cómo podemos renderizar listas de elementos con Svelte siguiendo la siguiente sintaxis:

```
1 // para iterar una lista para cada elemento
2 {#each lista as elemento}...{/each}
3 // para añadir el indice de cada elemento de la lista
4 {#each lista as elemento, indice}...{/each}
5 // para desestructurar las propiedades del elemento si es un objeto
6 {#each lista as {propiedad1, propiedad2}}...{/each}
7 // para usar el operador rest para guardar el resto de propiedades en una variable
8 {#each lista as {propiedad1, ...restoDePropiedades}}...{/each}
9 // para controlar cuando la lista no tiene elementos y renderizar otra cosa
10 {#each lista as elemento}...{:else}...{/each}
```

Ahora ya sabes cómo renderizar cada elemento de una lista iterable (que puede ser un Array, un String o cualquier tipo de objeto iterable de JavaScript).

Además, al final, hemos aprendido que no es necesario recurrir a la sintaxis de `#if` para controlar lo que debemos renderizar cuando no tengamos elementos en esa lista.

Fetching de datos y resolviendo promesas

Hoy en día muchas aplicaciones web se conectan a alguna API para obtener datos y mostrarlos en pantalla. Si queremos que nuestra página sea interactiva, vamos a necesitar obtener datos de forma asíncrona. En este capítulo vamos a ver cómo podemos hacerlo en *Svelte*.

Resolviendo promesas con declaraciones reactivas y condicionales

Cuando aprendemos Svelte, y ya conocemos otras bibliotecas como React, nos vemos tentados a guardar en el estado el resultado de la promesa.

Esto nos obliga a crear, **como mínimo**, dos estados: un `loading` para saber si la promesa se está resolviendo y otra para el valor de la promesa.

Por ejemplo, vamos a recuperar una imagen aleatoria de un zorro de la API `randomfox`. Crearemos las declaraciones reactivas `loading` y `foxImage` y la sintaxis del condicional `{#if}` para saber cuando estamos cargando y cuando tenemos la imagen.

```
1 <script>
2   let loading = true
3   let foxImage
4
5   const fetchRandomFox = async () => {
6     const response = await fetch('https://randomfox.ca/floof/')
7     const data = await response.json()
8     // actualizamos las declaraciones reactivas
9     loading = false
10    foxImage = data.image
11  }
12
13  fetchRandomFox()
14 </script>
15
16 {#if loading}
17   <p>...cargando</p>
```

```
18 { :else if foxImage}
19   <img src={foxImage} alt="Fox" />
20 {/if}
```

Enlace a la demo³¹

Esto funcionar... funciona. Lo bueno de Svelte es que, una vez que ya sabes cómo funcionan las declaraciones reactivas y la gestión del estado, si ya sabes JavaScript eres capaz de hacer casi cualquier cosa...

Pero hemos tenido que crear una variable `loading` para manejar cuando la promesa estaba cargando y cuando no. Además, **si queremos manejar si hay un error en la promesa**, deberíamos crear una variable más y empieza a generar demasiado boilerplate para algo tan sencillo.

¡Y ya sabemos que a Svelte **no le gusta el boilerplate!** Así que vamos a aprender una forma de evitarlo.

Resolviendo promesas con la sintaxis `#await`

Svelte proporciona una sintaxis de plantilla llamada `{#await}` que te permite trabajar directamente con la promesa sin necesidad de preocuparte en guardar diferentes estados para trabajar con la promesa, lo que **simplifica mucho el código**.

La sintaxis funciona de esta manera:

```
1 {#await promise}
2   <p>...cargando promesa</p>
3 { :then data}
4   <p>promesa resuelta con {data}</p>
5 {/await}
```

Para ver cómo nos podría ayudar, **vamos a reescribir nuestro ejemplo** de forma que haga uso de esta sintaxis:

³¹<https://svelte.dev/repl/ca4ed7ab2b97404b8822a4182016e169?version=3.31.0>

```

1 <script>
2   const fetchRandomFox = async () => {
3     const response = await fetch('https://randomfox.ca/floof/')
4     return response.json() // devolvemos directamente la promesa
5   }
6   // guardamos la promesa directamente en una variable
7   const foxImagePromise = fetchRandomFox()
8 </script>
9
10 {#await foxImagePromise}
11   <p>...cargando</p>
12 {:then data}
13   <img src={data.image} alt="Fox" />
14 {/await}

```

Enlace a la demo³²

¡Ajá! Menos líneas de código y menos variables que declarar. Por si fuera poco, en la sintaxis también podemos hacer directamente un catch de los errores de forma que, si la promesa no resuelve correctamente podremos renderizar un contenido diferente.

Te enseño cómo:

```

1 <script>
2   const fetchRandomFox = async () => {
3     const response = await fetch('https://randomfox.ca/floof/')
4     return response.json() // devolvemos directamente la promesa
5   }
6   // guardamos la promesa directamente en una variable
7   const foxImagePromise = fetchRandomFox()
8 </script>
9
10 {#await foxImagePromise}
11   <p>...cargando</p>
12 {:then data}
13   <img src={data.image} alt="Fox" />
14 {:catch error}
15   <p>Algo no ha ido bien!</p>
16 {/await}

```

Como ves, la sintaxis es muy sencilla y nos evita tener que crear variables para guardar los diferentes estados e información de la promesa. El código es más limpio y fácil de seguir.

³²<https://svelte.dev/repl/4b9c364fcf174fc2931c39442ded3e6e?version=3.31.0>

Usando la forma corta de la sintaxis

En el caso que tu promesa siempre resuelva (por lo tanto nunca entraría en el `catch`) y, además, no quieras controlar si la promesa está cargando, puedes usar la **sintaxis corta** con la que puedes usar directamente el valor de la promesa:

```
1  {#await promise then value}
2    <p>El valor que devuelve la promesa es {value}</p>
3  {/await}
```

En nuestro ejemplo quedaría de la siguiente forma:

```
1  <script>
2    const fetchRandomFox = async () => {
3      const response = await fetch('https://randomfox.ca/floof/')
4      return response.json() // devolvemos directamente la promesa
5    }
6    // guardamos la promesa directamente en una variable
7    const foxImagePromise = fetchRandomFox()
8  </script>
9
10 {#await foxImagePromise then data}
11   <img src={data.image} alt="Fox" />
12 {/await}
```

[Enlace a la demo³³](#)

Resumiendo la sintaxis #await

Teniendo en cuenta la cruzada contra el boilerplate que tiene Svelte en nuestros componentes y en **JavaScript** en general, no era ninguna sorpresa que tuviese expresiones en su plantilla preparadas para hacernos la vida más fácil con las promesas.

Además se parece mucho a la forma de trabajar con eventos asíncronos en **JavaScript con la sintaxis de `async/await` que se añadió en el lenguaje en ES2017**.

Con eso, podemos esperar a las promesas a resolverse y definir diferentes partes de nuestra interfaz dependiendo de si nuestra promesa todavía no se ha resultado (`await`), si se ha resuelto (`resolve/then`) o si ha fallado (`reject/catch`).

³³<https://svelte.dev/repl/458ba21ab528427e81cc06986d016fa3?version=3.31.0>

Creamos una pequeña app

Vamos a poner en práctica todo lo que hemos aprendido: declaraciones reactivas, props, eventos, sintaxis `#await`, etc.

Vamos a crear una pequeña aplicación que nos permita buscar películas con la API de [The OMDB³⁴](#). Tendremos un formulario para llenar la película que queremos buscar y mostraremos una lista de resultados. Además queremos que esto ocurra sólo cuando el usuario ha introducido una palabra de al menos 3 caracteres, para evitar hacer peticiones innecesarias.

Creando el buscador

Primero vamos a crear el buscador:

```
1 <script>
2   let value = ''
3   const handleInput = (event) => {
4     value = event.target.value
5   }
6 </script>
7
8 <input
9   placeholder="Introduce tu película"
10  value={value}
11  on:input={handleInput}
12 />
```

Mmmmmm, qué raro. ¿Realmente necesitamos crear una función para actualizar el valor de la variable `value`? ¿No podemos hacerlo directamente con otra directiva? Efectivamente, podemos usar `bind:value` para hacerlo:

³⁴<https://www.omdbapi.com/>

```
1 <script>
2   let value = ''
3 </script>
4
5 <input
6   placeholder="Introduce tu película"
7   bind:value={value}
8 />
```

¡Muchísimo mejor! Ahora vamos a envolver nuestro input en un formulario para escuchar el evento de submit. Así al escribir una película y pulsar Enter o Return se ejecutará la función handleSubmit:

```
1 <script>
2   let value = ''
3
4   const handleSubmit = (event) => {
5     alert(value)
6   }
7 </script>
8
9 <form on:submit|preventDefault={handleSubmit}>
10  <label>
11    Introduce tu película:
12    <input
13      name='movie'
14      placeholder="Avengers, The Matrix, etc."
15      bind:value={value}
16    />
17  </label>
18  <button type="submit">Buscar película</button>
19 </form>
```

Obteniendo los resultados

Ahora ya podemos hacer una petición a la API de The OMDB. Para ello vamos a usar la función `fetch` de la Web API. Esta función recibe una URL y devuelve una promesa que resuelve con un objeto Response. Este objeto tiene un método `json` que devuelve otra promesa que resuelve con el objeto JSON que contiene la respuesta de la API.

Consigue tu API KEY gratis desde la página web de OMDB^a El endpoint resultante sería este:
[http://www.omdbapi.com/?apikey=\\${API_KEY}&s=\\${textoABuscar}](http://www.omdbapi.com/?apikey=${API_KEY}&s=${textoABuscar})
^a<https://www.omdbapi.com/apikey.aspx>

```
1 <script>
2   let value = ''
3
4   const handleSubmit = (event) => {
5     fetch(`https://www.omdbapi.com/?apikey=422350ff&s=${value}`)
6       .then(response => response.json())
7       .then(data => {
8         console.log(data)
9       })
10    }
11 </script>
12
13 <form on:submit|preventDefault={handleSubmit}>
14   <label>
15     Introduce tu película:
16     <input
17       name='movie'
18       placeholder="Avengers, The Matrix, etc."
19       bind:value={value}
20     />
21   </label>
22   <button type="submit">Buscar película</button>
23 </form>
```

Ahora necesitamos guardar los resultados de la búsqueda en una variable para poder mostrarlos en la página. También vamos a controlar si está cargando o no la petición a la API:

```

1 <script>
2   let value = ''
3   let response // para guardar la respuesta de la API
4   let loading // para controlar si está cargando o no
5
6   const handleSubmit = (event) => {
7     loading = true
8     fetch(`https://www.omdbapi.com/?apikey=422350ff&s=${value}`)
9       .then(response => response.json())
10      .then(data => {
11        response = data.Search
12      }).finally(() => {
13        loading = false
14      })
15    }
16 </script>
17
18 <form on:submit|preventDefault={handleSubmit}>
19   <label>
20     Introduce tu película:
21   <input
22     name='movie'
23     placeholder="Avengers, The Matrix, etc."
24     bind:value={value}
25   />
26   </label>
27   <button type="submit">Buscar película</button>
28 </form>

```

Sí... ¡Demasiado código! ¿No hay una forma más sencilla de hacer esto? ¡Por supuesto! Vamos a usar la sintaxis de `#await` y así no tenemos que crear variables adicionales. Además podremos ya saber cuándo se está cargando la petición y cuándo no:

```

1 <script>
2   let value = ''
3   let promise // para guardar la promesa de la API
4   const handleSubmit = (event) => {
5     promise = fetch(`https://www.omdbapi.com/?apikey=422350ff&s=${value}`)
6       .then(response => response.json())
7   }
8 </script>
9
10 <form on:submit|preventDefault={handleSubmit}>

```

```

11 <label>
12   Introduce tu película:
13 <input
14   name='movie'
15   placeholder="Avengers, The Matrix, etc."
16   bind:value={value}
17   />
18 </label>
19 <button type="submit">Buscar película</button>
20 </form>
21
22 <div>
23   {#await promise}
24   <p>Cargando...</p>
25   { :then data }
26   <p>Cargado o de inicio</p>
27 </div>

```

Bueno, ya tenemos la petición de la API y sabemos cuando estamos esperando datos de la API. Sin embargo no somos capaces de diferenciar entre el estado inicial y cuando ya tenemos datos.

Para eso vamos a usar un renderizado condicional para que nos ayude. La API de OMDB te devuelve la propiedad `Response` con el string "False" en el caso de que no haya encontrado resultados. Si tiene resultados, tendrás en la propiedad `Search` un Array de películas. Si no tenemos ninguna de las dos propiedades, es que todavía no hemos usado el formulario:

Para no repetir constantemente el mismo código, voy a mostrar sólo la parte del renderizado de nuestra aplicación. El código completo lo puedes encontrar en [este enlace^a](#)

^a<https://svelte.dev/repl/ea0fe3585fe0427ba21144965770a166?version=3.31.0>

```

1 <div>
2   {#await promise}
3   <p>Cargando...</p>
4   { :then data }
5   { #if data?.Response === 'False' }
6     <p>No hay resultados</p>
7   { :else if data?.Search?.length > 0 }
8     <p>Hay resultados</p>
9   { :else }
10    <p>Usa el formulario para buscar una película</p>
11  {/if}

```

```
12  {/await}
13 </div>
```

Ahora ya podemos mostrar los resultados de la búsqueda. Para ello vamos a usar un bucle `each` para recorrer la lista de películas y mostrarlos en la página:

```
1 <div>
2   {#await promise}
3     <p>Cargando...</p>
4   {:then data}
5     {#if data?.Response === 'False'}
6       <p>No hay resultados</p>
7     {:else if data?.Search?.length > 0}
8       {#each data.Search as {Title, Year, Poster}}
9         <article>
10           <h3>{Title} ({Year})</h3>
11           <img alt={Title} src={Poster}>
12         </article>
13       {/each}
14     {:else}
15       <p>Usa el formulario para buscar una película</p>
16     {/if}
17   {/await}
18 </div>
```

Refactorización de la app

Si me preguntas, veo que se hacen demasiadas cosas en un sólo lugar. ¿Qué te parece si creamos componentes para solucionar esto? Te lo dejo como ejercicio para que pongas en práctica todo lo que has aprendido hasta ahora pero te muestro aquí una posible solución al reto:

```
1 <!-- Movie.svelte -->
2 <script>
3   export let title
4   export let year
5   export let poster
6 </script>
7
8 <article>
9   <h3>{title} ({year})</h3>
10  <img alt={title} src={poster}>
```

```
11 </article>
12
13 <!-- Movies.svelte -->
14 <script>
15   import Movie from './Movie.svelte'
16
17   export let movies
18 </script>
19
20 {#if movies?.length === 0}
21   <p>No hay resultados</p>
22 {:else if movies?.length}
23   {#each movies as movie}
24     <Movie {...movie} />
25   {/each}
26 {:else}
27   <p>Usa el formulario para buscar una película</p>
28 {/if}
29
30 <!-- Search.svelte -->
31 <script>
32   let value = ''
33   export let onSubmit
34 </script>
35
36 <form on:submit|preventDefault={() => onSubmit(value)}>
37   <label>
38     Introduce tu película:
39     <input
40       name='movie'
41       placeholder="Avengers, The Matrix, etc."
42       bind:value={value}
43     />
44   </label>
45   <button type="submit">Buscar película</button>
46 </form>
47
48 <!-- App.svelte -->
49 <script>
50   import Search from './Search.svelte'
51   import Movies from './Movies.svelte'
52
53   const API_URL = 'https://www.omdbapi.com/?apikey=422350ff'
```

```
54
55  let promise
56  let searchInput = ''
57
58  const handleSubmit = value => {
59    if (value.length <= 2) {
60      alert('Introduce más de 2 letras para buscar')
61      return
62    }
63    searchInput = value
64  }
65
66  const fetchMovies = async (search) => {
67    const response = await fetch(`[${API_URL}]&s=${search}`)
68    if (!response.ok) return []
69
70    const json = await response.json()
71    const {Response, Search} = await json
72
73    // hacemos la validación y mapeo de los
74    // datos en este punto
75    if (Response === 'False') return []
76
77    return Search.map(({Title, Year, Poster}) => ({
78      title: Title,
79      year: Year,
80      poster: Poster
81    }))
82  }
83
84  // declaración reactiva que debe ejecutarse cada vez
85  // que cambia el searchInput
86  $: if (searchInput.length > 2) {
87    promise = fetchMovies(searchInput)
88  }
89 </script>
90
91 <main>
92   <Search onSubmit={handleSubmit} />
93
94   {#await promise}
95     <p>Cargando...</p>
96   {:then movies}
```

```
97      <Movies movies={movies} />
98      {/await}
99  </main>
```

Playground con el código disponible³⁵

Como ves, además de crear componentes, hemos hecho que la lógica de la aplicación esté en el componente padre. Lo hemos logrado gracias a utilizar **una declaración reactiva**. Esto nos permite ejecutar código cada vez que cambia una variable. En este caso, cada vez que cambia `searchInput` se ejecuta el código que realiza la llamada a la API.

También hemos hecho que el componente `Movies` sea reutilizable, ya que podemos pasarle cualquier lista de películas y nos mostrará los resultados.

Faltaría añadir los estilos de la aplicación y también iterar un poco la información que mostramos, pero eso te lo dejo para que sigas practicando con Svelte.

³⁵<https://svelte.dev/repl/bfca0d06e7f34f0d8800e4bd841e97c7?version=3.31.0>

Crea un proyecto con Svelte

Hasta ahora hemos probado Svelte gracias al REPL de su página web. Es super útil para poder probar cosas rápidamente, pero no es muy útil para crear proyectos reales. Para eso necesitamos un entorno de desarrollo que, además, después podamos desplegar.

REPL significa Read-Eval-Print-Loop. Es un entorno de programación interactivo que permite ejecutar código y ver el resultado en tiempo real.

En este capítulo vamos a explicar cómo, paso a paso, puedes conseguir crear un nuevo proyecto con Svelte y tener listo tu entorno de desarrollo para poder desarrollar tu aplicación.

Para poder seguir este capítulo es tener instalado, al menos, **Node.js 16** y el administrador de paquetes **npm**.

Creando el proyecto de Svelte.js usando Vite

Para crear nuestro entorno de desarrollo local vamos a usar Vite.

Vite es un empaquetador de aplicaciones web modernas que se basa en el estándar ES Modules. Es muy rápido y tiene un modo de desarrollo que permite tener una experiencia de desarrollo muy fluida. Si conoces *Webpack*, *Vite* sería una alternativa.

```
1 npm create vite@latest my-svelte-app --template svelte
```

Vamos a **diseccionar un poco este comando**, para que entiendas perfectamente qué está haciendo:

npm: Es el empaquetador de paquetes de Node.js. Es el que se encarga de instalar los paquetes que necesitamos para nuestro proyecto.

create: Es un **sub-comando de npm para crear un proyecto de inicio**.

vite: La dependencia que usaremos como inicializador de nuestro proyecto. Algunos paquetes de npm cuentan con una plantilla que puede ser usada para crear un proyecto basado en él.

`@latest`: Es la versión de la dependencia que queremos instalar. En este caso, la última versión disponible.

`my-svelte-app`: **El nombre de nuestro proyecto.** Aquí podemos indicar el nombre de proyecto que queramos. Es importante notar que nos creará una carpeta con este nombre en el directorio desde el que hemos ejecutado el comando.

`--`: Es un separador que indica que los siguientes parámetros son para el comando `create` y no para `npm`.

`--template svelte`: Es un parámetro que indica que queremos usar la plantilla de `Svelte.js` para crear nuestro proyecto. Existen otras plantillas para `React`, `Vue`, etc.

Ahora que ya conocemos todos las partes del comando, vamos a proceder a ejecutarlo:

```
1 $ npm create vite@latest my-svelte-app -- --template svelte
2 Need to install the following packages:
3   create-vite@latest
4 Ok to proceed? (y) y
5
6 Scaffolding project in ~/midudev/my-svelte-app...
7
8 Done. Now run:
9
10 cd my-svelte-app
11 npm install
12 npm run dev
```

Ahora nos toca seguir las instrucciones de pantalla. Entramos en el directorio que hemos creado y ejecutamos el comando `npm install`.

Instala las dependencias y levanta el servidor

```
1 $ cd my-svelte-app
2 $ npm install
3
4 added 27 packages in 9s
```

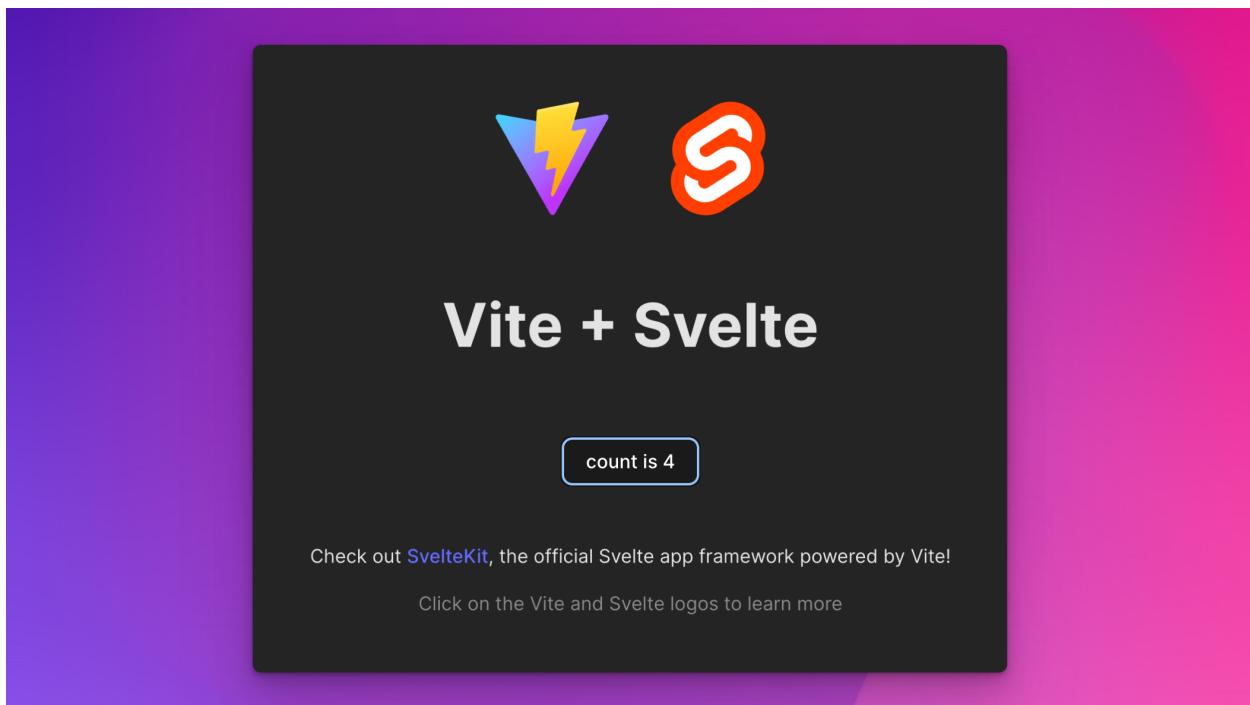
Una vez la instalación ha terminado, podremos ver los diferentes scripts que podemos ejecutar en nuestro proyecto. Para ello, ejecutamos el comando `npm run`:

```
1 npm run
2
3 Scripts available in my-svelte-app@0.0.0 via `npm run-script`:
4   dev
5     vite
6   build
7     vite build
8   preview
9     vite preview
```

Ya podemos ver que existen diferentes scripts que podemos ejecutar en nuestro proyecto. Actualmente nos interesa especialmente uno, el modo dev, así que ejecutamos el siguiente comando `npm run dev` y, al hacerlo, nos indicará una dirección localhost en el puerto 5173 (puede ser que a ti te salga otro) donde podremos acceder al servidor de desarrollo.

```
1 $ npm run dev
2
3 VITE v3.1.0 ready in 267 ms
4
5  Local: http://localhost:5173/
6  Network: use --host to expose
```

Ahora, si entramos en la dirección que nos facilita, veremos una típica aplicación de ejemplo realizada con *Svelte*.



Al entrar a la página, deberías ver algo similar a esto, que indica que la aplicación usa Svelte y Vite

Estructura de carpetas del proyecto

Si abrimos la carpeta con nuestro editor de código favorito, podremos ver la estructura de carpetas que nos ha creado. Vamos a revisarla aquí:

- `.vscode`: Una carpeta especial para Visual Studio Code. En ella se guarda la configuración de nuestro proyecto y ya viene inicializada con la recomendación de una extensión para trabajar con Svelte.
- `node_modules`: Donde tenemos todas las dependencias instaladas para nuestro proyecto (que hemos instalado previamente con `npm install`).
- `public`: La carpeta donde están los estáticos de nuestra aplicación. La que subiremos a Internet a la hora de hacer el deploy.
- `src`: Dentro se encuentra el código fuente de nuestra aplicación y contiene dos archivos. `main.js`, que es el punto de entrada de nuestra aplicación y `App.svelte` que es el componente que queremos renderizar.

También tenemos una lista de ficheros disponibles en la raíz:

- `.gitignore`: Un fichero especial para Git. En él se especifican los ficheros que no queremos que se suban a nuestro repositorio de Git.

- `index.html`: El fichero HTML que se servirá a los usuarios cuando accedan a nuestra aplicación. En él se incluye la carga del fichero `main.js` que es el punto de entrada de nuestra aplicación.
- `jsconfig.json`: Un fichero de configuración para el compilador de TypeScript. En él se especifican las opciones de compilación, cómo resolver dependencias y tipos.
- `package.json`: El fichero de configuración de nuestro proyecto. En él se especifican las dependencias del proyecto y los scripts que podemos ejecutar.
- `vite.config.js`: El fichero de configuración de Vite. En él se especifican las opciones de configuración para el empaquetador.

Revisando el código fuente

Vamos a revisar línea por línea para ver para qué sirven los dos ficheros que tenemos en la carpeta `src`. Empezamos con el archivo `main.js`, el punto de entrada de la app:

```
1 // importamos los estilos de nuestra aplicación
2 import './app.css'
3 // importamos el componente App
4 import App from './App.svelte'
5 // creamos una instancia de este componente
6 // y le pasamos un objeto como opciones
7 const app = new App({
8   // le indicamos el elemento del DOM donde queremos
9   // renderizar este componente
10  target: document.getElementById('app')
11 })
12
13 export default app
```

El archivo `App.svelte` importa un componente de Svelte, que es un contador, y un fichero `.svg`.

Ahora, si haces algún cambio a este componente y guardas los cambios, verás que la web que habías abierto se actualiza automáticamente y refleja los nuevos cambios. Esto es porque ya viene preparado para hacer *Hot Reloading*, de forma que no tengamos que refrescar nosotros a mano la pestaña.

Instala la extensión para tu editor favorito

Al entrar al archivo `App.svelte` es posible que veas todo el texto sin los colores resaltados, como si no soportase ese lenguaje. Para que tu editor pueda ayudaros a desarrollar en Svelte.js, debes instalar alguna extensión.

Si estáis usando, como yo, **Visual Studio Code**, entonces tenéis que instalar la [extension de Svelte³⁶](#) que da soporte a la sintaxis, además de hacer diagnósticos y dar información interesante al hacer hover sobre nuestras líneas de código.

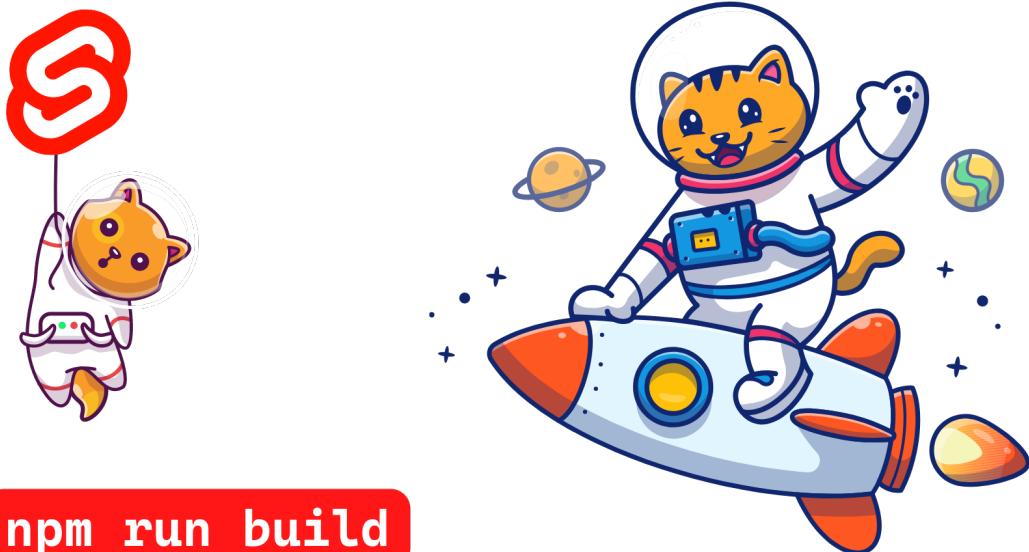
Conclusiones

Con esto ya tenemos todo nuestro entorno de desarrollo preparado para trabajar con Svelte. A partir de aquí, en los siguientes capítulos, veremos cómo podemos desplegar nuestra aplicación en producción, entre otras cosas.

³⁶<https://marketplace.visualstudio.com/items?itemName=svelte.svelte-vscode>

Despliegue de tu aplicación

Antes hemos creado una aplicación de ejemplo para buscar películas y ahora que ya sabemos cómo tener un entorno de desarrollo local, vamos a ver cómo desplegarla en un servidor para que sea accesible desde Internet.



Para que nuestra aplicación pueda ser disfrutada por cualquier persona, debemos publicarla en Internet.

Empezamos ejecutando el comando para crear nuestro proyecto:

```
1 $ npm create vite@latest search-movies -- --template svelte
```

Ahora nos toca seguir las instrucciones de pantalla. Entramos en el directorio que hemos creado y ejecutamos el comando `npm install`.

```
1 $ cd search-movies  
2 $ npm install
```

Una vez la instalación ha terminado, vamos a colocar nuestros archivos del proyecto. Te dejo aquí el playground con el código que necesitas³⁷.

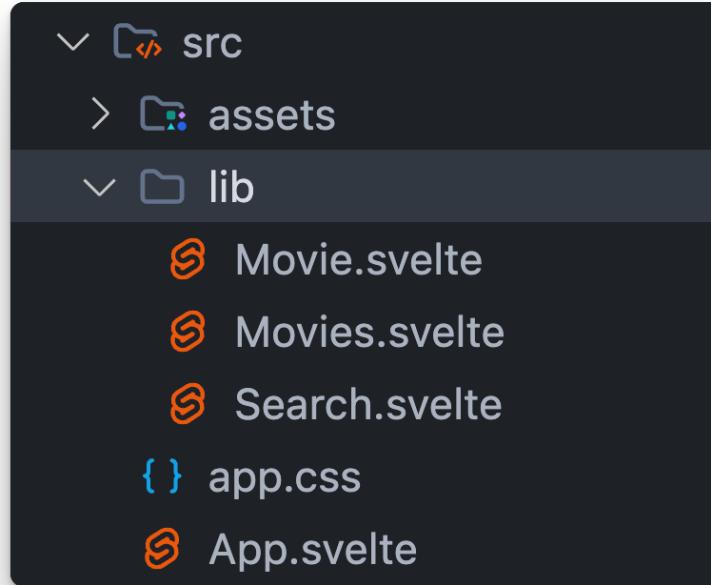
³⁷<https://svelte.dev/repl/bfca0d06e7f34f0d8800e4bd841e97c7?version=3.31.0>

El `App.svelte` de nuestro playground lo copiamos en el `src/App.svelte` de nuestro proyecto (machando lo que había antes). `Search.svelte` y `Movies.svelte` los copiamos en `src/lib`.

Recuerda cambiar los imports del fichero `src/App.svelte` para que apunten a los archivos correctos con la nueva estructura:

```
1 <!-- src/App.svelte -->
2 <script>
3   // import Search from "./Search.svelte"
4   // import Movies from "./Movies.svelte"
5   import Search from "./lib/Search.svelte"
6   import Movies from "./lib/Movies.svelte"
```

¿Quieres asegurarte de que todo funciona correctamente? Ejecuta el comando `npm run dev` y entra ala dirección que te ofrece en la consola. Si todo ha ido bien, deberías ver la aplicación funcionando (aunque es posible que con los estilos un poco diferentes)



La estructura de carpetas y ficheros debería haberte quedado muy similar a esta si has seguido los pasos

Una vez tenemos todo listo, ejecutamos el comando `npm run build` para generar la versión de producción de nuestra aplicación.

```
1 $ npm run build
2
3 > search-movies@0.0.0 build
4 > vite build
5
6 vite v3.1.0 building for production...
7 ✓ 9 modules transformed.
8 dist/index.html          0.44 KiB
9 dist/assets/index.65be3931.css 1.01 KiB / gzip: 0.54 KiB
10 dist/assets/index.2948b154.js 9.80 KiB / gzip: 4.20 KiB
```

Desplegando nuestra aplicación en Netlify

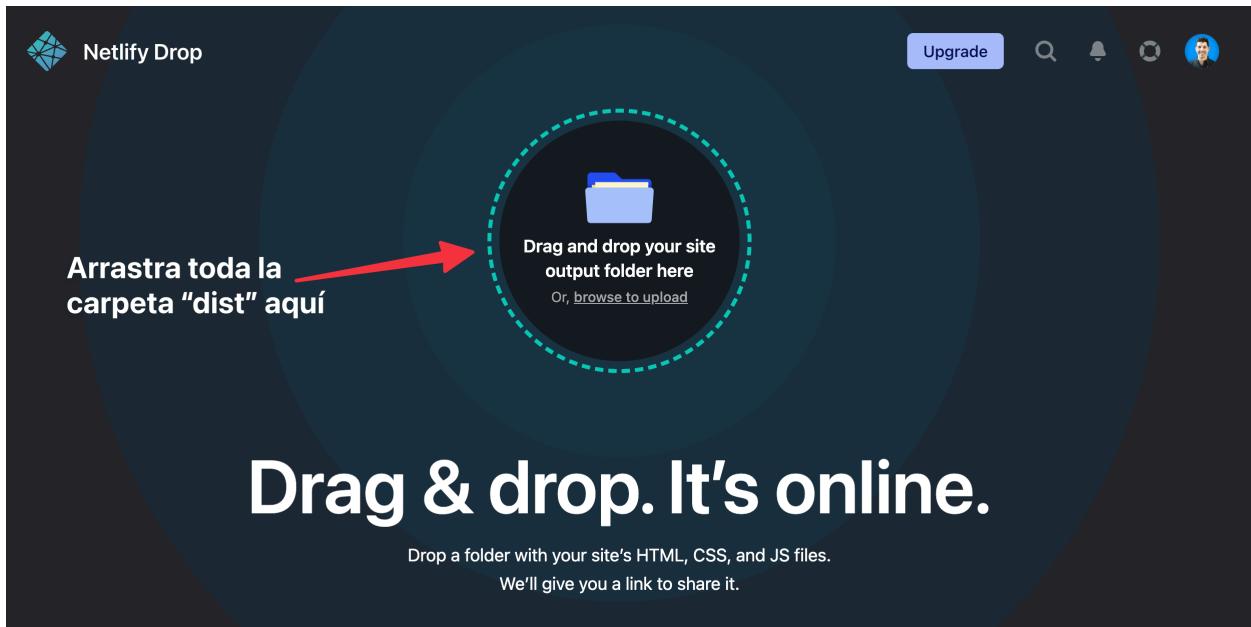
Ahora que tenemos nuestra aplicación lista para producción, vamos a desplegarla en un servidor. Para ello vamos a usar [Netlify Drop³⁸](#), un servicio gratuito que nos permite desplegar aplicaciones web estáticas en segundos.

Existen muchos servicios de hosting que nos permiten desplegar nuestra aplicación de Svelte de forma gratuita. Tienes otras alternativas como Vercel, Surge, GitHub Pages, etc. En este capítulo vamos a usar Netlify porque es muy sencillo pero queda en tu mano probar otras opciones.

Entra a [Netlify Drop³⁹](#) y arrastra el directorio dist de tu proyecto al círculo que aparece.

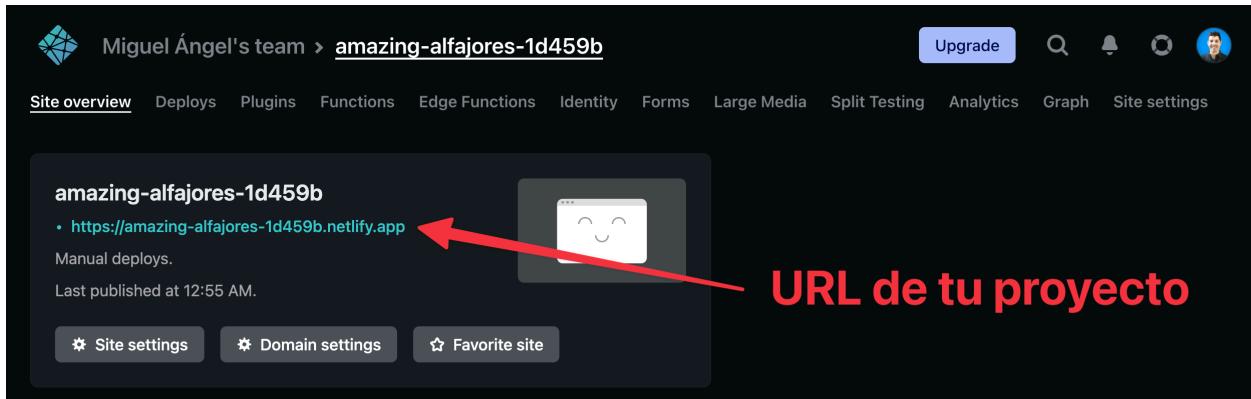
³⁸<https://app.netlify.com/drop>

³⁹<https://app.netlify.com/drop>



Al entrar en Netlify Drop deberías ver una página como esta. Arrastra la carpeta “dist” tras ejecutar “npm run build” donde señala la flecha

Si todo ha ido bien, deberías ver una pantalla como esta tras unos segundos de carga:



Tras subir tu carpeta “dist” te aparecerá una página como esta donde encontrarás la URL donde se ha desplegado tu aplicación

Si entras en la URL que te aparece, deberías ver tu aplicación funcionando y ya disponible en Internet para que sea visitada por cualquiera.

Configura el dominio de tu aplicación

Netlify te asigna una URL aleatoria para que puedas acceder a tu aplicación. Pero si quieres que tu aplicación tenga otro dominio, puedes configurarlo en Netlify.

Para ello, dentro de tu proyecto en Netlify, accede a:

Domain Settings -> Domains -> Options -> Edit site name.

The screenshot shows the Netlify 'Domain Settings' page for the site 'amazing-alfajores-1d459b'. The left sidebar has 'Domain management' selected, and 'Domains' is also highlighted with a red arrow. In the main 'Custom domains' section, there is a list containing 'amazing-alfajores-1d459b.netlify.app' (Default subdomain). To the right of this list is an 'Options' button with a red arrow pointing to it, and below it is a large red arrow pointing to a 'Edit site name' button.

Configura el dominio de tu aplicación en Netlify

Si lo prefieres, también puedes añadir un dominio personalizado. Para ello, accede a *Domain Settings -> Domains -> Add custom domain* y sigue las instrucciones.

Testing en Svelte

Ya sabemos desplegar nuestra aplicación pero todavía no hemos visto cómo podemos automatizar un proceso para asegurarnos de que lo que llevamos a producción es correcto.

En este capítulo vamos a ver cómo podemos hacer pruebas unitarias en Svelte. Para ello vamos a usar [Vitest⁴⁰](#), un framework de pruebas unitarias que está construido encima de Vite (justamente el empaquetador que usamos al crear aplicaciones web con Svelte).



Probar nuestros componentes de Svelte es muy fácil y nos permite refactorizar más tarde nuestro código con garantías

¿Qué es una prueba unitaria?

Una prueba unitaria es un tipo de prueba que se hace sobre una unidad de código. En el caso de Svelte, una unidad de código sería un componente. Por ejemplo, si tenemos un componente Button que tiene un botón con un texto, podemos hacer una prueba unitaria que compruebe que el texto del botón es el que esperamos.

Obviamente, también podemos aprovechar y hacer pruebas unitarias de funciones que no sean componentes. Por ejemplo, si tenemos una función que realiza alguna operación, podemos importar

⁴⁰<https://vitest.dev>

directamente la función en el test y hacer una prueba unitaria que compruebe que la función hace lo que esperamos.

Sin embargo, en este capítulo nos vamos a centrar en componentes de *Svelte*.

Hay bastante debate sobre si este tipo de pruebas son unitarias o de integración, ya que realmente estamos probando un componente que se integra con el framework de Svelte. En este capítulo vamos a usar el término *pruebas unitarias* entendiendo como unidad un componente de Svelte pero muchas veces, sobretodo cuando es un componente que importa otros componentes, estaríamos hablando de *pruebas de integración*.

Las dependencias que necesitamos

Para poder hacer pruebas unitarias en Svelte vamos a necesitar instalar tres dependencias:

- `vitest`: Framework de pruebas unitarias.
- `@testing-library/svelte`: Librería que nos va a ayudar a hacer las pruebas unitarias. Contiene multitud de utilidades para poder renderizar e interactuar con los componentes de Svelte, además de hacer introspección sobre los elementos renderizados.
- `happy-dom`: Librería que simula el DOM en Node.js. De esta forma podemos renderizar nuestros componentes de Svelte sin necesidad de levantar un navegador.

Todas son dependencias de desarrollo (esto significa que no se instalarán en producción), así que podemos instalarlas con el siguiente comando:

```
1 npm install --save-dev vitest @testing-library/svelte happy-dom
```

Añadiendo scripts y configuración

Ahora necesitamos añadir un script para poder ejecutar las pruebas unitarias. Para ello, vamos a añadir un script `test` en el `package.json`:

```
1 {
2   "scripts": {
3     "test": "vitest"
4   }
5 }
```

Antes de ejecutarlo, también necesitamos modificar el archivo de configuración de Vite `vite.config.js` para que use `happy-dom` como entorno para ejecutar los tests:

```

1 import { defineConfig } from 'vite'
2 import { svelte } from '@sveltejs/vite-plugin-svelte'
3
4 // https://vitejs.dev/config/
5 export default defineConfig({
6   plugins: [svelte()],
7   test: { // <-- añade este objeto a tu configuración
8     environment: 'happy-dom'
9   }
10 })

```

Tu primer test

Creamos una carpeta `test` y dentro creamos un nuevo fichero `Movie.test.js`. Este fichero contendrá todos los tests que queramos hacer sobre el componente `Movie`.

Para crear nuestro primer test, vamos a recuperar las utilidades de `vitest` y nuestro componente. Después creamos un test que comprueba que el componente está definido:

```

1 import {describe, expect, it} from 'vitest'
2 import Movie from "../src/lib/Movie.svelte"
3
4 describe('Movie', () => {
5   it('should be defined', () => {
6     expect(Movie).toBeDefined()
7   })
8 })

```

Vamos a explicar un poco el código:

- `describe` es una función que nos permite agrupar tests. En este caso, estamos agrupando todos los tests que queramos hacer sobre el componente `Movie`. En el primer parámetro le pasamos el nombre del grupo de tests, y en el segundo parámetro una función que contiene todos los tests.
- `it` es una función que nos permite crear un test. En este caso, estamos creando un test que comprueba que el componente está definido. En el primer parámetro le pasamos el nombre del test, y en el segundo parámetro una función que contiene el test.
- `expect` es una función que nos permite hacer una aserción. En este caso, estamos comprobando que el componente está definido. Existen una gran cantidad de aserciones que podemos usar, como `toBe`, `toReturn`, `toThrow`...

Si ejecutamos el comando `npm run test` veremos que el test pasa correctamente:

```

1 $ npm run test
2
3 > search-movies@0.0.0 test
4 > vitest
5
6
7 DEV v0.23.2 /Users/midudev/Dev/search-movies
8
9 ✓ test/Movie.test.js (1)
10
11 Test Files 1 passed (1)
12     Tests 1 passed (1)
13 Start at 10:34:34
14 Duration 652ms (transform 300ms, setup 0ms, collect 157ms, tests 3ms)
15
16 PASS Waiting for file changes...
17     press h to show help, press q to quit

```

Por defecto, después de ejecutar el script de test se queda a la escucha de cambios en los ficheros de test. Así cada vez que guardemos un fichero de test, se ejecutarán los tests de nuevo. Para salir debes pulsar la tecla q o ctrl + c.

Probando el componente

El test anterior no es que haya sido especialmente útil. Que un componente esté definido no nos dice mucho. Vamos a hacer un test un poco más interesante. Vamos a comprobar que el componente renderiza correctamente.

Para ello, vamos a usar la función render de `@testing-library/svelte`. Esta función nos permite renderizar un componente de *Svelte* y obtener un objeto con el componente renderizado y una serie de utilidades para poder interactuar con él.

```

1 import {describe, expect, it} from 'vitest'
2 import {render} from '@testing-library/svelte'
3 import Movie from "../src/lib/Movie.svelte"
4
5 describe('Movie', () => {
6   it('should be defined', () => {
7     expect(Movie).toBeDefined()
8   })
9
10  it('should render correctly', () => {

```

```

11     render(Movie, {
12       props: {
13         title: 'The Matrix',
14         year: '1999',
15         poster: 'https://image.tmdb.org/t/p/w500/6u1fYtxG5eqjhtCPDx04pJphQRW.jpg'
16       }
17     })
18   })
19 })

```

render de @testing-library/svelte recibe dos parámetros: el componente a renderizar y las propiedades que queramos pasar al componente. En este caso, le estamos pasando las propiedades title, year y poster.

Si ejecutamos el comando `npm run test` veremos que el test pasa correctamente.

Como ves no existe ningún `expect`. En este caso, puede tener sentido. Simplemente queremos comprobar que el componente se renderiza correctamente. Si el componente no se renderiza correctamente, render lanzará una excepción y el test fallará.

Pero... Aunque el test renderiza, lo cierto es que no sabemos si renderiza lo que esperamos y sería interesante asegurarnos de que el usuario verá la información que espera. Vamos a iterar un poco el test para conseguir al menos revisar que la imagen se renderiza correctamente.

```

1 it('should render correctly the poster', () => {
2   const {getByAltText} = render(Movie, {
3     props: {
4       title: 'The Matrix',
5       year: '1999',
6       poster: 'https://image.tmdb.org/t/p/w500/6u1fYtxG5eqjhtCPDx04pJphQRW.jpg'
7     }
8   })
9
10  const img = getByAltText('The Matrix')
11  expect(img.getAttribute('src')).toBe('https://image.tmdb.org/t/p/w500/6u1fYtxG5eqj\
12 htCPDx04pJphQRW.jpg')
13 })

```

La función `render` nos devuelve diferentes funciones para poder interactuar con el componente renderizado. En este caso, estamos usando `getByAltText` para obtener la imagen del componente pero **ofrece muchas más formas**. La lista de todas las disponibles es: `getByRole`, `getByLabelText`, `getByPlaceholderText`, `getByText`, `getByDisplayValue`, `getByTitle` y `getById`.

Aunque puedes tener la tentación de usar siempre `getByTestId` y simplemente añadir *data-attributes* a tus elementos, lo mejor es que siempre intentes que tus selectores sean lo más semánticos posible.

Pero no sólo puedes usar `getBy...` para recuperar elementos. También puedes usar `queryBy...` y `findBy...` .

La diferencia es que `getBy...` lanza una excepción si no encuentra el elemento, mientras que `queryBy...` devuelve `null` y `findBy...` devuelve una promesa que se resuelve cuando encuentra el elemento. Justo el último vendrá bien cuando tengamos contenido asíncrono, como veremos más adelante.

También puedes usar `getAllBy...` , `queryAllBy...` y `findAllBy...` para recuperar una lista de elementos a través de diferentes consultas.

Después hemos comprobado que la imagen tiene el atributo `src` con el valor que esperamos (que debe ser la URL de la imagen).

Ejercicio: Con todo lo que has aprendido hasta este punto, deberías poder asegurarte de que nuestro componente también renderiza el título y el año. Haz los cambios que consideres necesarios para poder comprobar que eso es así.

Un caso más complejo

Nuestro componente `Movie` es bastante sencillo. Es estático, no tiene interacción con el usuario y, además, es bastante pequeño. ¿Podríamos probar toda la aplicación con tests así? La respuesta es que sí. Y vamos a verlo.

```
1 import { fireEvent, render, } from "@testing-library/svelte"
2 import { describe, expect, it, vi } from "vitest"
3 import App from "../src/App.svelte"
4
5 describe('App', () => {
6
7   it('user should be able to search a movie and get results', async () => {
8     const { getByRole, findAllByText, findAllByAltText } = render(App)
9     // recuperamos el input para la película
10    const input = getByRole('textbox')
11    // recuperamos el formulario del input
12    const form = input.closest('form')
13
14    // escribimos la película que queremos buscar en el input
15    await fireEvent.input(input, { target: { value: 'Avengers' } })
16    // enviamos el formulario
17    await fireEvent.submit(form)
18
19    // buscamos todos los títulos de Avengers por su texto
20    const title = await findAllByText(/Avengers/i)
21    // recuperamos las imágenes de los resultados por su alt
22    const img = await findAllByAltText(/Avengers/i)
23
24    // revisamos que estén definidos
25    expect(title[0]).toBeDefined()
26    expect(img[0]).toBeDefined()
27  })
28})
```

Aunque no es el mejor test del mundo, sí que es bastante interesante. Básicamente estamos simulando el comportamiento de un usuario y nos estamos asegurando que el *happy path* funciona correctamente.

Con un solo test hemos cubierto la funcionalidad de búsqueda de películas. Y lo hemos hecho de una forma bastante sencilla.

Ten en cuenta que nuestra aplicación es tan sencilla que puedo recuperar el input usando simplemente un `getByRole('textbox')` porque sé que sólo existe uno. Normalmente lo ideal es que uses el placeholder o el label para recuperar el elemento.

Aunque en este test estamos usando `fireEvent`, también puedes usar `@testing-library/user-event`

que es una librería que ofrece una API más cercana a la de un usuario real. Te dejo como ejercicio que la pruebes y refactorices el test utilizándola.

Mocking de los datos

Supongo que te habrás dado cuenta de que en el test anterior estamos haciendo una llamada a la API de The Movie Database. Y eso es un problema porque, si no tenemos conexión a internet, el test fallará. O si la API está caída, también fallará. O, lo que es peor, **vamos a estar haciendo llamadas a la API cada vez que ejecutemos los tests**.

Obviamente esto es algo que no queremos. Por eso vamos a usar un *mock* para simular la llamada a la API. Por ahora vamos a ver una forma de hacerlo, pero más adelante veremos una forma mucho mejor.

Primero, necesitamos un *mock* de la llamada a la API. Para ello, vamos a crear un archivo `__mocks__-./api-omdb.js` con el siguiente contenido:

```
1 export const RESPONSE_WITH_AVENGERS_MOVIES = {
2   Search: [
3     {
4       Title: "The Avengers",
5       Year: "2012",
6       imdbID: "tt0848228",
7       Type: "movie",
8       Poster: "https://m.media-amazon.com/images/M/MV5BNDYxNjQyMjAtNTdiOS00NGYwLWFmN\TAtNThmYjU5ZGI2YTI1XkEyXkFqcGdeQXVyMTMxODk2OTU@._V1_SX300.jpg"
9     },
10    {
11      {
12        Title: "Avengers: Endgame",
13        Year: "2019",
14        imdbID: "tt4154796",
15        Type: "movie",
16        Poster: "https://m.media-amazon.com/images/M/MV5BMTc5MDE20DcwNV5BM15BanBnXkFtZ\TgwMzI2NzQ2NzM@._V1_SX300.jpg"
17      },
18    {
19      {
20        Title: "Avengers: Infinity War",
21        Year: "2018",
22        imdbID: "tt4154756",
23        Type: "movie",
```

```
24     Poster: "https://m.media-amazon.com/images/M/MV5BMjMxNjY2MDU1OV5BM15BanBnXkFtZ\TgwNzY1MTUwNTM@._V1_SX300.jpg"
25   }
26   ],
27   totalResults: "144",
28   Response: "True"
29 }
30 }
```

Ahora vamos a modificar nuestro test de forma que vamos a usar este *mock* en lugar de la API real. Para ello, vamos a usar la función `vi` que nos permite *mockear* funciones y le vamos a decir cómo queremos que devuelva esta información:

```
1 import { fireEvent, render, } from "@testing-library/svelte"
2 import { describe, expect, it, vi } from "vitest"
3 import App from "../src/App.svelte"
4 import { RESPONSE_WITH_AVENGERS_MOVIES } from '../__mocks__/api-omdb.js'
5
6 describe('App', () => {
7
8   it('user should be able to search a movie and get results', async () => {
9     const { getByRole, findAllByText, findAllByAltText } = render(App)
10
11    // mockeamos las llamadas al método fetch
12    // de forma que devolvemos siempre la respuesta mockeada
13    global.fetch = vi.fn().mockImplementationOnce(
14      () =>
15        Promise.resolve(
16          {
17            ok: true, // necesitamos que la respuesta sea correcta
18            json: () => Promise.resolve(RESPONSE_WITH_AVENGERS_MOVIES)
19          }
20        )
21    )
22
23    const input = getByRole('textbox')
24    const form = input.closest('form')
25
26    await fireEvent.input(input, { target: { value: 'Avengers' } })
27    await fireEvent.submit(form)
28
29    const title = await findAllByText(/Avengers/i)
30    const img = await findAllByAltText(/Avengers/i)
```

```
32     expect(title[0]).toBeDefined()  
33     expect(img[0]).toBeDefined()  
34   })  
35 })
```

Lo bueno de esta forma de hacerlo es que no tenemos que modificar nada de nuestra aplicación. Simplemente estamos *mockeando* la llamada a la API en el test.

Lo malo es que, claramente, nuestro test ahora sabe sobre la implementación de nuestro componente de Svelte. ¿Qué pasa si en lugar de usar `fetch` pasamos a usar `axios`? Pues que nuestro test se rompería. Y no, eso no es bueno.

El libro se centra en Svelte pero dejó a ejercicio del lector investigar cómo lograr una mejor solución utilizando un servicio como [msw⁴¹](#).

⁴¹<https://mswjs.io/>

Slots

Aunque ya hemos desplegado una aplicación de ejemplo, todavía nos quedan muchas cosas por descubrir en Svelte. En este capítulo vamos a conocer los *slots*, por qué son esenciales y cómo van a llevar la composición de tus componentes a otro nivel.

¿Qué son los slots?

Los componentes pueden envolver otros componentes o elementos HTML. Esto es clave para conseguir que nuestros componentes se puedan reutilizar de forma flexible. Por ejemplo, imagina un componente Button que queremos que su contenido sea configurable. Podríamos hacerlo de la siguiente forma:

```
1 <!-- Button.svelte -->
2 <button>
3   <slot><span>Click me</span></slot>
4 </button>
```

```
1 <!-- App.svelte -->
2 <script>
3   import Button from './Button.svelte';
4 </script>
5
6 <Button>
7   <strong>Awesome button!</strong>
8 </Button>
9
10 <Button></Button>
```

En este ejemplo, el contenido del componente Button es reemplazado por el texto `Awesome button!` en negrita. Si no se proporciona ningún contenido, el componente Button mostrará el texto `Click me` por defecto que hemos indicado dentro de las etiquetas `<slot>`.

En React este concepto se llama `children`. Y, aunque es muy similar, no es tan potente como los slots de Svelte.

Slots nombrados

Los *slots* nombrados son una característica muy útil que nos permite tener múltiples *slots* en un componente. Por ejemplo, podríamos tener un componente Card que tenga un slot para el título, otro para el footer y otro para el contenido que no es nombrado:

```
1 <!-- Card.svelte -->
2 <article class="card">
3   <header>
4     <slot name="title"><h3>Título por definir</h3></slot>
5   </header>
6   <div class="content">
7     <slot></slot>
8   </div>
9   <slot name="after"></slot>
10 </article>
```

Ahora podemos usar el componente Card e indicar dónde se tienen que renderizar los diferentes *slots*:

```
1 <!-- App.svelte -->
2 <script>
3   import Card from './Card.svelte';
4 </script>
5
6 <Card>
7   <h2 slot="title">Título de la Card</h2>
8   <p>Contenido de la card</p>
9   <footer slot="after">Pie de la card</footer>
10 </Card>
```

En este ejemplo:

- Donde teníamos el *slot title* vamos a renderizar un *h2* con el texto Título de la Card. Esto lo sabe gracias a que estamos usando el atributo *slot* en el *h2*.
- El párrafo que no tiene el atributo *slot* se va a renderizar en el *slot* no nombrado.
- El footer se va a renderizar en el *slot after*.

Al usar así el componente, el resultado final de lo que renderiza esta Card sería:

```
1 <article class="card">
2   <h2>Título de la Card</h2>
3   <div class="content">
4     <p>Contenido de la card</p>
5   </div>
6   <footer>Pie de la card</footer>
7 </article>
```

También funciona con componentes

Los *slots* también funcionan con componentes. Por ejemplo, imagina el ejemplo de la Card pero vamos a usar el componente Button que hemos creado al inicio del capítulo en el *slot after*:

```
1 <!-- App.svelte -->
2 <script>
3   import Card from './Card.svelte';
4   import Button from './Button.svelte';
5 </script>
6
7 <Card>
8   <h2 slot="title">Card con botón</h2>
9   <p>Contenido increíble</p>
10  <Button slot="after">Share card</Button>
11 </Card>
```

Rellena slots sin usar elementos o componentes

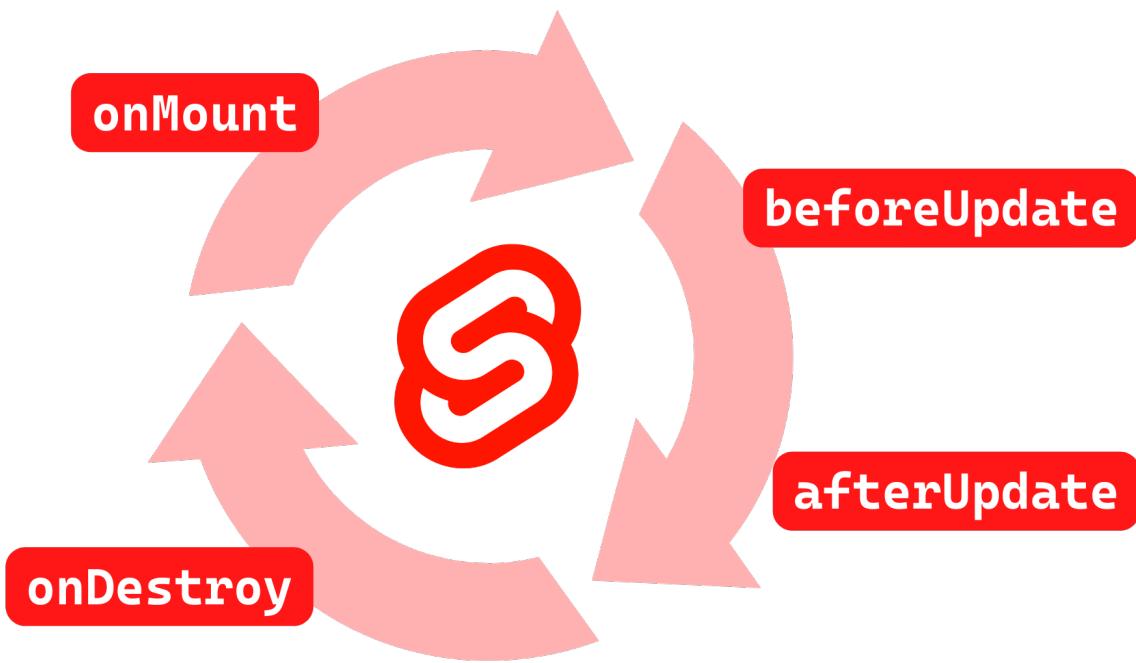
Hasta ahora hemos usado texto, un elemento HTML o un componente para llenar un slot pero... ¿Qué pasa si queremos, por ejemplo, rellenarlo con más de un elemento HTML sin envolverlo con un elemento padre?

Para ello podemos usar el elemento especial `<svelte:fragment>`, que es un elemento que no se renderiza en el DOM. Por ejemplo, si queremos llenar el *slot after* de la Card con dos botones

```
1 <!-- App.svelte -->
2 <script>
3   import Card from './Card.svelte';
4   import Button from './Button.svelte';
5 </script>
6
7 <Card>
8   <h2 slot="title">Card con botones</h2>
9   <p>Contenido increíble</p>
10  <svelte:fragment slot="after">
11    <Button>Share card</Button>
12    <Button>Like card</Button>
13  </svelte:fragment>
14 </Card>
```

Ciclo de vida de los componentes

En Svelte, todos los componentes tienen un ciclo de vida. Este ciclo de vida se puede dividir en cuatro fases: montaje, pre-actualización, post-actualización y destrucción. En cada una de estas fases, el componente puede ejecutar código.



Las 4 fases del ciclo de vida de los componentes de Svelte ofrecen funciones para ejecutar código en cada uno de ellos

Montaje

Cuando se crea un componente, se ejecuta el código que se encuentre en el método `onMount`. Este método se ejecuta una única vez, justo después de que el componente se haya montado en el DOM.

Su uso más frecuente es el de hacer llamadas a APIs externas para obtener datos que se mostrarán en el componente.

```
1 <script>
2   import { onMount } from 'svelte'
3
4   let data = []
5
6   // Se ejecuta justo después de que el componente se haya montado en el DOM
7   onMount(async () => {
8     const res = await fetch('https://jsonplaceholder.typicode.com/todos')
9     data = await res.json()
10    })
11  </script>
12
13 <ul>
14   {#each data as item}
15   <li>{item.title}</li>
16   {/each}
17 </ul>
```

Como ves, en este ejemplo, se hace una llamada a la API de [JSONPlaceholder⁴²](#) para obtener una lista de tareas. Una vez que se obtienen los datos, se asignan a la variable `data` y se renderizan en el componente.

Ademas, podemos usar sin problemas `async/await` dentro del método `onMount`.

Es posible que intentes hacer el `fetch` en el `<script>` pero sin usar `onMount`. Y, seguramente, funcionaría. Sin embargo... puede ser una mala práctica porque, como veremos más adelante, esto podría romperse si el componente se renderizase en el servidor.

Antes y después de actualizar

El método `beforeUpdate` se ejecuta justo antes de que el DOM se actualice con los cambios del componente. Y el método `afterUpdate` se ejecutaría justo después.

Estos métodos son útiles para ejecutar código que dependa de los cambios que se hayan producido en el componente.

Por ejemplo, si queremos hacer un scroll automático a un elemento del DOM, podemos hacerlo en el método `afterUpdate`.

⁴²<https://jsonplaceholder.typicode.com/>

```
1 <script>
2   import { afterUpdate } from 'svelte'
3
4   let data = [{}
5     title: 'Tarea 1'
6   }, {
7     title: 'Tarea 2'
8   }, {
9     title: 'Tarea 3'
10  }]
11
12  const addNew = () => {
13    data = [...data, { title: 'Nueva tarea' }]
14  }
15
16  afterUpdate(() => {
17    // Hacemos scroll automático al último elemento del DOM
18    // si no está en el scroll
19    document.querySelector('ul').lastElementChild.scrollIntoView()
20  })
21 </script>
22
23 <button on:click={addNew}>Add new item</button>
24 <ul>
25   {#each data as item}
26     <li>{item.title}</li>
27   {/each}
28 </ul>
```

Ten cuidado con el método `beforeUpdate` ya que la primera vez se ejecuta incluso antes que el componente esté montado. Por lo tanto, si intentas hacer algo con el DOM, puede que no exista.

Destrucción

El método `onDestroy` se ejecuta justo antes de que el componente se destruya. Este método es útil para limpiar cualquier evento o suscripción que se haya creado en el componente.

Por ejemplo, si en el componente se ha creado un `setInterval`, es necesario limpiarlo en el método `onDestroy`.

```
1 <script>
2   import { onDestroy } from 'svelte'
3
4   let counter = 0
5
6   const interval = setInterval(() => {
7     counter++
8   }, 1000)
9
10  onDestroy(() => clearInterval(interval))
11 </script>
12
13 <p>Counter: {counter}</p>
```

Los componentes se destruyen cuando se desmontan del DOM o cuando se actualiza el componente padre y no se necesita volver a renderizar el componente hijo.

Otro ejemplo podría ser, para limpiar la suscripción de un evento del DOM. Por ejemplo, al hacer un `addEventListener` para escuchar el evento `resize`.

```
1 <script>
2   import { onDestroy } from 'svelte'
3
4   let width = window.innerWidth
5
6   const handleResize = () => {
7     width = window.innerWidth
8   }
9
10  window.addEventListener('resize', handleResize)
11
12  onDestroy(() => window.removeEventListener('resize', handleResize))
13 </script>
```

Y otro, para devolver el foco al elemento que lo tenía antes de que el componente se montase (esto puede ocurrir cuando cierras una modal y quieras devolver el foco por temas de accesibilidad al elemento que lo abrió).

```
1 <script>
2   const prevFocusedElement = typeof document !== 'undefined' && document.activeEleme\
3 nt
4
5   if (prevFocusedElement) {
6     onDestroy(() => { prevFocusedElement.focus() })
7   }
8 </script>
```

Obviamente podrían existir más cosas en los que podría ser interesante usar el método `onDestroy`. Aunque la mayoría de las veces, como ves, siempre se trata de limpiar algún evento o suscripción o de devolver algo de la UI al estado que tenía antes de que el componente se montase.

Stores y estado global

Hasta el momento, cuando hemos hablado de estado, hemos estado hablando de estado local. Es decir, el estado que se almacena en una variable dentro de un componente. Pero, ¿qué pasa si queremos compartir el estado entre varios componentes? ¿Cómo podemos hacerlo?

En Svelte existe el concepto de *stores* (almacenes). Un *store* es un objeto que almacena un valor y que puede ser leído y escrito desde cualquier lugar. Además, un *store* puede ser compartido entre varios componentes.

Creando un store

Para crear un *store* en Svelte, podemos usar la función `writable`. Esta función recibe como parámetro el valor inicial del *store* y devuelve un objeto con dos propiedades: `subscribe`, `update` y `set`.

```
1 // src/stores.js
2 import { writable } from 'svelte/store'
3
4 export const count = writable(0)
```

En este ejemplo, hemos creado un *store* llamado `count` que almacena un valor numérico inicializado con el valor `0`. Este *store* se exporta para poder ser usado en otros componentes.

Usando un store

Para actualizar una *store* podemos usar el método `update` o el método `set`.

El método `update` recibe como parámetro una función que recibe como parámetro el valor actual del *store* y devuelve el nuevo valor.

El método `set` recibe como parámetro el nuevo valor del *store*.

Vamos a ver un ejemplo para actualizar el *store* `count` que hemos creado en el ejemplo anterior usando el método `update`

```
1 <!-- src/Increment.svelte -->
2 <script>
3   import { count } from "./stores.js"
4
5   const handleIncrement = () => {
6     // Actualizamos el valor del store count
7     // Usamos el método update que recibe
8     // una función con el valor anterior y
9     // devolvemos el nuevo valor
10    count.update(n => n + 1)
11  }
12 </script>
13
14 <button on:click={handleIncrement}>
15   Incrementar
16 </button>
```

En este ejemplo, hemos creado un componente que tiene un botón que al hacer click incrementa el valor del *store count* en 1. Pero no es que sea muy útil si no hay ningún sitio donde lo estemos mostrando.

Así que en el componente principal *App.svelte* vamos a leer la misma *store count* y la vamos a mostrar en pantalla. En este caso vamos a usar el método *subscribe* que recibe como parámetro una función que se ejecuta cada vez que el valor del *store* cambia.

```
1 <!-- src/App.svelte -->
2 <script>
3   import Increment from './Increment.svelte'
4   import { count } from './stores.js'
5
6   let currentCount = 0
7
8   count.subscribe((value) => {
9     currentCount = value
10    })
11 </script>
12
13 <h1>Contador: {currentCount}</h1>
14 <Increment />
```

En este ejemplo, hemos creado un componente *App.svelte* que muestra el valor del *store count* en pantalla y que tiene un componente *Increment*, el que hemos creado antes, que incrementa el valor del *store count* en 1.

Fíjate que estamos usando, efectivamente, un estado global. El valor del *store count* se puede leer y escribir desde cualquier componente.

Prueba el código de este ejemplo⁴³

Limpando la suscripción

Como justamente hemos visto en el capítulo anterior, cuando un componente se desmonta, se ejecuta el método `onDestroy` que se encarga de limpiar los eventos y las suscripciones.

En este caso, cuando el componente `App.svelte` se desmonta deberíamos limpiar la suscripción al *store count*.

Lo cierto es que es improbable que el componente `App` se desmonte, ya que es el componente principal de nuestra aplicación. Pero lo vamos a hacer como ejemplo.

```
1 <!-- src/App.svelte -->
2 <script>
3   import Increment from './Increment.svelte'
4   import { count } from './stores.js'
5
6   let currentCount = 0
7
8   // El método subscribe nos devuelve una forma
9   // de limpiar la suscripción
10  const unsubscribe = count.subscribe((value) => {
11    currentCount = value
12  })
13
14  // Al desmontar el componente, limpiamos la suscripción
15  onDestroy(() => unsubscribe())
16 </script>
17
18 <h1>Contador: {currentCount}</h1>
19 <Increment />
```

Auto-suscripción, la magia de Svelte

Seguro que lo has pensado: “Svelte habla de evitar boilerplate y no veas la de líneas que debemos añadir para suscribirnos a un store”. Pues tienes razón. Pero es que Svelte tiene una forma de hacerlo

⁴³<https://codesandbox.io/s/nice-fast-tv6ijd?file=/App.svelte>

mucho más fácil.

En vez de usar el método `subscribe` podemos usar la sintaxis `$count`. Svelte se encarga de suscribirnos al *store* y de actualizar el valor de la variable cada vez que el valor del *store* cambia.

```
1 <!-- src/App.svelte -->
2 <script>
3   import Increment from './Increment.svelte'
4   import { count } from './stores.js'
5 </script>
6
7 <h1>Contador: {$count}</h1>
8 <Increment />
```

Fíjate bien en la sintaxis `$count`. Svelte se encarga de suscribirnos al *store* `count` y de actualizar el valor de la variable cada vez que su *store* cambia.

Esto hace que la sintaxis se vuelva muchísimo más limpia que todo lo que hemos visto antes.

La auto-suscripción sólo funciona en las *stores* que son declaradas o importadas en el mismo componente. No funcionaría, por ejemplo, si pasas una store por prop (que tampoco es que haga falta).

También puedes usar la auto-suscripción en la parte de `script` de nuestro componente. Por ejemplo, si queremos que el valor de la variable `currentCount` se actualice cada vez que el valor del *store* `count` cambie, podemos hacerlo así:

```
1 <!-- src/App.svelte -->
2 <script>
3   import Increment from './Increment.svelte'
4   import { count } from './stores.js'
5
6   let currentCount = $count
7 </script>
8
9 <h1>Contador: {currentCount}</h1>
10 <Increment />
```

Es similar a lo anterior pero así puedes notar que también puedes usar la auto-suscripción en la parte de `script` de tu componente.

Svelte entiende que cualquier variable que empiece por \$ se refiere a una auto-suscripción de una *store*. Es por ello que no vas a poder crear variables que empiecen por \$, ya que es un símbolo reservado.

Stores de sólo lectura

Como hemos visto, hemos podido crear una *store* que se puede leer y escribir desde cualquier componente con el método `writable`. Pero no siempre queremos que sea así. A veces queremos que una *store* sea de sólo lectura.

Que sea de sólo lectura no significa que no se actualice. Simplemente las actualizaciones no se pueden hacer desde fuera de la *store*.

Por ejemplo, imagina que queremos **tener la información del ancho de la ventana del navegador y que sea accesible desde cualquier lugar de la aplicación**. Es una información que no tiene sentido que un componente intente actualizar esta información, ¿verdad?.

Para ello, podemos usar el método `readable` de Svelte. Este método recibe un valor inicial y una función que se ejecuta cuando el primer cliente se suscribe a esta *store*. Esto es ideal porque así, si ningún componente se suscribe a esta *store*, entonces no realizará trabajo innecesario.

Otro detalle importante es que la función que se ejecuta cuando el primer cliente se suscribe a esta *store* **debe devolver una función que se ejecuta cuando el último cliente se desuscribe**. Esto es importante porque así podemos limpiar cualquier evento o suscripción que hayamos creado.

```
1 // src/stores.js
2 import { readable } from "svelte/store"
3
4 // Este método se ejecutará cuando alguien se
5 // suscribe a esta store por primera vez
6 const onStart = (set) => {
7   // Creamos este método para usarlo tanto en el
8   // addEventListener como en el removeEventListener
9   const updateWidth = () => {
10     set(window.clientWidth) // actualiza el valor de la store
11   }
12
13 // Nos suscribimos al evento resize
```

```
14 window.addEventListener('resize', updateWidth)
15
16 // Este método se ejecutará cuando el último cliente
17 // elimine su suscripción de esta store
18 return () => {
19   window.removeEventListener('resize', updateWidth)
20 }
21 }
22
23 export const windowHeight = readable(window.clientWidth, onStart)
```

Ahora podríamos usar esta *store* en cualquier componente:

```
1 <!-- src/App.svelte -->
2 <script>
3   import { windowHeight } from './stores.js'
4 </script>
5
6 <h1>Ancho de la ventana: {$windowWidth}</h1>
```

Animaciones y transiciones

Con todo lo que hemos aprendido, ya podemos construir interfaces de usuario bastante potentes con Svelte. Pero ahora nos falta darle un poco de vida a nuestra aplicación. Para ello, Svelte nos ofrece de forma integrada formas muy sencillas de animar nuestros componentes.

Las transiciones

Las transiciones son una forma de animar los cambios de estado de nuestros componentes. Por ejemplo, podemos animar el desplazamiento de un componente cuando se añade o se elimina de la página o hacer que aparezca o desaparezca.

Normalmente, para animar un componente, tendríamos que usar una conjunción de CSS y JavaScript. O, en el caso de otros frameworks, usar librerías externas para conseguir la transición deseada.

Pero con Svelte es diferente. Nos ofrece de forma integrada en la biblioteca una serie de directivas que podemos usar en nuestros elementos.

En este ejemplo vamos a ver un checkbox que oculta o muestra un texto. Para lograrlo, haríamos algo así:

```
1 <script>
2   let visible = true
3 </script>
4
5 <label>
6   <input type="checkbox" bind:checked={visible}>
7   Mostrar texto
8 </label>
9
10 {#if visible}
11   <p>Este texto aparecerá o desaparecerá</p>
12 {/if}
```

[Playground con el código⁴⁴](#)

Como podemos ver, al hacer *clic* en el checkbox podemos ver como el texto aparece y desaparece. ¿Y si le añadimos una pequeña transición para que el texto haga un fundido? Lo haríamos así:

⁴⁴<https://svelte.dev/repl/1c9c69d9cf034dd88070cf6caa284cb0?version=3.50.0>

```

1 <script>
2   // importamos la transición de fundido
3   import { fade } from 'svelte/transition'
4   let visible = true
5 </script>
6
7 <label>
8   <input type="checkbox" bind:checked={visible}>
9     Mostrar texto
10 </label>
11
12 {*if visible}
13   <!-- con la directiva transition
14     usamos la transición 'fade' -->
15   <p transition:fade>
16     Este texto aparece o desaparece con fundido
17   </p>
18 {*}if

```

Playground de código⁴⁵

¡Fíjate lo fácil que ha sido crear la transición! Apenas dos líneas de código que hemos tocado y ya lo tenemos.

Parametrizando transiciones

Las transiciones también pueden recibir parámetros. En el ejemplo anterior, podríamos cambiar la duración de la transición a 5 segundos:

```

1 <p transition:fade={{ duration: 5000 }}>
2   Este texto aparece o desaparece con fundido
3   durante 5 segundos
4 </p>

```

También puedes retrasar la transición usando la propiedad `delay`:

```

1 <p transition:fade={{ delay: 1000, duration: 2000 }}>
2   Este texto aparece o desaparece con fundido
3   durante 2 segundos y un retraso de 1 segundo
4 </p>

```

⁴⁵<https://svelte.dev/repl/7e21cef09f3d4a298a30250a3f128f44?version=3.50.0>

Como te puedes imaginar, no sólo tienes la transición `fade`. La lista completa a día de hoy es: `fade` , `blur` , `fly` , `slide` , `scale` , `draw` y `crossfade`.

Algunas de estas transiciones tienen parámetros adicionales. Por ejemplo, la transición `fly` tiene los parámetro `x` y `y` para indicar la dirección del desplazamiento:

```
1 <p transition:fly={{ x: 100, y: 200 }}>
2   Este texto desde aparece desde 100px a la derecha y
3   200px desde abajo. Y al desaparecer se moverá allí.
4 </p>
```

Eventos de transición

Las transiciones también nos permiten ejecutar código cuando empiezan o terminan. Para ello, podemos usar la directiva `on` y después el evento que queramos escuchar.

Podemos saber cuando la animación de entrada empieza (`introstart`) y termina (`introend`). Y cuando la animación de salida empieza (`outrostart`) y termina (`outroend`).

Por ejemplo, podemos mostrar un mensaje en pantalla con el estado de la animación en cada momento.

```
1 <script>
2   import { fade } from 'svelte/transition'
3
4   let visible = true
5   let status = ''
6 </script>
7
8 <label>
9   <input type="checkbox" bind:checked={visible}>
10  Mostrar texto
11 </label>
12
13 {#if visible}
14   <p
15     transition:fade={{ duration: 1000 }}
16     on:introstart={() => status = 'intro empieza'}
17     on:introend={() => status = 'intro termina'}
18     on:outrostart={() => status = 'salida empieza'}
19     on:outroend={() => status = 'salida termina'}
20   >Este texto aparecerá o desaparecerá
21 </p>
22 {/#if}
23 <strong>Estado de la animación: {status}</strong>
```

Playground de código⁴⁶

⁴⁶<https://svelte.dev/repl/3699e5e081b34f72a467961bac0d98b9?version=3.20.1>

Svelte Kit

Hasta ahora hemos conocido *Svelte*. Y, aunque es genial, sólo puedes crear aplicaciones en el lado del cliente. Eso es, que no se ejecutan en el servidor y, por lo tanto, tienen ciertas desventajas a la hora de ser indexadas por los buscadores.

Para solucionar esto, *Svelte* ha creado *Svelte Kit*. Un framework que nos permite crear aplicaciones universales (que funcionan tanto en servidor como en cliente).

Próximamente

Routing con SvelteKit (enrutado)

Próximamente