



14 DE ENERO DE 2022

Svelte: qué es. Primeros pasos y cómo funciona

Frameworks JavaScript

Aplicaciones Distribuidas en Internet
ESTEBAN ANTÓN SELLÉS, JOAN GALIANA MAGÁN, JOSÉ MANUEL GARCÍA MOLINA

Contenido

Introducción a Svelte	3
Instalación y uso	4
Instalación	4
Utilización	5
Ejemplo práctico	6
Equivalencia en React.....	7
Para profundizar... Compilación en Svelte	12
Recordatorio de la creación predeterminada de componentes.....	12
Creación de un elemento	12
Actualización de un elemento	12
Borrado de un elemento	12
Adición de estilo a un elemento.....	13
Captura de evento click en un elemento	13
Sintaxis de Svelte	13
Componente básico	14
Estilado en un elemento	14
Uso de información dinámica	14
Manejadores de eventos	15
Operador asignación	16

Compilación de Svelte y código resultado	16
Resultado del componente básico	16
Resultado de la adición de información dinámica y manejadores	19
Enlace al código fuente y vídeo	29
Conclusiones.....	30
Referencias.....	31

Introducción a Svelte

Svelte es un framework web gratuito y de código abierto lanzado en abril de 2016. Las primeras versiones estaban escritas en JavaScript, pero a partir de la tercera, la actual, está escrito en TypeScript. Las aplicaciones Svelte no incluyen referencias al framework sino que la compilación de una aplicación Svelte genera código JavaScript para manipular el DOM, por lo que ahorra tiempo y tamaño en tiempos de arranque y en archivos transferidos. Svelte incluye su propio compilador y por tanto compila y convierte el código Svelte a código JavaScript. A diferencia de otros frameworks o librerías JavaScript, Svelte lo realiza en **tiempo de construcción, no de ejecución**. Si analizamos detenidamente este hecho, daremos con otra diferencia clave, y es que no envía código Svelte de ningún tipo al navegador, todo es JavaScript. De este modo, el navegador no tiene que volver a descargar la abstracción completa de la librería, como es el caso de React. Una comparación chocante que podemos aportar al respecto es:

- La implementación Svelte de la clásica web app ToDoList, ToDoMVC, pesa 3.6kb comprimido en zip. En contraparte, React y ReactDOM **sin ningún código de app** pesa unos 45kb comprimido. Aproximadamente trece veces más grande para el navegador sólo el hecho de evaluar React frente al arranque y ejecución de un ToDoList interactivo en Svelte.

Instalación y uso

Instalación

Una de las maneras más sencillas de comenzar es mediante el REPL de Svelte. Los REPL son espacios interactivos normalmente online que permiten explorar distintas herramientas exclusivas de lenguajes o entornos de desarrollo específicos. Más adelante, en el apartado de [Compilación en Svelte](#), hay un par de enlaces a dicho REPL. En el REPL, podemos escribir el código que necesitemos ya sea partiendo de cero o desde alguna de las plantillas que nos ofrecen. Cuando estemos satisfechos, descargaremos el fichero **svelte-app.zip** generado y lo descomprimiremos. Desde un terminal, nos colocamos en la raíz del directorio descomprimido y ejecutamos la orden **npm install**. Una vez hecho esto, **npm run dev** servirá nuestra app en localhost:5000 y la recargará utilizando el bundler Rollup cada vez que se guarde un cambio en algún fichero de nuestra app.

Otro método de instalación es degit, una herramienta de creación de proyectos. Para crear un nuevo proyecto en el directorio my-svelte-project, se procede de la siguiente manera.

```
npx degit sveltejs/template my-svelte-project
cd my-svelte-project
# to use TypeScript run:
# node scripts/setupTypeScript.js

npm install
npm run dev
```

Puedes generar una app lista para ponerse en producción mediante **npm run build**. El fichero README plantilla del proyecto contendrá instrucciones para desplegar la app mediante Vercel o Surge.

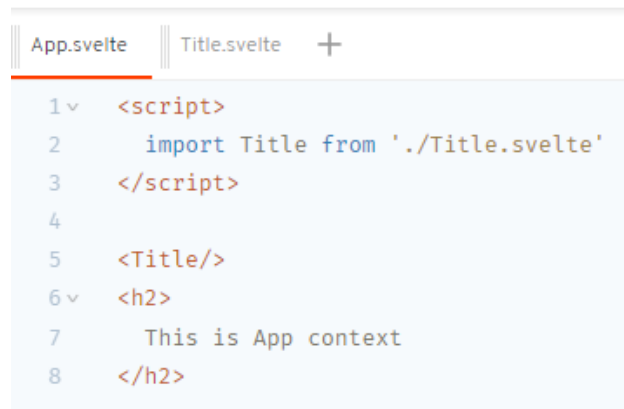
Utilización

La creación de un componente en Svelte se realiza creando un fichero .svelte en el cual escribiremos nuestras líneas de código. Un ejemplo sencillo puede ser



```
App.svelte  Title.svelte x +
1 <h1>
2   This is a title component
3 </h1>
```

Que se incluye en otros componentes mediante la importación del fichero contenedor del componente en cuestión



```
App.svelte  Title.svelte +
1 <script>
2   import Title from './Title.svelte'
3 </script>
4
5 <Title/>
6 <h2>
7   This is App context
8 </h2>
```

Este código produce la siguiente salida por pantalla.



```
Result  JS output  CSS output
This is a title component
This is App context
```

Y podríamos seguir haciendo importaciones en otros componentes y construir de manera modular nuestra aplicación, por ejemplo

```
App.svelte  Title.svelte  Main.svelte  +
1  <script>
2    import Main from './Main.svelte'
3  </script>
4
5  <Main></Main>
```

En la imagen, el código antes contenido en el fichero App.svelte ha sido movido al fichero Main.svelte, y hemos importado y usado ese componente. La salida generada es la misma que la que hemos visto anteriormente.

Ejemplo práctico

Siguiendo el [ejemplo](#) visto en clase proporcionado por el profesor en el cuál se realizaba una petición a una API y nos devolvía el tiempo en la ciudad indicada, hemos recreado esta aplicación y añadido tanto una nueva funcionalidad como importación de componentes.

Esta aplicación se ha implementado utilizando el [editor](#) (REPL) que Svelte nos proporciona desde su web, aunque podría haber sido escrita en nuestro entorno de desarrollo local.

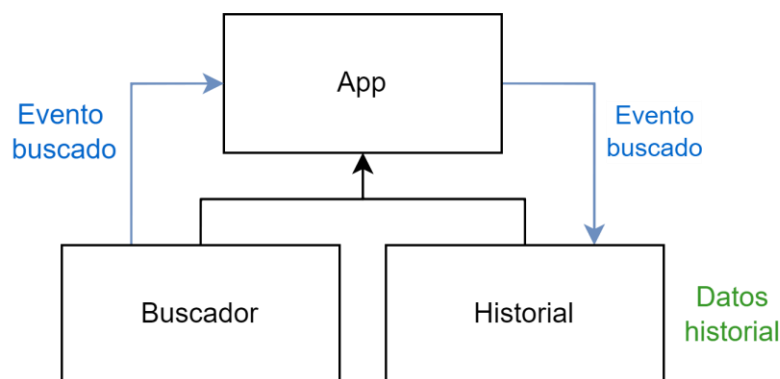
Comenzaremos creando un componente llamado **CityForm**, que contendrá un formulario con el campo de introducción de texto para la ciudad, y un botón de envío. Crearemos también un segundo componente llamado **ForecastInformation** que se encargará de mostrar los resultados devueltos por la API. Ambos componentes interactúan entre sí y están contenidos en un tercer componente, **ForecastComponent**, que se encarga de recibir el nombre de la ciudad al hacer clic en el botón del formulario, realizar la petición a la API y servir los datos al mostrador de información.

Equivalencia en React

A fin de realizar una comparación con React hemos implementado la misma aplicación de ejemplo utilizando este framework, respetando en la medida de lo posible la misma estructura, lógica y archivos.

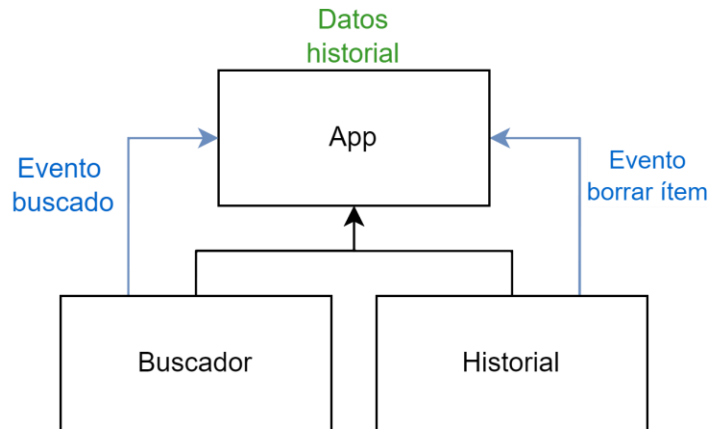
Durante la implementación se han observado claras diferencias, como por ejemplo el hecho de que React no permite ejecutar eventos de nodos hijos. Únicamente permite pasarle datos o funciones a los nodos hijos, que pueden ser ejecutadas por los hijos a modo de “evento”.

Los componentes buscador e historial son hijos de app. Historial contiene su propio estado, que se actualiza cuando se busca un elemento en el componente buscador, se emite un evento a app y app emite un evento a su hijo historial para que añada el elemento buscado.



Debido a que React no permite el envío de eventos a sus hijos y promueve el elevar el estado al elemento común más cercano, se ha cambiado de lugar los datos y funciones de manipulación del historial del componente historial al componente app.

Por lo tanto, para borrar un ítem del historial, el componente historial tiene que pedir al componente app que lo realice y el componente app tiene que encargarse de gestionar los datos del historial.



Por otra parte, respecto al maquetado, Svelte incluye sentencias de control mientras que React únicamente permite la utilización de javascript.

Svelte:

```
{#each history as item (item.id)}
<li>
  <span style="text-transform: capitalize;">
    {item.name}
  </span>
  <span style="cursor: pointer; margin: 1rem; color: red;" on:click={handleEraseItem(item.id)}>&times</span>
</li>
{/each}
</ol>
```

```
{#if error.message}
<p style="color: red">
  Error with <strong>{error.message}</strong> - Write a valid city name
</p>
{/if}
```

React:

```
<ol>
  {history.map(item => (
    <li key={item.id}>
      <span style={{ textTransform: 'capitalize' }}>
        {item.name}
      </span>
      <span style={{ cursor: 'pointer', margin: '1rem', color: 'red', }} onClick={() => handleEraseItem(item.id)}>&times</span>
    </li>
  ))}
</ol>
```

```
{error &&
  <p style={{ color: 'red' }}>
    Error with <strong>{error.message}</strong> - Write a valid city name
  </p>
}
```

Svelte también ofrece la posibilidad de manejar promesas directamente desde la maquetación.

Por lo que para realizar una petición de forma muy sencilla, la cual se volverá a ejecutar cuando cambie cityName de forma reactiva.

```
let information = fetchData();
$: url = `https:// ... r?q=${cityName}`;
async function fetchData() {
  const response = await fetch(url);

  if (response.ok) {
    return await response.json();
  } else {
    throw new Error(cityName);
  }
}

{#await information}
<p>
  ... waiting
</p>
{:then data}
<h1>
  {data.name}
</h1>
...
{:catch error}
<p style="color: red"> Error </p>
{/await}
```

Mientras que en React hay que hacer varias variables e ir cambiándolas según se realice la petición, además la reactividad es algo más confusa teniendo que usar explícitamente *useEffect* para crear un bloque reactivo.

```
useEffect(() => {
  const request = async () => {
    setIsLoading(true);
    setError(null);
    setForecastInfo(null);

    const url = `https:// ... r?q=${cityName}`;
    const response = await fetch(url);

    if (response.ok) {
      setForecastInfo(await response.json());
    } else {
      setError(new Error(cityName));
    }

    setIsLoading(false);
  };

  request();
}, [cityName, setForecastInfo]);

{isLoading && <p>... waiting</p>}

{forecastInfo &&
  <h1>
    {forecastInfo.name}
  </h1>
}

{error && <span>Error </span>}
```

Por otra parte, declarar una variable y cambiarla en Svelte es tan fácil como declararla y modificarla como se haría normalmente.

```
let cityName = "";  
// ...  
cityName = "valencia";
```

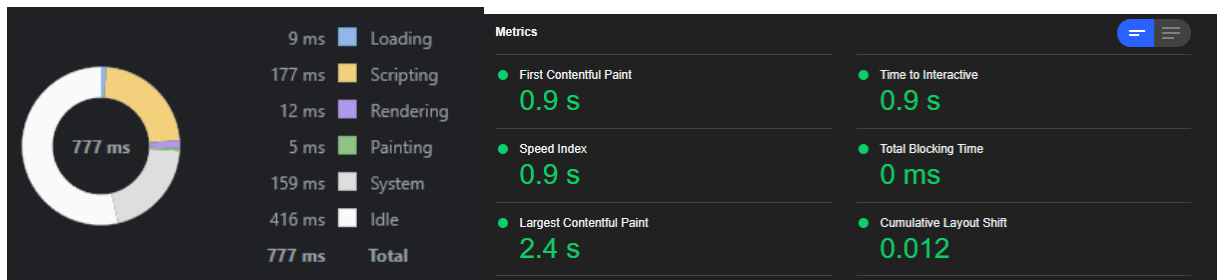
Mientras que en react es necesario usar “hooks”, las variables que no se declaran de esta forma pierden su valor cada vez que se vuelve a renderizar el componente, ya que se vuelve a ejecutar toda la función que lo compone.

```
const [cityName, setCityName] = useState("");  
// ...  
setCityName("valencia");
```

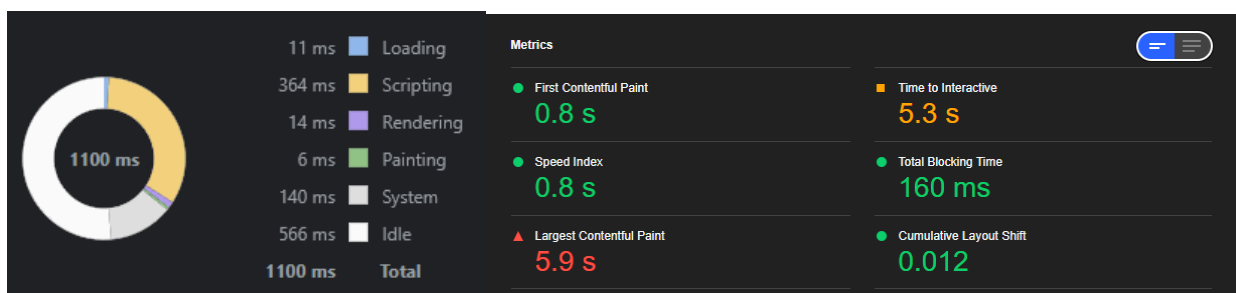
Para finalizar se realizará un pequeño benchmark de las dos aplicaciones obtenidas.

Del cual obtenemos los siguientes resultados:

Svelte



React



Hay que tener en cuenta que la aplicación sobre la que se ha realizado el benchmark no es muy complicada pero aun así podemos observar que los tiempos de ejecución de script han sido mucho menores en Svelte 177ms frente a 364 ms. Esto podría deberse a que React vuelve a ejecutar el código renderizando todos los componentes que podrían cambiar, mientras que Svelte únicamente ejecuta el código para modificar los nodos afectados. También se puede apreciar que el tiempo para ser interactivo de Svelte 0.9 s frente a 5.4 s es muy diferente, este valor es bastante importante debido a que cuanto menos tiempo tarde en poder interaccionar el usuario con la aplicación mejor sensación obtendrán los usuarios y también mejor puntuarán los indexadores de diferentes motores de búsqueda a la página web.

Para profundizar... Compilación en Svelte

Antes de comenzar, recordemos cómo se escriben web apps sin frameworks de ningún tipo.

Recordatorio de la creación predeterminada de componentes

Creación de un elemento

```
// create a h1 element
const h1 = document.createElement('h1');
h1.textContent = 'Hello World';
// ...and add it to the body
document.body.appendChild(h1);
```

Actualización de un elemento

```
// update the text of the h1 element
h1.textContent = 'Bye World';
```

Borrado de un elemento

```
// finally, we remove the h1 element
document.body.removeChild(h1);
```

Adición de estilo a un elemento

```
const h1 = document.createElement('h1');
h1.textContent = 'Hello World';
// add class name to the h1 element
h1.setAttribute('class', 'abc');
// ...and add a <style> tag to the head
const style = document.createElement('style');
style.textContent = '.abc { color: blue; }';
document.head.appendChild(style);
document.body.appendChild(h1);
```

Captura de evento click en un elemento

```
const button = document.createElement('button');
button.textContent = 'Click Me!';
// listen to "click" events
button.addEventListener('click', () => {
  console.log('Hi!');
});
document.body.appendChild(button);
```

Sintaxis de Svelte

Bien, una vez recordado esto, podemos comenzar a mirar los aspectos más básicos de la sintaxis de Svelte. Vamos a ver una serie de componentes básicos e incrementaremos la cantidad y complejidad de código poco a poco.

Componente básico

Aquí tenemos un componente básico

```
<h1>Hello World</h1>
```

[Pruébalo tú](#)

Estilado en un elemento

Al cual le podemos añadir estilo de la siguiente manera

```
<style>
  h1 {
    color: rebeccapurple;
  }
</style>
<h1>Hello World</h1>
```

[Pruébalo tú](#)

Uso de información dinámica

Nada impresionante hasta el momento, pero probemos a añadir información dinámica a nuestro componente. Para ello, procedemos de la siguiente manera

```
<script>
  let name = 'World';
</script>
<h1>Hello {name}</h1>
```

[Pruébalo tú](#)

Manejadores de eventos

También podemos añadir manejadores de eventos

```
<script>
  let count = 0;
  function onClickButton(event) {
    console.log(count);
  }
</script>
<button on:click={onClickButton}>Clicked {count}</button>
```

[Pruébalo tú](#)

Operador asignación

Y para manejar esa información, se utilizan los operadores de asignación

```
<script>
  let count = 0;
  function onClickButton(event) {
    count += 1;
  }
</script>
<button on:click={onClickButton}>Clicked {count}</button>
```

[Pruébalo tú](#)

Compilación de Svelte y código resultado

Una vez que hemos visto estos pequeños ejemplos, vamos a profundizar en cómo son compilados, qué código JavaScript se genera como salida e incrementalmente iremos entendiendo el funcionamiento del framework.

Resultado del componente básico

Para ello, comenzaremos analizando el primer y más simple de los ejemplos recién vistos, la utilización de una etiqueta `<h1>`. El componente se compila de la siguiente manera en nuestra aplicación Svelte.

```
function create_fragment(ctx) {
  let h1;

  return {
    c() {
      h1 = element('h1');
      h1.textContent = 'Hello world';
    },
    m(target, anchor) {
      insert(target, h1, anchor);
    },
    d(detaching) {
      if (detaching) detach(h1);
    },
  };
}

export default class App extends SvelteComponent {
  constructor(options) {
    super();
    init(this, options, null, create_fragment, safe_not_equal, {});
  }
}
```

Este código (JavaScript, recuerda) es el producto de la compilación de nuestro código Svelte, y podemos dividirlo en dos grandes bloques para su mejor entendimiento. El primero sería el bloque *create_fragment* y el segundo sería el bloque *export default class App extends SvelteComponent*.

Los componentes de Svelte son los bloques de construcción de una aplicación Svelte. Cada componente de Svelte se centra en construir su propio fragmento del DOM final. El bloque *create_fragment* proporciona un “manual de instrucciones” al componente Svelte de cómo trabajar con el fragmento del DOM. El bloque *create_fragment* devuelve funciones, tales

como **c()**, **m()** y **d()**. La primera función, **c()**, contiene las instrucciones necesarias para **crear** todos los elementos del fragmento del DOM. En este caso, podemos apreciar que contiene las instrucciones necesarias para crear el elemento `<h1>`

```
c() {  
  h1 = element('h1');  
  h1.textContent = 'Hello world';  
},
```

La siguiente función, **m(target, anchor)**, contiene las instrucciones necesarias para **montar** el elemento en el objetivo (target)

```
insert(target, h1, anchor);  
  
// http://github.com/sveltejs/svelte/tree/master/  
export function insert(target, node, anchor) {  
  target.insertBefore(node, anchor || null);  
}
```

Por último, **d(detaching)** contiene las instrucciones para eliminar los elementos del objetivo

```
detach(h1);  
  
// http://github.com/sveltejs/svelte/tree/master/  
function detach(node) {  
  node.parentNode.removeChild(node);  
}
```

Bien, veamos ahora el segundo bloque, *export default class App extends SvelteComponent*. Cada componente **es** una clase, que puede ser importada e instanciada mediante [esta API](#). En el constructor, inicializamos el componente con información generada por el propio componente en la función *create_fragment*. Svelte pasará únicamente la información necesaria y la eliminará cuando ya no sea necesaria. Esto podemos comprobarlo eliminando la etiqueta `<h1>` de nuestro código y volviendo a compilar, la salida ahora será

```
class App extends SvelteComponent {
  constructor(options) {
    super();
    init(this, options, null, null, safe_not_equal, {});
  }
}
```

Svelte automáticamente pasará **null** en lugar de *create_fragment* a la función *init*. ¿Y qué hace la función *init*? Es la encargada de preparar las configuraciones internas (a las que referenciaremos mediante `$$`) como pueden ser:

- Propiedades de componentes, `ctx` (luego entraremos en detalle) y el contexto.
- Eventos del ciclo de vida de los componentes.
- Mecanismo de actualización de los componentes.

Por último, Svelte llama a la función *create_fragment* para crear y montar los elementos en el DOM.

Resultado de la adición de información dinámica y manejadores

Obviemos el ejemplo del estilo de un componente y pasemos a ver cómo se traduce el código Svelte que utiliza información dinámica.

Recordemos de qué fragmento de código estamos hablando

```
<script>
  let name = 'World';
</script>
<h1>Hello {name}</h1>
```

Y veamos el código resultado de su compilación

```
function create_fragment(ctx) {
  // ...
  return {
    c() {
      h1 = element('h1');
      h1.textContent = `Hello ${name}`;
    },
    // ...
  };
}
let name = 'World';

class App extends SvelteComponent {
  // ...
}
```

A simple vista, podemos apreciar varios cambios. El primero es que todo lo que ha sido definido entre nuestras etiquetas `<script>` ahora está a nivel **global**. El siguiente cuesta un

poco más de ver, pero se puede apreciar que el contenido (texto) del elemento `<h1>` ahora es un literal de plantilla (escrito entre ``` y no entre `'`).

Bien pero, ¿y si quisiéramos usar una función que actualizara a quién saludamos? Aquí tenemos dicho código

```
<script>
  let name = 'World';
  function update() {
    name = 'Svelte';
  }
</script>
<h1>Hello {name}</h1>
```

Y en la siguiente página tenemos el código resultado de la compilación

```

function create_fragment(ctx) {
  return {
    c() {
      h1 = element('h1');
      t0 = text('Hello ');
      t1 = text(/*name*/ ctx[0]);
    },
    m(target, anchor) {
      insert(target, h1, anchor);
      append(h1, t0);
      append(h1, t1);
    },
    p(ctx, [dirty]) {
      if (dirty & /*name*/ 1) set_data(t1, /*name*/ ctx[0]);
    },
    d(detaching) {
      if (detaching) detach(h1);
    },
  };
}

function instance($$self, $$props, $$invalidate) {
  let name = 'World';

  function update() {
    $$invalidate(0, (name = 'Svelte'));
  }

  return [name];
}

export default class App extends SvelteComponent {
  constructor(options) {
    super();
    init(this, options, instance, create_fragment, safe_not_equal, {});
  }
}

```

Si nos fijamos, ahora el contenido textual del elemento `<h1>` está separado en dos nodos de texto distintos. Además, se crea una función `update` y se añade en la devolución del método `create_fragment`, **p(ctx, dirty)** junto con **c()**, **m(target, anchor)** y **d(detaching)**. Se crea una nueva función `instance` a la cual se mueven los atributos creados dentro de la etiqueta `<script>` y que anteriormente se encontraban a nivel global. Esta función `instance` es añadida en la llamada al método `init`. Por último, la variable `name` que se usaba en el `create_fragment` ahora se obtiene de `ctx[0]`.

Pero, ¿por qué estos cambios? El compilador de Svelte es capaz de monitorizar todos los cambios realizados en las variables declaradas en nuestra etiqueta `<script>`. Monitoriza tanto si la variable puede mutar, reasignarse, si es modificable (`const` vs `let`), si se referencia en las plantillas, etc.

En este ejemplo, el compilador de Svelte procesa que la variable `name` puede ser reasignada (por la línea `name = 'Svelte';` en el método `update`) por lo tanto divide en partes el contenido textual del elemento `<h1>`, para poder actualizar dinámicamente partes del texto.

De esto se encarga **p(ctx, dirty)**, ya que contiene instrucciones para actualizar los elementos basadas en qué ha cambiado en el estado (`dirty`) y el estado del componente (`ctx`).

Para permitir independencia entre distintos componentes del mismo tipo declarados en nuestra aplicación, la función `instance` es la que ahora almacenará el código mutable, en este caso la variable `name`. De este modo, conseguimos que los componentes se muestren en base a su estado interno actual

```
<App />
<App />
<App />

<!-- could possibly be -->
<h1>Hello world</h1>
<h1>Hello Svelte</h1>
<h1>Hello world</h1>
<!-- depending on the inner state of the component -->
```

Pero, ¿de qué se encarga la función *instance*? Devuelve una lista de variables de instancia, que son variables que son referenciadas en las plantillas o bien mutadas o reasignadas. Esta lista de variables es la que recibe el nombre de **ctx**. En la función *init*, el compilador de Svelte llama a la función *instance* para crear el ctx, y lo utiliza para la creación de los fragmentos para los componentes.

```
// conceptually,  
const ctx = instance(/*...*/);  
const fragment = create_fragment(ctx);  
// create the fragment  
fragment.c();  
// mount the fragment onto the DOM  
fragment.m(target);
```

De este modo, ahora nos referiremos a la variable name usando el ctx, en lugar de acceder a ella desde fuera del componente.

```
t1 = text(/*name*/ ctx[0]);
```

Por último, nos falta por ver la nueva función introducida en el código, puede que la más crucial. La función *\$\$invalidate* es el "secreto" detrás del sistema de reactividad en Svelte. Cada variable que haya sido usada en las plantillas o bien reasignada o mutada, contará con una llamada a la función *\$\$invalidate* justo después de la asignación o mutación, como por ejemplo:

```

name = 'Svelte';
count++;
foo.a = 1;

// compiled into something like
name = 'Svelte';
$$invalidate(/* name */, name);
count++;
$$invalidate(/* count */, count);
foo.a = 1;
$$invalidate(/* foo */, foo);

```

La función `$$invalidate` marca la variable como “dirty” y le programa una actualización para el componente.

```

// conceptually...
const ctx = instance(/*...*/);
const fragment = create_fragment(ctx);
// to track which variable has changed
const dirty = new Set();
const $$invalidate = (variable, newValue) => {
  // update ctx
  ctx[variable] = newValue;
  // mark variable as dirty
  dirty.add(variable);
  // schedules update for the component
  scheduleUpdate(component);
};

// gets called when update is scheduled
function flushUpdate() {
  // update the fragment
  fragment.p(ctx, dirty);
  // clear the dirty
  dirty.clear();
}

```

Bien, veamos por último cómo se añadirían manejadores de eventos y las diferencias que ello conlleva.

```
<script>
  let name = 'world';
  function update() {
    name = 'Svelte';
  }
</script>
<h1 on:click={update}>Hello {name}</h1>
```

Las diferencias son

```
function create_fragment(ctx) {
  // ...
  return {
    c() {
      h1 = element('h1');
      t0 = text('Hello ');
      t1 = text(/*name*/ ctx[0]);
    },
    m(target, anchor) {
      insert(target, h1, anchor);
      append(h1, t0);
      append(h1, t1);
      dispose = listen(h1, 'click', /*update*/ ctx[1]);
    },
    p(ctx, [dirty]) {
      if (dirty & /*name*/ 1) set_data(t1, /*name*/ ctx[0]);
    },
    d(detaching) {
      if (detaching) detach(h1);
      dispose();
    },
  };
}

function instance($$self, $$props, $$invalidate) {
  let name = 'world';

  function update() {
    $$invalidate(0, (name = 'Svelte'));
  }

  return [name, update];
}
```

Ahora la función *instance* devuelve dos variables en lugar de una, y además, se añaden los métodos *listen* y *dispose* en las funciones **m(...)** y **d(...)**, respectivamente. La función *update* se devuelve ahora como parte del ctx debido a que es referenciada en la plantilla.

Además, debemos entender que Svelte inyectará sólo el código necesario para añadir el fragmento al DOM, y lo borrará cuando el fragmento sea borrado del propio DOM. Si probamos añadiendo más manejadores

```
<h1
  on:click={update}
  on:mousedown={update}
  on:touchstart={update}>
  Hello {name}!
</h1>
```

El código resultado de la compilación será

```
// ...
dispose = [
  listen(h1, 'click', /*update*/ ctx[1]),
  listen(h1, 'mousedown', /*update*/ ctx[1]),
  listen(h1, 'touchstart', /*update*/ ctx[1], { passive: true }),
];
// ...
run_all(dispose);
```

Sin embargo, si solo contamos con un manejador de eventos, Svelte asignará a una variable llamada *dispose* el método *listen* con los parámetros pertinentes, y la llamará como función única y no mediante el método *run_all*.

Recapitulando... La sintaxis de Svelte es un súper conjunto de HTML. Cuando escribes un componente en Svelte, el compilador analizará y generará código JavaScript optimizado como salida. Ese código puede dividirse en 3 grandes bloques:

- *Create_fragment* devuelve un fragmento, que es un manual de instrucciones para construir y manipular el fragmento del DOM del componente (con sus funciones **c**, **p**, **m**, **d**).
- *Instance* contiene la mayoría del código escrito en las etiquetas `<script>`. Devuelve una lista de variables de instancia referenciadas en la plantilla. Inserta `$$invalidate` después de cada asignación y mutación de la variable de instancia.
- *Class App extends SvelteComponent* inicializa el componente con las funciones *create_fragment* y *instance*. Además, crea los componentes predeterminados internos y provee la [API de componentes](#).

Svelte hace hincapié en generar el JavaScript más compacto posible, no definiendo los métodos *create_fragment* o *instance* si no son necesarios, partiendo el contenido textual de `<h1>` en nodos mutables o no, generado *dispose* como array o función dependiendo del número de manejadores de evento, etc.

Enlace al código fuente y vídeo

[Vídeo](#)

[Código en Svelte](#)

[Código en React](#)

Recuerda iniciar sesión en la API del clima para obtener tu propia API Key y ponerla en tu código, pues la del ejemplo estará desactivada tras la entrega.

Conclusiones

En conclusión, Svelte ofrece muchas facilidades para el desarrollador, con una reactividad clara y eficiente, evitando el renderizado de todo el árbol de nodos afectados realizando únicamente los cambios necesarios. A la hora de desarrollar la versión en React ha destacado la complicación que ha supuesto que no permitiera que un nodo padre envíe eventos a un nodo hijo dificultando la implementación, el uso de hooks el cual es más bastante más complejo que la utilización de variables simples y la declaración de código reactivo para el cual hay que utilizar *useEffect* el cual añade un poco de complejidad al asunto. A pesar de todo esto hay que tener en cuenta que React es uno de los frameworks más usados hoy en día, lo cual provoca que muchos proyectos se desarrollen en esta tecnología únicamente por popularidad o por la facilidad de obtener desarrolladores. Creemos que Svelte es una tecnología por la que apostar, ofreciendo grandes soluciones para problemas de tamaño pequeño o moderado.

Referencias

<https://en.wikipedia.org/wiki/Svelte>

<https://www.youtube.com/watch?v=FNmvcswdjV8>

<https://react-etc.net/entry/react-vs-svelte-the-javascript-build-time-framework>

<https://svelte.dev/blog/frameworks-without-the-framework>

<https://svelte.dev/tutorial/basics>

<https://lihautan.com/compile-svelte-in-your-head-part-1/>

<https://lihautan.com/compile-svelte-in-your-head-part-2/>

<https://dev.to/silvio/how-to-create-a-web-components-in-svelte-2g4j>

https://svelte.dev/docs#Client-side_component_API

<https://svelte.dev/examples/hello-world>

<https://svelte.dev/tutorial/making-an-app>

<https://svelte.dev/blog/the-easiest-way-to-get-started>

<https://openweathermap.org/api>

<https://openweathermap.org/current>

https://ottocol.github.io/widget-tiempo-frameworksJS/MVC/tiempo_angularJS.html

https://ottocol.github.io/ADI_2122/teoria/frameworks_JS/intro_frameworks.pdf