Javascript. Cómo funcionan las comillas invertidas (``)

Descubre los principales usos de las comillas invertidas en Javascript

Gerardo Fernández Follow

Oct 8, 2019 · 5 min read ★

Photo by Nhu Nguyen on Unsplash

> *English version: https://medium.com/@ger86/javascript-how-backticks-work-de269e0fb8ba*

Como sabréis, en Javascript existen dos formas principales para declarar *strings:*

- Mediante comillas simples `''`.
- Mediante comillas dobles `""`.

Sin embargo, con la especificación ES2015 llegó la posibilidad de crear lo que se conoce como "*template literals*" o "*template strings*" para lo cual basta con encapsular texto entre comillas invertidas:

```
`some text`
```

Esto nos dará una serie de ventajas con respecto a los dos métodos principales que describiré a lo largo de este artículo. ¡Vamos a verlos!

. . .

Concatenar e interpolar strings

Probablemente éste sea uno de los usos más extendidos y popularizados de las comillas invertidas ya que nos va permitir concatenar e interpolar *strings* de modo que nuestro código quede mucho más limpio.

Vamos a verlos con unos cuantos ejemplos comparando la sintaxis "antigua" con la que nos proveen las comillas invertidas.

```
const name = 'Gerardo';
const surname = 'Fernández';
const telephone = '123 123 123';

// "Old syntax"

const userInfo = 'User info: ' + name + ' ' + surname + ' ' +
telephone;

// "New syntax"

const userInfo = `User info: ${name} ${surname} ${telephone}`;
```

Como podéis ver, gracias a los *template literals* obtenemos un código mucho más fácil de leer y legible, ahorrándonos la concatenación de múltiples *strings* mediante el operador `+`.

Por supuesto, también podemos ejecutar código dentro de los *template strings:*

```
const user = getUserFromApi();

// "Old syntax"

const userInfo = 'User info: ' + user.getName() + ' ' +
user.getEmail();

// "New syntax"

const userInfo = `User info: ${user.getName()} ${user.getEmail()}`;
```

. . .

No es necesario escapar los caracteres " y ' mediante \

Otra de las ventajas de emplear las comillas invertidas (``) es que ya no es necesario escapar las comillas dobles (`""`) o las simples (`''`), algo a lo que habitualmente nos vemos obligados cuando trabajamos con idiomas como el inglés:

```
const foo = 'Can\'t connect to the server';

const bar = `Can't connect to the server`;
```

O por ejemplo:

```
const foo = "Error: \"Introduce a valid email\"";

const bar = `Error: "Introduce a valid email"`;
```

De modo que nuevamente obtenemos un código mucho más legible.

. . .

Strings en varias líneas

En Javascript no es posible declarar *strings* en varias líneas. Por ejemplo, si intentamos lo siguiente:

```
const html = "<article>
<h1>Article title</h1>
</article>";
```

Obtenemos el siguiente error:

```
SyntaxError: "" string literal contains an unescaped line break
```

Lo cual es bastante incómodo, especialmente cuando estamos creando HTML. Las alternativa que teníamos hasta ECMA2015 era emplear el caracter de línea nueva (`\n`):

```
const html = "<article> \
<h1>Article title</h1> \
</article>";
```

pero que sin embargo trae problemas en tiempo de compilación:

> *The whitespace at the beginning of each line can't be safely stripped at compile time; whitespace after the slash will result in tricky errors; and while most script engines support this, it is not part of ECMAScript.*

o emplear la concatenación mediante el operador `+` :

```
const html = '<article>' +
'<h1>Article title</h1>' +
'</article>';
```

algo que queda realmente “raro”.

Sin embargo, con las comillas invertidas podemos escribir lo siguiente:

```
const html = `<article>
<h1>Article title</h1>
</article>`;
```

de modo que el código que obtenemos es realmente limpio.

Eso sí, es importante recordar que los espacios no son eliminados por lo que si creamos un *string* del siguiente modo:

```
const string = `First line
                Second line`
```

lo que obtendremos será lo siguiente:

```
First
                Second
```

por lo que tendremos que recurrir al método `trim` para evitar los espacios extra.

. . .

Tagged templates

Otra de las características que nos aportan las comillas invertidas es la posibilidad de crear lo que se conocen como *tagged templates* (también conocidos como *tag functions*),

algo que es usado por librerías tan populares como Apollo o Styled Components:

```
// Apollo query

const query = gql`
  query {
    ...
  }


// Styled components

const Container = styled.div`
  width: 1000px;
  background: red;
`;
```

Tanto `styled.div` como `gql` son realmente **funciones** (***tag functions***) que extraen sus argumentos de un *template literal* (es decir, texto encapsulado entre comillas invertidas).

En el caso por ejemplo del styled component `Container` lo que obtenemos es un React Component que representa un `div` con el siguiente aspecto:

```
<div style="width: 1000px; background:red">
...
</div>
```

Para entender el funcionamiento de este tipo de funciones supongamos que tenemos la función `sayHello` declarada del siguiente modo:

```
function foo() {
  console.log(arguments[0]);
  console.log(arguments[1]);
  console.log(arguments[2]);
}
```

Si ahora la invocamos como si fuera una *tag function*:

```
const varOne = 'bar';
const varTwo = 'zeta';

foo`This is bar: ${varOne} and zeta: ${varTwo}`;
```

lo que obtendremos por pantalla será:

```
["This is bar: ", " and zeta: ", ""] (array)
bar (string)
zeta (string)
```

Es decir, la *tag function* está recibiendo 3 argumentos:

- el primero es un array en el que aparece el literal partido en varios strings empleando las expresiones `${...}` como *breakpoints*
- el resto de argumentos (`arguments[1]` y `arguments[2]`) son el resultado de las variables ya interpoladas (`bar` y `zeta`)

Aprovechando el *spread operator* podemos reescribir nuestra función `foo` del siguiente modo:

```
function foo(literals, ...expressions) {
  console.log(literals);
  console.log(expressions[0]);
  console.log(expressions[1]);
}
```

De este modo la longitud del array `literals` será siempre el del array `expressions` más uno.

A partir de este momento las posibilidades que nos ofrecen las *tag functions* son muy variadas tal y como demuestran las librerías Apollo y Styled Components o por ejemplo el siguiente ejemplo donde podemos ver cómo se usan este tipo de funciones para traducir textos de nuestra aplicación:

```
const name = 'Gerardo';
const email = 'info@mail.com';

console.log(i18n`Hi ${name}, your email is ${email}`);

// Hola Gerardo, tu email es info@mail.com
```

. . .

Conclusiones

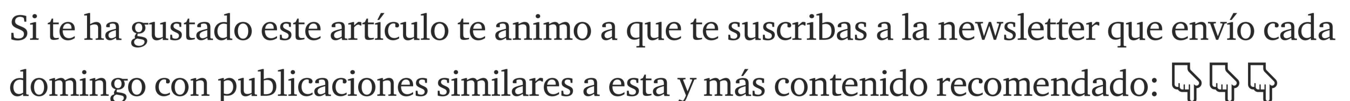
Como habéis podido ver, aunque al principio pueda ser una característica de Javascript que pase desapercibida, los *tagged template literals* nos dan mucha versatilidad a la hora de escribir código y se han vuelto cada vez más populares gracias a librerías como Apollo o los Styled Components.

Espero que este artículo os haya servido para tener una primera aproximación a ellos o reforzar conceptos como las *tag functions*.

. . .

¿Quieres recibir más artículos como este?

Si te ha gustado este artículo te animo a que te suscribas a la newsletter que envío cada domingo con publicaciones similares a esta y más contenido recomendado: 👇👇👇



Latte and code

Estás a punto de suscribirte a la newsletter de Latte and Code que recibirás cada domingo. — Los dos últimos artículos...

eepurl.us20.list-manage.com

)

JavaScript Javascript Tips Javascript Development Vanillajs Front End Development

About Help Legal