Lab5 - Eventos y formularios

Pepe Calderón, Fernando Sola, Daniel Ayala, Inma Hernández, Margarita Cruz, Carlos Arévalo, David Ruiz

Escuela Técnica Superior de
Ingeniería Informática

Índice

1. Objetivo 1
2. Introducción 2
3. Reaccionando a clicks 3
4. Otros eventos relevantes 5
4.1. mouseenter / mouseleave 5
4.2. focus / blur 6
4.3. change 6
4.4. submit 6
5. Validación de formularios 7
6. Módulos validadores 11
7. Actualización en GitHub 13

1 Objetivo

El objetivo de esta práctica es introducir la gestión básica de eventos mediante JavaScript, desarrollando código que se ejecute cuando ocurre una determinada acción en la página Web. Además, se presentarán los conceptos fundamentales sobre la validación de los formularios de la página.

El alumno aprenderá a:

- Asignar funciones al lanzamiento de determinados eventos.
- Conocer los tipos de eventos más comunes.
- Realizar validación de formularios en una aplicación Web.

2

Introducción

Hasta ahora hemos realizado cambios en las vistas de nuestra aplicación Web de manera proactiva: mediante nuestro código, la página ha sufrido modificaciones en el momento que nosotros hemos considerado más oportuno, normalmente, tras la carga de ésta. Sin embargo, buena parte de la experiencia de usuario se centra en reaccionar a las interacciones de éste con la página: hacer algo cuando se pulsa un botón, se selecciona un campo, se pone el ratón en una zona o se envía un formulario.

Este código, a priori, es imposible conocer de antemano cuándo se ejecutará, ya que depende de la interacción del usuario. La gestión de eventos en JavaScript consiste en desarrollar código asociado a cada una de estas situaciones, que se ejecutará cada vez que se cumpla una determinada condición.

Un aspecto muy relacionado es la validación de formularios en cliente: para evitar trabajo innecesario al servidor, se asocia un determinado código de validación al evento de envío de un formulario, cancelando el envío si el usuario ha introducido algún valor incorrecto.

En las siguientes secciones abordaremos estos aspectos y aprenderemos a gestionar los eventos más comunes de una aplicación Web.

3 Reaccionando a clicks

Uno de los eventos más comunes es `click`, que ocurre cuando el usuario hace click en un elemento determinado de la página, generalmente un botón.

Para hacer una prueba, añadiremos un botón a cualquier parte de nuestra vista principal. Es importante darle un `id`, ya que lo usaremos para acceder al botón en nuestro código JavaScript:

```
<button class="btn btn-primary" id="test-button">Press me!</button>
```

Es posible asociar directamente funciones que gestione los eventos relacionados con el botón en el código HTML, pero se desaconseja. En su lugar, accederemos al botón en nuestro código JS, y le asociaremos una función al evento `click`:

```
function main() {
    let button = document.getElementById("test-button");
    button.onclick = function (event) {
        alert("You've pressed the button!");
    };
}
```

Observe cómo la función asociada al evento `click` se asigna a un atributo llamado `onclick`. En general, las funciones asociadas a cualquier evento se asocian a un atributo que se llama igual que el evento, con el prefijo **on**. Pruebe a refrescar la página y comprobará que, al pulsar el botón, el navegador muestra un mensaje de alerta.

Además, las funciones que gestionan eventos reciben un parámetro de tipo Event}, al que se le suele llamar `event`. Este objeto Event puede usarse, entre otras cosas, para saber qué elemento concreto provocó el evento o detener la propagación del evento.

En el ejemplo anterior, se le ha asignado al atributo `onclick` del botón una función anónima, dado que la operación a realizar era simple. Si un evento tiene asociada una operación más compleja, que requiere de varias líneas de código, puede ser definida aparte y referenciada por su nombre.

Por ejemplo, podemos usar el parámetro `event` para acceder al nodo HTML que provocó el evento, y mostrar el texto que contiene:

```
function main() {
    let button = document.getElementById("test-button");
    button.onclick = clickHandler;
}

function clickHandler(event) {
  let target = event.target;
  let text = target.textContent;
  alert(text);
}
```

En este caso, cuando pulsemos el botón se nos mostrará el texto del propio botón, ya que es éste el elemento causante del evento:

4

Otros eventos relevantes

Pese a que `click` suele ser el evento más usado, hay muchos otros que merece la pena comentar. A continuación se detallan algunos de ellos:

4.1. mouseenter / mouseleave

Los eventos `mouseenter` y `mouseleave` se producen, respectivamente, cuando el cursor del mouse entra y sale de un determinado elemento. Usándolos en conjunto, es posible alterar un elemento cuando el usuario pone el cursor sobre él, y devolverlo a su estado original una vez lo mueve fuera.

Por ejemplo, podemos asignar a todas las tarjetas de la galería una función que, al poner el ratón sobre ellas, se ilumine su borde usando la propiedad CSS `border`:

```
function main() {
    let cards = document.querySelectorAll("div.card");
    for (let card of cards) {
        card.onmouseenter = handleMouseEnter;
        card.onmouseleave = handleMouseLeave;
    }
}

function handleMouseEnter(event) {
    let card = event.target;
    card.style.border = "2px solid blue"
}
```

```
function handleMouseLeave(event) {
    let card = event.target;
    card.style.border = "none";
}
```

4.2. focus / blur

Los eventos `focus` y `blur` están asociados a etiquetas de tipo `<input>`, y se lanzan cuando el elemento en cuestion recibe o pierde el foco de entrada, respectivamente.

4.3. change

El evento `change` también está asociado a etiquetas `<input>`, y se activa cuando el valor del campo ha cambiado. Generalmente, esto suele ocurrir cuando el elemento pierde el foco tras haber recibido cambios, o al pulsar Enter.

4.4. submit

El evento `submit` está asociado al envío de un formulario por parte de un usuario, usando el botón correspondiente. Este evento se suele usar para validar el formulario en cuestión cuando el usuario ha terminado de rellenar los campos, y se puede evitar el envío del formulario si algún campo introducido no es correcto. En la siguiente sección mostramos un ejemplo de uso.

5

Validación de formularios

Una aplicación Web suele contener uno o más formularios, que el usuario debe cumplimentar para enviar datos al servidor. Estos formularios deben ser validados en el navegador antes de su envío, para evitarle al servidor el procesamiento de formularios no válidos. Dado que los `<form>` envían un evento `submit` cuando el usuario envía el formulario, se suele asociar a este evento código de validación, que impide el envío del formulario si se producen errores.

Como ejemplo, realizaremos una validación del formulario de registro de nuevo usuario, ubicando en `register.html`. Para ello, crearemos un archivo `register.js` y lo vincularemos a la vista de registro:

```
<script src="js/register.js" type="module"></script>
```

Además, le daremos un `id` al formulario de registro, para poder acceder a él mediante JavaScript:

```
<form id="register-form">
  ...
</form>
```

En el archivo `register.js`, estableceremos el punto de entrada habitual:

```
"use strict";

function main() {
  ...
}
```

```
document.addEventListener("DOMContentLoaded", main);
```

Recordemos que la función `main()` se ejecuta cuando ha cargado todo el contenido de la página. En ese momento, podemos asignar una función al evento de envío del formulario:

```
function main() {
  let registerForm = document.getElementById("register-form");
  registerForm.onsubmit = handleSubmitRegister;
}

function handleSubmitRegister(event) {
  alert("Form sent!");
}
```

Si enviamos el formulario, comprobaremos que se emite la alerta al capturarse el evento de envío.

A continuación, podemos proceder a acceder a cada uno de los campos que queramos validar. Para ello, nos ayudaremos de los objetos `FormData`, que permiten encapsular un formulario para acceder a su contenido programáticamente:

```
function handleSubmitRegister(event) {
  let form = event.target;
  let formData = new FormData(form);
}
```

En este caso, `event.target` hace referencia al `<form>` que está siendo enviado. Podemos, entonces, construir un `FormData` a partir del formulario en cuestión. Con este objeto, podemos usar los métodos `formData.get()` para acceder a los campos introducidos por el usuario. Este método recibe el atributo `name` del input cuyo contenido se desea consultar:

```
let firstName = formData.get("firstName");
let lastName = formData.get("lastName");
let password = formData.get("password");
let password2 = formData.get("password2");
```

Suele ser útil definir al principio un array de errores, que podemos ir rellenando en el caso de que no se cumplan determinadas condiciones:

```
function handleSubmitRegister(event) {
    let form = event.target;
    let formData = new FormData(form);

    let errors = [];

    let firstName = formData.get("firstName");
```

```
    let lastName = formData.get("lastName");
    let password = formData.get("password");
    let password2 = formData.get("password2");

    if (firstName.length < 3 || lastName.length < 3) {
        errors.push("The first and last name should have more than 3
characters");
    }

    if (password !== password2) {
        errors.push("The passwords must match");
    }
}
```

Lógicamente, debemos informar al usuario de todos los errores ocurridos. Una manera habitual es definir un `<div>` dedicado a mostrar mensajes al usuario. En nuestro caso, crearemos un `<div id="errors">` en la parte superior de la página, antes del formulario en sí, que usaremos para mostrar estos errores.

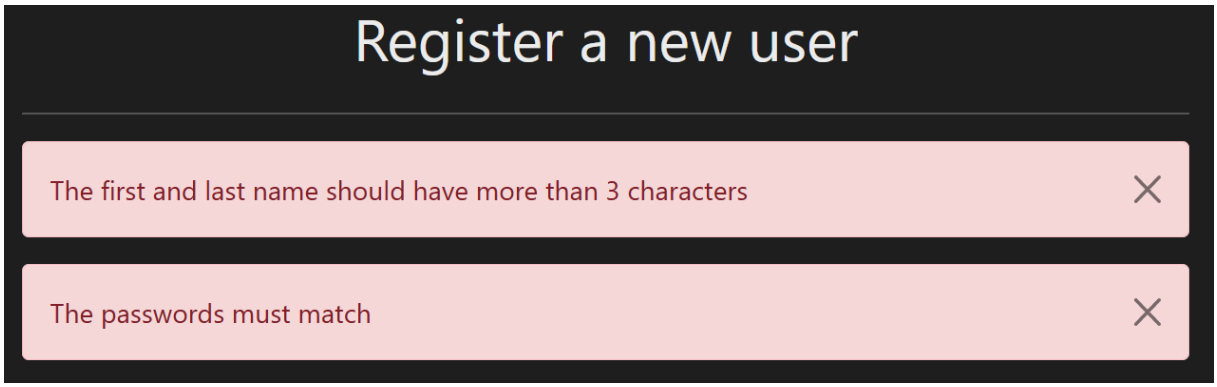
Para plasmar estos errores, existe un renderizador incluído por defecto en el proyecto, en el archivo `js/renderers/messages.js`. Este renderizador permite mostrar mensajes de información, error o éxito, que aparecerán automáticamente en cualquier `<div>` con `id=errors` que haya en la página:

```
import { messageRenderer } from "/js/renderers/messages.js";

function handleSubmitRegister(event) {
    let form = event.target;
    let formData = new FormData(form);

    let errors = [];

    // ...

    if (errors.length > 0) {
        event.preventDefault();
        let errorsDiv = document.getElementById("errors");
        errorsDiv.innerHTML = "";

        for (let error of errors) {
            messageRenderer.showErrorMessage(error);
        }
    }
}
```

Además, en el caso de que existan errores, deshabilitaremos el envío del formulario mediante `event.preventDefault()`:

El renderizador de mensajes, a diferencia de los demás, no devuelve un nodo HTML sino que coloca directamente el mensaje en el `<div id="errors">` que exista en la página.

Es importante limpiar los mensajes de error anteriores, para evitar que se acumulen entre diferentes envíos:



Si no existen errores, no se mostrará ningún mensaje de error y se enviará el formulario. En posteriores prácticas, veremos cómo realizar este envío mediante una petición POST con AJAX a la API del proyecto.

6

Módulos validadores

En la sección anterior, vimos que el código para validar un formulario se suele encapsular en una función, que accede a los campos relevantes para realizar una serie de comprobaciones, y genera una serie de errores que deben ser mostrados al usuario para su subsanación.

Sin embargo, por su propia naturaleza, estas funciones suelen tener una cierta longitud, que aumentan la longitud del archivo JavaScript asociado a la vista. Además, es posible que el código de validación del formulario deba ser reutilizado en varias vistas, cosa que no es posible si se hace de forma específica para una vista concreta.

La solución es encapsular la función de validación en un módulo validador. Estos módulos se encuentran en la carpeta `js/validators/` y, de manera similar a los renderizadores, exportan un objeto que puede ser importado en otras vistas para su uso. En el caso del registro, dado que la entidad en cuestión son los usuarios, crearemos un archivo `users.js` en la carpeta mencionada, con el siguiente contenido:

```
"use strict";

const userValidator = {

  validateRegister: function (formData) {
    ...
  }
};

export { userValidator };
```

En la función de validación del registro, incorporaremos el código anterior:

```
validateRegister: function (formData) {
  let errors = [];

    let firstName = formData.get("firstName");
    let lastName = formData.get("lastName");
    let password = formData.get("password");
    let password2 = formData.get("password2");

    if (firstName.length < 3 || lastName.length < 3) {
        errors.push("The first and last name should have more than 3
characters");
    }

    if (password !== password2) {
        errors.push("The passwords must match");
    }

  return errors;
}
```

De este modo, el módulo de validación ofrece un método para, dado un objeto `FormData` concerniente al formulario de registro, devolver un array de errores relativos al formulario. Esto permite simplificar en gran medida el código de la vista en `register.js`:

```
import { messageRenderer } from "/js/renderers/messages.js";
import { userValidator } from "/js/validators/users.js";

function main() {
  let registerForm = document.getElementById("register-form");
  registerForm.onsubmit = handleSubmitRegister;
}

function handleSubmitRegister(event) {
  event.preventDefault();
  let form = event.target;
  let formData = new FormData(form);

  let errors = userValidator.validateRegister(formData);

  if (errors.length > 0) {
    let errorsDiv = document.getElementById("errors");
    errorsDiv.innerHTML = "";

    for (let error of errors) {
      messageRenderer.showErrorMessage(error);
    }
  }
}
```

7

Actualización en GitHub

Actualice su proyecto en GitHub con los cambios hechos durante esta sesión. Recuerde los comandos relevantes:

- `git add .` añade los cambios efectuados en todos los archivos al próximo `commit` a efectuar.
- `git commit -m "mensaje"` crea un nuevo `commit` con los cambios efectuados a los archivos añadidos con el comando anterior, y con el mensaje indicado.
- `git push` actualiza el repositorio remoto con los cambios efectuados.

> **Info**
>
> Versión PDF disponible