AJAX

Sitio: Aula Virtual do IES de Teis
Curso: Desarrollo web en entorno servidor 2023-24 (DAW-DUAL-A)
Libro: AJAX

Impreso por: Joaquín Lafuente Espino
Data: Sábado, 16 de Marzo de 2024, 20:43

Táboa de contidos

1. AJAX ¿Qué es?

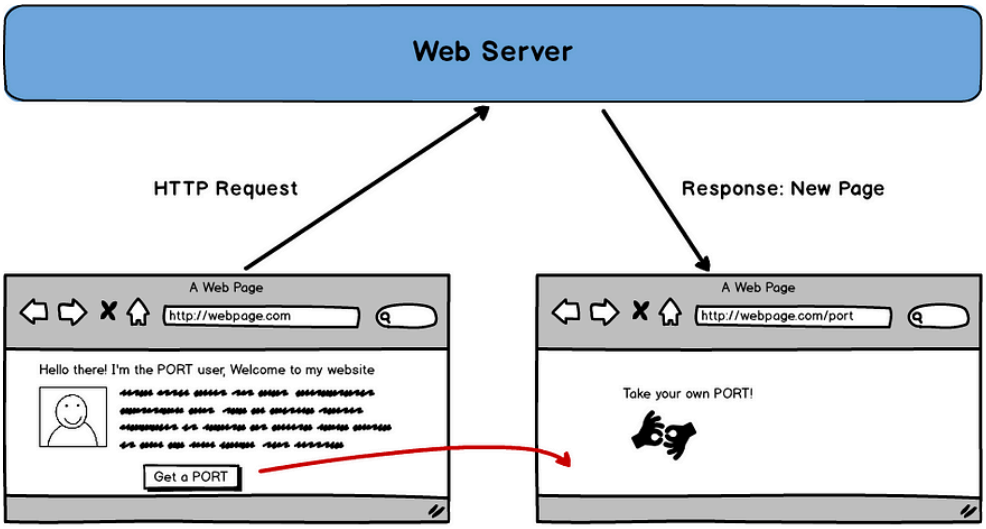
AJAX viene del inglés **Asynchronous Javascript And XML**.

Describe un modo de utilizar conjuntamente varias tecnologías existentes: **HTML, CSS, JavaScript, DOM, XML, XSLT y HTTP**

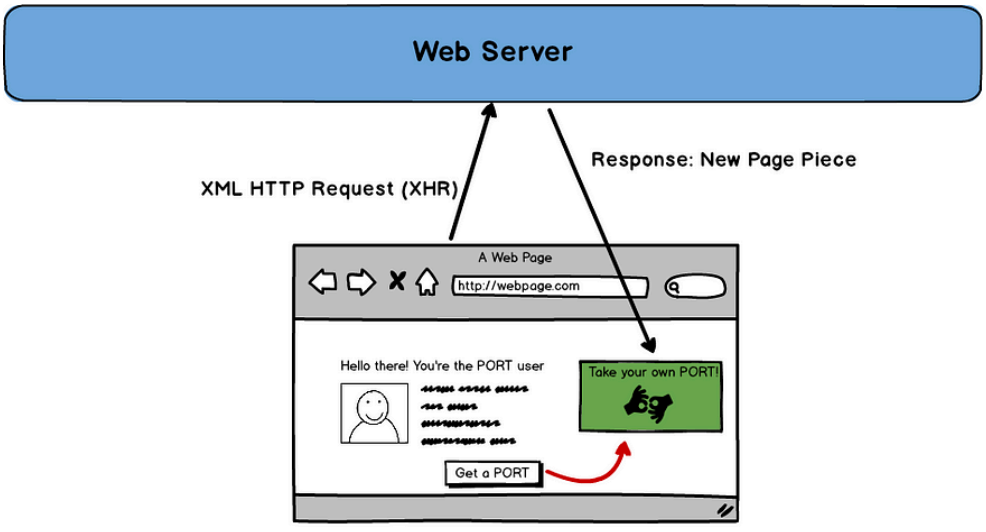
Permite que las aplicaciones web sean capaces de actualizar una parte del DOM sin tener que volver a cargar la página completa. Esto crea aplicaciones más rápidas y con mejor respuesta a las acciones del usuario.

Permite realizar peticiones HTTP al lado servidor, obtener sus datos (XML, JSON, TEXT, etc) y modificar una sección del HTML con los datos obtenidos.

Las aplicaciones **SPA (Single Page Application)** utilizan AJAX para obtener los datos que muestran.



Classical SSR Scenario (Server Side Rendering): Full Page Response



Typical Partial Rendering (AJAX): XMLHttpRequest - Response

Fuente: https://davidjguru.medium.com/single-page-application-un-viaje-a-las-spa-a-trav%C3%A9s-de-angular-y-javascript-337a2d18532

1.1. JavaScript

JavaScript es un lenguaje de programación multiplataforma orientado a objetos que se utiliza para hacer que las páginas web sean interactivas

JavaScript de lado del cliente proporciona objetos para controlar un navegador y su *Modelo de objetos de documento* (DOM por *Document Object Model*). Por ejemplo, permite que una aplicación coloque elementos en un formulario HTML y responda a eventos del usuario, como clics del ratón, formularios para introducir de datos y navegación entre páginas.

También hay versiones de JavaScript de lado del servidor más avanzadas, como Node.js.

- El objeto `window` representa la ventana que contiene un documento DOM
- La propiedad `document` apunta al DOM document cargado en esa ventana.
- A través del objeto DOM `window`, se puede utilizar su evento `load` que se dispara al final del proceso de carga del documento. En este punto, todos los objetos del documento son DOM, y todas las imágenes y sub-frames han terminado de cargarse.

1.2. Métodos de realización de llamadas AJAX

- XMLHttpRequest: la primera versión que soportaba AJAX.
- jQuery y $.ajax: La librería de JavaScript jQuery tiene su propia función $.ajax para realizar llamadas con una sintaxis más simplificada, pero sigue usando XMLHttpRequest de forma subyacente.
- API Fetch: API de JavaScript del 2015 con sintaxis más concisa y clara.

Métodos HTTP

Hasta el momento, con <form> HTML solo hemos trabajado con 2 métodos HTTP: GET para obtener datos y POST para llevar a cabo modificaciones en el servidor o enviar datos sensibles.

HTTP define varios métodos diferentes, algunos de los cuales son muy imporantes en una API REST.

La convención que parece ser utilizada en su mayoría es la siguiente.

- `GET` se utilizarán para obtener datos
- `POST` se utilizará para crear datos
- `DELETE` se utilizarán para eliminar los datos
- `PUT` se utilizará para modificar todos los datos
- `PATCH` se utilizarán para modificar parcialmente los datos

URL REST

Por convención las APIs web REST utilizan URLs que no contengan un verbo. Por ejemplo, si se trabaja con una entidad libro, las rutas `/addLibro`, `/readLibro`, o `/updateLibro` no serían correctas. Si cada petición HTTP lleva ya un método implícito, sería redundante.

Serán utilizadas rutas como:

- `/libros` con el método HTTP `POST` para crear un libro.
- `/libros/:id` con el método HTTP `GET` para obtener datos de un libro con el id :id. Por ejemplo: /libros/1
- `/libros/:id` con el método HTTP `PATCH` para modificar parcialmente los datos del libro con el id :id. Por ejemplo: /libros/2 y los campos de modificación irán en el cuerpo de la petición HTTP

1.3. API de Fetch

El API es una interfaz JavaScript para acceder y manipular comunicaciones con HTTP

Tiene además un método global que permiter acceder a recursos de forma asíncrona también llamado `fetch()`

El método fetch:

- Su versión más simple toma un argumento (la ruta del recurso que quieres obtener) y devuelve un objeto Promise o promesa (un objeto que representa la terminación o el fracaso de una operación asíncrona) conteniendo la respuesta, un objeto `Response`.
- Un objeto `Response` modela una respuesta HTTP. Para extraer el contenido en el cuerpo del JSON desde la respuesta, se usa el método `json()`. Existen otros posibles métodos dependiendo del tipo de respuesta esperada.
- Solo será rechazado ante un fallo de red o si algo impidió completar la solicitud. Si el servidor devuelve un 404 o un 500 devolverá un estado llamado "ok" con el valor false
- Por defecto, `fetch` no enviará ni recibirá cookies del servidor. Si se desea enviarlas, hay que usar la opción init

Una petición GET básica con el API de fetch usando promesas, con una respuesta en formato JSON, tendría esta forma:

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

El método `fetch()` puede aceptar opcionalmente un segundo parámetro, un objeto `init` que permite controlar un numero de diferentes ajustes. Un ejemplo para enviar un objeto JSON con fetch es el siguiente:

```
var url = "https://example.com/profile";
var data = { username: "example" }; //datos a enviar

fetch(url, {
  method: "POST", // el método HTTP con el que se enviará la petición
  body: JSON.stringify(data), // data can be `string` or {object}!
  headers: {
    "Content-Type": "application/json", //tipo de datos enviados -> JSON
  },
})
  .then((res) => res.json()) //cuando se reciba la respuesta, la pasamos a formato JSON
  .catch((error) => console.error("Error:", error)) //Si hay un error, lo mostramos en consola js
  .then((response) => console.log("Success:", response)); //Si ha habido éxito, la mostramos en consola js
```

Más información:
https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch
https://es.javascript.info/fetch
https://lenguajejs.com/javascript/asincronia/async-await/

Vamos a ver un par de ejemplos muy sencillo en el repositorio: https://github.com/dudwcs/UD7_hola_mundo_js.git

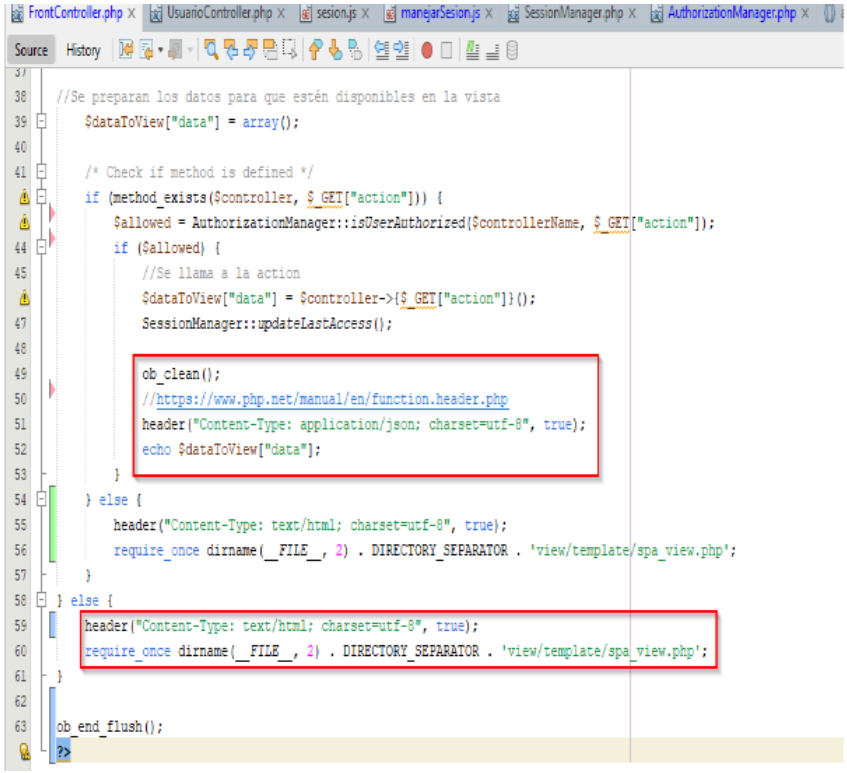
1.4. Ejemplo básico de una SPA

Vamos a ver un ejemplo de cómo podríamos imitar el funcionamiento de una SPA con parte de la infraestructura que tenemos organizada en MVC:

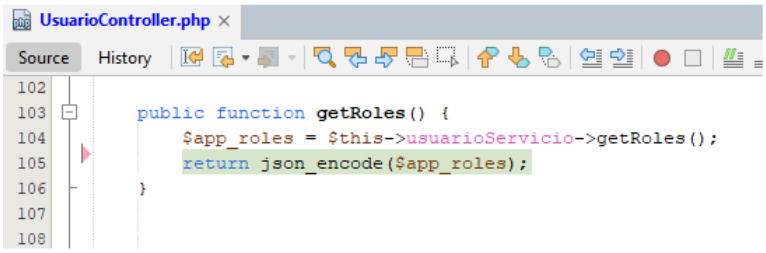
Está disponible en la URL: https://github.com/dudwcs/UD7_Ejemplo_SPA.git

El **FrontController.php** se ha modificado para que en lugar de crear una página HTML completa con header + mainView + footer, sirva una página **spa_view.php** si no recibe parámetros controller ni action.

Si los recibe, llamará al método correspondiente, pero indicando en la cabecera que la respuesta esta vez será en formato JSON y se escribirá directamente en la salida:

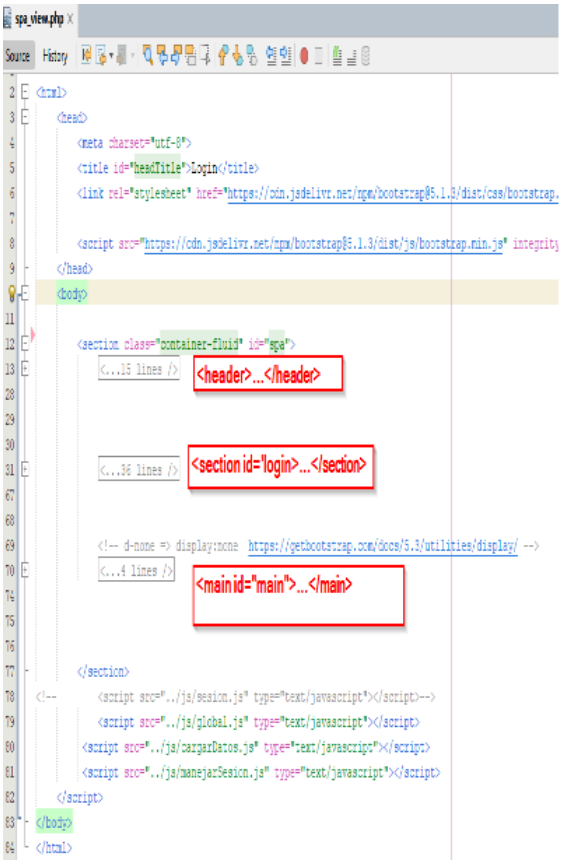


En los métodos de cada controlador devolveremos los datos en formato JSON con la función json_encode de PHP:



spa_view.php es una página HTML con los siguientes apartados destacados:

- header
- section login: se mostrará al inicio
- main: estará inicialmente oculta y se irá rellenando con los datos que corresponda en función de la interacción con el usuario.
- scripts que realizarán las llamadas al servidor:
  - global.js: las variables globales y la función onceLoaded() que establecerá los escuchadores de eventos de la sección login
  - manejarSesion.js: las funciones para iniciar y cerrar sesión (de usuario) y sus funciones auxiliares
  - cargarDatos.js con las funciones que permitirán obtener datos del servidor y sus funciones auxiliares



En global.js, además de establecer las funciones de algunos eventos se llama a getRoles(), definido en cargarDatos.js.

getRoles() realiza una petición GET con el API Fetch a UsuarioController para obtener los roles en formato JSON. Cuando los recibe, crea los elementos <option> que se mostrarán en la lista desplegable del apartado de login.

La función login() de manejarSesion.js simula una petición POST al método login de UsuarioController, también con el API Fetch. Para que los datos enviados estén disponible en $_POST, es necesario enviarlos con FormData().

```
let login_url = "?controller=Usuario&action=login";

//preparamos los datos que se enviarían al servidor como si se enviasen  por POST  desde el formulario
const data = new FormData();
data.append('email', email);
data.append('pwd', pwd);
data.append('rol', rol);

const request = new Request(base_url + login_url, {
    method: "POST",
    body: data
});
```

En el servidor ya no tiene sentido utilizar redirecciones, pues causarían la recarga de toda la página, por lo que se perdería el objeto de una SPA.

Si hay algún problema, devolveremos un JSON {'error': true}. En caso contrario devolveremos un objeto { userId: 1, email: "user1@edu.es" }

```php
0 references | 0 overrides
public function login()
{
    //Para simplificar la implementación del ejemplo de SPA vamos a obviar la redirección en caso de qu


    $this->page_title = 'Inicio de sesión';
    $this->view = self::VIEW_FOLDER . DIRECTORY_SEPARATOR . 'login';

    if (isset($_POST["email"]) && isset($_POST["pwd"]) && isset($_POST["rol"])) {
        $email = $_POST["email"];
        $pwd = $_POST["pwd"];
        $rolId = $_POST["rol"];

        $userResult = $this->usuarioServicio->login($email, $pwd, $rolId);

        if ($userResult == null) {
            //400 Bad Request
            http_response_code(400);
            $response["error"] = true;
            return json_encode($response);
        } else {

            SessionManager::iniciarSesion();
            $_SESSION["userId"] = $userResult->getId();
            $_SESSION["email"] = $userResult->getEmail();
            $_SESSION["roleId"] = $rolId;
            $_SESSION["ultimoAcceso"] = time();


            $response["userId"] = $userResult->getId();
            $response["email"] = $userResult->getEmail();
            return json_encode($response);
        }
    } else {
        //400 Bad Request
        http_response_code(400);
        $response["error"] = true;
        return json_encode($response);
    }
}
```

1.5. ¿Cómo recuperar objetos enviados en formato JSON desde PHP?

En el login de la Actividad 6.4 vimos como enviar datos desde un formulario creando un objeto de tipo FormData() y provocando que los datos enviados estén disponibles en $_POST.

Si en su lugar deseamos enviar datos en formato JSON, los datos enviados no estarán disponibles través de las variables $_POST, sino a través de un flujo de entrada llamado php://input . Se trata de un flujo de solo lectura que permite leer datos del cuerpo de la petición HTTP.

En su lugar deberemos recuperar los datos ayudándonos de

- file_get_contents: leerá el contenido del flujo en un único string
- json_decode(): interpretará el string para transformarlo a un objeto o a un array asociativo si se usa el 2º argumento true.

```
$data = json_decode(file_get_contents("php://input"), true);
```

2. Diálogos modales con Bootsrap

Un diálogo modal es una ventana que se abre en la página web por encima del contenido activo y que impiden interactuar con la página hasta que se cierre la misma

En Bootstrap 5 podéis encontrar diferentes ejemplos de diálogos modales en la URL: https://getbootstrap.com/docs/5.3/components/modal/

Para utilizarlos, comprobaremos si tenemos incluido el CDN de JS disponible aquí: https://getbootstrap.com/docs/5.0/getting-started/introduction/ y lo situaremos lo más cerca posible y antes de la etiqueta </body> y antes, a su vez, de nuestras propias etiquetas <script>. Bootstrap 5 ya no incluye jQuery, pero podrían ser utilizados conjuntamente.

- Los modales se construyen con HTML, CSS y JavaScript.
- Se cierran por defecto cuando se hace clic fuera del modal
- No se permiten modales anidados en Bootstrap
- Usan atributos de HTML data-* para permitir asociar datos a un elemento HTML de forma estandarizada. Por ejemplo, para cerrar el modal haciendo clic fuera del modal, se usa

```
data-bs-dismiss="modal"
```

- Es posible abrirlos sin JavaScript añadiendo al botón que disparará su aparación los atributos **data-bs-toggle="model"** y **data-bs-target="#el_id_del_modal"** como se ve en la documentación.

También es posible interactuar con ellos a través de JavaScript:

```
var myModal = new bootstrap.Modal(document.getElementById('myModal'), options)
```

options se pueden consultar en la documentación: https://getbootstrap.com/docs/5.3/components/modal/#options

y los métodos:

- `myModal.show()`: Muestra el modal
- `myModal.hide()`: Oculta el modal

Hay que tener cuidado porque `myModal.hide() y modal.show()` devuelven el control a la siguiente línea sin asegurar que se haya cerrado o mostrado el modal. Para asegurarse de que se haya cerrado, habría que usar los eventos: `hidden.bs.modal y shown.bs.modal`

```
var myModalEl = document.getElementById('myModal')
```

```
myModalEl.addEventListener('hidden.bs.modal',
    function (event) {
  // do something...
```

```
})
```

Los eventos del modal de Bootstrap y su significado se pueden encontrar en https://getbootstrap.com/docs/5.3/components/modal/#events

Tenéis esta y más información en el apartado de Usage: https://getbootstrap.com/docs/5.3/components/modal/#usage

2.1. Función auxiliar para usar diálogos modales

Basándonos en los modales de Bootstrap, vamos a ver una función genérica de Javascript que nos puede servir de base para añadir modales a nuestra SPA:

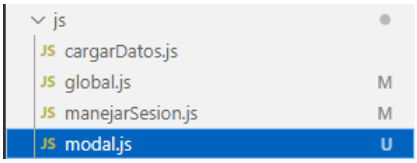
1- Nos aseguramos de que tenemos las hojas de estilo y javascript que provee Bootstrap dentro del <head> de nuestro html:

```
<head>
    <meta charset="utf-8">
    <title id="headTitle">Login</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
        integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.min.js"
        integrity="sha384-QJHtvGhmr9XOIpI6YVutG+2QOK9T+ZnN4kzFN1RtK3zEFEIsxhlmWl5/YESvpZ13"
        crossorigin="anonymous"></script>
</head>
```

2- Añadimos el marcado html a **spa_view.php**

```
        <main id="main" class="d-none">
            <!-- Aquí la vista particular -->
            <h2>Esto es el main</h2>

        </main>


        <!-- Modal -->
        <div class="modal fade" id="spa_modal" tabindex="-1" aria-labelledby="exampleModalLabel" aria-hidden="true">
            <div class="modal-dialog">
                <div class="modal-content">
                    <div class="modal-header">
                        <h5 class="modal-title" id="modal_title">Modal title</h5>
                        <button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>
                    </div>
                    <div class="modal-body" id='modal_msg'>

                    </div>
                    <div class="modal-footer">
                        <button type="button" class="btn btn-secondary" data-bs-dismiss="modal"
                            id='opt_cancel'>Cancelar</button>
                        <button type="button" class="btn btn-primary" id='opt_ok'>Aceptar</button>
                    </div>
                </div>
            </div>
        </div>
```

3- Añadimos el fichero **modal.js** dentro de js



4- Añadimos el fichero modal.js a la lista de scripts en spa_view.php

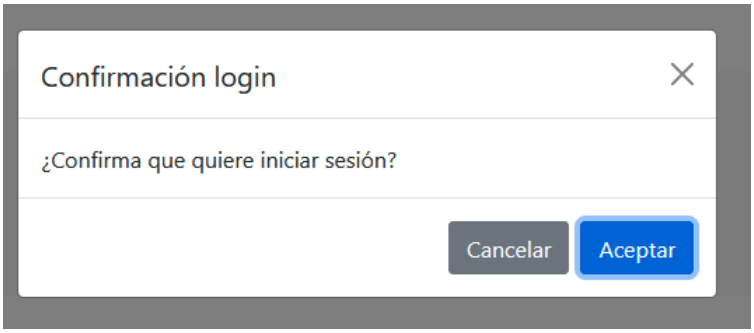
```
    </section>

    <script src="../js/global.js" type="text/javascript"></script>
    <script src="../js/modal.js" type="text/javascript"></script>
    <script src="../js/cargarDatos.js" type="text/javascript"></script>
    <script src="../js/manejarSesion.js" type="text/javascript"></script>
    </script>
</body>

</html>
```

5- modal.js contiene la función auxiliar showModal(...):

```
1   const OK_TEXT = "Aceptar";
2   const CANCEL_TEXT = "Cancelar";
3
4   /**
5    *  Muestra un modal con el id especificado (sin #)
6    * @param {string} modal_id
7    * @param {string} title Titulo del modal
8    * @param {string}  msg Mensaje con la pregunta que se planteará al usuario
9    * @param {string} opt_ok_text Texto a mostrar en el botón de Aceptar. Si no existe, se mostrará el
     contenido en el html inicialmente.
10   * @param {string} opt_cancel_text Texto a mostrar en el botón de Cancelar. Si no existe, se mostrará el
     contenido en el html inicialmente.
11   * @param {function} opt_ok_function Función a ejecutar si el usuario ha hecho clic en el botón de
     aceptar. Se deberá ejecutar después de cerrar el diálogo. Si no se aporta una función, simplemente se
     cerrará el diálogo.
12   * @param {function} opt_cancel_function Función a ejecutar si el usuario ha hecho clic en el botón de
     cancelar. Se deberá ejecutar después de cerrar el diálogo.  Si no se aporta una función, simplemente se
     cerrará el diálogo.
13
14   */
15  function showModal(modal_id, title, msg,
16      opt_ok_text = null,
17      opt_cancel_text = null,
18      opt_ok_function = null,
19      opt_cancel_function = null) {
20
```

Vamos a ver un ejemplo de uso de un modal de Bootstrap con esta función showModal en el repositorio https://github.com/dudwcs/UD7_Ejemplo_SPA_modal.git.

En este ejemplo veremos cómo se pide confirmación al usuario antes de iniciar sesión con loginJSON.



3. Opciones de validación en el lado servidor

Un recurso sobre validación de formularios en URL https://developer.mozilla.org/es/docs/Learn/Forms/Form_validation desde la sección ¿Qué es la validación de formularios? En dicha URL se hablaba de la importancia de **combinar la validación del lado cliente con la validación en el lado servidor.**

En PHP contamos con funciones que pueden ayudarnos a hacer comprobaciones con los datos llegados desde el cliente como:

FUNCIONES DE FILTRADO

- filter_has_var — Comprueba si existe una variable de un tipo concreto. El tipo puede ser: `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER`, or `INPUT_ENV` que hacen alusión a los arrays superglobales.
- filter_input — Toma (de los arrays superglobales) una variable por su nombre y opcionalmente la filtra. Si se omite el filtro, se utilizará `FILTER_DEFAULT`, que es equivalente a `FILTER_UNSAFE_RAW`. Esto resultará en que no se realice ningún tipo de filtro de forma. filter_input devuelve el valor de la variable pedida en caso de éxito, `false` si el filtro falla o `null` si la variable no está definida.predeterminada.
- filter_var — Toma una variable externa concreta por su nombre y opcionalmente la filtra. Retorna los datos filtrados o `false` si el filtro falla.

TIPOS DE FILTROS

Existen diferentes tipos de filtros:
Filtros de validación: Comprueban que los datos cumplen con un determinado patrón.

- `FILTER_VALIDATE_BOOLEAN`, `FILTER_VALIDATE_BOOL`
- `FILTER_VALIDATE_DOMAIN`
- `FILTER_VALIDATE_EMAIL`
- `FILTER_VALIDATE_FLOAT`
- `FILTER_VALIDATE_INT`
- `FILTER_VALIDATE_IP`
- `FILTER_VALIDATE_MAC`
- `FILTER_VALIDATE_REGEXP`
- `FILTER_VALIDATE_URL`

Filtros de saneamineto (sanitize filters): Limpiará los datos, de modo que los modificará eliminando los caracteres no deseados. Por ejemplo, pasándole `FILTER_SANITIZE_EMAIL` eliminará los caracteres que no son apropiados para una dirección de correo electrónico. Sin embargo, no valida los datos.

- `FILTER_SANITIZE_EMAIL: Elimina todos los caracteres menos letras, dígitos y !#$%&'*+-=?^_`{|}~@.[].`
- `FILTER_SANITIZE_ENCODED: Sanea una cadena para que pueda formar parte de una URL. Espacios son sustituidos por %20. Más sustituciones:https://www.w3schools.com/tags/ref_urlencode.ASP`
- `FILTER_SANITIZE_ADD_SLASHES: Escapa con \ los caracteres ' " \ y NUL`
- `FILTER_SANITIZE_NUMBER_FLOAT`
- `FILTER_SANITIZE_NUMBER_INT`
- `FILTER_SANITIZE_SPECIAL_CHARS: Escapa caracteres HTML '"<>& y caracteres con valores ASCII menores que 32, opcionalmente elimina o codifica caracteres especiales.`
- `FILTER_SANITIZE_FULL_SPECIAL_CHARS FILTER_SANITIZE_STRING => usar mejor htmlspecialchars()`
- `FILTER_SANITIZE_URL: Elimina todos los caracteres excepto letras, dígitos y $-_.+!*'(),{}|\\^~[]`<>#%";/?:@&=.`

```
<body>
```

```
        <?php
```

```
        $term_html = filter_input(INPUT_GET, 'term', FILTER_SANITIZE_SPECIAL_CHARS);
```

```
        $term_url_encoded = filter_input(INPUT_GET, 'term', FILTER_SANITIZE_ENCODED);
```

```
        ?>
```

```
        <form  method="get">
```

```
            <label for="term"> Search </label>
```

```
            <input type="search" name="term" id="term" value="<?php echo $term_html ?>">
```

```
            <input type="submit" value="Search">
```

```
        </form>
```

```
        <?php
```

```
        if (null !== $term_html) {

            echo "El resultado es <mark> $term_html </mark>.<br/>";

        }

        if (null !== $term_url_encoded) {

            echo "El resultado es <mark> $term_url_encoded </mark>.<br/>";

        }
/*Se puede ver la diferencia en el código fuente con CTRL+ U*/

        ?>

    </body>
```

En la documentación veréis que los filtros tienen opciones y flags.
Las opciones se pueden pasar en un array asociativo. Por ejemplo para validar una expresión regular:

```
    $variable= filter_input(INPUT_POST, "nombre_variable", FILTER_VALIDATE_REGEXP, array("options" => array("regexp" =>
$password_regex )));
```

O para comprobar un rango de enteros:

```
$int_a = '1';

$options = array(
    'options' => array(
                    'min_range' => 0,
                    'max_range' => 3,
                    )
);
if (filter_var($int_a, FILTER_VALIDATE_INT, $options) !== FALSE) {
    echo "Este entero (int_a) es válido (entre 0 y 3).\n";
}
```

Las *banderas* o *flags* se usan opcionalmente tanto con la validación como con el saneamiento para adaptar el comportamiento según las necesidades. Por ejemplo, para permitir introducir una IPv4 o una IPv6 se puede añadir al ejemplo anterior con una disyunción lógica de las banderas FILTER_FLAG_IPV4 y FILTER_FLAG_IPV6 :

```
  $valid = filter_input(INPUT_GET, 'term', FILTER_VALIDATE_IP, FILTER_FLAG_IPV4| FILTER_FLAG_IPV6);

        if($valid!==false && $valid !==null){

            echo "La IP es válida: $valid";

        }

        else{

            echo "No se ha introducido una IP válida";

        }
```

Como norma general, para las validaciones de entrada, no se aconseja usar los filtros de saneamiento, pues podría darse el caso de dar por válidas entradas que en realidad no lo son. Es una mejor práctica informar al usuario de lo que debe cambiar, para que sea él mismo quien corrija la entrada.

Si los arrays superglobales se modifican en el código, filter_input no se percata. Sin embargo, filter_var sí.

```
$_GET["term"] = 'algo@algo.com';

$new_get_filter_input = filter_input(INPUT_GET, 'term', FILTER_VALIDATE_EMAIL);
echo '<br/> new_get_filter_input no se encuentra: ' . $new_get_filter_input .'<br/>';
var_dump($new_get_filter_input);

$new_get_filter_var = filter_var($_GET['term'], FILTER_VALIDATE_EMAIL);
echo '<br/> new_get_filter_var sí se encuentra: ' . $new_get_filter_var. '<br/>';
var_dump($new_get_filter_var);
```

Filtros con ejemplos en la URL:
https://www.php.net/manual/es/filter.examples.php
https://www.php.net/manual/es/intro.filter.php