Desarrollo Web en entorno servidor

Unidad 5:
Introducción a AJAX

Tabla de contenido

1. Separación de las lógicas

Como comentábamos en el primer tema, una manera de construir aplicaciones web es utilizar una arquitectura a tres capas (o lógicas):

- Lógica de presentación: Parte de la aplicación que se ocupa de mostrar la información al usuario y permite interactuar con el sistema al usuario.
- Lógica de negocio: Es la lógica especifica de la aplicación, donde realmente se desarrolla la funcionalidad de esta. Procesa la información que introduce el usuario y maneja la base de datos.
- Lógica de datos: Es la lógica que gestiona la base de datos, dando debida respuesta a las peticiones recibidas desde la lógica de negocios.

Figura 1. –Arquitectura en tres capas

Tras haber construido nuestra primera aplicación web nos encontramos con un problema muy evidente, **las lógicas de nuestra aplicación de pedidos están mezcladas.** Vemos que en multitud de páginas usamos etiquetas HTML, mezcladas con bloques de código PHP y, a su vez, código HTML que generamos en PHP a través de echo.

Esto hace que nuestro código sea muy poco reutilizable para otras aplicaciones web que tuviéramos que desarrollar **y dificulta enormemente las labores de mantenimiento y actualización** de nuestra aplicación.

Para resolver estos problemas, podemos plantear varios enfoques.

El primero es refactorizar nuestra aplicación para separar adecuadamente las lógicas entre sí.

El segundo es utilizar una arquitectura del tipo Modelo Vista Controlador (MVC[1]). Para ello, el primer paso sería aislar la salida en funciones específicas que se ocuparan exclusivamente de la salida.

---

[1] El MVC define en qué capas dividimos nuestra aplicación, pero además detalla las responsabilidades exactas de cada capa y la forma que tienen de relacionarse entre sí. Si programamos según esta arquitectura, estamos dividiendo la aplicación en tres capas, pero no al contrario: podemos programar en 3 capas sin necesidad de seguir esta arquitectura.

La división en capas de presentación, negocio y datos no es la misma que propone MVC. La capa Modelo incluiría la lógica de negocio y datos, la Vista incluiría exclusivamente el interfaz externo (una parte de la presentación), y el papel de la capa Controlador será una mezcla de presentación y lógica de negocio.

Mejor aún, se podría utilizar una librería de plantillas. Esta arquitectura la veremos en los temas dedicados al framework Symfony.

El tercer enfoque consiste en llevar la lógica de presentación al cliente. Al fin y al cabo, en el cliente es donde se muestra la información. El lenguaje más extendido en el lado del cliente es JavaScript, que se ejecuta dentro del navegador.

AJAX[2] es una técnica de diseño de aplicaciones web que se basa en hacer peticiones al servidor desde JavaScript. En los capítulos anteriores la salida de la página web se genera por completo en el servidor. Por ejemplo, la tabla de productos de la aplicación de pedidos.

En AJAX, simplificando, el servidor devuelve solo los datos que hay que mostrar. Es decir, los datos de los productos. En el cliente, la interfaz se actualiza según los datos recibidos, también utilizando JavaScript. De esta manera se desacopla la lógica de negocio de la de presentación. Tiene varias ventajas:

- El código es más fácil de mantener y modificar, al no estar mezclado.
- El código del servidor se puede reutilizar con otros clientes, como una aplicación de móvil.
- Se puede trabajar en la parte del cliente y en la del servidor en paralelo.

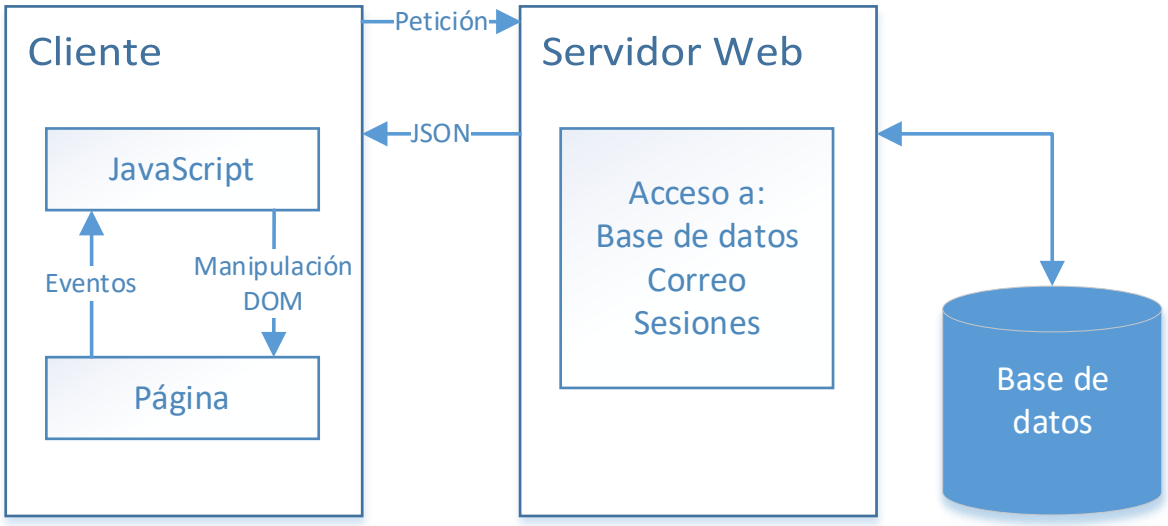

Figura 2 - Diagrama de aplicación con AJAX

Esto nos permite construir aplicaciones web de una sola página, ya que todas las peticiones al servidor, salvo la inicial, se hacen de forma dinámica utilizando AJAX. Las peticiones al servidor web son respondidas por esto mediante JSON[3].

Siguiendo este enfoque, la aplicación del tema anterior quedará dividida en una página que gestiona la base de datos, bd.php, las páginas que gestionan las peticiones del cliente (hacer login, añadir al carrito, …) y una única página con las vistas del lado del cliente.

---

[2] AJAX: Asyncronous JavaScript and XML. Técnica de diseño de aplicaciones web que se basa en realizar peticiones al servidor desde JavaScript.

[3] JSON: JavaScript ObjectNotation. Lenguaje para intercambio de datos en formato texto muy extendido en desarrollo web.

2. Tecnologías y librerías asociadas

El código JavaScript también llega al cliente desde el servidor incluido en páginas HTML. Se ejecuta en el navegador y puede manipular la página para cambiar su aspecto, mostrar mensajes de alerta y establecer comunicaciones con el servidor.

Para modificar la página web, JavaScript utiliza el DOM, un estándar del W3C. Es una interfaz que permite acceder y modificar el contenido, la estructura y el estilo de un documento HTML. Los documentos se representan mediante una estructura de árbol en la que los nodos son los elementos HTML. Los elementos forman parte de una jerarquía y se puede hablar de elementos padres, hijo, descendientes y antecesores.

```
1:  <!DOCTYPE html>
2:  <html>
3:      <head>
4:          <title>Ejemplo DOM</title>
5:      </head>
6:      <body>
7:      </body>
8:  </html>
```

El elemento html es el nodo raíz, y es el padre de head y body. También es antecesor del resto de elementos de la página. De la misma manera, title es hijo de head y descendiente de head. Desde JavaScript se puede modificar el contenido de los elementos, sus atributos y su estilo.

En el siguiente cuadro, resumimos las funciones que usaremos a lo largo del capítulo para trabajar con el DOM

Función	Descripción
document.createElement(<etiqueta>)	Crea un nuevo elemento
document.getElementByld(id)	Devuelve el elemento de la página que tiene ese id
elem.appendChild(elem_hijo)	Inserta elem_hijo como hijo de elem
elem.innerHTML	Propiedad que representa el contenido de un elemento

Tabla 1.- Funciones y atributos básicos para manipular el DOM

3. Obtención remota de información

Hasta ahora, hemos visto que se solicita una página servidor:

- Cuando se accede una página introduciendo la URL en navegador.
- Al seguir un vínculo
- Al enviar un formulario.
- Como consecuencia de una redirección.

En todos los casos, **el navegador cambia el contenido que esté mostrando con la respuesta que obtiene del servidor**. Este modelo es bastante restrictivo y condiciona el diseño de las páginas, ya que **cada interacción con el servidor requiere que este acabe devolviendo una página web completa**.

Es posible superar esta limitación utilizando JavaScript, en concreto AJAX. Por ejemplo, una página web puede incluir un script que cada minuto consulte las últimas noticias y muestre los titulares en una sección lateral. El script se encarga de obtener los datos y de modificar el contenido solo de esa sección, sin tener que recargar la página.

El elemento central de AJAX es el objeto javascript XMLHttpRequest, que representa una petición al servidor. Utilizándolo es posible hacer una petición al servidor sin que su respuesta se muestre automáticamente como una nueva página en el navegador. En lugar de eso, la respuesta se procesa mediante JavaScript.

A pesar de tener la palabra "XML" en su nombre, se puede operar sobre cualquier dato, no solo en formato XML. Admite otros formatos además de Http, como por ejemplo file y ftp. Fue diseñado por Microsoft y adoptado por Mozilla, Apple y Google. Actualmente es un estándar de la W3C.

Para crear una petición, es decir un objeto XMLHttpRequest, basta con llamar a su constructor, el cual no tiene parámetros:

```
var xhttp = new XMLHttpRequest();
```

Por otra parte, XMLHttpRequest tiene varios métodos y propiedades.

Los métodos básicos para llevar a cabo una petición son:

- `open(método, destino, [tipo])`. Recibe el método HTTP con el que se realiza la petición, la ruta que se solicita y el tipo de petición que se realiza (síncrona o asíncrona). El tipo por defecto de la petición es asíncrona. Para ello, el tercer argumento toma el valor TRUE.
- `send([params])`. Realiza el envío. Tiene un argumento opcional para añadir parámetros a la petición.
- `setRequestHeader(cabecera)`. Para establecer las cabeceras de la petición. Se usa, si es necesario, entre `open()` y `send()`.

Las propiedades o atributos básicos para comprobar el estado de la petición y su respuesta son:

- `readyState`: Devuelve el estado de una petición, puede tomar 5 valores:

Valor	Nombre	Descripción
0	UNSET	No se ha llamado a `open()`
1	OPENED	Se ha llamado a `open()`, pero no a `send()`
2	HEADERS_RECEIVED	Se han recibido las cabeceras de la respuesta
3	LOADING	Se está recibiendo el cuerpo de la respuesta
4	DONE	Se ha recibido la respuesta

Tabla 2.- Estado de una petición

- `onreadystatechange`: permite asociar una función a un cambio en el estado de la petición. El estado de la petición va cambiando según se desarrolla la comunicación con el servidor. En cada cambio, se llama a la función indicada en `onreadystatechange`.
- `status`: El estado de la respuesta al pedido que devuelve el servido a la petición HTTP realizada. Si el status es 200 entonces el pedido es exitoso
- `response`: Devuelve el cuerpo de la respuesta según el tipo de la petición enviada.

Para más información sobre XMLHttpRequest, sus métodos y sus propiedades, consultar: XMLHttpRequest - Web APIs | MDN (mozilla.org)

AJAX sigue evolucionando, y actualmente existe un método (fetch) que simplifica el cómo hacer peticiones respecto a cómo se realizan con XMLHttpRequest.

Sin embargo, en el desarrollo web XMLHttpRequest se usa por tres razones:

- Código anterior a fetch: necesitamos soportar scripts existentes con XMLHttpRequest.
- Soporte a navegadores viejos que no soportan fetch aún.
- Comprobar el progreso de subida de un formulario al servidor. fetch aún no puede hacerlo.

3.1. Peticiones síncronas y asíncronas

Las peticiones pueden ser síncronas o asíncronas. En el primer caso, la ejecución se detiene hasta que se recibe la respuesta. En el segundo, la ejecución continúa y, cuando se recibe una respuesta, se procesa.

Por tanto, **XMLHttpRequest** tiene dos modos de operación: síncrona y asíncrona.

Vamos a trabajar con la hora del servidor para ver las diferencias entre una petición y otra y para trabajar las peticiones asíncronas. Usamos **hora_servidor.php** que contiene simplemente un echo:

```
1:  <?php
2:      echo date("h:m:s");
```

Comenzamos con **sincrona.html** que contiene el siguiente código:

```
 1:  <script type="text/javascript">
 2:      var xhttp = new XMLHttpRequest();
 3:      xhttp.open("GET", " hora_servidor.php ", false);
 4:      xhttp.send();
 5:      if (xhttp.status == 200) {
 6:          alert("OK");
 7:      } else {
 8:          alert("Error");
 9:      }
10:      alert(xhttp.response);
11:  </script>
```

Para pedir la ruta hora_servidor.php usando el método GET de manera **síncrona**, usamos:

```
var xhttp = new XMLHttpRequest();
xhttp.open("GET", " hora_servidor.php ", false);
xhttp.send();
```

Vemos que el tercer argumento de open es false, lo que quiere decir que la petición es síncrona.

Al ser una petición síncrona, la ejecución del script se detiene hasta que se recibe una respuesta o pasa el tiempo máximo de espera. Una vez recibida, se almacena en la propiedad response de `XMLHttpRequuest()`. El código HTTP de la respuesta se almacena en status.

```
    if (xhttp.status == 200) {…

}
```

Si la petición utiliza el método POST, hay que incluir una cabecera específica y añadir una cadena de parámetros.

```
xhttp.open("POST", ruta, true);
xhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");
xhttp.send("param1=value1&param2-value2");
```

En las **peticiones asíncronas**, que son las más habituales, la situación se complica un poco. Al realizar la petición, el script sigue ejecutándose sin esperar a la respuesta.

Por lo tanto, hay que indicar de alguna manera qué código se encargará de gestionar la respuesta cuando llegue. Para hacerlo se utiliza la propiedad `onreadystatechange`.

Como decíamos anteriormente, el estado de la petición va cambiando según se desarrolla la comunicación con el servidor. **En cada cambio** se llama a la función `onreadystatechange`.

Veamos **asincrona.html**.

```
 1:  <script type = "text/javascript">
 2:  var xhttp = new XMLHttpRequest();
 3:  xhttp.onreadystatechange = function() {
 4:       if (this.readyState == 4 && this.status == 200) {
 5:         alert(this.response);
 6:     }
 7:  };
 8:  xhttp.open("GET", "hora_servidor.php", true);
 9:  xhttp.send();
10:
11:  </script>
```

Para ver las llamadas a la función de la línea 3, vamos a añadir entre las líneas 3 y 4 el siguiente código:

```
alert(this.readyState);
```

De esta forma, podemos ver cómo va evolucionando el estado de la petición hasta que se resuelve, mostrando, sucesivamente, los estados 1, 2, 3 y 4.

Es bastante habitual que estas funciones comiencen comprobando que el estado es 4, comunicación finalizada, y que el servidor ha devuelto el código 200, que indica que no ha habido errores. La primera comprobación es necesaria para que no se ejecute el código cada vez que cambia el estado.

Si la petición se resuelve sin problemas, el servidor devuelve el código 200 (OK). Otros códigos habituales son 404 (Not Found), 400 (Bad Request) o 500 (Internal Server Error).

Ejercicio 1

Modifica el código precedente para que compruebe el código de error 404 y muestre una alerta cuando ocurra.

4. Respuesta del servidor

Como hemos visto en los ejemplos anteriores, la respuesta a una petición no tiene por qué consistir en HTML. En casos sencillos puede ser texto sin formato.

Para devolver datos estructurados (como una lista de productos) se puede utilizar JSON o XML. Cuando se creó AJAX lo habitual era devolver los datos como XML, de ahí el nombre.

Actualmente, en el desarrollo web está muy extendido JSON porque, en el lado del cliente, se pueden transformar en una variable de JavaScript con la función `JSON.parse()`.

El ejemplo **datos_categorias_json.php** devuelve los datos de un array codificados como JSON.

```
1:  <?php
2:      $cat1 = array("cod" => 1, "nombre" => "Comida");
3:      $cat2 = array("cod" => 2, "nombre" => "Bebida");
4:      $array = array($cat1, $cat2);
5:      $json = json_encode($array);
6:      echo $json;
```

Si se accedemos al fichero directamente desde el navegador, obtenemos:

[{"cod":1,"nombre":"Comida"},{"cod":2,"nombre":"Bebida"}]

Ejercicio 2

Modifica el ejemplo datos_categorias_json.php para que cargue las categorías de la base de datos de la aplicación de pedidos (capítulo 4).

5. Modificación estructura de página

En muchos casos, la respuesta del servidor consiste en serie de datos que hay que mostrar de una forma u otra en la página. Puede ser que simplemente haya actualizar una sección o que haya modificar la estructura página, introduciendo y eliminando elementos.

En el ejemplo **hora.php** se muestra un caso básico. Se trata una que simplemente muestra la hora del servidor.

En lugar de una alerta, como en el caso anterior, modifica el contenido de la página utilizando el DOM. Cada cinco segundos solicita el fichero **hora_servidor.php**. Al acceder por primera vez solo se verá el título. Cinco segundos después aparecerá la hora.

- La hora del servidor se muestra en la sección con `id="hora"`, línea 24. Inicialmente está vacía.
- En las líneas 7-18 se declara la función `loadDoc()`. Se encarga de solicitar `hora_servidor.php` al servidor y de poner la respuesta en la página (líneas 11-12).
- Para seleccionar la sección utiliza `document.getElementById("hora")`. El contenido se modifica con la propiedad `innerHTML`.
- La petición se realiza en la línea 15. Los parámetros son el método HTTP mediante el que se realizará la petición, la ruta solicitada y si la petición es asíncrona o no. Con true se especifica que la petición es asíncrona.
- En la línea 19, la función `setInterval()` hace que se llame a `loadDoc()` cada 5 segundos.

```
 1:  <!DOCTYPE html>
 2:  <html>
 3:     <head>
 4:         <title>Hora en el servidor</title>
 5:         <meta charset = "UTF-8">
 6:         <script>
 7:             function loadDoc() {
 8:                 var xhttp = new XMLHttpRequest();
 9:                     xhttp.onreadystatechange = function() {
10:                     if (this.readyState == 4 && this.status == 200) {
11:                         document.getElementById("hora").innerHTML =
12:                             "Hora en el servidor:" + this.response;
13:                     }
14:                 };
15:                 xhttp.open("GET", "hora_servidor.php", false);
16:                 xhttp.send();
17:                 return false;
18:             }
19:             setInterval(loadDoc, 5000);
20:         </script>
21:     </head>
22:     <body>
23:     <?php echo date("h:m:s");?>
24:         <h1>Hora en el servidor</h1>
25:         <section id="hora"></section>
26:     </body>
27:  </html>
```

5.1. Cambiando la vista mediante una petición asíncrona

En este apartado vamos a ver el primer ejemplo sencillo de cómo generar la página web en el lado del cliente usando AJAX. Para ello vamos a definir un script que realice una petición asíncrona, **funciones.js**. Este script será incluido en la página **lista_categorias.html**

Veamos **funciones.js**. En este script mostramos cómo procesar datos en JSON. Solicitamos al servidor **datos_categorias_json.php** y, con los datos que recibe, creamos una lista de categorías.

La función solicita el fichero en la línea 23 de manera asíncrona. Cuando se recibe la respuesta, si esta ha sido correcta (línea 4), entonces se procede de la siguiente manera:

En la línea 6: se crea un elemento ul.

En la línea 8: se transforma la respuesta en un array.

En las líneas 10-16: se recorre la respuesta creando un elemento 14 por cada elemento del array. El texto será el campo nombre. Estos elementos se añaden a la lista.

En la línea 19: se elimina el contenido que pueda haber en la sección principal.

Por último, en la línea línea 20: inserta la lista dentro del elemento principal.

```
 1:   function categorias(){
 2:     var xhttp = new XMLHttpRequest();
 3:     xhttp.onreadystatechange = function() {
 4:      if (this.readyState == 4 && this.status == 200) {
 5:         // crear lista
 6:         var lista = document.createElement("ul");
 7:         // meter los datos de la respuesta en un array
 8:         var cats = JSON.parse(this.response);
 9:         // para cada elemento del array
10:         for(var i = 0; i < cats.length; i++){
11:             //se crea un elemento ul con el campo  nombre
12:             var elem = document.createElement("li");
13:             elem.innerHTML = cats[i]["nombre"];
14:             // se añade a la lista
15:             lista.appendChild(elem);
16:         }
17:         var body = document.getElementById("principal");
18:         // eliminar el contenido actual
19:         body.innerHTML = "";
20:         body.appendChild(lista);
21:      }
22:     };
23:     xhttp.open("GET", "datos_categorias_json.php", true);
24:     xhttp.send();
25:     // para que no se siga el link que llama a esta función
26:     return false;
27:   }
```

La página **lista_categorias.html** es muy sencilla:

- Incluye el fichero en el que está la función con la etiqueta script.
- Define la sección en la que irá la lista.
- Llama a la función.

```
 1:  <!DOCTYPE html>
 2:  <html>
 3:     <head>
 4:         <title>AJAX</title>
 5:         <meta charset = "UTF-8">
 6:         <script type = "text/javascript" src = "funciones.js"></script>
 7:     </head>
 8:     <body>
 9:         <section id = "principal"></section>
10:         <script type = "text/javascript">categorias();</script>
11:     </body>
12:  </html>
```

Ejercicio 3

Modifica el ejemplo anterior para que los elementos de la lista sean vínculos. El texto de los vínculos tiene que ser el nombre de cada categoría.

Tienen que apuntar a productos.php?categoria=<código>, donde <código> es el código de la categoría.

6. Captura de eventos

Muchas veces se llama a una función JavaScript cuando el usuario sigue un vínculo o envía un formulario. Como ya se ha comentado, en estos casos el navegador por defecto carga la respuesta como una nueva página.

Para evitarlo, lo más sencillo es utilizar el atributo `onclick` (para los vinculos). Con este atributo se asocia el evento `click` con una función. Si esta función devuelve FALSE, no se sigue el vínculo. En los formularios se utiliza el atributo `onsubmit`.

El siguiente ejemplo, **lista_categorias2.html**, es similar al anterior. En lugar de mostrar la lista de categorías directamente, muestra un vínculo que llama a la función.

```
 1:  <!DOCTYPE html>
 2:  <html>
 3:     <head>
 4:         <title>AJAX</title>
 5:         <meta charset = "UTF-8">
 6:         <script type = "text/javascript" src = "funciones.js"></script>
 7:     </head>
 8:     <body>
 9:         <section id = "principal"></section>
10:         <a href = "#" onclick = "return categorias();">Categorías</a>
11:     </body>
12:  </html>
```

Ejercicio 4

Modifica el ejemplo anterior para añadir un vínculo que cargue en la sección principal la hora en el servidor (utilizando hora_servidor.php).

7. Aplicaciones de una sola página

Una aplicación web, como la aplicación de pedidos, implica una serie de intercambios entre cliente y servidor. En las aplicaciones de una sola página todas estas comunicaciones, salvo la inicial, se hacen utilizando JavaScript.

En estas aplicaciones la página nunca se recarga por completo. Se va actualizando desde JavaScript como se ha visto a lo largo del capítulo. Desde el punto de vista del usuario, como no hay recargas, estas páginas son más parecidas a aplicaciones de escritorio.

Este enfoque se ha popularizado en los últimos años y se utiliza en varios frameworks muy extendidos, como AngularJs o React.

Por último, una librería útli para trabajar con AJAX y manipular el DOM es JQuery: https://jquery.com/

8. Resumen

- Repartir la lógica entre cliente y servidor permite obtener código más reutilizable.
- La lógica de presentación se desplaza al cliente, donde se utiliza JavaScript.
- El servidor devuelve los datos en JSON, XML o cualquier otro formato.
- Las peticiones al servidor se realizan utilizando el objeto XMLHttptequest
- Para peticiones POST hay que añadir una cabecera y pasar una cadena de parámetros
- Las peticiones pueden ser síncronas o asíncronas.
- En las peticiones asíncronas hay una función encargada de procesar la respuesta
- El código en el cliente se encarga de mostrar los datos recibidos. También modifica la estructura de la página.
- JavaScript realiza cambios en la página utilizando el DOM.
- En las aplicaciones de una sola página, todas las solicitudes al servidor, salvo la primera, se realizan desde Javascript.

9. Ejercicios

Usando AJAX y PHP resuelve los siguientes ejercicios:

1. Escribe una página que muestre un número aleatorio. Cada cinco segundos, la página tiene que solicitar al servidor un nuevo número y mostrarlo.

2. Escribe una página web con un formulario que permita sumar dos números. Utiliza AJAX para enviar el formulario y mostrar el resultado. Para ello, ten en cuenta la siguiente estructura de formulario:

```
<form onsubmit="funcion()">
    Introduce un nombre: <input type="text">
    <input type="submit">
</form>
```

3. Escribe un fichero PHP que devuelva en JSON los datos de la tabla de productos de la aplicación de pedidos.

4. A partir del ejercicio anterior, escribe una página que cree una tabla con los productos.

5. Añade un vínculo al ejercicio anterior para recargar la página.

6. Escribe un formulario de login para la tabla de Restaurantes del capítulo anterior. Muestra el resultado en una alerta.