UT09.1: Ajax

Contenido

1. Introducción

AJAX es el acrónimo de Asynchronous JavaScript And XML (JavaScript asíncrono y XML). Se trata de una técnica que permite realizar comunicación asíncrona con el servidor en segundo plano, es decir, el usuario no es consciente de que se está produciendo esta comunicación ya que no hay una recarga de la página que está visualizando el usuario. Con los datos recibidos en la comunicación, podemos actualizar la información de la página utilizando el API de DOM, mejorando con ello la interactividad, velocidad y usabilidad en las aplicaciones.

Aunque podemos configurar peticiones AJAX síncronas, lo habitual es que las peticiones sean asíncronas, es decir, vamos a realizar la petición y debemos configurar un manejador de eventos que será invocado cuando el navegado haya obtenido la respuesta del servidor, evitando con ello que el hilo de ejecución quede bloqueado esperando la respuesta del servidor.

En primera instancia, la respuesta obtenida del servidor estaba en formato HTML o en formato XML. Actualmente lo habitual es que la respuesta este en formato JSON.

Para facilitar el trabajo, vamos a utilizar una serie de objetos que se utilizan en el intercambio de información en las peticiones. Los objetos con los que vamos a trabajar son:

- Documentos JSON utilizados en el intercambio de información. Un API de tipo rest realiza intercambio de objetos JSON.
- Objetos `URL` y `FormData` para la gestión de formularios.
- Objetos binarios.

2. Documento JSON

JSON es el acrónimo de JavaScript Object Notation. Es un formato de texto sencillo para el intercambio de datos. El formato está basado en los objetos literales, pero traducidos a un *string* para facilitar el intercambio. El formato JSON ha sustituido a XML para el intercambio de información en aplicaciones web por su simplicidad y por su menor peso.

Los tipos de datos que admite JSON son:

- string
- number
- object
- array
- Boolean
- null

Los tipos que no admite JSON son los siguientes:

- function
- date
- undefined

Aunque estos tipos no son admitidos en JSON, siempre podemos pasarlos con *string* para su posterior recuperación. En concreto los tipos *date* suelen utilizar este mecanismo para poder ser intercambiados. Las funciones es más raro que tengamos que intercambiarlas.

2.1. Método `stringify()`

Este método estático está integrado en el objeto `JSON`, permite transformar un objeto literal en un *string* con el formato JSON. En el siguiente ejemplo podemos ver como a partir de un objeto literal, podemos transformar el objeto literal en JSON invocando el método. Los tipo de datos *date* se transforman automáticamente en un *string*.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,
  publication: new Date(),
};
JSON.stringify(product)
```

Podemos traducir también un array.

```
const products = [
  {
    serial: '111-111-111',
    name: 'Portátil',
    price: 750,
    publication: new Date(),
  },
  {
    serial: '222-222-222',
    name: 'Tablet',
    price: 850,
    publication: new Date(),
  },
  {
    serial: '333-333-333',
    name: 'Teléfono',
    price: 500,
    publication: new Date(),
  },
];
JSON.stringify(products)
```

El método, de forma opcional, le podemos pasar una función de tipo callback que será invocada por cada una de las propiedades del objeto literal que queremos transformar. La función recibe dos argumentos, el nombre de la clave de la propiedad y su valor. Si la función devuelve un valor `null` o `undefined`, la propiedad no se integra en el documento JSON. El valor retornado de la función será el que contendrá la propiedad en el documento JSON.

El siguiente ejemplo tenemos una función `replacer()` que será invocada para todas las propiedades de cada objeto literal. En caso de que la propiedad sea “publication” realiza una transformación para generar la fecha en formato local.

```
function replacer(key, value) {
  if (key === "publication") {
    return new Date(value).toLocaleDateString();
  }
  return value;
}
JSON.stringify(product, replacer);
JSON.stringify(products, replacer);
```

2.2. Método `parse()`

Este método también pertenece al objeto `JSON` y es estático. Su función es la inversa que en el caso anterior, dado un *string* con un formato JSON, obtenemos un objeto literal que podamos procesar. En el siguiente ejemplo tenemos dos documentos JSON con un objeto literal y un array, podemos obtener una referencia con sus respectivos objetos literales.

```
const productJSON = '{"serial":"111-111-
111","name":"Portátil","price":750,"publication":"2021-04-
14T20:52:38.721Z"}';
const productsJSON = '[{"serial":"111-111-
111","name":"Portátil","price":750,"publication":"2021-04-
14T20:52:38.721Z"},{"serial":"222-222-
222","name":"Tablet","price":850,"publication":"2021-04-
14T20:52:38.721Z"},{"serial":"333-333-
333","name":"Teléfono","price":500,"publication":"2021-04-
14T20:52:38.721Z"}]';
let product = JSON.parse(productJSON);
let products = JSON.parse(productsJSON);

console.log(product.serial + " " + product.name + " " + product.price + "
 " + product.publication);
console.log(products.length);
```

El segundo argumento del método es opcional denominado *reviver*, nos permite pasar una función de callback que se ejecutará para cada propiedad, con la clave y el valor de la propiedad como argumentos. La función nos permite realizar una transformación del valor de la propiedad, por ejemplo, transformar en un objeto `Date` la propiedad *publication*.

```
function reviver(key, value) {
  if (key === 'publication') {
    console.log(new Date(value));
    return new Date(value);
  }
  return value;
}

product = JSON.parse(productJSON, reviver);
products = JSON.parse(productsJSON, reviver);
console.log(`${product.serial} ${product.name} ${product.price}
console.log(product.publication instanceof Date);
console.log(`${products[1].serial} ${products[1].name}
${products[1].price} ${products[1].publication}`);
console.log(products[1].publication instanceof Date);
```

3. Gestión de URLs y formularios

Tenemos objetos que nos permiten interactuar con URLs y con formularios para adaptarlos a una petición asíncrona.

3.1. Objeto `URL`

Trabajar con URLs siempre es tedioso, primero porque es necesario montarlas en base a strings, y por otro lado que debe cumplir con el estándar **RFC3986** que define los caracteres que están permitidos. La clase URL nos brinda un interfaz para poder trabajar con URLs de forma transparente y cómoda.

Se puede crear un objeto a partir del string con la URL. Utilizamos una URL de la Wikipedia.

```
const url1 = new
URL('https://es.wikipedia.org/wiki/JavaScript#Nacimiento_de_JavaScript');
```

También lo podemos hacer a partir de una base como segundo argumento y creando la ruta relativa. No debemos olvidar la barra del final de la base.

```
const url2 = new URL('JavaScript#Nacimiento_de_JavaScript',
'https://es.wikipedia.org/wiki/');
```

Ambas URLs son la misma en este caso.

```
console.log(url1);
console.log(url2);
```

A partir del objeto podemos acceder a los componentes de la URL fácilmente sin tener que parsearla.

```
console.log(url1.href);
console.log(url1.origin);
console.log(url1.protocol);
console.log(url1.host);
console.log(url1.hostname);
console.log(url1.port);
console.log(url1.pathname);
console.log(url1.search);
console.log(url1.hash);
```

El objeto dispone de la propiedad `searchParams` con un objeto `URLSearchParams` a través de la cual podemos gestionar los parámetros de búsqueda mediante los nombres y los valores de los parámetros, despreocupándonos de la codificación. La interfaz es la siguiente:

- `append(name, value)`: añade un parámetro con un nombre y un valor.
- `delete(name)`: elimina el parámetro a partir del nombre.
- `get(name)`: obtiene el parámetro por su nombre.
- `getAll(name)`: devuelve un array con todos los valores de todos parámetros con el mismo nombre, utilizados en una lista de chequeo, por ejemplo.
- `has(name)`: comprueba si existe un parámetro con ese nombre.
- `set(name, value)`: establece o reemplaza el parámetro con ese nombre.

Además, el objeto `searchParams` es iterable.

```
url1.searchParams.append('serial', '111 111 111');
url1.searchParams.append('name', 'Portátil');
url1.searchParams.append('price', 750);
```

```
url1.searchParams.append('price', 1000);
url1.searchParams.append('publication', new Date());
url1.searchParams.append('test', 'Caracteres especiales: ?+%&/^()');
console.log(url1.href);
console.log(url1.searchParams.get('test'));
console.log(url1.searchParams.getAll('price'));
url1.searchParams.delete('price');
console.log(url1.searchParams.has('price'));
url1.searchParams.set('test', 'ñÑçáéíóú');
for (const [name, value] of url1.searchParams) {
  console.log(`Nombre: ${name}. Valor: ${value}`);
}
```

3.2. Objeto `FormData`

Este tipo de objeto nos permite manipular un formulario, pudiendo crearlo desde 0, o embeber un formulario HTML para capturar todos sus campos, y si fuera necesario asignarle nuevos elementos. La diferencia de `FormData` con un `Form` de DOM es que solamente tiene en cuenta los parámetros para empaquetar en la petición.

La ventaja de este objeto es que puede ser utilizados por diversos métodos de red para adjuntarlo como cuerpo de la petición. El formulario es enviado como *multipart/form-data* por lo que podríamos adjuntar archivos.

Partimos del formulario de login que trabajamos en la unidad de formularios.

```
<form name="login1" role="form">
    <div class="form-group">
        <label for="email">Email address:</label>
        <input type="email" class="form-control" id="email" name="email"
value="test">
    </div>
    <div class="form-group">
        <label for="pwd">Password:</label>
        <input type="password" class="form-control" id="pwd" name="pwd"
value="12345">
    </div>
    <div class="checkbox">
        <label><input type="checkbox" name="remember">Remember me</label>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

El constructo puede invocarse sin argumentos para crear un formulario vacío e ir añadiendo elementos, o podemos pasarle un formulario de DOM con lo que capturará todos sus campos.

```
const formData = new FormData(document.forms.login1);
```

El interfaz que disponemos es el siguiente:

- `formData.append(name, value)`: agrega un campo al formulario con el nombre y el valor pasados como argumentos. Si añadimos varios campos con el mismo nombre se añadirán todos.

- `formData.append(name, blob, fileName)`: permite agregar un campo de tipo *file*, el tercer argumento fileName establece el nombre del archivo.
- `formData.delete(name)`: elimina el campo a partir de su nombre.
- `formData.get(name)`: obtiene el valor del campo con el nombre dado.
- `formData.has(name)`: si existe el campo con el nombre devolverá true.
- `formData.set(name, value)` y `formData.set(name, blob, fileName)`: en caso de existir el campo con el nombre dado, o los campos, los borra y crea un nuevo.

Un objeto `FormData` es iterable.

```
const formData = new FormData(document.forms.login1);
console.log(formData.get('email'));
console.log(formData.get('pwd'));
formData.append('test', 'prueba1');
formData.append('test', 'prueba2');
formData.append('field', 'prueba');
formData.set('field', 'nueva prueba');
formData.append('deleted', 'prueba');
console.log(formData.has('deleted'));
formData.delete('deleted');

for (const [name, value] of formData) {
  console.log(`${name} = ${value}`);
}
```

4. AJAX de forma nativa

El escenario habitual de una petición AJAX lo podríamos resumir en los siguientes pasos:

1. Un evento ocurre en la página, por ejemplo, cliqueamos en un botón o en un enlace.
2. En el manejador del evento debemos crear un objeto del tipo `XMLHttpRequest`. Este objeto es el que nos va a permitir realizar la petición.
3. El objeto `XMLHttpRequest` envía una petición al servidor web donde está alojada nuestra aplicación.
4. El servidor procesa la petición y construye una respuesta. El formato de la respuesta podría ser texto, HTML, XML, JSON o un objeto binario.
5. El servidor envía la respuesta de vuelta al navegador.
6. La respuesta es recibida en el navegador, disparándose un evento para avisarnos de que la hemos recibido.
7. En el manejador del evento debemos actualizar el contenido de la página con los datos que hemos recibo en la respuesta.

La siguiente imagen muestra el ciclo de este escenario.

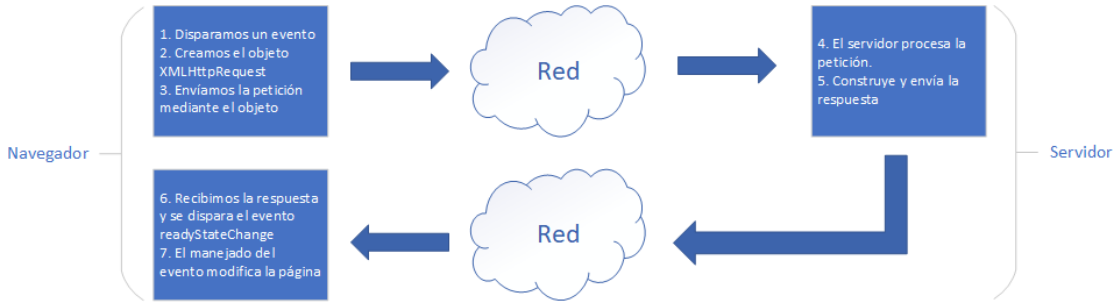

Imagen 1 Escenario petición AJAX

4.1. Implementación de petición AJAX

Vamos a crear una función anónima que podamos asignar como manejador de eventos. En primer lugar, debemos crear el objeto `XMLHttpRequest`, se trata de un objeto nativo que permite realizar solicitudes HTTP. El objeto permite recuperar cualquier tipo de respuesta, no solamente XML. Los tipos de respuesta son:

- Texto.
- Datos binarios.
- Documentos XML o HTML, los cuales se pueden acceder a sus datos utilizando DOM.
- Objetos JSON.

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

Para poder enviar la petición, debemos invocar el método `open()` sobre el objeto `XMLHttpRequest` para especificar los parámetros principales de la petición, estos son por orden de invocación:

- *Método*: Básicamente serían GET o POST.
- *URL*: la URL que estamos solicitando.
- *Async*: optativamente indicamos si la petición debe ser asíncrona, valor por defecto `true`, o síncrona, valor `false`. Una petición síncrona bloquea el hilo de ejecución hasta que se recibe la respuesta, actualmente está deprecado.
- *Usuario y contraseña*: en el caso de que tengamos una autenticación básica HTTP. No suelen utilizarse.

Vamos a realizar una petición al api pública https://httpbin.org que actúa de eco para los argumentos que recibe empaquetándolos en un documento JSON. Creamos una petición utilizando el método GET a la URL https://httpbin.org/get de forma asíncrona.

```
xhttp.open('GET', 'https://httpbin.org/get', true);
```

El método `send()` envía la petición.

```
xhttp.send();
```

Por último, debemos escuchar los eventos de respuesta. El evento `load` se dispara cuando la solicitud está completa. El manejador de eventos asignado a él será el encargado de atender la respuesta, es en ese momento cuando el objeto `XMLHttpRequest` tendrá disponible la propiedad `status` con el código de respuesta HTTP, y la propiedad `response` con la respuesta.

Tenemos que entender que un código de respuesta 200 es una respuesta que el servidor ha atendido correctamente y por tanto obtenemos una respuesta válida, pero también un código 400 es una respuesta válida pero que no se ha podido atender de forma satisfactoria en el servidor. El manejador del evento debe atender ambos tipos de respuesta porque ambas son correctas. En el siguiente ejemplo vemos como atendemos ambos tipos de casos.

```
xhttp.onload = function () {
  if (this.status !== 200) {
    console.log(`Error ${this.status}: ${this.statusText}`);
  } else {
    console.log(`Hecho, obtenidos ${this.response.length} bytes`);
    console.log(this.response);
  }
};
```

Vamos a hacer una petición a una URL que el servidor no encuentre el recurso en concreto para que genere un código 404. En este caso el manejado debe gestionar el error.

```
xhttp.open('GET', 'https://httpbin.org/error', true);
xhttp.send();
```

El evento `progress` se dispara periódicamente mientras se descarga la respuesta. Con este evento podríamos implementar una barra de estado de descarga, por ejemplo. El evento tiene una propiedad `lengthComputable` que indica si hemos recibido la cabecera `Content-Length` que nos permite conocer el tamaño final del recurso descargado. La propiedad `loaded` nos dice el número de byte recibidos hasta el momento.

```
xhttp.onprogress = function (event) {
  if (event.lengthComputable) {
    console.log(`Recibidos ${event.loaded} de ${event.total} bytes`);
  } else {
    console.log(`Recibidos ${event.loaded} bytes`); // sin Content-Length
  }
};
```

A medida que avanza la petición, el objeto `XMLHttpRequest` cambia de estado. Podemos observar los estados por los que avanza la petición capturando el evento `readystatechange` en su propiedad `readyState`. Los estados son:

- Estado inicial. UNSENT = 0;
- Llamada abierta. OPENED = 1;

- Cabecera de respuesta recibidas. HEADERS_RECEIVED = 2;
- La respuesta está cargando. LOADING = 3;
- Solicitud completada. DONE = 4;

Capturamos el evento para ver estos cambios.

```
xhttp.onreadystatechange = function () {
  if (this.readyState === 2) {
    console.log('readystatechange: HEADERS_RECEIVED ');
  }
  if (this.readyState === 3) {
    console.log('readystatechange: LOADING');
  }
  if (this.readyState === 4) {
    console.log('readystatechange: DONE');
  }
};
```

En caso de error de la red o una URL inválida se disparará el evento `error`.

```
xhttp.onerror = function () {
  console.log('Solicitud fallida');
};
```

Forzamos una petición a una URL inválida.

```
xhttp.open('GET', 'https://noexiste', true);
xhttp.send();
```

Por último, podemos configurar un timeout de la petición. Si se alcanza la petición será cancelada, y el evento `timeout` se activa.

```
xhttp.ontimeout = function (event) {
  console.log('Salta el timeout');
};
```

Lanzamos una petición con el timeout activado con un valor muy bajo.

```
xhttp.timeout = 1;
xhttp.open('GET', 'https://noexiste', true);
xhttp.send();
```

Una petición se puede abortar a través del método `abort()`.

```
xhttp.abort();
```

4.2. Cabeceras HTTP

`XMLHttpRequest` permite personalizar las cabeceras HTTP, como leer cabeceras en la respuesta.

Para personalizar una cabecera tenemos el método `setRequestHeader()` cuyos argumentos son el nombre y el valor de la cabecera. El método no permite eliminar una cabecera, ni sobrescribirla, en caso de añadir la misma cabecera dos veces, tan solo agregaría nueva información a la cabecera. Las cabeceras `Host` y `Referer` no se pueden personalizar, ya que su gestión depende del navegador. Algunos ejemplos podrían ser los siguientes para configurar un formulario de tipo POST, o para solicitar que el contenido de la respuesta sea de tipo JSON.

```
xhttp.setRequestHeader('Content-type', 'application/x-www-form-
urlencoded');
xhttp.setRequestHeader('Content-Type', 'application/json');
```

Con los métodos `getResponseHeader()` y `getAllResponseHeaders()` podemos obtener una cabecera de la respuesta partir de su nombre, o el listado completo en un string, por lo que habría que parsearlo a partir del separado '\r\n'. Podemos añadir el siguiente código en un evento `load` para reducir las cabeceras de respuesta a un objeto literal.

```
const headers = this
  .getAllResponseHeaders()
  .split('\r\n')
  .reduce((result, current) => {
    const [name, value] = current.split(': ');
    result[name] = value;
    return result;
  }, {});
```

4.3. Peticiones de tipo GET

Vamos a crear una petición GET contra el api pública https://httpbin.org a modo de ejemplo. El api devuelve un eco de los parámetros enviados en formato JSON.

Construimos la URL con los parámetros que necesitemos, en este caso reutilizamos el ejemplo con el objeto `URL`.

```
const url = new URL('https://httpbin.org/get');
url.searchParams.append('serial', '111 111 111');
url.searchParams.append('name', 'Portátil');
url.searchParams.append('price', 750);
url.searchParams.append('price', 1000);
url.searchParams.append('publication', new Date());
url.searchParams.append('test', 'Caracteres especiales: ?+%&/^()');
```

Creamos el objeto XMLHttpRequest y abrimos la URL anterior. Vamos a configurar la respuesta de tipo JSON, enviamos la petición.

```
const xhttp = new XMLHttpRequest();
xhttp.open('GET', url, true);
xhttp.responseType = 'json';
xhttp.send();
```

En el evento `load` capturamos la respuesta. En este caso el objeto response ya se ha transformado en JSON al haber configurado que el tipo de respuesta tuviera ese formato, si hubiéramos utilizado el tipo de respuesta texto, deberíamos parsear el JSON. Ahora tenemos un objeto literal con el que podemos interactuar por lo que mostramos los argumentos recibidos por el servidor y que nos han empaquetado en la respuesta.

```
xhttp.onload = function () {
    if (this.status !== 200) {
        console.log(`Error ${this.status}: ${this.statusText}`);
    } else {
        console.log(this.response.args);
    }
};
```

4.4. Peticiones de tipo POST

Las peticiones POST embeben los parámetros en el cuerpo de la petición. Para este tipo de peticiones tenemos que indicar la cabecera de tipo `application/x-www-form-urlencoded` con el método `setRequestHeader()`, pero si utilizamos un objeto FormData no debemos hacerlo porque debemos recordar que la petición será `multipart/form-data` por defecto. Por último, pasamos los parámetros en el método `send()` como argumento, para nuestro ejemplo a través de un objeto `FormData`. El servidor debe aceptar una petición para este tipo de método.

Los argumentos de la respuesta vienen empaquetados en la propiedad `form`.

```
const url = new URL('https://httpbin.org/post');
const xhttp = new XMLHttpRequest();
xhttp.open('POST', url, true);
xhttp.responseType = 'json';
const formData = new FormData(document.forms.login1);
xhttp.send(formData);

xhttp.onload = function () {
    if (this.status !== 200) {
        console.log(`Error ${this.status}: ${this.statusText}`);
    } else {
        console.log(this.response.form);
    }
};
```

4.5. Ejemplo generación de usuario aleatorios

Veamos un ejemplo de cómo podemos integrar contenido a partir de un documento JSON. El servicio https://randomuser.me/api/ ofrece la generación de documentos JSON con información aleatoria que podemos integrar en nuestra página.

Creamos un formulario y lo dejamos accesible en *index.html*. El formulario recoge el número de usuarios a generar y su género.

```
<div class="container " id="cRandomUsers">
    <h1 class="display-5">Genera usuarios</h1>
    <form name="fRandomUsers" role="form" class="row g-3" novalidate>
        <div class="col-md-3">
            <div class="input-group">
                <label class="input-group-text" for="rufNumber">Nº de
usuarios. *</label>
                <input type="number" class="form-control" id="rufNumber"
name="rufNumber" value="" required
                    min="1" max="20">
            </div>
        </div>
        <div class="col-md-3">
            <div class="input-group">
                <label class="input-group-text" for="rufType"
style="color: #faa541">Género</label>
                <select class="form-select" name="rufType" id="rufType">
                    <option selected>Ambos</option>
                    <option value="male">Hombres</option>
                    <option value="female">Mujeres</option>
                </select>
            </div>
        </div>
        <div class="col-md-1">
            <button class="btn btn-primary" type="submit">Enviar</button>
        </div>
    </form>
</div>
```

Creamos una función que genere una tarjeta de información de cada usuario.

```
function getCard(name, phone, email, picture){
  let card = $(`<div class="card col-3" style="width: 18rem;">
    <img class="card-img-top" src="${picture}" alt="${name}">
    <div class="card-body">
      <h5 class="card-title">${name}</h5>
      <p class="card-text">${phone}</p>
      <p class="card-text">${email}</p>
    </div>
  </div>`);
  return card;
}
```

Capturamos el evento `submit` del formulario. En el manejador construimos la URL, añadiéndole dos parámetros *results* con el número de usuarios a generar y *gender* con el género de los usuarios. Ambos valores los recogemos del formulario. Construimos el objeto `XMLHttpRequest` y enviamos la petición de tipo GET para obtener resultados JSON.

En el evento `load` de la petición construimos un *grid* para mostrar los usuarios, y recorremos el array de la propiedad *result* de la respuesta para invocar a la función `getCard()` por cada usuario.

Por último, cancelamos el comportamiento por defecto del `submit`.

```
document.forms.fRandomUsers.addEventListener('submit', (event) => {
  const url = new URL('https://randomuser.me/api/');
  url.searchParams.append('results',
event.currentTarget.rufNumber.value);
  url.searchParams.append('gender', event.currentTarget.rufType.value);

  const xhttp = new XMLHttpRequest();
  xhttp.open('GET', url, true);
  xhttp.responseType = 'json';
  xhttp.send();

  xhttp.onload = function () {
    if (this.status !== 200) {
      $$result.log(`Error ${this.status}: ${this.statusText}`);
    } else {
      $$result.log(this.response);
      const main = document.querySelector('main');
      main.replaceChildren();
      const container = document.createElement('div');
      container.classList.add('container');
      container.insertAdjacentHTML('afterbegin', '<div
class="row"></div>');
      const row = container.querySelector('div.row');
      main.append(container);
      this.response.results.forEach((user) => {
        const name = `${user.name.title} ${user.name.first}
${user.name.last}`;
        const { phone } = user;
        const { email } = user;
        const picture = user.picture.large;
        row.append(getCard(name, phone, email, picture));
      });
    }
  };

  event.preventDefault();
});
```

5. AJAX con jQuery

El framework dispone de dos métodos para interactuar con peticiones asíncronas.

5.1. Evento `load()`

El método lo utilizamos para cargar contenido HTML en nuestra página. En el primer ejemplo pasamos una URL como argumento al método sobre un elemento. El contenido del elemento es sustituido por el contenido del HTML recibido.

Creamos un documento HTML.

```
<div class="container">
    <h1>Ejemplo de carga de elementos</h1>
    <ul>
        <li>Dato 1</li>
        <li>Dato 2</li>
        <li>Dato 3</li>
        <li>Dato 4</li>
        <li>Dato 5</li>
    </ul>
    <div id="second" class="container">
        <h2>Datos secundarios</h2>
        <ul>
            <li>Dato A</li>
            <li>Dato B</li>
            <li>Dato C</li>
            <li>Dato D</li>
            <li>Dato E</li>
        </ul>
    </div>
</div>
```

En la página principal cargamos el contenido del documento en el elemento `main`.

```
let categories = $('#categories');
categories.empty();
let bLoad = $(`<button class="btn btn-primary">Cargar HTML 1</button>`);
categories.append(bLoad);

bLoad.click(function (event){
  $('main').load('load.html');
})
```

En lugar de cargar todo el contenido, podemos seleccionar parte de él utilizando un selector concreto en la invocación. En este ejemplo solo mostramos el contenido de la capa *second*.

```
let bsecondLoad = $(`<button class="btn btn-primary m-
1">Cargar HTML 2</button>`);
categories.append(bsecondLoad);
bsecondLoad.click(function (event){
  $('main').load('load.html #second');
})
```

El siguiente ejemplo obtiene los elementos `li` y los añade a nuestra página.

```
let bLiLoad = $(`<button class="btn btn-primary m-
1">Cargar HTML 3</button>`);
categories.append(bLiLoad);
bLiLoad.click(function (event){
  let main = $('main');
  main.empty();
  main.append(`<div class="container"><ul></ul></div>`);
  $('main ul').load('load.html li');
})
```

Podemos tener un parámetro con la función de callback ejecutada al completar la acción. Esta función recibe tres argumentos, la respuesta, el estado de la petición, y el objeto con la petición `XMLHTTPRequest` incluido en un objeto jQuery.

```
let bComplete = $(`<button class="btn btn-primary m-
1">Cargar HTML 4</button>`);
categories.append(bComplete);
bComplete.click(function (event){
  $('main').load('load.html', function(response, status, xhr){
    if ( status == 'success' ) {
      $('main').append('<div class="container"><strong>La acción ha sido
completada.</strong></div>');
    }
  });
})
```

Con el argumento *status* podemos gestionar si ha habido algún error en la petición.

```
let bError = $(`<button class="btn btn-primary m-
1">Cargar HTML 5</button>`);
categories.append(bError);
bError.click(function (event){
  $('main').load('noExist.html', function(response, status, xhr){
    if ( status == 'error' ) {
      $('main').empty();
      $('main').append('<div class="container"><strong>Se ha producido un
 error en la solicitud.</strong></div>');
    }
  });
})
```

5.2. Crear peticiones a bajo nivel. Método `ajax()`

Este método nos permite crear la petición personalizándola para recibir un contenido del servidor. El método se configura en base a objetos literales. En primer lugar, vamos a crear el botón para realizar la petición para simular el mismo ejemplo que hemos hecho con la forma nativa.

```
let categories = $('#categories');
categories.empty();
let bAjax = $(`<button class="btn btn-primary m-
1">Uso de Ajax</button>`);
categories.append(bAjax);
```

En el evento `click` vamos a realizar una petición AJAX utilizando el método `ajax()`. Al método le pasamos como argumento un objeto literal con la propiedad `url` que indica el recurso que solicitamos al servidor, y una propiedad `success` con una función que será invocada cuando la petición tenga éxito. Esta función recibe como argumento la respuesta del servidor para que sea procesada, en nuestro caso el fragmento HTML que integramos en la página.

El método devuelve un objeto del framework jQuery con la petición `XMLHttpRequest` embebido. Sobre este objeto podemos invocar varios métodos.

- El método `done()` será invocado cuando la respuesta esté lista en el navegador. Este método invocará la función pasada como argumento, la cuya recibe la respuesta, la cual podría ser tratada en este punto en sustitución de la función de la propiedad `success`, el estado y el objeto `XMLHttpRequest`.
- El método `fail()` es invocado en caso de que la petición no haya tenido éxito. Pudiéndolo utilizar para gestionar errores.
- El método `always()` se ejecuta siempre, independientemente si se ha tenido éxito o no.

```
bAjax.click(function (event){
  $.ajax({
    url: 'load.html',
    success: function (data) {
      $('main').empty();
      $('main').append(data);
    }
  }).done(function (rp, status, xhr) {
    $('main').append('<div class="container"><strong>Petición realizada.<
/strong></div>');
      console.log('DONE');
  }).fail(function () {
    $('main').append('<div class="container"><strong>Se ha producido un e
rror en la solicitud.</strong></div>');
  }).always(function () {
    $('main').append('<div class="container"><strong>Acabamos de realizar
 una petición.</strong></div>');
  })
})
```

5.3. Peticiones de documentos JSON

Repetimos el ejemplo de la generación de usuarios aleatorios pero utilizando el método `ajax()`.

La función `getCard()` la rediseñamos para utilizar jQuery. Este paso realmente no es necesario porque podríamos haber reutilizado la función.

```
function getCard(name, phone, email, picture){
  let card = $(`<div class="card col-3" style="width: 18rem;">
    <img class="card-img-top" src="${picture}" alt="${name}">
    <div class="card-body">
      <h5 class="card-title">${name}</h5>
      <p class="card-text">${phone}</p>
      <p class="card-text">${email}</p>
    </div>
  </div>`);
  return card;
}
```

Vamos a crear un botón que al cliquearse solicite 8 usuarios al servicio a través del método `ajax()`. En el objeto literal de configuración de la petición, vamos a configurar las siguientes propiedades:

- `url`: Fijamos la URL del servicio.
- `dataType`: indicamos que el tipo MIME del recurso a descargar será de formato JSON.
- `method`: Nos permite configurar el tipo de petición, en este caso un GET.
- `data`: nos permite indicar los parámetros que vamos a pasar en la petición a través de un objeto literal. En este aso solamente queremos pasar el número total de resultados a mostrar.
- `success`: contiene la respuesta obtenida. Al tratarse de un array, podemos iterar sobre la respuesta obtenida. No es necesario parsear la respuesta, ya que jQuery se ha encargado de hacerlo por nosotros.
- `error`: Aunque no está configurado, podemos asignar una función que será invocada en caso de la que la petición no tenga éxito.

```
let categories = $('#categories');
categories.empty();
let bAjaxJSON = $(`<button class="btn btn-primary m-
1">8 usuarios</button>`);
categories.append(bAjaxJSON);
bAjaxJSON.click(function (event){
  $.ajax({
    url: 'https://randomuser.me/api/',
    dataType: 'json',
    method: 'get',
    data : { results : 8 },
    success: function(data) {
      let main = $('main');
      main.empty();

      let row = $(`<div class="container"><div class="row"></div></div>`)
;
      main.append(row);
      row = row.find('.row');

      data.results.forEach(user => {
        let name = user.name.title + ' ' + user.name.first + ' ' + user.n
ame.last;
        let phone = user.phone;
        let email = user.email;
        let picture = user.picture.large;

        row.append(getCard(name, phone, email, picture));
      });

      console.log(data);
    }
  });
});
```

El servicio está preparado para especificar el género de los usuarios. Para obtener 4 mujeres de forma aleatoria tenemos que pasar el siguiente objeto literal en la propiedad `data` en la invocación del método.

```
data: {results: 4, gender: 'female'}
```