PCW

PROGRAMACIÓN DEL CLIENTE WEB

Tema 06 – JavaScript avanzado

Dept. de Ciència de la Computació i Intel·ligència Artificial
Dpto. de Ciencia de la Computación e Inteligencia Artificial

Universitat d'Alacant
Universidad de Alicante

JavaScript avanzado

JavaScript avanzado

- ✓ AJAX
- ✓ Fetch API
- ✓ Excepciones
- ✓ Promises
- ✓ Temporizadores

AJAX

Asynchronous JavaScript And XML

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

- **Conjunto de varias tecnologías independientes** que se combinan para crear aplicaciones interactivas o RIA (*Rich Internet Applications*).
- Estas aplicaciones se ejecutan en el navegador y mantienen la comunicación asíncrona con el servidor en segundo plano.
- Permite realizar cambios sobre las páginas web sin necesidad de recargarlas.
- Aplicaciones conocidas que utilizan AJAX: Gmail, Google Maps, Google Docs, Flickr, Yahoo Mail, ...

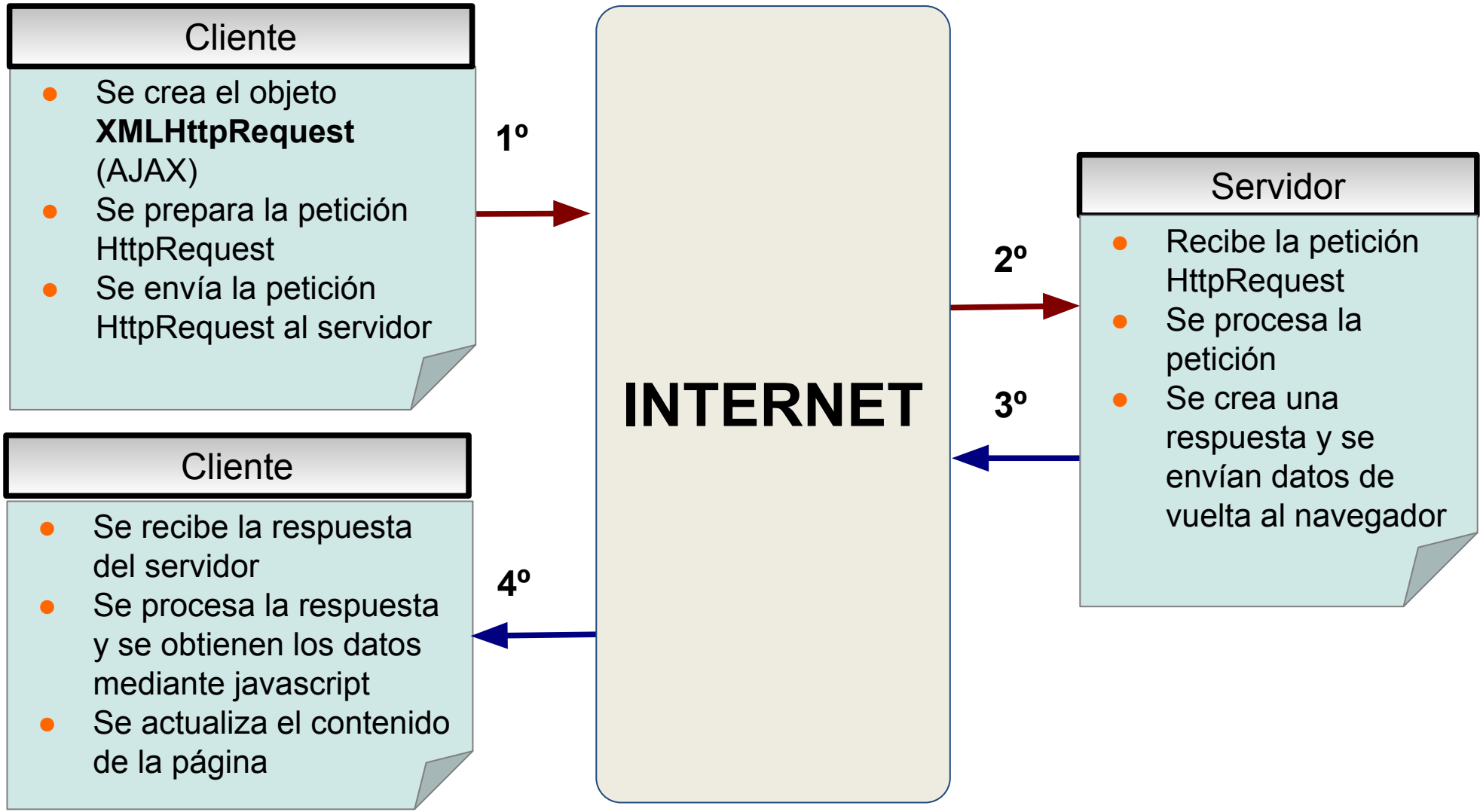
AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Las tecnologías que forman AJAX son:

- **XHTML / HTML** y **CSS**, para crear una presentación basada en estándares.
- **DOM**, para la interacción y manipulación dinámica de la presentación.
- **XML** o **JSON**, para el intercambio y la manipulación de información.
- **XMLHttpRequest**, para el intercambio asíncrono de información.
- **JavaScript**, para unir todas las demás tecnologías.

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Funcionamiento:



AJAX: *Asynchronous JavaScript And XML*

Objeto *XMLHttpRequest*

- Objeto de Javascript utilizado para obtener información de una URL de forma sencilla sin recargar la página web.
- Permite realizar peticiones HTTP y HTTPS, síncronas o asíncronas, por las que obtiene una respuesta.
- Los datos transferidos pueden ir en diferentes formatos: texto plano, XML, JSON, HTML.
- Fue diseñado por Microsoft y adoptado por Mozilla, Apple y Google, acabando convirtiéndose en un estándar.

AJAX: *Asynchronous JavaScript And XML*

Objeto *XMLHttpRequest*.

Métodos y propiedades relacionados con la petición.

- `open(`*`método, URL [, asíncrono [, usuario [, password]] ] ]`*`)`: Especifica las características de la conexión.
  - *método*: "GET" o "POST". Mismo funcionamiento que en formularios HTML.
  - *URL*: Dirección URL, relativa o absoluta, hacia el recurso en el servidor.
  - *Asíncrono*: Especifica si la petición se gestionará de forma asíncrona (valor *true*), o síncrona (valor *false*). El valor por defecto es *true*.
- `setRequestHeader(`*`etiqueta, valor`*`)`: Añade un par *etiqueta*/*valor* a la cabecera HTTP a enviar al servidor.
- `timeout`: Permite especificar el tiempo máximo (en milisegundos) a esperar la respuesta.

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Objeto *XMLHttpRequest*.

Métodos y propiedades relacionados con la petición.

- `withCredentials`: Con valor `true` permite realizar peticiones de tipo *cross-origin* a un servidor externo que lo permita. Valor por defecto: `false`.
- `upload`: Devuelve el objeto XMLHttpRequestUpload que permite monitorear el proceso de envío de datos al servidor.
- `send( [datos] )`: Inicia la petición al servidor. En *datos* se envía al servidor el cuerpo de la petición, es decir, la información cuando el método de envío elegido es POST. Cuando el método de envío es GET, se añade a la *URL* del servidor.
- `abort()`: Permite detener la petición en curso.

AJAX: *Asynchronous JavaScript And XML*

Objeto *XMLHttpRequest*.

Métodos y propiedades relacionados con la respuesta.

- `responseURL`: Devuelve la url de la respuesta. Por lo general coincidirá con la url utilizada en el método `open()` de la petición.
- `status`: Devuelve el código numérico que representa el estado en el que se encuentra la respuesta: 404 (documento no encontrado); 200 (OK); etc.
- `statusText`: Devuelve el texto descriptivo del estado de la respuesta indicado en la propiedad `status`.
- `getResponseHeader(`*`etiqueta`*`)`: Devuelve el valor de la cabecera HTTP indicada en *`etiqueta`*.
- `getAllResponseHeaders()`: Devuelve todas las cabeceras HTTP de la respuesta (etiquetas y valores) como una cadena.

AJAX: *Asynchronous JavaScript And XML*

Objeto *XMLHttpRequest*.

Métodos y propiedades relacionados con la respuesta.

- `overrideMimeType(formatoMime)`: Permite cambiar el tipo MIME (formato de la información de intercambio) usado por el servidor y utilizar el indicado en *formatoMime*. Posibles valores: text/html, text/xml, text/plain
- `responseType`: Permite establecer o consultar el type de los datos que hay en el cuerpo de la respuesta: `json`, `text`, `blob`, ...
- `response`: Devuelve el contenido del cuerpo de la respuesta en el formato indicado por `responseType`.
- `responseText`: Devuelve el contenido del cuerpo de la respuesta en formato texto si `responseType` tiene el valor `'text'` o no se ha establecido valor.
- `responseXML`: Devuelve el contenido del cuerpo de la respuesta como un documento HTML o XML si `responseType` tiene el valor `'document'`.

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Objeto *XMLHttpRequest*.

Manejadores de eventos.

- `onloadstart`: Se dispara al iniciar la carga (antes del envío).
- `onprogress`: Se dispara periódicamente con información de estado.
- `onabort`: Se dispara al abortar la petición.
- `onerror`: Se dispara cuando se ha producido un error en la petición que hace que falle.
- `onload`: Se dispara cuando ha finalizado con éxito la petición.
- `ontimeout`: Se dispara cuando se ha agotado el tiempo de espera y no se ha completado la petición.
- `onreadystatechange`: Se dispara con cada cambio de estado.
- `onloadend`: Se dispara cuando la petición se ha completado, independientemente de que se haya completado con éxito o haya fallado.

AJAX: *Asynchronous JavaScript And XML*

Ejemplo de uso: Petición de tipo **GET**

```
var obj = new XMLHttpRequest(); // variable que guarda el objeto XMLHttpRequest
function peticionAJAX_GET(url) {
  if(obj) { // Si se ha creado el objeto, se completa la petición ...
    var login = document.getElementById("login").value; // Se preparan los
    var pass  = document.getElementById("pass").value;  // argumentos ...
    url += "?l=" + login + "&p=" + pass; // se añaden los argumentos a la url
    url += "&v=" + (new Date()).getTime(); // Truco: evita utilizar la cache
    // Se establece la función (callback) a la que llamar cuando cambie el estado:
    obj.onreadystatechange = procesarCambio; // función callback: procesarCambio
    obj.open("GET", url, true); // Se crea petición GET a url, asíncrona ("true")
    obj.send(); // Se envía la petición
  }
  return false;
}
```

AJAX: *Asynchronous JavaScript And XML*

Ejemplo de uso: Petición de tipo **POST**

```
var obj = new XMLHttpRequest(); // variable que guarda el objeto XMLHttpRequest
function peticionAJAX_POST(url) {
  if (obj) { // Si se ha creado el objeto, se completa la petición ...
    // Argumentos:
    var login = document.getElementById("login").value;
    var pass  = document.getElementById("pass").value;
    var args = "l=" + login + "&p=" + pass;
    args += "&v=" + (new Date()).getTime(); // Truco: evita utilizar la cache
    // Se establece la función (callback) a la que llamar cuando cambie el estado:
    obj.onreadystatechange = procesarCambio; // función callback: procesarCambio
    obj.open("POST", url, true); // Se crea petición POST a url, asíncrona
    // Es necesario especificar la cabecera "Content-type" para peticiones POST
    obj.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    obj.send(args); // Se envía la petición. Los argumentos van aquí.
  }
  return false;
}
```

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Ejemplo de uso: función callback y html

```
function procesarCambio(){
  if(obj.readyState == 4){ // respuesta recibida y lista para ser procesada
    if(obj.status == 200){ // El valor 200 significa "OK"
      // Aquí se procesa lo que se haya devuelto:
      document.getElementById("miDiv").innerHTML = obj.responseText;
    } else alert("Hubo un problema con los datos devueltos"); // ERROR
  }
}
```

ejemploAjax.html
Petición AJAX en evento *submit* de formulario

```
<!doctype html>
<html lang="es">
  <head>
    <title>Ejemplo de petici&oacute;n AJAX</title>
    <script> // Aquí va el código javascript anterior </script>
  </head>
  <body>
    <form onsubmit="return peticionAJAX_POST('rest/servicio/');">
      <label for="login">Login:</label><input type="text" id="login" name="login">
      <label for="pass">Pass:</label><input type="password" id="pass" name="pass">
      <input type="submit" value="Enviar">
    </form>
    <div id="miDiv"></div>
  </body>
</html>
```

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Ejemplo de uso: función callback y html

ejemploAjax.html
Petición AJAX mediante llamada a función javascript

```
<!doctype html>
<html lang="es">
  <head>
    <title>Ejemplo de petici&oacute;n AJAX</title>
    <script> // Aquí va el código javascript anterior </script>
  </head>
  <body>
    <div id="miDiv"></div>
    ....
    <script>
      // Se lanza la petición AJAX al servidor
      peticionAJAX_GET('rest/servicio/');
    </script>
  </body>
</html>
```

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Ejemplo de uso: Ejemplo más optimizado

```
function peticionAJAX_GET(url) {
  let xhr = new XMLHttpRequest();
  if(xhr) { // Si se ha creado el objeto, se completa la petición ...
    let login = document.getElementById("login").value, // Se preparan los
        pass  = document.getElementById("pass").value;  // argumentos ...
    url += "?l=" + login + "&p=" + pass; // se añaden los argumentos a la url
    url += "&v=" + (new Date()).getTime(); // Truco: evita utilizar la cache
    xhr.open("GET", url, true); // Se crea petición GET a url, asíncrona
    // El manejador de eventos onload ejecuta la función callback cuando
    // finaliza la petición y se recibe la respuesta.
    xhr.onload = function(){ // función callback
        document.getElementById("miDiv").innerHTML = xhr.responseText;
    };
    xhr.send(); // Se envía la petición
  }
  return false;
}
```

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Interfaz **FormData**

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Interfaz **FormData**

- Un objeto FormData representa una **lista ordenada de entradas**. Cada una de estas entradas no es más que un par *nombre/valor*.
- **Permite encapsular todos los campos de un formulario** HTML, junto a sus valores, en un único objeto *FormData*. Este objeto *FormData* se puede enviar como único *parámetro* en una llamada ajax de tipo **POST**.
- En el servidor los datos se reciben como si fuese el propio formulario el que los enviara.

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Interfaz **FormData**

Constructor

```
var fd = new FormData( formulario_HTML )
```

formulario_HTML. *Opcional*. Es el elemento `<form>` del DOM que contiene los campos con la información a encapsular.

- **Método *append***. Permite añadir nuevas entradas a la lista de pares nombre/valor.

```
fd.append( nombre, valor )
```

Añade el nuevo par *nombre*/*valor* a la lista de entradas que ya tuviera el objeto *FormData*.

AJAX: *A*synchronous *J*avaScript *A*nd *X*ML

Interfaz **FormData**

Ejemplo:

JavaScript

```
function peticionAjax(frm){
  var fd = new FormData( frm );
  var xhr= new XMLHttpRequest();

  fd.append('op','3'); // se añade una nueva entrada a la lista de entradas

  xhr.open('POST', 'url/recurso/api/restful', true);
  xhr.responseType = 'json'; // se pide la respuesta en formato json
  xhr.onload = function(){ // Se ha finalizado la petición correctamente y
    // se tienen los datos recibidos del servidor
    let r = xhr.response; // Se recogen los datos en formato json
  };

  xhr.send(fd);
  return false;
}
```

HTML

```
<form onsubmit="return peticionAJAX(this);">
  <label>Título: <input type="text" name="titulo"></label>
  <label>Foto: <input type="file" name="foto"></label>
  <input type="submit" value="Enviar">
</form>
```

Fetch API

Fetch API

- Permite hacer peticiones de red similares a las que se realizan con el objeto `XMLHttpRequest`.
- Diferencia: Fetch API utiliza **`Promises`**, haciendo más claro y limpio el código.
- Proporciona las interfaces **`Request`**, **`Response`**, **`Headers`** y **`Body`** para facilitar el trabajo con las peticiones de red.
- Para hacer una petición y recuperar la respuesta se utiliza el método **`fetch()`**, implementado en **`Window`**. Este método devuelve una promesa.

Fetch API

Interfaz Headers

Permite realizar diferentes acciones sobre las cabeceras de las peticiones y respuestas HTTP.

Constructor:

```
new Headers()
```

Métodos:

- **`append(`***`cabecera, valor`***`)`**. Añade una nueva cabecera o un nuevo valor a una ya existente.
- **`set(`***`cabecera, valor`***`)`**. Añade una nueva cabecera o cambia el valor a una ya existente.
- **`delete(`***`cabecera`***`)`**. Borra una cabecera de las ya existentes en el objeto.
- **`get(`***`cabecera`***`)`**. Devuelve el valor de la cabecera indicada, o `null` si no existe.
- **`has(`***`cabecera`***`)`**. Devuelve `true` si existe la cabecera indicada en el objeto.
- **`entries()`**, **`values()`**, **`keys()`**. Devuelve un objeto de tipo `iterator` que permite recorrer los pares clave/valor de la colección de cabeceras.

Fetch API

Interfaz Headers

Ejemplo:

```
var myHeaders = new Headers(); // Crea un objeto Headers

myHeaders.append('Content-Type', 'text/html'); // Añade una cabecera
myHeaders.set('Vary', 'Accept-Language'); // Añade otra cabecera
myHeaders.append('Accept-Encoding', 'deflate'); // Añade nueva cabecera
myHeaders.append('Accept-Encoding', 'gzip'); // Añade nuevo valor a cabecera

// Muestra los pares clave/valor
for (var pair of myHeaders.entries()) {
   console.log(pair[0]+ ': '+ pair[1]);
}
```

```
accept-encoding: deflate, gzip
content-type: text/xml
vary: Accept-Language
>
```

```
console.log(myHeaders.has('Content-type')); // true

console.log(myHeaders.get('Content-type')); // "text/html"

console.log(myHeaders.delete('Content-type'));

console.log(myHeaders.get('Content-type')); // null
```

Fetch API

Interfaz Request

Representa una petición de un recurso.

Constructor:

```
new Request( input[, init])
```

Parámetros:

- ***input***. Define el recurso a obtener. Puede ser:
  - La `URL` directa al recurso a obtener.
  - Otro objeto `Request` del que se creará una copia.
- ***init***. Parámetro opcional. Permite configurar el tipo de petición:
  - `method`: El método de la petición `get`, `post`, `put`, `del`. Valor por defecto es `get`.
  - `headers`: Cualquier cabecera que se quiera añadir, ya sea mediante un objeto `Headers` o mediante texto.

Fetch API

Interfaz Request

Constructor (*y II*):

Parámetros:

- `init` (*y II*).
  - `body`: Cualquier contenido que se quiera añadir a la petición. Puede ser cualquier objeto del tipo Blob, BufferSource, FormData, USVString, URLSearchParams o ReadableStream. La petición GET no tiene `body`.
  - `mode`: El modo a utilizar para la petición: `cors`, `no-cors`, `same-origin` o `navigate`. El valor por defecto es `cors`, excepto en Chrome.
  - `credentials`: Las credenciales para la petición: `omit`, `same-origin` o `include`. El valor por defecto es `omit`, excepto en Chrome.
  - `cache`: Modo de caché a utilizar: default, no-store, reload, no-cache, force-cache, only-if-cached.
  - `referer`: String especificando `no-referer`, `client` o una `URL`. El valor por defecto es `client`.
  - `integrity`: Contiene el valor de `subresource integrity` de la petición.

Fetch API

Interfaz Request

Propiedades:

- **`.method`**: Contiene el método de la petición (`GET`, `POST`, ...).
- **`.url`**: Contiene la `URL` de la petición.
- **`.headers`**: Contiene el objeto `Headers` asociado a la petición.
- **`.referrer`**: Contiene la URL que hizo la petición.
- **`.referrerPolicy`**: Permite especificar qué información referente a `referrer` debe enviarse en la petición.
- **`.mode`**: Contiene el modo de la petición (`cors`, `no-cors`, `same-origin`, `navigate`).
- **`.credentials`**: Credenciales de la petición (`omit`, `same-origin`, `include`).
- **`.redirect`**: Contiene el modo en que se deben hacer las redirecciones: `follow`, `error`, `manual`.
- **`.integrity`**: Contiene el valor de `subresource integrity` de la petición.
- **`.cache`**: Contiene el modo de caché de la petición.
- **`.body`**: Permite acceder al contenido del cuerpo de la petición en modo como un objeto ReadableStream.

Fetch API

Interfaz Request

Propiedades (*y //*):

- **`.bodyUsed`**: Almacena un objeto `Boolean` que indica si el cuerpo ya ha sido usado/leído en una respuesta.

Métodos:

- **`.clone()`**: Crea una copia del objeto `Request` actual.
- **`.arrayBuffer()`**: Devuelve una promesa que resuelve con una representación `ArrayBuffer` del cuerpo de la petición.
- **`.blob()`**: Devuelve una promesa que resuelve con una representación `Blob` del cuerpo de la petición.
- **`.formData()`**: Devuelve una promesa que resuelve con una representación `FormData` del cuerpo de la petición.
- **`.json()`**: Devuelve una promesa que resuelve con una representación `JSON` del cuerpo de la petición.
- **`.text()`**: Devuelve una promesa que resuelve con una representación `USVString` del cuerpo de la petición.

Fetch API

Interfaz Request

Ejemplo:

```
const myRequest = new Request('http://localhost/flowers.jpg');

const myURL = myRequest.url; // http://localhost/flowers.jpg
const myMethod = myRequest.method; // GET
const myCred = myRequest.credentials; // omit
const bodyUsed = myRequest.bodyUsed; // false
```

```
const myRequest = new Request('http://localhost/api',
                              {method: 'POST', body: '{"foo":"bar"}'});
const myURL = myRequest.url; // http://localhost/api
const myMethod = myRequest.method; // POST
const myCred = myRequest.credentials; // omit
const bodyUsed = myRequest.bodyUsed; // true
```

Fetch API

Interfaz Response

Representa la respuesta a una petición de un recurso.

Propiedades:

- **`.body`**: Permite acceder al contenido del cuerpo de la respuesta en modo como un objeto ReadableStream.
- **`.bodyUsed`**: Almacena un objeto `Boolean` que indica si el cuerpo ya ha sido usado/leído en una respuesta.
- **`.headers`**: Contiene el objeto `Headers` asociado a la respuesta.
- **`.ok`**: Contiene un objeto `Boolean` que toma valor `true` si la respuesta fue correcta (código `HTTP` de estado `200-299`), o `false` si no lo fue.
- **`.redirected`**: Contiene un objeto `Boolean` que toma valor `true` si la respuesta fue redireccionada, o `false` si no lo fue.
- **`.status`**: Contiene el código `HTTP` de estado de respuesta (por ejemplo, `200`).
- **`.statusText`**: Contiene el mensaje asociado al código `HTTP` de estado de la respuesta (por ejemplo, `OK` para código `200`).
- **`.type`**: Contiene el tipo de respuesta (`basic`, `cors`, `error`, `opaque`).
- **`.url`**: Contiene la `URL` de la respuesta (incluido redireccionamiento).

Fetch API

Interfaz Response

Métodos:

- **`.arrayBuffer()`**: Devuelve una promesa que resuelve con una representación `ArrayBuffer` del cuerpo de la respuesta.
- **`.blob()`**: Devuelve una promesa que resuelve con una representación `Blob` del cuerpo de la respuesta.
- **`.clone()`**: Crea y devuelve un nuevo objeto `Response` que es una copia del original.
- **`.formData()`**: Devuelve una promesa que resuelve con una representación `FormData` del cuerpo de la respuesta.
- **`.json()`**: Devuelve una promesa que resuelve con una representación `JSON` del cuerpo de la respuesta.
- **`.text()`**: Devuelve una promesa que resuelve con una representación `USVString` del cuerpo de la respuesta.

Fetch API

Cómo hacer la petición

Fetch API proporciona el método global **`fetch()`** para obtener, de forma asíncrona, recursos de la red de una manera fácil y lógica.

- **`fetch()`** devuelve una promesa que siempre resuelve (*`resolve`*), nunca hará *`reject`* independientemente del código de error `HTTP` (`404`, `500`, ...) de la respuesta.
- En el caso de producirse un código de error `HTTP`, `fetch()` resolverá con la propiedad `ok` a `false`.
- El único caso en que `fetch()` no hará `resolve` es si se produce un fallo en la red. En este caso se puede utilizar **`.catch()`** para capturar el error.

Fetch API

Cómo hacer la petición

Se invoca a fetch de cualquiera de las dos siguientes maneras:

```
fetch( objeto_Request)
fetch( input [, init])
```

Parámetros posibles:

- Objeto `Request` con la configuración de la petición.
- Mismos parámetros que un objeto `Request`.

Fetch API

Cómo hacer la petición

Ejemplo: Petición GET

```
function peticionFetchAPI_GET() {
  var url  = 'rest/comentario?u=5';

  // fetch usa el método GET por defecto
  fetch(url).then(function(response){
    if(response.ok){ // if(response.status==200)
      response.json().then(function(data) { // se tiene la respuesta y con
        console.log(data); // json() se recoge en data como objeto javascript
      });
    }
    else{
      console.log('Error(' + response.status + '): ' + response.statusText);
      return;
    }
  }).catch(function(err) {
        console.log('Fetch Error: ' + err);
  });
}
```

Fetch API

Cómo hacer la petición

Ejemplo: Petición POST

```
function peticionFetchAPI_POST( form_HTML, clave ) {
  var url  = 'rest/login',
      fd   = new FormData(form_HTML), // se utiliza un objeto FormData()
      init = { method:'post', body:fd, headers:{'Authorization':clave} };

  fetch(url,init).then(function(response){ // fetch utilizará el método POST
    if(response.ok){ // if(response.status==200)
      response.json().then(function(data) { // se tiene la respuesta
        console.log('Nombre:' + data.nombre); // data es un objeto javascript
      });
    }
    else{
      console.log('Error(' + response.status + '): ' + response.statusText);
      return;
    }
  }).catch(function(err) {
        console.log('Fetch Error: ' + err);
  });
}
```

Excepciones

Excepciones

- JavaScript dispone de un tratamiento de excepciones muy parecido al de otros lenguajes de programación.
- Se trata de una estructura que utiliza las palabras reservadas **`try`**, **`catch`** y **`finally`**.

```
try{
   // bloque de instrucciones en el que se van
   // a controlar las excepciones
   instrucciones
}catch(excepción){
   // bloque de instrucciones a ejecutar en caso de producirse
   // una excepción en el bloque de instrucciones del try
   instrucciones
}finally{
   // bloque de instrucciones que se ejecutará siempre,
   // independientemente de si se produce una excepción o no
   instrucciones
}
```

Excepciones

- Un bloque **try** debe ir acompañado de un bloque **catch** y/o de un bloque **finally**.

```
var article = document.createElement('article');

try{ // bloque de código a controlar
  var html = '<p>Maecenas interdum tincidunt nisl, eu.</p>';
  article.innerHTML = html;
  var b = document.body.querySelector('section:first-child');
  b.appendChild(article);
}catch(e){ // si hay alguna excepción, la muestra por la consola
  console.log('ERROR: ' + e);
  // console.error(e); // alternativa
}finally{ // siempre muestra en la consola el contenido del article
  console.log('article.innerHTML:' + article.innerHTML);
}
```

Excepciones

- Se pueden generar excepciones manualmente mediante la palabra reservada **throw**.

Ejemplo:

```
var dividendo = Math.round(10 * Math.random()); // número entre 0 y 10
var divisor   = Math.round(2  * Math.random()); // número entre 0 y 2

try{ // bloque de código a controlar
  if(divisor==0){
    resultado = 'División por cero!!!';
    throw new Error('El divisor es cero'); // se lanza la excepción
    // throw 'El divisor es cero'; // también es posible
  }
  // las siguientes dos líneas de código se ejecutarán si no
  // se lanza la excepción
  resultado = dividendo / divisor;
}catch(e){ // si hay alguna excepción, la muestra por la consola
  console.log('ERROR: ' + e);
}finally{ // siempre muestra en la consola resultado
  console.log('Resultado de ' + dividendo + '/' + divisor + ': ' + resultado);
}
```

Promises

Promises

- Una **promesa** es un objeto que ejecuta una tarea asíncrona o diferida y permite definir código a ejecutar para cuando la tarea finaliza correctamente (**se resuelve**) y código para cuando no finaliza correctamente (**se rechaza**).
- Cuando la promesa finaliza, **puede devolver información** que es utilizada por el código a ejecutar asociado al resultado de la misma.
- Una promesa puede estar en los siguientes **estados**:
  - ✓ `pending` (pendiente): estado inicial, no cumplida o rechazada.
  - ✓ `fulfilled` (cumplida): operación satisfactoria.
  - ✓ `rejected` (rechazada): operación fallida.
  - ✓ `settled` (finalizada): operación cumplida o rechazada.
- Las promesas **se pueden encadenar**, es decir, cuando se resuelven o se rechazan pueden devolver otra promesa.

Promises

Sintaxis

```
new Promise( function(resolve, reject) { ... } );
```

- La función que se pasa como argumento al constructor contiene el código de la tarea a ejecutar por la promesa.
- Dentro de ese código, si la tarea finaliza correctamente, se llama a la función *resolve*. En caso contrario se llama a la función *reject*. Estas funciones admiten un parámetro mediante el que se le puede pasar información del resultado.
- El resultado de la promesa lo recoger su método **.then** que invocará al correspondiente código asociado, en función del resultado.
- También se puede utilizar su método **.catch** para capturar cualquier posible error que impidiera su finalización.

Promises

Sintaxis

Ejemplo:

```
var promesa = new Promise(
  function(resolve, reject){
    ...
    if(tarea_finalizada_con_éxito)
      resolve('éxito');
    else
      reject('error');
  }
);
```

El valor devuelto por las funciones **resolve()** y **reject()** puede ser cualquier objeto de JavaScript

```
...
promesa.then( function(valor){ // finaliza correctamente (se resuelve)
    // se ejecutará cuando se llame a resolve() desde la promesa
    ...
}, function(valor){ // no finaliza correctamente (se rechaza)
    // se ejecutará cuando se llame a reject() desde la promesa
    ...
});
```

Promises

Ejemplo:

```
var promesa = new Promise(function(resolve,reject){
  window.setTimeout(function(){ // sólo para hacerlo asíncrono
    let numero = ( Math.trunc( Math.random() * 9 ) + 1 );
    if( numero % 2 == 0 ){ // el número aleatorio es par ...
      resolve(numero); // ... se cumple la promesa
    }else{ // el número aleatorio es impar ...
      reject(numero); // ... se rechaza la promesa
    }
  }, Math.random() * 2000 + 1000);
});

promesa.then(function(valor){ // se ha cumplido la promesa
    console.log('La promesa se ha cumplido: ' + valor);
  }, function(valor){ // se ha rechazado la promesa
    console.log('La promesa ha sido rechazada: ' + valor);
  }
);

console.log('Promesa lanzada. A esperar resultado ...');
```

Promises

Ejemplo: Petición AJAX

```
function peticionAjax(url, timeout){
  return new Promise(function(resolve,reject){
    let xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);
    xhr.timeout = timeout;
    xhr.onload = function(){
      let r = JSON.parse(xhr.responseText);
      if(r.RESULTADO=='OK')
        resolve(r);
      else
        reject(r);
    };
    xhr.send();
  });
}

peticionAjax('rest/receta/?t=tomate', 3000).then(function(datos){
  // todo OK: r.RESULTADO=='OK'
  ...
},function(datos){  // error: r.RESULTADO!='OK'
  ...
});
```

Promises

Múltiples promesas

Se puede esperar al **cumplimiento de múltiples promesas**, combinadas en un array que se pasa al objeto global Promise y que se resolverá cuando todas las promesas se hayan resuelto (satisfactoriamente o no). Para ello es necesario utilizar el método `.all` del objeto **`Promise`**.

Ejemplo:

```
function fetchAsync(url, timeout, onData, onError){ … }

function fetchPromised(url, timeout){
  return new Promise(function(resolve, reject){
    fetchAsync(url, timeout, resolve, reject);
  });
}
```

```
Promise.all([
  fetchPromised("http://backend/foo.txt", 500),
  fetchPromised("http://backend/bar.txt", 500),
  fetchPromised("http://backend/baz.txt", 500)
]).then(function(data){
    let [ foo, bar, baz ] = data;
    console.log(`Éxito: foo=${foo} bar=${bar} baz=${baz}`)
   },function(err){ console.log(`error: ${err}`) });
```

Promises

Múltiples promesas

Se puede esperar al **cumplimiento de la primera de múltiples promesas**, combinadas en un array que se pasa al objeto global Promise y que se resolverá cuando una de ellas (la primera de todas) se resuelva satisfactoriamente. Para ello es necesario utilizar el método **.any** del objeto **Promise**.

Ejemplo:

```
function fetchAsync(url, timeout, onData, onError){ … }

function fetchPromised(url, timeout){
  return new Promise(function(resolve, reject){
    fetchAsync(url, timeout, resolve, reject);
  });
}
```

```
Promise.any([
  fetchPromised("http://backend/foo.txt", 500),
  fetchPromised("http://backend/bar.txt", 500),
  fetchPromised("http://backend/baz.txt", 500)
]).then(function(data){
    console.log(`Éxito. Valor devuelto: ${data}`)
   },function(err){ console.log(`error: ${err}`) });
```

Temporizadores

Temporizadores

- No son propios de JavaScript
- Están proporcionados por la especificación estándar de HTML
- Permiten la ejecución programada de código
- Se implementan en la interfaz **`WindowTimers`**:

```
interface WindowTimers {
  long setTimeout(función, [long milisegundos,
                  argumentos_para_la_función]);
  void clearTimeout(long id_intervalo);
  long setInterval(función, [long milisegundos,
                   argumentos_para_la_función]);
  void clearInterval(long id_intervalo);
}
```

Temporizadores

Explicación de los métodos de la interfaz

- Los métodos para crear los temporizadores son **`setTimeout()`** y **`setInterval()`**.
  - El primer argumento de estos dos métodos puede ser el nombre de la función, o el *handler* (puntero) a la función, a ejecutar.
  - El segundo argumento es el número de milisegundos a transcurrir antes de lanzar la función indicada como primer argumento.
  - El tercer argumento y siguientes son los argumentos a pasar a la función indicada como primer argumento.
- Los dos métodos devuelven el identificador del temporizador creado. Este identificador, pasado como argumento a los métodos **`clearTimeout()`** y **`clearInterval()`**, permite parar el temporizador.

Temporizadores

Diferencias

- **`setTimeout()`** crea un temporizador que esperará los milisegundos indicados y lanzará la función pasándole los argumentos especificados. El temporizador se disparará una única vez, salvo que se detenga antes invocando al método **`clearTimeout()`**.

- **`setInterval()`** crea un temporizador que cada vez que transcurra el intervalo de tiempo indicado en milisegundos, lanzará la función pasándole los argumentos especificados. El temporizador se disparará indefinidamente hasta que se detenga mediante el método **`clearInterval()`**.

Temporizadores

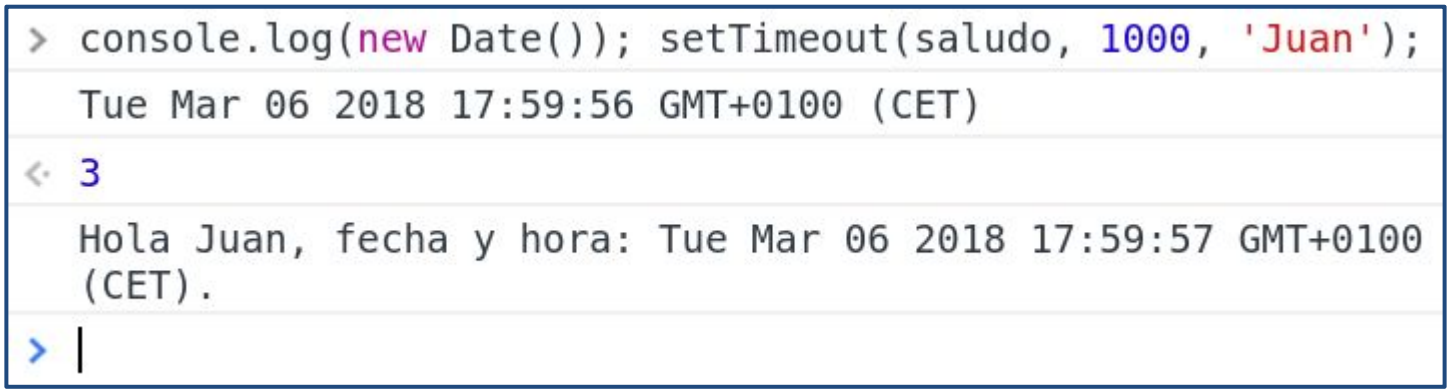
Ejemplos:

```
function saludo(nombre){
    let hora = new Date();

    console.log(`Hola ${nombre}, fecha y hora: ${hora}.`);
}

console.log(new Date()); setTimeout(saludo, 1000, 'Juan');
```

Resultado:



Temporizadores

Ejemplos:

```
console.log(new Date());
t = setInterval(saludo, 1000, 'Juan');
```

Resultado:

```
> console.log(new Date())
  Tue Mar 06 2018 22:57:28 GMT+0100 (CET)
< undefined
> t = setInterval(saludo, 1000, 'Juan');
< 2
  Hola Juan, fecha y hora: Tue Mar 06 2018 22:57:30 GMT+0100 (CET).
  Hola Juan, fecha y hora: Tue Mar 06 2018 22:57:31 GMT+0100 (CET).
  Hola Juan, fecha y hora: Tue Mar 06 2018 22:57:32 GMT+0100 (CET).
  Hola Juan, fecha y hora: Tue Mar 06 2018 22:57:33 GMT+0100 (CET).
  Hola Juan, fecha y hora: Tue Mar 06 2018 22:57:34 GMT+0100 (CET).
  Hola Juan, fecha y hora: Tue Mar 06 2018 22:57:35 GMT+0100 (CET).
> clearInterval(t)
< undefined
>
```

Temporizadores

Otra método para ejecutar código programado

- `window.requestAnimationFrame(callback)`
  - Está pensado para realizar animaciones en la interfaz de usuario mediante javascript.
  - Se ejecuta 60 veces por segundo.
  - Permite ejecutar código programado de forma más eficiente, en comparación con `setTimeout()` y `setInterval()`, dejando que el navegador gestione la animación.
  - Se utiliza de forma similar a `setTimeout()`.

Temporizadores

Otra método para ejecutar código programado

- `long requestAnimationFrame(callback)`.
  - Cada vez que se ejecuta `requestAnimationFrame()` se llama a la función *`callback`*. La función *`callback`* recibe como argumento un valor `DOMHighResTimeStamp` que es la cantidad de milisegundos transcurridos desde la última vez que se cargó el documento html en el navegador. La precisión de este valor es de microsegundos.
  - Permite al navegador optimizar las animaciones, haciéndolas más suaves, y hacer más eficiente el uso de los recursos.
  - Las animaciones se detienen cuando la pestaña (o la ventana) del navegador no es visible. Esto significa un ahorro notable del uso de memoria, de CPU, de GPU y, por consiguiente, de batería.

Temporizadores

Otra método para ejecutar código programado

- `void` **`cancelAnimationFrame`**`(`*`idTemporizador`*`)`.
  - Permite detener la animación lanzada, antes de que se lleve a cabo, pasando como argumento el valor devuelto por `requestAnimationFrame()`.

Ejemplo:

```
...
let inicio = null;

function animar(timestamp) {
  if(!inicio) inicio = timestamp;
  let lapso_tiempo = timestamp - inicio;
  if(lapso_tiempo < 2000) requestAnimationFrame(animar);
}

...
requestAnimationFrame(animar);
```