

TEMA 8

SERVICIOS WEB Y AJAX


APLICACIONES WEB - GIS - CURSO 2019/20

Marina de la Cruz [marina.cruz@ucm.es]
Dpto de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid



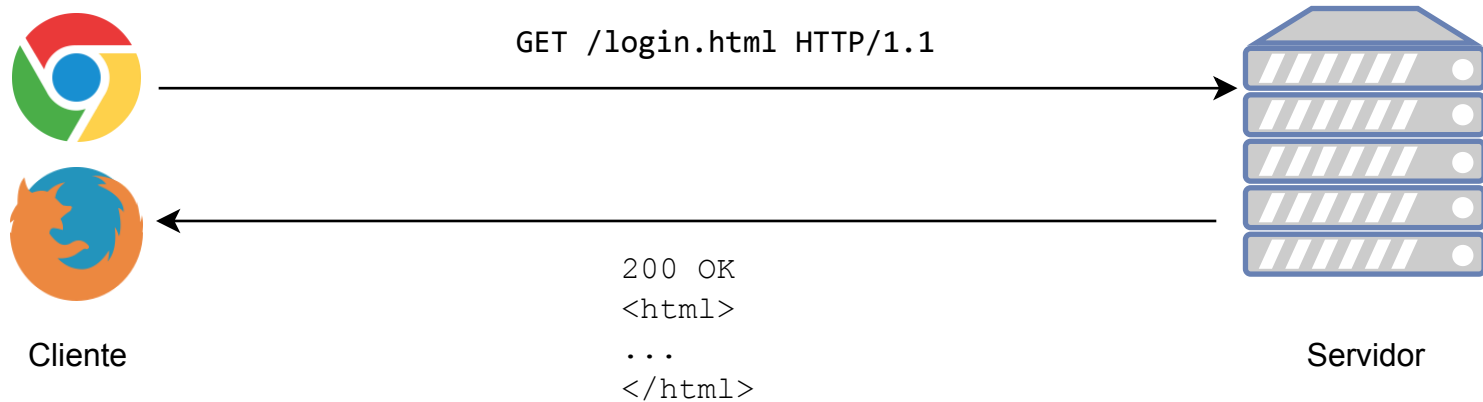
Esta obra está bajo una
Licencia CC BY-NC-SA 4.0 Internacional.

Este documento está basado en <https://manuelmontenegro.github.io/AW-2017-18/08.html#/> de Manuel Montenegro [montenegro@fdi.ucm.es] bajo una Licencia CC BY-NC-SA 4.0 Internacional.

- 
1. INTRODUCCIÓN
 2. FORMATOS XML Y JSON
 3. SERVICIOS REST
 4. PETICIONES AJAX

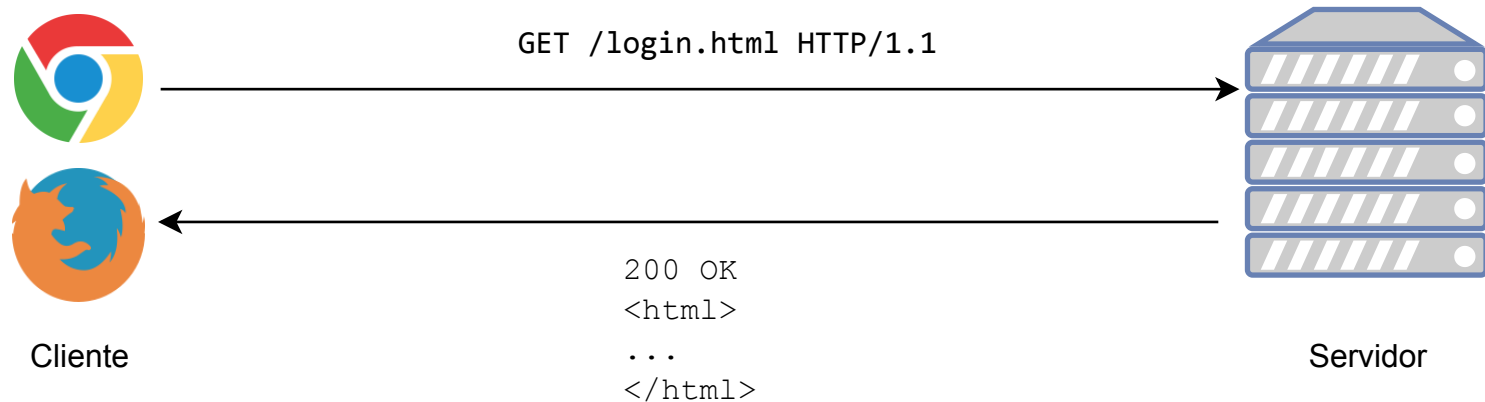
INTRODUCCIÓN

INTRODUCCIÓN



Hasta ahora hemos estudiado el protocolo HTTP utilizado en la comunicación entre un **navegador web** y un **servidor**.

INTRODUCCIÓN



El servidor devolvía al cliente: documentos HTML (**.html**), imágenes (**.png**, **.jpg**, **.svg**), código Javascript para ser ejecutado en navegador (**.js**), etc.

El destinatario de estos contenidos es el **navegador web**.

INTRODUCCIÓN



¿Y si utilizamos el mismo esquema y **el mismo protocolo HTTP** para que el servidor devuelva **otro tipo de contenido**?

El contenido estaría destinado a ser consumido por **un programa que tome el papel del cliente**; no necesariamente un navegador.

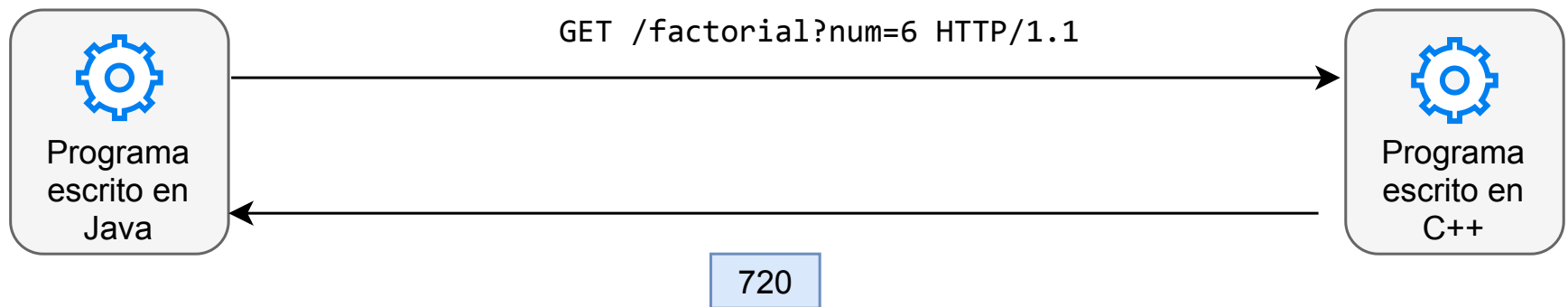
SERVICIOS WEB

Un **servicio web** es un servicio ofrecido por un dispositivo a otros dispositivos a través de la web.

Los servicios web están diseñados para la comunicación máquina a máquina.

La comunicación requerida por el servicio se realiza normalmente mediante HTTP.

La principal ventaja es la **interoperabilidad** entre distintos componentes, independientemente del lenguaje de programación en el que estén implementados y de la máquina en la que se ejecuten.



TIPOS DE SERVICIOS WEB

- **REST** (*REpresentational State Transfer*)

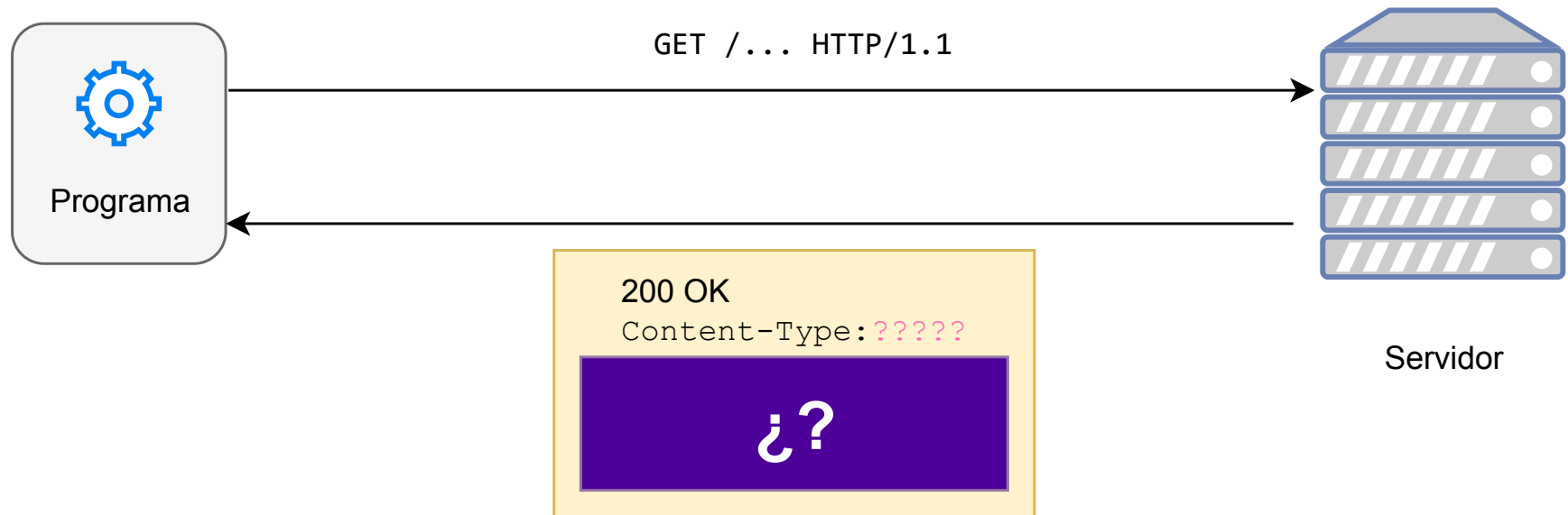
Utilizan los distintos tipos de peticiones HTTP (**GET**, **POST**, **PUT**, **DELETE**) para especificar las operaciones a realizar. Las operaciones se realizan sobre recursos, cada uno identificado mediante una URL.

- **SOAP** (*Simple Object Access Protocol*)

Utilizan mensajes en formato XML para especificar el tipo de operación que se desea realizar y sus parámetros.

FORMATOS XML Y JSON

FORMATOS DE INTERCAMBIO DE DATOS



Hemos visto que el servidor responde al programa cliente con la información que este último ha pedido.
¿En qué formato se transmite esta información?

REQUISITOS DE UN FORMATO DE INTERCAMBIO DE DATOS

- Debe ser lo suficientemente **genérico** como para poder representar cualquier tipo de respuesta.
- Debe de ser fácilmente analizable por un programa.
... y si es legible, mejor.

Formatos de intercambio más populares:

- ***XML*** (*eXtensible Markup Language*)
- ***JSON*** (*JavaScript Object Notation*)

Nosotros utilizaremos principalmente JSON.

FORMATO XML

```
<agenda>

  <contacto id="c1">
    <nombre>David Jiménez</nombre>
    <fecha-nacimiento>20/01/1989</fecha-nacimiento>
    <telefonos>
      <tel tipo="casa">917371235</tel>
      <tel tipo="trabajo">914483124</tel>
    </telefonos>
  </contacto>

  <contacto id="c2">
    <nombre>Luis García</nombre>
    <fecha-nacimiento>15/03/1979</fecha-nacimiento>
    <telefonos>
    </telefonos>
  </contacto>

</agenda>
```

Sintaxis similar a la de HTML, con ciertas restricciones:

- Toda etiqueta de inicio tiene una etiqueta de cierre, y las etiquetas han de estar correctamente anidadas.
- Los atributos de etiquetas están delimitados por comillas dobles.
- Existe un único elemento raíz.

Cualquier documento que cumpla estas tres condiciones
está **bien formado**.

FORMATO JSON

```
{
  "c1": {
    "nombre": "David Jiménez",
    "fechaNacimiento": "20/01/1989",
    "telefonos": [
      { "tipo": "casa", "numero": 917371235 },
      { "tipo": "trabajo", "numero": 914483124 }
    ]
  },
  "c2": {
    "nombre": "Luis García",
    "fechaNacimiento": "15/03/1979",
    "telefonos" : []
  }
}
```

Sintaxis similar a los objetos de Javascript.

... con una excepción: **los nombres de los atributos van siempre delimitados por comillas.**

Incorrecto:

```
{ nombre: "Lucía", edad: 20 }
```

Correcto:

```
{ "nombre": "Lucía", "edad": 20 }
```


SINTAXIS JSON

- **Números**

```
34    2.40    -2.0e+16
```

- **Cadenas**

```
"Hola"    "Esto es una cadena"
```

- **Booleanos:** `true`, `false`

- Valor nulo: `null`

- **Arrays**

```
[ 23, "Pepe", 3.45, null, 2 ]
```

- **Objetos** (listas de pares clave-valor)

```
{ "nombre" : "Javier", "asignaturas": ["MDLM", "ABD", "MMI"] }
```

UTILIZAR JSON EN JAVASCRIPT

Las siguientes funciones están disponibles tanto en el servidor (Node) como en el cliente (navegadores).

- **JSON.parse(*objetoJSON*)**

Convierte un texto en formato JSON en un objeto Javascript.

- **JSON.stringify(*objetoJavaScript*)**

Obtiene la representación JSON de un objeto Javascript.

```
let obj = [ {x: 13, y: 12}, {x: 0, y: 0} ];
JSON.stringify(obj);
// → Devuelve la cadena ' [{"x":13,"y":12},{ "x":0,"y":0}] '

JSON.parse(' [{"x":13,"y":12},{ "x":0,"y":0}] ');
// → Devuelve el objeto [ {x: 13, y: 12}, {x: 0, y: 0} ]
```

SERVICIOS REST

SERVICIOS DE TIPO REST

Utilizan la semántica de los métodos HTTP (**GET**, **POST**, etc.) para indicar la acción a realizar sobre un determinado recurso, identificado por una URI.

Principios básicos:

1. Todo es un recurso.
2. Los recursos están identificados por URIs.
3. Se utilizan los verbos HTTP estándar.
4. Un recurso puede representarse de múltiples formas.
5. La comunicación con los servicios web no tiene estado.

RECURSOS Y URIs

Cada recurso tiene una URI que lo identifica.

Ejemplos:

- `/contactos/elisa`
- `/contactos/elisa/telefonos/casa`
- `/contactos`
- `/calculadora/factorial`

Las URIs no hacen referencia necesariamente a un fichero físico en el servidor, sino a un recurso «abstracto»

MÉTODOS HTTP

Suelen utilizarse las acciones **GET**, **POST**, **PUT** y **DELETE**, que se corresponden con las operaciones CRUD habituales.

CRUD = *Create, Read, Update, Delete*

Método HTTP	Semántica
GET	Leer un recurso
POST	Crear un nuevo recurso
PUT	Actualizar un recurso existente
DELETE	Eliminar un recurso

EJEMPLOS

- Acceder al contacto cuyo identificador es **elisa**
GET /contactos/elisa
- Calcular factorial
GET /calculadora/factorial?num=6
- Añadir un nuevo contacto
POST /contactos
- Actualizar el contacto con identificador **elisa**
PUT /contactos/elisa

La descripción semántica de estos recursos es «orientativa».

Nada nos impide utilizar un método para realizar una acción distinta a la especificada por su semántica.

No obstante, algunos métodos imponen de manera implícita ciertas propiedades, por ejemplo:

- El método **GET** no altera el estado del recurso al que hace referencia.

CÓDIGOS DE RESPUESTA

Método	Código de respuesta
GET	200 OK - Tuvo éxito
	404 Not Found - Recurso no encontrado
	500 Internal Server Error - Otros errores
POST	201 Created - Tuvo éxito
	500 Internal Server Error - Otros
PUT	201 OK - Tuvo éxito
	500 Internal Server Error - Otros
DELETE	200 OK, 204 No Content - Tuvo éxito
	404 Not Found - Recurso no existe
	500 Internal Server Error - Otros

REPRESENTACIONES DE UN RECURSO

Un recurso puede ser representado de distintas formas.

Representación JSON:

```
{  
  "id": "elisa",  
  "nombre": "Elisa Trujillo",  
  "telefonos": [{"casa": "912382722"}]  
}
```

Representación XML:

```
<contacto id="elisa">  
  <nombre>Elisa Trujillo</nombre>  
  <telefonos>  
    <tel tipo="casa">912382722</tel>  
  </telefonos>  
</contacto>
```

En las peticiones de tipo **GET** el cliente especifica en la **cabecera** el tipo de representación que desea recibir:

```
Accept: application/json
```

```
Accept: text/xml
```

En las peticiones de tipo **POST** o **PUT**, el cliente especifica en la **cabecera** de la petición la representación del nuevo objeto.

```
Content-Type: application/json
```

```
Content-Type: text/xml
```

AUSENCIA DE ESTADO

En los servicios web de tipo REST se respeta la carencia de estado propia del protocolo HTTP.

IMPLEMENTACIÓN DE SERVICIOS WEB DE TIPO REST CON EXPRESS.JS

Se puede utilizar el framework *Express.js* para implementar servicios web de tipo REST.

Supongamos un objeto de la clase *Application*:

```
let app = express();
```

Ya conocemos los métodos *get()* y *post()* para definir funciones que manejan las peticiones GET y POST.

- *app.get(url, funcionManejadora)*
- *app.post(url, funcionManejadora)*

Existen sendos métodos `put()` y `delete()` para manejar las peticiones PUT y DELETE

- `app.put(url, funcionManejadora)`
- `app.delete(url, funcionManejadora)`

Al igual que en los métodos `get()` y `post()`, es posible añadir middleware específico en cada petición:

```
app.put("/contactos/:id",  
        middleware_1, middleware_2,  
        function(request, response) {  
            ...  
        }  
    );
```

Middleware intermedio

EJEMPLO: AGENDA DE CONTACTOS

La agenda de contactos está almacenada en un array de objetos, donde cada uno de ellos contiene un nombre y un número de teléfono:

```
let agenda = [  
  { nombre: "Juan", telefono: "89731982" },  
  { nombre: "Carmen", telefono: "28329828" },  
  { nombre: "David", telefono: "827272728" }  
];
```

El objetivo es **diseñar un servicio web de tipo REST** para que puedan realizarse las siguientes operaciones:

1. Obtener todos los objetos.
2. Obtener un objeto a partir de su índice.
3. Añadir un objeto.
4. Eliminar un objeto.
5. Actualizar un objeto.

1. OBTENER TODOS LOS OBJETOS

Método	GET
URL	/contactos
Parám. entrada	Ninguno
Códigos respuesta	200 si tuvo éxito, 500 en caso de error.
Tipos resultado	JSON
Resultado	Lista de objetos. Cada objeto tiene dos atributos: nombre y telefono.

IMPLEMENTACIÓN

El método `response.json()` permite devolver una respuesta en formato JSON.

Recibe un objeto Javascript que transformará en JSON.

Es una **operación terminal**. Llama a `end()` automáticamente.

En este caso se devuelve el array `agenda`.

```
app.get("/contactos", function(request, response) {  
  // Por defecto se devuelve el código 200, por  
  // lo que no hace falta indicarlo mediante el  
  // método response.status().  
  response.json(agenda);  
});
```

2. OBTENER UN OBJETO A PARTIR DE SU ÍNDICE

Método	GET
URL	<i>/contactos/:indice</i>
Parám. entrada	Índice del objeto a recuperar (desde 0 hasta la longitud del array menos uno), incluido en la URL.
Códigos respuesta	200 si tuvo éxito, 404 si no se proporciona un índice válido, y 500 en otro caso de error.
Tipos resultado	JSON
Resultado	Un objeto con dos atributos: <i>nombre</i> y <i>telefono</i> .

Existen varias formas de pasar información como parámetro a un servicio:

- Mediante **URLs paramétricas**
 - Por ejemplo: `/contactos/:indice`
 - Se acceden mediante `request.params`
 - Por ejemplo, al acceder a la URL `/contactos/2`, la variable `request.params.indice` toma el valor "2".
- Mediante ***query strings***
 - Por ejemplo: `/contactos?indice=2`
 - Se acceden mediante `request.query`
 - Por ejemplo `request.query.indice` vale "2".

En el ejemplo se utilizará una URL paramétrica:

```
app.get("/contactos/:indice", function(request, response) {  
  let indice = Number(request.params.indice);  
  if (!isNaN(indice) && agenda[indice] !== undefined) {  
    // El parámetro "indice" es un número y  
    // está dentro de los límites del array.  
    // Se devuelve el elemento correspondiente.  
    let elem = agenda[indice];  
    response.json(elem);  
  } else {  
    // En caso contrario, se devuelve el error 404  
    response.status(404);  
    response.end();  
  }  
});
```

Si se quisiera pasar el índice mediante *query strings* (p. ej. `/contacto?indice=2`), se utilizaría `request.query`:

```
app.get("/contacto", function(request, response) {  
  let indice = Number(request.query.indice);  
  ...  
  // igual que antes  
  ...  
});
```

3. AÑADIR UN OBJETO

Método	POST
URL	/contactos
Parám. entrada	JSON con el objeto a insertar.
Códigos respuesta	201 si tuvo éxito, y 500 en caso de error.
Tipos resultado	Ninguno
Resultado	Ninguno

¿Cómo se pasa un JSON como parámetro a la petición?

El JSON debe ir en el **cuerpo** de la misma.

TIPOS DE CUERPO DE PETICIÓN

Hasta ahora se han visto dos tipos:

- **application/x-www-form-urlencoded**

Opción por defecto en formularios. El contenido puede ser accedido mediante el middleware **body-parser**.

```
let bodyParser = require("body-parser");  
...  
app.use(bodyParser.urlencoded({extended: false}));
```

- **multipart/form-data**

Utilizado en aquellos formularios que adjuntan ficheros. En este caso se puede usar el middleware **multer**.

Existe un tercer tipo:

- **application/json**

Utilizado para pasar información a servicios web que soporten JSON. En este caso también se puede utilizar el middleware **body-parser**.

```
let bodyParser = require("body-parser");  
...  
app.use(bodyParser.json());
```

El método **bodyParser.json()** devuelve un middleware que convierte el JSON contenido en el cuerpo del mensaje en un objeto Javascript, y lo asigna al atributo **request.body**.

Con este middleware se puede implementar la operación de inserción de elementos al final del array:

```
app.use(bodyParser.json());  
// ...  
  
// Añadir contactos  
app.post("/contactos", function(request, response) {  
    let nuevoElemento = request.body;  
    agenda.push(nuevoElemento);  
    response.status(201);  
    response.end();  
});
```

4. ELIMINAR UN OBJETO

Método	DELETE
URL	<i>/contactos/:indice</i>
Parám. entrada	Índice del elemento a eliminar. Viene incluido en la URL.
Códigos respuesta	200 si tuvo éxito, 404 si el índice no es correcto, y 500 en otro caso de error.
Tipos resultado	Ninguno
Resultado	Ninguno

IMPLEMENTACIÓN

```
app.delete("/contactos/:indice", function(request, response) {  
  let indice = Number(request.params.indice);  
  if (!isNaN(indice) && agenda[indice] !== undefined) {  
    agenda.splice(indice, 1);  
    // Código 200 = OK  
    response.status(200);  
  } else {  
    // Error 404 = Not found  
    response.status(404);  
  }  
  response.end();  
});
```

5. ACTUALIZAR UN OBJETO

Método	PUT
URL	<i>/contactos/:indice</i>
Parám. entrada	Índice del elemento a actualizar. Viene incluido en la URL. Nuevo valor del elemento (objeto con atributos nombre y telefono) dentro del cuerpo del mensaje, en formato JSON.
Códigos respuesta	201 si tuvo éxito, 404 si el índice no es correcto, y 500 en otro caso de error.
Tipos resultado	Ninguno
Resultado	Ninguno

IMPLEMENTACIÓN

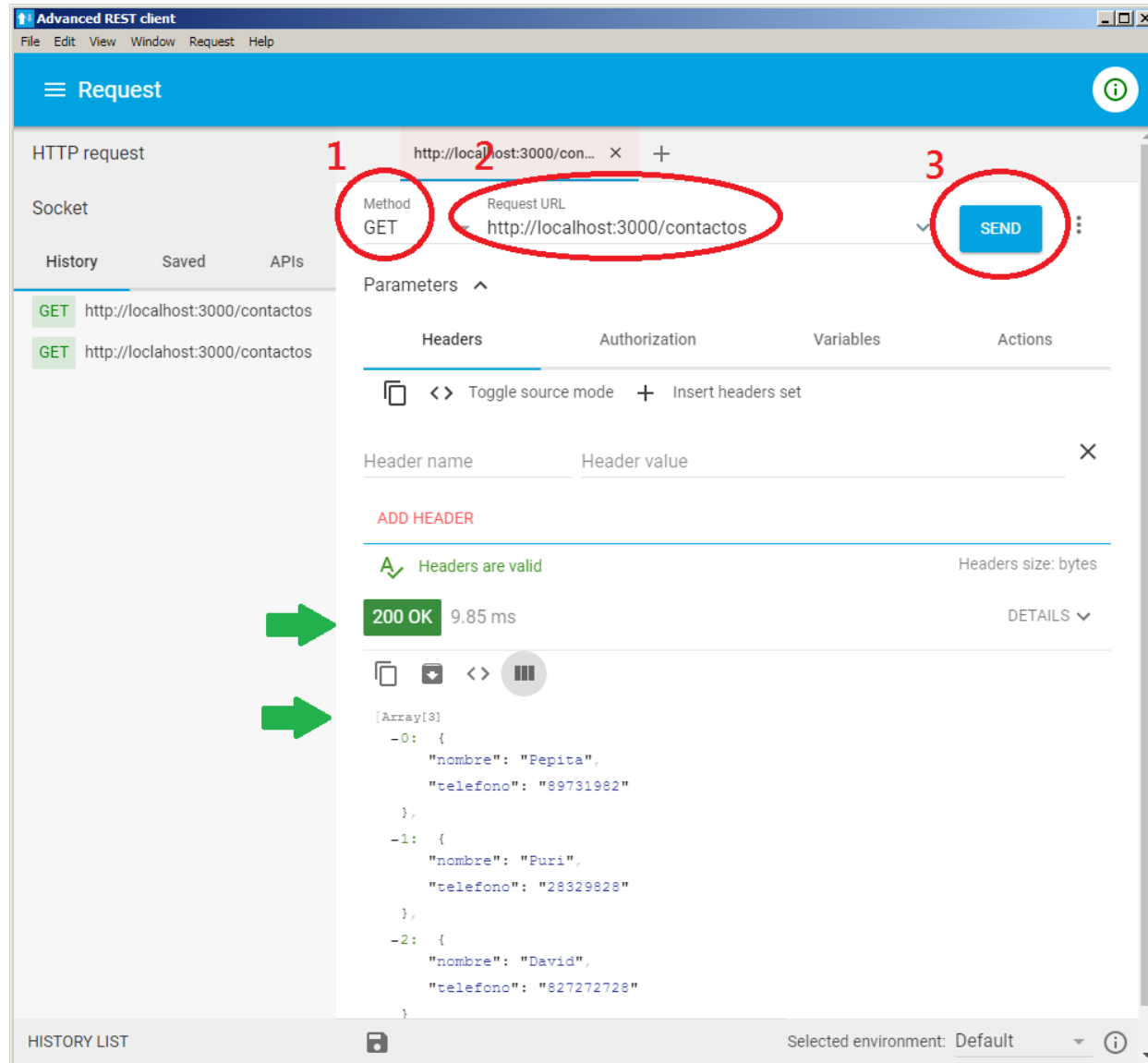
```
app.put("/contactos/:indice", function(request, response) {  
  let indice = Number(request.params.indice);  
  if (!isNaN(indice) && agenda[indice] !== undefined) {  
    agenda[indice] = request.body;  
  } else {  
    // Error 404 = Not Found  
    response.status(404);  
  }  
  response.end();  
});
```

COMPROBAR EL FUNCIONAMIENTO DE UN SERVICIO WEB

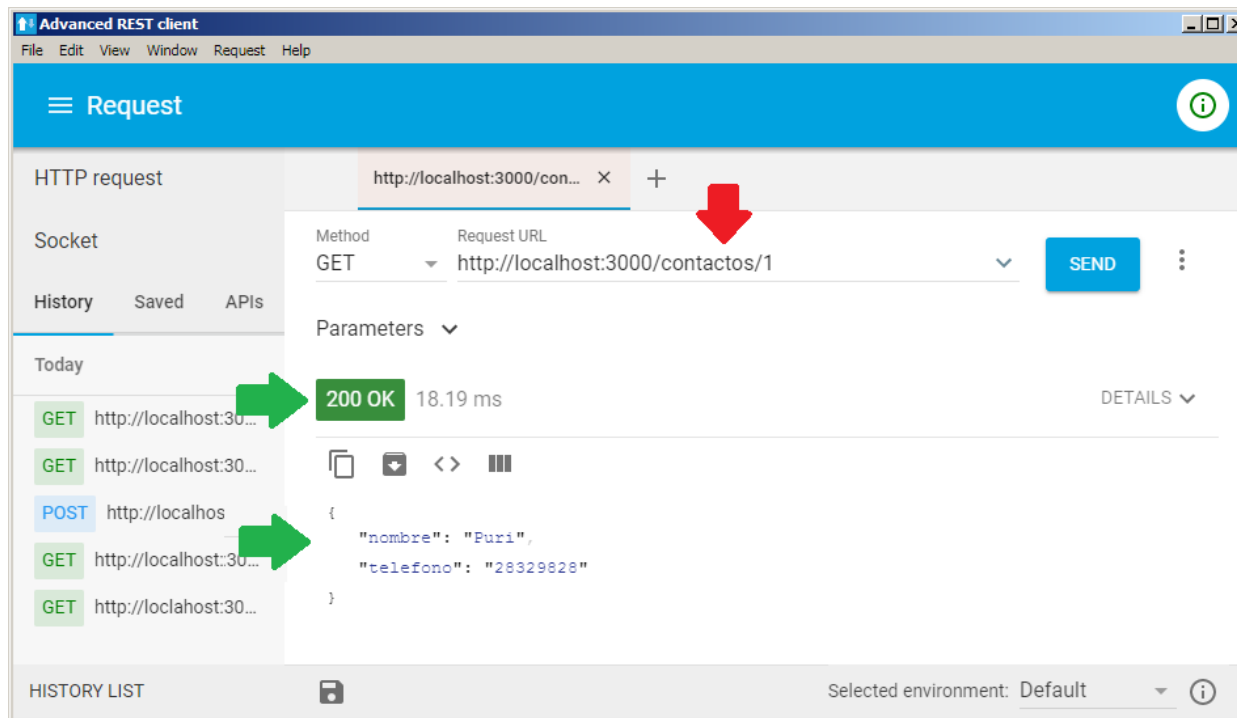
Existen varias herramientas que permiten al usuario realizar cualquier tipo de peticiones HTTP a un servidor.

- En Firefox: **RESTClient**
<http://restclient.net/>
- En Chrome: **Advanced REST Client**
<https://advancedrestclient.com/>

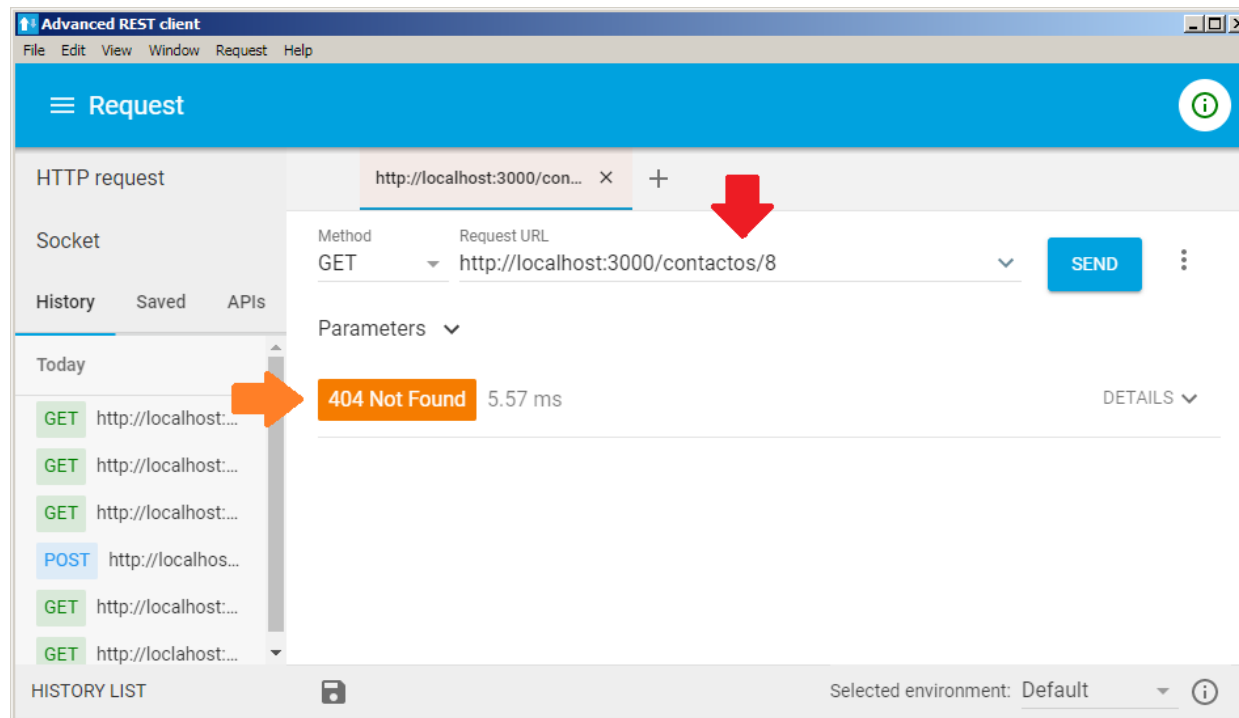
PETICIONES GET EN ADVANCED REST CLIENT (1)



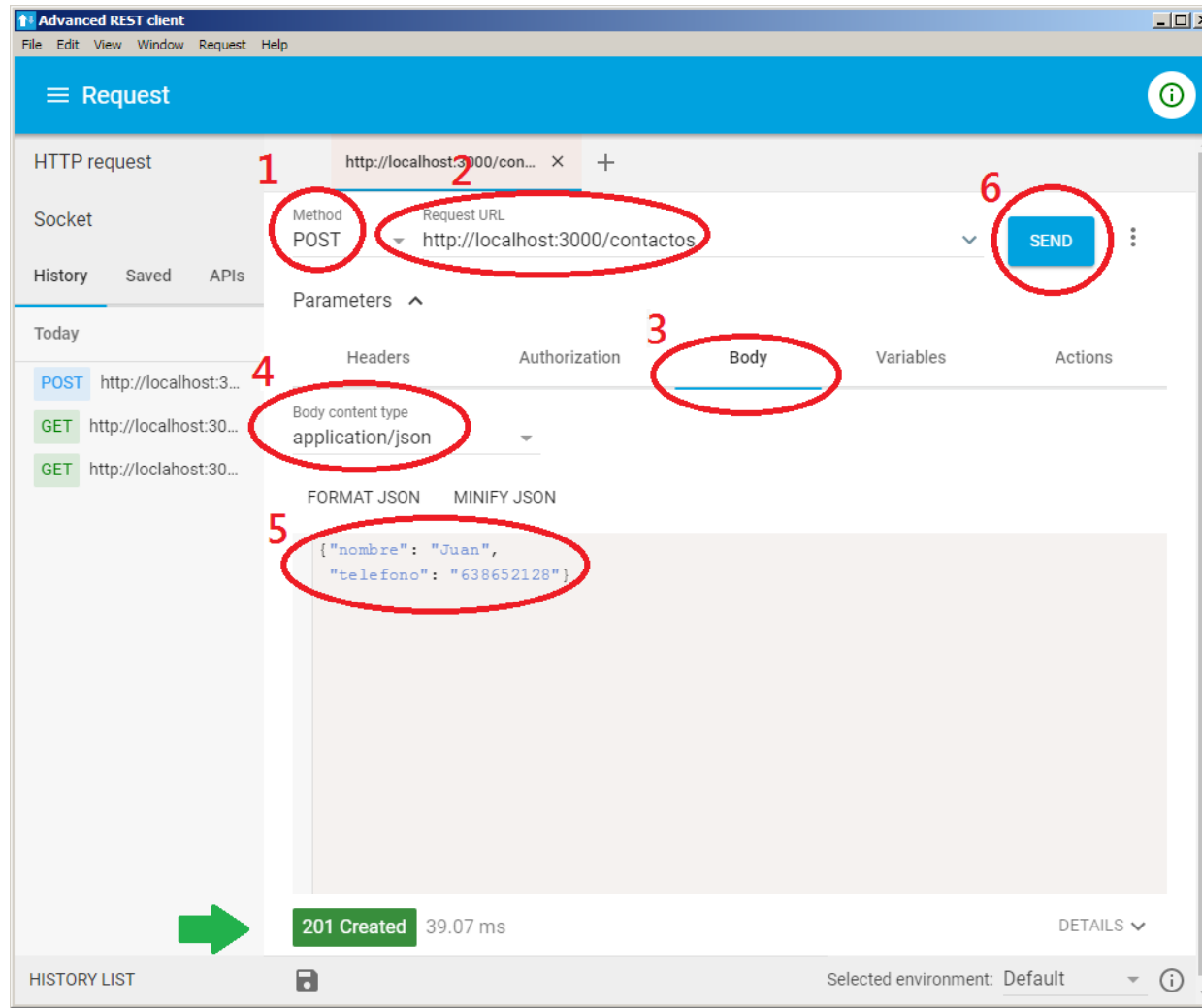
PETICIONES GET EN ADVANCED REST CLIENT (2)



PETICIONES GET EN ADVANCED REST CLIENT (3)



PETICIONES POST EN ADVANCED REST CLIENT



PETICIONES AJAX

PETICIONES AJAX

Hasta ahora hemos contemplado tres formas en las que el navegador realiza una petición HTTP:

- El usuario introduce una URL en la barra de direcciones del navegador. Este último envía una petición **GET** con la URL introducida.
- El usuario hace clic en un enlace. El navegador realiza una petición **GET** con la URL indicada en el atributo **href**.
- El usuario hace clic en el botón *Submit* del formulario. El navegador realiza una petición **GET** o **POST** (según el atributo **method** del formulario) a la URL indicada en el atributo **action** del formulario.

Existe otro mecanismo mediante el cual es posible realizar peticiones HTTP mediante **Javascript** en el navegador

AJAX = *Asynchronous Javascript and XML*

Anteriormente se utilizaba el formato XML para pasar parámetros a las peticiones AJAX, y para recibir los resultados.

Nosotros utilizaremos JSON.

Existen diferencias entre los distintos navegadores sobre cómo realizar peticiones AJAX.

- Según el estándar W3C: clase `XMLHttpRequest`.
- En Internet Explorer versión ≤ 6 : control ActiveX `Microsoft.XMLHTTP`.

Nosotros realizaremos las peticiones AJAX mediante la librería *jQuery*, que abstrae todas estas diferencias en una única función: `$.ajax()`

LA FUNCIÓN \$.ajax()

`$.ajax(opciones)`

El parámetro *opciones* es un objeto que especifica los detalles de la petición. Contiene los siguientes atributos:

- **method**

Método HTTP a utilizar: GET, POST, PUT, DELETE, etc.

- **url**

URL sobre la que realizar la petición.

- **success, error**

Funciones *callback* llamadas cuando la petición ha tenido éxito o ha fallado, respectivamente.

- **data**

Datos a adjuntar en la petición, ya sea como parámetros de la *query string* (esto es, detrás de la URL en el caso de peticiones GET), o dentro del cuerpo de la petición (en el caso de peticiones POST o PUT).

- **contentType**

Tipo MIME del cuerpo de la petición. Por defecto es **application/x-www-form-urlencoded**.

FUNCIÓN CALLBACK: **success**

Cuando el servidor responde la petición con un código de éxito (**2xx**), se llamará a esta función con tres argumentos:

```
$.ajax({ ...  
  success: function(data, textStatus, jqXHR) {  
    ...  
  }  
});
```

- **data**: Datos contenidos en el cuerpo de la respuesta HTTP. Si el formato de la respuesta es JSON, se convierte automáticamente en un objeto Javascript.
- **statusText**: Cadena que describe el estado de la petición (normalmente "**success**").
- **jqXHR**: Objeto con más información sobre la respuesta (envoltorio del objeto **XMLHttpRequest**).

FUNCIÓN CALLBACK: error

```
$.ajax({ ...  
  error: function(jqXHR, textStatus, errorThrown) {  
    ...  
  }  
});
```

- **jqXHR**: Envoltorio del objeto **XMLHttpRequest**.
- **statusText**: Cadena que describe el estado de la petición ("timeout", "parseerror", "abort", etc.)
- **errorThrown**: Texto adjunto a la respuesta HTTP devuelta ("Not Found", "Internal Server Error", etc).

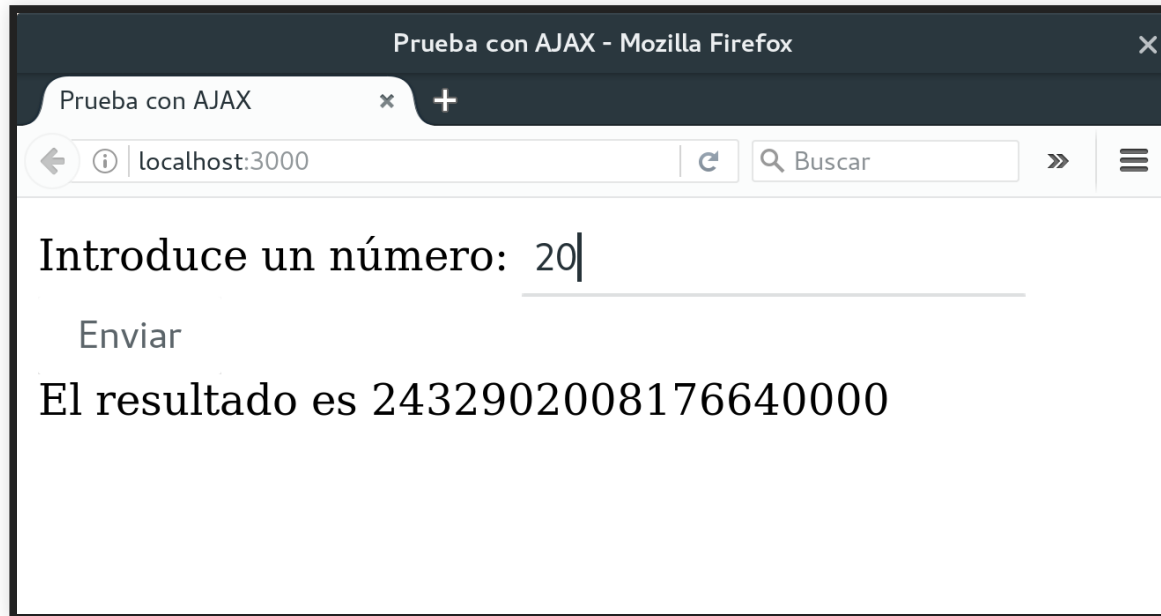
EJEMPLO

Supongamos un servicio web para calcular el factorial de un número dado:

Método	GET
URL	<code>/factorial/:num</code>
Parám. entrada	Número cuyo factorial se desea calcular. Viene incluido en la URL paramétrica (parámetro <code>num</code>)
Códigos respuesta	<code>200</code> si tuvo éxito, <code>400</code> (<i>Bad Request</i>) si no se ha introducido un número, y <code>500</code> en otro caso de error.
Tipos resultado	JSON
Resultado	Un objeto <code>{"result": <i>fact</i>}</code> , donde <i>fact</i> es el factorial calculado.

```
app.get("/factorial/:num", function(request, response) {  
  let numero = Number(request.params.num);  
  if (!isNaN(numero) && numero >= 0) {  
    // Cálculo del factorial  
    let f = 1;  
    for (let i = 2; i <= numero; i++) {  
      f *= i;  
    }  
  
    // Devolución del resultado  
    response.json({ result: f });  
  } else {  
    response.status(400);  
    response.end();  
  }  
});
```

Realizamos llamadas a este servicio desde una página Web:



```
<div>
  <label for="cuadroTexto">Introduce un número: </label>
  <input type="text" id="cuadroTexto">
</div>
<button id="botonEnviar">Enviar</button>
<div id="resultado"></div>
```

Aquí se mostrará el resultado

Código Javascript:

```
$(function() {  
  
    // Cada vez que se pulse el botón de 'Enviar'  
    $("#botonEnviar").on("click", function() {  
  
        // Obtener el valor contenido en el cuadro de texto  
        let valor = $("#cuadroTexto").val();  
  
        // Realizar la petición al servidor  
        $.ajax({  
            method: "GET",  
            url: "/factorial/ " + valor,  
  
            // En caso de éxito, mostrar el resultado  
            // en el documento HTML  
            success: function (data, textStatus, jqXHR) {  
                console.log(textStatus);  
                $("#resultado").text(  
                    "El resultado es " + data.result);  
            },  
  
            // En caso de error, mostrar el error producido  
            error: function (jqXHR, textStatus, errorThrown) {  
                alert("Se ha producido un error: " + errorThrown);  
            }  
        });  
    });  
});
```

PETICIONES GET CON PARÁMETROS

En este ejemplo hemos considerado que la URL era paramétrica:

Por ejemplo: `/factorial/5`

Ahora supongamos que el número viene dado como parámetro en la petición GET:

Por ejemplo: `/factorial?num=5`

Código del servidor:

```
app.get("/factorial", function(request, response) {  
  let numero = Number(request.query.num);  
  // ...  
  // igual que antes  
  // ...  
});
```

Código en el navegador:

```
$.ajax({  
  method: "GET",  
  url: "/factorial",  
  
  // Parámetros de la petición. Como la petición es de tipo GET,  
  // se pasarán como cadena al final de la URL. Por ejemplo:  
  // /factorial?num=5  
  data: {  
    num: valor  
  },  
  
  // ... igual que antes ...  
});
```

Cuando se pasa un objeto Javascript a la opción **data**, este se transforma en una *query string*.

Por ejemplo, **{key1: val1, ..., keyn: valn}** se transforma en **key1=val1&key2=val2&...&keyn=valn**

PETICIONES CON JSON

Ahora suponemos que el número cuyo factorial se desea calcular viene dado en el **cuerpo** de la petición, en forma de un JSON:

```
{ "num": 6 }
```

Con una petición de tipo GET no podemos adjuntar un cuerpo, por lo que utilizamos una petición POST.

En el lado del servidor necesitamos utilizar el middleware **body-parser**, que interpreta el JSON contenido en el cuerpo del mensaje y añade el objeto correspondiente a la propiedad **request.body**:

```
app.use(bodyParser.json());  
// ...  
  
app.post("/factorial", function(request, response) {  
    let numero = Number(request.body.num);  
    // ... igual que antes ...  
});
```

En el lado del cliente, debemos adjuntar la cadena JSON en la propiedad **data**. Para ello utilizamos el método **JSON.stringify()**.

Por otro lado, indicamos mediante **contentType** el tipo MIME de la información enviada en el cuerpo del mensaje.

```
$.ajax({  
  method: "POST",  
  url: "/factorial",  
  contentType: "application/json",  
  data: JSON.stringify({ num: valor }),  
  // ...  
});
```

BIBLIOGRAFÍA

BIBLIOGRAFÍA

- V. Bojinov
RESTful Web API Design with Node.js,
2nd edition
Packt Publishing, 2016
- B. Bibeault, Y. Katz, A. De Rosa
jQuery in Action, 3rd edition
Manning Publications, 2015

