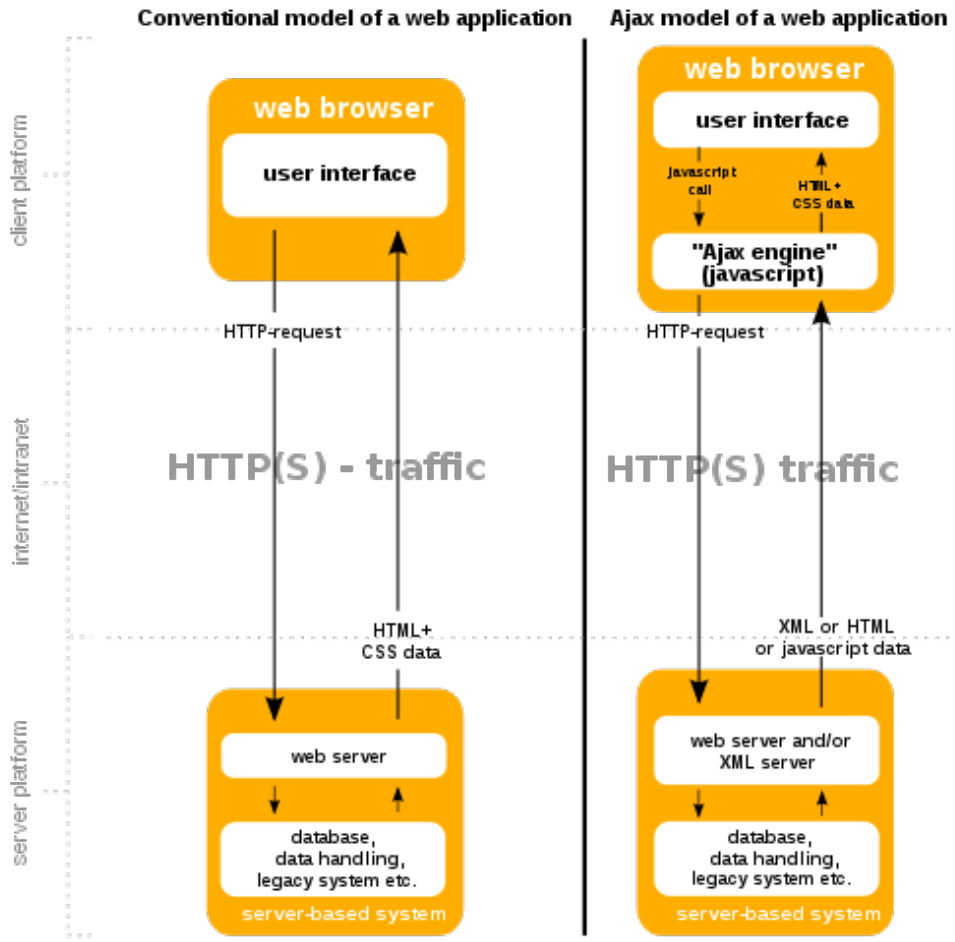
UD06 - Ajax y promesas

1. Introducción

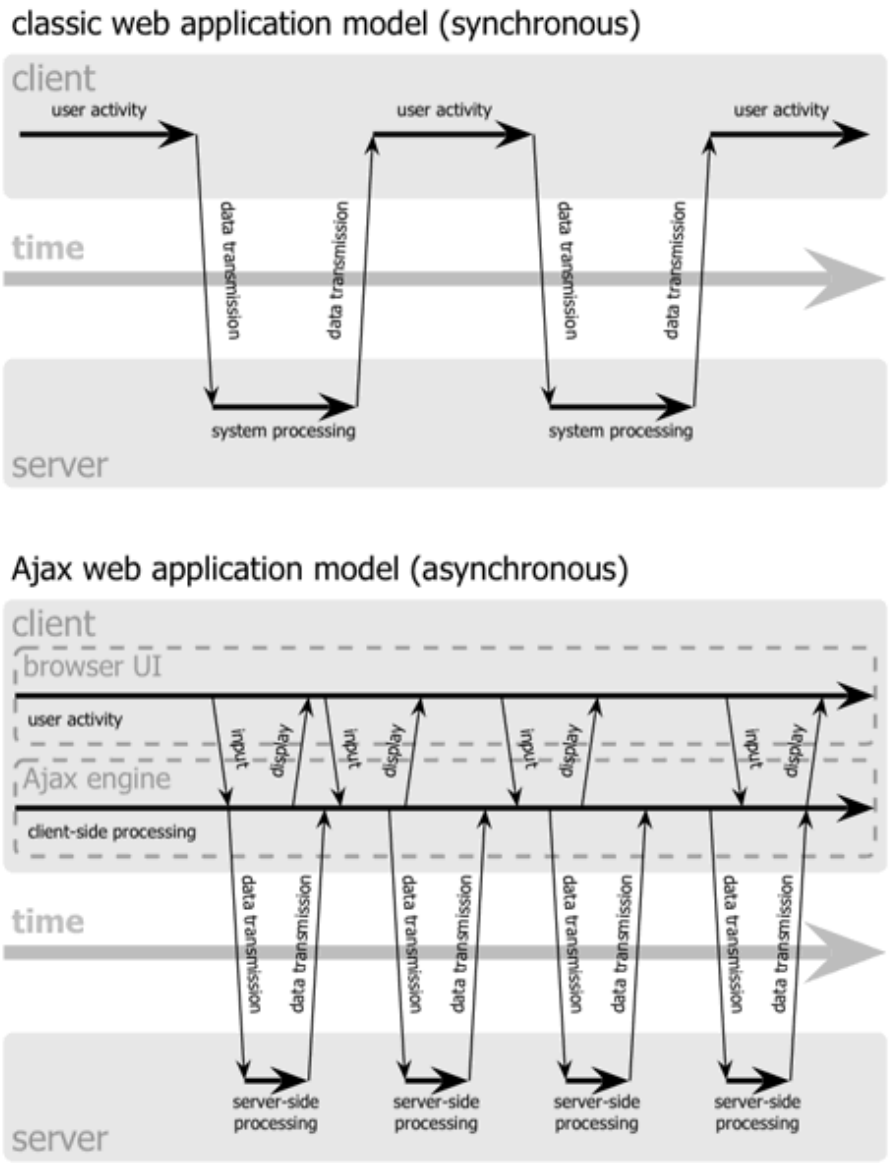
AJAX es el acrónimo de ***Asynchronous Javascript And XML*** (Javascript asíncrono y XML) y es lo que usamos para hacer peticiones asíncronas al servidor desde Javascript. Cuando hacemos una petición al servidor no nos responde inmediatamente (la petición tiene que llegar al servidor, procesarse allí y enviarse la respuesta que llegará al cliente).

Lo que significa **asíncrono** es que la página no permanecerá bloqueada esperando esa respuesta sino que continuará ejecutando su código e interactuando con el usuario, y en el momento en que llegue la respuesta del servidor se ejecutará la función que indicamos al hacer la llamada Ajax. Respecto a **XML**, es el formato en que se intercambia la información entre el servidor y el cliente, aunque actualmente el formato más usado es **JSON** que es más simple y legible.

Básicamente Ajax nos permite poder mostrar nuevos datos enviados por el servidor sin tener que recargar la página, que continuará disponible mientras se reciben y procesan los datos enviados por el servidor en segundo plano.



Sin Ajax cada vez que necesitamos nuevos datos del servidor la página deja de estar disponible para el usuario hasta que se recarga con lo que envía el servidor. Con Ajax la página está siempre disponible para el usuario y simplemente se modifica (cambiando el DOM) cuando llegan los datos del servidor:



Fuente Uniwebsidad

2. Métodos HTTP

Las peticiones Ajax usan el protocolo HTTP (el mismo que utiliza el navegador para cargar una página). Este protocolo envía al servidor unas cabeceras HTTP (con información como el *userAgent* del navegador, el idioma, etc), el tipo de petición y, opcionalmente, datos o parámetros (por ejemplo en la petición que procesa un formulario se envían los datos del mismo).

Hay diferentes tipos de petición que podemos hacer:

- **GET**: suele usarse para obtener datos sin modificar nada (equivale a un SELECT en SQL). Si enviamos datos (ej. la ID del registro a obtener) suelen ir en la url de la petición (formato URIEncoded). Ej.: locahost/users/3, https://jsonplaceholder.typicode.com/users o www.google.es?search=js
- **POST**: suele usarse para añadir un dato en el servidor (equivalente a un INSERT). Los datos

enviados van en el cuerpo de la petición HTTP (igual que sucede al enviar desde el navegador un formulario por POST)
- **PUT**: es similar al *POST* pero suele usarse para actualizar datos del servidor (como un UPDATE de SQL). Los datos se envían en el cuerpo de la petición (como en el POST) y la información para identificar el objeto a modificar en la url (como en el GET). El servidor hará un UPDATE sustituyendo el objeto actual por el que se le pasa como parámetro
- **PATCH**: es similar al PUT pero la diferencia es que en el PUT hay que pasar todos los campos del objeto a modificar (los campos no pasados se eliminan del objeto) mientras que en el PATCH sólo se pasan los campos que se quieren cambiar y en resto permanecen como están
- **DELETE**: se usa para eliminar un dato del servidor (como un DELETE de SQL). La información para identificar el objeto a eliminar se envía en la url (como en el GET)
- existen otros tipos que no veremos aquí (como *HEAD*, *PATCH*, etc)

El servidor acepta la petición, la procesa y le envía una respuesta al cliente con el recurso solicitado y además unas cabeceras de respuesta (con el tipo de contenido enviado, el idioma, etc) y el código de estado. Los códigos de estado más comunes son:

- 2xx: son peticiones procesadas correctamente. Las más usuales son 200 (*ok*) o 201 (*created*, como respuesta a una petición POST satisfactoria)
- 3xx: son códigos de redirección que indican que la petición se redirecciona a otro recurso del servidor, como 301 (el recurso se ha movido permanentemente a otra URL) o 304 (el recurso no ha cambiado desde la última petición por lo que se puede recuperar desde la caché)
- 4xx: indican un error por parte del cliente, como 404 (*Not found*, no existe el recurso solicitado) o 401 (*Not authorized*, el cliente no está autorizado a acceder al recurso solicitado)
- 5xx: indican un error por parte del servidor, como 500 (error interno del servidor) o 504 (*timeout*).

En cuanto a la información enviada por el servidor al cliente normalmente serán datos en formato JSON o XML (cada vez menos usado) que el cliente procesará y mostrará en la página al usuario. También podría ser HTML, texto plano, ...

El formato JSON es una forma de convertir un objeto Javascript en una cadena de texto para poderla enviar, por ejemplo el objeto

```
let alumno = {
  id: 5,
  nombre: Marta,
  apellidos: Pérez Rodríguez
}
```

se transformaría en la cadena de texto

```
{ "id": 5, "nombre": "Marta", "apellidos": "Pérez Rodríguez" }
```

y el array

```
let alumnos = [
  {
    id: 5,
    nombre: "Marta",
    apellidos: "Pérez Rodríguez"
  },
  {
    id: 7,
    nombre: "Joan",
    apellidos: "Reig Peris"
  },
]
```

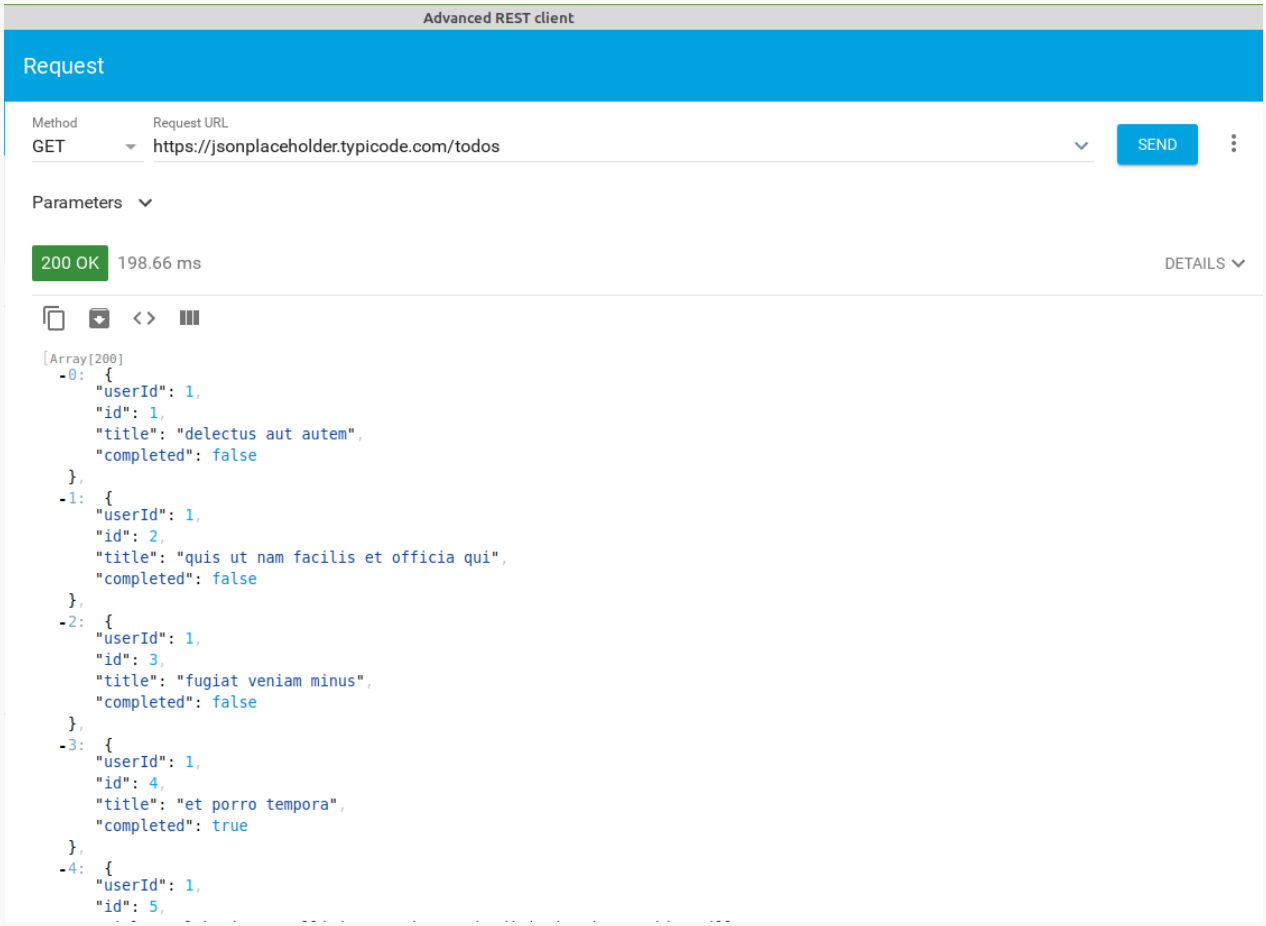
en la cadena:

```
[{ "id": 5, "nombre": Marta, "apellidos": Pérez Rodríguez }, { "id": 7, "nombre": "Joan", "apellidos": "Reig Peris" }]
```

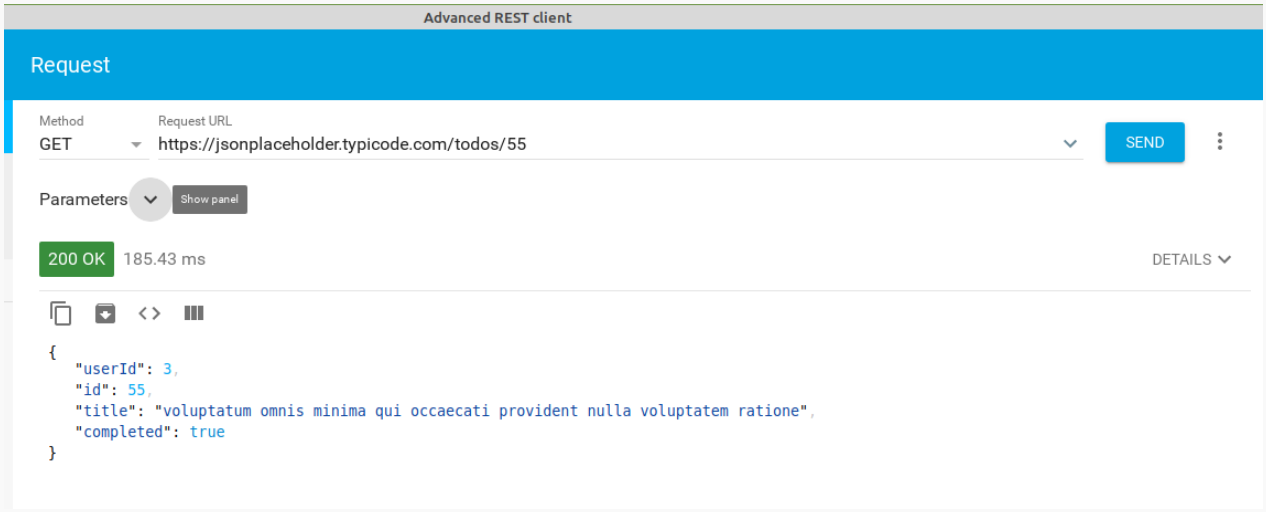
EJERCICIO 1_UD06: Vamos a realizar diferentes peticiones HTTP a la API https://jsonplaceholder.typicode.com, en concreto trabajaremos contra la tabla **todos** con tareas para hacer. Las peticiones GET podríamos hacerlas directamente desde el navegador pero para el resto debemos instalar alguna de las extensiones de cliente REST en nuestro navegador. Por tanto instalaremos dicha extensión (por ejemplo *Yet Another REST Client* para Chrome o *RestClient* para Firefox y haremos todas las peticiones desde allí (incluyendo los GET) lo que nos permitirá ver los códigos de estado devueltos, las cabeceras, etc.

Lo que queremos hacer en este ejercicio es:

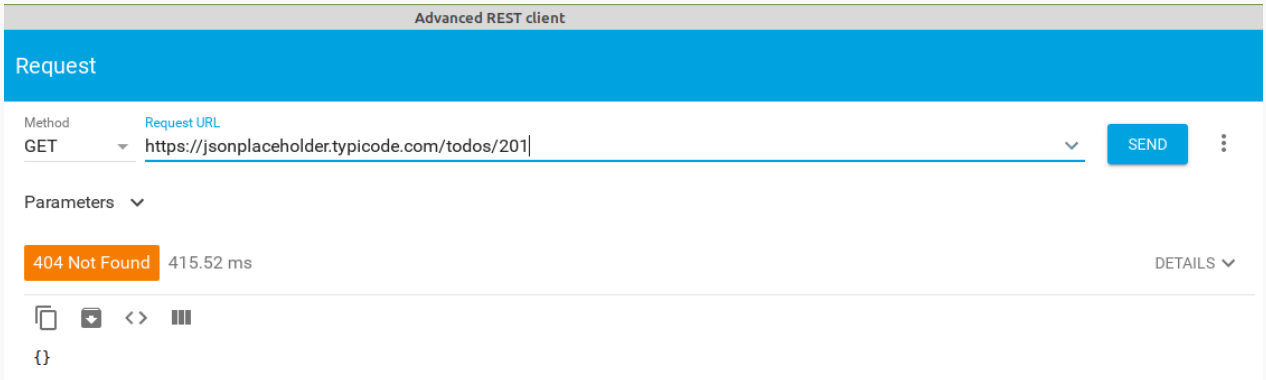
- obtener todas las tareas (devuelve un array con todas las tareas y el código devuelto será 200 - Ok)



- obtener la tarea con id 55 (devuelve el objeto de la tarea 55 y el código devuelto será 200 - Ok)

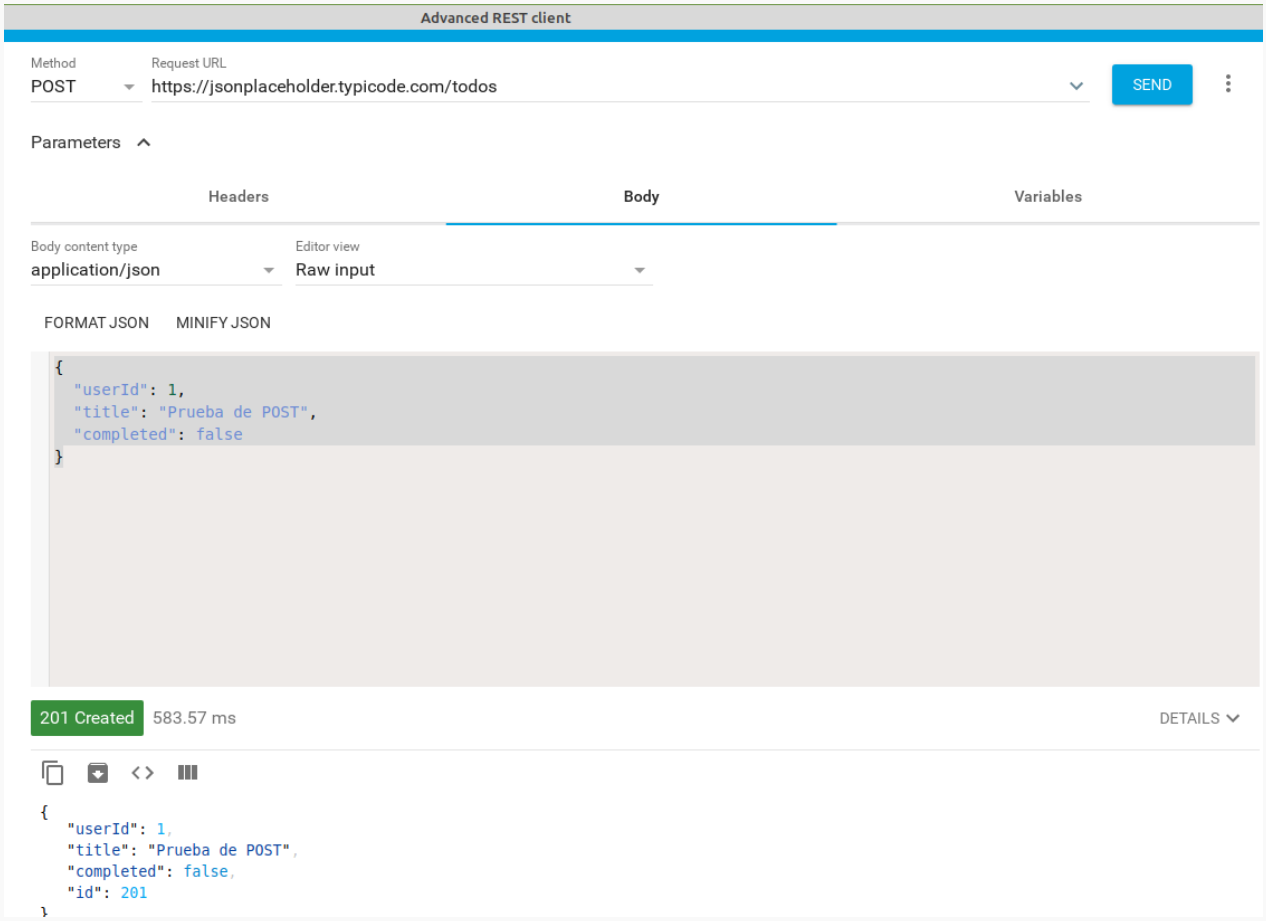


- obtener la tarea con id 201 (como no existe devolverá un objeto vacío y como código de error 404 - Not found)

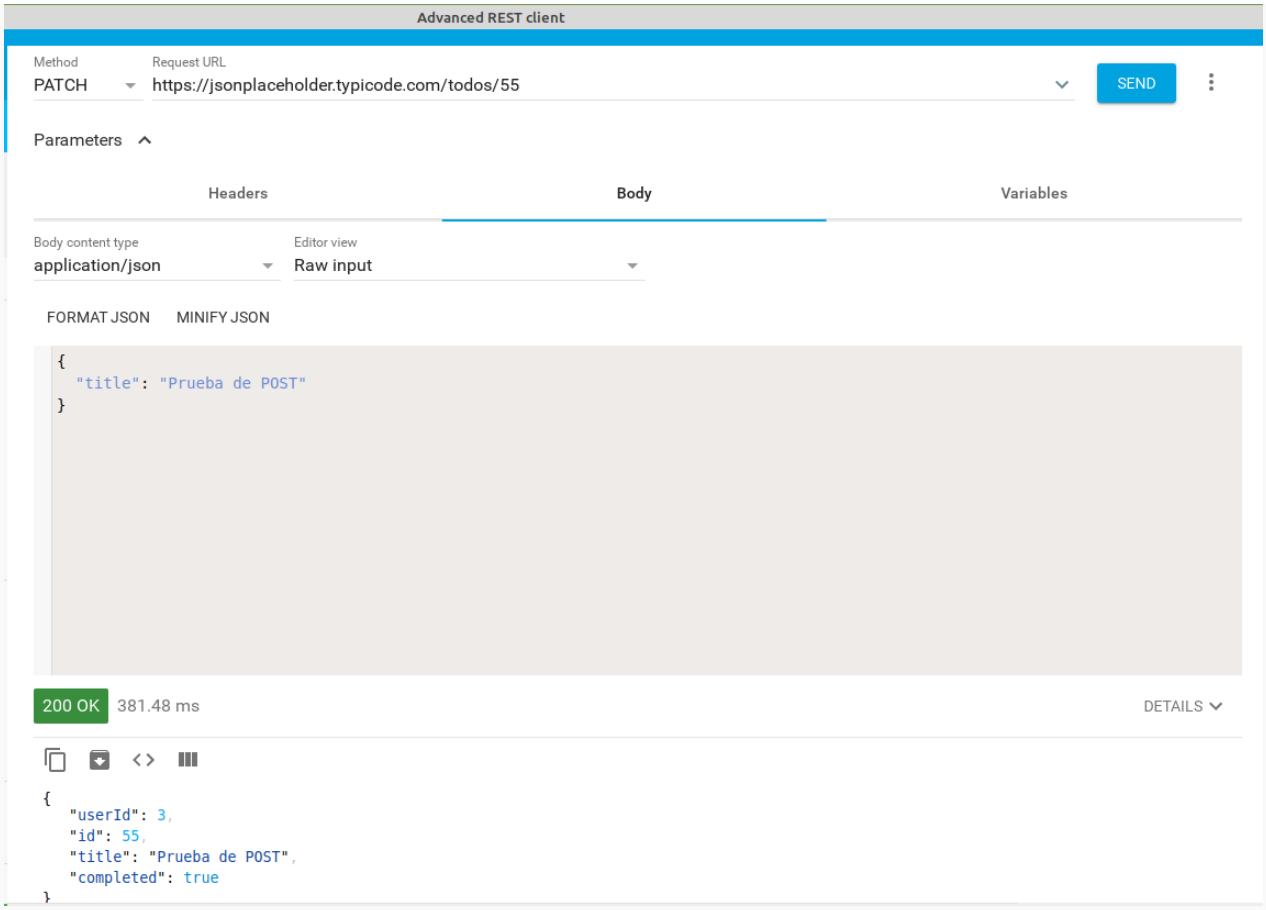


- crear una nueva tarea. En el cuerpo de la petición le pasaremos sus datos: userID: 1, title:

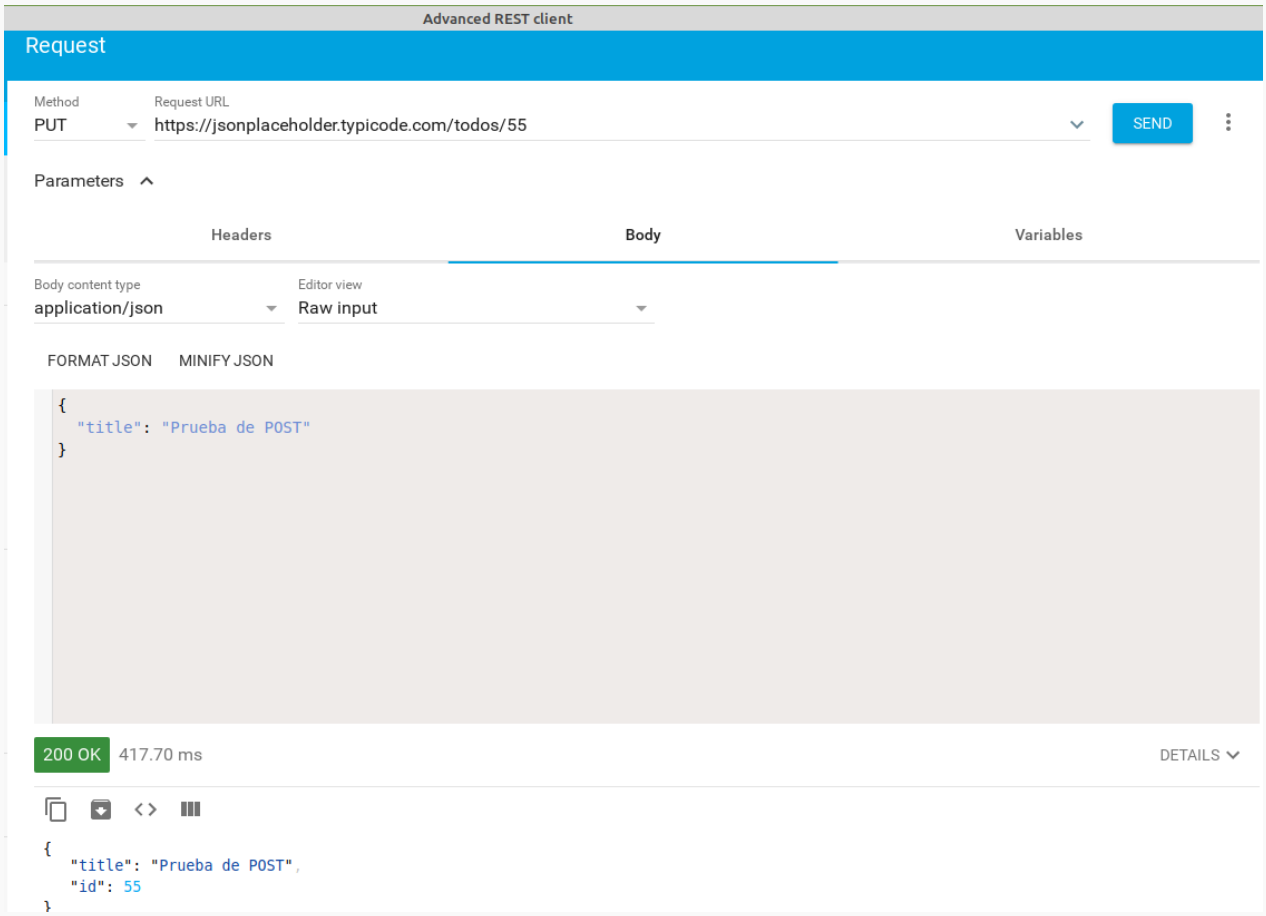
Prueba de POST y completed: false. No se le pasa la id (de eso se encarga la BBDD). La respuesta debe ser un código 201 (created) y el nuevo registro creado con todos sus datos incluyendo la id. Como es una API de prueba en realidad no lo está añadiendo a la BBDD por lo que si luego hacemos una petición buscando esa id nos dirá que no existe.



- modificar con un PATCH la tarea con id 55 para que su title sea 'Prueba de POST'. Devolverá el nuevo registro con un código 200. Como veis al hacer un PATCH los campos que no se pasan se mantienen como estaban

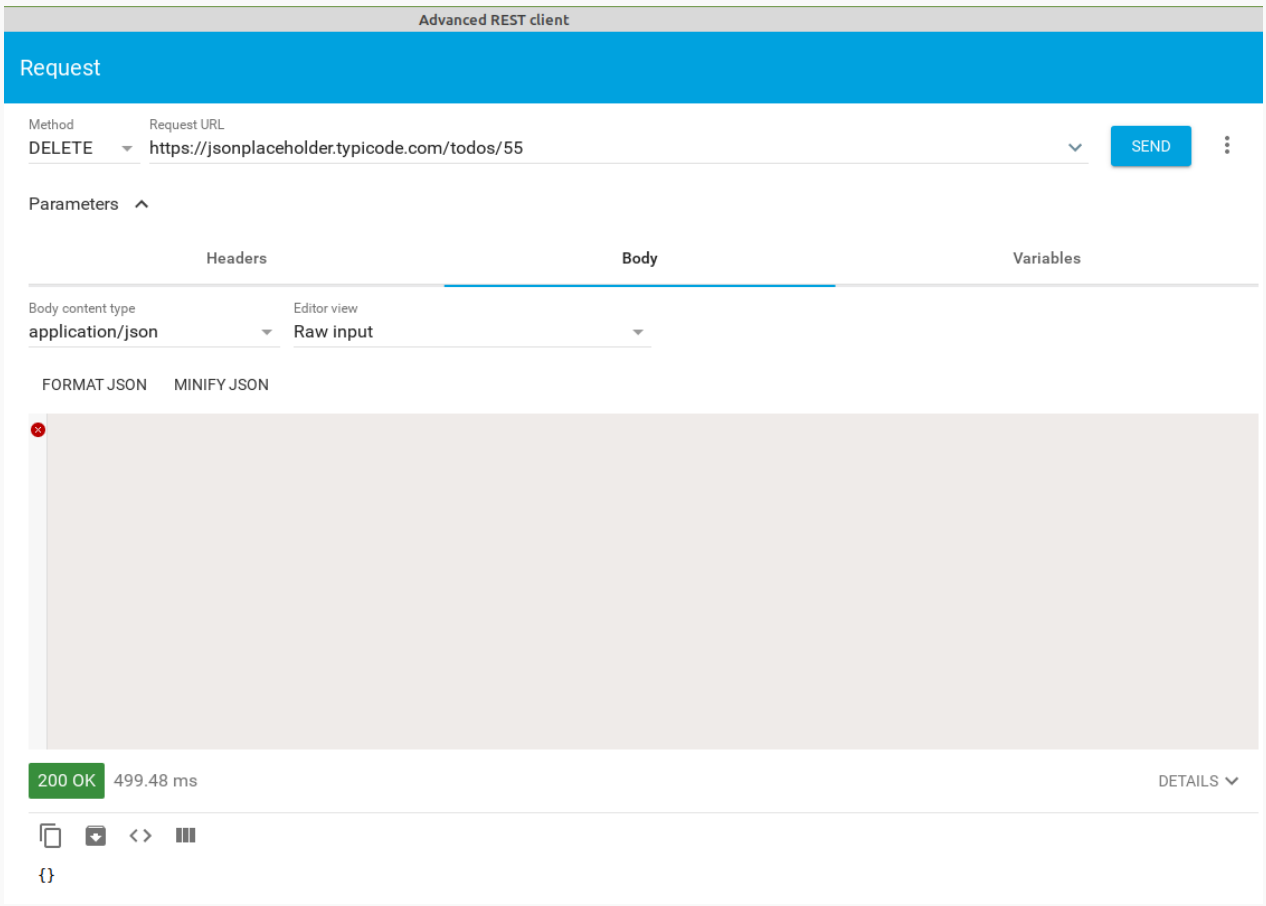


- modificar con un PUT la tarea con id 55 para que su title sea 'Prueba de POST'. Devolverá el nuevo registro con un código 200. Como veis en esta API los campos que no se pasan se eliminan; en otras los campos no pasados se mantienen como estaban



- eliminar la tarea con id 55. Como veis esta API devuelve un objeto vacío al eliminar; otras

devuelven el objeto eliminado



2.1 Json Server

Las peticiones Ajax se hacen a un servidor que proporcione una API. Como ahora no tenemos ninguno podemos utilizar Json Server que es un servidor API-REST que funciona bajo Node.js (que ya tenemos instalado para usar NPM) y que utiliza un fichero JSON como contenedor de los datos en lugar de una base de datos.

Para instalarlo en nuestra máquina (lo instalaremos global para poderlo usar en todas nuestras prácticas) ejecutamos:

```
npm install -g json-server
```

Para que sirva un fichero datos.json:

```
json-server datos.json
```

Le podemos poner la opción *--watch* para que actualice los datos si se modifica el fichero *.json* externamente (si lo editamos).

El fichero *datos.json* será un fichero que contenga un objeto JSON con una propiedad para cada "*tabla*" de nuestra BBDD. Por ejemplo, si queremos simular una BBDD con las tablas *users* y *posts* vacías el contenido del fichero será:

```
{
  "users": [],
  "posts": []
}
```

La API escucha en el puerto 3000 y servirá los diferentes objetos definidos en el fichero *.json*. Por ejemplo:

- http://localhost:3000/users: devuelve un array con todos los elementos de la tabla *users* del fichero *.json*
- http://localhost:3000/users/5: devuelve un objeto con el elemento de la tabla *users* cuya propiedad *id* valga 5

También pueden hacerse peticiones más complejas como:

- http://localhost:3000/users?rol=3: devuelve un array con todos los elementos de *users* cuya propiedad *rol* valga 3

Para más información: https://github.com/typicode/json-server.

Si queremos acceder a la API desde otro equipo (no desde *localhost*) tenemos que indicar la IP de la máquina que ejecuta *json-server* y que se usará para acceder, por ejemplo si vamos a ejecutarlo en la máquina 192.168.0.10 pondremos:

```
json-server --host 192.168.0.10 datos.json
```

Y la ruta para acceder a la API será `http://192.168.0.10:3000`.

> **EJERCICIO 2_UD06**: instalar json-server en tu máquina. Ejecútalo indicando un nombre de fichero que no existe: como verás crea un fichero json de prueba con 3 tablas: *posts*, *comments* y *profiles*. Ábrelo en tu navegador para ver los datos

2.2 REST client

Para probar las peticiones GET podemos poner la URL en la barra de direcciones del navegador pero para probar el resto de peticiones debemos instalar en nuestro navegador una extensión que nos permita realizar las peticiones indicando el método a usar, las cabeceras a enviar y los datos que enviaremos a servidor, además de la URL.

Como ya hemos visto en el ejercicio 1, existen multitud de aplicaciones para realizar peticiones HTTP, como Yet Another REST client.

3. Realizar peticiones Ajax

Hasta ahora hemos hecho un repaso a lo que es el protocolo HTTP. Ahora que lo tenemos claro y hemos instalado un servidor que nos proporciona una API (json-server) vamos a realizar peticiones HTTP en nuestro código javascript usando Ajax.

Para hacer una petición debemos crear una instancia del objeto **XMLHttpRequest** que es el que controlará todo el proceso. Los pasos a seguir son:

1. Creamos la instancia del objeto: `const peticion=new XMLHttpRequest()`
2. Para establecer la comunicación con el servidor ejecutamos el método **.open()** al que se le pasa como parámetro el tipo de petición (GET, POST, ...) y la URL del servidor: `peticion.open('GET', 'https://jsonplaceholder.typicode.com/users')`
3. OPCIONAL: Si queremos añadir cabeceras a la petición HTTP llamaremos al método **.setRequestHeader()**. Por ejemplo si enviamos datos con POST hay que añadir la cabecera *Content-type* que le indica al servidor en qué formato van los datos: `peticion.setRequestHeader('Content-type', 'application/x-www-form-urlencoded)`
4. Enviamos la petición al servidor con el método **.send()**. A este método se le pasa como parámetro los datos a enviar al servidor. Si es una petición GET o DELETE no le pasaremos datos (`peticion.send()`) y si es un POST, PUT o PATCH le pasaremos una cadena de texto con los datos a enviar: `peticion.send('dato1='+encodeURIComponent(dato1)+'&dato2='+encodeURIComponent(dato2))`
5. Escuchamos los eventos que se producen nuestro objeto *peticion* para saber cuándo está disponible la respuesta del servidor

3.1 Eventos de XMLHttpRequest

Tenemos diferentes eventos que el servidor envía para informarnos del estado de nuestra petición y que nosotros podemos capturar. El evento **readystatechange** se produce cada vez que el servidor cambia el estado de la petición. Cuando hay un cambio en el estado cambia el valor de la propiedad **readyState** de la petición. Sus valores posibles son:

- 0: petición no iniciada (se ha creado el objeto XMLHttpRequest)
- 1: establecida conexión con el servidor (se ha hecho el *open*)
- 2: petición recibida por el servidor (se ha hecho el *send*)
- 3: se está procesando la petición
- 4: petición finalizada y respuesta lista (este es el evento que nos interesa porque ahora tenemos la respuesta disponible)
  A nosotros sólo nos interesa cuando su valor sea 4 que significa que ya están los datos. En ese momento la propiedad **status** contiene el estado de la petición HTTP (200: *Ok*, 404: *Not found*, 500: *Server error*, ...) que ha devuelto el servidor. Cuando *readyState* vale 4 y *status* vale 200 tenemos los

datos en la propiedad **responseText** (o **responseXML** si el servidor los envía en formato XML). Ejemplo:

```
let peticion = new XMLHttpRequest();
console.log("Estado inicial de la petición: " + peticion.readyState);
peticion.open('GET', 'https://jsonplaceholder.typicode.com/users');
console.log("Estado de la petición tras el 'open': " +
peticion.readyState);
peticion.send();
console.log("Petición hecha");
peticion.addEventListener('readystatechange', function() {
    console.log("Estado de la petición: " + peticion.readyState);
    if (peticion.readyState === 4) {
        if (peticion.status === 200) {
            console.log("Datos recibidos:");
            let usuarios = JSON.parse(peticion.responseText);  //
Convertirmos los datos JSON a un objeto
            console.log(usuarios);
        } else {
            console.log("Error " + peticion.status + " (" +
peticion.statusText + ") en la petición");
        }
    }
})
console.log("Petición acabada");
```

El resultado de ejecutar ese código es el siguiente:

```
Estado inicial de la petición: 0                                    main.js:2:1
Estado de la petición tras el 'open': 1                             main.js:4:1
Petición hecha                                                      main.js:6:1
Petición acabada                                                    main.js:19:1
Estado de la petición: 2                                            main.js:8:5
Estado de la petición: 3                                            main.js:8:5
Estado de la petición: 4                                            main.js:8:5
Datos recibidos:                                                    main.js:11:13
▶ Array(10) [ {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…} ]   main.js:13:13
```

Fijaos cuándo cambia de estado (*readyState*) la petición:

- vale 0 al crear el objeto XMLHttpRequest
- vale 1 cuando abrimos la conexión con el servidor
- luego se envía al servidor y es éste el que va informando al cliente de cuándo cambia su estado

MUY IMPORTANTE: notad que la última línea ('Petición acabada') se ejecuta antes que las de 'Estado de la petición'. Recordad que es una **petición asíncrona** y la ejecución del programa continúa sin esperar a que responda el servidor.

Como normalmente no nos interesa cada cambio en el estado de la petición sino que sólo queremos saber cuándo ha terminado de procesarse tenemos otros eventos que nos pueden ser de utilidad:

- **load**: se produce cuando se recibe la respuesta del servidor. Equivale a *readyState===4*. En *status* tendremos el estado de la respuesta
- **error**: se produce si sucede algún error al procesar la petición (de red, de servidor, ...)
- **timeout**: si ha transcurrido el tiempo indicado y no se ha recibido respuesta del servidor. Podemos cambiar el tiempo por defecto modificando la propiedad *timeout* antes de enviar la petición
- **abort**: si se cancela la petición (se hace llamando al método **.abort()** de la petición)
- **loadend**: se produce siempre que termina la petición, independientemente de si se recibe respuesta o sucede algún error (incluyendo un *timeout* o un *abort*)

Ejemplo de código que sí usaremos:

```
const peticion=new XMLHttpRequest();
peticion.open('GET', 'https://jsonplaceholder.typicode.com/users');
peticion.send();
peticion.addEventListener('load', function() {
    if (peticion.status===200) {
        let usuarios=JSON.parse(peticion.responseText);
        // procesamos los datos que tenemos en usuarios
    } else {
        muestraError();
    }
})
peticion.addEventListener('error', muestraError);
peticion.addEventListener('abort', muestraError);
peticion.addEventListener('timeout', muestraError);

function muestraError() {
    if (this.status) {
        console.log("Error "+this.status+" ("+this.statusText+") en la petición");
    } else {
        console.log("Ocurrió un error o se abortó la conexión");
    }
}
```

Recuerda que tratamos con peticiones asíncronas: tras la línea

```
peticion.addEventListener('load', function() {
```

no se ejecuta la línea siguiente

```
    if (peticion.status===200) {
```

sino la de

```
peticion.addEventListener('error', muestraError);
```

Una petición asíncrona es como pedir una pizza: tras llamar por teléfono lo siguiente no es ir a la puerta a recogerla sino que seguimos haciendo cosas por casa y cuando suena el timbre de casa entonces vamos a la puerta a por ella.

3.2 Ejemplos de envío de datos

Vamos a ver algunos ejemplos de envío de datos al servidor con POST. Supondremos que tenemos una página con un formulario para dar de alta nuevos productos:

```
<form id="addProduct">
    <label for="name">Nombre: </label><input type="text" name="name" id="name" required><br>
    <label for="descrip">Descripción: </label><input type="text" name="descrip" id="descrip" required><br>

    <button type="submit">Añadir</button>
</form>
```

3.3 Enviar datos al servidor en formato JSON

```
document.getElementById('addProduct').addEEventListener('submit', (event) => {
  ...
  const newProduct={
      name: document.getElementById("name").value,
      descrip: document.getElementById("descrip").value,
  }
  const peticion=new XMLHttpRequest();
  peticion.open('POST', 'https://localhost/products');
  peticion.setRequestHeader('Content-type', 'application/json');  // Siempre tiene que estar esta línea si se envían datos
```

```
  peticion.send(JSON.stringify(newProduct));              // Hay que convertir el objeto a una cadena de texto JSON para enviarlo
  peticion.addEventListener('load', function() {
    // procesamos los datos
    ...
  })
})
```

Para enviar el objeto hay que convertirlo a una cadena JSON con la función **JSON.stringify()** (es la opuesta a **JSON.parse()**). Y siempre que enviamos datos al servidor debemos decirle el formato que tienen en la cabecera de *Content-type*:

```
peticion.setRequestHeader('Content-type', 'application/json');
```

3.4 Enviar datos al servidor en formato URIEncoded

```
document.getElementById('addProduct').addEEventListener('submit', (event) => {
  ...
  const name=document.getElementById("name").value;
  const descrip=document.getElementById("descrip").value;

  const peticion=new XMLHttpRequest();
  peticion.open('GET', 'https://localhost/products');
  peticion.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
  peticion.send('name='+encodeURIComponent(name)+'&descrip='+encodeURIComponent(descrip));
  peticion.addEventListener('load', function() {
    ...
  })
})
```

En este caso los datos se envían como hace el navegador por defecto en un formulario. Recordad siempre codificar lo que introduce el usuario para evitar problemas con caracteres no estándar y **ataques *SQL Injection***.

3.5 Enviar ficheros al servidor con FormData

FormData es una nueva interfaz de XMLHttpRequest que permite construir fácilmente pares de `clave=valor` para enviar los datos de un formulario. Se envían en el mismo formato en que se enviarían directamente desde un formulario ("multipart/form-data") por lo que no hay que poner encabezado de 'Content-type'.

Vamos a añadir al formulario un campo donde el usuario pueda subir la foto del producto:

```
<form id="addProduct">
    <label for="name">Nombre: </label><input type="text" name="name" id="name" required><br>
    <label for="descrip">Descripción: </label><input type="text" name="descrip" id="descrip" required><br>
    <label for="photo">Fotografía: </label><input type="file" name="photo" id="photo" required><br>

    <button type="submit">Añadir</button>
</form>
```

Podemos enviar al servidor todo el contenido del formulario:

```
document.getElementById('addProduct').addEventListener('submit', (event) => {
  ...
  const peticion=new XMLHttpRequest();
  const datosForm = new FormData(document.getElementById('addProduct'));
  // Automáticamente ha añadido todos los inputs, incluyendo tipo 'file', blob, ...
  // Si quisiéramos añadir algún dato más haríamos:
  formData.append('otrodato', 12345);
  // Y lo enviamos
  peticion.open('POST', 'https://localhost/products');
  peticion.send(datosForm);
  peticion.addEventListener('load', function() {
    ...
  })
})
```

También podemos enviar sólo los campos que queramos:

```
document.getElementById('addProduct').addEEventListener('submit', (event) => {
  ...
```

```
  const formData=new FormData();  // creamos un formData vacío
  formData.append('name', document.getElementById('name').value);
  formData.append('descrip', document.getElementById('descrip').value);
  formData.append('photo', document.getElementById('photo').files[0]);

  const peticion=new XMLHttpRequest();
  peticion.open('POST', 'https://localhost/products');
  peticion.send(formData);
  peticion.addEventListener('load', function() {
    ...
  })
})
```

Podéis ver más información de cómo usar formData en MDN web docs.

4. Ejemplo de petición Ajax

Vamos a ver un ejemplo de una llamada a Ajax. Vamos a hacer una página que muestre en una tabla los posts del usuario indicado en un input. La parte del body de la página sería así:

```
  <h1>Mostrar posts de un usuario</h1>
  <form action="#" id="form-show">
    <label for="id-usuario">Introduce el id del usuario: </label>
    <input type="text" id="id-usuario" size="2">
    <input type="submit" value="Mostrar posts">
  </form>
  <table>
    <thead>
      <tr>
        <th>userId</th>
        <th>Id</th>
        <th>Título</th>
        <th>Post</th>
      </tr>
    </thead>
    <tbody></tbody>
  </table>
  <p>Total de posts mostrados: <span id="num-posts"></span></p>
```

En resumen lo que hacemos es:

1. El usuario de nuestra aplicación introduce el código del usuario del que queremos ver sus posts

2. Tenemos un escuchador para que al introducir un código de un usuario llamamos a una función *getPosts()* que:

- Se encarga de hacer la petición Ajax al servidor
- Si se produce un error se encarga de informar al usuario de nuestra aplicación

1. Cuando se reciben deben pintarse en la tabla

Si no estuviéramos haciendo una petición asíncrona el código de todo esto será algo como el siguiente (ATENCIÓN, este código **NO FUNCIONA**):

```
const SERVER = 'https://jsonplaceholder.typicode.com';
const tbody = document.getElementsByTagName('tbody')[0];

window.addEventListener('load', function() {
  document.getElementById('form-show').addEventListener('submit', (event)
=> {
    event.preventDefault();
    let idUser = document.getElementById('id-usuario').value;
    if (isNaN(idUser) || idUser == '') {
      alert('Debes introducir un número');
    } else {
      const posts = getPosts(idUser);
      tbody.innerHTML = ''; // borramos el contenido de la tabla
      datos.forEach(post => {
        const newPost = document.createElement('tr');
        newPost.innerHTML = `
            <td>${post.userId}</td>
            <td>${post.id}</td>
            <td>${post.title}</td>
            <td>${post.body}</td>`;
        tbody.appendChild(newPost);
      })
      document.getElementById('num-posts').textContent = datos.length;
    }
  })
})

function getPosts(idUser) {
  const peticion = new XMLHttpRequest();
  peticion.open('GET', SERVER + '/posts?userId=' + idUser);
  peticion.send();
  peticion.addEventListener('load', function() {
    if (peticion.status === 200) {
```

```
      const datos = JSON.parse(peticion.responseText); // Convertirmos los datos JSON a un objeto
      return datos
    } else {
      console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
    }
  })
  peticion.addEventListener('error', () => console.error('Error en la petición HTTP'));
}
```

Pero esto no funciona porque el valor de `posts` siempre es *undefined*. Esto es porque cuando se llama a `getPosts` esta función no devuelve nada (por eso *posts* es undefined) sino que devuelve tiempo después, cuando el servidor conteste, pero entonces ya no hay nadie escuchando.

La solución es que todo el código, no sólo de la petición Ajax sino también el de qué hacer con los datos cuando llegan, se encuentre en la función que pide los datos al servidor:

```
const SERVER = 'https://jsonplaceholder.typicode.com';
const tbody = document.getElementsByTagName('tbody')[0];

window.addEventListener('load', function() {
  document.getElementById('form-show').addEventListener('submit', (event) => {
    event.preventDefault();
    let idUser = document.getElementById('id-usuario').value;
    if (isNaN(idUser) || idUser == '') {
      alert('Debes introducir un número');
    } else {
      getPosts(idUser);
    }
  })
})

function getPosts(idUser) {
  const peticion = new XMLHttpRequest();
  peticion.open('GET', SERVER + '/posts?userId=' + idUser);
  peticion.send();
  peticion.addEventListener('load', function() {
    if (peticion.status === 200) {
      const datos = JSON.parse(peticion.responseText); // Convertirmos los datos JSON a un objeto
```

```
      tbody.innerHTML = ''; // borramos el contenido de la tabla
      datos.forEach(post => {
        const newPost = document.createElement('tr');
        newPost.innerHTML = `
            <td>${post.userId}</td>
            <td>${post.id}</td>
            <td>${post.title}</td>
            <td>${post.body}</td>`;
        tbody.appendChild(newPost);
      })
      document.getElementById('num-posts').textContent = datos.length;
    } else {
      console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
    }
  })
  peticion.addEventListener('error', () => console.error('Error en la petición HTTP'));
}
```

Esta forma de programar tiene una pega: tenemos que tratar los datos (en este caso pintarlos en la tabla) en la función que gestiona la petición porque es la que sabe cuándo están disponibles esos datos. Por tanto nuestro código queda poco claro.

4.1 Funciones *callback*

Esto se podría mejorar usando una función ***callback***. La idea es que creamos una función que procese los datos (`renderPosts`) y se la pasamos a `getPosts` para que la llame cuando tenga los datos:

```
const SERVER = 'https://jsonplaceholder.typicode.com'
const tbody = document.querySelector('tbody')

window.addEventListener('load', function() {
  document.getElementById('form-show').addEventListener('submit', (event) => {
    event.preventDefault();
    let idUser = document.getElementById('id-usuario').value
    if (isNaN(idUser) || idUser.trim() == '') {
      alert('Debes introducir un número')
    } else {
      getPosts(idUser, renderPosts)
    }
```

```
  })
})

function renderPosts(datos) {
  tbody.innerHTML = '' // borramos el contenido de la tabla
  datos.forEach(post => {
    const newPost = document.createElement('tr')
    newPost.innerHTML = `
        <td>${post.userId}</td>
        <td>${post.id}</td>
        <td>${post.title}</td>
        <td>${post.body}</td>`
    tbody.appendChild(newPost)
  })
  document.getElementById('num-posts').textContent = datos.length;
}

function getPosts(idUser, callback) {
  const peticion = new XMLHttpRequest()
  peticion.open('GET', SERVER + '/posts?userId=' + idUser)
  peticion.send()
  peticion.addEventListener('load', function() {
    if (peticion.status === 200) {
      callback(JSON.parse(peticion.responseText));
    } else {
      console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición")
    }
  })
  peticion.addEventListener('error', () => console.error('Error en la petición HTTP'))
}
```

Hemos creado una función que se ocupa de renderizar los datos y se la pasamos a la función que gestiona la petición para que la llame cuando los datos están disponibles. Utilizando la función *callback* hemos conseguido que *getPosts()* se encargue sólo de obtener los datos y cuando los tenga los pasa a la encargada de pintarlos en la tabla.

4.2 Promesas

Sin embargo hay una forma más limpia de resolver una función asíncrona y que el código se parezca al primero que hicimos y que no funcionaba, el que hacía:

```
    ...
    let idUser = document.getElementById('id-usuario').value;
    if (isNaN(idUser) || idUser == '') {
      alert('Debes introducir un número');
    } else {
      const posts = getPosts(idUser);
      // y aquí usamos los datos recibidos, en este caso para pintar los
posts
    }
    ...
```

Esto podemos conseguirlo mediante el uso de *promesas*. Una llamada a una promesa cuenta con 2 métodos:

- **.then(*function(datos) { ... }*)**: al resolverse la promesa satisfactoriamente se ejecuta la función pasada como parámetro del *then*. Esta recibe como parámetro los datos que se envían al resolver la promesa (normalmente los datos devueltos por la función asíncrona a la que se ha llamado)
- **.catch(*function(datos) { ... }*)**: la función pasada como parámetro se ejecuta si se rechaza la promesa (normalmente porque se ha recibido una respuesta errónea del servidor). Esta función recibe como parámetro la información pasada por la promesa al ser rechazada (normalmente información sobre el error producido).

De esta manera nuestro código quedaría:

```
    ...
    let idUser = document.getElementById('id-usuario').value;
    if (isNaN(idUser) || idUser == '') {
      alert('Debes introducir un número');
    } else {
      getPosts(idUser)
        .then((posts) => {
          tbody.innerHTML = ''; // borramos el contenido de la tabla
          posts.forEach((post) => {
            const newPost = document.createElement('tr');
            newPost.innerHTML = `
                <td>${post.userId}</td>
                <td>${post.id}</td>
                <td>${post.title}</td>
                <td>${post.body}</td>`;
```

```
            tbody.appendChild(newPost);
          })
          document.getElementById('num-posts').textContent =
posts.length;
        })
        .catch((error) => console.error(error))
    }
```

La llamada a la función asíncrona se hace desde dentro de una función que devuelve una promesa (`getPosts`). Para que una función devuelva una promesa haremos que devuelva un objeto de tipo *Promise* (`return new Promise()`) cuyo parámetro es una función que recibe 2 parámetros:

- *resolve*: función *callback* a la que se llamará cuando se resuelva la promesa satisfactoriamente
- *reject*: función *callback* a la que se llamará cuando se resuelva la promesa con errores

Cuando la promesa se resuelva satisfactoriamente *getPosts* llama a la función `resolve()` y le pasa los datos recibidos por el servidor (que los recibirá quien llamó a la promesa en su *.then*). Si se produce algún error se rechaza la promesa llamando a la función `reject()` y pasando como parámetro la información de que ha fallado la llamada y por qué (esto le llegará a quien la llamó en su *.catch*). Por tanto nuestra función *getPosts* ahora quedará así:

```
function getPosts(idUser) {
  return new Promise((resolve, reject) => {
    const peticion = new XMLHttpRequest();
    peticion.open('GET', SERVER + '/posts?userId=' + idUser);
    peticion.send();
    peticion.addEventListener('load', () => {
      if (peticion.status === 200) {
        resolve(JSON.parse(peticion.responseText));
      } else {
        reject("Error " + this.status + " (" + this.statusText + ") en la
petición");
      }
    })
    peticion.addEventListener('error', () => reject('Error en la petición
HTTP'));
  })
}
```

Fijaos que el único cambio es la primera línea donde convertimos nuestra función en una promesa, y que luego para "devolver" los datos a quien llama a *getPosts* en lugar de hacer un *return*, que ya hemos visto que no funciona, se hace un *resolve* si todo ha ido bien o un *reject* si ha fallado.

Desde donde llamamos a la promesa nos suscribimos a ella usando los métodos ***.then()*** y * ***.catch()*** que hemos visto anteriormente.

Básicamente lo que nos van a proporcionar las promesas es un código más claro y mantenible ya que el código a ejecutar cuando se obtengan los datos asíncronamente estará donde se piden esos datos y no en una función escuchadora o en una función *callback*.

Utilizando promesas vamos a conseguir que la función que pide los datos sea quien los obtiene y los trate o quien informa si hay un error.

El código del ejemplo de los posts usando promesas sería el siguiente:

```
const SERVER = 'https://jsonplaceholder.typicode.com'
const tbody = document.querySelector('tbody')

window.addEventListener('load', function() {
  document.getElementById('form-show').addEventListener('submit', (event) => {
    event.preventDefault();
    let idUser = document.getElementById('id-usuario').value
    if (isNaN(idUser) || idUser.trim() == '') {
      alert('Debes introducir un número')
    } else {
      getPosts(idUser)
        .then(function(posts) {
          tbody.innerHTML = '' // borramos el contenido de la tabla
          posts.forEach(post => {
            const newPost = document.createElement('tr')
            newPost.innerHTML = `
                <td>${post.userId}</td>
                <td>${post.id}</td>
                <td>${post.title}</td>
                <td>${post.body}</td>`
            tbody.appendChild(newPost)
          })
          document.getElementById('num-posts').textContent = posts.length
        })
        .catch(function(error) {
          console.error(error)
        })
    }
  })
})

```

```
function getPosts(idUser) {
  return new Promise(function(resolve, reject) {
    let peticion = new XMLHttpRequest()
    peticion.open('GET', SERVER + '/posts?userId=' + idUser)
    peticion.send()
    peticion.addEventListener('load', () => {
      if (peticion.status === 200) {
        resolve(JSON.parse(peticion.responseText))
      } else {
        reject("Error " + this.status + " (" + this.statusText + ") en la petición")
      }
    })
    peticion.addEventListener('error', () => reject('Error en la petición HTTP'))
  })
}
```

Podéis consultar aprender más en MDN web docs.

5. *fetch*

La API Fetch proporciona una interfaz para realizar peticiones Ajax mediante el protocolo HTTP, que devuelve en forma de **promesas**. Básicamente lo que hace es encapsular en una función todo el código que se repite siempre en una petición AJAX (crear la petición, hacer el *open*, el *send*, escuchar los eventos, ...). La función *fetch* se similar a la función *getPosts* que hemos creado antes pero genérica para que sirva para cualquier petición pasándole la URL. Si la respuesta está en formato JSON hay que llamar a un método de la respuesta (*.json*) para que haga el `JSON.parse` que transforme la cadena de respuesta en un objeto o un array. Este método devuelve una nueva promesa a la que suscribirnos con un nuevo *.then*. Ejemplo.:

```
fetch('https://jsonplaceholder.typicode.com/posts?userId=' + idUser)
  .then(response => response.json())    // tenemos los datos en formato JSON, los transformamos en un objeto
  .then(myData => {       // ya tenemos los datos en _myData_ como un objeto o array  que podemos procesar
    // Aquí procesamos los datos (en nuestro ejemplo los pintaríamos en la tabla)
    console.log(myData)
  })
  .catch(err => console.error(err));
```

El código anterior hace una petición al servidor 'https://jsonplaceholder.typicode.com/posts' pidiéndole los posts de un determinado usuario. Cuando se resuelve la promesa se obtiene el resultado como cadena JSON en el objeto *response*. Dicho objeto tiene unas propiedades (*status*, *statusText*, *ok*, ...) y unos métodos como *json()* que devuelve una nueva promesa que cuando se resuelve contiene los datos de la respuesta pasada. Usando *fetch* nos ahorramos toda la parte de crear la petición y gestionarla.

5.1 Propiedades y métodos de la respuesta

La respuesta devuelta por *fetch()* tiene las siguientes propiedades y métodos:

- **status**, **statusText**: el código y el texto del estado devuelto por el servidor (200 / Ok, 404 / Not found, ...)
- **ok**: booleano que vale *true* si el status está entre 200 y 299 y *false* en caso contrario (para no tener que tener en cuenta si es 200 o 201 en un POST)
- **json()**: devuelve una promesa que se resolverá con los datos JSON de la respuesta convertidos a un objeto (les hace automáticamente un *JSON.parse()*)
- otros métodos de obtener los datos: **text()**, **blob()**, **formData()**, ... Todos devuelven una promesa con los datos de distintos formatos convertidos.

El ejemplo que hemos visto con las promesas, usando *fetch* quedaría:

```
const SERVER = 'https://jsonplaceholder.typicode.com'
const tbody = document.querySelector('tbody')

window.addEventListener('load', () => {
  document.getElementById('form-show').addEventListener('submit', (event) => {
    event.preventDefault();
    let idUser = document.getElementById('id-usuario').value
    if (isNaN(idUser) || idUser.trim() == '') {
      alert('Debes introducir un número')
    } else {
      fetch(SERVER + '/posts?userId=' + idUser)
        .then((response) => response.json())
        .then((posts) => {
          tbody.innerHTML = '' // borramos el contenido de la tabla
          posts.forEach((post) => {
            const newPost = document.createElement('tr')
            newPost.innerHTML = `
                  <td>${post.userId}</td>
                  <td>${post.id}</td>
                  <td>${post.title}</td>
```

```
                    <td>${post.body}</td>`
                tbody.appendChild(newPost)
            })
            document.getElementById('num-posts').textContent = posts.length
        })
        .catch((error) => console.error(error))
    }
  })
})
```

5.2 Cabeceras de la petición

Para peticiones que no sean GET la función *fetch()* admite como segundo parámetro un objeto con la información a enviar en la petición HTTP. Ej.:

```
fetch(url, {
  method: 'POST', // o 'PUT', 'GET', 'DELETE'
  body: JSON.stringify(data), // los datos que enviamos al servidor en el 'send'
  headers:{
    'Content-Type': 'application/json'
  }
}).then
```

Ejemplo de petición comprobando el estado:

```
fetch(url, {
  method: 'POST',
  body: JSON.stringify(data), // los datos que enviamos al servidor en el 'send'
  headers:{
    'Content-Type': 'application/json'
  }
})
.then(response => {
  if (response.ok) {
    response.json().then(datos => datosServidor=datos)
  } else {
    console.log('Error en la petición HTTP: '+response.status+' ('+response.statusText+')');
  }
```

```
14  })
15  .catch(err => {
16    console.log('Error en la petición HTTP: '+err.message);
17  })
```

Podéis ver mś ejemplos en MDN web docs y otras páginas.

6. *async / await*

Estas nuevas instrucciones introducidas en ES2016 nos permiten escribir el código de peticiones asíncronas como si fueran síncronas lo que facilita su comprensión. Tened en cuenta que NO están soportadas por navegadores antiguos.

Usando esto sí funcionaría el primer ejemplo que hicimos.

La palabra **async** se antepone a *function* al declarar una función e indica que esa función va a hacer una llamada asíncrona. Al anteponerle *async* se 'envuelve' automáticamente esa función en una promesa (o sea que esa función pasa automáticamente a devolver una promesa, a la que podríamos ponerle un `.then()`).

A cualquier función declarada con *async* se la puede llamar anteponiendo la palabra **await** a la llamada, lo que provocará que la ejecución se "espere" a que se resuelva la promesa devuelta por esa función. Esto hace que nuestro código se asemeje a código síncrono ya que no continua ejecutándose el código que hay después de un *await* hasta que esa petición se ha resuelto.

Tened en cuenta que si una función contiene un *await* se convierte en una función asíncrona ya que su ejecución se detendrá, por lo que debemos anteponerle al declararla la palabra *async*.

Podéis ver algunos ejemplos del uso de *async* / *await* en la página de MDN.

Siguiendo con el ejemplo de *fetch*:

```
1  async function pideDatos() {
2    const response = await
   fetch('https://jsonplaceholder.typicode.com/posts?userId=' + idUser);
3    const myData = await response.json();
4    return myData;
5  }
6  ...
7  // Y la llamaremos con
8  const myData = await pideDatos();
```

Fijaos en la diferencia: si hago

```
  const response = fetch('https://jsonplaceholder.typicode.com/posts?userId=' + idUser);
```

estaré obteniendo en *response* una promesa y para obtener el valor debería hacer `response.then()`, pero si hago

```
  const response = await fetch('https://jsonplaceholder.typicode.com/posts?userId=' + idUser);
```

lo que obtengo en *response* es ya el valor devuelto por la promesa cuando se resuelve.

Con esto conseguimos que llamadas asíncronas se comporten como instrucciones síncronas lo que aporta claridad al código.

El ejemplo de los posts quedaría:

```
const SERVER = 'https://jsonplaceholder.typicode.com'
const tbody = document.querySelector('tbody')

window.addEventListener('load', () => {
  document.getElementById('form-show').addEventListener('submit', async (event) => {
    event.preventDefault();
    let idUser = document.getElementById('id-usuario').value
    if (isNaN(idUser) || idUser.trim() == '') {
      alert('Debes introducir un número')
    } else {
      const posts = await getData(idUser)
      // La ejecución se para en la sentencia anterior hasta que
      // contesta la función getData
      tbody.innerHTML = ''
      posts.forEach((post) => {
        const newPost = document.createElement('tr');
        newPost.innerHTML = `
                <td>${post.userId}</td>
                <td>${post.id}</td>
                <td>${post.title}</td>
                <td>${post.body}</td>`
        tbody.appendChild(newPost)
      })
      document.getElementById('num-posts').textContent = posts.length
    }
  })
})
```

```
28
29 async function getData(idUser) {
30   const response = await fetch(SERVER + '/posts?userId=' + idUser)
31   const posts = await response.json()
32   return posts
33 }
```

6.1 Gestión de errores en *async/await*

En este código no estamos tratando los posibles errores que se pueden producir. Con *async* / *await* los errores se tratan como en las excepciones, con *try ... catch*:

```
1  const SERVER = 'https://jsonplaceholder.typicode.com'
2  const tbody = document.querySelector('tbody')
3
4  window.addEventListener('load', () => {
5    document.getElementById('form-show').addEventListener('submit', async
   (event) => {
6      event.preventDefault()
7      let idUser = document.getElementById('id-usuario').value
8      if (isNaN(idUser) || idUser.trim() == '') {
9        alert('Debes introducir un número')
10     } else {
11       try {
12         var posts = await getData(idUser)
13         // La ejecución se para en la sentencia anterior hasta que
14         // contesta la función getData
15       } catch (err) {
16         console.error(err)
17         return;
18       }
19       tbody.innerHTML = ''
20       posts.forEach((post) => {
21         const newPost = document.createElement('tr')
22         newPost.innerHTML = `
23                 <td>${post.userId}</td>
24                 <td>${post.id}</td>
25                 <td>${post.title}</td>
26                 <td>${post.body}</td>`
27         tbody.appendChild(newPost)
28       })
```

```
      document.getElementById('num-posts').textContent = posts.length
    }
  })
})

async function getData(idUser) {
  const response = await fetch(SERVER + '/posts?userId=' + idUser)
  const posts = await response.json()
  return posts
}
```

6.2 Hacer varias peticiones simultáneamente. Promise.all

En ocasiones necesitamos hacer más de una petición al servidor. Por ejemplo para obtener los productos y sus categorías podríamos hacer:

```
function getTable(table) {
  return new Promise((resolve, reject) => {
    fetch(SERVER + table)
      .then((response) => response.json())
      .then((data) => resolve(data))
      .catch((error) => reject(error))
  })
}

function getData() {
  getTable('/categories')
    .then((categories) => categories.forEach((category) => renderCategory(category)))
    .catch((error) => renderErrorMessage(error))
  getTable('/products')
    .then((products) => products.forEach((product) => renderProduct(product)))
    .catch((error) => renderErrorMessage(error))
}
```

Pero si para renderizar los productos necesitamos tener las categorías este código no nos lo garantiza ya que el servidor podría devolver antes los productos aunque los pedimos después.

Una solución sería no pedir los productos hasta tener las categorías:

```
function getData() {
  getTable('/categories')
    .then((categories) => {
      categories.forEach((category) => renderCategory(category))
      getTable('/products')
        .then((products) => products.forEach((product) =>
renderProduct(product)))
        .catch((error) => renderErrorMessage(error))
    })
    .catch((error) => renderErrorMessage(error))
}
```

pero esto hará más lento nuestro código al no hacer las 2 peticiones simultáneamente. La solución es usar el método `Promise.all()` al que se le pasa un array de promesas a hacer y devuelve una promesa que:

- se resuelve en el momento en que todas las promesas se han resuelto satisfactoriamente
- se rechaza en el momento en que alguna de las promesas es rechazada

El código anterior de forma correcta sería:

```
function getData() {
  Promise.all([
    getTable('/categories')
    getTable('/products')
  ])
  .then(([categories, products]) => {
    categories.forEach((category) => renderCategory(category)))
    products.forEach((product) => renderProduct(product)))
  })
  .catch((error) => renderErrorMessage(error))
}
```

Lo mismo pasa si en vez de promesas usamos *async/await*. Si hacemos:

```
async function getTable(table) {
    const response = await fetch(SERVER + table)
    const data = await response.json()
    return data
}

async function getData() {
  const responseCategories = await getTable('/categories');
  const responseProducts = await getTable('/products');
  categories.forEach((category) => renderCategory(category)))
  products.forEach((product) => renderProduct(product)))
}
```

tenemos el problema de que no comienza la petición de los productos hasta que se reciben las categorías. La solución con `Promise.all()` sería:

```
async function getData() {
  const [categories, products] = await Promise.all([
    getTable('/categories')
    getTable('/products')
  ])
  categories.forEach((category) => renderCategory(category)))
  products.forEach((product) => renderProduct(product)))
}
```

7. Single Page Application

Ajax es la base para construir SPAs que permiten al usuario interactuar con una aplicación web como si se tratara de una aplicación de escritorio (sin 'esperas' que dejen la página en blanco o no funcional mientras se recarga desde el servidor).

En una SPA sólo se carga la página de inicio (es la única página que existe) que se va modificando y cambiando sus datos como respuesta a la interacción del usuario. Para obtener los nuevos datos se realizan peticiones al servidor (normalmente Ajax). La respuesta son datos (JSON, XML, …) que se muestran al usuario modificando mediante DOM la página mostrada (o podrían ser trozos de HTML que se cargan en determinadas partes de la página, o ...).

8. Resumen de llamadas asíncronas

Una llamada Ajax es un tipo de llamada asíncrona fácil de entender que podemos hacer en Javascript aunque hay muchas más, como un `setTimeout()` o las funciones manejadoras de eventos. Como hemos visto, para la gestión de las llamadas asíncronas tenemos varios métodos y los más comunes son:

- funciones *callback*
- *promesas*
- *async* / *await*

Cuando se produce una llamada asíncrona el orden de ejecución del código no es el que vemos en el programa ya que el código de respuesta de la llamada no se ejecutará hasta completarse esta. Podemos ver un ejemplo de esto extraído de **todoJS** usando **funciones *callback***.

Además, si hacemos varias llamadas tampoco sabemos el qué orden se ejecutarán sus respuestas ya que depende de cuándo finalice cada una como podemos ver en este otro ejemplo.

Si usamos funciones *callback* y necesitamos que cada función no se ejecute hasta que haya terminado la anterior debemos llamarla en la respuesta a la función anterior lo que provoca un tipo de código difícil de leer llamado *callback hell*.

Para evitar esto surgieron las ***promesas*** que permiten evitar las funciones *callback* tan difíciles de leer. Podemos ver el primer ejemplo usando promesas. Y si necesitamos ejecutar secuencialmente las funciones evitaremos la pirámide de llamadas *callback* como vemos en este ejemplo.

Aún así el código no es muy limpio. Para mejorarlo tenemos ***async* y *await*** como vemos en este ejemplo . Estas funciones forman parte del estándar ES2017 por lo que no están soportadas por navegadores antiguos (aunque siempre podemos transpilar con *Babel*).

Fuente: todoJs: Controlar la ejecución asíncrona

9. CORS

Cross-Origin Resource Sharing (CORS) es un mecanismo de seguridad que incluyen los navegadores y que por defecto impiden que se pueden realizar peticiones Ajax desde un navegador a un servidor con un dominio diferente al de la página cargada originalmente.

Si necesitamos hacer este tipo de peticiones necesitamos que el servidor al que hacemos la petición añada en su respuesta la cabecera *Access-Control-Allow-Origin* donde indiquemos el dominio desde el que se pueden hacer peticiones (o * para permitirlas desde cualquier dominio).

El navegador comprobará las cabeceras de respuesta y si el dominio indicado por ella coincide con el dominio desde el que se hizo la petición, esta se permitirá.

Como en desarrollo normalmente no estamos en el dominio de producción (para el que se permitirán las peticiones) podemos instalar en el navegador la extensión *allow CORS* que al activarla deshabilita la seguridad CORS en el navegador.