JavaScript - Ajax 1

Introducción

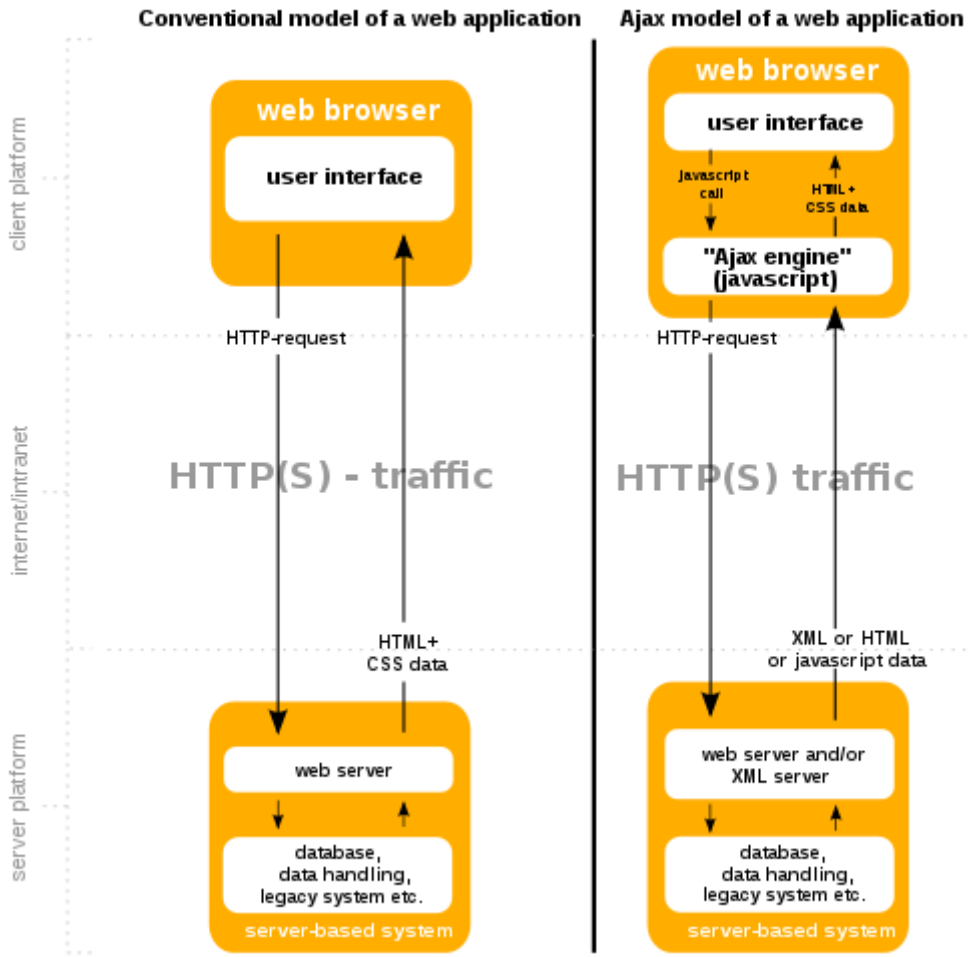
AJAX es el acrónimo de ***Asynchronous Javascript And XML*** (Javascript asíncrono y XML) y es lo que usamos para hacer peticiones asíncronas al servidor desde Javascript.

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 (cliente) 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 habíamos indicado al hacer la llamada Ajax.

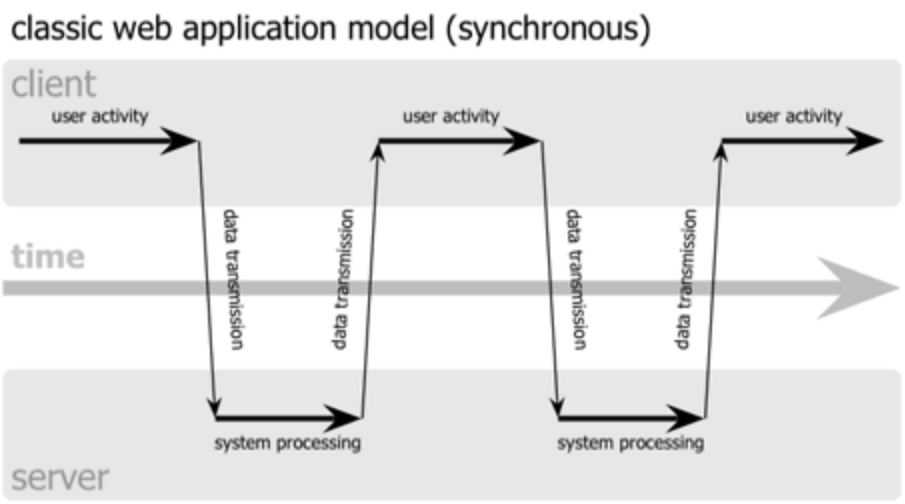
Respecto a **XML**: es el formato en que se intercambia la información entre el servidor y el cliente, aunque actualmente el formato más usado es **JSON** que es más simple y legible. Podríamos decir que hoy no se usa Ajax y sí Ajaj, pero suena a que te pica la garganta, así que seguiremos diciendo Ajax aunque usemos JSON.

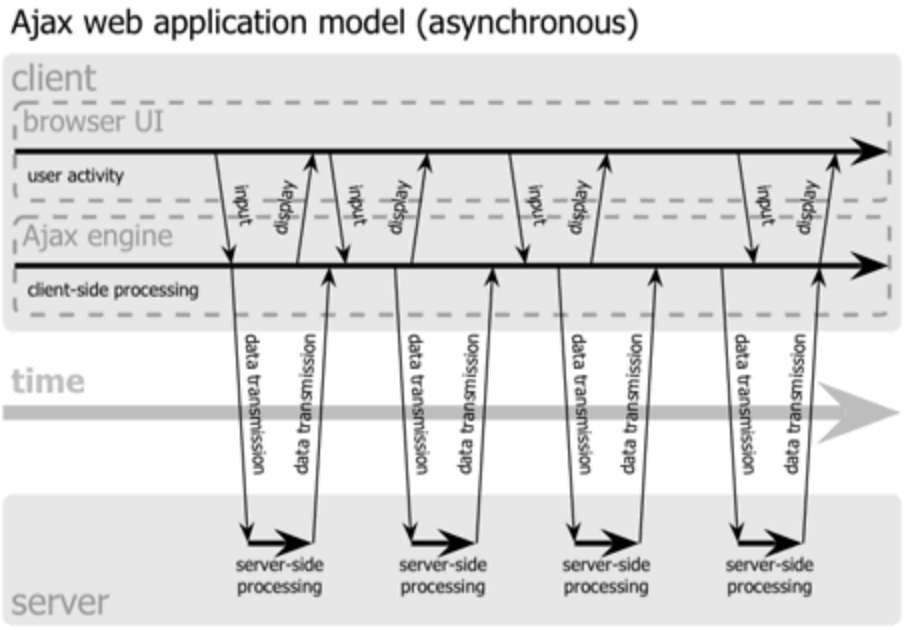
Básicamente Ajax nos permite poder mostrar nuevos datos enviados por el servidor sin tener que recargar la página, que continuará disponible mientras se reciben y procesan los datos enviados por el servidor en segundo plano.



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:





Métodos HTTP

Las peticiones Ajax usan el protocolo HTTP (el mismo que utiliza el navegador para cargar una página).

El protocolo HTTP envía al servidor:

a) Unas cabeceras HTTP con información como el *userAgent* del navegador, el idioma, etc.
b) El tipo de petición.
c) 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. el 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). La información para identificar el objeto

a modificar va 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 el 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 de peticiones que no trataremos en esta documentación, que son:
  - HEAD: pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.
  - CONNECT: establece un túnel hacia el servidor identificado por el recurso.
  - OPTIONS: es utilizado para describir las opciones de comunicación para el recurso de destino.
  - TRACE: realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso de destino.

El servidor acepta la petición, la procesa y envía al cliente una respuesta que consiste en:

a) El recurso solicitado.
b) Adjunta unas cabeceras de respuesta que incluyen: el tipo de contenido enviado, el idioma, etc.
c) 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. Ejemplos: 500 (error interno del servidor) o 504 (*timeout*, el servidor no responde).

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, …

Formato JSON

El formato *JSON* (acrónimo de JavaScript Object Notation, 'notación de objeto de JavaScript') es una forma de convertir objetos Javascript en una cadena de texto para poderlos enviar.

La estructura de datos JSON se compone de un conjunto de objetos o arrays que contendrán números, cadenas booleanos y nulos.

Un **objeto** JSON comienza y termina con llaves, y contiene una colección desordenada de pares **nombre-valor**.

Cada **nombre** y **valor** están separados por dos puntos, y los pares están separados por comas.

La coma final está prohibida.

El **nombre** es una cadena entre comillas dobles. Los caracteres de comillas no deben ser inclinadas o “inteligentes”.

En los **números**, los ceros a la izquierda están prohibidos; un punto decimal debe estar seguido al menos por un dígito.

En las cadenas deben estar entre comillas dobles. No se permiten todos los caracteres de escape; sí se permiten los caracteres de separador de línea Unicode (U+2028) y el separador de párrafo (U+2029).

Un array JSON comienza y termina con corchetes y contiene una colección ordenada de valores separados por comas. Un valor puede ser una cadena entre comillas dobles, un número, un booleano true o false, nulo, un objeto JSON o un array.

Los objetos y los arrays JSON se pueden anidar, lo que posibilita una estructura jerárquica de datos.

En el siguiente ejemplo, se muestra una estructura de datos JSON con dos objetos válidos.

```
{
    "id": 1006410,
    "title": "Amazon Redshift Database Developer Guide"
}
{
    "id": 100540,
    "name": "Amazon Simple Storage Service User Guide"
}
```

En el siguiente se muestran los mismos datos como dos arrays JSON:

```
[
    1006410,
    "Amazon Redshift Database Developer Guide"
]
[
    100540,
    "Amazon Simple Storage Service User Guide"
]
```

Conversiones de objetos

El objeto alumno:

```
let alumno = {
  id: 5,
  nombre: 'Ana',
  apellidos: 'Zubiri Peláez'
}
```

se transformaría en la cadena de texto:

```
{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }
```

y el array:

```
let alumnos = [
  {
    id: 5,
    nombre: "Ana",
    apellidos: "Zubiri Peláez"
  },
```

```
  {
    id: 7,
    nombre: "Carlos",
    apellidos: "Pérez Ortíz"
  },
]
```

en la cadena:

```
[{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }, { "id": 7, "nombre": "Carlos", "apellidos": "Pérez Ortíz" }]
```

Nótese que tanto las claves como los valores van entrecomillados (con comillas dobles). No sirven comillas simples.

Estructura de los datos

Los mismos datos pueden tener distinta estructura. Ejemplo:

Archivo colores1.json

```
{
  "arrayColores":[{
      "nombreColor":"rojo",
      "valorHexadec":"#f00"
    },
    {
      "nombreColor":"verde",
      "valorHexadec":"#0f0"
    },
    {
      "nombreColor":"azul",
      "valorHexadec":"#00f"
    },
    {
      "nombreColor":"cyan",
      "valorHexadec":"#0ff"
    },
    {
      "nombreColor":"magenta",
      "valorHexadec":"#f0f"
    },
    {
      "nombreColor":"amarillo",
      "valorHexadec":"#ff0"
    },
    {
      "nombreColor":"negro",
      "valorHexadec":"#000"
    }
  ]
}
```

Archivo colores2.json

```
{
  "arrayColores":[{
      "rojo":"#f00",
      "verde":"#0f0",
      "azul":"#00f",
      "cyan":"#0ff",
      "magenta":"#f0f",
      "amarillo":"#ff0",
      "negro":"#000"
    }
  ]
}
```

Archivo colores3.json

```
{
  "rojo":"#f00",
  "verde":"#0f0",
  "azul":"#00f",
  "cyan":"#0ff",
  "magenta":"#f0f",
  "amarillo":"#ff0",
  "negro":"#000"
}
```

Los ejemplos anteriores representan lo que podrían ser archivos JSON conteniendo datos en formato JSON.

Se trata de 3 archivos que contienen aproximadamente la misma información. Sin embargo, hay algunas diferencias:

- En el archivo **colores1.json** existe un único objeto de datos donde el nombre es *arrayColores* y su valor es un array de objetos JSON. Cada objeto del array está formado por los pares (*nombreColor* y su valor), y (*valorHexadec* y su valor). En este ejemplo en concreto el array consta de 7 elementos con información correspondiente a 7 colores.
- En el archivo **colores2.json** existe un único objeto de datos donde el nombre es *arrayColores*, cuyo valor es un array que contiene un único objeto JSON formado por siete pares (nombre – valor) que representa información sobre siete colores.
- En el archivo **colores3.json** existe un único objeto de datos que está formado por siete pares (nombre – valor) que representa información sobre siete colores.

Siendo las 3 formas válidas, se deberá utilizar aquella que se indique en las instrucciones o, de no existir pautas precisas, utilizar aquel diseño que favorezca el desarrollo y mantenimiento de la aplicación.

Métodos del objeto JSON

Para convertir objetos en cadenas de texto *JSON* y viceversa, Javascript proporciona 2 métodos:

- **JSON.stringify(*objeto*)**: recibe un objeto JS y devuelve la cadena de texto correspondiente.

```
//convierte de formato JSON a objeto
const cadenaAlumnos = JSON.stringify(alumnos)
```

- **JSON.parse(*cadena*)**: realiza el proceso inverso, convirtiendo una cadena de texto en un objeto.

```
//convierte un objeto a formato JSON
const alumnos = JSON.parse(cadenaAlumnos)
```

Json Server

Las peticiones Ajax se hacen a un servidor que proporcione una API.

Se puede utilizar **Json Server** que es un servidor API-REST que funciona bajo Node.js

(nota para mí: me ha fallado Node ver.18, instalé versión 16 y bien)

Si aún no está instalado Node.js, se puede hacer desde https://nodejs.org/es/

Node.js utiliza un fichero JSON como contenedor de los datos en lugar de una base de datos.

Para instalar json-server en nuestra máquina, lo instalaremos de forma global para poderlo utilizar en otros ejercicios, desde cualquier terminal hay que ejecutar **npm** (Node Package Manager) de la siguiente forma:

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

Para que sirva (provea de este servicio los datos de) un fichero datos.json, se ejecuta la sentencia:

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

Se puede poner la opción `--watch` (o `-w`) 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": []
}
```

Otro ejemplo de un fichero json con 2 tablas: películas y clasificaciones

Películas y clasificaciones son dos entidades distintas.

```
{
```

```
"peliculas": [
  {
    "id": 1,
    "nombre": "El sexto sentido",
    "director": "M. Night Shyamalan",
    "clasificacion": "Drama"
  },
  {
    "id": 2,
    "nombre": "Pulp Fiction",
    "director": "Tarantino",
    "clasificacion": "Acción"
  },
  {
    "id": 3,
    "nombre": "Todo Sobre Mi Madre",
    "director": "Almodobar",
    "clasificacion": "Drama"
  },
  {
    "id": 4,
    "nombre": "300",
    "director": "Zack Snyder",
    "clasificacion": "Acción"
  },
  {
    "id": 5,
    "nombre": "El silencio de los corderos",
    "director": "Jonathan Demme",
    "clasificacion": "Drama"
  },
  {
    "id": 6,
    "nombre": "Forrest Gump",
    "director": "Robert Zemeckis",
    "clasificacion": "Comedia"
  },
  {
    "id": 7,
    "nombre": "Las Hurdes",
    "director": "Luis Buñuel",
    "clasificacion": "Documental"
  }
],
"clasificaciones": [
  {
    "nombre": "Drama",
    "id": 1
  },
  {
    "nombre": "Comedia",
    "id": 2
```

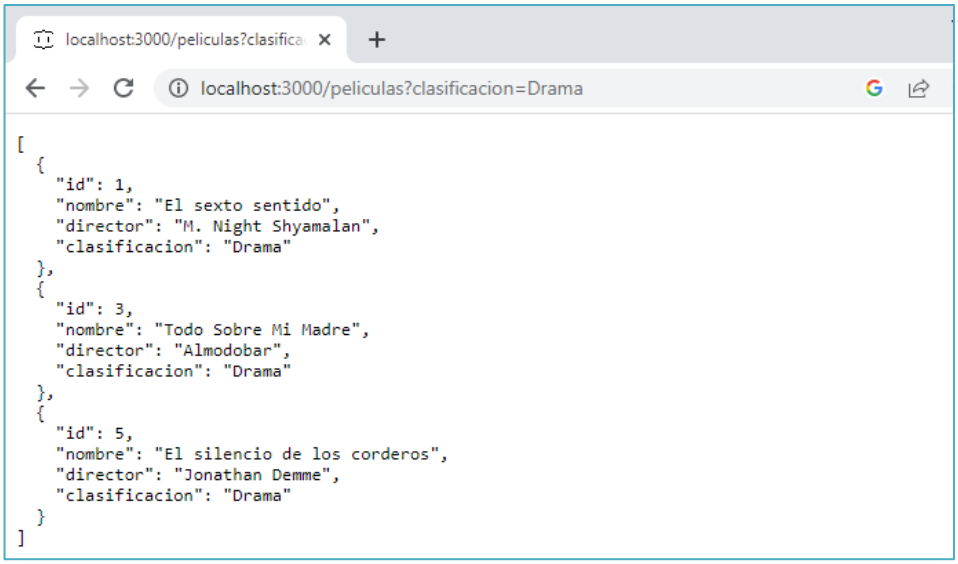
```
    },
    {
      "nombre": "Acción",
      "id": 4
    },
    {
      "nombre": "Terrorifica",
      "id": 15
    }
  ]
}
```

AL iniciar el servicio, por defecto la API escucha en el puerto 3000 y servirá los diferentes objetos definidos en el fichero *.json*. Por ejemplo:

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

También pueden hacerse peticiones más complejas como:

- http://localhost:3000/peliculas?clasificacion=Drama: devuelve un array con todos los elementos de *peliculas* cuya propiedad *clasificacion* valga *Drama. La siguiente imagen muestra el resultado:*



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 ejecutar el servidor en la máquina 192.168.0.10 pondremos:

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

Si se desea cambiar el puerto por defecto (3000) donde escucha el servidor, a otro (por ejemplo, el 4200):

```
json-server --host 192.168.0.10 -p 4200 datos.json
```

Y la ruta para acceder a la API será `http://192.168.0.10:4200`.

EJERCICIO: 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

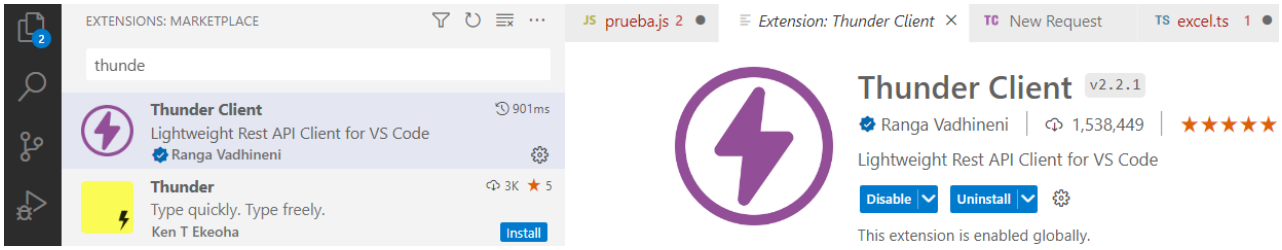
REST client

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.

Existen multitud de aplicaciones para realizar peticiones HTTP, como Advanced REST client, Postman, etc.

Además, cada navegador tiene sus propias extensiones para hacer esto, como *Advanced Rest Client* para Chrome o *RestClient* para Firefox.

También se puede utilizar una extensión para Visual Studio Code llamada *Thunder Client*.

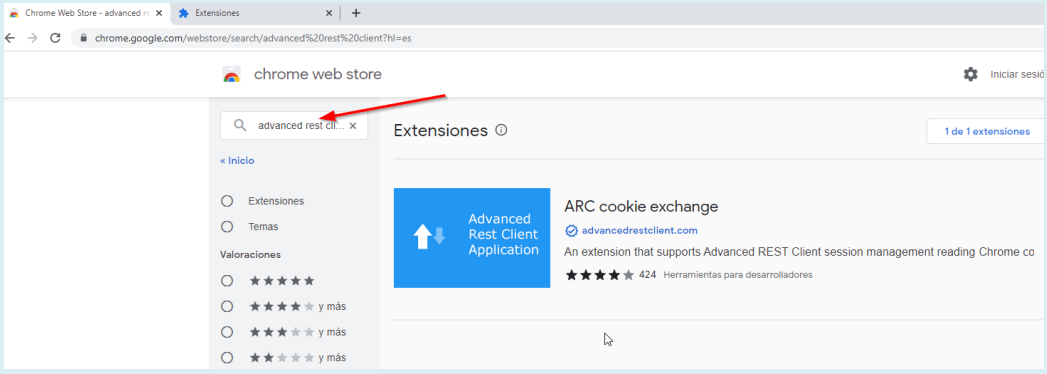


Para realizar el siguiente ejercicio hay que instalar alguna de las extensiones mencionadas y hacer todas las peticiones desde allí (incluyendo los GET), lo que permitirá ver los códigos de estado devueltos, las cabeceras, etc.

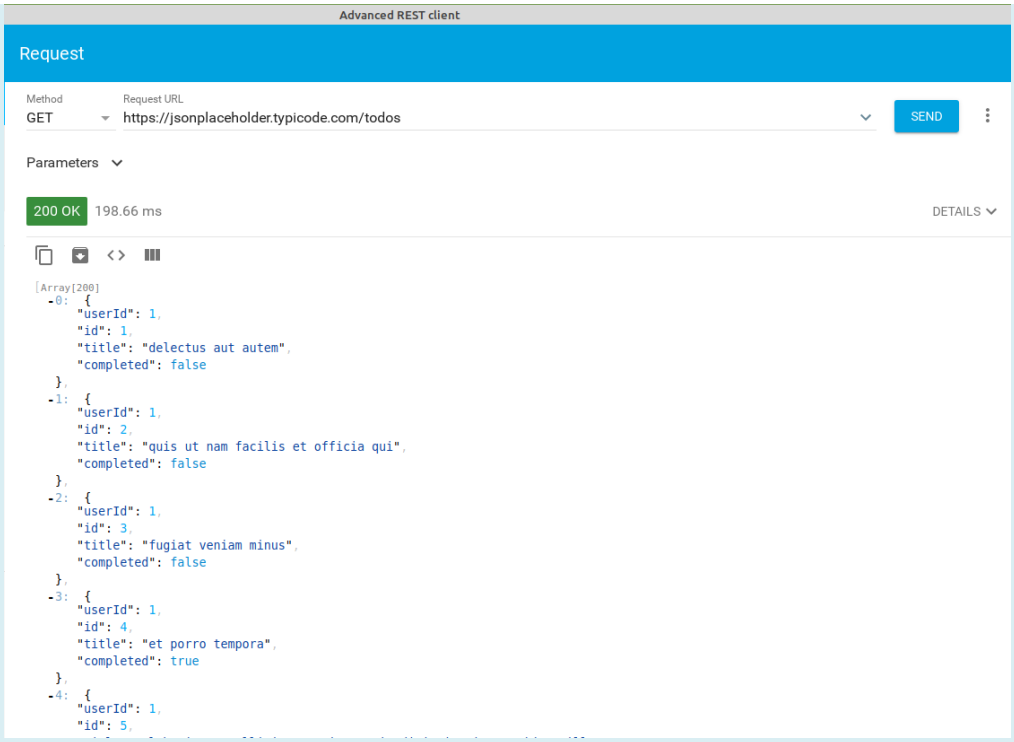
EJERCICIO: Vamos a realizar diferentes peticiones HTTP a la **API https://jsonplaceholder.typicode.com**. En concreto trabajaremos contra la tabla *todos* (que contiene **tareas** para hacer).

Esta API está disponible para hacer pruebas sin necesidad de montar un servidor.

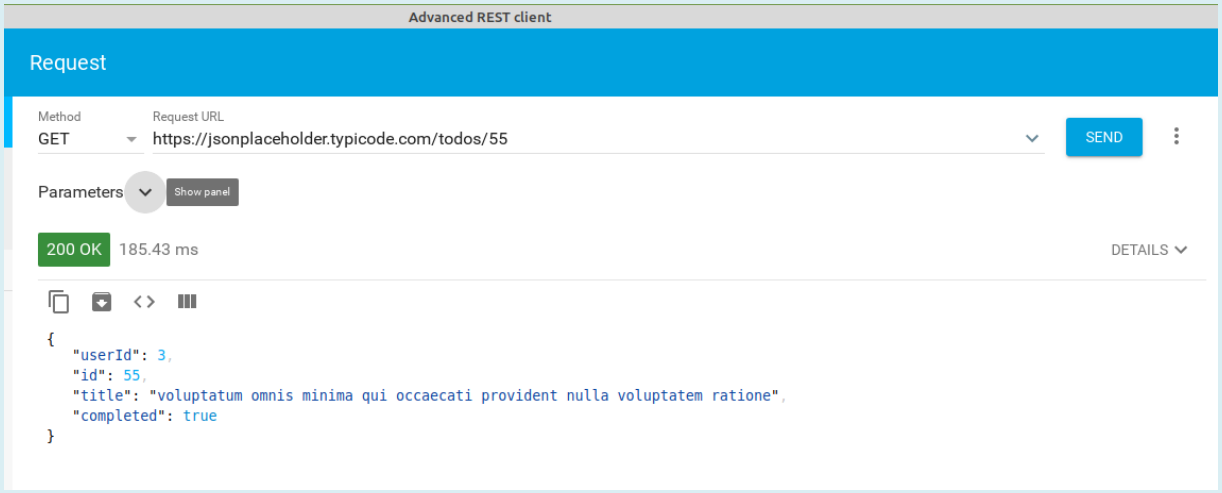
Para la resolución de este ejercicio se instaló la extensión Advanced Rest Client en Google Chrome:



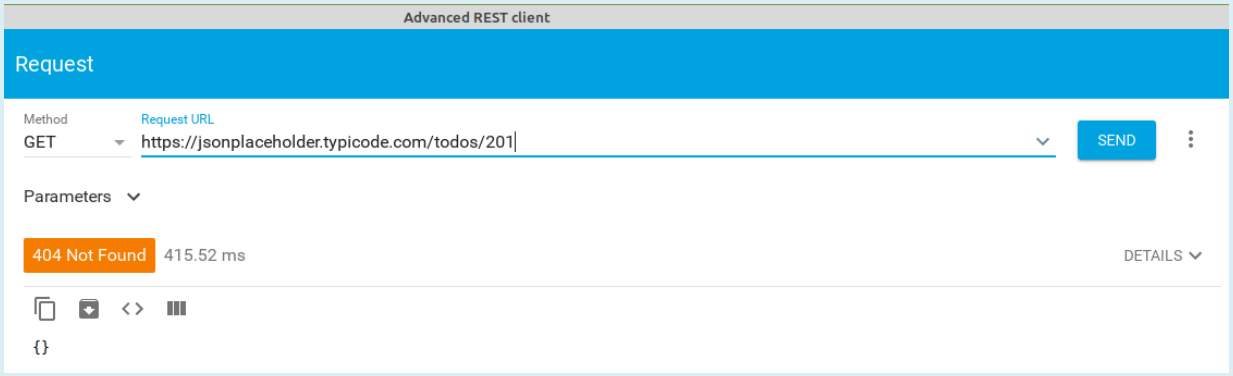
1º Obtener todas las tareas de la tabla todos. Devolverá un array con todas las tareas y el código devuelto será 200 – Ok



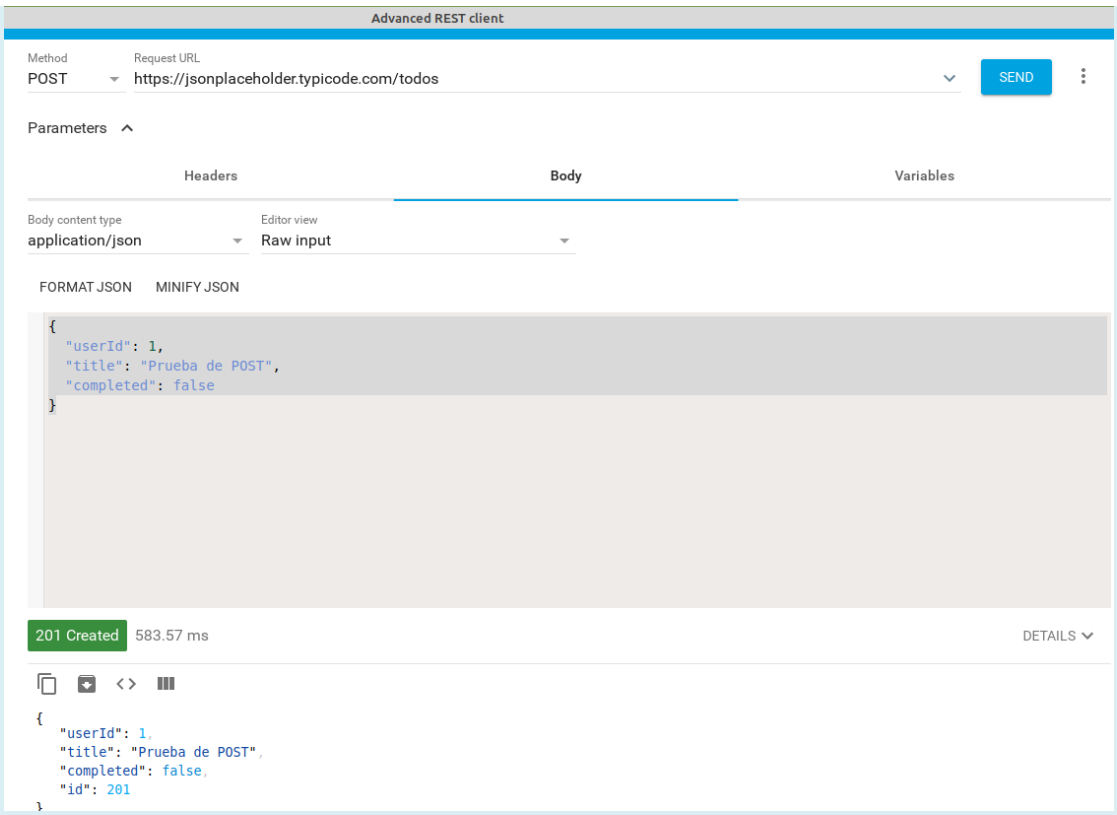
2º Obtener la tarea cuyo id sea 55. Devolverá el objeto de la tarea 55 y el código devuelto será 200 – Ok



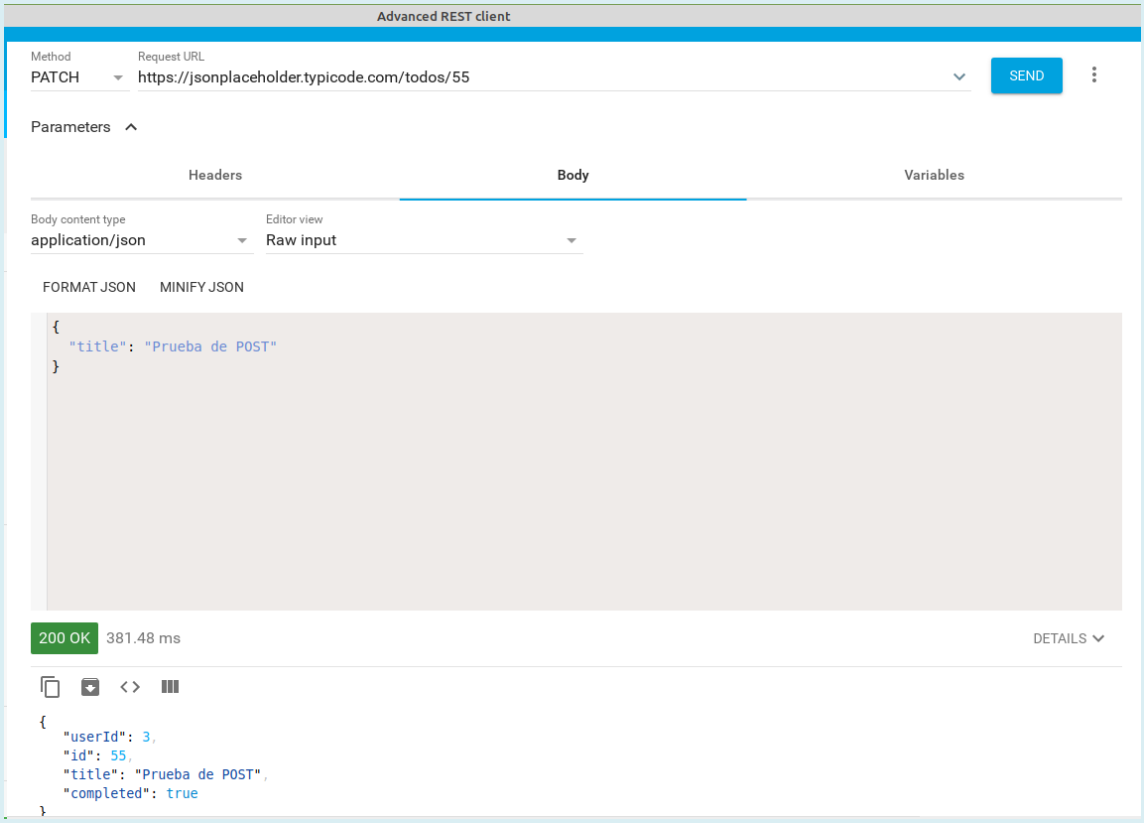
3º Obtener la tarea con id 201. Como no existe, devolverá un objeto vacío y el código de error 404 - Not found.



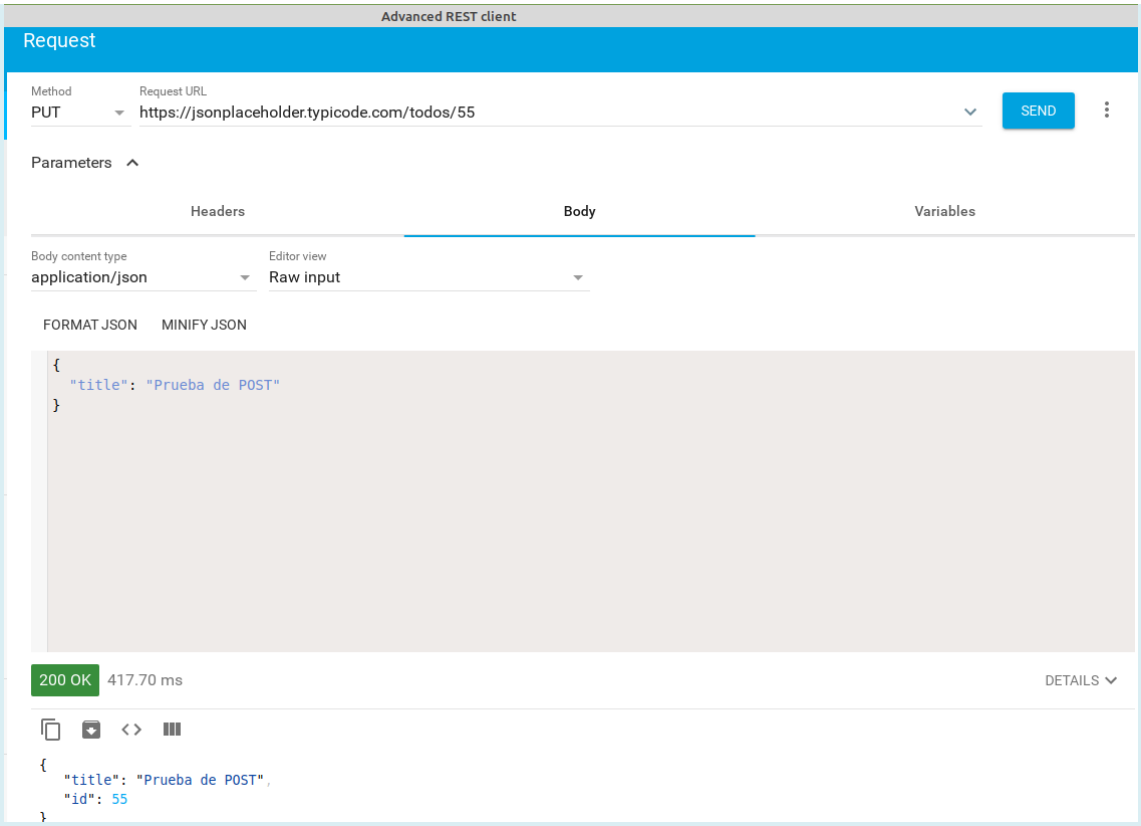
4º Crear una nueva tarea. En el cuerpo de la petición le pasaremos sus datos: *userID*: 1, *title*: Prueba de POST y *completed*: false. No se le pasa la id (de eso se encarga la BBDD, que la creará automática). La respuesta debe ser un código 201 (created) y adjuntará el nuevo registro creado con todos 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.



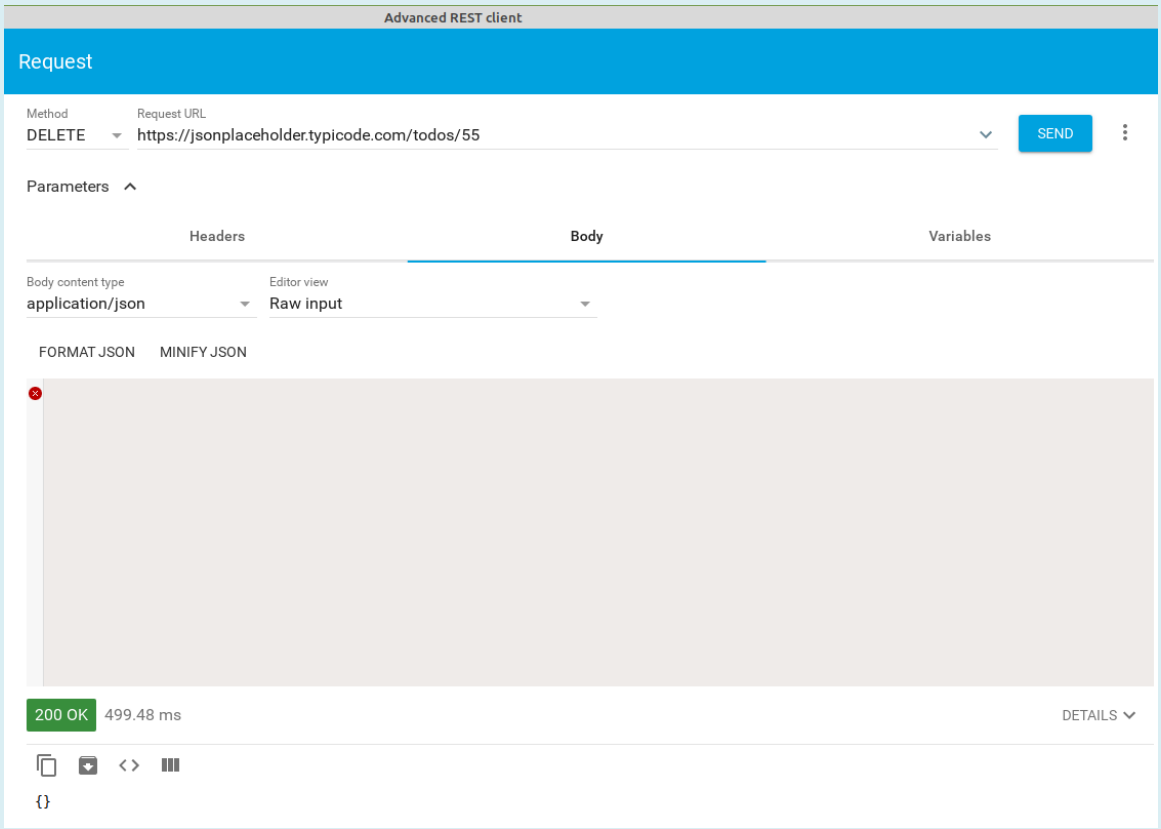
5º Modificar con un PATCH la tarea con id 55 para que su title sea 'Prueba de POST'. Devolverá el nuevo registro con un código 200. Se observa que al hacer un PATCH los campos que no se pasan se mantienen como estaban.



6º 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 se aprecia en la imagen, en esta API los campos que no se pasan se eliminan; en otras los campos no pasados se mantienen como estaban.

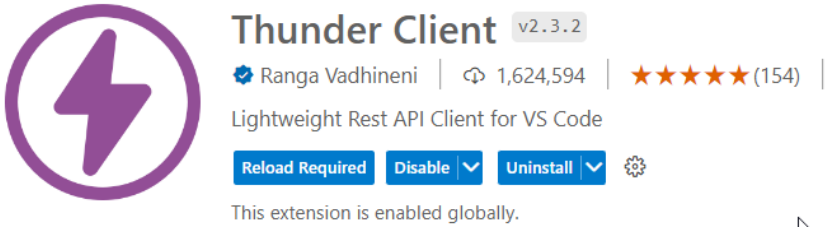


7º Eliminar con DELETE la tarea con id 55. Como se ve en la siguiente imagen, esta API devuelve un objeto vacío al eliminar; otras devuelven el objeto eliminado.

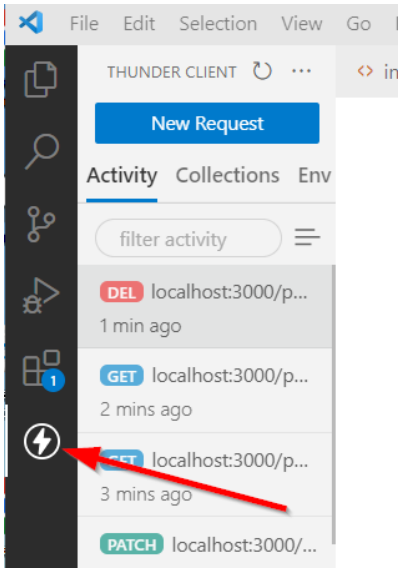


Thundert Client para VSC y json-server

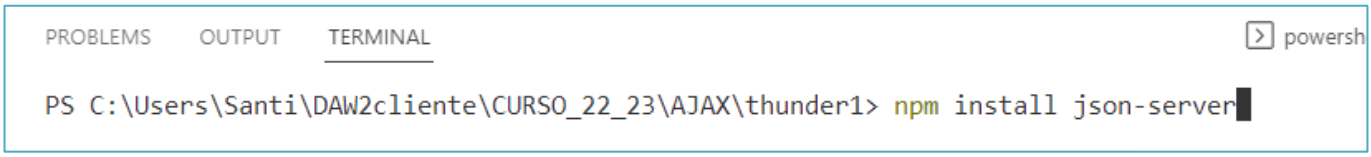
Se instala como una versión de Visual Studio Code



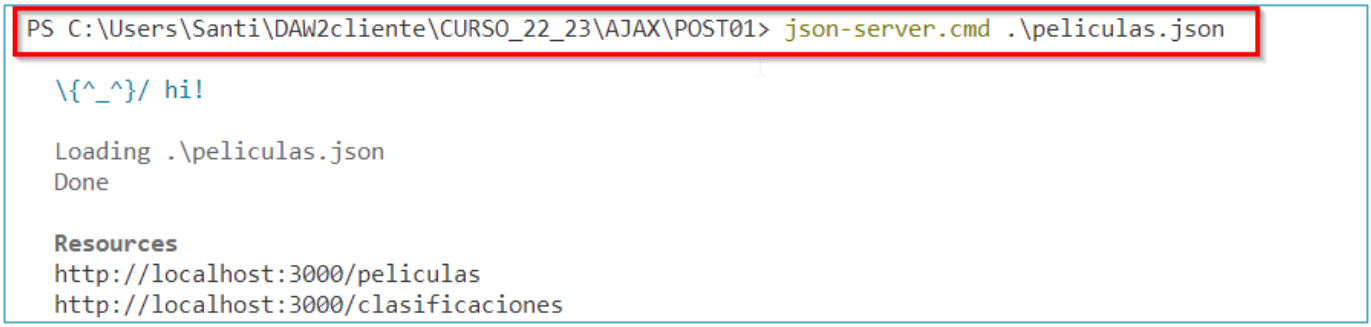
Una vez instalada nos proporciona en VSC una interfaz cómoda para realizar peticiones.



Para trabajar con json-server, podemos utilizar un terminal de VSC, tanto para instalar json-server:



como para servir los datos de un fichero json:



Si en algún momento en el terminal nos apareciera un mensaje advirtiendo que no se pueden ejecutar scripts, podemos desde el mismo terminal o desde un terminal PowerShell activar la política que deseemos sobre la ejecución de scripts. Se recomienda el modo RemoteSigned.