

## Servicios WEB en PHP. Nivel I

junio, 2019



## Objetivos del nivel

- Entender la computación orientada a servicios
- Aprender a estructurar datos en el formato JSON
- Aprender a consumir datos de un servicio web REST
- Crear servicios Web REST en PHP

## Prerrequisitos del nivel

- PHP Nivel III

## Acerca de este manual

Este manual pertenece al Centro de Asesoramiento y Desarrollo Informático C.A. (CADIF1). Para obtener más información sobre este u otros cursos visite nuestro sitio Web [www.cadif1.com](http://www.cadif1.com), escribanos a la dirección de correo [cadi@cadif1.com](mailto:cadi@cadif1.com) o visítenos en nuestra sede ubicada en la Av. Pedro León Torres con calle 59, Centro Comercial Sotavento, piso 2 oficina 27, Barquisimeto estado Lara, Venezuela. Tlf. 0251-7179247, 0251-4410268.

Las marcas mencionadas en este manual son propiedad de sus respectivos dueños.  
Copyright 2019. Todos los derechos reservados.

ACADEMIA DE SOFTWARE



## Contenido del nivel

### Capítulo 1. Paradigma Backend - Frontend

- 1.1.- Servicios Web (el Backend).
- 1.2.- Cliente.
- 1.3.- Comunicación.
- 1.4.- Ventajas y Desventajas.

### Capítulo 2. Métodos de Http

- 2.1.- Métodos de Envío.
- 2.2.- Códigos de Respuesta.
- 2.3.- El Header.



### Capítulo 5. Php Como Cliente

- 5.1.- Curl.
- 5.2.- Configuración de Petición.
- 5.3.- Envio de Headers.
- 5.4.- Envio de Body.

### Capítulo 6. Patrón Front Controller

- 6.1.- Mvc.

- 6.2.- Front Controller.
- 6.3.- Modelo.
- 6.4.- Vista.

## Capítulo 7. Rest

- 7.1.- Rest.
- 7.2.- Operaciones.
- 7.3.- Rest en Php.

## Capítulo 8. Apikey

- 8.1.- Api Key.
- 8.2.- Generación.
- 8.3.- Verificación.

## Capítulo 9. Get. Parte 1

- 9.1.- Todos Los Recursos.
- 9.2.- Query Params.
- 9.3.- Recursos Específicos.

## Capítulo 10. Get. Parte 2

- 10.1.- Paginación.
- 10.2.- Filtro.

## Capítulo 11. Post. Parte 1

- 11.1.- Recibiendo Los Datos (request).
- 11.2.- Procesamiento.
- 11.3.- Respuesta (response).

## Capítulo 12. Post. Parte 2

- 12.1.- Validaciones.
- 12.2.- Validación de Estructura.

### 12.3.- Validación de Dato.

## Capítulo 13. Put y Delete

13.1.- Put.

13.2.- Delete.

## Capítulo 14. Jwt. Parte 1

14.1.- Jwt.

14.2.- Estructura y Encoding.

14.3.- Generación.

## Capítulo 15. Jwt. Parte 2

- 15.1.- Claim.
- 15.2.- Envio Jwt.
- 15.3.- Decodificar.

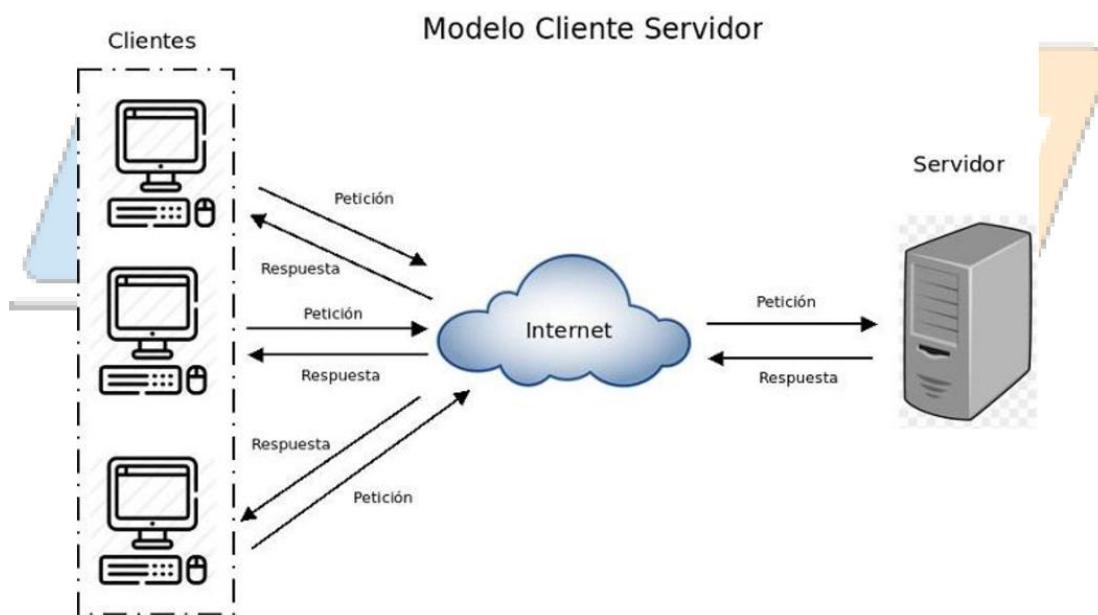


## Capítulo 1. PARADIGMA BACKEND - FRONTEND

### 1.1.- Servicios Web (el Backend)

La Arquitectura Cliente - Servidor es un modelo de diseño de software en el que las tareas o funciones se reparten entre los proveedores de recursos, llamados servidores, y donde existen demandantes de esos recursos, llamados clientes.

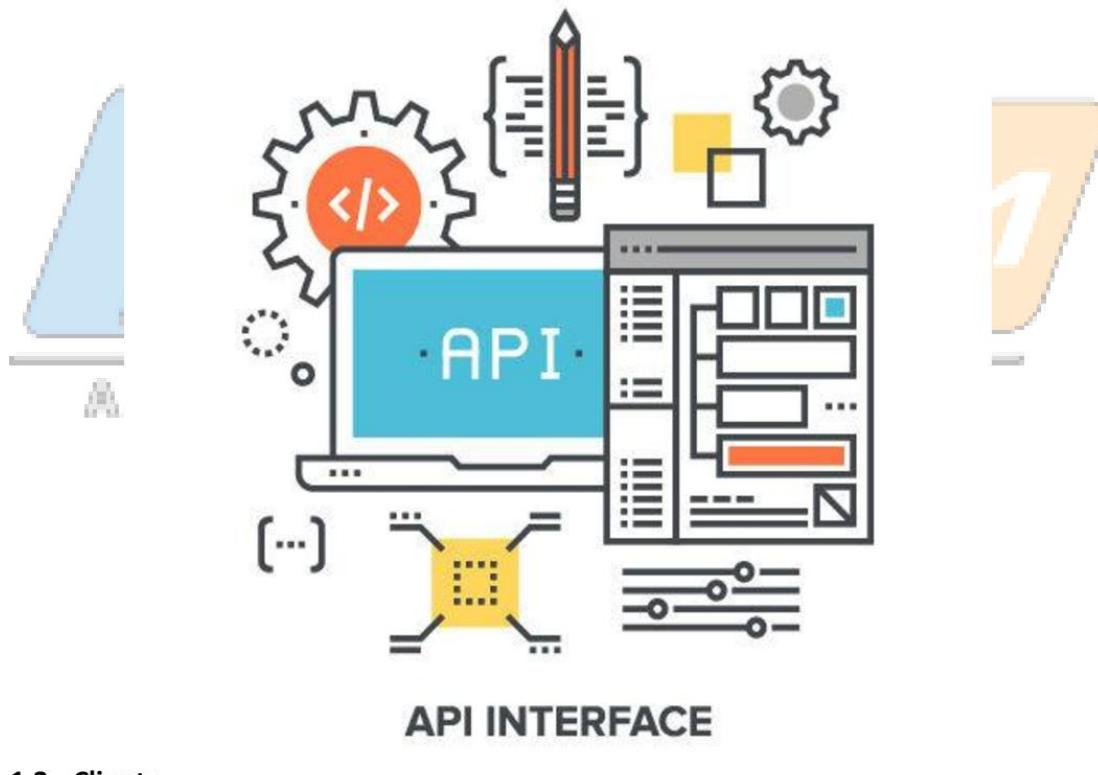
La aplicación que se aloja en el servidor debe tener la capacidad de brindar de funciones a sus clientes, permitiendo hacer operaciones desde el y otorgando de privilegios a quien lo utiliza, brindando así, una variedad de servicios a los clientes con dicha funciones.



En este paradigma, el servidor es el encargado de recibir todas las peticiones de un cliente y responder a ella de manera efectiva, ya sea con datos o archivos almacenados en este, como si se estuviera hablando de una librería de la cual se consumen sus métodos.

Todo esto en conjunto, tanto la capa de datos, como la aplicación en el servidor y hasta el mismo software servidor, son también conocidos o llamados el “Backend”.

Para que los desarrolladores de clientes puedan hacer uso del Backend, debe existir una forma de acceder a este, conocida con el nombre de API. Una API, siglas de Application Programming Interface o Interfaz de Programación de Aplicaciones, es un conjunto de código que se puede emplear para que varias aplicaciones se comuniquen entre ellas. Es similar a la interfaz de usuario a la hora de permitir la interacción entre persona y programa, solo que entre aplicaciones.

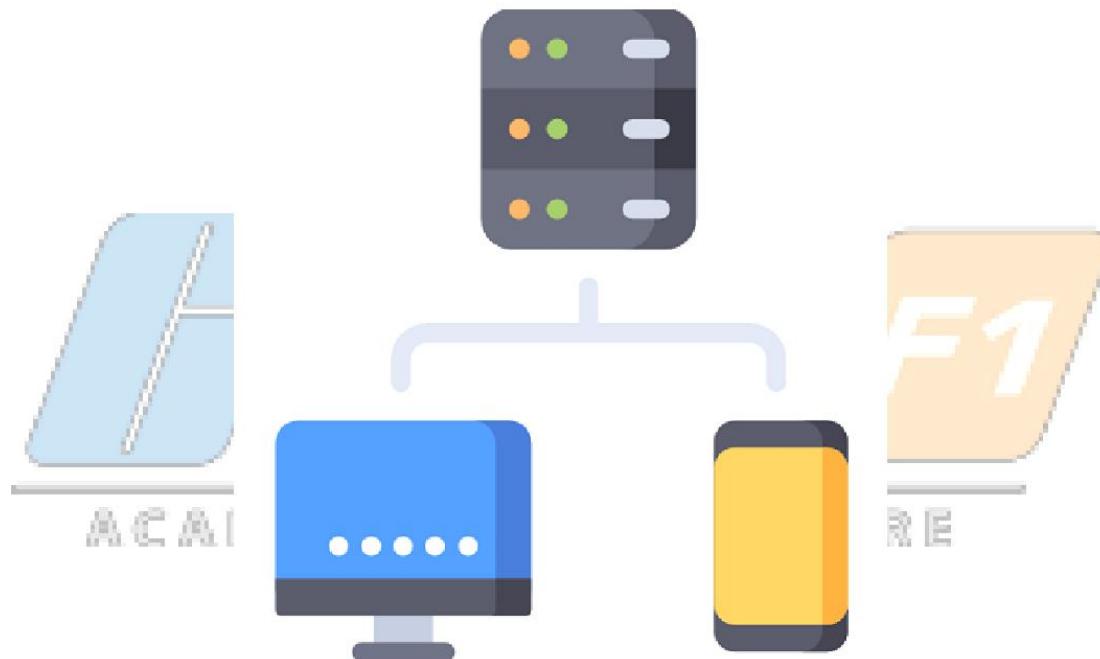


### 1.2.- Cliente

El cliente o "Frontend" es la parte de un programa o dispositivo a la que un usuario puede acceder y ver los datos recibidos de un backend. Son todas las tecnologías que se

encargan de la interactividad con los usuarios, es decir, es la interfaz que el usuario puede ver y manipular para poder acceder a los datos que se encuentran en un servidor.

Sin embargo, hay que tener en cuenta que al momento de hablar de clientes en la arquitectura cliente - servidor no solo se hace referencia a los clientes Web (JavaScript) que pueda estar consumiendo de un backend. Un cliente puede ser todo aquel programa que consuma información de un centro de datos (backend).



Hay que resaltar que un cliente puede estar en varios contextos, como se mencionó con anterioridad, puede ser una página web, pero además puede ser un arduino, un SmartTv, una lavadora, inclusive otro backend.



La razón por la que un backend se vuelve un cliente, es debido a que no todos los backend son autónomos en su información. Algunos necesitan conectarse a otros servicios para poder funcionar, algunos ejemplos de dichos servicios son: servicios de autenticación (como Google, Facebook, Twitter, Oauth, etc), servicio de notificaciones (como OneSignal para generar notificaciones "PUSH" en dispositivos móviles), Google Maps (para poder ver una ubicación o un sitio específico), entre otros. El consumir de estos servicios de terceros hace que el backend se convierta en un consumidor o cliente de otro backend, formando así una cadena de clientes y servidores.

### 1.3.- Comunicación

La forma en que los clientes y servidores se comunican para poder hacer la transmisión de datos, es a través del protocolo HTTP. Hypertext Transfer Protocol, más conocido como HTTP, es un protocolo de transferencia de datos entre servidor y cliente a través de Internet utilizado para poder navegar por la red y visitar páginas web.

De este protocolo de comunicación existen dos versiones, la primera versión del protocolo HTTP que data de 1965 y que conforma la gran parte de internet, y la versión del protocolo HTTPS, la cual es mucho más nueva, y no aparecieron las primeras conexiones a través de él hasta 1994, trayendo a este protocolo la "S" de "Secure", el cual brinda una capa de seguridad adicional.

Inicialmente, las conexiones HTTPS utilizaban el protocolo de seguridad SSL (Secure Socket Layer) para proteger las conexiones de los usuarios al utilizar plataformas muy concretas, como PayPal o webs bancarias. Hoy en día, este protocolo ha cambiado de SSL por TLS, (Transport Layer Security), protocolo que brinda mayor seguridad y mayor privacidad en las conexiones.



Un certificado SSL (Secure Sockets Layer) o TLS,(Transport Layer Security) es un título digital que autentifica la identidad de un sitio web cifrando la información que se envía al servidor. El cifrado es el proceso de mezclar datos en un formato indescifrable que sólo puede volver al formato legible con la clave de descifrado adecuada.

Un certificado sirve como un "pasaporte" electrónico que establece las credenciales de una entidad en línea al hacer negocios en la Web. Cuando un usuario de Internet intenta enviar información de credenciales a un servidor web, el navegador del usuario accede al certificado digital del servidor y establece una conexión segura.

#### 1.4.- Ventajas y Desventajas

Ventajas:

- Centralización del control: los accesos, recursos y la integridad de los datos son controlados por el servidor de forma que un programa cliente defectuoso o no autorizado no pueda dañar el sistema.
- Escalabilidad: se puede aumentar la capacidad de clientes y servidores por separado. Cualquier elemento puede ser aumentado (o mejorado) en cualquier momento, o se pueden añadir nuevos nodos a la red (clientes y/o servidores).
- Fácil mantenimiento: al estar distribuidas las funciones y responsabilidades entre varios ordenadores independientes, es posible reemplazar, reparar, actualizar, o incluso trasladar un servidor, mientras que sus clientes no se verán afectados por ese cambio (o se afectarán mínimamente).

Desventajas:

- La congestión del tráfico ha sido siempre un problema en el paradigma de Cliente - Servidor. Cuando una gran cantidad de clientes envían peticiones simultáneas al mismo servidor, puede ser que cause muchos problemas para éste (a mayor número de clientes, más problemas para el servidor).
- Cuando un servidor está caído, las peticiones de los clientes no pueden ser satisfechas. Por ende, la hasta que el servidor no sea reactivado los clientes quedan inútiles para el usuario.
- El software y el hardware de un servidor son generalmente muy determinantes. Normalmente se necesita software y hardware específico, sobre todo en el lado del servidor, para satisfacer el trabajo. Por supuesto, esto aumentará el costo.

## Capítulo 2. MÉTODOS DE HTTP

### 2.1.- Métodos de Envío

El protocolo HTTP define un conjunto de métodos de petición para indicar la acción que se desea realizar para acceder a un recurso determinado. Aunque estos también pueden ser sustantivos, estos métodos de solicitud a veces son llamados HTTP verbs. Cada uno de ellos implementan una semántica diferente, aunque la manera en la que se maneja la petición es muy parecida, tienen algunas reglas y formas de enviar y recibir los datos particulares para algunos. Algunos de los métodos HTTP disponibles son:

- GET
- HEAD
- OPTIONS
- POST
- PUT
- DELETE
- PATCH
- CONNECTION.

El OPTIONS representa una solicitud de información acerca de las opciones de comunicación disponibles en el canal de solicitud/respuesta (Petición). En otras palabras, éste método es el que se utiliza para describir las opciones de comunicación existentes de un recurso destino. Un uso para este método es el solicitar los métodos disponibles de un servidor.

```
1 curl -X OPTIONS http://blog.michelletorres.mx -i
```

```
1 HTTP/1.1 200 OK
2 Date: Wed, 8 Nov 2017 12:28:53 GMT
3 Server: Apache/2.2.14 (Win32)
4 Allow: GET,HEAD,POST,OPTIONS,TRACE
5 Content-Type: httpd/unix-directory
```

El GET se utiliza cuando se necesita adquirir un archivo o recurso que se encuentran en un servidor web.

```
1 GET /index.html HTTP/1.1
2 User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
3 Host: www.blog.michelletorres.mx
4 Accept-Language: es-mx
5 Accept-Encoding: gzip, deflate
6 Connection: Keep-Alive
```

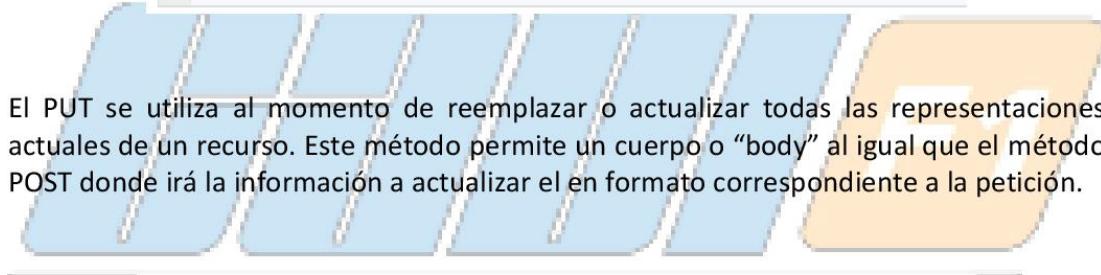


```
1 Ejemplo respuesta del servidor:
2 HTTP/1.1 200 OK
3 Date: Wed, 08 Nov 2017 12:28:53 GMT
4 Server: Apache/2.2.14 (Win32)
5 Last-Modified: Mon, 22 Jul 2014 19:15:56 GMT
6 ETag: "34aa387-d-1568eb00"
7 Vary: Authorization,Accept
8 Accept-Ranges: bytes
9 Content-Length: 88
10 Content-Type: text/html
11 Connection: Closed
```

El POST se usa para enviar información para cargar/crear un elemento nuevo. Este tipo de método se usa principalmente en el envío de formularios que se encuentran en las páginas web. La singularidad de este método se observa debido que contiene un cuerpo o body donde irá la información o archivos que se desea enviar al servidor para crear el recurso.

```
1 POST /cgi-bin/process.cgi HTTP/1.1
2 User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
3 Host: www.blog.michelletorres.mx
4 Content-Type: text/xml; charset=utf-8
5 Content-Length: 88
6 Accept-Language: es-mx
7 Accept-Encoding: gzip, deflate
8 Connection: Keep-Alive
```

```
1 HTTP/1.1 201 Created
2 Date: Mon, 27 Nov 2017 12:28:53 GMT
3 Server: Apache/2.2.14 (Win32)
4 Content-type: text/xml
5 Content-length: 30
6 Connection: Closed
```



El PUT se utiliza al momento de reemplazar o actualizar todas las representaciones actuales de un recurso. Este método permite un cuerpo o “body” al igual que el método POST donde irá la información a actualizar el en formato correspondiente a la petición.

```
1 PUT /index.htm HTTP/1.1
2 User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
3 Host: www.blog.michelletorres.mx
4 Accept-Language: es-mx
5 Connection: Keep-Alive
6 Content-type: text/html
7 Content-Length: 182
```

```
1 HTTP/1.1 200 OK
2 Date: Mon, 27 Nov 2017 12:28:53 GMT
3 Server: Apache/2.2.14 (Win32)
4 Content-type: text/html
5 Content-length: 30
6 Connection: Closed
```

El DELETE como su nombre lo dice, es el método por defecto para borrar algún recurso del servidor. Este método no contiene body, por lo tanto, la forma de borrar el elemento deseado es a través de un identificador o ruta específica.

```
1 DELETE /hello.htm HTTP/1.1
2 User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
3 Host: www.blog.michelletorres.mx
4 Accept-Language: es-mx
5 Connection: Keep-Alive
```

```
1 HTTP/1.1 200 OK
2 Date: Mon, 27 Jul 2009 12:28:53 GMT
3 Server: Apache/2.2.14 (Win32)
4 Content-type: text/html
5 Content-length: 30
6 Connection: Closed
7
8 <html>
9 <body>
10 <h1>URL deleted.</h1>
11 </body>
```

## 2.2.- Códigos de Respuesta

Cada solicitud que se genera, realizada desde un cliente recibe una respuesta. Dicha respuesta contiene un código de estado que indica si se ha completado satisfactoriamente o no la solicitud HTTP generada. Las respuestas se agrupan en cinco clases: respuestas informativas, respuestas satisfactorias, redirecciones, errores de los clientes y errores de los servidores.

Algunos de los códigos de estados más comunes son:

200 - OK:

Refiere a que la solicitud ha tenido éxito. El significado de un éxito varía dependiendo del método HTTP. Ejemplo:

GET: El recurso se ha obtenido y se transmite en el cuerpo del mensaje.

HEAD: Los encabezados de entidad están en el cuerpo del mensaje.

PUT o POST: El recurso que describe el resultado de la acción se transmite en el cuerpo del mensaje.

DELETE : El recurso solicitado fue borrado con éxito.

201 - Created:

La solicitud ha tenido éxito y se ha creado un nuevo recurso como resultado de ello. Ésta es típicamente la respuesta enviada después de una petición POST o PUT.

400 - Bad Request:

Esta respuesta significa que el servidor no pudo interpretar la solicitud debido a una sintaxis inválida.

401 - Unauthorized:

Esta respuesta refiere a que es necesario autenticarse para obtener la respuesta solicitada. Esta es similar a 403, pero en este caso, la autenticación es posible.

403 - Forbidden:

El cliente no posee los permisos necesarios para cierto contenido, por lo que el servidor está rechazando otorgar una respuesta apropiada. Esta respuesta es principalmente para que nadie pueda acceder a ese recurso, excepto que sea de uso interno del "Backend"

404 - Not Found:

El servidor no pudo encontrar el contenido solicitado. Este código de respuesta es uno de los más famosos dada su alta ocurrencia en la web.

500 - Internal Server Error:

El servidor ha encontrado una situación que no sabe cómo manejarla. Suele ocurrir cuando el Backend no sabe cómo responder o ocurre un error inesperado.

### 2.3.- El Header

Los HTTP headers son la parte informativa de las peticiones HTTP/S y transmiten información relacionada con el navegador del cliente, de la página solicitada, de los datos enviados en el body, del servidor, etc. En otras palabras, no son más que la manera en que el servidor y el cliente pueden reconocer los datos que están recibiendo y enviando, ya que de otra forma estarían ajenos a saber qué tipo de datos contiene la petición.

Cada vez que se hace una visita a cualquier sitio o se hace una consulta o petición a una API, en dicha petición se envían las cabeceras de petición, estas cabeceras informan acerca de qué contiene dicha petición permitiendo especificar cuál es el método HTTP, desde dónde se realizó y la URL de destino. Todo esto para que cuando cada una de estas peticiones tenga respuesta observemos los datos en la página.



## Capítulo 3. JSON

### 3.1.- Objetos

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos. Está basado en un subconjunto del Lenguaje de Programación JavaScript, en el estándar ECMASCIPT.

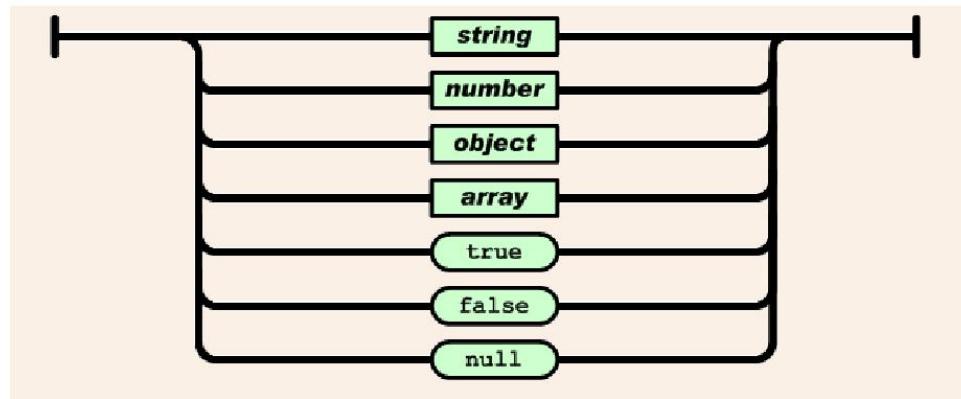
JSON es un formato de texto que es completamente independiente del lenguaje, pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C.

Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos. debido a esto, el formato JSON es usado es la mayoría de lenguajes del lado del servidor para recibir y enviar datos de manera eficaz a sus clientes. Hay que resaltar que al momento de enviar JSON entre clientes y servidor este se envía como un string y el receptor de la petición tiene que convertirlo a JSON o su cercano en el lenguaje en uso para poder manipularlo.

La estructura de un JSON se compone de una colección de pares de nombre/valor. En varios lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo. Esta es la manera en la que compone un JSON permitiendo que la mayoría de los lenguajes sean capaces de manipularlos sin tanta complejidad

Además de eso, un formato JSON permite una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o secuencias.

Los tipos de datos que permite contener un JSON son: String, Number, Boolean, Object, Array y null.



Aquí se puede observar un ejemplo de la estructura de un formato JSON, donde esta ordenado en forma de CLAVE - VALOR y donde los corchetes indican un arreglo o lista de datos.

The screenshot shows a CAD software interface with a toolbar at the top and a drawing area below. In the drawing area, there are three blue markers placed on a surface. To the right, a code editor displays the following JSON structure:

```
{  
  "markers": [  
    {  
      "name": "Rixos The Palm Dubai",  
      "position": [25.1212, 55.1535],  
    },  
    {  
      "name": "Shangri-La Hotel",  
      "location": [25.2084, 55.2719]  
    },  
    {  
      "name": "Grand Hyatt",  
      "location": [25.2285, 55.3273]  
    }  
}
```

### 3.2.- Arreglos

En el formato JSON, no solo se permiten datos de tipo string, number o booleano como se vio anteriormente, sino que hay un tipo de dato que permite agrupar una cantidad de datos de manera más eficiente, este tipo son los Arreglos o Array. Estos son expresados con corchetes ( [ ] ) y representa una lista de valores, dichos valores pueden ser string, number, boolean, objetos JSON o hasta mismos arreglos, esto con el fin de poder almacenar una cantidad de datos bastante amplia con la posibilidad de poder iterar sobre ellos para poder recorrerlos.

Dentro de un formato JSON, los Array se observan de la siguiente manera:



### 3.3.- Funciones de Php Para Json

Algunos lenguajes de programación modernos tienen incluido herramientas o manipulan los formatos JSON de manera sencilla, en el caso de JavaScript, es la forma native de definir sus objetos. En el caso de PHP, es uno de los lenguajes que no integra el formato JSON de manera directa o primitiva, ya que no existe esa estructura dentro del lenguaje. Sin embargo, incorpora una semántica parecida en cuanto al formato CLAVE - VALOR, los cuales son los arreglos asociativos, por ende, en PHP hay funciones que permiten convertir un formato JSON en un arreglo asociativo y viceversa.

### json\_decode():

Esta función convierte un string con un formato JSON en un objeto o en un array asociativo.

```
<?php  
$json = '{"a":1,"b":2,"c":3,"d":4,"e":5}';  
  
var_dump(json_decode($json));  
var_dump(json_decode($json, true));  
  
?>
```

```
object(stdClass)#1 (5) {  
    ["a"] => int(1)  
    ["b"] => int(2)  
    ["c"] => int(3)  
    ["d"] => int(4)  
    ["e"] => int(5)  
}  
  
array(5) {  
    ["a"] => int(1)  
    ["b"] => int(2)  
    ["c"] => int(3)  
    ["d"] => int(4)  
    ["e"] => int(5)  
}
```

### ACADEMIA DE SOFTWARE

Esta función hace todo lo contrario a json\_decode(). Convierte un array asociativo a un string en formato JSON. Hay que tener en cuenta que no convierte un objeto directamente a JSON, primero debe hacerse un casting para convertirlo en Array asociativo.

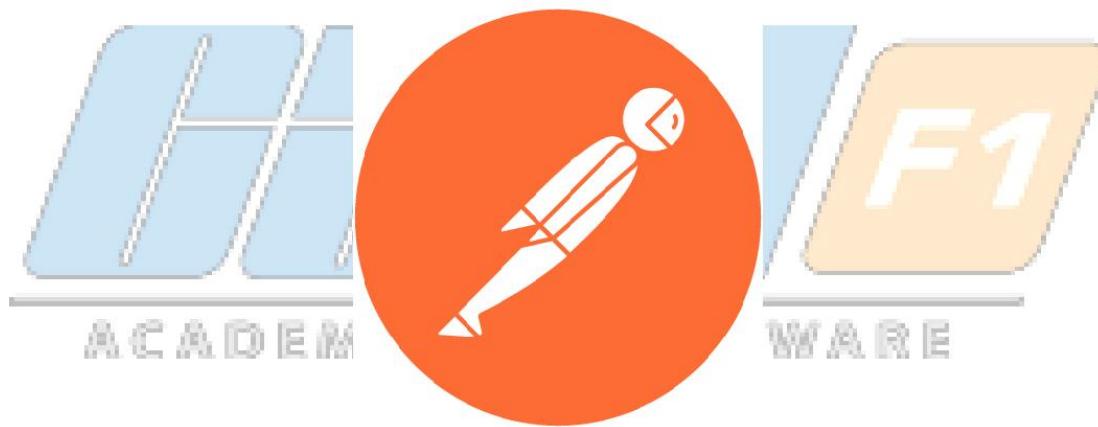
```
<?php  
$arr = array('a' => 1, 'b' => 2, 'c' => 3, 'd' => 4, 'e' => 5);  
  
echo json_encode($arr); // {"a":1,"b":2,"c":3,"d":4,"e":5}  
?>
```

## Capítulo 4. POSTMAN

### 4.1.- Conociendo Postman

Postman nace como una herramienta que permite crear peticiones sobre APIs de una forma muy sencilla y poder, de esta manera, probarlas. Todo basado en una extensión de Google Chrome.

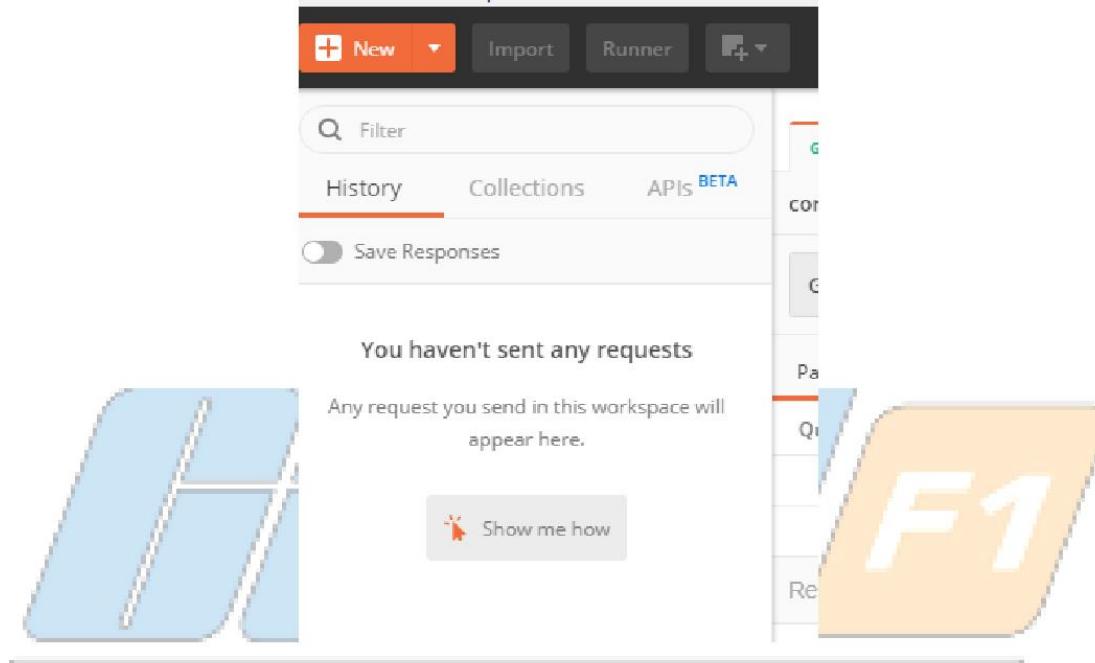
El usuario de Postman puede ser un desarrollador que esté comprobando el funcionamiento de una API para desarrollar sobre ella o un operador el cual esté realizando tareas de monitorización sobre un API.



Postman ofrece un conjunto de utilidades adicionales para poder gestionar las APIs de una forma más sencilla, como herramientas para documentar los APIs, realizar una monitorización sobre las APIs, crear equipos sobre un API para que trabajen de forma colaborativa, etc. convirtiendo a Postman plataforma de desarrollo de APIs que se basa por un modelo de desarrollo.

La interfaz de Postman está compuesta de tres partes esenciales:

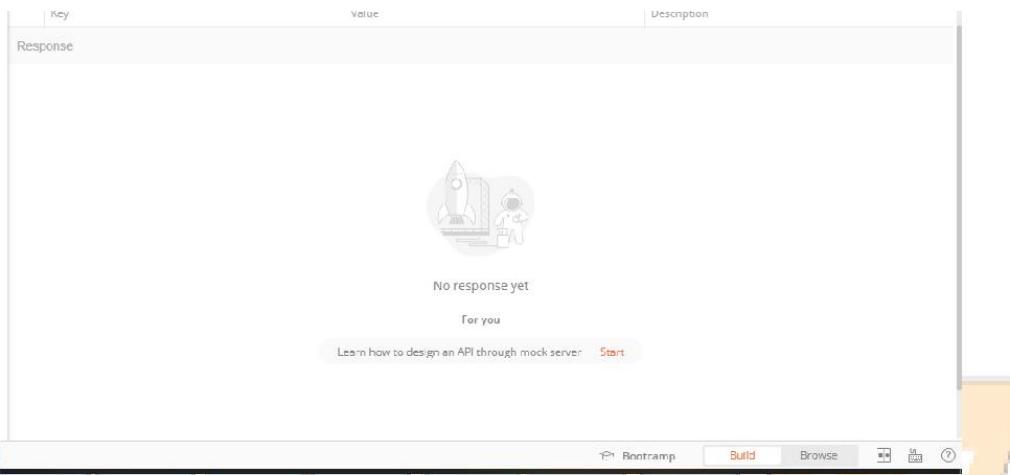
Un menú lateral izquierdo en el cual se encuentra el historial de peticiones, las colecciones, donde podremos guardar nuestras consultas en carpetas y tenerlas ordenadas.



En el centro superior, se ubica la barra de búsqueda donde se coloca la URL de la API a consultar. Adicionalmente, se encuentra un select del lado izquierdo de la barra donde se puede definir el método HTTP de la petición (GET, POST, PUT, DELETE). Del lado derecho de la barra de búsqueda está el botón SEND para enviar la petición, y abajo de esta barra se encuentre el cuadro de parámetros, donde se ubican todos los campos y parámetros necesarios para poder enviar la petición.



Por último ya al centro de la aplicación, está el cuadro de resultado (Response) donde se observa la respuesta que arroja la API consultada, en el formato en que lo devuelve, que puede ser: html, json, xml ,etc.



#### 4.2.- Creación de Peticiones

Al momento de generar alguna petición con Postman hay que tener a la mano la URL del backend a consulta, además de conocer los parámetros necesarios para poder hacer dicha petición. Lo primero a tomar en cuenta es el método de la petición, donde se tiene que seleccionar qué método se va usar; luego escribir la URL donde se va hacer la petición y por último revisar que los headers que se envían son los necesarios para la petición, este último se ubica en el Tab de headers, una vez terminado todas la configuración, se envía la petición con el botón SEND y después de unos segundo, (dependiendo de la conexión de internet) se verá la respuesta en la parte baja de la Barra de búsqueda.

The screenshot shows the Postman interface with a failed API request. The URL is `http://www.omdbapi.com/?apikey=A%26%869kmsjj={sadABNMK}`. The Headers tab shows an `Accept-Encoding` header set to `application/json`. The response status is `401 Unauthorized` with a message: `"Response": "False", "Error": "Invalid API key!"`.

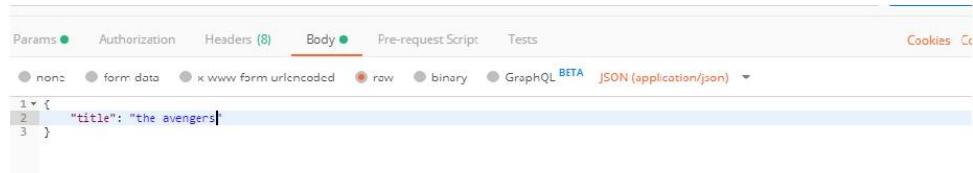
### 4.3.- Envío de Datos

Cuando se selecciona un método HTTP que contiene cuerpo (body) y se desea enviar un dato o muchos datos, Postman brinda una manera sencilla de enviar los datos sin necesidad de crear una interfaz o archivo externo para poder colocar esos datos. Para ello se ubica el botón de BODY en la barra de búsqueda. Existen varias formas de enviar los datos, dependerá de la API, para poder seleccionar el tipo correcto para enviar los datos.

Enviar los datos con un formato de Form-data:

The screenshot shows a successful API request using the `form-data` body type. The URL is `http://www.omdbapi.com/`. The Body tab shows a key-value pair: `title` with value `the avengers`. The response status is `401 Unauthorized` with a message: `"Response": "False", "Error": "Invalid API key!"`.

Enviar los datos en un formato JSON :



The screenshot shows the 'Body' tab of a Postman request configuration. The 'JSON (application/json)' radio button is selected. The JSON payload is:

```
1 ▾ {  
2   "title": "the avengers"  
3 }
```



## Capítulo 5. PHP COMO CLIENTE

### 5.1.- Curl

Cuando un cliente necesita consumir de una API para usar sus servicios, es necesario una herramienta que permita hacer las peticiones necesarias. Una librería dedicada para dicho trabajo es cURL.

cURL es una librería especializada para hacer la conexión con servidores web y comunicarse trabajar con ellos como clientes, soportando los protocolos más comunes: HTTP, FTP, HTTPS, etc.



En lo que respecta a PHP, cURL está incluido dentro del lenguaje, de manera que esta librería se puede utilizar desde cualquier script PHP. Aunque para ello la instalación de PHP tiene que incluir el soporte a cURL.

Por ende, cURL es el encargado de manejar las conexiones HTTP que se realizan en un cliente PHP y controla desde el método HTTP a usar, que headers se van a enviar y cuales datos (body) se enviarán en la petición.

Para poder usar la librería de cURL incluida en PHP hay que conocer los métodos disponibles para poder iniciar la petición. Existen 3 métodos que permiten en conjunto poder iniciar, ejecutar y cerrar la petición.

Los métodos de cURL son:

`curl_init ()`:

Permite iniciar una nueva sesión para hacer peticiones y devuelve el manipulador curl para poder hacer uso de las funciones curl\_setopt (), curl\_exec () y curl\_close ().

```
$ch = curl_init();
```

curl\_exec ():

Esta función lo que permite es poder ejecutar la petición que se configuró, y devuelve la respuesta (response) de la petición.

```
$response = curl_exec($ch);
```

curl\_close ():

Por último, para poder cerrar la sesión y no dejar abierta la petición con la configuración previa, se utiliza curl\_close (), cerrando lo que queda de petición y dando por finalizada la misma.

```
curl_close($ch);
```

## 5.2.- Configuración de Petición

Al momento de generar una petición con cURL, hay que tener en cuenta que hay que hacer una configuración previa con los datos necesarios para la API que se desee consultar, ya que cada API es distinta y necesita de datos particulares. Para poder hacer estas configuraciones, cURL incluye una función que se llama `curl_setopt ()`, esta es la encargada de configurar todos los parámetros disponibles.

Esta función necesita de tres parámetros: el manejador de sesión, que devuelve `curl_init ()`, un valor entero que define qué parámetro de configuración se va a modificar y el valor que se va enviar para poder modificarlo.

El valor entero mencionado anteriormente es el que va a permitir indicarle a la función `curl_setopt ()` que se va a modificar. PHP define una serie de constantes de dichos enteros para poder hacer referencia a ellos de una manera más sencilla.

```
curl_setopt($ch, CURLOPT_URL, "https://example.com/");
```

Algunas de las constantes pre definidas son:

- `CURLOPT_URL`: permite establecer la url de la petición.
- `CURLOPT_RETURNTRANSFER`: si se establece esta configuración, la respuesta (response) de la petición será devuelta en formato string si es posible.
- `CURLOPT_CUSTOMREQUEST`: establece el método HTTP a usar (GET, POST, DELETE, PUT, etc.), por defecto la petición siempre es GET.
- `CURLOPT_HTTPHEADER`: estos son los headers de la petición.
- `CURLOPT_POSTFIELDS`: este es el cuerpo (body) de la petición, aquí se envían los datos que necesita una petición con el método POST.
- `CURLOPT_UPLOAD`: preparar cURL para subir un archivo.

### 5.3.- Envío de Headers

Si antes de realizar una petición hay que hacer un ajuste en los headers, aparte de los generales que son enviados, cURL permite a través de sus configuraciones hacer el cambio. Para ello, en se usa la constante CURLOPT\_HTTPHEADER, el cual recibe como parámetro un array de string con los headers necesarios.

Estos headers tienen que estar compuestos por su clave, dos puntos (:) y el valor, muy parecido con la sintaxis de JSON.

```
curl_setopt($ch, CURLOPT_HTTPHEADER, array(  
    "Authorization:eyJ0eXAiOi",  
    "Content-Type:application/json"  
));
```

### 5.4.- Envío de Body

## ACADEMIA DE SOFTWARE

Cuando se realiza una petición POST, lo ideal es que se envíe un cuerpo con los datos a enviar. cURL permite crear ese body, específicamente se usa la constante CURLOPT\_POSTFIELDS para esa configuración.

Ahora, como cada API es diferente en como recibe los datos. Se debe tener la precaución de enviar los datos en el formato correcto, ya que no es lo mismo enviar los datos en formato JSON a enviarlo de otra manera distinta. Por defecto, esta configuración viene incluido con form/multipart, pero si se quiere enviar como x-www-urlencoded o JSON hay que hacer un ajuste extra a la petición.

En formato multipart/form-data:

```
curl_setopt($ch, CURLOPT_POSTFIELDS, array(  
    "empresa" => "Cadif1",  
    "curso" => "PHP",  
    "cantidad" => 4  
));
```

En formato x-www-urlencoded:

```
curl_setopt($ch, CURLOPT_POSTFIELDS, http_build_query(  
    array(  
        "empresa" => "Cadif1",  
        "curso" => "PHP",  
        "cantidad" => 4  
    )));
```

En formato JSON:

```
curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode(  
    array(  
        "empresa" => "Cadif1",  
        "curso" => "PHP",  
        "cantidad" => 4  
    )));
```

## Capítulo 6. PATRÓN FRONT CONTROLLER

### 6.1.- Mvc

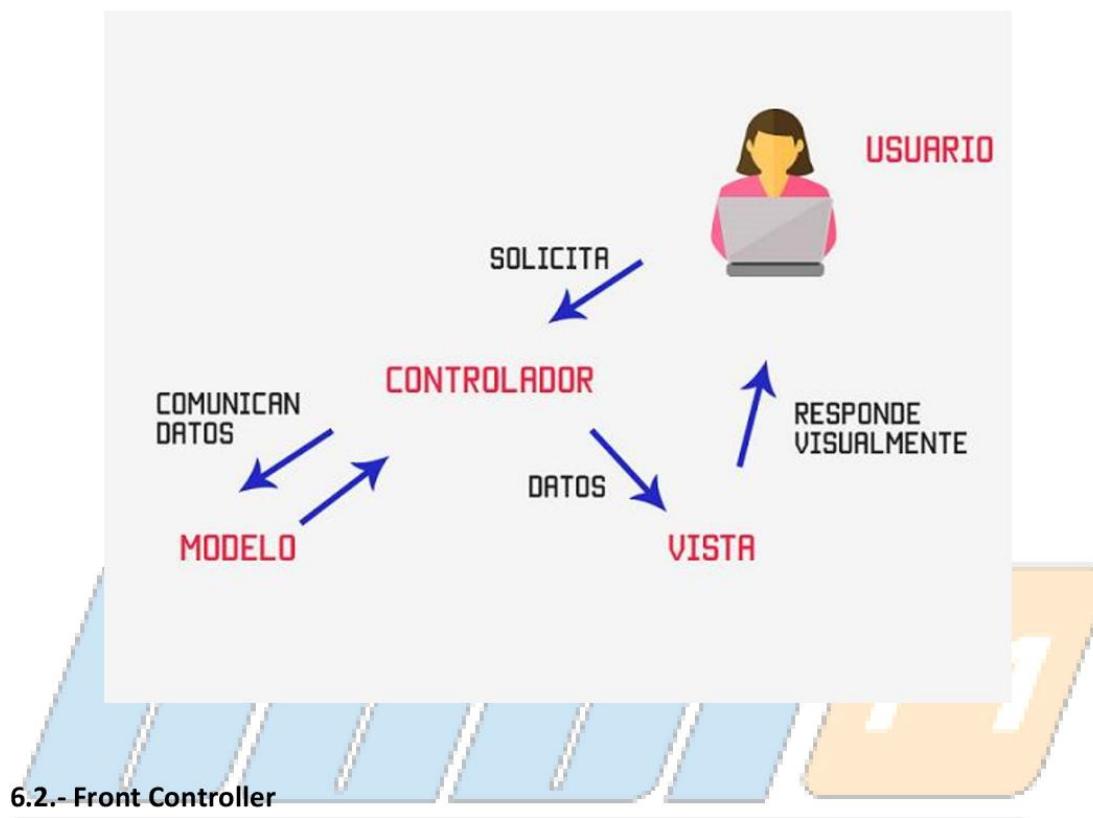
El patrón MVC o Modelo-Vista-Controlador, es un patrón de arquitectura de software, que utilizando 3 componentes (Vistas, Models y Controladores) separa la lógica de negocio de la lógica de la vista en una aplicación. Es una arquitectura importante puesto que se utiliza tanto en componentes gráficos básicos hasta sistemas empresariales. La mayoría de los frameworks modernos utilizan MVC (o alguna adaptación del MVC) para la arquitectura.

El patrón MVC considera 3 elementos fundamentales:

- **MODELO:**  
Se encarga de los datos, generalmente (pero no obligatoriamente) consultando la base de datos. Actualizaciones, consultas, búsquedas, etc. todo eso va en el modelo.
- **VISTA:**  
Son la representación visual de los datos, todo lo que tenga que ver con la interfaz gráfica. Ni el modelo ni el controlador se preocupan de cómo se verán los datos, esa responsabilidad es únicamente de la vista.
- **CONTROLADOR:**  
Se encarga de controlar, recibe las órdenes del usuario y se encarga de solicitar los datos al modelo y de comunicarlos a la vista.

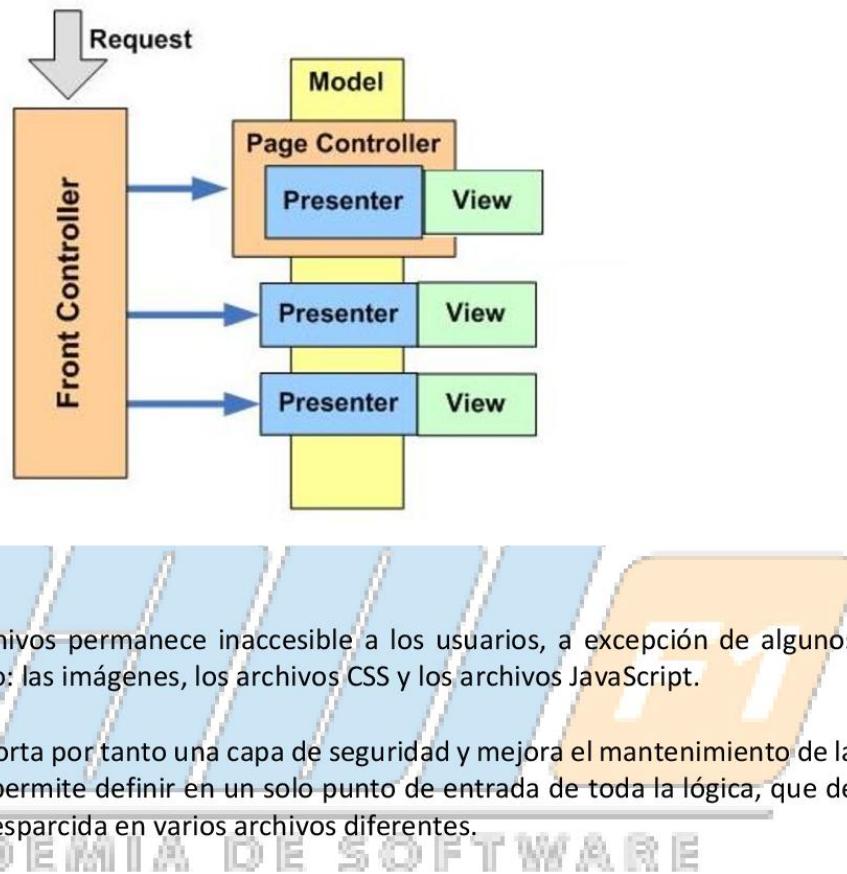
En web, el patrón MVC funciona de la siguiente manera: cuando un FrontEnd envía una petición al Backend, el controlador responde a la solicitud, ya que es el encargado de manejar la lógica de la aplicación. Una vez que el controlador nota que el usuario solicitó un recurso, le pide al modelo la información del recurso.

El modelo, que se encarga de los datos de la aplicación, consulta la base de datos y responde al controlador con los datos que pidió. Una vez el controlador tiene los datos, estos son enviados a la vista. La vista aplica los estilos, organiza la información y construye la página que se envía en el navegador.



Front Controller es un patrón de diseño de software muy utilizado en aplicaciones web, que consiste en definir un único punto de acceso para todas las peticiones HTTP.

A diferencia de las aplicaciones web clásicas, donde el usuario puede ejecutar directamente cualquier script referenciando directamente en la barra de direcciones de su navegador, con Front Controller solo se accede a un punto central único.



Este modelo está basado en el patrón MVC con la salvedad de que la forma de acceder a los controladores de cada vista es a través de un controlador principal que redirecciona a los controladores del resto de las vistas, es decir, que el patrón Front Controller se compone de Modelo, Vista, Controlador.

Ventajas:

- Se centraliza en un único punto la gestión de las peticiones.
- Permite la reusabilidad de código.
- Mejora la gestión de la seguridad.

Desventajas:

- La velocidad de respuesta disminuye al tener que ser procesadas las peticiones primero por el controlador.

### 6.3.- Modelo

Debido a que el paradigma Front controller suele estar basado en MVC, el modelo tiene la misma responsabilidad, es el encargado de llevar toda la lógica de negocio y las reglas de la aplicación. El modelo puede centrarse o tener diferentes responsabilidades, dependiendo del diseño de la aplicación, ya que en algunos diseños de frameworks el modelo se encarga solo de ser la plantilla de las tablas que existen en la base de datos, y no como un modelo de consulta, cuando es así, suele llamarse Colecciones.

Un modelo de consulta no es más que una clase o script de código, el cual se constituye de funciones específicas haciendo referencia a los métodos de CRUD (create, read, update, delete), es decir, cuando se habla de un modelo de consulta, es aquel que se constituye de todas las consultas a la base de datos de una tabla específica. Esto hace que todas las funciones que conforman este modelo se utilicen para hacer cambios y manipular la base de datos.

Un modelo llevado a Colección, es, por ejemplo, cómo funciona Eloquent en Laravel. Es un tipo de modelo que permite tener una referencia de que datos tiene una tabla dentro de una base de datos, ya que dentro de este modelo se establecen los atributos y funciones que tendrá ese objeto y que se podrán usar para este.

### 6.4.- Vista

La vista dentro del patrón de diseño MVC es la encargada de definir la forma en la que el usuario observa y manipula los datos que son consultados por el modelo y manejados por el controlador. Cuando se habla del patrón de diseño Front Controller, la forma de acceder a esas vistas es por medio del controlador principal que distribuye a el controlador dedicado.

Cuando de un servicio web se refiere, el patrón Front Controller está presente en el diseño de la misma, pero, como se ha visto con anterioridad, una API puede o no devolver una vista, es decir, que puede devolver un archivo HTML con los datos organizados o los datos puros para su uso dentro del paradigma cliente - servidor.

Por ende, Front Controller dentro del paradigma Backend-Frontend se maneja como una variante del patrón MVC, donde en vez de enviar la vista, el controlador envía los datos en bruto y no tiene que organizar nada en un HTML.



## Capítulo 7. REST

### 7.1.- Rest

En ocasiones, al momento de hablar de API, se suele presentar la posibilidad de encontrarse con el término REST, y de lo que son las API REST, a primera instancia parece un tipo distinto de API, pero no, en realidad REST solo es un extra que se le da a API.

El término REST (Representational State Transfer) se originó en el año 2000, descrito en la tesis de Roy Fielding, padre de la especificación HTTP. Un servicio REST no es una arquitectura software, sino un conjunto de restricciones con las que se puede crear un estilo de arquitectura software, la cual se puede usar para crear aplicaciones web respetando HTTP.



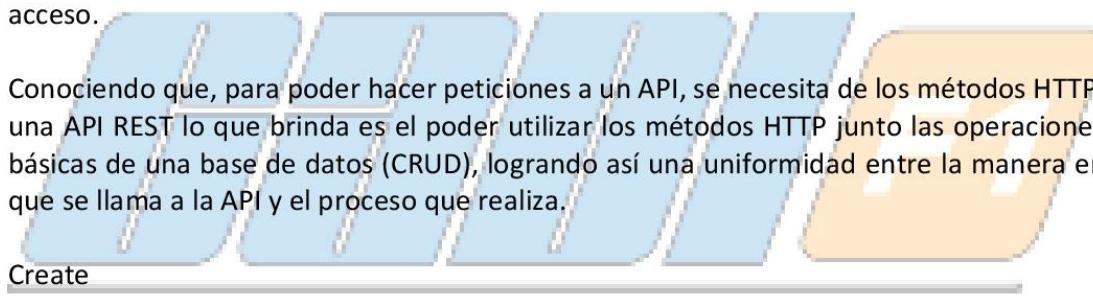
Según Fielding las restricciones que definen a un sistema REST serían:

- Cliente-servidor: esta restricción mantiene al cliente y al servidor débilmente acoplados. Esto quiere decir que el cliente no necesita conocer los detalles de implementación del servidor y el servidor se “despreocupa” de cómo son usados los datos que envía al cliente.
- Sin estado: cada petición que recibe el servidor debería ser independiente, es decir, no es necesario mantener sesiones.

- Interfaz uniforme: define una interfaz genérica para administrar cada interacción que se produzca entre el cliente y el servidor de manera uniforme, lo cual simplifica y separa la arquitectura. Esta restricción indica que cada recurso del servicio REST debe tener una única dirección, "URI".
- Sistema de capas: el servidor puede disponer de varias capas para su implementación. Esto ayuda a mejorar la escalabilidad, el rendimiento y la seguridad.

## 7.2.- Operaciones

Al utilizar las operaciones básicas de una base de datos (create, read, update, delete) como medio para cada trámite que se haga en el modelo hacia la base de datos, es decir, ELIMINAR una cuenta, AGREGAR un producto, ACTUALIZAR un precio, etc. Es necesario, que, dentro de un servicio web, estas operaciones tengan una forma coherente de acceso.



Conociendo que, para poder hacer peticiones a un API, se necesita de los métodos HTTP, una API REST lo que brinda es el poder utilizar los métodos HTTP junto las operaciones básicas de una base de datos (CRUD), logrando así una uniformidad entre la manera en que se llama a la API y el proceso que realiza.

Create

Cuando se va a crear un recurso dentro de una API el método HTTP por defecto para ello es el método POST, a través de su body, va toda la información necesaria para poder crear el recurso y guardarlo dentro del backend. Las rutas de los métodos POST deberían ser de la siguiente manera:

- POST <http://www.example.com/customers>
- POST <http://www.example.com/customers/12345/orders>

Read

Al momento de listar u obtener una serie de datos de la base de datos, hace referencia a que solo necesitamos obtener información o traernos los recursos, para ello, el método HTTP que se define para obtener algo del servidor es el método GET, a través de él, se hacen todas las peticiones donde se necesiten traer información del backend. Las rutas del método GET deberían de ser de la siguiente manera:

- GET http://www.example.com/customers/12345
- GET http://www.example.com/customers/12345/orders
- GET http://www.example.com/buckets/sample

Update:

Para actualizar un recurso dependiendo del caso puede usarse el método PUT o el método PATCH de HTTP, ya que cada uno se establece para una actualización o un tipo de actualización del recurso diferente.

En el caso de PUT, hace referencia a una actualización total del recurso, si se fuera a actualizar un registro de una base de datos, se actualizan todos los campos de ese registro.

- PUT http://www.example.com/customers/12345
- PUT http://www.example.com/customers/12345/orders/98765
- PUT http://www.example.com/buckets/secret\_stuff

El caso de PATCH es para cuando solo se va a cambiar una parte del registro, es decir, que solo va solicitar algunos campos para actualizar el registro, pero no va a actualizar todos los campos.

- PATCH http://www.example.com/customers/12345
- PATCH http://www.example.com/customers/12345/orders/98765
- PATCH http://www.example.com/buckets/secret\_stuff

Delete

El DELETE tanto como método HTTP o como operación de base de datos, da entender que su función es de eliminar, debido a que su nombre es el verbo en sí mismo, pero en este caso, se hace referencia a eliminar un recurso en el backend. Las rutas para poder eliminar un recurso deberían ser de la siguiente manera:

- DELETE http://www.example.com/customers/12345
- DELETE http://www.example.com/customers/12345/orders
- DELETE http://www.example.com/bucket/sample

### 7.3.- Rest en Php

Para poder usar PHP y establecer las reglas de una REST API, podemos usar una variable global del lenguaje llamada `$_SERVER`, dentro de esta variable tenemos todo lo relacionado con los datos recibidos de una petición que se haga a un script de PHP, esta variable está presente en cualquier script de PHP.

Algunos atributos que tiene `$_SERVER` son:

"PHP\_SELF"

El nombre del archivo de script ejecutándose actualmente, relativa al directorio raíz de documentos del servidor. Por ejemplo, el valor de `$_SERVER["PHP_SELF"]` en un script ejecutado en la dirección `http://example.com/foo/bar.php` será `/foo/bar.php`.

"REQUEST\_METHOD"

Método de petición empleado para acceder a la página, por ejemplo "GET", "HEAD", "POST", "PUT".

"REMOTE\_ADDR"

La dirección IP desde la cual está viendo la página actual el usuario.

"REQUEST\_URI"

La URI que se empleó para acceder a la página. Por ejemplo `"/index.html"`.

"PATH\_INFO"

Contiene cualquier información sobre la ruta proporcionada por el cliente. Por ejemplo, si el script actual se accede a través de la URL `http://www.example.com/php/path_info.php/some/stuff?foo=bar`, entonces `$_SERVER["PATH_INFO"]` contendrá `/some/stuff`.

## Capítulo 8. APIKEY

### 8.1.- Api Key

Las API al manejar enormes cantidades de datos, una de las principales preocupaciones es la seguridad. La idea de que los datos deben ser secretos, que no deben cambiarse y que deben estar disponibles para su manipulación es clave al momento de hablar sobre la gestión y el manejo de datos de la API.



El proceso de autenticación dentro de cualquier aplicación es el camino para demostrar la identidad de manera rápida de un usuario. Un método típico para hacerlo es a través del uso de correo/usuario y contraseña. Sin embargo, a menudo cuando se diseñan aplicaciones del lado del backend, hay que validar la identidad de los clientes que hacen peticiones, por razones de conveniencia y seguridad de los datos. Por esa razón, la forma de identificar y autorizar a dichos clientes que se conectan a la API es a través de las API Key o llave de API en español.

La API Key es un token secreto que se utiliza para identificar al cliente al momento de consumir de un servicio. Se envía junto con cada una de las solicitudes hacia el servicio

web, que por supuesto, permite el uso de dicho token, ya que no todas las API hacen uso de un token de seguridad.

Las API key son fáciles de usar, son cortas, estáticas y no caducan a menos que se revoquen, permitiendo así una manera fácil para que múltiples servicios se comuniquen de forma segura.

Otros beneficios de otorgar usar una API Key es básicamente para un mejor soporte al cliente y poder capturar los datos de uso de la aplicación para que los desarrolladores de servicios sepan qué servicios mantener o mejorar.

Al momento de enviar una API Key, esta puede estar ubicada tanto en la URL de la petición, como el header y en el caso de ser una petición a través de los métodos que permiten body, en el body de la petición.



En el siguiente ejemplo se envía el API Key en los headers de la petición con el nombre de Authorization:

ACADEMIA DE SOFTWARE

A screenshot of the Postman application interface. It shows a GET request to "http://www.music.com/". The "Headers" tab is selected, indicated by a red underline. There are two entries in the headers table: "Content-Type" with value "application/x-www-form-urlencoded" and "Authorization" with value "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6IjQ0liwibm9tYnJlIjoiVGF0b...". Other tabs like "Params", "Body", "Pre-request Script", and "Tests" are also visible.

En el siguiente ejemplo se envía el API Key, igualmente en el header de la petición, pero con el nombre de x-api-key

KEY	VALUE
<input checked="" type="checkbox"/> Content-Type	application/x-www-form-urlencoded
<input checked="" type="checkbox"/> x-api-key	9wYWlzljoIMSlkFQSV9USU1FljoxNTYxNTg0ODE5fQ.syWaxl-PxdW_U6sZ...
Key	Value

## 8.2.- Generación

No hay un estándar para la generación de una API Key. La estructura de una API Key será definida arbitrariamente por el programador, junto con el cómo va ser utilizada dentro de la aplicación. Una recomendación para generar una API Key es que esté compuesta por caracteres alfanuméricos, caracteres especiales y un patrón que permita a reconocer y validar dicha API Key.

También una API Key puede ser generada con un conjunto de datos a través de un proceso de encriptación. Los procesos de encriptación para que esa API Key sea única se logra a través de los métodos md5() y sha1().

Siempre considerando que las API Key en sí misma es una forma de identidad, que permite identificar la aplicación o el usuario, por ende, debe ser única, aleatoria y no adivinable.

En el caso de que se quiera integrar un API Key en una aplicación, esta debe ser guardada en la base de datos para poder tener constancia de ella y a cuál usuario o cliente está asociada. En la siguiente imagen se muestra un ejemplo de una API Key de un sitio web:

Your API key for [info@apievangelist.com](mailto:info@apievangelist.com) is:

**QYC8QbKSVr6sSlnJOJSZbx8eUazwed5s5FRPw4KC**

Otros ejemplos de API Key pueden ser:

- zaCELgLoi.mfnc8mVLWwsAawjYr4Rx-Af50DDqtlx.
- 2823 - asd5 - r8Fe - 23ER
- hGMICuy)BG6d3LRWU?sa58

Hay que tener en cuenta que no siempre una API Key va a tener este aspecto, ya que como se mencionó con anterioridad, pueden estar compuesta de: guiones, puntos, caracteres especiales, pueden estar divididas y tener datos distintos, todo depende de qué manera se quiere armar la API Key para poder generarlas, tomando en cuenta que mientras más seguro sea, mejor.

### 8.3.- Verificación

Dado que la API Key proporciona acceso a los datos, se comporta como una contraseña que proporciona un usuario de una aplicación para obtener acceso a datos privados, por ello, validar cada petición es fundamental.

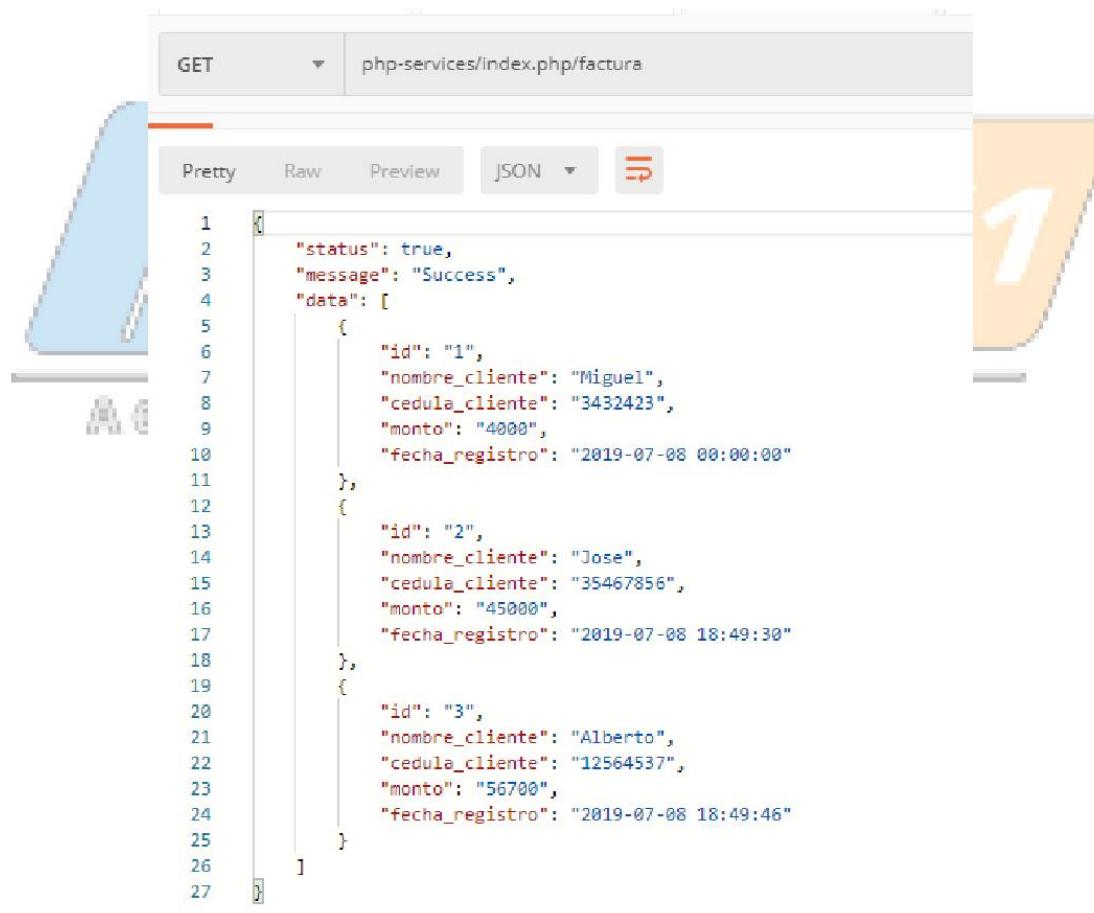
La forma de validación dependerá de la manera en que el desarrollador de la API Key la genere, de esta manera, solo la aplicación puede ser capaz de interpretarla, logrando una capa extra de seguridad.

Lo más recomendable, es que debido a que la API Key debería de estar almacenada en la base de datos, esta sea validada comparándola con los registros de una tabla donde se almacenen las API Key existentes, en el caso de que sea válida, se le otorga el paso al cliente al servicio, en caso contrario, se le envía un error de autorización al cliente.

## Capítulo 9. GET. PARTE 1

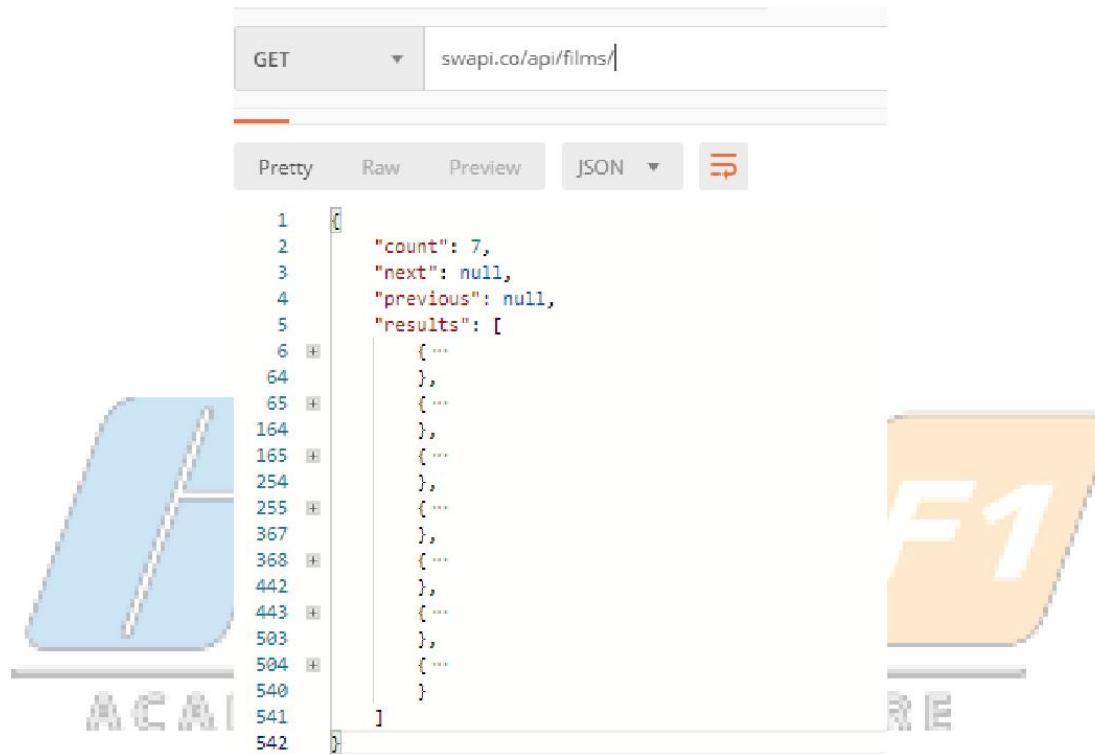
### 9.1.- Todos Los Recursos

Como se observó en capítulos anteriores, el método GET está relacionado dentro de una API Rest con el método READ de las operaciones de base de datos, por ende, al momento de hacer una petición de todos los registros asociados a un recurso en una API, tras cámaras se ejecuta un SELECT a la tabla donde se almacenan los registros solicitados para devolverlos al cliente. En el siguiente ejemplo se solicita todos los registros del servicio Factura:



```
1  {
2      "status": true,
3      "message": "Success",
4      "data": [
5          {
6              "id": "1",
7              "nombre_cliente": "Miguel",
8              "cedula_cliente": "3432423",
9              "monto": "4000",
10             "fecha_registro": "2019-07-08 00:00:00"
11         },
12         {
13             "id": "2",
14             "nombre_cliente": "Jose",
15             "cedula_cliente": "35467856",
16             "monto": "45000",
17             "fecha_registro": "2019-07-08 18:49:30"
18         },
19         {
20             "id": "3",
21             "nombre_cliente": "Alberto",
22             "cedula_cliente": "12564537",
23             "monto": "56700",
24             "fecha_registro": "2019-07-08 18:49:46"
25         }
26     ]
27 }
```

En el siguiente ejemplo se está haciendo una petición al servicio de Films de la API de Star Wars, obteniendo todos los resultados:



The screenshot shows a browser developer tools Network tab. A GET request is made to `swapi.co/api/films/`. The response is displayed in JSON format under the "JSON" tab. The JSON object contains the following structure:

```
1  [
2   "count": 7,
3   "next": null,
4   "previous": null,
5   "results": [
6     {
7       ...
8     },
9     {
10    ...
11  },
12  {
13    ...
14  },
15  {
16    ...
17  },
18  {
19    ...
20  },
21  {
22    ...
23  },
24  {
25    ...
26  },
27  {
28    ...
29  },
30  {
31    ...
32  },
33  {
34    ...
35  },
36  {
37    ...
38  },
39  {
40    ...
41  },
42  {
43    ...
44  },
45  {
46    ...
47  },
48  {
49    ...
50  },
51  {
52    ...
53  },
54  {
55    ...
56  },
57  {
58    ...
59  },
59  {
60    ...
61  }
62 ]
```

La manera de generar una respuesta desde una API propia con todos los recursos disponibles, sería la siguiente:

```
public function GET() // CONTROLADOR
{
    header('Content-Type:application/json');
    echo (json_encode($this->cuenta->getCuentas()));
}

public function getCuentas() // MODELO
{
    $query = $this->database
        ->query(
            "SELECT * FROM cuenta",
            PDO::FETCH_ASSOC
        );
    return $this->build($query);
}
```

## 9.2.- Query Params

En ocasiones, es necesario enviar de alguna manera datos a la API por el método GET. El problema está en que este método no contiene BODY, por ello, para enviar parámetros a una API REST se utilizan los Query Params. Query Params es una serie de datos formados por clave-valor que se agregan al final de la URL, justo después del signo de interrogación (?).

En la siguiente URL para consultar los clientes de una API, se envía por query params el dato "óscar" (el valor) identificado bajo el nombre "name" (la clave):

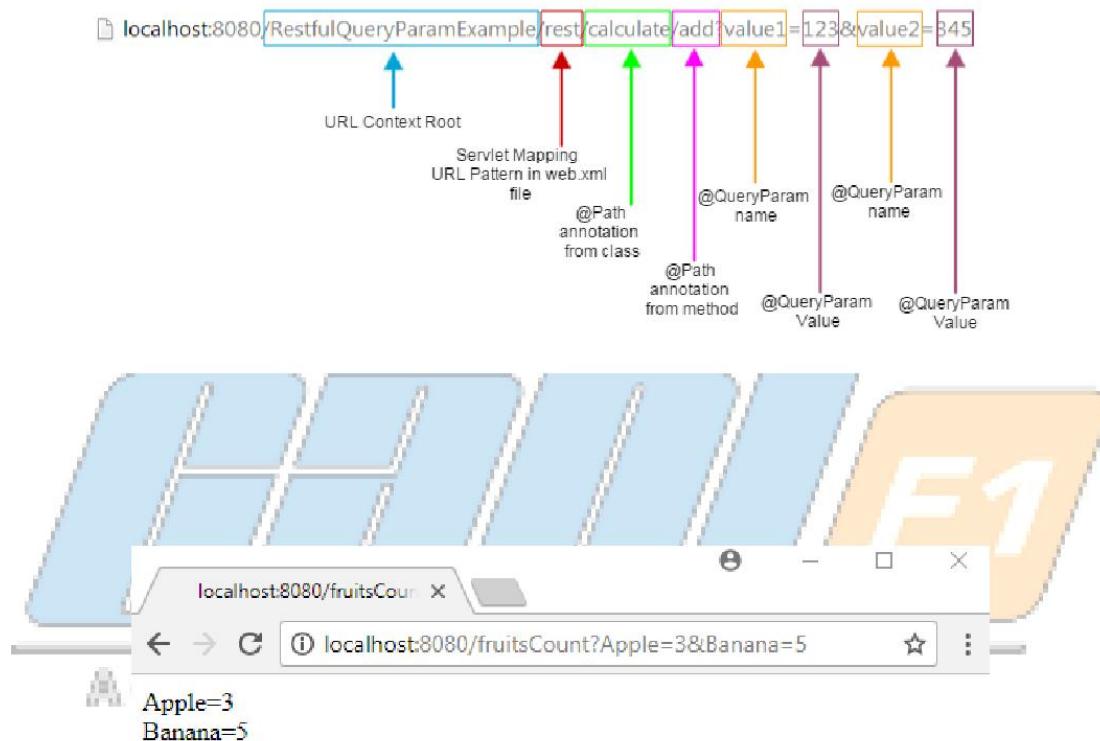
<http://myapi.com/customers?name=oscar>

Una URL puede tener N query params, como el siguiente ejemplo:

<http://myapi.com/customers?firstname=oscar&lastname=blancarte&status=active>

Esta URL se utiliza para buscar a todos los clientes donde su "name" es óscar, su "lastname" es blancarte y su "estatus" es activo. Cuando se utiliza más de un Query param, es importante separar cada uno mediante el símbolo &.

Esta es la estructura de los Query Params y como se componen dentro las reglas de una API Rest:



### 9.3.- Recursos Específicos

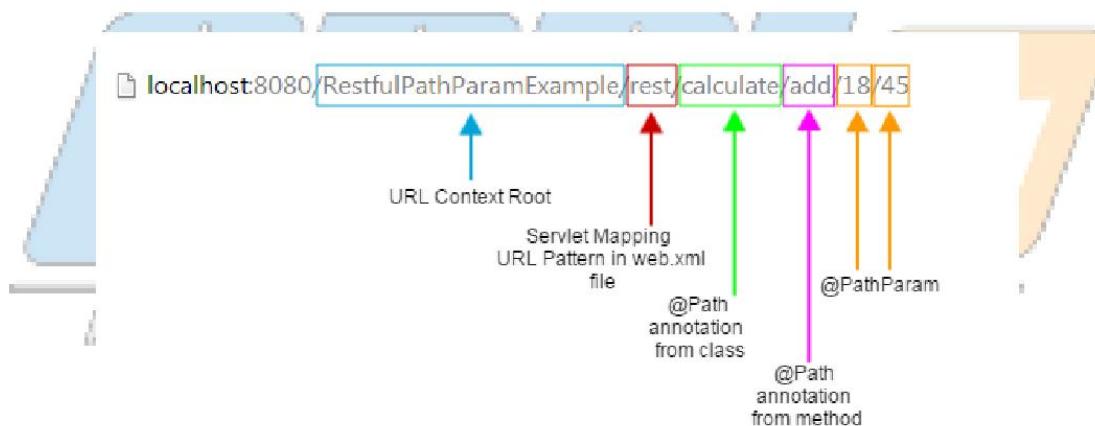
En ocasiones, se requiere obtener de una API un recurso específico. Esto, debido a que la petición de todos los recursos puede llegar a pesar mucho, ocasionando una respuesta

más lenta, sin contar que una vez que la respuesta llegue, hay que recorrer los datos para buscar el recurso individual.

Por esta razón, en una API debe existir un servicio que permita obtener un recurso de forma individual, sin necesidad de solicitar todos los recursos, otorgando mejoras en la respuesta y en el procesamiento del lado del cliente.

Cuando se llama a dicho servicio, se tiene que enviar un identificador, el cual permite obtener el recurso y devolverlo en la respuesta.

Hay diferentes maneras de enviar este identificador a la API para que pueda ser usada, dependerá de cómo quiera ser manejada la petición por parte de los desarrolladores de la API. Las formas más comunes de enviar este identificador es por la URL o por el header.



Se obtiene del servicio de Planets de la API de Star Wars, un planeta específico, en este caso el planeta número 3:

```
1 [ {  
2   "name": "Yavin IV",  
3   "rotation_period": "24",  
4   "orbital_period": "4818",  
5   "diameter": "10200",  
6   "climate": "temperate, tropical",  
7   "gravity": "1 standard",  
8   "terrain": "jungle, rainforests",  
9   "surface_water": "8",  
10  "population": "1000",  
11  "residents": [],  
12  "films": [  
13    |   "https://swapi.co/api/films/1/"  
14  ],  
15  "created": "2014-12-10T11:37:19.144000Z",  
16  "edited": "2014-12-20T20:58:18.421000Z",  
17  "url": "https://swapi.co/api/planets/3/"  
18 } ]
```

En el siguiente ejemplo se obtiene el recurso con el identificador 1 del servicio de Cuenta:

## ACADEMIA DE SOFTWARE

```
1 [ {  
2   "id": "1",  
3   "nombre": "admin",  
4   "username": "root"  
5 } ]
```

Llevando el concepto de recurso individual a PHP seria de la siguiente manera:

```
public function GET() // CONTROLADOR
{
    header('Content-Type:application/json');
    if (isset($_REQUEST['id'])) {
        echo (json_encode(
            $this->cuenta->getCuenta($_REQUEST['id'])
        ));
        exit();
    }
    echo json_decode(false);
}

public function getCuenta($ID) // MODELO
{
    $query = $this->database
        ->query(
            "SELECT * FROM cuenta WHERE id='$ID'",
            PDO::FETCH_ASSOC
        );
    return $query->rowCount() == 0 ? null : $this->build($query)[0];
}
```

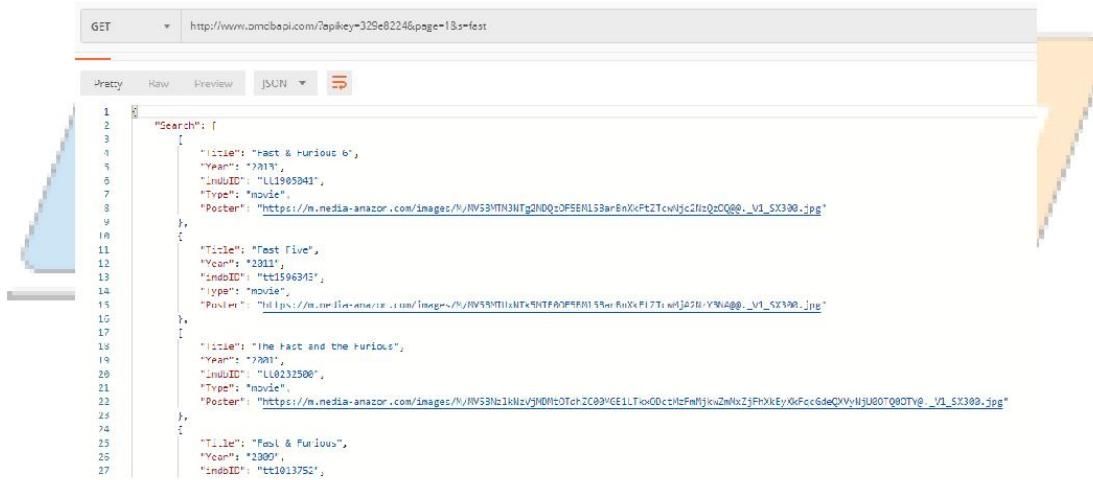
ACADEMIA DE SOFTWARE

## Capítulo 10. GET. PARTE 2

### 10.1.- Paginación

La paginación dentro de una petición GET a una API no es más que una forma de devolver los datos con un límite, es decir, que pueda acceder a todos los registros, pero con un límite por consulta, permitiendo así poder conformar varias “paginas” para hacer la consulta, dando así un conjunto de peticiones para no recargar de recursos al cliente.

En este ejemplo se observa que se está enviando un parámetro en la URL que especifica que página se desea obtener, en este caso la pagina 1:



```
1  "Search": [
2    {
3      "Title": "Fast & Furious 6",
4      "Year": "2013",
5      "ImdbID": "tt1905941",
6      "Type": "movie",
7      "Poster": "https://m.media-amazon.com/images/M/MV5BMTIwNTg2NDQzOF5ERl33arBnXcPt2TcwNjczQzQG@._V1_SX300.jpg"
8    },
9    {
10      "Title": "Fast Five",
11      "Year": "2011",
12      "ImdbID": "tt1596343",
13      "Type": "movie",
14      "Poster": "https://m.media-amazon.com/images/M/MV5BMTIwNTg2NDQzOF5ERl33arBnXcPt2TcwNjczQzQG@._V1_SX300.jpg"
15    },
16    {
17      "Title": "The Fast and the Furious",
18      "Year": "2001",
19      "ImdbID": "tt0232300",
20      "Type": "movie",
21      "Poster": "https://m.media-amazon.com/images/M/MV5BMTk1kNzjN0MtOTchZC09GE1LToiC0ctMzFnMjkwZjFhkEyXkFccGdeQVYyNjU807Q0OTVg._V1_SX300.jpg"
22    },
23    {
24      "Title": "Fast & Furious",
25      "Year": "2001",
26      "ImdbID": "tt1013752",
27    }
]
```

En continuación a la petición anterior, en el caso de que se desee acceder a las demás páginas del servicio de Películas, se cambia el parámetro Page de "1" a "2"

GET http://www.omdbapi.com/?apikey=329c8224&page=2&s=fast

Pretty Raw Preview JSON

```
1 [ { "Search": [ 2 { "Title": "The Fast Show", 3 "Year": "1994-2014", 4 "imdbID": "tt0108771", 5 "Type": "series", 6 "Poster": "https://m.media-amazon.com/images/M/MV5BMjI3ODA1MjI5M15BnBnXkFtZTcwNjE0ODYyM0@.V1_SX300.jpg" 7 }, 8 { "Title": "Fast, Cheap & Out of Control", 9 "Year": "1997", 10 "imdbID": "tt0119107", 11 "Type": "movie", 12 "Poster": "https://m.media-amazon.com/images/M/MV5BMjEyMTQ4OTg3NF5BN15BnBnXkFtZTcwNzE3NDIyM0@.V1_SX300.jpg" 13 }, 14 { "Title": "Fast Girls", 15 "Year": "2012", 16 "imdbID": "tt1700803", 17 "Type": "movie", 18 "Poster": "https://m.media-amazon.com/images/M/MV5BNjEzY2NzAyN15BnBnXkFtZTcwNzI3NDM5N@.V1_SX300.jpg" 19 }, 20 { "Title": "Angels in Fast Motion", 21 "Year": "2005", 22 "imdbID": "tt0403369", 23 }, 24 { "Title": "The Fast and the Furious", 25 "Year": "2001", 26 "imdbID": "tt0182522", 27 } ] }
```

Para representar un GET de paginación dentro del contexto de PHP sería de la siguiente manera:



ACADEMIA DE SOFTWARE

```
public function GET() // CONTROLADOR
{
    header('Content-Type:application/json');
    if (isset($_REQUEST['pag'])) {
        echo (json_encode(
            $this->cuenta->getCuentaPag($_REQUEST['pag'] . '0')
        ));
        exit();
    }
    echo json_decode(false);
}

public function getCuentaPag($LIMIT) // MODELO
{
    $query = $this->database
        ->query(
            "SELECT * FROM cuenta LIMIT $LIMIT",
            PDO::FETCH_ASSOC
        );
    return $this->build($query);
}
```

#### 10.2.- Filtro

Cuando se necesita obtener una cantidad de recursos a través de algún parámetro específico como el nombre de un cliente, el país de una empresa o la ciudad de destino de un vuelo, es necesario filtrar según sea el parámetro, todos los recursos disponibles en una API y que estos sean devueltos en la respuesta al cliente. Para ello es necesario de los Query param.

En este ejemplo se puede observar cómo se envía un parámetro de filtrado, donde se especifica que la búsqueda será por el nombre de la película, y el valor de dicho parámetro es "avenger":

```

GET http://www.omdbapi.com/?apikey=329e8224&i=t0108771
Pretty Raw Preview JSON
1 [
2   "Search": [
3     {
4       "Title": "Captain America: The First Avenger",
5       "Year": "2011",
6       "imdbID": "tt1408839",
7       "Type": "movie",
8       "Poster": "https://m.media-amazon.com/images/M/MV5BMTYzOTkxMjUyNzIwMjQwMjIwXkFtZTlzb1Y0NDQ0QzEzV1_SX300.jpg"
9     },
10    {
11      "Title": "The Toxic Avenger",
12      "Year": "1984",
13      "imdbID": "tt0086100",
14      "Type": "movie",
15      "Poster": "https://m.media-amazon.com/images/M/MV5BNTkxMjUyNzIwMjQwMjIwXkFtZTlzb1Y0NDQ0QzEzV1_SX300.jpg"
16    },
17    {
18      "Title": "The Toxic Avenger Part II",
19      "Year": "1988",
20      "imdbID": "tt0086101",
21      "Type": "movie",
22      "Poster": "https://m.media-amazon.com/images/M/MV5BNTkxMjUyNzIwMjQwMjIwXkFtZTlzb1Y0NDQ0QzEzV1_SX300.jpg"
23    },
24    {
25      "Title": "Citizen Toxie: The Toxic Avenger IV",
26      "Year": "1996",
27      "imdbID": "tt0112849"
28    }
29  ]
30 ]

```

En esta petición se hace un filtrado por el ID de la película:

```

GET http://www.omdbapi.com/?apikey=329e8224&i=t0108771
Pretty Raw Preview JSON
1 [
2   {
3     "Title": "The Fast Show",
4     "Year": "1994-2014",
5     "Rated": "TV-14",
6     "Released": "30 Mar 1998",
7     "Runtime": "30 min",
8     "Genre": "Comedy",
9     "Director": "N/A",
10    "Writer": "N/A",
11    "Actors": "Paul Whitehouse, Charlie Higson, Arabella Weir, Simon Day",
12    "Plot": "UK comedy sketch show depicting most forms of stereotypical mid-90's British society.",
13    "Language": "English",
14    "Country": "UK",
15    "Awards": "6 wins & 2 nominations.",
16    "Poster": "https://m.media-amazon.com/images/M/MV5BMTI3ODA1MjI5M15BM15BnBnXkFtZTcwNjE0ODYyMQ@@._V1_SX300.jpg",
17    "Ratings": [
18      {
19        "Source": "Internet Movie Database",
20        "Value": "8.2/10"
21      }
22    ],
23    "Metascore": "N/A",
24    "imdbRating": "8.2",
25    "imdbVotes": "3,610",
26    "imdbID": "tt0108771",
27    "Type": "series",
28    "totalSeasons": "2"
29  }
30 ]

```

Ejemplo de cómo sería la consulta desde PHP para hacer un GET con filtrado de resultados:

```
public function GET() // CONTROLADOR
{
    header('Content-Type:application/json');
    if (isset($_REQUEST['nombre'])) {
        echo json_encode(
            $this->cuenta->getCuentaByNombre($_REQUEST['nombre'])
        );
        exit();
    }
    echo json_decode(false);
}

public function getCuentaByNombre($nombre) // MODELO
{
    $query = $this->database
        ->query(
            "SELECT * FROM cuenta WHERE nombre = '$nombre'",
            PDO::FETCH_ASSOC
        );
    return $this->build($query);
}
```

ACADEMIA DE SOFTWARE

## Capítulo 11. POST. PARTE 1

### 11.1.- Recibiendo Los Datos (request)

Cuando un cliente desea crear un recurso dentro de una API, esta llama al método POST del servicio necesario, el cual será capaz de crear el recurso. Para ello es necesario que el cliente envíe la información del recurso dentro del body.

Cuando se reciben datos provenientes de un cliente en una API, esta tiene que establecer específicamente un tipo de formato de recepción de los datos, para mantener la uniformidad, es decir, que, al momento de recibir los datos de un cliente, el cliente enviará los datos en un formato, para ser interpretado por la API.

Una API puede soportar varios tipos de formato, pero lo ideal no es que abarque todos, sino, que sea lo más uniforme posible, que pueda aceptar en cada uno de sus servicios un formato por defecto, el que permite, varios formatos, puede dar la ventaja para algunos clientes, pero no que unos métodos sea de un formato y otros métodos de otro, eso sería un error grave.

Los formatos más usados son form-data, x-www-urlencoded y JSON, para poder obtener estos formatos dentro de Php se necesita hacer unos pasos intermedios para obtenerlos.

Para poder obtener los datos enviados a una API a través del formato "Form-data", solo hay que hacer uso de la variable global "\$\_POST"

```
$contentType = $_SERVER['CONTENT_TYPE'];
if (preg_match('/multipart/form-data/', $contentType))
    $data = $_POST;
```

En el caso de que se desee obtener los datos en el formato "x-www-encoded", es necesario usar una función de PHP, llamada "file\_get\_contents ()". Esta permite obtener el string que es enviado desde el cliente en este formato.

```
$contentType = $_SERVER['CONTENT_TYPE'];
if (preg_match('/application/x-www-form-urlencoded/', $contentType))
    parse_str(file_get_contents("php://input"), $data);
```

Para recibir el formato JSON, se utiliza de igual forma la función "file\_get\_contents ()", con la excepción de que se tiene que limpiar los espacios en blanco con la función "trim ()" y luego decodificar los datos para volverlo un array asociativo con las funciones de JSON en PHP.

```
$contentType = $_SERVER['CONTENT_TYPE'];
if (preg_match('/application/json/', $contentType))
    $data = json_decode(trim(file_get_contents("php://input")), true);
```

## 11.2.- Procesamiento

Con los datos ya ubicados y obtenidos del formato correspondiente, el siguiente paso es el procesamiento de la información. Consiste en manipular la información, para que pueda ser guardada en un lugar específico, por ejemplo, los datos de un usuario, los cuales se guardan en la base de datos para poder usarlo más tarde.

Hay que tomar en cuenta que el procesamiento de los datos no implica necesariamente el guardar en la base de datos, puede ser un servicio que necesite de información valiosa por el body y esta sea procesada para que sea utilizada en una app.

```
$DATA = $_POST;

if(isset($DATA['user']) && isset($DATA['name']) && isset($DATA['ID'])){
    $isSave = Model::CreateUser($DATA['user'], $DATA['name'], $DATA['ID']);

    Response::OK($isSave, 200);
}
```

### 11.3.- Respuesta (response)

Al momento que una API responde a la petición que se le solicitó, la forma de respuesta dependerá de cómo está estructurada, no hay un patrón definido de cómo tiene que estar compuesto un objeto JSON dentro de una respuesta de una API, pero si hay consideraciones y buenas prácticas que permiten tener un mejor manejo de mensajes de errores y peticiones completas dentro de la API.

La primera consideración a tomar en cuenta, es que se tiene que tener un uso correcto de los códigos de error. Es importante que el usuario o aplicación que reciba los datos, comprenda que ha sucedido, no solo por la respuesta en sí, sino por algo mucho más básico, como los códigos de error.

Una muy buena práctica si la respuesta viene en formato JSON, es que contenga un atributo booleano que indique si fue exitosa o no la operación, es decir que vengan o no datos dentro de la respuesta, y un segundo atributo que explique de forma humana que sucedió, este atributo suele ser llamado mensaje, y es donde si ocurre un error, poder identificar qué fue lo que pasó. Por supuesto, esta respuesta tiene que ser coherente con el código de error.



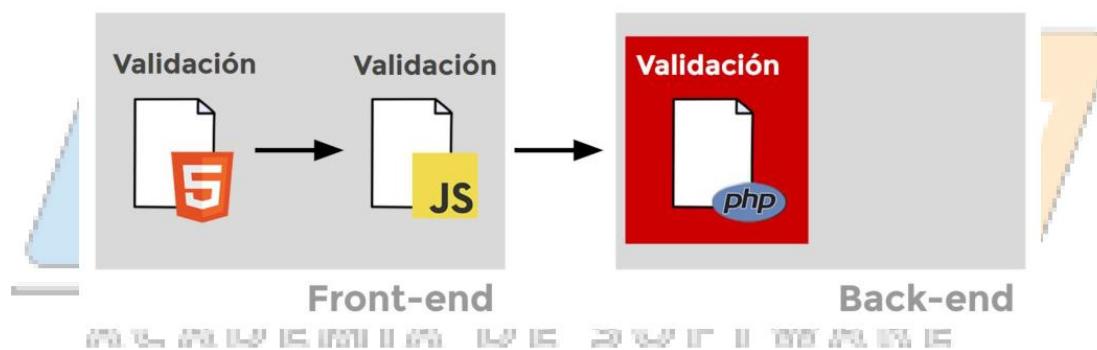
Otro aspecto a tomar en cuenta, es la coherencia de datos, que se quiere decir con esto, que si una petición es para obtener todos los datos, una respuesta, tal vez esperada es que los datos estuvieran en un array, porque son un conjunto de datos, a diferencia que lo hiciera si se estuviera solicitando un registro único, en ese caso sería un poco incoherente que viniera dentro de un array, cuando no es un conjunto de datos sino uno solo, ese tipo de errores pueden llegar a confundir en la manera que un usuario interpreta la respuesta.

## Capítulo 12. POST. PARTE 2

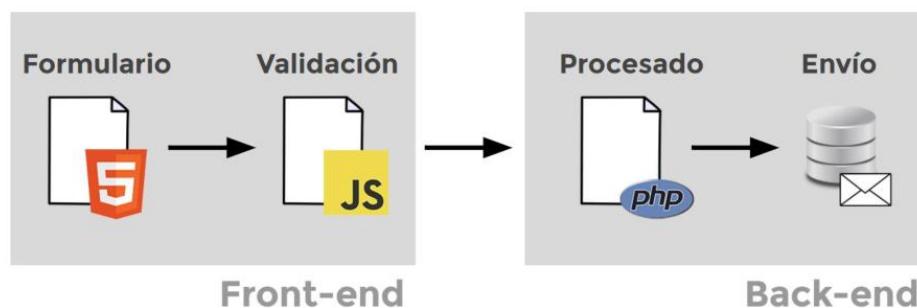
### 12.1.- Validaciones

Al momento de procesar los datos que se reciben dentro de una API Rest, hay que considerar un aspecto muy importante. En ocasiones, los datos que envía un usuario o cliente no son completamente correctos o no son enviados, por ello, las validaciones en los datos recibidos es la primera línea de seguridad ante cualquier dato no válido o un error en el backend.

La petición llega del cliente y la API valida los resultados:



Si la validación es correcta se procesan los datos y se sigue con la ejecución del controller:



Considerando que al momento de utilizar el método POST se tiene que guardar uno o varios registros por petición, hay que comprobar que los datos enviados en la petición están completamente correctos para guardarlo en la base de datos, de otra manera, si no estuvieran correctos se pudiera generar alguna inconsistencia en la aplicación, algún ataque a la API o hasta un error de clave foránea de alguna columna de la tabla. He de ahí la importancia de validar los datos antes de guardarlo o manipularlos dentro de Backend.

Hay que tener presente que no todo lo que envíe el cliente es perfecto, y el hecho de validar no es un proceso en vano. En ocasiones, los clientes no hacen las validaciones necesarias y por ello el backend es la última línea de defensa antes de que envíen información no correcta a la API.

Existen dos tipos de Validaciones, las validaciones de estructura y las validaciones de datos.

### **12.2.- Validación de Estructura**

Cuando en una base de datos, o reglas de negocio se establece que una contraseña no puede tener menos de 8 caracteres, en ese momento, se está definiendo una validación, una validación que tiene que estar al recibir el dato, porque en caso contrario, si es menor a 8, simplemente va a arrojar un error la base de datos y por ende un error al cliente de parte del backend. Estas validaciones son las validaciones de estructura.

En pocas palabras, las validaciones de estructura son todas aquellas validaciones que hacen referencia a cómo debería ser la estructura del dato recibido, para que, cuando un usuario mande dicho dato de manera errónea, se le devuelva un error con la información necesaria para poder ajustar sus datos.

### **12.3.- Validación de Dato**

Además de validar si el dato recibido tiene una estructura esperada, puede nacer otro tipo de problemas. En el caso de que el dato que se esté solicitando es el ID de una tabla a la cual queremos hacer referencia y el usuario por error envía un ID que no se encuentra en la base de datos, esto causaría un problema con la base de datos, ocasionando una inconsistencia en los datos.

Para ello, además de tener que validar que, si el dato cumple las reglas de su estructura, también tiene que pasar por las validaciones que permitan identificar que el dato enviado sea válido igualmente dentro de la base de datos.

Estas validaciones no son más que una consulta a la base de datos, es decir, en el caso de validar un ID de un registro para saber si existe o no, hay que hacer una consulta a la base de datos para saber esa información, en el caso de que sea verdadero, pasará la validación y continuará con el proceso, en caso contrario, enviará un error al usuario mostrándole dónde está el error y por qué.



## Capítulo 13. PUT Y DELETE

### 13.1.- Put

El método PUT, como se vio en clases anteriores, es el método de actualización por defecto según las reglas establecidas por Rest, por ende, para poder actualizar un recurso, hay que hacerlo a través de él.

El proceso de actualización de un recurso, es muy parecido a el método de creación (POST), debido a que, ambos métodos permiten el envío del body, es decir contener datos en la petición. Que el método PUT permita body, trae como consecuencia que se tenga que validar cada uno del parámetro que se envían en la petición, ya que, al igual que en método POST, un error en los datos y puede comprometer la base de datos.

La manera en que se compone la URL en el método PUT es muy parecida a un GET de un recurso individual, hay que enviarle a través de los parámetros en URL el ID del recurso que se desea modificar, mientras que en body estará toda la información a actualizar.

```
PUT https://localhost/user/52  
PUT https://localhost/categoria/tipo/52  
PUT https://localhost/mode/complete/8  
PUT https://localhost/image/perfil.png
```

De igual manera, una vez que han recibido los datos, validados y procesados, hay que devolver una respuesta de éxito o error al cliente para que esté informado de que la operación fue un éxito o no.

En PHP no existe una variable global `$_PUT` que permita de manera sencilla obtener dichos datos. la manera de obtener el body de la petición es a través de `file_get_content()`, el mismo que usa en el método POST, para poder obtener los datos en formato x-www-urlencoded y JSON. Esto quiere decir, que estos son los únicos dos formatos que son soportados en php para poder actualizar un dato.

### 13.2.- Delete

El método DELETE es el encargado de eliminar un recurso, como se vio con anterioridad, su método en las operaciones de base de datos se llama exactamente igual, pero es el método que hay que manipular con mucho cuidado, debido a que es el que puede tener repercusiones en nuestra base de datos si no se valida de manera correcta.

La operación DELETE, dentro de una base de datos es una operación muy restringida, ya que un error en la operación y podemos perder los registros en ella. En una API donde el eliminar puede estar al alcance de los clientes que consumen de ese servicio, por ello las validaciones son fundamentales.

La manera en la que se dispone la URL para poder eliminar un recurso es igual a la de un GET de un recurso individual o un PUT, pasando por los parámetros de la URL el identificador del recurso, y solo con eso ya será suficiente para eliminar el recurso.

En el caso de enviarle más parámetros para especificar ajustes extra, se tendría que usar los Query Params para poder hacer la configuración extra.

```
DELETE https://localhost/user/85  
DELETE https://localhost/categoria/tipo/1?content=all  
DELETE https://localhost/mode/complete/9  
DELETE https://localhost/image/logo.png
```

## Capítulo 14. JWT. PARTE 1

### 14.1.- Jwt

Json Web Token es un conjunto de datos cifrados con una clave de seguridad para enviar información por peticiones http. Esta forma de enviar datos está pensada principalmente para ser transferidos entre dos partes (cliente y servidor). Las partes de un JWT se codifican como un objeto JSON que está firmado digitalmente.

El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario que se desea autenticar, manda sus datos de inicio de la sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, luego en cada petición realizada al servidor, el cliente envía este token que el servidor usa para verificar que el usuario esté correctamente autenticado y saber quién es.

Este no es el único caso de uso para los JWT, es posible usarlo para transferir cualquier dato entre servicios de nuestra aplicación y asegurarnos de que sean siempre válido. Por ejemplo, si tenemos un servicio de envío de email, se podría enviar una petición con un JWT junto al contenido del correo, permitiendo así que se esté seguro que esos datos no fueron alterados de ninguna forma.

También nos añade más seguridad. Al no utilizar cookies para almacenar la información del usuario, podemos evitar ataques informáticos que manipulen la sesión que se envía al backend. Por supuesto podemos hacer que el token expire después de un tiempo lo que le añade una capa extra de seguridad.

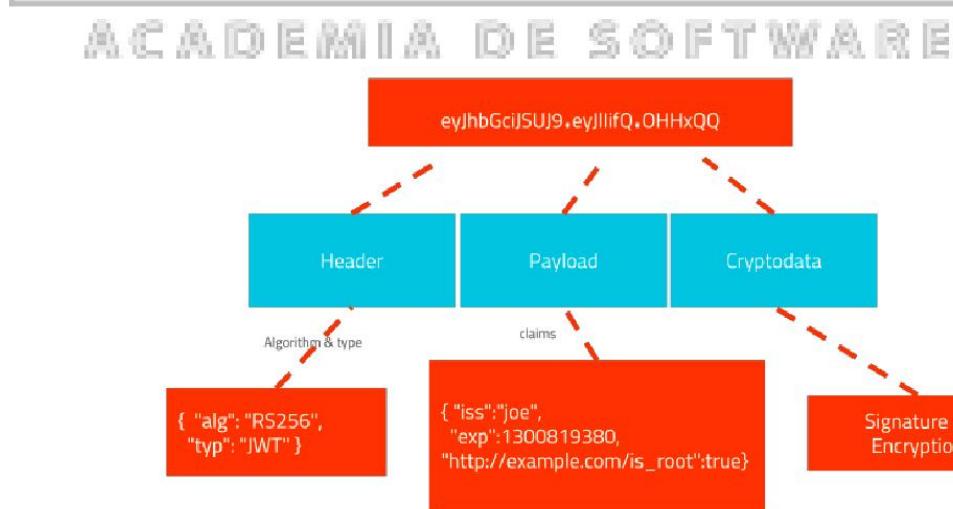
Este es un ejemplo de un JSON Web Token

eyJhbGciOiJIUzUxMiIsInR5cCI6  
IkpXVCJ9.eyJpc3MiOiJtb3VzZSI  
sImV4cCI6MTUzNDk20TA1Mywic3V  
iIjoiSkMiLCJsdmwiOiJjbGllbnR  
IiwiWF0IjoxNTM0OTY4NzUzfQ.  
Er1Gzegyv1Vxs7kLammEX67tIeAc  
Sr3KDEnnw8AiZG1EBVyzhm1L350e  
OYUacEeAUISlsRQdXHxeMRb2y6Cm  
Iw

#### 14.2.- Estructura y Encoding

Los JWT tienen una estructura definida y estándar basada en tres partes constituidas por strings. Cada una de ellas cumple un papel principal y están cifradas con una llave (clave).

El formato de un JWT está compuesto por 3 strings separados por un punto. Ejemplo:



### Header:

El Header es la cabecera del token, contiene el tipo de cifrado y el tipo de dato. El tipo de dato, en este caso siempre será “JWT” y la codificación utilizada comúnmente es el algoritmo HMAC SHA256. Este JSON es codificado en Base64. El contenido sin codificar es el siguiente:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

### Payload:

El payload de un JWT es un JSON que puede tener cualquier propiedad, aquí es el lugar donde se enviar toda la información que se desea transportar. Este JSON es codificado a Base64. Ejemplo:

```
{  
  "id": "1",  
  "username": "Cadif1",  
  "nombre": "José Rojas"  
}
```

### Signature:

El signature es la tercera y última parte del JSON Web Token. Está formada por los anteriores componentes (Header y Payload) cifrados en Base64 con una clave secreta (almacenada en nuestro backend). Así sirve de Hash para comprobar que todo está bien.

### 14.3.- Generación

La construcción de un JWT está constituida de 3 strings: el Header, Payload y Signature. El Header y Payload van cifrados en Base64 y separados por un punto “.”. Serán la cabecera y cuerpo del JWT.

Luego se encuentra el Signature, que es la última parte que compone a un JWT y es donde va la clave privada para cifrar el JWT. El Signature está compuesto de el header y payload codificados en Base64 y luego todo esto, cifrado con HS256 con la clave privada.

El contenido de un JWT puede ser visto por cualquiera, es información es completamente visible y publica, lo que no se puede ver es la clave privada, por ello hay que tener precaución de los datos que se envían en un JWT, aunque es una manera segura de validar datos, los datos son visibles por cualquiera.

Existen páginas como [JWT.io](#) que permiten ver el contenido de los JWT y validar su clave, porque, aunque el contenido de un JWT sea visible y la clave no, esta página internamente valida la clave en el caso que se desee ver si el JWT no este manipulado externamente

The screenshot shows the [JWT.io](#) interface. On the left, under 'Encoded', there is a large input field containing a long string of characters: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SfIKxwRJSMeKKF2Q.T4fwpMeJf36P0k6yJV_adQssw5c`. On the right, under 'Decoded', the token is split into two sections: 'HEADER: ALGORITHM & TOKEN TYPE' and 'PAYLOAD: DATA'. The Header section contains: `{ "alg": "HS256", "typ": "JWT" }`. The Payload section contains: `{ "sub": "1234567890" }`.

con respecto a la clave privada, debe ser completamente segura, que nadie además de un pequeño grupo de personas debe de tener acceso a él, ya que, la clave privada será el medio por el cual el JWT pueda ser decodificado.

En el caso de que alguien consiguiera la clave secreta, podrían firmar JWT, haciéndose pasar por nosotros, cuando en realidad, es información manipulada y fraudulenta.

Existen muchas librerías para la generación de un JWT e distintos lenguajes de programación. Una librería para generar un JWT con PHP de manera sencilla es con "firebase/php-jwt". Ejemplo:

```
<?php
include_once 'JWT/BeforeValidException.php';
include_once 'JWT/ExpiredException.php';
include_once 'JWT/JWT.php';
include_once 'JWT/SignatureInvalidException.php';

use \Firebase\JWT\JWT;

$key = "example_key";
$data = [
    'nombre' => 'Jose',
    'apellido' => 'Rojas'
];
$token = JWT::encode($data, $key);
var_dump($token);

// Resultado
'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJsb2xlIjoiZmRzIiwiY29tbyI6ImZzZGQifQ.
vwERvFbAKabepc3XvaoqAMEWG830XVPIHWR2QInFzQI'
```

## Capítulo 15. JWT. PARTE 2

### 15.1.- Claim

Los claims son los atributos que van dentro del payload de un JWT. Aunque en el payload puede ir cualquier JSON con información, cada uno de los atributos de ese JSON son llamados claim.

Como estándar, existen algunos claims predefinidos, que permiten definir algunos atributos y mantener el estándar, para que sin importar el lenguaje de programación que se use, el JWT tenga sus datos estandarizados.

Algunos de esos Claim son:

- Creador (iss) - Identifica a quien creo el JWT
- Razón (sub) - Identifica la razón del JWT, se puede usar para limitar su uso a ciertos casos.
- Audiencia (aud) - Identifica quien se supone que va a recibir el JWT. Un ejemplo puede ser web, android o ios. Quien use un JWT con este campo debe además de usar el JWT enviar el valor definido en esta propiedad de alguna otra forma.
- Tiempo de expiración (exp) - Una fecha que sirva para verificar si el JWT está vencido y obligar al usuario a volver a autenticarse.
- No antes (nbf) - Indica desde qué momento se va a empezar a aceptar un JWT.
- Creado (iat) - Indica cuando fue creado el JWT.
- ID (jti) - Un identificador único para cada JWT.

### 15.2.- Envío Jwt

Una vez recibido en el cliente el JWT, este debe ser enviando en cada una de las peticiones a el servidor, de esta manera el servidor siempre va a tener información del usuario que esta autenticado en el backend.

En caso de que el JWT no sea válido hay que devolver una respuesta de error de autenticación al cliente, para que sea notificado que su información ha sido comprometida.

Por lo general un JWT es enviado en header de la petición HTTP, no tiene un nombre específico dentro del header, puede ser enviado bajo cualquier clave del header, mientras esta sea lo más específica posible.

Un ejemplo de envío de JWT en postman a través de header de la petición con la clave "X-TOKEN":

localhost/prueba/user?7

GET | localhost/prueba/user?7

Send Save

Headers (1)

KEY	VALUE	DESCRIPTION
X-TOKEN	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJlb2xhIjoiZmRzIiw...	

Response

```
{ "id": "7", "name": "John Doe", "email": "john.doe@example.com" }
```

**15.3.- Decodificar**

El proceso de decodificación de un JWT es un poco complicado si se realiza de manera manual, es decir, sin uso de alguna librería, ya que el decodificar el JWT para obtener la información contenida en el payload, se tendría que decodificar el HS256, lo cual no es una tarea sencilla.

Con el uso de la librería "firebase/php-jwt" el decodificar un JWT, es mucho más sencillo, y es una manera alternativa a hacerlo de forma manual. Ejemplo:

```
$key = "example_key";
$decode = JWT::decode($token, $key, array('HS256'));
var_dump($decode);

//Resultado
object(stdClass)[2]
| public 'Hola' => string 'fds' (length=3)
| public 'como' => string 'fsdd' (length=4)
```

