php avanzado

Implementación de API REST con PHP como back-end

José Antonio Carmona Fombella - G1 IS

Introducción a la Ingeniería del Software y los Sistemas de Información

TABLA DE CONTENIDO

[OBJETIVO, TECNOLOGÍAS Y HERRAMIENTAS DE DESARROLLO 3](#_Toc486927605)

[ESCENARIO HIPOTÉTICO Y CONTEXTO 5](#_Toc486927606)

[DISEÑO E IMPLEMENTACIÓN DE LA BASE DE DATOS 5](#_Toc486927607)

[CONTRATO DE LA API REST 7](#_Toc486927608)

[Contrato para el recurso ‘Dispositivo’ 7](#_Toc486927609)

[Dispositivos: Ejemplo JSON con los datos a proporcionar en el cuerpo 9](#_Toc486927610)

[Dispositivos: Ejemplo JSON con los datos devueltos (GET) 9](#_Toc486927611)

[Contrato para el recurso ‘Fabricantes’ 10](#_Toc486927612)

[Fabricantes: Ejemplo JSON con los datos a proporcionar en el cuerpo 11](#_Toc486927613)

[Fabricantes: Ejemplo JSON con los datos devueltos (GET) 11](#_Toc486927614)

[OAUTH 2.0 12](#_Toc486927615)

[Adaptación de la librería al contexto y Oracle Database 13](#_Toc486927616)

[Portal de emisión de tokens 15](#_Toc486927617)

[API REST: Conexión a la BD mediante Objetos PDO 16](#_Toc486927618)

[API REST: Configuración del Servidor Apache 17](#_Toc486927619)

[API REST: Comprobación de la URI 18](#_Toc486927620)

[API REST: Comprobación del método de la petición 19](#_Toc486927621)

[API REST: Procesamiento del método GET 20](#_Toc486927622)

[API REST: Procesamiento del método POST 21](#_Toc486927623)

[API REST: Procesamiento del método PUT 22](#_Toc486927624)

[API REST: Procesamiento del método DELETE 23](#_Toc486927625)

[API REST: Respuestas HTTP 24](#_Toc486927626)

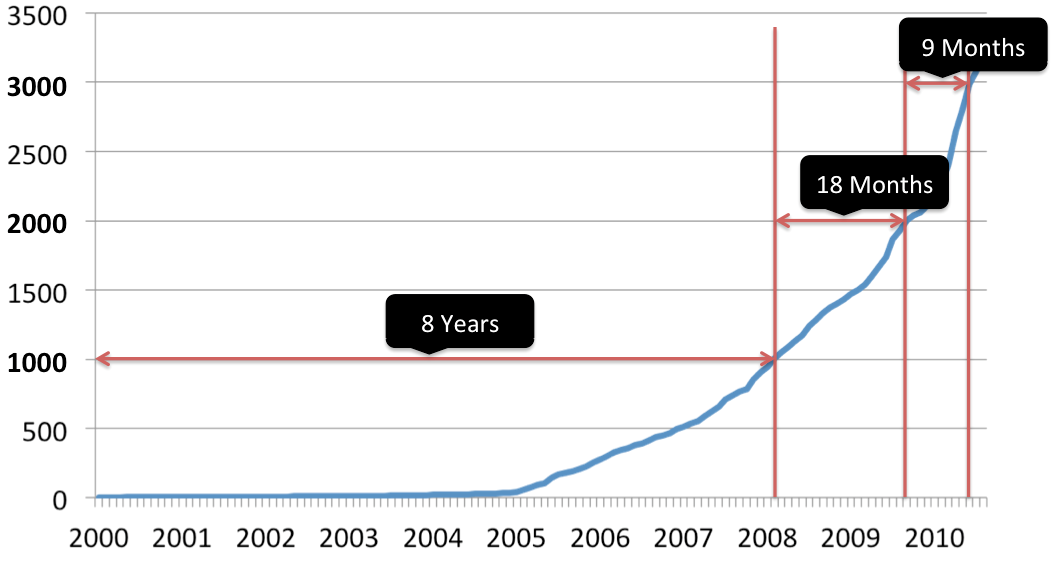
[API REST: Representación de los recursos en JSON 25](#_Toc486927627)

[Pruebas, ejemplos de consulta y respuestas 26](#_Toc486927628)

[Bibliografía 26](#_Toc486927629)

# OBJETIVO, TECNOLOGÍAS Y HERRAMIENTAS DE DESARROLLO

Durante los últimos años, el uso de APIs se ha visto gratamente extendido en toda la web, sobre todo en el campo de los *web services.* Esto hace que el proceso de creación y desarrollo de las mismas sea una etapa fundamental a considerar en cualquier aplicación web.

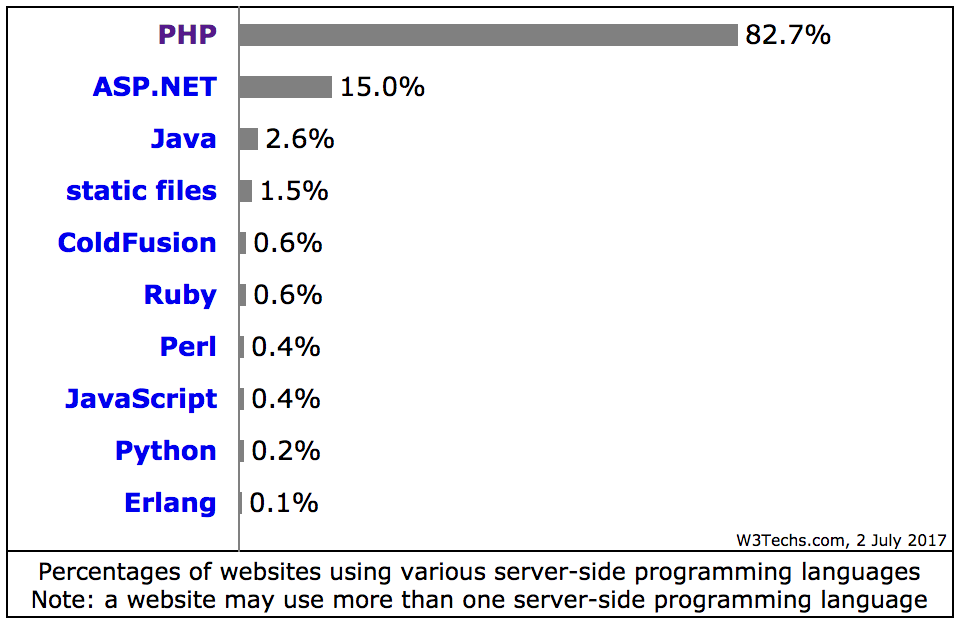


Evolución del número de APIs. En 2013 existían más de 8700 APIs. [1]

Existen varias tecnologías para la implementación de una web API:

* Servicios web RESTful
* Servicios web SOAP
* Javascript
* XML-RPC
* Etc

Nosotros nos centraremos en la primera, ya que es la más usada actualmente debido a su simplicidad y al uso de HTTP como protocolo de aplicación, ampliamente soportado en todas las plataformas.



Market-share de lenguajes de script de servidor [2]

La elección de PHP para implementar esta característica no es una casualidad. Como podemos ver en las estadísticas, de entre todas las opciones disponibles, este lenguaje sigue siendo el predominante para el procesamiento en el lado del servidor. En nuestro caso usaremos PHP v.

Un aspecto fundamental de una API es la autenticación, es decir, el uso de credenciales para restringir el acceso a servicios a ciertos usuarios. Para este cometido usaremos el estándar libre Open Authorization (OAuth) en su versión más reciente (2.0). Este protocolo permite la autorización segura de una API mediante el uso de unos códigos llamados tokens que identifican al usuario sin exponer todos sus datos.

Para implementar la capa de datos (donde guardaremos todos nuestros recursos de la API, así como todo lo relacionado con OAuth2.0) usaremos la base de datos proporcionada por Oracle. Es también una base de datos ampliamente usada, si bien podríamos usar cualquier otra pues su función únicamente será la consulta e inserción de datos.

Durante la fase de desarrollo de la API usaremos varias herramientas:

* XAMPP: Usaremos su servidor web Apache y su intérprete PHP para poder desarrollar en local, sin tener que desplegar cada vez que queramos probar una nueva funcionalidad.
* Oracle Database 11g: Base de datos de Oracle en su versión 11g.
* Oracle SQL Developer: IDE para manejar la BD de Oracle.
* Regex101: Aplicación web que permite la depuración y testeo de expresiones regulares.
* Sublime Text 2: Editor de código multiplataforma con soporte para múltiples lenguajes, entre ellos SQL, PHP, HTML y CSS.
* RESTClient: Plugin para el navegador web Firefox que nos permite depurar y testear de una forma sencilla servicios web RESTful.

# ESCENARIO HIPOTÉTICO Y CONTEXTO

Vamos a desarrollar una API orientada a los principales fabricantes de dispositivos inteligentes de la actualidad. Con ella, se espera que éstos puedan controlar los diferentes terminales que tienen a la venta y las diferentes versiones de los mismos.

Con carácter informativo, cualquier persona debería poder consultar los datos de los fabricantes que existen actualmente en la BD, así como obtener los detalles de un dispositivo en concreto o de todos si lo desea. Sin embargo, operaciones como la modificación, creación o eliminación (aquellas que producen un cambio en el estado de la BD) deben estar restringidas a usuarios registrados. Además, para evitar la manipulación de datos, sólo el propio fabricante debería poder realizar estas operaciones sobre sus datos, así como sobre los dispositivos que tenga asociados.

Con esto en cuenta, se nos pide desarrollar una API RESTful.

# DISEÑO E IMPLEMENTACIÓN DE LA BASE DE DATOS

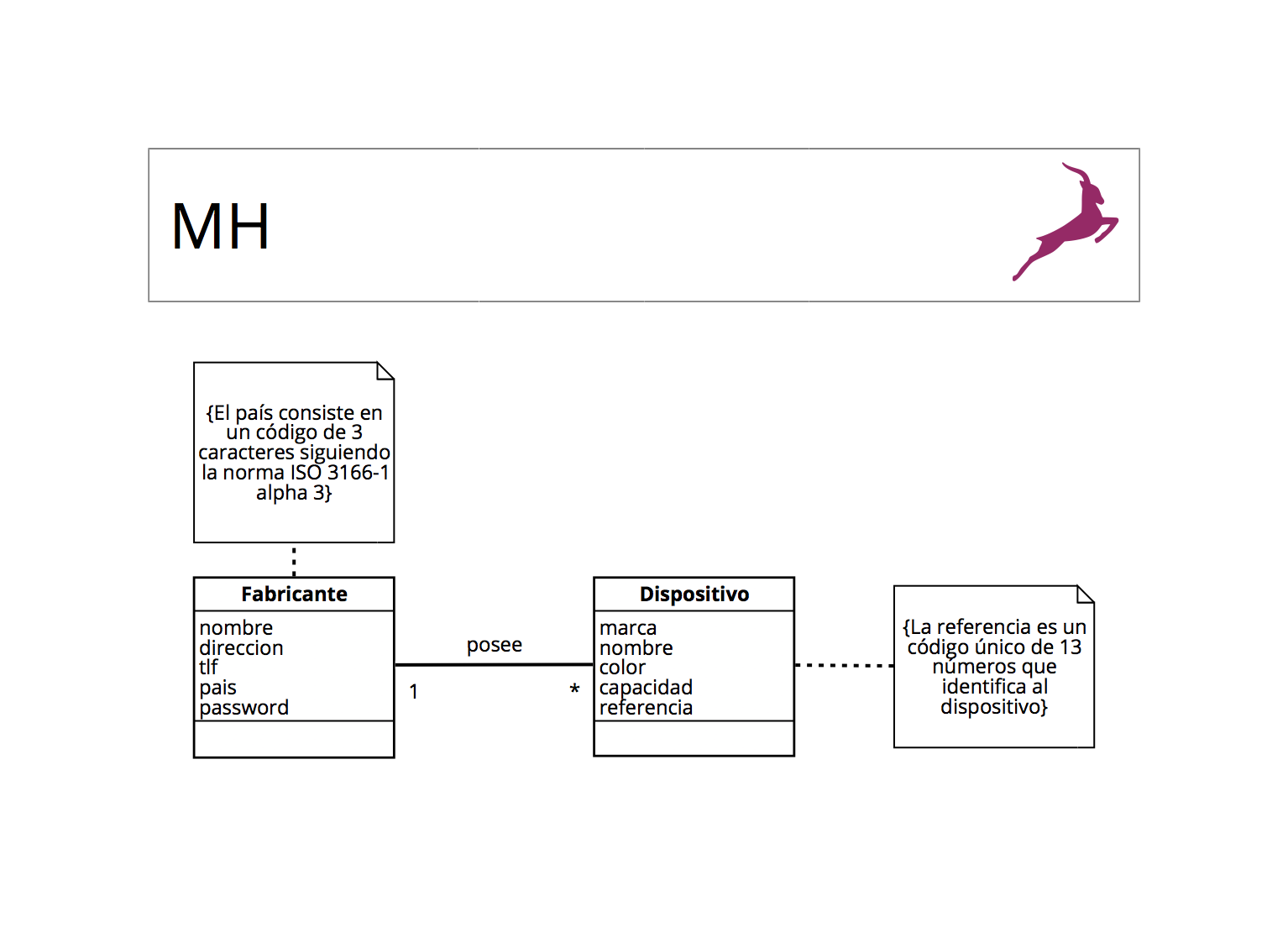
Vamos a suponer que se nos ha pedido (o hemos identificado la necesidad) conocer de cada *Fabricante* los siguientes datos:

* Nombre: El nombre del *Fabricante.* (no se puede repetir)
* Dirección: La dirección del *Fabricante.*
* Teléfono: El número de teléfono del *Fabricante*.
* País: El país donde el *Fabricante* tiene su sede directiva, en formato ISO 3166-1 alpha 3.
* Contraseña: La contraseña que el *Fabricante* desea usar para acceder a los servicios con restricción.

De igual manera, procedemos con los *Dispositivos:*

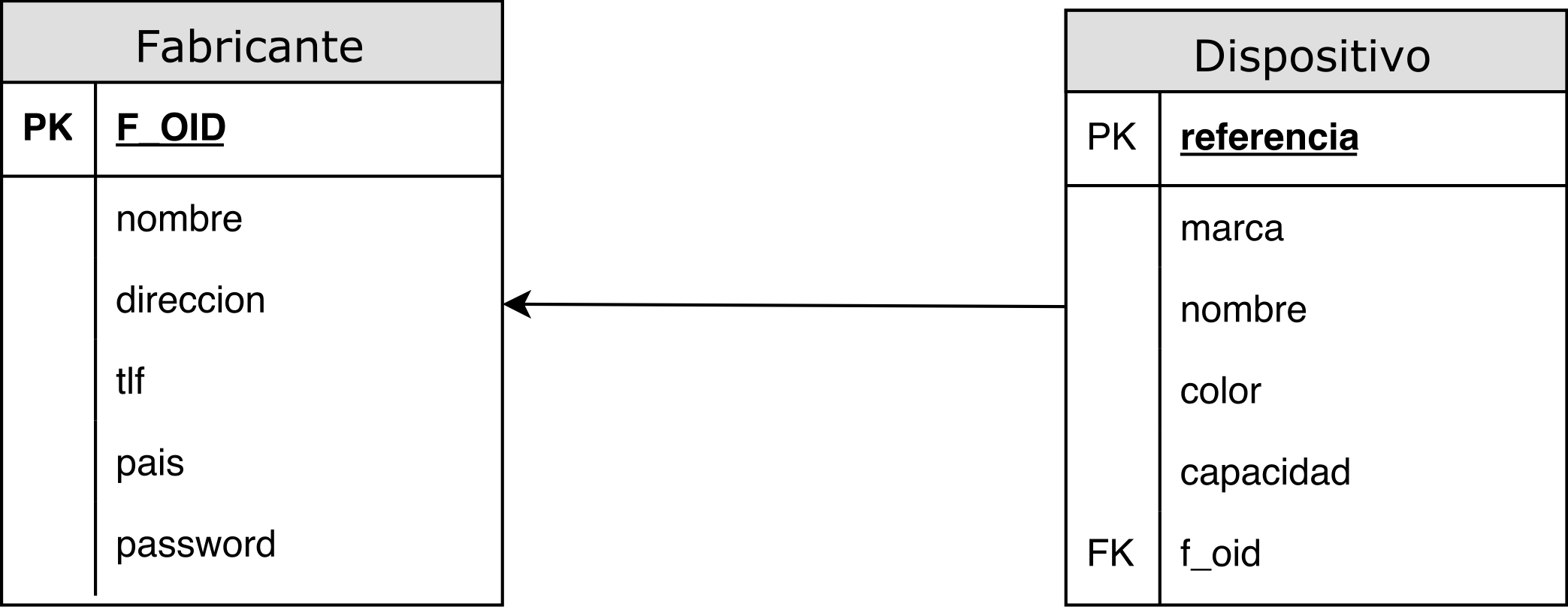
* Marca: El nombre de la marca comercial con la que el *Dispositivo* se lanza al mercado. No necesariamente tiene que ser el nombre del fabricante, pues este puede tener varias submarcas.
* Nombre: El nombre del *Dispositivo*.
* Color: El color de este *Dispositivo.*
* Capacidad: El tamaño de la memoria interna del *Dispositivo*
* Referencia: Código único de 13 números que identifica a un *Dispositivo*. Dos dispositivos con la misma marca y nombre pero con diferente color o capacidad tendrán una referencia diferente.

Teniendo en cuenta estos datos, realizamos un modelo conceptual con los recursos de nuestra API.

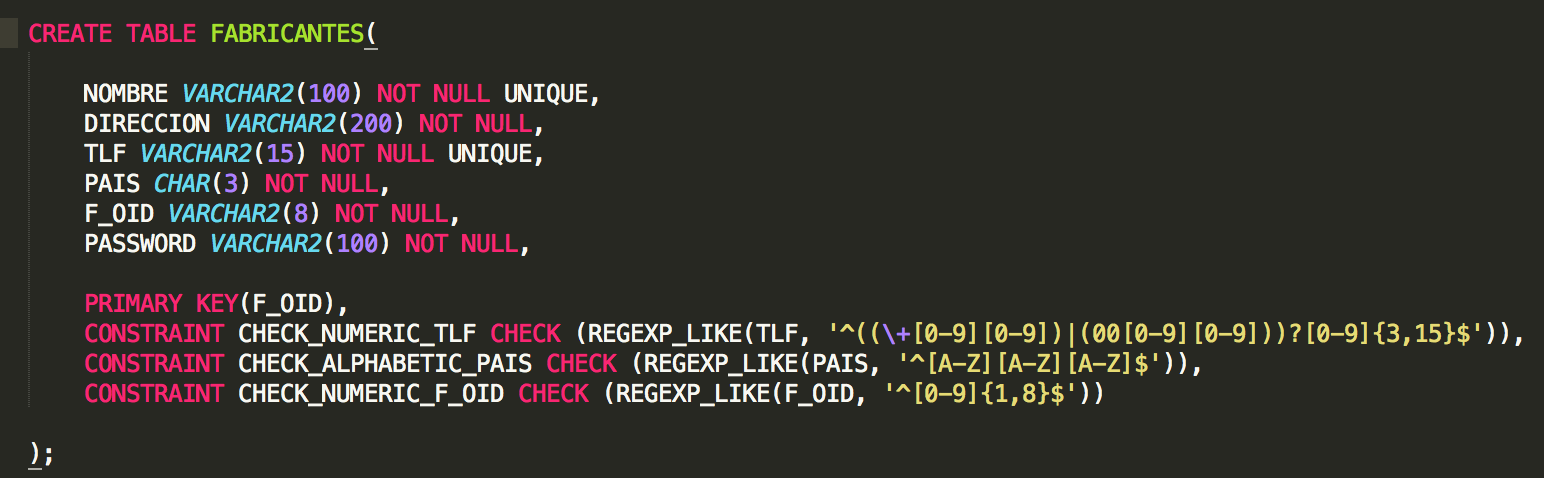


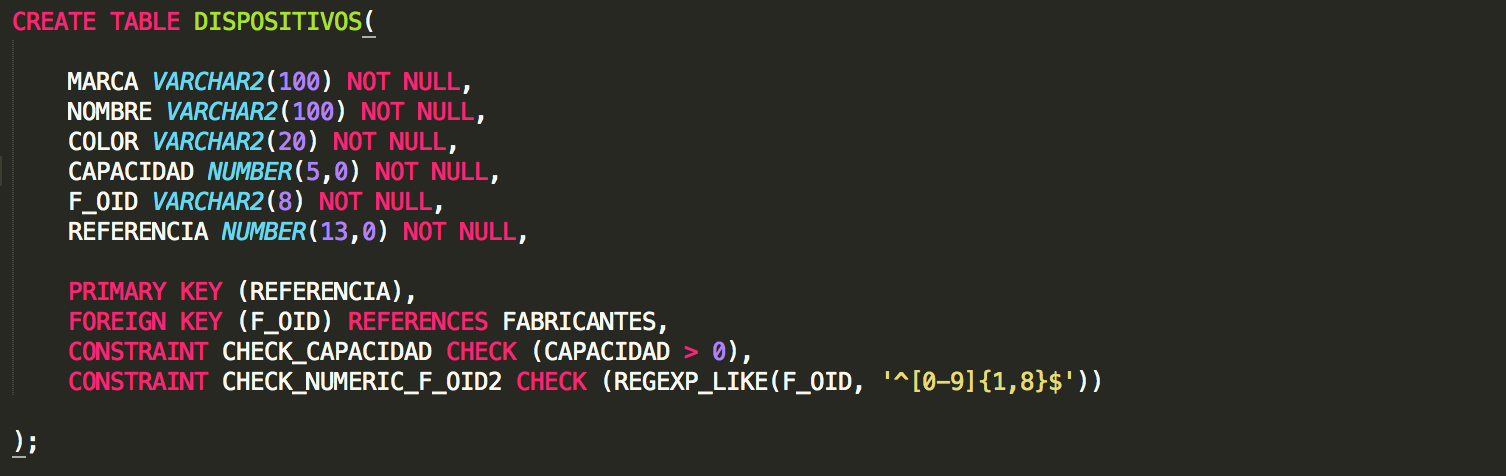
Modelo Conceptual propuesto para la solución al escenario hipotético.

Partiendo de este modelo conceptual, podemos transformarlo en un modelo relacional para su implementación en una BD:



Ya con este modelo en mente, es fácil convertir el problema en sentencias para construir tablas y relaciones SQL en la BD de Oracle:





# CONTRATO DE LA API REST

Un contrato de una API es un documento que la define, exponiendo para ello los recursos a los que podemos acceder, los parámetros, los verbos HTTP y cualquier otra información relevante.

En nuestra API se pretende trabajar con dos recursos: ‘Fabricantes’ y ‘Dispositivos’.

## Contrato para el recurso ‘Dispositivo’

Recurso: *Dispositivo*

|  |  |  |  |
| --- | --- | --- | --- |
| **HTTP** | **URI** | **DESCRIPCIÓN** | **PARÁMETROS** |
| GET | /dispositivos | Devuelve todos los *dispositivos* de la BD en formato JSON.   * **200 Successful Operation**. * **404 Not Found**, si no se encuentran dispositivos en la BD. | limit: número de resultados devueltos. Por defecto es 10.  offset: primer elemento devuelto. Por defecto es 1. |
| GET | /dispositivos/{referencia} | Devuelve el *dispositivo* con la referencia en formato JSON.   * **200 Successful Operation**, si el dispositivo se ha encontrado. * **404 Not Found**, si no hay ningún dispositivo con tal referencia. | Ninguno. |
| POST | /dispositivos | Añade el *dispositivo* que se pasa por el cuerpo en formato JSON.   * **201 Created**, si la operación se realiza con éxito. * **400 Bad Request**, si alguno de los campos no es válido. * **401 Unauthorized**, si el token no es válido. | *token:* código temporal que autoriza la operación.  JSON en el cuerpo con los datos del dispositivo. |
| PUT | /dispositivos/{referencia} | Actualiza el *dispositivo* que tiene la *referencia* usando los valores que se envían en el cuerpo en formato JSON.   * **204 No Content**, si se actualiza correctamente. * **400 Bad Request**, si alguno de los campos no es válido. * **401 Unauthorized**, si el token no es válido. * **403 Forbidden**, si el token de usuario no corresponde con el fabricante del dispositivo a actualizar. | *token:* código temporal que autoriza la operación.  JSON en el cuerpo con los datos del dispositivo a actualizar. |
| DELETE | /dispositivos/{referencia} | Borra un *dispositivo* dado su *referencia.*   * **204 No Content**, si se realiza correctamente. * **401 Unauthorized**, si el token no es válido * **403 Forbidden**, si el token de usuario no corresponde con el fabricante del dispositivo a eliminar. | *token:* código temporal que autoriza la operación. |

## Dispositivos: Ejemplo JSON con los datos a proporcionar en el cuerpo

POST y PUT:

1. {
2. "REFERENCIA": "1000000000000",
3. "MARCA": "Apple",
4. "NOMBRE": "iPhone 7 Plus",
5. "COLOR": "Jet Black",
6. "CAPACIDAD": "128",
7. }

## Dispositivos: Ejemplo JSON con los datos devueltos (GET)

Resultado de la consulta a la URI: <http://localhost/PHP_API/dispositivos?limit=2>

1. [
2. {
3. "Total Recursos": 11,
4. "Resultados Consulta": 2
5. },
6. [
7. {
8. "RNUM": "1",
9. "MARCA": "Apple",
10. "NOMBRE": "iPhone 7 Plus",
11. "COLOR": "Jet Black",
12. "CAPACIDAD": "128",
13. "F\_OID": "1",
14. "REFERENCIA": "1000000000000"
15. },
16. {
17. "RNUM": "2",
18. "MARCA": "Apple",
19. "NOMBRE": "iPad Air 2",
20. "COLOR": "Space Gray",
21. "CAPACIDAD": "64",
22. "F\_OID": "1",
23. "REFERENCIA": "1000000000001"
24. }
25. ]
26. ]

## 

## Contrato para el recurso ‘Fabricantes’

Recurso: *Fabricantes*

|  |  |  |  |
| --- | --- | --- | --- |
| **HTTP** | **URI** | **DESCRIPCIÓN Y CÓDIGOS** | **PARÁMETROS** |
| GET | /fabricantes | Devuelve todos los *fabricantes* de la BD en formato JSON.   * **200 Successful Operation**. * **404 Not Found**, si no se encuentran fabricantes en la BD. | limit: número de resultados devueltos. Por defecto es 10.  offset: primer elemento devuelto. Por defecto es 1. |
| GET | /fabricantes/{f\_oid} | Devuelve el *fabricante* con la *f\_oid* en formato JSON.   * **200 Successful Operation**, si el fabricante se ha encontrado. * **404 Not Found**, si no hay ningún fabricante con tal *f\_oid*. | Ninguno. |
| PUT | /fabricantes | Actualiza el *fabricante* del token actual, usando los valores que se envían en el cuerpo en formato JSON.   * **204 No Content,** si se actualiza correctamente. * **400 Bad Request**, si alguno de los campos no es válido. * **401 Unauthorized,** si el token no es válido. | *token:* código temporal que autoriza la operación.  JSON en el cuerpo con los datos del dispositivo a actualizar. |

## Fabricantes: Ejemplo JSON con los datos a proporcionar en el cuerpo

PUT:

1. {
2. "NOMBRE": "Apple Inc.",
3. "DIRECCION": "1 Infinite Loop, California",
4. "PAIS": "USA",
5. "TLF": "+1987654320"
6. }

## Fabricantes: Ejemplo JSON con los datos devueltos (GET)

Resultado de la consulta a la URI: <http://localhost/PHP_API/fabricantes?limit=2>

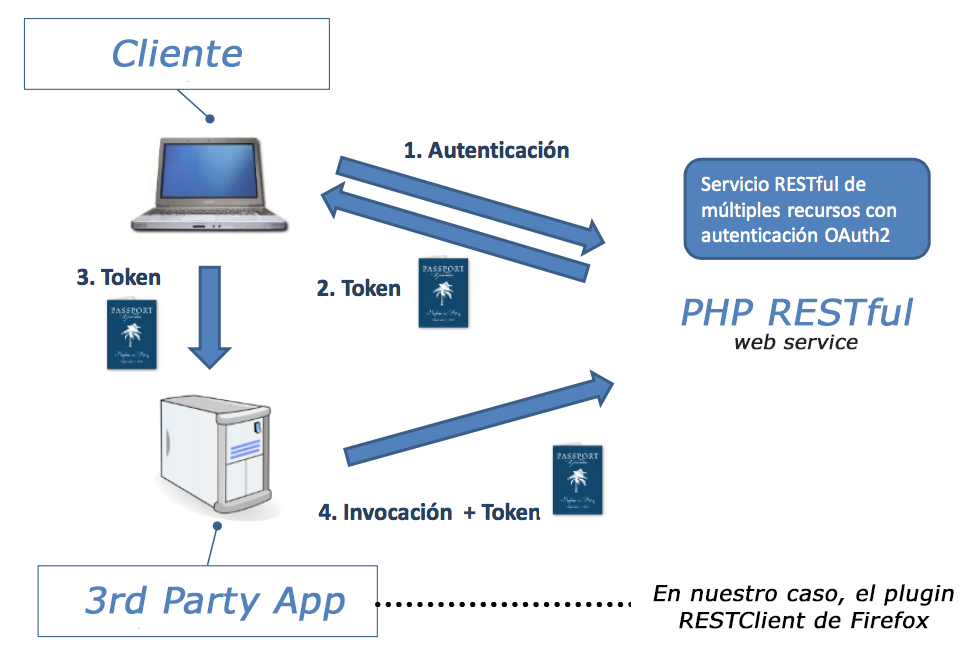
1. [
2. {
3. "Total Recursos": 5,
4. "Resultados Consulta": 2
5. },
6. [
7. {
8. "RNUM": "1",
9. "NOMBRE": "Apple Inc.",
10. "F\_OID": "1",
11. "DIRECCION": "1 Infinite Loop, California",
12. "PAIS": "USA",
13. "TLF": "+1987654320"
14. },
15. {
16. "RNUM": "2",
17. "NOMBRE": "BQ",
18. "F\_OID": "2",
19. "DIRECCION": "Las Rozas, Madrid",
20. "PAIS": "ESP",
21. "TLF": "+34911829384"
22. }
23. ]
24. ]

## 

# OAUTH 2.0

Es un estándar abierto que permite la autorización segura de una API mediante el uso de unos códigos llamados tokens, que identifican al usuario sin exponer todos sus datos. Es ampliamente usado en la web para servicios que requieran el uso de credenciales y el acceso a terceros.

En la siguiente imagen encontraremos un flujo normal de autenticación con OAuth2.0, donde tendremos a nuestra API PHP como proveedor del servicio:



Como vemos, el cliente procede a autenticarse directamente con el proveedor de servicios, no compartiendo ninguna credencial con la aplicación de terceros. Cuando el cliente se identifica, se le proporciona un token temporal que éste suministra a la aplicación que quiere acceder a sus datos, la cual invoca al servicio usando el código proporcionado. Normalmente esta aplicación se ejecuta en otro terminal, aunque en nuestro caso sea en el mismo.

De esta forma, garantizamos que sólo el *Fabricante* en cuestión puede modificar sus datos y los de sus *Dispositivos* asociados, denegando las operaciones a cualquier otro usuario que intente realizar cambios.

Para implementar esta funcionalidad me he basado en el trabajo de Bshaffer [3], el cual no provee con una librería en PHP para abordar este problema.

## Adaptación de la librería al contexto y Oracle Database

La librería proporcionada viene preparada para el su uso en sistemas con BD como MySQL, SQLite, PostgreSQL ó MS SQL Server. Para poder usarla con la BD de Oracle y adaptarla a nuestro problema hay que realizar unos pequeños cambios que veremos a continuación.

En primer lugar, debemos crear 7 tablas que contendrán información específica del protocolo OAuth2.0 en nuestra BD, como por ejemplo los códigos tokens, los clientes, los usuarios, los scopes o permisos, …

En OAuth 2.0, denominamos cliente a la aplicación que pide la emisión de un token. Un usuario por el contrario, es la persona o entidad que introduce sus credenciales para tener acceso a los datos.

En la guía de la librería se nos proporciona con el siguiente código para la creación de las tablas:

**CREATE** **TABLE** oauth\_clients (client\_id **VARCHAR**(80) **NOT** **NULL**, client\_secret **VARCHAR**(80), redirect\_uri **VARCHAR**(2000) **NOT** **NULL**, grant\_types **VARCHAR**(80), scope **VARCHAR**(100), user\_id **VARCHAR**(80), **CONSTRAINT** clients\_client\_id\_pk **PRIMARY** **KEY** (client\_id));

**CREATE** **TABLE** oauth\_access\_tokens (access\_token **VARCHAR**(40) **NOT** **NULL**, client\_id **VARCHAR**(80) **NOT** **NULL**, user\_id **VARCHAR**(255), expires **TIMESTAMP** **NOT** **NULL**, scope **VARCHAR**(2000), **CONSTRAINT** access\_token\_pk **PRIMARY** **KEY** (access\_token));

**CREATE** **TABLE** oauth\_authorization\_codes (authorization\_code **VARCHAR**(40) **NOT** **NULL**, client\_id **VARCHAR**(80) **NOT** **NULL**, user\_id **VARCHAR**(255), redirect\_uri **VARCHAR**(2000), expires **TIMESTAMP** **NOT** **NULL**, scope **VARCHAR**(2000), **CONSTRAINT** auth\_code\_pk **PRIMARY** **KEY** (authorization\_code));

**CREATE** **TABLE** oauth\_refresh\_tokens (refresh\_token **VARCHAR**(40) **NOT** **NULL**, client\_id **VARCHAR**(80) **NOT** **NULL**, user\_id **VARCHAR**(255), expires **TIMESTAMP** **NOT** **NULL**, scope **VARCHAR**(2000), **CONSTRAINT** refresh\_token\_pk **PRIMARY** **KEY** (refresh\_token));

**CREATE** **TABLE** oauth\_users (username **VARCHAR**(255) **NOT** **NULL**, password **VARCHAR**(2000), first\_name **VARCHAR**(255), last\_name **VARCHAR**(255), **CONSTRAINT** username\_pk **PRIMARY** **KEY** (username));

**CREATE** **TABLE** oauth\_scopes (scope **TEXT**, is\_default **BOOLEAN**);

**CREATE** **TABLE** oauth\_jwt (client\_id **VARCHAR**(80) **NOT** **NULL**, subject **VARCHAR**(80), public\_key **VARCHAR**(2000), **CONSTRAINT** jwt\_client\_id\_pk **PRIMARY** **KEY** (client\_id));

Sin embargo, debemos tener en cuenta que los tipos de dato **TEXT** y **BOOLEAN** no están disponibles en Oracle SQL, por lo que debemos cambiarlos a otros que cumplan su función de forma equivalente.

En nuestro caso, optaremos por **VARCHAR(2000)** y **NUMBER(1)** para reemplazarlos, ya que de todas formas no es algo demasiado relevante al no usar scopes.

El código quedaría de la siguiente forma:

**CREATE** **TABLE** oauth\_clients (client\_id **VARCHAR**(80) **NOT** **NULL**, client\_secret **VARCHAR**(80), redirect\_uri **VARCHAR**(2000) **NOT** **NULL**, grant\_types **VARCHAR**(80), scope **VARCHAR**(100), user\_id **VARCHAR**(80), **CONSTRAINT** clients\_client\_id\_pk **PRIMARY** **KEY** (client\_id));

**CREATE** **TABLE** oauth\_access\_tokens (access\_token **VARCHAR**(40) **NOT** **NULL**, client\_id **VARCHAR**(80) **NOT** **NULL**, user\_id **VARCHAR**(255), expires **TIMESTAMP** **NOT** **NULL**, scope **VARCHAR**(2000), **CONSTRAINT** access\_token\_pk **PRIMARY** **KEY** (access\_token));

**CREATE** **TABLE** oauth\_authorization\_codes (authorization\_code **VARCHAR**(40) **NOT** **NULL**, client\_id **VARCHAR**(80) **NOT** **NULL**, user\_id **VARCHAR**(255), redirect\_uri **VARCHAR**(2000), expires **TIMESTAMP** **NOT** **NULL**, scope **VARCHAR**(2000), **CONSTRAINT** auth\_code\_pk **PRIMARY** **KEY** (authorization\_code));

**CREATE** **TABLE** oauth\_refresh\_tokens (refresh\_token **VARCHAR**(40) **NOT** **NULL**, client\_id **VARCHAR**(80) **NOT** **NULL**, user\_id **VARCHAR**(255), expires **TIMESTAMP** **NOT** **NULL**, scope **VARCHAR**(2000), **CONSTRAINT** refresh\_token\_pk **PRIMARY** **KEY** (refresh\_token));

**CREATE** **TABLE** oauth\_users (username **VARCHAR**(255) **NOT** **NULL**, password **VARCHAR**(2000), first\_name **VARCHAR**(255), last\_name **VARCHAR**(255), **CONSTRAINT** username\_pk **PRIMARY** **KEY** (username));

**CREATE** **TABLE** oauth\_scopes (scope **VARCHAR(2000)**, is\_default **NUMBER(1)**);

**CREATE** **TABLE** oauth\_jwt (client\_id **VARCHAR**(80) **NOT** **NULL**, subject **VARCHAR**(80), public\_key **VARCHAR**(2000), **CONSTRAINT** jwt\_client\_id\_pk **PRIMARY** **KEY** (client\_id));

Seguidamente, deberemos dirigirnos al archivo *server.php* y cambiar los datos de acceso a la BD por los de la nuestra (el usuario, la contraseña y el nombre de origen de datos).

Existe otro problema que hace que la librería de BShaffer no funcione. Tras testear el código y mediante depuración pude comprobar que cuando se acceden a los campos de datos devueltos por objetos PDO se utiliza el nombre del índice en minúscula para acceder a la información. Por experiencia, sabemos que cuando trabajamos con objetos PDO y Oracle BD debemos indicar el nombre del campo al que queremos acceder en mayúscula, pues si no es así se nos indicará que el índice no existe.

La solución consiste en cambiar todos los índices que se usan para acceder a campos devueltos por objetos PDO a mayúscula.

Por último, cada vez que se cree un token, además de registrar qué cliente lo pidió, queremos conocer cuál fue el usuario que autorizó esa petición de emisión. Existe un campo en la tabla que almacena los tokens (OAUTH\_ACCESS\_TOKENS) llamado USER\_ID que nos permite guardar tal información.

En nuestro caso, guardaremos el identificador del *Fabricante* que emitió ese token en esta columna, guardando este dato en sesión y modificando un poco la parte del código encargada guardarlo en la BD.

En el archivo GrantType/ClientCredentials.php modificarmos la función *getUserId()* tal que:

public function getUserId()

{

$this->loadClientData();

return **isset($\_SESSION['user\_id\_passthrough']) ? $\_SESSION['user\_id\_passthrough']** : ( isset($this->clientData['USER\_ID']) ? $this->clientData['USER\_ID'] : null );

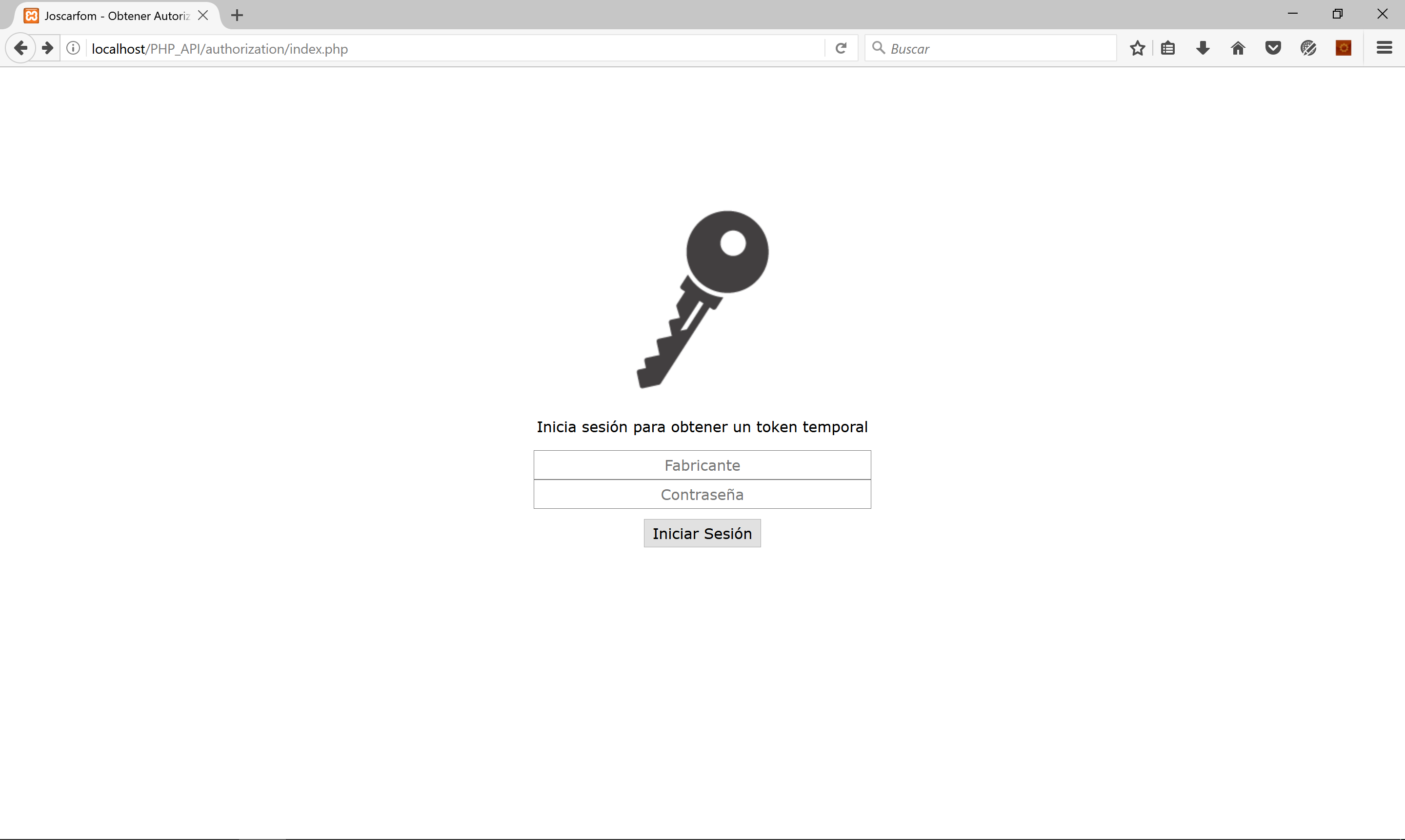
}

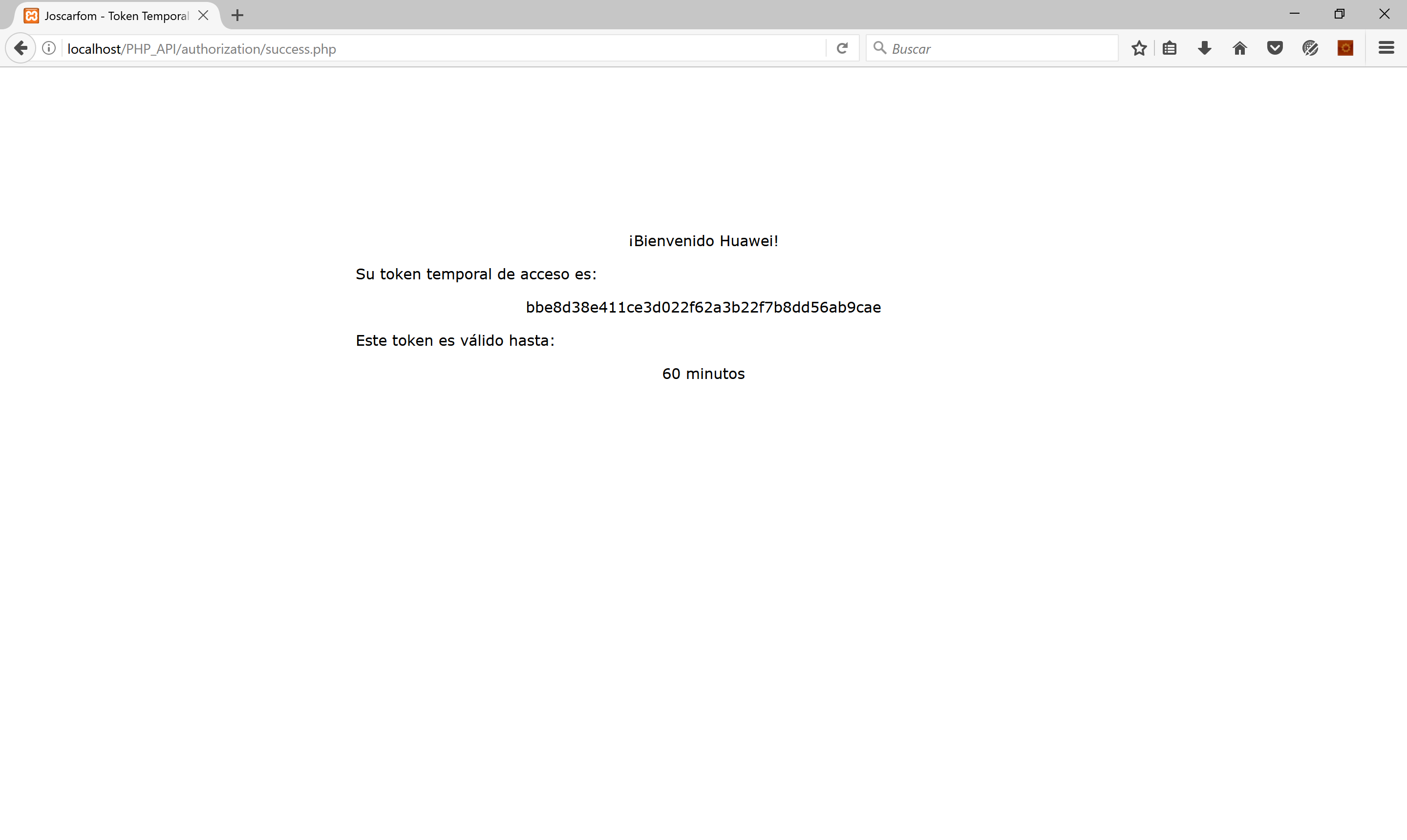
Insertando esta sentencia en el código original conseguiremos que si existiera un identificador de *Fabricante* en la sesión, se usaría éste. Si no fuera así, se continuaría con el flujo de operación normal de la librería.

Una vez realizadas estas modificaciones, ya tendríamos finalizada toda la parte de datos de nuestra API.

## Portal de emisión de tokens

Como hemos visto anteriormente, se precisa de una herramienta que nos permita autenticar al usuario ante el servicio web y proveerle con un token en caso de éxito. Es por ello que hemos creado un simple portal web que loguea al *Fabricante* en nuestro servidor y emite un token válido durante 60 minutos. En la práctica, éste sería el sitio web donde las aplicaciones de terceros redirigirían en un flujo normal de OAuth2.0 para que el usuario iniciase la comunicación con el proveedor de servicios.

****

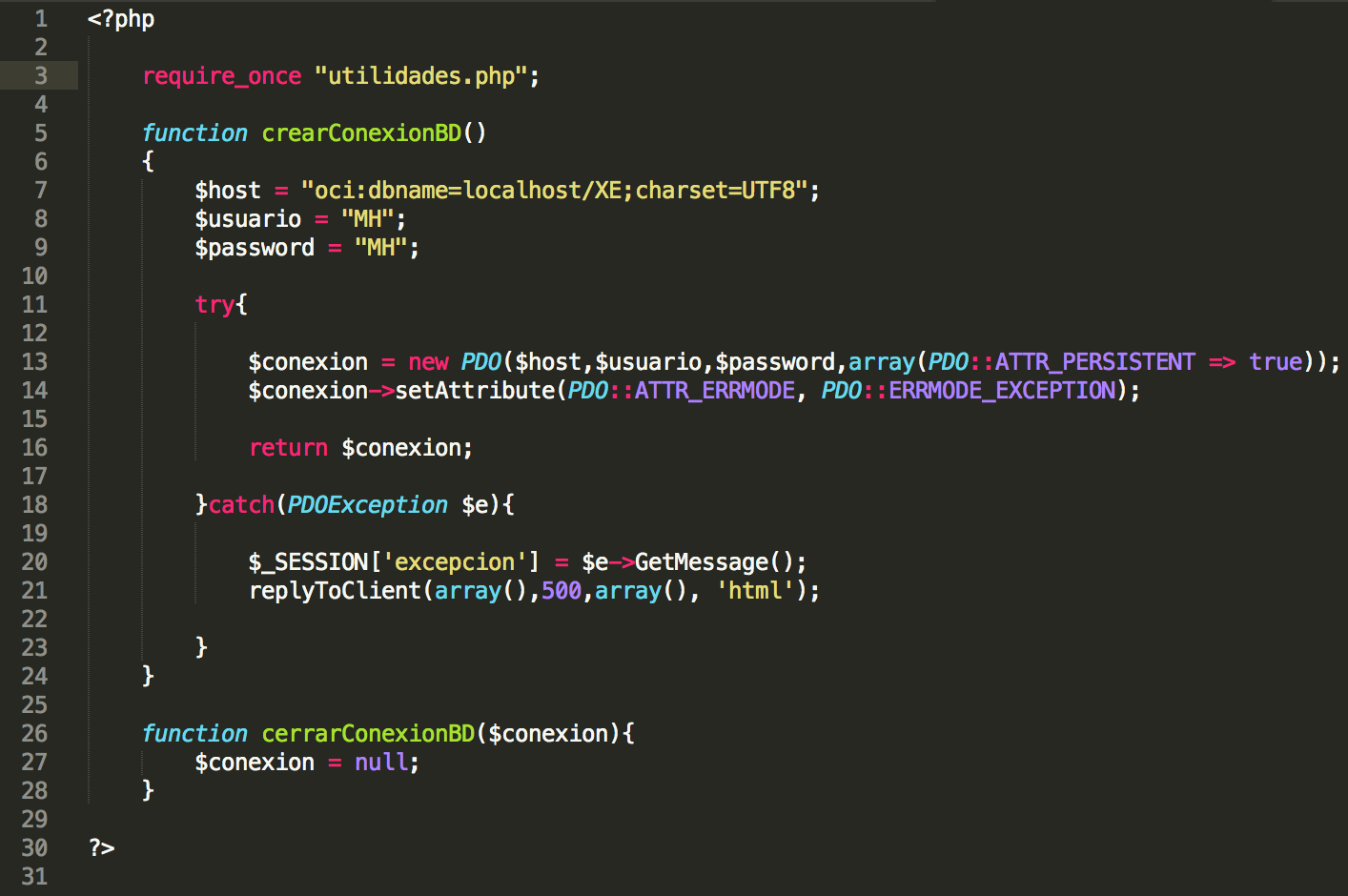


Con todo esto hecho, podemos proceder a la implementación en PHP de nuestra API RESTful.

# API REST: Conexión a la BD mediante Objetos PDO

En primer lugar, debemos ser capaces de acceder a la BD que contiene toda la información que hemos definido más arriba. Una forma fácil de hacerlo en PHP es mediante el uso de objetos PDO.

Para hacer sencillo este proceso, hemos creado un archivo PHP llamado *gestionBD.php* en el que definiremos el objeto PDO y dos métodos: el de creación y destrucción del mismo.



En él intentamos crear un objeto PDO con los datos de acceso a nuestra PDO y establecemos el modo de error y de excepciones. Si todo ocurre correctamente, se devuelve este objeto. En cualquier otro caso, se guarda el error en una variable en sesión y se responde al usuario con un código HTTP 500, que indica Error Interno del servidor.

La destrucción del objeto es tan sencilla como asignarle el valor de null.

# API REST: Configuración del Servidor Apache

Para nuestra API, necesitaremos acceder a URL que en nuestro servidor no corresponden a ningún recurso, por lo que la respuesta natural del mismo será informar de ello al usuario. Por ejemplo, para accerder a los *Fabricantes* se debería usar la siguiente URI: http:/localhost/PHP\_API/fabricantes, pero no existe ningún directorio o archivo con ese nombre. ¿Cómo solucionarlo?

Para ello crearemos un archivo ***.htaccess****,* que es un archivo de configuración para servidores web basados en Apache que permite a los administradores aplicar distintas políticas de acceso a directorios o archivos.

Para ello basta con crear un archivo llamado htaccess.txt en la raíz de nuestro directorio y renombrarlo luego mediante la consola de comandos a .htaccess.

En Mac/Linux:

mv htaccess.txt .htaccess

En Windows:

rename htaccess.txt .htaccess

De esta forma conseguiremos que sea reconocido por el sistema de archivos de Apache.

En él guardamos el siguiente código:

RewriteEngine on

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteRule ^(.\*)$ PHP\_API/main.php?req\_path=$1 [L,QSA]

Analicemos el código línea por línea:

1. Iniciamos el motor de reescritura de Apache para establecer nuestra regla
2. Establecemos una condición de reescritura: Si se cumple se aplicará la regla de reescritura. En este caso la condición comprueba que no es un archivo cualquiera existente en el servidor (!-f 🡪 not a regular **f**ile ).
3. Establecemos una condición de reescritura: Si se cumple se aplicará la regla de reescritura. En este caso la condición comprueba que no es un directorio existente en el servidor (!-d 🡪 not a **d**irectory ).
4. Regla de reescritura que se aplicará.

^(.\*)$ 🡪 Selecciona toda la URI a la que se está intentando acceder

PHP\_API/main.php?req\_path=$1 🡪 Ruta a la que redirige la URI que se ha intentado acceder como un parámetro llamado ‘req\_path’.

L 🡪 (LAST) Fuerza la detención del proceso de reescritura y no aplica ninguna regla más.

QSA 🡪 (QSAPPEND) Añade cualquier cadena de texto al final del parámetro ‘req\_path’. Se usa para añadir los parámetros que se adjuntaran con la URI a la que se ha intentado acceder (por ejemplo ?param1=hola).

Con este archivo hemos conseguido que cuando no se encuentre un archivo o directorio se redireccione a *main.php* (el archivo principal de nuestra API). Además podremos acceder a la ruta URL que se intentó acceder mediante el parámetro *req\_path.*

# API REST: Comprobación de la URI

Procederemos ahora a crear el archivo principal de nuestra API, donde comenzará el procesamiento cada vez que se realice una petición.



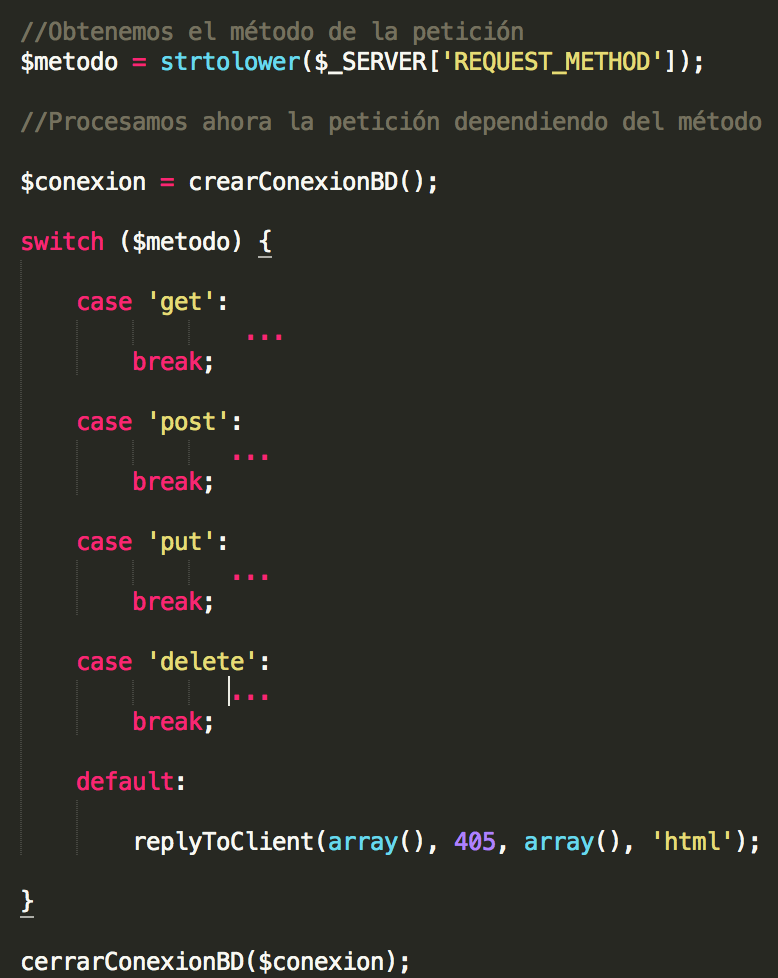
El primer paso es comprobar que la URI es correcta y que no se está intentando acceder a una inválida o el propio archivo de la API directamente. Para ello comprobamos que el parámetro *req\_path* existe y que el primer campo de la URI coincide con alguno de nuestros recursos.

Si no existiera *req\_path* significaría que se ha intentado acceder al archivo directamente, por lo que se respondería con un código de error HTTP 400 (Bad Request).

Si por el contrario existe pero no coincide con ninguno de los recursos disponibles (*Fabricantes o Dispositivos*) devolveríamos un código de error HTTP 404 (Not Found).

# API REST: Comprobación del método de la petición

Una vez obtenida y comprobada la URI de la petición, continuamos identificando el método HTTP asociado a la petición, verificando que es uno de los que tenemos en nuestra API y procesándolo en caso de ser correcto.



Para ello obtenemos el método de la petición mediante la variable global *$\_SERVER* e iniciamos una conexión con la BD que será usada para el procesamiento de aquellos métodos válidos.

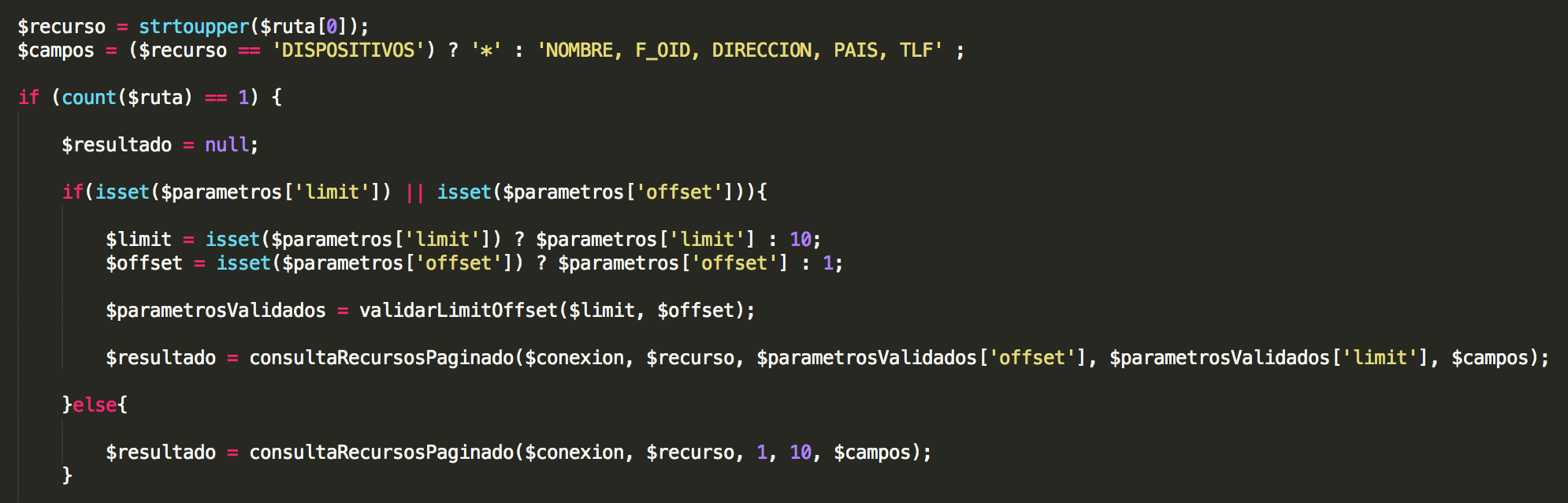
Mediante un bloque de control, verificamos que el método sea GET, POST, PUT ó DELETE, y en caso de no serlo le indicamos al cliente que el método que ha usado no está permitido mediante el código de estado HTTP 405 (Method not Allowed).

Si el método es correcto procederemos a su procesamiento.

En cualquiera de los casos, se acaba cerrando la conexión previamente abierta.

# API REST: Procesamiento del método GET

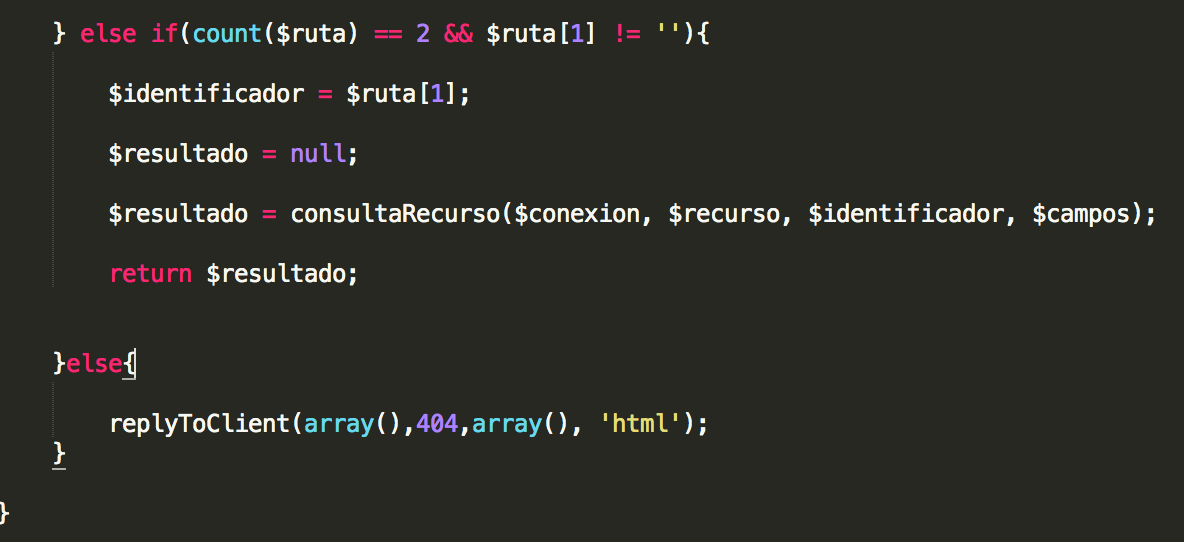
Una vez que se ha identificado que el método solicitado es del tipo GET, procederemos a determinar cuál de las 4 posibilidades que existen en nuestro contrato de la API es la que está siendo requerida.



Se comienza por identificar el recurso al que va dirigida la petición ya que, en nuestro caso, existe la posibilidad de realizar la consulta para *Fabricantes* o *Dispositivos.*

Seguidamente, se comprueba la longitud de la ruta. Si observamos la API, sólo existen dos patrones a seguir para las peticiones GET: uno con un campo (GET de todos los recursos) y otro con dos (GET de un recurso específico).

En el caso de sólo tener un campo, se prosigue con la identificación y validación de los parámetros que pudieran estar presente. Si no existiesen, se realizaría una consulta con los valores por defecto de los mismos. Finalmente, se devuelve el resultado de una consulta a la BD con toda la información que hemos identificado.

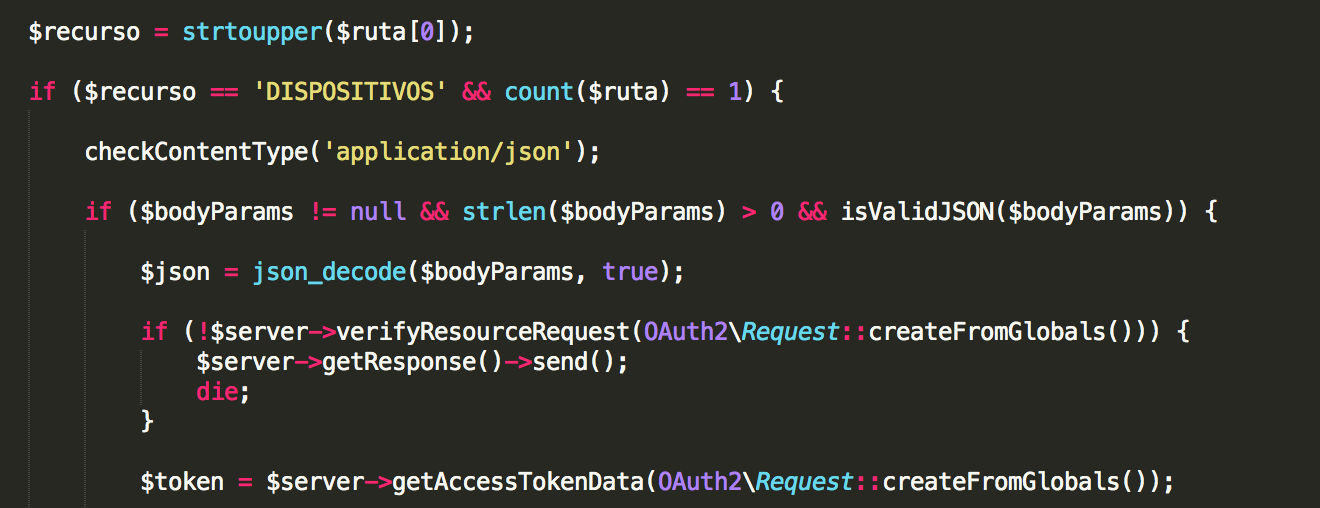


En caso de tener dos campos, el segundo sería el identificador del recurso al que se quiere hacer acceder. Comprobamos que no sea vacío y procedemos a realizar la consulta a la BD.

Cualquier otra opción de la ruta no está contemplada en nuestra API, por lo que debemos indicar el cliente que la URI a la que intenta acceder no está contemplada mediante un código de estado HTTP 404 (Not Found).

# API REST: Procesamiento del método POST

Este caso es en cierto modo algo más sencillo que el anterior, ya que en nuestra API sólo se permite el uso de este método para el recurso *Dispositivos*.



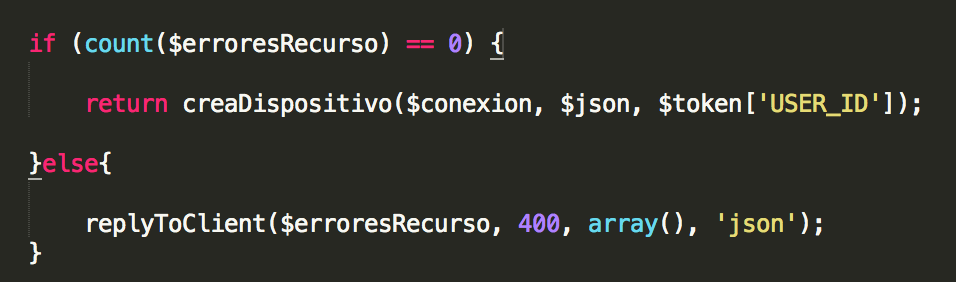
Se comienza de igual manera que en el anterior, identificando el recurso accedido. Esta vez verificamos que sea ‘*Dispositivos*’ porque para *Fabricantes* no existe esta opción.

Recordemos que necesitamos adjuntar en el cuerpo los datos del *Dispositivo* que se quiere añadir a la BD en formato JSON, por lo que debe existir una cabecera HTTP que indique el formato del contenido. Ésta es ‘*Content-Type*’, y debería ser igual a ‘*application/json*’. Una vez comprobado esto, se verifica exista contenido adjunto y que éste sea válido.

Si no lo fuese, se le indicaría al cliente mediante un código de estado HTTP 400 (Bad Request).

Si el contenido fuese válido se descodificaría y convertiría en un array asociativo, para su posterior validación.

Debemos recordar ahora que el método POST sólo puede ser accedido si se usa un token expedido por el usuario que autorice al cliente a ello. Con el uso de la librería y habiendo seguido los pasos previos, es tan sencillo como utilizar una simple sentencia de control que sirva como cortocircuito si no se tiene autorización.



Una vez comprobado, validamos los datos adjuntados. La validación es un proceso importante ya que estamos introduciendo datos en la BD que el programador no puede controlar a ciencia cierta.

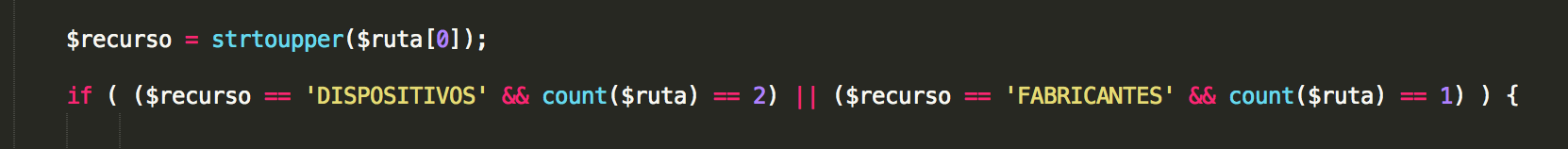
Si no existen errores, se procede a crear el *Dispositivo.*

Si existiesen, se haría saber al usuario su presencia mediante un código de estado HTTP 400 (Bad Request).

# API REST: Procesamiento del método PUT

Como este método sigue un patrón que es muy similar al anterior, nos centraremos en explicar algunas de sus diferencias.

En primer lugar el método PUT se puede realizar para ambos recursos de nuestra API y, dependiendo del que estemos intentado acceder, la longitud de la ruta será una y otra.

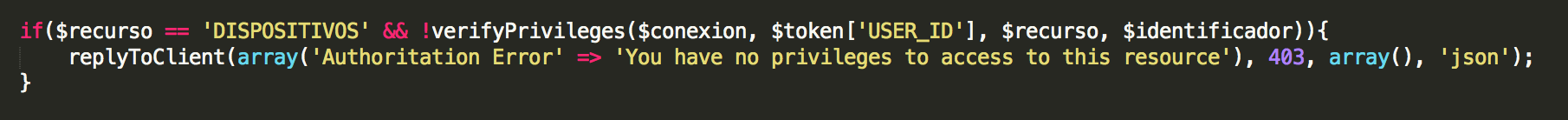


Este sería el primer paso, comprobar a qué recurso accedemos y si la longitud de la ruta concuerda con el recurso al que se está accediendo y con el contrato de la API. Si no fuera así se le haría saber al cliente mediante un código de estado HTTP 404 (Not Found), ya que esa ruta no tiene sentido en nuestro servidor.

Después de la debida comprobación de los datos adjuntos en el cuerpo y la cabecera, vamos a remarcar un detalle importante en este flujo de ejecución.

Para realizar una petición POST, bastaba con identificarse como usuario registrado en la BD con un token de acceso temporal, pues el recurso a crear sería asociado al perfil del *Fabricante* que expidió el token.

Sin embargo, en una petición PUT se debe comprobar la identidad del usuario que la realiza para ver si tiene privilegios suficientes para poder modificar esos datos. Por ejemplo, ‘Samsung Mobile’ no debería ser capaz de modificar las características de un *Dispositivo* asociado a ‘Huawei’.



Es por ello por lo que mediante una secuencia de control se verifican los privilegios del usuario asociado al token y, si no se contara con estos, se interrumpe la operación notificando al usuario mediante un mensaje y el uso del código de estado HTTP 403 (Forbidden).

Si por el contrario se tuvieran permisos, se finalizaría el procesamiento de este método procediendo con la manipulación del recurso.

# API REST: Procesamiento del método DELETE

Finalmente, procedemos con el método DELETE.



El esquema general a seguir es muy similar a lo anteriores. En primer lugar, debemos recordar que en nuestra API sólo el recurso *Dispositivos* admite este método y por lo tanto la ruta se deberá ajustar a su esquema. Es por ello que, al inicio, comprobaremos la validez de la ruta y el recurso al que se accede. Si esta inválida se le notificaría al usuario como venimos haciendo habitualmente mediante el uso del código de estado HTTP 404 (Not Found).

Este método también cuenta con acceso restringido mediante tokens y privilegios, por lo que procederemos de la misma forma que en el caso del método PUT.

Si se tuvieran permisos, se finalizaría el procesamiento de este método procediendo con la eliminación del recurso.

Si no se contara con privilegios, se interrumpe la operación notificando al usuario mediante un mensaje y el uso del código de estado HTTP 403 (Forbidden).

# API REST: Respuestas HTTP

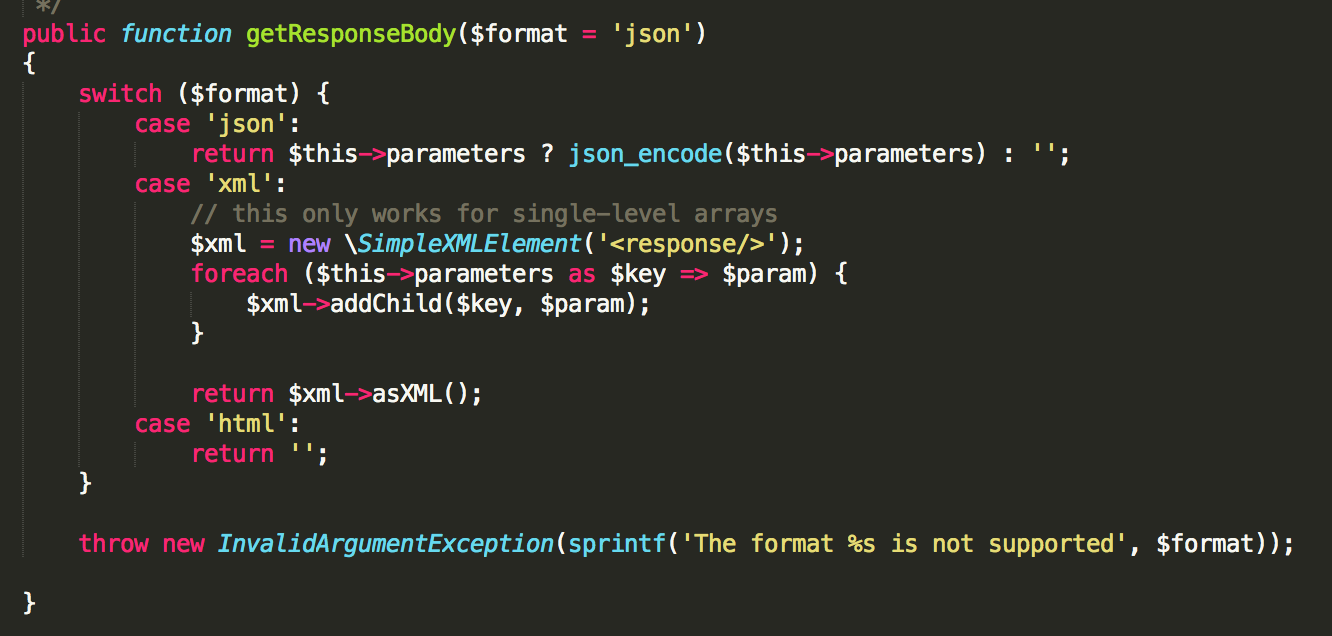
Para ser capaces de comunicarnos con el usuario debemos contar con un mecanismo que nos permita responderle cada vez que nos realiza una petición. Este mecanismo son las Respuestas HTTP, que no son más que simples mensajes que surgen como reacción a la recepción de una petición.

Para implementar las respuestas HTTP, usaremos el componente *Response.php* que incluye la librería de BShaffer con ligeras modificaciones. Este componente tiene la capacidad de construir respuestas en formato XML y JSON por defecto, pero añadiremos la posibilidad de usar el formato *text/html (Texto plano/HTML).*

En nuestro caso, nos basta con una respuesta que contenga un código de estado HTTP y el cuerpo del mensaje en HTML vacío, pues con esto ya es suficiente para que el usuario lo identifique y entienda.

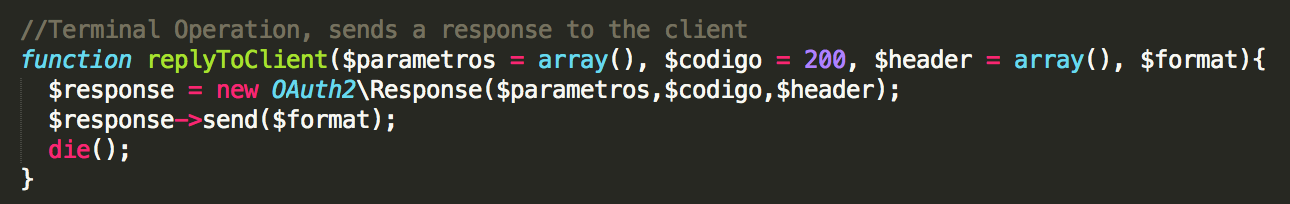


En el método send() (que se encarga de enviar la respuesta), añadiremos a la sentencia ‘*switch*’ el caso *‘html’*, que establece la cabecera ‘Content-Type’ con el valor ‘*text/html*’.



En el método getResponseBody() (que se encarga de obtener el cuerpo la respuesta), añadiremos a la sentencia ‘*switch*’ el caso *‘html’*, que devuelve un string vacío.

Ya sólo nos queda implementar un método al que podemos recurrir cada vez que necesitemos enviar una respuesta al cliente.



En este método, creamos mediante los argumentos recibidos un nuevo objeto del tipo *Response.php* y lo enviamos al cliente en el formato indicado. Una vez enviado, se debe terminar el procesamiento, por lo que invocamos al método *die().*

# API REST: Representación de los recursos en JSON

Cada vez que el usuario demanda acceso a información de la BD relacionada con los recursos, debemos tener en cuenta que el fin último será presentar estos datos al usuario en una forma clara y concisa.

JSON es un formato de texto ligero que permite el intercambio de datos y actualmente es uno de los más populares en la web (junto a XML). Por su simplicidad, facilidad de uso con PHP, aceptabilidad, ... usaremos JSON para la presentación de los recursos de cara al usuario.

Para su implementación usaremos un objeto de los definidos en el apartado anterior, estableciendo JSON como formato de la respuesta. Internamente, el objeto de la librería usa el método *json\_encode* de PHP para codificar la respuesta correctamente en este formato.

# Pruebas, ejemplos de consulta y respuestas

En esta sección veremos ejemplos de consultas a la API Rest y sus respuestas esperadas, que se usarán como pruebas para comprobar el correcto funcionamiento de la misma.

Primero debemos crear la base de datos y poblarla con información de prueba. Deberemos crear un nuevo usuario en la BD y una nueva conexión asociada a éste, con username y password ‘*MH*’ (es el que usa por defecto nuestra API PHP).

Una vez creado

# Bibliografía

[1] <http://www.slideshare.net/jmusser/open-apis-state-of-the-market-2011> por John Musser. ProgrammableWeb.

[2] <https://w3techs.com/technologies/overview/programming_language/all>

[3] <http://bshaffer.github.io/oauth2-server-php-docs/>