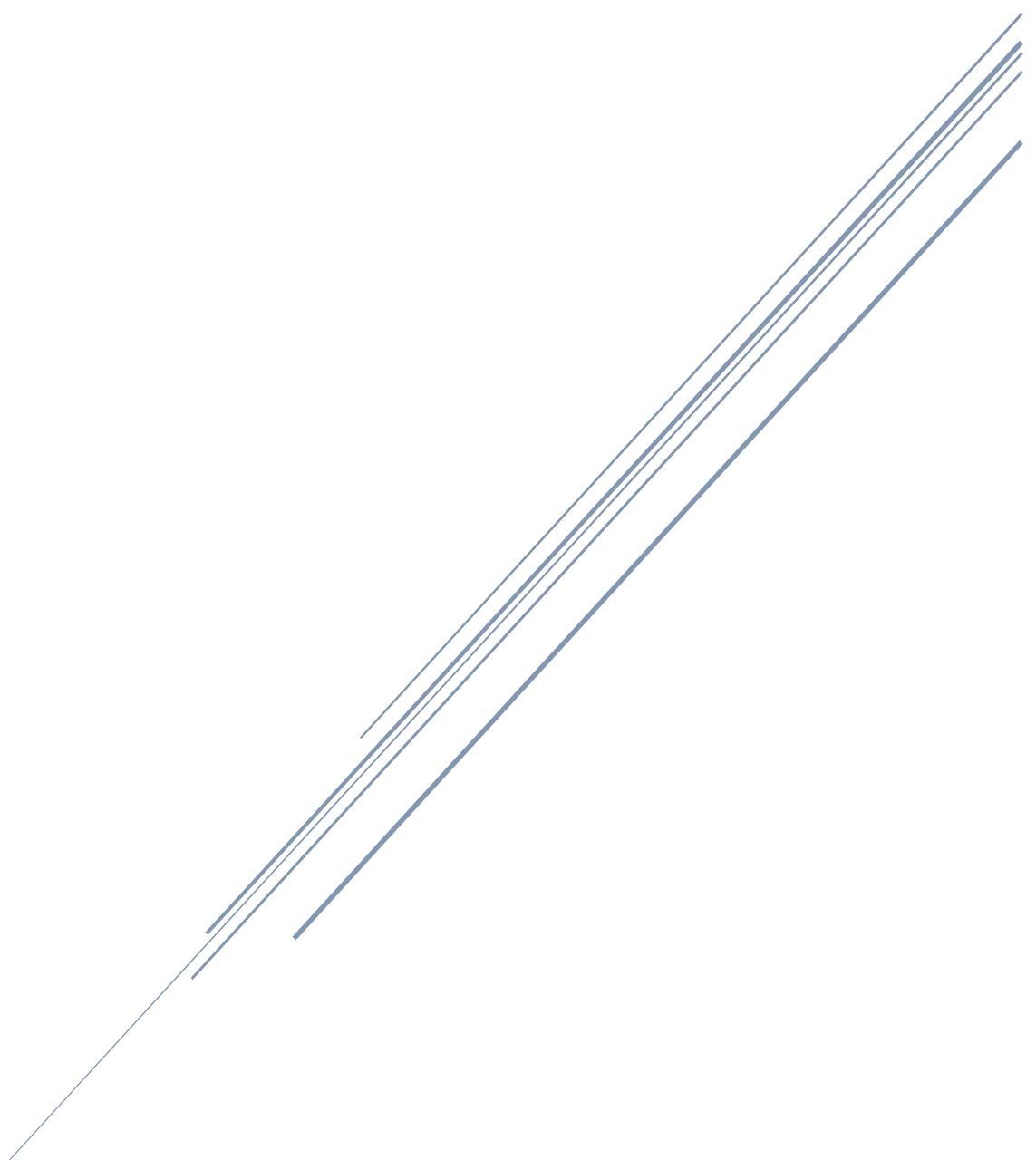


PHP AVANZADO

Implementación de API REST con PHP como back-end



José Antonio Carmona Fombella - G1 IS

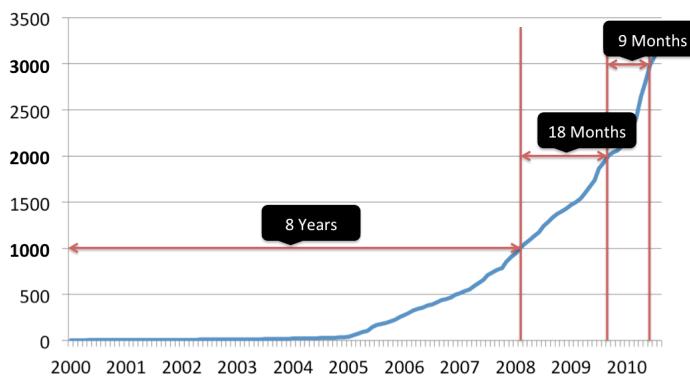
Introducción a la Ingeniería del Software y los Sistemas de Información

TABLA DE CONTENIDO

OBJETIVO, TECNOLOGÍAS Y HERRAMIENTAS DE DESARROLLO	3
ESCENARIO HIPOTÉTICO Y CONTEXTO	5
DISEÑO E IMPLEMENTACIÓN DE LA BASE DE DATOS	5
CONTRATO DE LA API REST.....	7
CONTRATO PARA EL RECURSO ‘DISPOSITIVO’	7
DISPOSITIVOS: EJEMPLO JSON CON LOS DATOS A PROPORCIONAR EN EL CUERPO	9
DISPOSITIVOS: EJEMPLO JSON CON LOS DATOS DEVUELTOS (GET)	9
CONTRATO PARA EL RECURSO ‘FABRICANTES’	10
FABRICANTES: EJEMPLO JSON CON LOS DATOS A PROPORCIONAR EN EL CUERPO.....	11
FABRICANTES: EJEMPLO JSON CON LOS DATOS DEVUELTOS (GET)	11
OAUTH 2.0	12
ADAPTACIÓN DE LA LIBRERÍA AL CONTEXTO Y ORACLE DATABASE	13
PORTAL DE EMISIÓN DE TOKENS	15
API REST: CONEXIÓN A LA BD MEDIANTE OBJETOS PDO	16
API REST: CONFIGURACIÓN DEL SERVIDOR APACHE	17
API REST: COMPROBACIÓN DE LA URI	18
API REST: COMPROBACIÓN DEL MÉTODO DE LA PETICIÓN	19
API REST: PROCESAMIENTO DEL MÉTODO GET	20
API REST: PROCESAMIENTO DEL MÉTODO POST.....	21
API REST: PROCESAMIENTO DEL MÉTODO PUT.....	22
API REST: PROCESAMIENTO DEL MÉTODO DELETE.....	23
API REST: RESPUESTAS HTTP	24
API REST: REPRESENTACIÓN DE LOS RECURSOS EN JSON	25
PRUEBAS, EJEMPLOS DE CONSULTA Y RESPUESTAS	26
BIBLIOGRAFÍA	28

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.



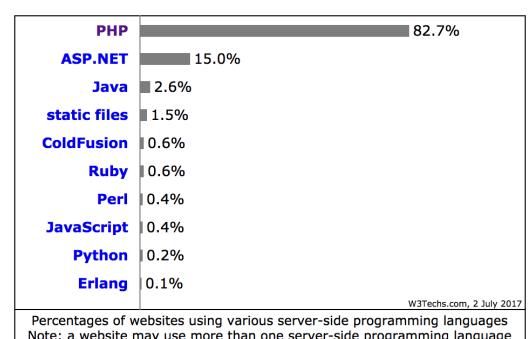
Evolución del número de APIs. En 2013 existían más de 8700 APIs. [1]

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.

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 5.6.30



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

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. [3]



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 plataforma ampliamente usada, si bien podríamos usar cualquier otra pues su función únicamente será la consulta e inserción de información.

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

- XAMPP [4]: 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 [5]: Base de datos de Oracle en su versión 11gR2.
- Oracle SQL Developer [6]: IDE para manejar la BD de Oracle.
- Regex101 [7]: Aplicación web que permite la depuración y testeo de expresiones regulares.
- Sublime Text 2 [8]: Editor de código multiplataforma con soporte para múltiples lenguajes, entre ellos SQL, PHP, HTML y CSS.
- RESTClient [9]: 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 en 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, el propio fabricante debería ser el único capaz 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:

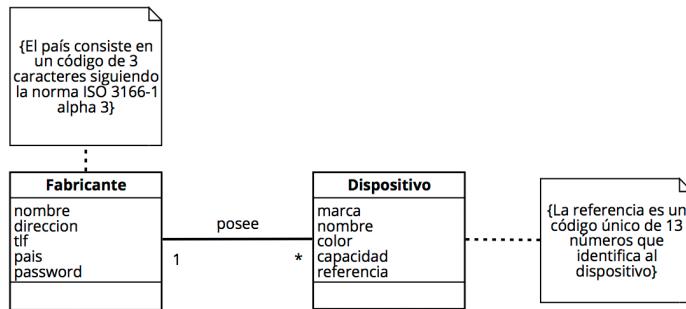
- **Nombre:** El nombre del *Fabricante*. (no se puede repetir)
- **Dirección:** La dirección del *Fabricante*.
- **Teléfono:** El número de teléfono del *Fabricante*.
- **País:** El país donde el *Fabricante* tiene su sede directiva, en formato ISO 3166-1 alpha 3.
- **Contraseña:** La contraseña que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

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 submarcas¹.
- **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.

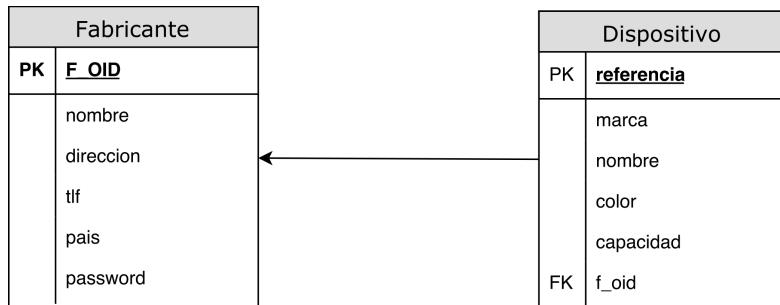
Teniendo en cuenta estos datos, realizamos un modelo conceptual con los recursos de nuestra API:

¹ Por ejemplo, el caso de Honor, la línea low-cost de Huawei.



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 SQL para construir tablas y relaciones en la BD de Oracle:

```

CREATE TABLE FABRICANTES_
(
    NOMBRE VARCHAR2(100) NOT NULL UNIQUE,
    DIRECCION VARCHAR2(200) NOT NULL,
    TLF VARCHAR2(15) NOT NULL UNIQUE,
    PAIS CHAR(3) NOT NULL,
    F_OID VARCHAR2(8) NOT NULL,
    PASSWORD VARCHAR2(100) NOT NULL,

    PRIMARY KEY(F_OID),
    CONSTRAINT CHECK_NUMERIC_TLF CHECK (REGEXP_LIKE(TLF, '^(\\+[0-9] [0-9])|(00[0-9] [0-9]))?[0-9]{3,15}$')),
    CONSTRAINT CHECK_ALPHABETIC_PAIS CHECK (REGEXP_LIKE(PAIS, '^[A-Z] [A-Z] [A-Z]$')),
    CONSTRAINT CHECK_NUMERIC_F_OID CHECK (REGEXP_LIKE(F_OID, '^[0-9]{1,8}$'))

);

CREATE TABLE DISPOSITIVOS_
(
    MARCA VARCHAR2(100) NOT NULL,
    NOMBRE VARCHAR2(100) NOT NULL,
    COLOR VARCHAR2(20) NOT NULL,
    CAPACIDAD NUMBER(5,0) NOT NULL,
    F_OID VARCHAR2(8) NOT NULL,
    REFERENCIA NUMBER(13,0) NOT NULL,

    PRIMARY KEY (REFERENCIA),
    FOREIGN KEY (F_OID) REFERENCES FABRICANTES,
    CONSTRAINT CHECK_CAPACIDAD CHECK (CAPACIDAD > 0),
    CONSTRAINT CHECK_NUMERIC_F_OID2 CHECK (REGEXP_LIKE(F_OID, '^[0-9]{1,8}$'))

);

```

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	<p>Devuelve todos los <i>dispositivos</i> de la BD en formato JSON.</p> <ul style="list-style-type: none">• 200 Successful Operation.• 404 Not Found, si no se encuentran dispositivos en la BD.	<u>limit</u> : número de resultados devueltos. Por defecto es 10. <u>offset</u> : primer elemento devuelto. Por defecto es 1.
GET	/dispositivos/{referencia}	<p>Devuelve el <i>dispositivo</i> con la referencia en formato JSON.</p> <ul style="list-style-type: none">• 200 Successful Operation, si el dispositivo se ha encontrado.• 404 Not Found, si no hay ningún dispositivo con tal referencia.	Ninguno.
POST	/dispositivos	<p>Añade el <i>dispositivo</i> que se pasa por el cuerpo en formato JSON.</p> <ul style="list-style-type: none">• 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.	<u>token</u> : código temporal que autoriza la operación. <u>JSON en el cuerpo</u> con los datos del dispositivo.

		<p>Actualiza el <i>dispositivo</i> que tiene la <i>referencia</i> usando los valores que se envían en el cuerpo en formato JSON.</p> <ul style="list-style-type: none"> • 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. 	<p><i>token</i>: código temporal que autoriza la operación.</p> <p><u>JSON en el cuerpo</u> con los datos del dispositivo a actualizar.</p>
DELETE	/dispositivos/{referencia}	<p>Borra un <i>dispositivo</i> dado su <i>referencia</i>.</p> <ul style="list-style-type: none"> • 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. 	<p><i>token</i>: código temporal que autoriza la operación.</p>

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

POST y PUT:

```
{  
    "REFERENCIA": "10000000000000",  
    "MARCA": "Apple",  
    "NOMBRE": "iPhone 7 Plus",  
    "COLOR": "Jet Black",  
    "CAPACIDAD": "128",  
}
```

Dispositivos: Ejemplo JSON con los datos devueltos (GET)

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

```
[  
    {  
        "Total Recursos": 11,  
        "Resultados Consulta": 2  
    },  
    [  
        {  
            "RNUM": "1",  
            "MARCA": "Apple",  
            "NOMBRE": "iPhone 7 Plus",  
            "COLOR": "Jet Black",  
            "CAPACIDAD": "128",  
            "F_OID": "1",  
            "REFERENCIA": "100000000000"  
        },  
        {  
            "RNUM": "2",  
            "MARCA": "Apple",  
            "NOMBRE": "iPad Air 2",  
            "COLOR": "Space Gray",  
            "CAPACIDAD": "64",  
            "F_OID": "1",  
            "REFERENCIA": "100000000001"  
        }  
    ]  
]
```

Contrato para el recurso 'Fabricantes'

Recurso: *Fabricantes*

HTTP	URI	DESCRIPCIÓN Y CÓDIGOS	PARÁMETROS
GET	/fabricantes	<p>Devuelve todos los <i>fabricantes</i> de la BD en formato JSON.</p> <ul style="list-style-type: none"> • 200 Successful Operation. • 404 Not Found, si no se encuentran fabricantes en la BD. 	<p>limit: número de resultados devueltos. Por defecto es 10.</p> <p>offset: primer elemento devuelto. Por defecto es 1.</p>
GET	/fabricantes/{f_oid}	<p>Devuelve el <i>fabricante</i> con la <i>f_oid</i> en formato JSON.</p> <ul style="list-style-type: none"> • 200 Successful Operation, si el fabricante se ha encontrado. • 404 Not Found, si no hay ningún fabricante con tal <i>f_oid</i>. 	Ninguno.
PUT	/fabricantes	<p>Actualiza el <i>fabricante</i> del token actual, usando los valores que se envían en el cuerpo en formato JSON.</p> <ul style="list-style-type: none"> • 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. 	<p><i>token</i>: código temporal que autoriza la operación.</p> <p>JSON en el cuerpo con los datos del dispositivo a actualizar.</p>

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

PUT:

```
{  
    "NOMBRE": "Apple Inc.",  
    "DIRECCION": "1 Infinite Loop, California",  
    "PAIS": "USA",  
    "TLF": "+1987654320"  
}
```

Fabricantes: Ejemplo JSON con los datos devueltos (GET)

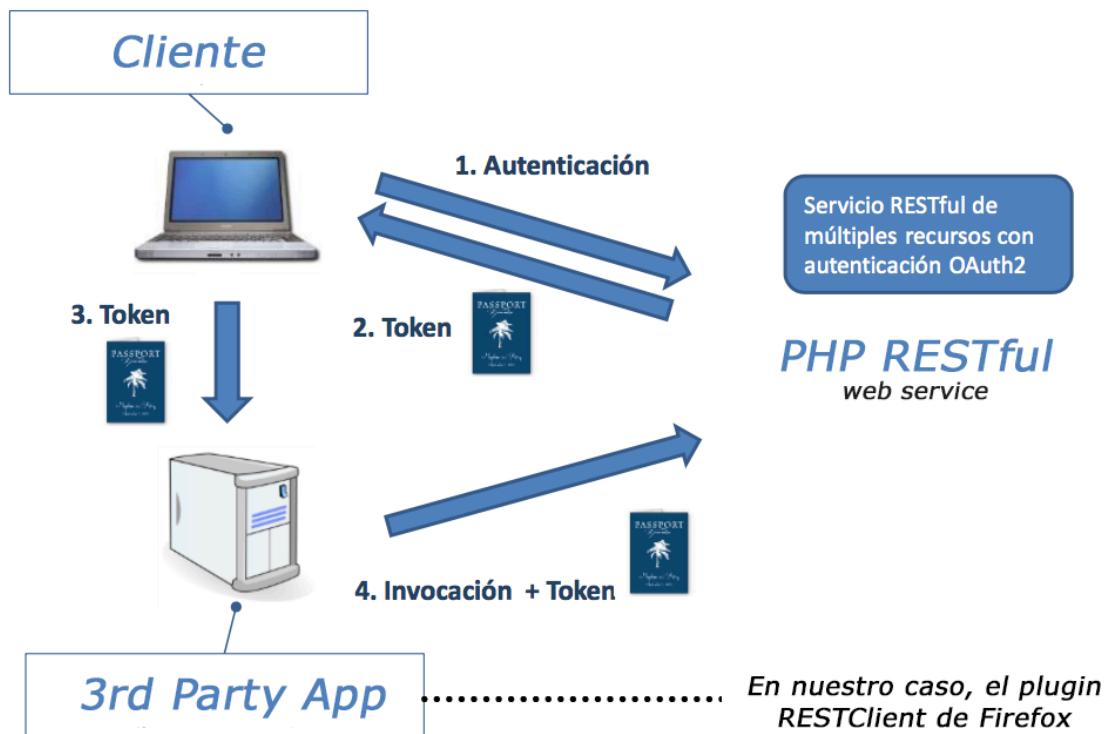
Resultado de la consulta a la URI: http://localhost/PHP_API/fabricantes?limit=2

```
[  
    {  
        "Total Recursos": 5,  
        "Resultados Consulta": 2  
    },  
    [  
        {  
            "RNUM": "1",  
            "NOMBRE": "Apple Inc.",  
            "F_OID": "1",  
            "DIRECCION": "1 Infinite Loop, California",  
            "PAIS": "USA",  
            "TLF": "+1987654320"  
        },  
        {  
            "RNUM": "2",  
            "NOMBRE": "BQ",  
            "F_OID": "2",  
            "DIRECCION": "Las Rozas, Madrid",  
            "PAIS": "ESP",  
            "TLF": "+34911829384"  
        }  
    ]  
]
```

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 [9], el cual nos 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 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 dichas 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, aunque 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, usando una variable de sesión y modificando un poco la parte del código encargada guardarla en la BD.

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

```
public function getUserId()
{
    $this->loadClientData();

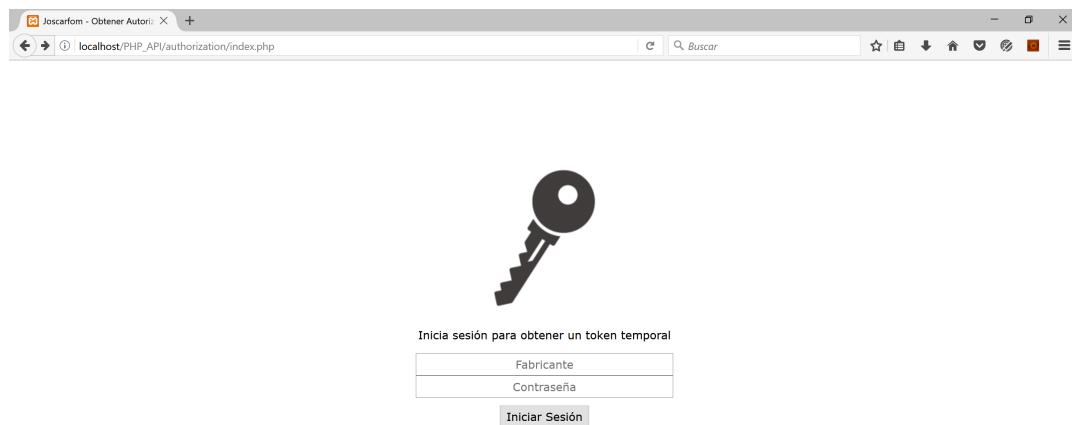
    return isset($_SESSION['user_id_passthrough']) ?
        $_SESSION['user_id_passthrough'] : ( isset($this-
            >clientData['USER_ID']) ? $this->clientData['USER_ID'] : null );
}
```

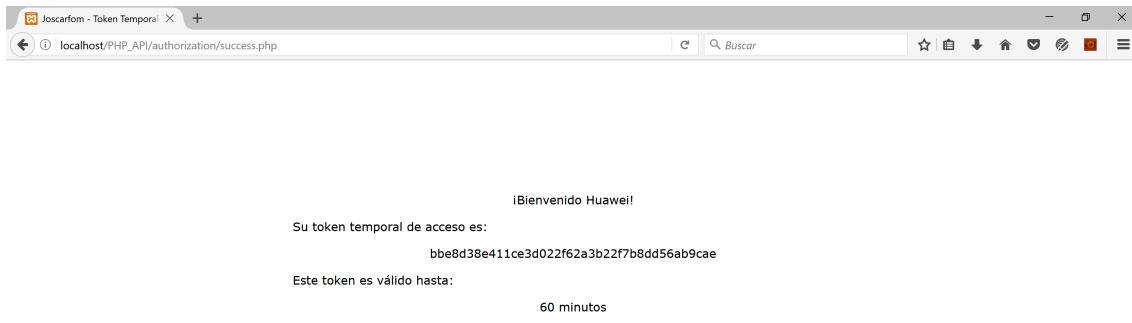
Insertando esta sentencia en el código original conseguiremos que, si existiera un identificador de *Fabricante* en la sesión, se use éste. Si no fuera así, se continuaría con el flujo de operación normal de la librería.

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 lugar 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.





Cabe destacar que la aplicación de terceros tendría que identificarse de alguna forma y estar registrada en nuestra BD para así restringir el uso de esta función a sólo un conjunto limitado y preprobado de sitios.

Esto se consigue mediante el uso de dos datos (*clientId* y *clientSecret*) que serán comprobados al inicio del proceso y que la aplicación de 3ros deberá adjuntar. En nuestro ejemplo, suponemos que la página principal del login ha recibido ya estos datos correctamente.

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

API REST: Conexión a la BD mediante Objetos PDO

a

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.

```
1  <?php
2
3      require_once "utilidades.php";
4
5      function crearConexionBD()
6      {
7          $host = "oci:dbname=localhost/XE;charset=UTF8";
8          $usuario = "MH";
9          $password = "MH";
10
11         try{
12
13             $conexion = new PDO($host,$usuario,$password,array(PDO::ATTR_PERSISTENT => true));
14             $conexion->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
15
16             return $conexion;
17
18         }catch(PDOException $e){
19
20             $_SESSION['excepcion'] = $e->getMessage();
21             replyToClient(array(),500,array(), 'html');
22
23         }
24
25     }
26
27     function cerrarConexionBD($conexion){
28         $conexion = null;
29
30     }
31 ?>
```

En él intentamos crear un objeto PDO con los datos de acceso a nuestra BD y establecemos el modo de error y de excepciones. Si todo transcurre 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 URLs que en nuestro servidor no corresponden a ningún archivo, por lo que su respuesta predeterminada será informar de ello al usuario.

Por ejemplo, para acceder 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 [10]:

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 file).
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 directory).
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 se redirige la URI a la que se ha intentado acceder en forma de 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 de la petición 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 fichero 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.

```
//Obtenemos la ruta a la que se está intentando acceder por medio del parámetro req_path (si está definido)
$req_path = (isset($_GET['req_path'])) ? $_GET['req_path'] : null;

if ($req_path == null) {
    replyToClient(array(), 400, array(), 'html');
}

$ruta = explode('/', $req_path);

//Ahora tenemos la ruta en un array asociativo
$recursoAccedido = $ruta[0];

//Comprobamos que estamos accediendo a uno de los recursos disponibles en nuestra API
if(!in_array($recursoAccedido, $recursos)){
    replyToClient(array(), 404, array(), 'html');
}
```

El primer paso es verificar 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.

```
//Obtenemos el método de la petición
$metodo = strtolower($_SERVER['REQUEST_METHOD']);

//Procesamos ahora la petición dependiendo del método
$conexion = crearConexionBD();

switch ($metodo) {
    case 'get':
        ...
        break;
    case 'post':
        ...
        break;
    case 'put':
        ...
        break;
    case 'delete':
        ...
        break;
    default:
        replyToClient(array(), 405, array(), 'html');
}

cerrarConexionBD($conexion);
```

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 negativo le indicamos al cliente que éste 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.

```
$recurso = strtoupper($ruta[0]);
$campos = ($recurso == 'DISPOSITIVOS') ? '*' : 'NOMBRE, F_OID, DIRECCION, PAIS, TLF' ;

if (count($ruta) == 1) {
    $resultado = null;
    if(isset($parametros['limit']) || isset($parametros['offset'])){
        $limit = isset($parametros['limit']) ? $parametros['limit'] : 10;
        $offset = isset($parametros['offset']) ? $parametros['offset'] : 1;
        $parametrosValidados = validarLimitOffset($limit, $offset);
        $resultado = consultaRecursosPaginado($conexion, $recurso, $parametrosValidados['offset'], $parametrosValidados['limit'], $campos);
    }else{
        $resultado = consultaRecursosPaginado($conexion, $recurso, 1, 10, $campos);
    }
}
```

Se comienza por determinar 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 tener un solo campo, se prosigue con la identificación y validación de los parámetros que pudieran estar presentes. 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 .

```
} else if(count($ruta) == 2 && $ruta[1] != ''){
    $identificador = $ruta[1];
    $resultado = null;
    $resultado = consultaRecurso($conexion, $recurso, $identificador, $campos);
    return $resultado;
}

}else{
    replyToClient(array(),404,array(), 'html');
}
}
```

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 en nuestro servidor 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*.

```
$recurso = strtoupper($ruta[0]);  
  
if ($recurso == 'DISPOSITIVOS' && count($ruta) == 1) {  
  
    checkContentType('application/json');  
  
    if ($bodyParams != null && strlen($bodyParams) > 0 && isValidJSON($bodyParams)) {  
  
        $json = json_decode($bodyParams, true);  
  
        if (!$server->verifyResourceRequest(0Auth2\Request::createFromGlobals())) {  
            $server->getResponse()->send();  
            die;  
        }  
  
        $token = $server->getAccessTokenData(0Auth2\Request::createFromGlobals());
```

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 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.

```

if (count($erroresRecurso) == 0) {
    return creaDispositivo($conexion, $json, $token['USER_ID']);
} else{
    replyToClient($erroresRecurso, 400, array(), 'json');
}

```

Una vez comprobado, validamos los datos adjuntos. 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 de 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 intentando acceder, la longitud de la ruta será una u otra.

```

$recurso = strtoupper($ruta[0]);
if ( ($recurso == 'DISPOSITIVOS' && count($ruta) == 2) || ($recurso == 'FABRICANTES' && count($ruta) == 1) ) {

```

Este sería pues el primer paso, comprobar a qué recurso accedemos y si la longitud de la ruta concuerda con la especificada en 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’.

```
if($recurso == 'DISPOSITIVOS' && !verifyPrivileges($conexion, $token['USER_ID'], $recurso, $identificador)){
    replyToClient(array('Authoritation Error' => 'You have no privileges to access to this resource'), 403, array(), 'json');
}
```

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 notificándole 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.

```
$recurso = strtoupper($ruta[0]);
if ($recurso == 'DISPOSITIVOS' && count($ruta) == 2) {
    if (!$server->verifyResourceRequest(0Auth2\Request::createFromGlobals())) {
        $server->getResponse()->send();
        die;
    }
    $token = $server->getAccessTokenData(0Auth2\Request::createFromGlobals());
    if(!verifyPrivileges($conexion, $token['USER_ID'], $recurso, $ruta[1])){
        replyToClient(array('Authoritation Error' => 'You have no privileges to access to this resource'), 403, array(), 'json');
    }
    $resultado = eliminaRecurso($conexion, $recurso, $ruta[1]);
    return $resultado;
} else {
    replyToClient(array(), 404, array(), 'html');
}
```

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 misma y el recurso al que se accede. Si ésta fuera 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 con la eliminación del recurso.

Si no se contara con privilegios, se interrumpiría 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 de denomina Respuestas HTTP, que no son más que simples mensajes que surgen como reacción a la recepción de una petición.

Para su implementación, usaremos el componente *Response.php* que incluye la librería de BShaffer con algunas modificaciones ligeras. Este módulo 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 entienda.

```
    public function send($format = 'json') {
        // headers have already been sent by the developer
        if (headers_sent()) {
            return;
        }

        switch ($format) {
            case 'json':
                $this->setHttpHeader('Content-Type', 'application/json');
                break;
            case 'xml':
                $this->setHttpHeader('Content-Type', 'text/xml');
                break;
            case 'html':
                $this->setHttpHeader('Content-Type', 'text/html');
                break;
        }
        // status
        header(sprintf('HTTP/%s %s %s', $this->version, $this->statusCode, $this->statusText));

        foreach ($this->getHttpHeaders() as $name => $header) {
            header(sprintf('%s: %s', $name, $header));
        }
        echo $this->getResponseBody($format);
    }
```

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*'.

```

/*
public function getResponseBody($format = 'json')
{
    switch ($format) {
        case 'json':
            return $this->parameters ? json_encode($this->parameters) : '';
        case 'xml':
            // this only works for single-level arrays
            $xml = new \SimpleXMLElement('<response/>');
            foreach ($this->parameters as $key => $param) {
                $xml->addChild($key, $param);
            }

            return $xml->asXML();
        case 'html':
            return '';
    }

    throw new InvalidArgumentException(sprintf('The format %s is not supported', $format));
}

```

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.

```

//Terminal Operation, sends a response to the client
function replyToClient($parametros = array(), $codigo = 200, $header = array(), $format){
    $response = new OAuth2\Response($parametros,$codigo,$header);
    $response->send($format);
    die();
}

```

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 a la función `die()` de PHP.

API REST: Representación de los recursos en JSON

Cada vez que el cliente 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.

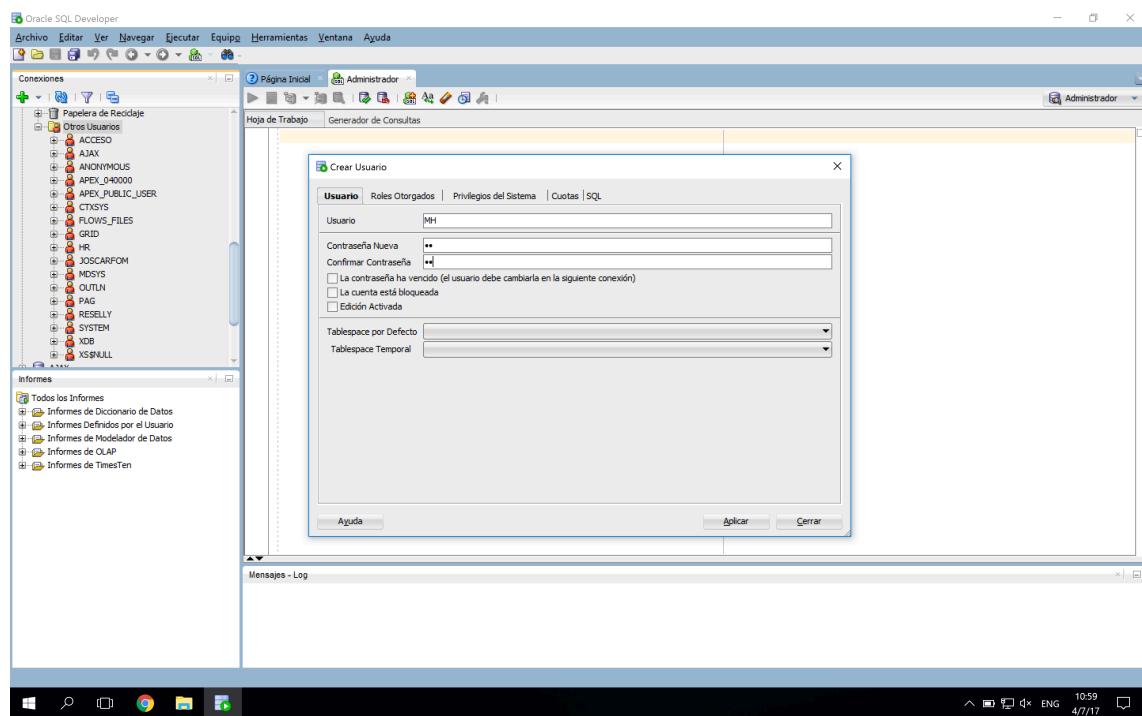
Para su implementación usaremos un objeto de los definidos en el apartado anterior, estableciendo JSON como formato de la respuesta. Internamente, el módulo de la

librería usa el método `json_encode()` de PHP para codificar la respuesta correctamente en este formato.

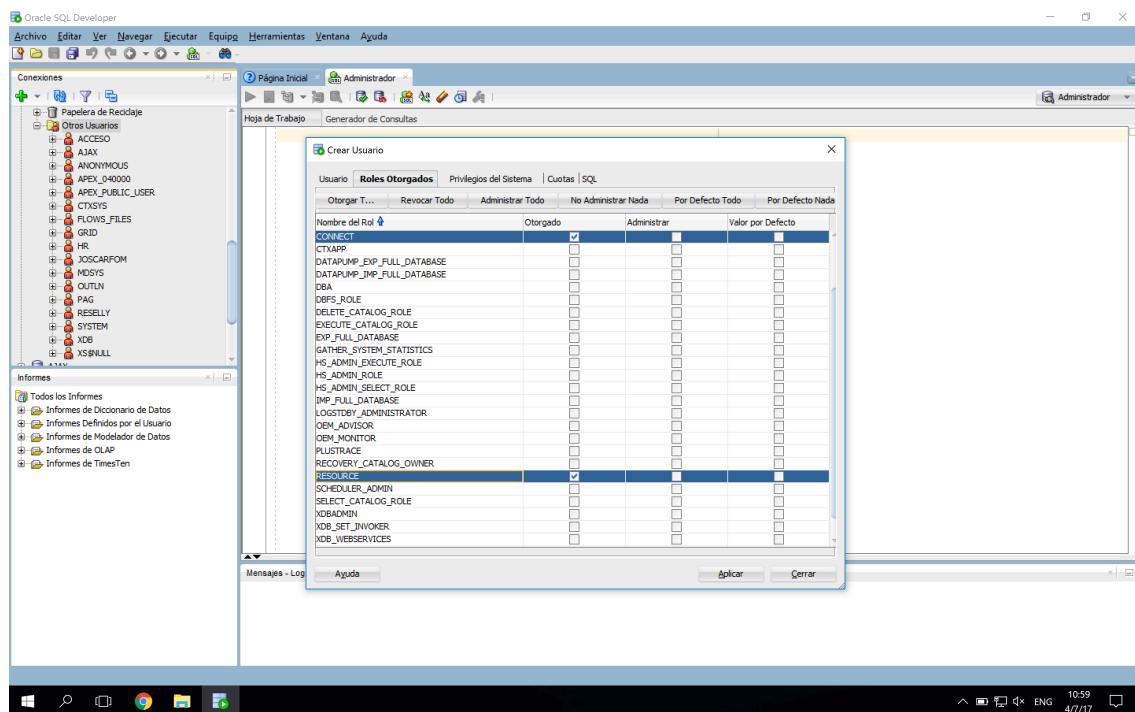
Pruebas, ejemplos de consulta y respuestas

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.

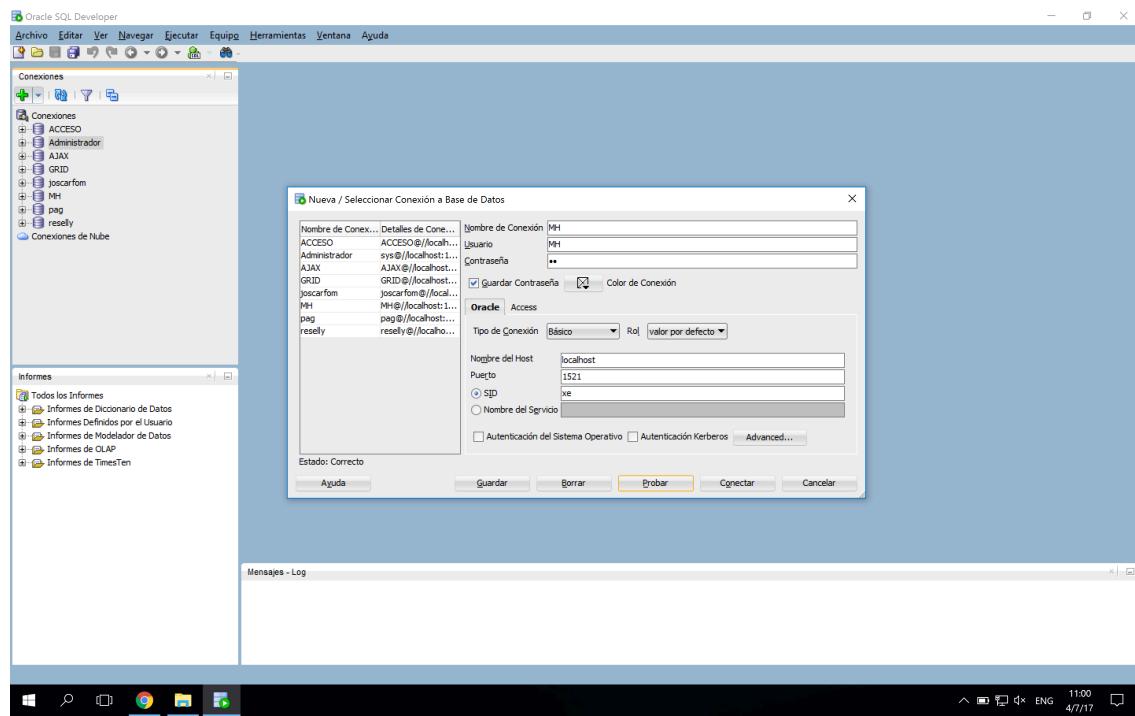
Primero debemos crear la base de datos y poblarla con información de prueba. Deberemos crear un nuevo usuario en la BD y una nueva conexión asociada a éste, con username y password ‘MH’ (es el que usa por defecto nuestra API PHP).



1 - Creación del Usuario ‘MH’ en la BD



2 - Debemos recordar otorgarle los roles Connect y Resource.



3 – Creación de la Conexión a la BD con el usuario 'MH'

Una vez completado, ejecutamos los scripts encontrados en la carpeta SQL (*definicion.sql* y *datos.sql*) en SQL Developer para crear las tablas y poblarlas con datos de prueba.

Una vez ejecutados, creamos una carpeta en el directorio htdocs de XAMPP con el nombre ‘PHP_API’ y procedemos a iniciar el programa.

Iniciamos Firefox e iniciamos el plugin RESTClient (debemos instalarlo previamente si no disponemos de él).

Prueba 1: GET de todos los Dispositivos/Fabricantes

El objetivo de esta prueba será el de obtener todos los recursos (en este caso en concreto, *Dispositivos*) disponibles en la BD (10 *Dispositivos* en total). Mirando en el contrato de la API, podemos ver que no se necesita autenticación.

Parámetros: Ninguno

URI: http://localhost/PHP_API/dispositivos

The screenshot shows the RESTClient interface in a browser window. The request section has 'Method' set to 'GET' and 'URL' set to 'http://localhost/PHP_API/dispositivos'. The response section displays a JSON array of 10 device objects:

```
1. [
2.   {
3.     "Total Recursos": 10,
4.     "Resultados Consulta": 10
5.   },
6.   [
7.     [
8.       {
9.         "RNUM": "1",
10.        "MARCA": "Apple",
11.        "NOMBRE": "iPhone 7 Plus",
12.        "COLOR": "Jet Black",
13.        "CAPACIDAD": "128",
14.        "F OID": "1",
15.        "REFERENCIA": "1000000000000000"
16.      },
17.      {
18.        "RNUM": "2",
19.        "MARCA": "Apple",
20.        "NOMBRE": "iPad Air 2",
21.        "COLOR": "Space Gray",
22.        "CAPACIDAD": "64"
```

Podemos comprobar que la prueba se ha completado con éxito y que se nos devuelven los 10 *Dispositivos*, así como un contador del número que existen en la BD y los devueltos por la consulta.

Prueba 2: GET de todos los Dispositivos/Fabricantes (con parámetros)

Mediante el uso de los parámetros predefinidos en el contrato para esta consulta, se soporta la paginación. Por defecto el número de recursos devueltos es 10 y el offset es 1, pero podemos especificar cualquier otro valor de forma manual.

En esta prueba, indicaremos que queremos 2 *Dispositivos* como máximo, empezando por el 3. Para ello usaremos parámetros en la URI.

Parámetros: limit=2, offset=3

URI: http://localhost/PHP_API/dispositivos?limit=2&offset=3

The screenshot shows the RESTClient interface. In the 'Request' section, the method is set to 'GET', the URL is 'IP_API/dispositivos?limit=2&offset=3', and the 'SEND' button is visible. In the 'Body' section, there is a 'Request Body' field containing the following JSON:

```
1. [
2.   {
3.     "Total Recursos": 10,
4.     "Resultados Consulta": 2
5.   },
6.   [
7.     {
8.       "RNUM": "3",
9.       "MARCA": "BQ",
10.      "NOMBRE": "Aquaris U",
11.      "COLOR": "Blanco",
12.      "CAPACIDAD": "8GB",
13.      "F_OID": "2",
14.      "REFERENCIA": "20000000000000"
15.    },
16.    {
17.      "RNUM": "4",
18.      "MARCA": "BQ",
19.      "NOMBRE": "Aquaris M",
20.      "COLOR": "Negro",
21.      "CAPACIDAD": "16GB"
22.    }
23.  ]
24. }
```

In the 'Response' section, the 'Response Body (Raw)' tab is selected, showing the same JSON structure. The 'Response Headers' tab is also visible.

Efectivamente comprobamos cómo la API nos devuelve 2 resultados, empezando por el *Dispositivo 3* en la BD. Los campos 'Resultado Consulta' y 'RNUM' nos indican fácil y rápidamente estos datos.

Prueba 3: GET de todos los Dispositivos/Fabricantes (con parámetros erróneos)

Nuestra API debe estar preparada para la recepción de datos erróneos, proporcionando una solución si se diera el caso. En esta prueba comprobaremos que, si en los parámetros usados para paginar se introdujeseen datos no válidos, la API devolverá el resultado de una consulta con los valores por defecto (1 para el offset y 10 para el limit).

Parámetros: limit=-1, offset=absj

URI: http://localhost/PHP_API/dispositivos?limit=-1&offset=absj

```

1. [
2.   {
3.     "Total Recursos": 10,
4.     "Resultados Consulta": 10
5.   },
6.   [
7.     {
8.       "RNUM": "1",
9.       "MARCA": "Apple",
10.      "NOMBRE": "iPhone 7 Plus",
11.      "COLOR": "Jet Black",
12.      "CAPACIDAD": "128",
13.      "FE_OID": "1",
14.      "REFERENCIA": "1000000000000000"
15.    },
16.    {
17.      "RNUM": "2",
18.      "MARCA": "Apple",
19.      "NOMBRE": "iPad Air 2",
20.      "COLOR": "Space Gray",
21.      "CAPACIDAD": "64"
22.    }
23.  ]
24. ]

```

Podemos observar cómo se obtiene una respuesta idéntica a la de la Prueba 1, donde también se usaban parámetros por defecto.

Prueba 4: GET de todos los Dispositivos/Fabricantes (sin datos en la BD)

Si volvemos a ejecutar el script '*definicion.sql*' nuestras tablas estarán vacías y no existirán recursos en la BD. En este caso, el contrato estipula que se debería devolver un código de estado 404.

URI: http://localhost/PHP_API/dispositivos

```

1. Status Code : 404 Not Found
2. Connection : Keep-Alive
3. Content-Length : 66
4. Content-Type : application/json
5. Date : Tue, 04 Jul 2017 10:36:43 GMT
6. Keep-Alive : timeout=5, max=100
7. Server : Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
8. X-Powered-By : PHP/5.6.30

```

Además, si vemos el cuerpo de la respuesta podremos observar que se nos adjunta un mensaje que nos informa que no se han podido encontrar resultados.

Prueba 5: GET de un Dispositivo/Fabricante

Podemos obtener un recurso en concreto pasando su referencia a través de la URI, como nos indica el contrato.

URI: http://localhost/PHP_API/dispositivos/10000000000001

The screenshot shows the RESTClient interface. In the 'Request' section, the method is set to 'GET' and the URL is 'IP_API/dispositivos/10000000000001'. The 'Response' section displays the JSON response:

```
1. [
2.   {
3.     "Total Recursos": 10,
4.     "Resultados Consulta": 1
5.   },
6.   [
7.     {
8.       "MARCA": "Apple",
9.       "NOMBRE": "iPad Air 2",
10.      "COLOR": "Space Gray",
11.      "CAPACIDAD": "64",
12.      "P_OID": "1",
13.      "REFERENCIA": "10000000000001"
14.    }
15. ]
```

Ahora, podemos verificar que el *Dispositivo* devuelto coincide con el que se lista en los resultados totales y tiene su misma referencia.

Prueba 6: GET de un Dispositivo/Fabricante (referencia errónea)

Si la referencia de un recurso no existe, el servidor deberá indicárnoslo mediante el uso del código de estado HTTP 404. Además, en el cuerpo del mensaje se nos mostrará una advertencia.

URI: http://localhost/PHP_API/dispositivos/abcdef

The screenshot shows the RESTClient extension in a browser window. The request method is set to GET, and the URL is `http://localhost/PHP_API/dispositivos/abcdef`. The response pane displays the following headers:

```
1. Status Code : 404 Not Found
2. Connection : Keep-Alive
3. Content-Length : 66
4. Content-Type : application/json
5. Date : Tue, 04 Jul 2017 10:50:16 GMT
6. Keep-Alive : timeout=5, max=100
7. Server : Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
8. X-Powered-By : PHP/5.6.30
```

Prueba 7: POST de un Dispositivo

Este método requiere autenticación ya que vamos a crear un recurso en la BD, por lo que usaremos el formulario que vimos anteriormente para obtener las credenciales.

Para crear un recurso, debemos proporcionarlo en el cuerpo en formato JSON y estableces el atributo *Content-Type* a *application/json*.

The screenshot shows the RESTClient extension in a browser window. The request method is set to POST, and the URL is `http://localhost/PHP_API/dispositivos`. The Headers section contains `Content-Type: application/json`. The Body section contains the JSON payload: `{"MARCA":"Apple","NOMBRE":"iPhone X Edition","COLOR":"Deep Indigo","CAPACIDAD":"64","REFERENCIA":"100000000002"}`. The response pane displays the following headers:

```
1. Status Code : 201 Created
2. Connection : Keep-Alive
3. Content-Length : 0
4. Content-Type : text/html; charset=UTF-8
5. Date : Tue, 04 Jul 2017 11:17:00 GMT
6. Keep-Alive : timeout=5, max=100
7. Server : Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
8. X-Powered-By : PHP/5.6.30
```

Dispositivo:

```
{ "MARCA": "Apple", "NOMBRE": "iPhone X Edition", "COLOR": "Deep Indigo", "CAPACIDAD": "64", "REFERENCIA": "1000000000002" }
```

URI: http://localhost/PHP_API/dispositivos?access_token=TOKEN

Obtenemos el código 201 Created y comprobamos en la BD que efectivamente se ha creado el dispositivo.

Prueba 8: POST de un Dispositivo (con datos en cuerpo erróneos)

En caso de adjuntar un recurso que no esté en formato JSON o con campos erróneos, se deberá informar al usuario con ello mediante un código de estado HTTP 400 Bad Request.

The screenshot shows the RESTClient interface. In the 'Request' section, the method is set to 'POST', the URL is 'http://localhost/PHP_API/dispositivos', and the body contains the JSON string: {"MARCA": "Apple", "NOMBRE": "", "COLOR": "Deep Indigo", "CAPACIDAD": "64", "REFERENCIA": "1000a00000003"}. In the 'Response' section, the status is 400 Bad Request, and the response body is: [1. [2. "El nombre del dispositivo no debe estar vacío",3. "La referencia del dispositivo debe ser numérica"]].

Dispositivo Erróneo:

```
{ "MARCA": "", "NOMBRE": "iPhone X Edition", "COLOR": "Deep Indigo", "CAPACIDAD": "64", "REFERENCIA": "10000a0000003" }
```

En la pestaña de Respuestas confirmamos que el código de estado HTTP es 400 Bad Request y posteriormente, que no se han producido alteraciones en la BD.

Prueba 9: POST de un Dispositivo (sin token)

En caso de no incluir un token a la hora de crear el recurso (o incluir uno inválido), se debería elevar el código de estado 401 Unauthorized.

Token inválido: ed773a27c86dff1a0975bfe61546475df83012.9

The screenshot shows the RESTClient application window. In the 'Request' section, the method is set to 'POST', the URL is '[:86dff1a0975bfe61546475df83012.9](#)', and the body contains the following JSON:

```
{"MARCA": "Apple", "NOMBRE": "iPhone X Edition", "COLOR": "Deep Indigo", "CAPACIDAD": "64", "REFERENCIA": "100000000002"}
```

In the 'Response' section, the status code is 401 Unauthorized, and the response body is:

```
1. Status Code : 401 Unauthorized
2. Cache-Control : no-store
3. Connection : Keep-Alive
4. Content-Length : 84
5. Content-Type : application/json
6. Date : Tue, 04 Jun 2017 11:41:00 GMT
7. Keep-Alive : timeout=5, max=100
8. Server : Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
9. WWW-Authenticate :
   Bearer realm="Service", error="invalid_token", error_description="The access token provided is invalid"
10. X-Powered-By : PHP/5.6.30
```

En la pestaña de la respuesta del cuerpo, podemos comprobar que se nos remite un pequeño mensaje indicando que el token no es válido.

Prueba 10: PUT de Dispositivo/Fabricante

Comenzaremos actualizando un recurso correctamente. En nuestro caso, al estar probando el recurso *Dispositivos*, usaremos para ello un token del fabricante asociado con el mismo y un dispositivo existente.

URI: http://localhost/PHP_API/dispositivos/100000000002?access_token=TOKEN

The screenshot shows the RESTClient extension in a browser window. The request section has a PUT method and the URL `http://localhost/PHP_API/dispositivo/1`. The body contains the JSON object: `{"MARCA": "Apple", "NOMBRE": "iPad Pro 2018", "COLOR": "Deep Indigo", "CAPACIDAD": "512", "REFERENCIA": "100000000002"}`. The response section shows the following headers:

```
1. Status Code : 204 No Content
2. Connection : Keep-Alive
3. Content-Type : text/html; charset=UTF-8
4. Date : Tue, 04 Jul 2017 11:52:53 GMT
5. Keep-Alive : timeout=5, max=100
6. Server : Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
7. X-Powered-By : PHP/5.6.30
```

Nuevo Dispositivo:

```
{ "MARCA": "Apple", "NOMBRE": "iPad Pro 2018", "COLOR": "Deep Indigo", "CAPACIDAD": "512", "REFERENCIA": "100000000002" }
```

Podemos comprobar que el código es correcto y que se ha actualizado correctamente realizando un GET.

Prueba 11: PUT de Dispositivo/Fabricante (token inválido)

Si adjuntamos un token inválido, se nos debe mostrar el código 401 Unauthorized.

The screenshot shows the RESTClient extension in a browser window. The request section has a PUT method and the URL `>86dff1a0975bfe61546475df83012.9`. The body contains the JSON object: `{"MARCA": "Apple", "NOMBRE": "iPad Pro 2018", "COLOR": "Deep Indigo", "CAPACIDAD": "512", "REFERENCIA": "100000000002"}`. The response section shows the following headers:

```
1. Status Code : 401 Unauthorized
2. Cache-Control : no-store
3. Connection : Keep-Alive
4. Content-Length : 84
5. Content-Type : application/json
6. Date : Tue, 04 Jul 2017 12:01:09 GMT
7. Keep-Alive : timeout=5, max=100
8. Server : Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
9. WWW-Authenticate : Bearer realm="Service", error="invalid_token", error_description="The access token provided is invalid"
10. X-Powered-By : PHP/5.6.30
```

Token inválido: ed773a27c86dff1a0975bfe61546475df83012.9

Prueba 12: PUT de Dispositivo/Fabricante (token expedido por otro)

Si se adjuntara un token que es válido pero pertenece a otro *Fabricante* (que no tiene asociado el *Dispositivo* en nuestro caso), se debería mostrar el código 403 Forbidden

The screenshot shows the RESTClient interface. In the 'Request' section, a PUT method is selected with URL 'be3603258442d9315ed46e4a4be63'. The 'Headers' section contains 'Content-Type: application/json'. The 'Body' section contains a JSON object: {"MARCA": "Apple", "NOMBRE": "iPad Pro 2018", "COLOR": "Deep Indigo", "CAPACIDAD": "512", "REFERENCIA": "1000000000002"}. In the 'Response' section, the status code is 403 Forbidden, and the response headers include:

1. Status Code	:	403 Forbidden
2. Connection	:	Keep-Alive
3. Content-Length	:	75
4. Content-Type	:	application/json
5. Date	:	Tue, 04 Jul 2017 12:04:32 GMT
6. Keep-Alive	:	timeout=5, max=100
7. Server	:	Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
8. X-Powered-By	:	PHP/5.6.30

Token: OTRO_TOKEN_FABRICANTE

Prueba 13: DELETE de Dispositivo

Vamos a empezar borrando un *Dispositivo* correctamente. Esto es, indicando su referencia y un token de acceso válido y con privilegios.

En concreto, vamos a eliminar el *Dispositivo* que creamos en la prueba del método POST, consiguiendo así dejar la BD como al comienzo.

En caso de resultar exitoso, se nos deberá mostrar el código de estado HTTP 204 No Content.

URI: http://localhost/PHP_API/dispositivos/1000000000002?access_token=TOKEN

The screenshot shows a RESTClient window with the following details:

- Request:** Method: DELETE, URL: `4ac738fb770e913877a67ca80ad4f`, SEND button.
- Body:** Request Body is empty.
- Response:**
 - Response Headers:
 - Status Code : 204 No Content
 - Connection : Keep-Alive
 - Content-Type : text/html; charset=UTF-8
 - Date : Tue, 04 Jul 2017 15:58:38 GMT
 - Keep-Alive : timeout=5, max=100
 - Server : Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
 - X-Powered-By : PHP/5.6.30

At the bottom, there are links for Home, Github, Issues, and Donate, and a Back to top link. The taskbar at the bottom shows icons for File Explorer, Task View, and a browser, with the date 4/7/17 and time 17:59.

Prueba 14: DELETE de Dispositivo (token inválido)

Si adjuntamos un token inválido, se nos debe mostrar el código 401 Unauthorized.

Token inválido: ed773a27c86dff1a0975bfe61546475df83012.9

The screenshot shows a RESTClient window with the following details:

- Request:** Method: DELETE, URL: `:86dff1a0975bfe61546475df83012.9`, SEND button.
- Body:** Request Body is empty.
- Response:**
 - Response Headers:
 - Status Code : 401 Unauthorized
 - Cache-Control : no-store
 - Connection : Keep-Alive
 - Content-Length : 84
 - Content-Type : application/json
 - Date : Tue, 04 Jul 2017 16:03:51 GMT
 - Keep-Alive : timeout=5, max=100
 - Server : Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
 - WWW-Authenticate : `Bearer realm="Service", error="invalid_token", error_description="The access token provided is invalid"`
 - X-Powered-By : PHP/5.6.30

At the bottom, there are links for Home, Github, Issues, and Donate, and a Back to top link. The taskbar at the bottom shows icons for File Explorer, Task View, and a browser, with the date 4/7/17 and time 16:03.

Prueba 15: DELETE de Dispositivo (token expedido por otro)

Si el token fuera válido, pero expedido por un *Fabricante* diferente al que el *Dispositivo* referencia, se debería mostrar un código de estado HTTP 403 Forbidden al usuario, además de un texto explicativo en el cuerpo del mismo.

The screenshot shows the RESTClient application window. In the Request tab, a DELETE method is selected, and the URL is set to `bd62abccf898ed16be48e86c2863a6`. The Body section is empty. In the Response tab, the Response Headers table shows the following:

Header	Value
Status Code	: 403 Forbidden
Connection	: Keep-Alive
Content-Length	: 75
Content-Type	: application/json
Date	: Tue, 04 Jul 2017 16:08:36 GMT
Keep-Alive	: timeout=5, max=100
Server	: Apache/2.4.25 (Win32) OpenSSL/1.0.2j PHP/5.6.30
X-Powered-By	: PHP/5.6.30

Token: TOKEN_OTRO_FABRICANTE

Bibliografía

- [1] Musser, J. (2011). Open APIs - State of the Market 2011. Julio 1, 2017, de ProgrammableWeb Sitio web: <https://www.slideshare.net/jmusser/open-apis-state-of-the-market-2011>
- [2] w3Techs. (2017). Usage of server-side programming languages for websites. Julio 2, 2017, de w3Techs Sitio web:
https://w3techs.com/technologies/overview/programming_language/all
- [3] <https://oauth.net/2/>
- [4] <https://www.apachefriends.org/es/index.html>
- [5] <http://www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html>
- [6] <http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html>
- [7] <https://regex101.com/>
- [8] <https://addons.mozilla.org/es/firefox/addon/restclient/>
- [9] Shaffer, B. (2014). A library for implementing an OAuth2 Server in php. Junio 20, 2017, de GitHub Sitio web: <http://bshaffer.github.io/oauth2-server-php-docs/>
- [10] http://httpd.apache.org/docs/current/mod/mod_rewrite.html