cipsa.net


# PHP & MySQL

## *Anexo 11.- Servicios Web RESTful*

**DISTRIBUIDO POR:**

**CENTRO DE INFORMÁTICA PROFESIONAL S.L.**

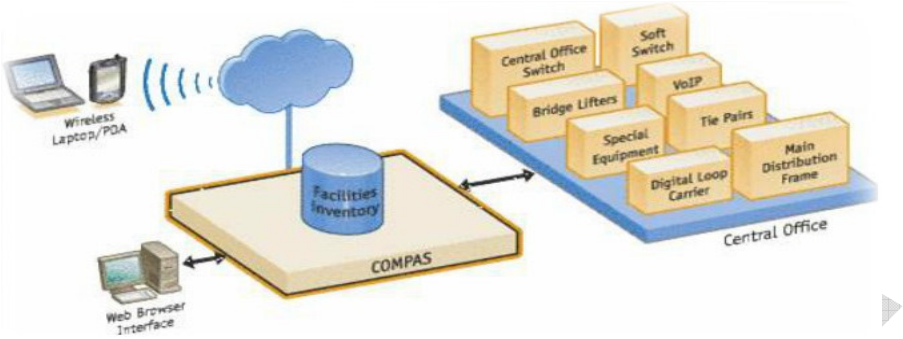
C/ URGELL, 100
08011 BARCELONA
TFNO: 93 426 50 87

C/ RAFAELA YBARRA, 10
48014 BILBAO
TFNO: 94 448 31 33

www.cipsa.net

# *Servicios web*

Los servicios web son un mecanismo que permite la comunicación entre aplicaciones creadas mediante distintas tecnologías. Esto permite crear aplicaciones distribuidas en las que los usuarios emplean aplicaciones en Java, .NET o Android, para obtener datos de una base de datos central a través de un servicio web programado bien en PHP, .NET, Javascript, PHP…, etc.




## Servicios web *RESTful*

Los servicios web RESTful son servicios orientados a ***recursos*** a los cuales se accede mediante peticiones HTTP dirigidas contra ciertas URLs. El conjunto de estas URLs conforman la estructura del servicio web. Estas URLs conforman la API del servicio web:

- URL representando todos los elementos de un recurso
  ***https://midominio.com/version/recurso***

- URL representando un elemento determinado.
  ***https://midominio.com/version/recurso/id***

- URL representando un subconjunto de elementos de un recurso
  ***https://midominio.com/version/recurso?campo=valor***

El campo "***version***" hace referencia a la versión del servicio web. Se emplea para no provocar problemas en las aplicaciones consumidoras de versiones existentes cuando se actualiza el servicio web.

El campo "***recurso***" representa un conjunto de elementos semejante a los registros de una tabla de una base de datos. La URL al recurso es la clave para obtener, insertar, modificar y eliminar los elementos almacenados. Si el recurso es "usuarios", los elementos serán cada usuario registrado.

El campo "***id***" representa el valor que permite identificar un elemento concreto del recurso. Por ejemplo; en el caso de recurso "personas", el id sería el valor del DNI;

## Formato de datos de transferencia

El formato de datos de transferencia determina el modo en que se codifican los datos al transferirse entre el servicio web y la aplicación consumidora del mismo.

Los servicios *RESTful* pueden devolver datos en cualquier aunque lo más común es emplear XML o JSON. La codificación a emplear se empleado la cabecera "*Content-Type*" seguida del tipo MIME correspondiente:

- Cabecera de formato JSON: **`Content-Type: application/json`**
- Cabecera de formato XML: **`Content-Type: text/xml`**

## Operaciones con datos

Las operaciones sobre los datos es determinada por el tipo de petición HTTP con la que se accede a la URL del recurso o elemento.

Para obtener datos se emplea siempre una petición ***HTTP_GET***. El valor o conjunto de valores obtenidos viene determinado por la URL contra la que se invoca la petición.

- Todos los usuarios:

  `GET: http://localhost/1.0/usuarios`

- Un usuario determinado dado su DNI:

  `GET: https://midominio.com/1.0/usuarios/11993489K`

- Parte de los usuarios que satisfacen una condición:

  `GET: https://midominio.com/1.0/usuarios?edad=20`

Para insertar un nuevo elemento se envía una petición ***HTTP_POST*** dirigida a la URL del recurso. Los datos del nuevo elemento se integran en la propia petición codificados en XML, JSON, texto…, etc.

`POST: https://midominio.com/1.0/usuarios`

Para modificar un elemento se envía una petición ***HTTP_PUT*** dirigida a la URL del elemento. Los nuevos valores se integran como parte de la propia petición codificados.

`PUT: https://midominio.com/1.0/usuarios/11993489K`

Para eliminar un elemento se envía una petición ***HTTP_DELETE*** dirigida a la URL del elemento.

`DELETE: https://midominio.com/1.0/usuarios/11993489K`

## Códigos de respuesta

Los servicios web RESTful emplean los *códigos de estado HTTP*. Estos van incluidos en las respuestas HTTP e informan del resultado de las peticiones:

| Código | Significado | Uso |
|---|---|---|
| ***200*** | ***OK*** | Indica que el recurso solicitado existe y se envían los datos como parte de la respuesta a una petición GET |
| ***201*** | ***Created*** | Indica que se ha creado un nuevo como respuesta una petición POST. Suele ser aconsejable enviar una cabecera "*Location*" indicando la URL al nuevo recurso. |
| ***204*** | ***No Content*** | Indica el resultado correcto a una petición que no retorna sin embargo ningún valor. Se emplea habitualmente como respuesta correcta a una petición DELETE/PUT |
| ***304*** | ***No Modified*** | Indica que un recurso no ha sido modificado. Se emplea para indicar que una petición POST, PUT o DELETE no ha tenido éxito. |
| ***401*** | ***Unauthorized*** | Indica que el usuario no tiene permiso para realizar la petición. |
| ***404*** | ***Not Found*** | Indica que no existe el recurso indicado en la URL solicitada. |
| ***405*** | ***Mehot Not Allowed*** | Indica que no se permite el tipo de petición indicado contra la URL. Suele emplearse para indicar un recurso de sólo lectura que no permite modificaciones. |
| ***422*** | ***Unprocessable Entity*** | Indica que los datos asociados a la petición no son correctos y no puede procesarse. Esto se emplea cuando los datos enviados al servidor como parte de una petición POST o PUT no son válidos o no están correctamente formateados. |
| ***500*** | ***Internal Server Error*** | Indica un error interno del servidor al generar la respuesta a la petición. |
| ***503*** | ***Service Unavailable*** | Indica que el servicio web no está operativo temporalmente. |

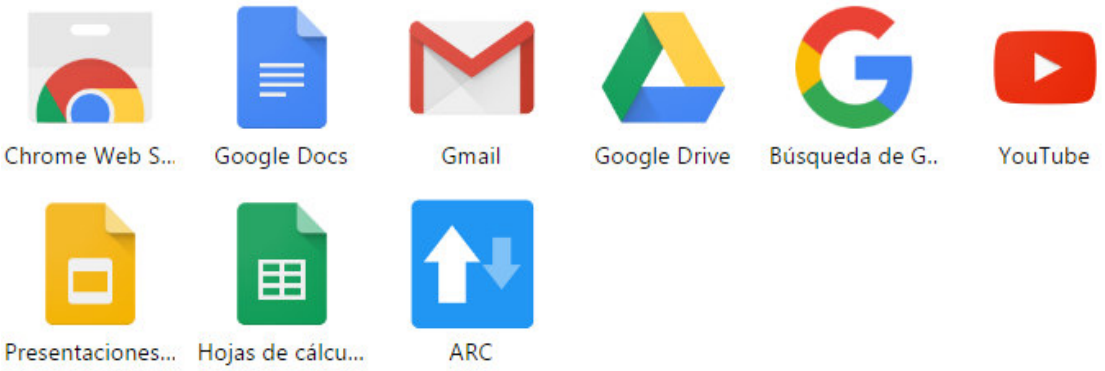
## Herramientas de verificación

Cuando se programa o prueba un servicio web RESTful conviene tener una herramienta que nos permita enviar los diferentes tipos de peticiones a las URLs del servicio web y comprobar los devueltos como respuesta. A diferencia de con las aplicaciones web, para probar un servicio web no vale con un simple navegador.

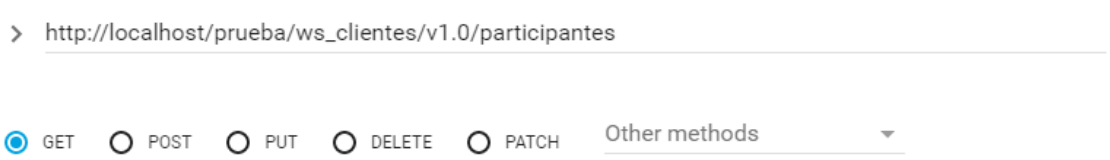
Una herramienta que podemos emplear es el ***ARC*** (*Advanced REST Client*). Se trata de un complemento del navegador Google Chrome que puede descargarse libremente desde el sitio web Chrome:

***https://chrome.google.com/webstore/category/extensions***

Una vez instalado este complemento, podemos iniciar la aplicación desde el navegador ***Chrome*** indicando la URL: ***chrome://apps/.*** Se muestra entonces una página con las aplicaciones instaladas:

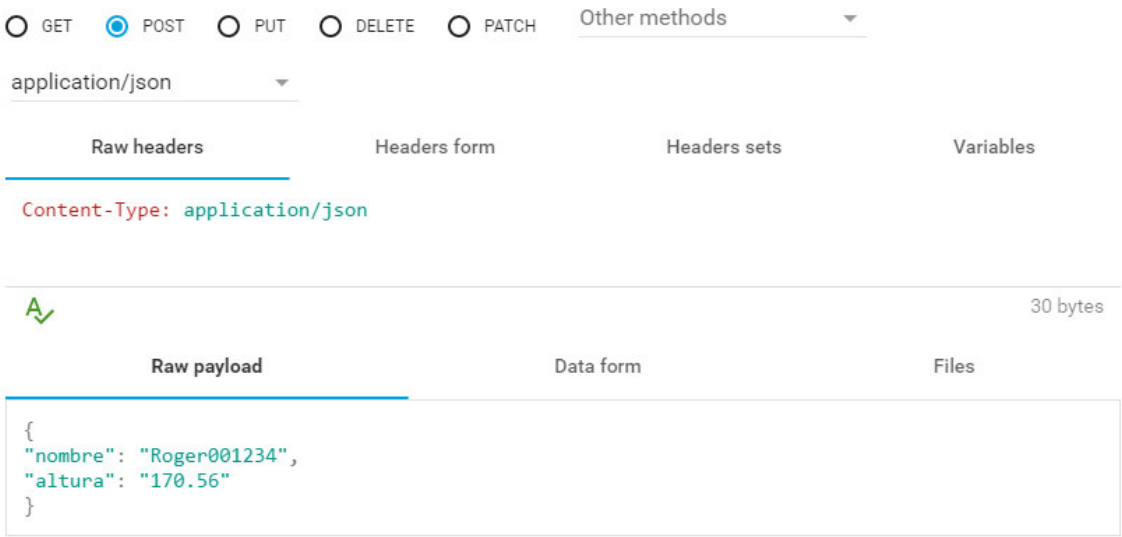



La herramienta nos permite indicar una URL, el tipo de petición HTTP que deseamos enviar ( GET, POST, PUT, DELETE ).



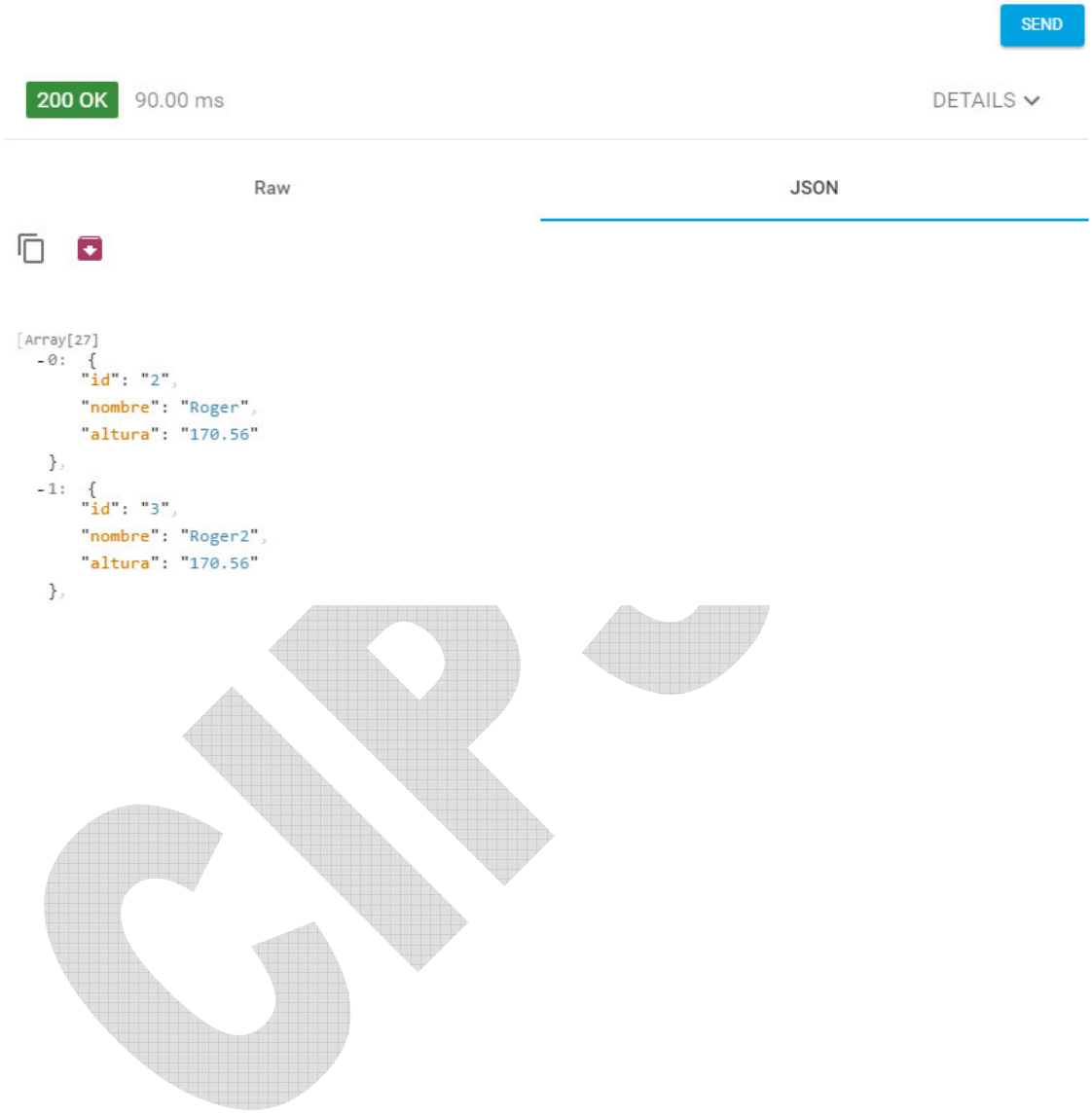

Para aquellas peticiones que requieren el envío de datos ( POST, y PUT ), debemos indicar el tipo de codificación y los propios datos en el cuadro "***Raw payload***":

***Ejemplo:*** Envío en una petición POST de un objeto con dos atributos "nombre" y "altura" codificado en JSON:



Al pulsar el botón de envío "*SEND*", la aplicación muestra el código HTTP de la respuesta así como los datos enviados por el servidor web.

***Ejemplo:*** Respuesta correcta (código 200) a una petición *GET* en la que se envía una matriz de 27 objetos con atributos "*id*", "*nombre*" y "*altura*" codificada en JSON.



## Creación de servicio web RESTful PHP

Como ejemplo vamos a crear un servicio web RESTful que permita la gestión de los participantes de un equipo de baloncesto.

Los datos de los participantes estarán almacenados en la tabla ***PARTICIPANTES*** de una base de datos ***BALONCESTO*** almacenada en un servidor de *MySQL*. La definición de la tabla es la siguiente:

| # | Nombre | Tipo | Cotejamiento | Atributos | Nulo | Predeterminado | Extra |
|---|---|---|---|---|---|---|---|
| 1 | **id** | int(11) | | | No | *Ninguna* | AUTO_INCREMENT |
| 2 | **nombre** | varchar(20) | | | No | *Ninguna* | |
| 3 | **altura** | decimal(5,2) | | | No | *Ninguna* | |

El campo "*id*" es autonumérico, por lo que es generado por la propia base de datos al dar de alta cada nuevo participante.

Los datos se codificarán en JSON ( *content-type: application/json* )

## URLs del servicio web

Las URLs que conforman el servicio web son las siguientes.

- ***GET: /participantes***

  Devuelve una matriz de objetos participantes (*id*, *nombre*, *altura*) con todos los participantes registrados en la base de datos.

- ***GET: /participantes/:id***

  Devuelve el objeto participante (*id*, *nombre*, *altura*) del participante con el identificador indicado. Si no existe el participante, retorna *null* como respuesta.

- ***POST: /participantes***

  Inserta el objeto participante enviado (*nombre*, *altura*). Retorna un objeto con un atributo "*id*" y el identificador del nuevo participante.

- ***PUT: /participantes/:id***

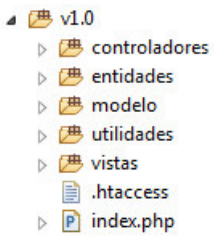
  Modifica los datos del participante con el identificador indicado con los datos del objeto participante enviado (nombre, altura). Si el participante existe retorna un objeto con el atributo *rows=1*. En caso contrario, el atributo *rows=0.*

- ***DELETE: /participante/:id***

  Elimina el participante con el identificador indicado. Si el participante existe retorna un objeto con el atributo *rows=1*. En caso contrario, el atributo *rows=0*.

## Estructura del proyecto

Los servicios web al igual que las aplicaciones web pueden crearse empleando el patrón MVC. Como punto de partida podemos empezar creando la siguiente estructura para el proyecto:




En cada una de las carpetas organizaremos las diferentes clases que componen el modelo, la vista y la capa de control del servicio web.

## Implementación del Modelo

Las clases de la capa modelo se encargan de obtener y manipular los datos de los participantes. Los ficheros y clases que la componen son los siguientes:

***modelo/config.ini***

```
[database]
host     = 192.168.9.180
user     = alumno
password = cipsa
database = baloncesto
```

Definimos la clase ***Modelo/Database*** que encapsula la conexión a la base de datos y toma los datos de conexión del fichero de configuración: *config.ini*.

***Código**: Modelo/Database*

```
<?php
namespace Modelo;
class Database
{
	private $_dbUser;
	private $_dbPassword;
	private $_dbHost;
	protected $_dbName;

	private $_connection;
	private static $_instance;

	private function __construct()
	{
		try {
			// Obtención de parámetros del archive de configuración
			$config = parse_ini_file(APPPATH . '\modelo\config.ini');
			$this->_dbHost = $config["host"];
			$this->_dbUser = $config["user"];
			$this->_dbPassword = $config["password"];
			$this->_dbName = $config["database"];
			// Instanciación del objeto conexión
			$this->_connection = new \PDO('mysql:host='.$this->_dbHost.'; dbname='.$this-
>_dbName, $this->_dbUser, $this->_dbPassword);
			$this->_connection->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
			$this->_connection->exec("SET CHARACTER SET utf8");
		}
		catch (\PDOException $e)
		{
			print "Error!: " . $e->getMessage();
			die();
		}
```

```
        }

        /**
         * Metodo de ejecucion de consulta parametrizada.
         * @param mixed $sql Consulta
         * @param array $args parametros de sustitucion para consulta
         * @return PDOStatement
         */
        public function run($sql, $args = [])
        {
                $stmt = $this->_connection->prepare($sql);
                $stmt->execute($args);
                // Retorno de objeto PDOStatement.
                return $stmt;
        }

        /**
         * Método de obtención de instancia única de conexión a la base de datos.
         * @return [object] [class database]
         */
        public static function instance()
        {
                if (!isset(self::$_instance))
                {
                        $class = __CLASS__;
                        self::$_instance = new $class;
                }
                return self::$_instance;
        }

        /**
         * Método de llamada a métodos de objeto PDOConnection
         * @param unknown $method Método a llamar
         * @param unknown $args Parámetros para el método
         * @return mixed
         */
        public function __call($method, $args)
        {
                // Ejecuta el método estático invocado con los argumentos dados
                // sobre el objeto PDO existente.
                return call_user_func_array(array($this->_connection, $method), $args);
        }

        /**
         * [__clone Evita que el objeto se pueda clonar]
         * @return [type] [message]
         */
        public function __clone()
        {
                trigger_error('La clonación de este objeto no está permitida', E_USER_ERROR);
        }
}
```

Definimos a continuación la clase abstracta ***Modelo/Modelo*** que es la clase padre de todas las clases de gestión de datos de la capa modelo. Sus métodos permiten las operaciones básicas de listado, obtención, alta, baja y modificación de los datos:

**Código:** *Modelo/Modelo*

```
<?php
namespace Modelo;

abstract class Modelo {
        // Obtener todos los elementos.
        public abstract static function listar();
        // Obtener un elemento dada su $id
        public abstract static function obtener($id);
        // Insertar un elemento $obj
        public abstract static function insertar($obj);
        // Actualizar los datos del elemento $id con los del objeto $obj.
        public abstract static function actualizar($id, $obj);
        // Elimina el elemento dada su $id
        public abstract static function eliminar($id);
}
```

Definimos la clase ***Entidades/Participante*** que representa los datos de cada participante tal como está almacenado en la tabla *PARTICIPANTES* de la base de datos:

***Código:*** *Entidades/Participante:*

```php
<?php
namespace Entidades;

class Participante {

	public $id;
	public $nombre;
	public $altura;

	public function __construct( $_id, $_nombre, $_altura )  {
		$this->id = $_id;
		$this->nombre = $_nombre;
		$this->altura = $_altura;
	}
}
```

Definimos la clase ***Modelo/ModeloParticipantes*** que hereda de la clase padre Modelo/Modelo sobrescribiendo los métodos *listar(), obtener(), insertar(), actualizar()* y *eliminar()* para manejar los datos de los participantes de la base de datos.

***Código:*** *Modelo/ModeloParticipantes*

```php
<?php
namespace modelo;

use Entidades\Participante;
use Utilidades\WSException;

class ModeloParticipantes extends Modelo {

	/**
	 * Devuelve todos los participantes almacenados
	 * @throws WSException
	 * @return array matriz de objetos Entidades/Participantes
	 */
	public static function listar() {
		try {
			$datos = array();
			$db = Database::instance();
			$sql = "SELECT ID, NOMBRE, ALTURA FROM PARTICIPANTES";
			$query = $db->run($sql);
			while( $reg = $query->fetch()) {
				array_push($datos,
					new Participante($reg['ID'], $reg['NOMBRE'], $reg['ALTURA']));
			}
			return $datos;
		} catch(\PDOException $e) {
			throw new WSException(WSException::ESTADO_ERROR_BD, $e->getMessage());
		}
	}

	/**
	 * Devuelve el objeto participante con el identificador dado
	 * @param unknown $id Identificador del participante a retornar
	 * @throws WSException
	 * @return NULL|\Entidades\Participante
	 */
	public static function obtener($id) {
		try {
			$db = Database::instance();
			$sql = "SELECT ID, NOMBRE, ALTURA FROM PARTICIPANTES WHERE ID = :id";
			$query = $db->run($sql, [ ':id' => $id ]);
			$reg = $query->fetch();
			return ( ($reg)
				  ?new Participante($reg['ID'], $reg['NOMBRE'], $reg['ALTURA'])
				  :NULL );
		} catch(\PDOException $e) {
			throw new WSException(WSException::ESTADO_ERROR_BD, $e->getMessage());
		}
	}
```

```php
        /**
         * Inserta el participante indicado
         * @param unknown $obj objeto Entidades/Participante a insertar
         * @throws WSException
         * @return unknown Identificador autogenerado del nuevo registro insertado
         */
        public static function insertar($obj) {
                try {
                        $db = Database::instance();
                        $sql = "INSERT INTO PARTICIPANTES(NOMBRE,ALTURA) VALUES (:nombre,:altura )";
                        $query = $db->run($sql, [
                                                ':nombre' => $obj->nombre,
                                                ':altura' => $obj->altura ]);
                        return $db->lastInsertId();
                } catch(\PDOException $e) {
                        throw new WSException(WSException::ESTADO_ERROR_BD, $e->getMessage());
                }
        }
        /**
         * Modifica el registro participante del identificador dado
         * con los datos del objeto Entidades/Participante dado.
         * @param unknown $id Identificador del registro a modificar
         * @param unknown $obj Objeto con los datos del nuevo participante
         * @throws WSException
         * @return unknown nº de registros modificados 1|0
         */
        public static function actualizar($id, $obj) {
                try {
                        $db = Database::instance();
                        $sql = "UPDATE PARTICIPANTES SET NOMBRE=:nombre, ALTURA=:altura WHERE ID=:id";
                        $query = $db->run($sql, [
                                                ':nombre' => $obj->nombre,
                                                ':altura' => $obj->altura,
                                                ':id' => $id]);
                        return $query->rowCount();
                } catch(\PDOException $e) {
                        throw new WSException(WSException::ESTADO_ERROR_BD, $e->getMessage());
                }
        }
        /**
         * Elimina el registro participante del identificador indicado.
         * @param unknown $id Identificador del participante a eliminar
         * @throws WSException
         * @return unknown nun de registros eliminados 1|0
         */
        public static function eliminar($id) {
                try {
                        $db = Database::instance();
                        $sql = "DELETE FROM PARTICIPANTES WHERE ID = :id";
                        $query = $db->run($sql, [':id' => $id]);
                        return $query->rowCount();
                } catch(\PDOException $e) {
                        throw new WSException(WSException::ESTADO_ERROR_BD, $e->getMessage());
                }
        }
}
```

## Implementación de la Vista

La capa vista de una aplicación web se encarga de generar las páginas HTML que son enviadas como respuesta al navegador del cliente. En el caso de un servicio web RESTful, codificar los datos que son enviados como respuesta a la aplicación consumidora.

Las clases que la componen son las siguientes:

### *Vistas/Vista*

Esta es una clase abstracta que define el método abstracto ***imprimir()***. Este método se encarga de codificar los datos que recibe como parámetro a un determinado formato y enviarlos a la aplicación consumidora del servicio web. Este método es sobrescrito por las clases hijas implementando la codificación en diferentes formatos.

***Código:*** *Vistas/Vista*

```php
<?php
namespace Vistas;

abstract class Vista{

	// Código de respuesta HTTP
	public $estado;

	// Método de codificación de los datos indicados en el parámetro $cuerpo.
	public abstract function imprimir($cuerpo);
}
```

### *Vistas/VistaJSON*

Esta clase hereda de la clase ***Vistas/Vista*** y sobrescribe el método abstracto ***imprimir()*** para codificar los datos en formato *JSON*:

***Código:*** *Vistas/VistaJSON*

```php
<?php
namespace Vistas;

/**
 * Clase para imprimir en la salida respuestas con formato JSON
 */
class VistaJson extends Vista
{
	public function __construct($estado = 200)
	{
		$this->estado = $estado;
	}
	 /**
	  * Envia los datos en formato JSON con el código HTTP de respuesta (por defecto:200)
	  * @param mixed $cuerpo de la respuesta a enviar
	  */
	 public function imprimir($cuerpo)
	 {
		 if ($this->estado) {
			http_response_code($this->estado);
		 }
		  header('Content-Type: application/json; charset=utf8');
		  echo json_encode($cuerpo, JSON_PRETTY_PRINT);
		  exit;
	 }
}
```

## Reescritura de URLs

Las clases de la capa Controlador reciben las peticiones contra cada uno de los recursos publicados por el servicio web RESTful. Estas clases deben llamarse como los recursos y poseer los métodos para responder a cada uno de los distintos tipos de peticiones HTTP (*get*, *post*, *put*, *delete*)

El fichero ***.htaccess*** configura el modulo de apache de reescritura de URLs (*mod_rewrite*) de modo que una petición dirigida a una URL amigable como:

http://localhost/1.0/participantes/2

Se convierta en:

http://localhost/1.0/index.php?PATH_INFO=participantes/2

Y todas sean atendidas de este modo por el script ***index.php***.

***Código:*** *.htaccess*

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php?PATH_INFO=$1 [L,QSA]
```

De este modo, el script ***index.php*** sirve de punto de entrada para todas las peticiones dirigidas al servicio web. Es trabajo de este script obtener el valor del parámetro ***PATH_INFO*** presente en la URL reescrita, y extraer el nombre del recurso y sus parámetros:

```
// Decodificacion de URL
$peticion = explode('/', $_GET['PATH_INFO']);
$recurso = "Controladores\\".array_shift($peticion);
$metodo = strtolower($_SERVER['REQUEST_METHOD']);
```

Sea una petición GET "*/1.0/participantes/2*", el código anterior obtendría los siguientes valores:

$recurso:

```
C:\xampp7\htdocs\prueba\ws_clientes\v1.0\index.php:50:string 'Controladores\participantes' (length=27)
```

$metodo:

```
C:\xampp7\htdocs\prueba\ws_clientes\v1.0\index.php:52:string 'get' (length=3)
```

$peticion

```
C:\xampp7\htdocs\prueba\ws_clientes\v1.0\index.php:54:
array (size=1)
  0 => string '2' (length=1)
```

- La variable ***$recurso*** determina el nombre de la clase controlador que atenderá la petición.
- La variable ***$metodo*** indica el nombre del método de esa clase controladores que atenderá la petición.
- La variable ***$peticion*** es lo que se pasará como parámetro a los métodos que lo requieran: *get()*, *put()* y *delete()*.

La ejecución del método indicado por la variable $*metodo* de la clase controlador indicada por la variable $*recurso* pasando como parámetro la matriz almacenada en la variable $*peticion* se realiza mediante el siguiente código:

```php
// Ejecución de controlador
if ( class_exists($recurso) == true ) {
        $respuesta = call_user_func(array($recurso, $metodo), $peticion);
        $vista->imprimir($respuesta);
} else {
        throw new WSException(WSException::ESTADO_INEXISTENCIA_RECURSO,"Recurso no existente");
}
```

En caso que no exista ninguna clase controlador con el nombre del recurso indicado en la URL, se entiende que ese recurso no existe y se lanza una excepción ***WSException*** con el código *INEXISTENCIA_RECURSO*.

### *Utilidades/WSException*

Esta es una clase de ayuda para manejar los errores que puedan producirse al procesar las peticiones recibidas por el servicio web. El objetivo, es que aunque se produzca un error; se emita una respuesta al cliente indicando la causa.

***Código:*** *Utilidades/WSException*

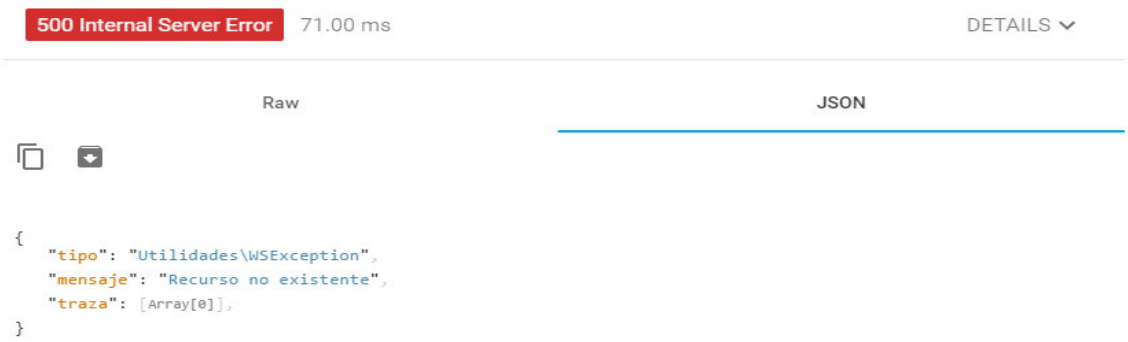
```php
<?php
namespace Utilidades;
class WSException extends \Exception{
        const ESTADO_URL_INCORRECTA = 2;
        const ESTADO_INEXISTENCIA_RECURSO = 3;
        const ESTADO_METODO_NO_PERMITIDO = 4;
        const ESTADO_ERROR_BD = 5;
        const ESTADO_JSON_FORMATO_INCORRECTO = 6;

        public $estado;
        public function __construct($estado, $mensaje, $codigo = 400)
        {
                $this->estado = $estado;
                $this->message = $mensaje;
                $this->code = $codigo;
        }
}
```

Para capturar las excepciones que puedan producirse se define en el fichero index.php una función manejadora de excepciones mediante la función ***set_exception_handler()***:

```php
// Manejador de excepciones
set_exception_handler(function ($exception) use ($vista) {
                // Se crea una matriz con los datos de la excepcion
                $cuerpo = array(
                                "tipo" => get_class($exception),
                                "mensaje" => $exception->getMessage()
                );
                if ( get_class($exception) == "WSException") {
                        // La excepcion es de tipo WSException
                        $cuerpo['estado'] = $exception->estado;
                        $vista->estado = $exception->getCode();
                } else {
                        // La excepcion es de otro tipo.
                        $cuerpo['traza'] = $exception->getTrace();
                        $vista->estado = 500;
                }
                $vista->imprimir($cuerpo);
        }
);
```

En el caso de producirse una excepción se genera una respuesta con código de estado 500 y un objeto provisto de los atributos "*tipo*" y "*mensaje*" descriptivos del error.

Ejemplo: Respuesta devuelta por el servicio web al acceder a un recurso inexistente:




El código del script ***index.php*** queda por tanto del siguiente modo:

```php
<?php
//directorio del proyecto
define("APPPATH", dirname(__FILE__));

// Función de autocarga de clases
function autoload_classes($class_name)
{
        $filename = APPPATH. '/' . str_replace('\\', '/', $class_name) .'.php';
        if(is_file($filename)){
                include_once $filename;
        }
}
spl_autoload_register('autoload_classes');

use Utilidades\WSException;
use Vistas\VistaJson;

// Generador de vistas
$vista = new VistaJson();

// Manejador de excepciones
set_exception_handler(function ($exception) use ($vista) {
                // Se crea una matriz con los datos de la excepcion
                $cuerpo = array(
                                "tipo" => get_class($exception),
                                "mensaje" => $exception->getMessage()
                );
                if ( get_class($exception) == "WSException") {
                        // La excepcion es de tipo WSException
                        $cuerpo['estado'] = $exception->estado;
                        $vista->estado = $exception->getCode();
                } else {
                        // La excepcion es de otro tipo.
                        $cuerpo['traza'] = $exception->getTrace();
                        $vista->estado = 500;
                }
                $vista->imprimir($cuerpo);
        }
);

// Decodificacion de URL
$peticion = explode('/', $_GET['PATH_INFO']);
$recurso = "Controladores\\".array_shift($peticion);
$metodo = strtolower($_SERVER['REQUEST_METHOD']);
// ENRUTADO: Comprobacioon: Existe el controlador?
if ( class_exists($recurso) == true ) {
        // Ejecucion del método del controlador y obtencion de repsuesta
        $respuesta = call_user_func(array($recurso, $metodo), $peticion);
        // Envio de respuesta codificada por la vista.
        $vista->imprimir($respuesta);
} else {
        // ERROR -> Controlador no existente - Recurso desconocido
        throw new WSException(WSException::ESTADO_INEXISTENCIA_RECURSO, "Recurso no existente");
}
```

***Utilidades/Mapper***

Esta clase dispone del método estático ***MapCheck()*** que se emplea en la implementación del controlador para validar que los objetos enviados en las peticiones *POST* y *PUT* poseen los atributos requeridos.

***Código:*** *Utilidades/Mapper*

```php
<?php
namespace Utilidades;
class Mapper {
        /**
         * Comprueba que el objeto indicado tenga las propiedades indicadas
         * @param unknown $obj objeto a comprobar
         * @param array $props matriz de propiedades a comprobar
         * @return boolean Cierto si el objeto tiene las propiedades indicadas, o falso en caso
contrario
         */
        public static function MapCheck( $obj, $props = Array()) {
                foreach ($props as $prop) {
                        if (!property_exists($obj, $prop)) return false;
                }
                return true;
        }
}
```

***Ejemplo:*** Para comprobar si un objeto obtenido a partir de una petición tiene dos atributos “nombre” y “altura requeridos, se emplearía el siguiente código:

```php
Mapper::MapCheck($participante, ['nombre', 'altura'])
```

Esto se hace para evitar errores en el servicio web al procesar peticiones con datos mal formateados.

## Implementación del controlador

### *Controladores/Controlador*

Esta es una clase abstracta que sirve de modelo para el resto de clases controladores hijas. Sus métodos se corresponden con las peticiones HTTP:

***Código:*** *Controladores/Controlador*

```php
<?php
namespace Controladores;
abstract class Controlador {
        public abstract static function get($peticion);
        public abstract static function post();
        public abstract static function put($peticion);
        public abstract static function delete($peticion);
}
```

### *Controladores/Participantes*

Esta es la clase controladora que recibe las peticiones dirigidas al recurso *participantes*. Cada uno de sus métodos get(), post(), put() y delete() debe implementar las operaciones para atender cada uno de los tipos de peticiones que puede recibir:

***GET*** → Petición de obtención de todos los participantes registrados:

```
GET: /participantes
```

Debe devolver una matriz de objetos participante codificada en JSON incluyendo los atributos id, nombre y altura de cada uno.

```json
[
    {
        "ID": "2",
        "NOMBRE": "Roger",
        "ALTURA": "170.56"
    },
    {
        "ID": "3",
        "NOMBRE": "Roger2",
        "ALTURA": "170.56"
    },
    {
        "ID": "4",
        "NOMBRE": "Roger3",
        "ALTURA": "170.56"
    }
],
```

***GET*** → Petición de obtención del participante registrado con la identificación indicada:

```
GET: /participantes/1
```

Devuelve un único objeto codificado en JSON incluyendo los atributos id, nombre y altura del participante con el identificador indicado. En caso de no existir ningún participante con el identificador indicado el servicio web retorna ***null***.

```json
    {
        "ID": "2",
        "NOMBRE": "Roger",
        "ALTURA": "170.56"
    }
```

Implementación del método ***get($peticion)***

```
        public static function get($peticion) {
                if (empty($peticion[0]))
                        // GET /participantes
                        return Modelo::listar();
                else
                        // GET /participantes/:id
                        return Modelo::obtener($peticion[0]);
        }
```

Este método atiende las peticiones ***GET***. El parámetro $*peticion* recibe una matriz con los parámetros asociados a la URL.

- Si la URL es de tipo *"/participantes"*, el parámetro $*peticion* no tiene ningún valor. Se ejecuta entonces el método *listar().*

- Si la URL es de tipo "*/participantes/:id*", el parámetro $*peticion* contiene un único valor en la posición 0. Se ejecuta entonces el método *obtener()* del modelo pasando como parámetro el identificador.

***POST*** → Petición de inserción de un nuevo participante:

La petición debe incluir el objeto participante que se desea dar de alta codificado en JSON incluyendo los atributos "*nombre*" y "*altura*". El atributo "*id*" no se incluye ya que es generado por la base de datos al insertarse el registro correspondiente.

***POST: /participantes***

Código JSON de objeto enviado:

```
{
"nombre": "Paco2",
"altura": "180.56"
}
```

El servicio web devuelve como respuesta un objeto codificado en JSON con un el valor del atributo "*id*" correspondiente al participante dado de alta.

```
{
   "id": "33"
}
```

Implementación del método ***post()***

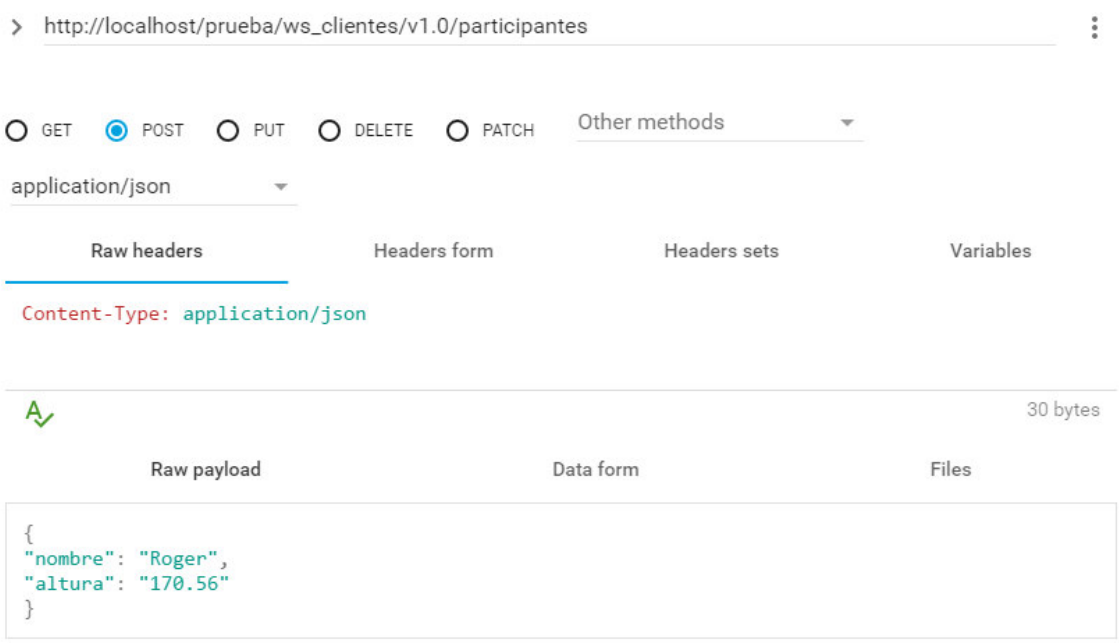
```
        public static function post()
        {
                // POST: /participantes
                  // Obtencion de los datos inscritos en la peticion
                  $body = file_get_contents('php://input');
                  $nuevo_participante = json_decode($body);
                // Comprobacion del objeto enviado
                if ( !Mapper::MapCheck($nuevo_participante, ['nombre', 'altura']) )
                        throw new WSException(WSException::ESTADO_JSON_FORMATO_INCORRECTO,
                                                        "Formato JSON incorrecto");
                // Insercion en el modelo y obtención del identificador
                $idContacto = Modelo::insertar($nuevo_participante);
                // Retorno de matriz con identificador como respuesta.
                  return [
                          "id" => $idContacto
                  ];
        }
```

Este método atiende las peticiones POST y requiere un objeto integrado en la petición y codificado en JSON con los datos del nombre y la altura del nuevo participante, que se obtiene mediante ***file_get_contents('php://input')***, y se decodifica mediante la función ***json_decode()***.

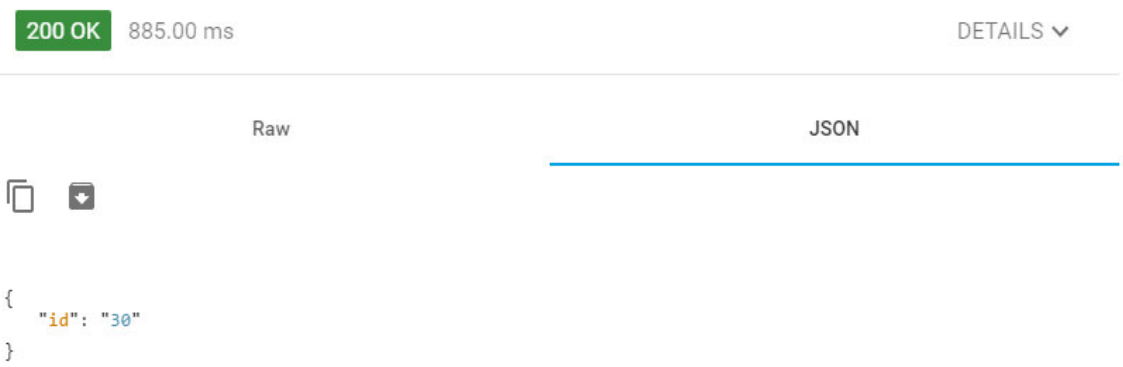
El método estático ***MapCheck()*** de la clase ***Utilidades/Mapper*** comprueba que el objeto recibido contiene las propiedades esperadas "*nombre*" y "*altura*". Si el objeto recibido no posee los atributos esperados se lanza una excepción ***WSException*** con el código *JSON_FORMATO_INCORRECTO*".

Una vez comprobado que el objeto recibido es correcto, se inserta mediante el método ***insertar()*** del modelo obteniendo el identificador del nuevo registro. El identificador del nuevo registro es retornado en una matriz asociativa con índice "*id*".

***Ejemplo:*** Generación de una petición ***POST*** con la herramienta ARC para insertar un nuevo participante ( *nombre = "Roger", altura = "170.56"* ):




El resultado devuelto por el servidor es un objeto con un atributo "id" y el valor del campo ID del registro del nuevo participante:



***DELETE*** → Petición de eliminación de un participante dado su identificador:

```
DELETE: /participantes/1
```

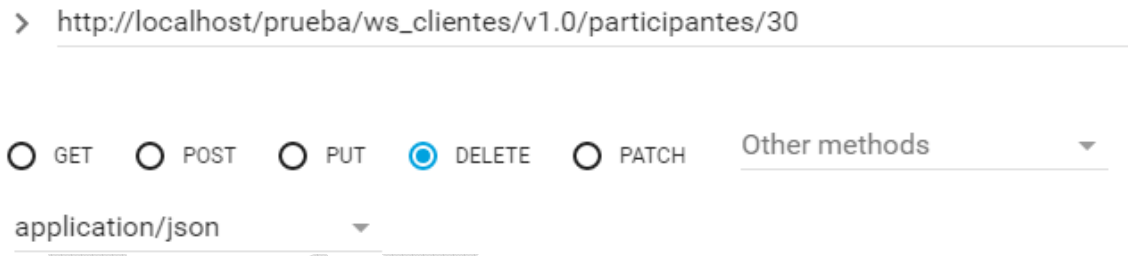
Implementación del método ***delete()***

```
public static function delete($peticion)
{
        // DELETE: /participantes/:id
        // Comprobacion de existencia de parámetro :id
        if ( count($peticion) == 1) {
                // Obtención de parámetro :id
                $id = $peticion[0];
                // Eliminación de registro
                $rowCount = Modelo::eliminar($id);
                return [
                        "rows" => $rowCount
                ];
        } else throw new \WSException(\WSException::ESTADO_URL_INCORRECTA,
                                        "Se esperaba un parámetro");
}
```

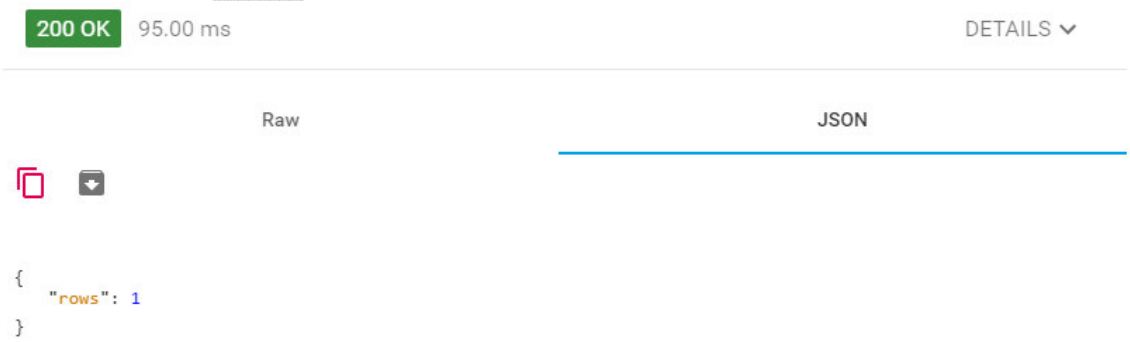
Este método atiende las peticiones ***DELETE*** y requiere un parámetro asociado a la URL con el identificador del participante que se desea eliminar. En caso de no recibirse dicho parámetro se lanza una excepción ***WSException*** con el código *URL_INCORRECTA*.

A continuación se elimina el registro con el identificador indicado mediante el método ***eliminar()*** del modelo. Este devuelve el nº de registros eliminados que es retornado en una matriz asociativa con la clave "*rows*":

***Ejemplo:*** Generación de una petición ***DELETE*** para eliminar el participante con *ID = 30*.




El resultado devuelto por el servidor es un objeto con un atributo "*rows*" y el nº de registros eliminados. Si existía el participante con ID = 30, el valor retornado será 1. En caso contrario será 0.



***PUT*** → Petición de modificación de los datos de un participante dado su identificador. La petición debe incluir un objeto codificado en JSON con el valor de los atributos "nombre" y "altura" a actualizar:

```
PUT: /participantes/1
```

Código JSON de objeto enviado:

```
{
"nombre": "Paco2",
"altura": "180.56"
}
```

Implementación del método ***put($peticion)***

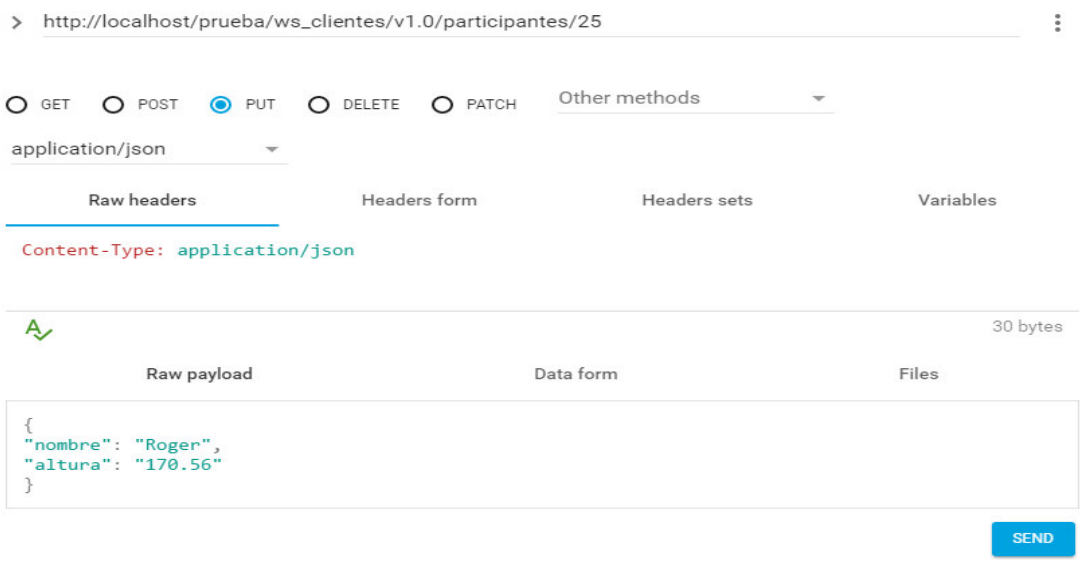
```
        public static function put($peticion)
        {
                // PUT: /participantes/:id
                // Comprobacion de existencia de parámetro :id
                if ( count($peticion) == 1 ) {
                        // Obtención de parámetro :id
                        $id = $peticion[0];
                        // Obtencion y decodificación de objeto inscritos en la petición
                        $body = file_get_contents('php://input');
                        $participante = json_decode($body);
                        // Comprobacion del objeto enviado
                        if ( !Mapper::MapCheck($participante, ['nombre', 'altura']) )
                             throw new WSException(WSException::ESTADO_JSON_FORMATO_INCORRECTO,
                                                   "Formato JSON incorrecto");
                        // Actualizacion en el modelo y obtención nº registros modificados
                        $rowCount = Modelo::actualizar($id, $participante);
                        return [
                                "rows" => $rowCount
                        ];
                } else throw new WSException(WSException::ESTADO_URL_INCORRECTA,
                                             "Se esperaba un parámetro");
        }
```

Este método atiende las peticiones ***PUT*** y requiere de un parámetro asociado a la URL con el identificador del participante a modificar, y un objeto integrado en la petición y codificado en JSON con los atributos "*nombre*" y "*altura*" a actualizar.

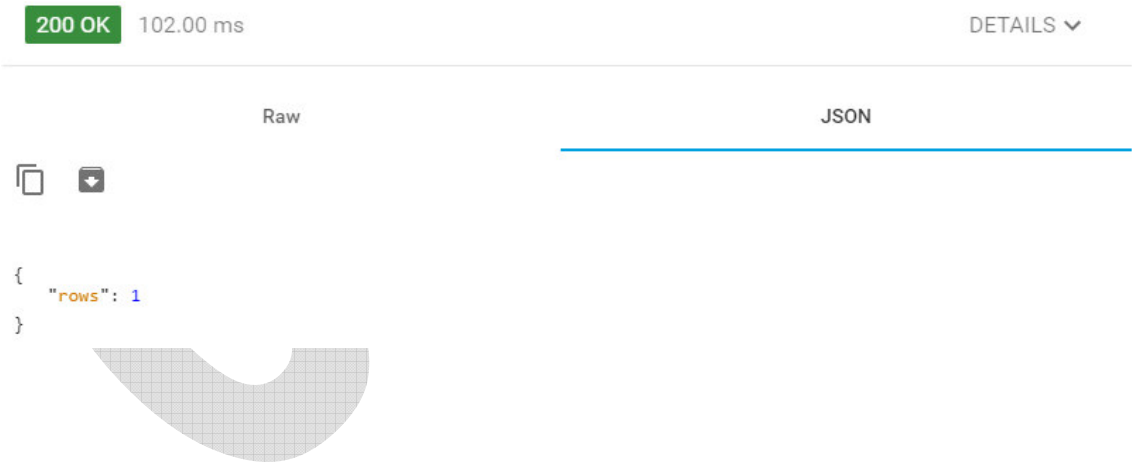
- Si la URL no incluye el parámetro el parámetro ID, se lanza una excepción ***WSException*** con el código *URL_INCORRECTA*.
- Si la petición no incluye un objeto con los atributos "*nombre*" y "*altura*" se lanza una excepción ***WSException*** con código *JSON_FORMATO_INCORRECTO*.

Si todo es correcto se realiza la actualización mediante el método actualizar() del modelo, que devuelve el nº de registros modificados. Este valor es retornado en una matriz asociativa con la clave "***rows***":

***Ejemplo:*** Generación de una petición ***PUT*** para asignar al participante ID = 25, el nombre “Roger” y la altura “170.56”.



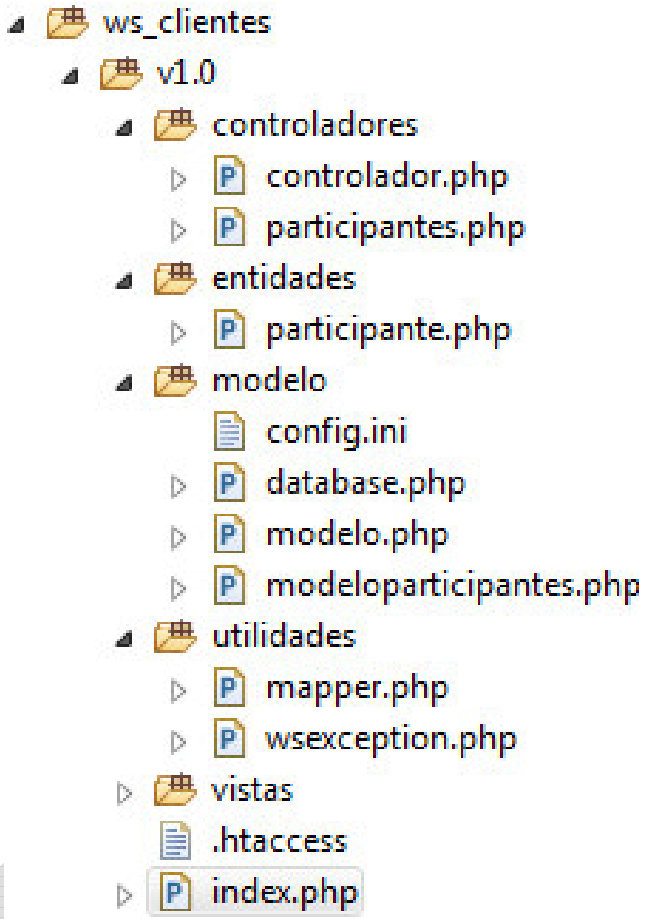

La respuesta devuelve un objeto con el nº de registros modificados en el atributo “*rows*”. Si existe el participante ID = 25, devuelve 1. En caso contrario retorna 0.



***Código Completo:*** *Controladores/Participantes*

```php
<?php
namespace Controladores;
use Utilidades\WSException;
use Utilidades\Mapper;
use Modelo\ModeloParticipantes as Modelo;
class Participantes extends Controlador {
        public static function get($peticion) {
                if (empty($peticion[0]))
                        // GET /participantes
                        return Modelo::listar();
                else
                        // GET /participantes/:id
                        return Modelo::obtener($peticion[0]);
        }
        public static function post()   {
                // POST: /participantes
                // Obtencion de los datos inscritos en la peticion
                $body = file_get_contents('php://input');
                // Decodificacion del objeto enviado
                $nuevo_participante = json_decode($body);
                // Comprobacion del objeto enviado
                if ( !Mapper::MapCheck($nuevo_participante, ['nombre', 'altura']) )
                        throw new WSException(WSException::ESTADO_JSON_FORMATO_INCORRECTO,
"Formato JSON incorrecto");
                // Insercion en el modelo y obtención del identificador
                $idContacto = Modelo::insertar($nuevo_participante);
                // Retorno de matriz con identificador como respuesta.
                return [
                                "id" => $idContacto
                ];
        }
        public static function put($peticion)   {
                // PUT: /participantes/:id
                // Comprobacion de existencia de parámetro :id
                if ( count($peticion) == 1 ) {
                        // Obtención de parámetro :id
                        $id = $peticion[0];
                        // Obtencion y decodificación de objeto inscritos en la petición
                        $body = file_get_contents('php://input');
                        $participante = json_decode($body);
                        // Comprobacion del objeto enviado
                        if ( !Mapper::MapCheck($participante, ['nombre', 'altura']) )
                                throw new
WSException(WSException::ESTADO_JSON_FORMATO_INCORRECTO, "Formato JSON incorrecto");
                        // Actualizacion en el modelo y obtención nº registros modificados
                        $rowCount = Modelo::actualizar($id, $participante);
                        return [
                                        "rows" => $rowCount
                        ];
                } else throw new WSException(WSException::ESTADO_URL_INCORRECTA, "Se esperaba
un parámetro");
        }
        public static function delete($peticion)        {
                // DELETE: /participantes/:id
                // Comprobacion de existencia de parámetro :id
                if ( count($peticion) == 1) {
                        // Obtención de parámetro :id
                        $id = $peticion[0];
                        // Eliminación de registro
                        $rowCount = Modelo::eliminar($id);
                        return [
                                        "rows" => $rowCount
                        ];
                } else throw new \WSException(\WSException::ESTADO_URL_INCORRECTA, "Se
esperaba un parámetro");
        }

}
```

**Estructura final del servicio web RESTful**



## Implementación de peticiones con parámetros en la URL

Las peticiones de tipo GET se emplean para obtener elementos de un determinado recurso dependiendo de la URL empleada:

- `GET: /recurso` → Devuelve todos los elementos de un recurso
- `GET: /recurso/id` → Devuelve el elemento del recurso con el identificador dado

Sin embargo, en muchas ocasiones es necesario realizar búsquedas más avanzadas incluyendo criterios de filtrado, ordenación y proyección. Estos pueden especificarse como parámetros añadidos a la URL tras el identificador del recurso.

Las búsquedas avanzadas sobre los elementos de un determinado recurso pueden realizarse añadiendo parámetros a la URL para filtrar, ordenar y proyectar.

**Filtrado**

El filtrado consiste en limitar el nº de resultados que se desea obtener indicando una *condición de filtrado* que deben cumplir. Esta condición es referida al valor de una propiedad de los valores. Para indicar la condición de filtrado se emplea un parámetro por cada campo por el que se desea filtrar los resultados

***Ejemplo:*** Supongamos que deseamos obtener todos los participantes cuyo nombre es "Roger":

```
GET: /participantes?nombre=Roger
```

Supongamos que deseamos obtener todos los participantes cuya altura es superior a un 1.50

```
GET: /participantes?altura=>1.50
```

**Ordenación**

La ordenación consiste en indicar el orden en que se desean obtener los resultados empleando los valores de un determinado campo. Para indicar el campo por el que se desea que se ordenen los valores se emplea un parámetro (p.ej "***sort***") con el nombre de alguna propiedad precedido por un signo "+" para indicar orden ascendente, y "-" para orden descendente respectivamente.

***Ejemplo:*** Supongamos que deseamos obtener todos los participantes ordenados alfabéticamente por su nombre:

```
GET: /participantes?sort=+nombre
```

Supongamos que deseamos obtener los participantes empezando por los más altos y terminando por los más bajos:

```
GET: /participantes?sort=-altura
```

**Proyección**

La proyección permite indicar las propiedades que se desean obtener por cada elemento devuelto. Para indicarlas se emplea un parámetro (p.ej: "***fields***") con los nombres de las propiedades separadas por comas:

***Ejemplo:*** Supongamos que deseamos obtener el nombre y el identificador de los participantes registrados:

```
GET: /participantes?fields=id,nombre
```

## Ejemplo de implementación

Vamos a modificar el servicio web RESTful de participantes visto como ejemplo anteriormente para poder realizar peticiones *GET* sobre el recurso *participantes* especificando operaciones filtrado, ordenación y proyección sobre los resultados devueltos.

Lo primero es definir una función que llamaremos ***getParameterURL()*** que retorna los parámetros presentes concatenados al final de la URL. Esta función la definimos dentro del script ***index.php***:

```
function getParametrosURL() {
        $parametros = $_GET;
        unset($parametros['PATH_INFO']);
        return $parametros;
}
```

Esta función retorna una matriz con todos los parámetros concatenados a la URL eliminando el parámetro "***PATH_INFO***" añadido por la reescritura de URLs definida en el fichero ***.htaccess***.

A continuación debemos modificar el método estático ***listar()*** de la clase ***Modelo/ModeloParticipantes*** para que tenga en cuenta los parámetros de filtrado, proyección y ordenación que pueda haber presentes en la URL. Estos son:

- ***fields*** → Indica una lista de columnas que deben mostrarse por cada participante recuperado.
- ***order*** → Indica el nombre de una columna por cuyo valor deben ordenarse los participantes.
- ***campo=valor*** → Indica el valor para una determinada propiedades que debe cumplirse para ser devuelto el participante.

***Ejemplo:*** Supóngase que deseamos obtener el nombre y altura de todos los participantes de nombre "Roger" ordenados del más alto al más bajo.

La consulta tendría la siguiente forma:

*http://Localhost/v1.0/participantes?fields=nombre,altura&order=-altura&nombre=Roger*

El nuevo método ***listar()*** obtiene la matriz con los parámetros concatenados en la URL y compone la consulta a la base de datos según los encontrados.

Primero se localiza el parámetro "*fields*", si se encuentra se almacena en la variable $fields, y se elimina de la matriz de parámetros. Después se localiza el parámetro "*order*", si se encuentra se almacena en la variable $order, y se elimina de la matriz de parámetros. Por último; se emplean el resto de parámetros para el filtrado.

Como los resultados pueden tener cualquier combinación de campos, los registros se obtienen como objetos anónimos empleando el método ***fetch(/PDO::FETCH_OBJ).***

Código: función ***listar()*** en ***Modelo/ModeloParticipantes***:

```
        /**
         * Devuelve todos los participantes almacenados
         * @throws WSException
         * @return array matriz de objetos Entidades/Participantes
         */
        public static function listar() {
                $params = getParametrosURL();
                // Proyeccion -> parametro FIELDS
                if ( isset($params['fields'])) {
                        $fields = $params['fields'];
                        unset($params['fields']);
                } else $fields = "ID, NOMBRE, ALTURA ";
                // Ordenacion
                if ( isset($params['order'])) {
                        $order = " ORDER BY ".substr($params['order'], 1);
                        if ( substr($params['order'], 0, 1) == "-") {
                                $order.=" DESC";      // Partícula para orden descendente.
                        }
                        unset($params['order']);
                } else $order = "";
                // filtrado
                if ( !empty($params)) {
                        $where = "WHERE ";
                        foreach($params as $campo => $valor ) {
                                $where .= $campo. " = '".$valor."' AND ";
                        }
                        $where = substr($where, 0, -4);  // Quita el ultimo AND sobrante.
                } else $where = "";
                // Composicion de la consulta
                $sql = "SELECT $fields FROM PARTICIPANTES $where $order";
                try {
                        $datos = array();
                        $db = Database::instance();
                        $query = $db->run($sql);
                        while( $reg = $query->fetch(\PDO::FETCH_OBJ)) {
                                array_push($datos, $reg);
                        }
                        return $datos;
                } catch(\PDOException $e) {
                        throw new WSException(WSException::ESTADO_ERROR_BD, $e->getMessage());
                }
        }
```

# *Ejercicio*

Se pide implementar mediante PHP un servicio web RESTful con los siguientes recursos y especificaciones:

- ***Recurso Usuarios*** → Permite el acceso a los usuarios registrados para comprobar la autentificación del usuario que inicia la aplicación.

- ***Recurso Tareas*** → Permite el acceso a las tareas asignadas a un determinado usuario, o a una tarea en concreto, permitiendo la inserción y eliminación.

**Recurso Usuarios:**

**HTTP_GET**: Requiere el paso de dos parámetros por URL con el mail y la contraseña del usuario mediante los parámetros "***mail***" y "***pass***".

```
GET: /usuarios?mail=none@none.com&pass=none
```

El servicio web devuelve un objeto ***Usuario*** codificado en *JSON*. Si el mail y la contraseña son correctos, el objeto contiene todos los datos del usuario incluyendo el identificador interno:

```
{
    "id": "1",
    "mail": "none@none.com",
    "pass": "none"
}
```

En caso contrario devuelve un objeto codificado en JSON con un único atributo "id" con valor -1.

```
{
    "id": -1
}
```

**Recurso Tareas:**

**HTTP_GET:** Requiere el paso de dos parámetros por URL:

Parámetro "***user***" → Indica el identificador numérico del usuario del que desean recuperarse las tareas asignadas. ( *obligatorio* )

Parámetro "***date***" → Indica un valor numérico con la fecha actual de UNIX (*timestamp*) actual para recuperar las tareas asignadas al usuario en la fecha/hora indicadas.

***Ejemplo:*** Obtener todas las tareas asignadas al usuario con ID = 1.

```
GET: /tareas?user=1
```

***Ejemplo:*** Obtener todas las tareas asignadas al usuario con ID = 1 para la fecha/hora indicada por el valor UNIX: 1298217600 = ( *2012-2-20 17:00:00* )

```
GET: /tareas?user=1&date=1298217600
```

El servicio web devuelve una matriz de objetos ***Tarea*** codificada en *JSON*.


```
-0:  {
     "id": "14",
     "id_usuario": "1",
     "titulo": "uno",
     "fecha": 1292134250,
     "importancia": "1",
     "descripcion": "lsks"
 },
-1:  {
     "id": "27",
     "id_usuario": "1",
     "titulo": "cero",
     "fecha": 1422545065,
     "importancia": "2",
     "descripcion": "cero"
 },
-2:  {
     "id": "45",
     "id_usuario": "1",
     "titulo": "bingo",
     "fecha": 1434465000,
     "importancia": "2",
     "descripcion": "vy"
 },
```


Si el usuario no tiene ninguna tarea asignada, el servicio web retorna una matriz vacía:

```
[Array[0]],
```

**HTTP_POST:** Requiere un objeto ***Tarea*** integrado en la petición codificado en JSON. No se incluye la propiedad "id" puesto que ésta es generada por la base de datos al almacenarse la tarea.

***Ejemplo:*** objeto *Tarea* codificado en *JSON* enviado por petición *POST*:

***POST: /tareas***

```
{
        "id_usuario": "1",
        "titulo": "tarea de prueba de tiempo",
        "fecha":"1298217600",
        "importancia":"1",
        "descripcion":"Es una tarea de prueba"
}
```

El servicio retorna el mismo objeto ***Tarea*** codificado en JSON con el identificador correspondiente asignado en su atributo *id*. En caso de error retorna ***null***.


```
{
    "id_usuario": "1",
    "titulo": "tarea de prueba de tiempo",
    "fecha": "2011-02-20 17:00:00",
    "importancia": "1",
    "descripcion": "Es una tarea de prueba",
    "id": "56"
}
```

***Detalles importantes:***

- El atributo "***fecha***" debe pasarse como un valor numérico entero largo (*long*). Este puede obtenerse del método *getTime()* de *Date*, o *getTimeInMillis()* de *Calendar*.

- El atributo "***id_usuario***" de la tarea debe coincidir con el valor del atributo "*id*" del usuario que ha iniciado sesión. Este valor se obtiene al iniciar sesión el usuario.

**HTTP_GET**: Devuelve los datos de la tarea asociada al identificador interno indicado como parte de la URL.

***Ejemplo:*** Obtención de datos de la tarea con ID = 3

```
GET: /tareas/3
```

El servicio web devuelve el objeto ***Tarea*** correspondiente codificado en *JSON*.

```
{
    "id": "3",
    "id_usuario": "2",
    "titulo": "Pagar el agua",
    "fecha": 1298217600,
    "importancia": "2",
    "descripcion": "Llevar el dinero"
}
```

En caso de no existir ninguna tarea registrada con el identificador indicado retorna un objeto codificado en JSON con un único atributo "id" con valor -1:

```
{
    "id": -1
}
```

**HTTP_DELETE:** Elimina del servidor la tarea asociada al identificador interno indicado como parte de la URL.

***Ejemplo:*** Eliminación de la tarea con ID = 50

```
DELETE: /tareas/3
```

El servicio web devuelve un número ***1*** sin codificación si la tarea es eliminada satisfactoriamente. Si el identificador interno indicado en la URL no coincide con el de ninguna tarea registrada retorna un valor ***0***.

## Origen de datos

Para implementar este servicio web puedes utilizar la siguiente base de datos:

```
[database]
host     = 192.168.9.180
user     = alumno
password = cipsa
database = tareas
```

La base de datos está compuesta de dos tablas con los siguientes campos:

Tabla ***usuarios***:

| # | Nombre | Tipo | Cotejamiento | Atributos | Nulo | Predeterminado | Extra |
|---|---|---|---|---|---|---|---|
| 1 | **id** | int(11) | | | No | *Ninguna* | AUTO_INCREMENT |
| 2 | **mail** | varchar(45) | | | No | *Ninguna* | |
| 3 | **pass** | varchar(45) | | | No | *Ninguna* | |

Tabla ***tareas***

| # | Nombre | Tipo | Cotejamiento | Atributos | Nulo | Predeterminado | Extra |
|---|---|---|---|---|---|---|---|
| 1 | **id** | int(11) | | | No | *Ninguna* | AUTO_INCREMENT |
| 2 | **mail** | varchar(45) | | | No | *Ninguna* | |
| 3 | **pass** | varchar(45) | | | No | *Ninguna* | |