1 Slim Framework - API REST

1.1 Sumario

2 Introducción a REST

REST al igual que otras tecnologías, como Git y CSS, requiere un poco de tiempo para su comprensión. En este breve manual nos centraremos en la filosofía que está detrás de REST(así como sus diferencias con SOAP), y en los aspectos prácticos. ¿Cómo podemos implementar REST hoy?

Ver conceptos sobre API REST

2.1 ¿Que es es REST?

REST son las siglas de **Representational State Transfer**. Fue definido hace una década por Roy Fielding en su tesis doctoral, y proporciona una forma sencilla de interacción entre sistemas, la mayor parte de las veces a través de un navegador web y HTTP. Esta cohesión con HTTP viene también de que Roy es uno de los principales autores de HTTP.

REST es un estilo arquitectónico, un conjunto de convenciones para aplicaciones web y servicios web, que se centra principalmente en la manipulación de recursos a través de especificaciones HTTP. Podemos decir que REST es una interfaz web estándar y simple que nos permite interactuar con servicios web de una manera muy cómoda.

Gracias a REST la web ha disfrutado de escalabilidad como resultado de una serie de diseños fundamentales clave:

- Un **protocolo cliente/servidor sin estado**: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión (algunas de estas prácticas, como la reescritura de URLs, no son permitidas por REST).
- Un **conjunto de operaciones bien definidas** que se aplican a todos los recursos de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son **POST**, **GET**, **PUT** y **DELETE**. Con frecuencia estas operaciones se equiparan a las operaciones CRUD que se requieren para la persistencia de datos, aunque POST no encaja exactamente en este esquema.

- Una **sintaxis universal para identificar los recursos**. En un sistema REST, cada recurso es direccionable únicamente a través de su URI.

- El **uso de hipermedios**, tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST son típicamente HTML o XML. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

Vamos a ver primero lo que son las URIs. Una URI es esencialmente un identificador de un recurso. Veamos el siguiente ejemplo:

```
GET /amigos
```

Podríamos llamar a esto un recurso. Cuando esta ruta es llamada, siguiendo los patrones REST, se obtendrán todos los amigos (generalmente de una base de datos), y se mostrarán en pantalla o se devolverán en un formato determinado a quien lo solicite.

Pero, cómo haremos si queremos especificar un amigo en particular?.

```
GET /amigos/marta
```

Como se puede ver es fácilmente comprensible. Esa es una de las claves de la arquitectura RESTful. Permite el uso de URIs que son fácilmente comprensibles por los humanos y las máquinas.

Piensa en un recurso como un nombre en plural. Contactos, estados, usuarios, fotos --- todos éstos serían elecciones perfectas.

Hasta ahora, hemos visto como identificar a una colección y a elementos individuales en esa colección:

```
GET /amigos
GET /amigos/marta
```

De hecho, encontrarás que estos dos segmentos son todo lo que tendrías que haber necesitado siempre. Pero podemos profundizar un poco más en la potencia de HTTP para indicar cómo queremos que el servidor responda a esas peticiones. Veamos:

Cada petición HTTP especifica un método, o un verbo, en sus encabezados. Generalmente te sonarán un par de ellos como GET y POST.

Por defecto el verbo utilizado cuando accedemos o vemos una página web es **GET**.

Para cualquier URI dada, podemos referenciar hasta 4 tipos diferentes de métodos: **GET, POST, PUT, PATCH y DELETE**.

```
GET /amigos
POST /amigos
PUT /amigos
DELETE /amigos
```

Esencialmente, estos verbos HTTP indican al servidor que hacer con los datos especificados en la URI. Una forma fácil de asociar estos verbos a las acciones realizadas, es comparándolo con **CRUD** (Create-Read-Update-Delete).

```
GET => READ
POST => CREATE
PUT => UPDATE
DELETE => DELETE
```

Anteriormente hemos dicho que GET es el método utilizado por defecto, pero también te debería sonar POST. Cuando enviamos datos desde un formulario al servidor, solemos utilizar el método POST. Por ejemplo si quisiéramos añadir nuevos Tweets a nuestra base de datos, el formulario debería hacer un POST de los tweets POST /tweets, en lugar de hacer /tweets/añadirNuevoTweet.php.

Ejemplos de URIs que son no RESTful y que no se recomienda utilizar:

```
/tweets/añadirNuevoTweet.php
 /amigos/borrarAmigoPorNombre.php
 /contactos/actualizarContacto.php
```

Ejemplos de URIs que son RESTful y que serían un buen ejemplo:

```
GET /tickets/12/messages - Devuelve una lista de mensajes para el ticket #12
 GET /tickets/12/messages/5 - Devuelve el mensaje #5 para el ticket #12
 POST /tickets/12/messages - Crea un nuevo mensaje en el ticket #12
 PUT /tickets/12/messages/5 ? Actualiza el mensaje #5 para el ticket #12
 PATCH /tickets/12/messages/5 - Actualiza parcialmente el mensaje #5 para el ticket #12
 DELETE /tickets/12/messages/5 - Borra el mensaje #5 para el ticket #12
```

¿Pero entonces, cuáles serían las URIs correctas para presentar un formulario al usuario, con el objetivo de añadir o editar un recurso?

En situaciones como esta, tiene más sentido añadir URIs como:

```
GET /amigos/nuevo
GET /amigos/marta/editar
```

La primera parte de la trayectoria /amigos/nuevo, debería presentar un formulario al usuario para añadir un amigo nuevo. Inmediatamente después de enviar el formulario, debería usarse una solicitud POST, ya que estamos añadiendo un nuevo amigo.

Para el segundo caso /amigos/marta/editar, este formulario debería editar un usuario existente en la base de datos. Cuando actualizamos los datos de un recurso, se debería utilizar una solicitud PUT.

Más información de cómo nombrar las URI en una API REST:

http://www.restapitutorial.com/lessons/restfulresourcenaming.html

Otro libro recomendable sobre RESTful: http://restcookbook.com/

2.2 Verbos disponibles en REST

Antes de seguir adelante con ejemplos concretos, vamos a revisar un poco más los verbos utilizados en las peticiones a una API REST.

2.2.1 GET (Recuperar)

GET es el método HTTP utilizado por defecto en las peticiones web. Una advertencia a tener en cuenta es que deberíamos utilizar GET, para hacer peticiones sólo de lectura, y deberíamos obtener siempre el mismo tipo de resultado, independientemente de las veces que sea llamado ese método.

Como programador puedes hacer lo que quieras cuando se hace una llamada a las rutas en la URI, pero una buena práctica es seguir las reglas generales para diseñar una API REST correctamente.

2.2.2 POST (Crear)

El segundo método que te resultará familiar es POST. Se utilizará para indicar que vamos a crear un subconjunto del recurso especificado, o también si estamos actualizando uno o más subconjuntos del recurso especificado.

Por ejemplo vamos a crear un recurso nuevo por ejemplo enviando un nuevo usuario para darlo de alta, entonces lo haremos a la URL /amigos

```
POST /amigos
```

2.2.3 PUT (Actualizar)

Utiliza PUT cuando quieras actualizar un recurso específico a través de su localizador en la URL.

Por ejemplo si un artículo está en la URL http://miweb.local/api/articulos/1333, podemos actualizar ese recurso enviando todos los campos a actualizar desde un formulario (método POST):

```
PUT /api/articulos/1333
```

Si no conocemos la dirección del recurso actual, for ejemplo para añadir un nuevo artículo, entonces utilizaríamos la acción POST. Por ejemplo.

```
POST /api/articulos
```

2.2.4 DELETE (Borrado)

Por último DELETE debería se usado cuando queremos borrar el recurso especificado en la URI. Por ejemplo si ya no somos más amigos de macarena, siguiendo los principios de REST, podríamos borrarla usando una petición delete a la URI:

```
DELETE /amigos/macarena
```

2.2.5 PATCH (Actualizaciones parciales)

Una solicitud de tipo PATCH se utiliza para realizar una actualización parcial de un recurso es decir para actualizar ciertos campos del recurso y no el recurso al completo. Los campos a actualizar se enviarían desde un formulario por POST y el tipo de petición es PATCH.

```
PATCH /api/articulos/1333
```

2.2.6 HEAD (Solicitud de cabeceras)

- Una solicitud de tipo HEAD es como una solicitud GET, con la salvedad que solamente se devuelven las cabeceras HTTP y el código de respuesta, no el documento en sí mismo.
- Con este método el navegador puede comprobar si un documento ha sido modificado, por temas de caché por ejemplo. También puede comprobar si un documento existe o no.
- Por ejemplo, si tienes un montón de enlaces en tu web, periódicamente podrías comprobar mediante peticiones HEAD si los hiperenlaces son correctos o están rotos. Éste tipo de comprobación es muchísimo más rápido que usar GET.
- **Más información sobre cabeceras HTTP en: http://code.tutsplus.com/tutorials/http-headers-for-dummies--net-8039**

2.3 Buenas prácticas en el diseño de una API REST

Consulta la siguiente dirección dónde se muestran guías y buenas prácticas para la creación de una API REST para nuestra aplicación:

http://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-restful-pragmatica/

http://restcookbook.com/

2.4 Creación de una RESTFUL API o API REST

Para crear una API REST o RESTFUL API (en inglés) en php podremos hacerlo utilizando un fichero .htaccess dónde programamos todos los tipos de URI's que gestionaremos en la API o bien utilizando un framework que nos facilite dicha programación.

Veremos para ello el Slim micro Framework, que es bastante sencillo y nos facilita muchísimo este tipo de creación, con lo que podremos crear una API REST para cualquier aplicación que ya tengamos programada de forma muy sencilla.

2.4.1 Rutas de la API REST que vamos a programar

#	Ruta	Método	Tipo	Ruta Completa	Descripción
1	/usuarios	GET	JSON	http://tudominio.com/api/v1/usuarios/	Obtener información de todos los usuarios.
2	/usuarios/{id}	GET	JSON	http://tudominio.com/api/v1/usuarios/1	Obtener información del usuario con el ID proporcionado.
3	/crear	POST	JSON	http://tudominio.com/api/v1/crear/	Crear un nuevo usuario en la base de datos.
4	/actualizar/{id}	PUT	JSON	http://tudominio.com/api/v1/actualizar/1	Actualizar información del empleado con el ID proporcionado.
5	/eliminar/{id}	DELETE	JSON	http://tudominio.com/api/v1/eliminar/1	Eliminar el usuario con el ID proporcionado.

2.4.2 Código fuente de la API REST con Slim framework

Hay que cargar los módulos correspondientes dentro de la carpeta del proyecto.

En el ejemplo dentro de una carpeta llamada: proyecto3

```
# Cargamos los módulos correspondientes:
# Instalamos el framework
composer require slim/slim:3.*

# Instalamos el gestor de plantillas php/view

# Instalamos el módulo para mensajes Flash.
composer require slim/flash

# Creamos las carpetas src/templates

# La estructura de directorios será:

|-- proyecto3
    |-  docs
          slimframework.sql
    |-  public
          index.php
    |-  src
        |- templates
               listadousuarios.php
               nuevousuario.php
```

```
        basedatos.php
        config.php

    |- vendor
```

FICHERO proyecto3/public/index.php:

```
<?php
// proyecto3/public/index.php
session_start();

/*
# Instalamos el framework
composer require slim/slim:3.*

# Instalamos el gestor de plantillas php/view
composer require slim/php-view

# Instalamos el módulo para mensajes Flash.
composer require slim/flash

// https://github.com/slimphp/Slim-Flash
*/


use Psr\Http\Message\ResponseInterface as Response;
use Psr\Http\Message\ServerRequestInterface as Request;

# Ajustar a la carpeta dónde tengamos vendor.
require __DIR__ . '/../vendor/autoload.php';

// Cargamos la clase basedatos.php
require __DIR__ . '/../src/basedatos.php';

// Nos conectamos a la base de datos
$pdo = Basedatos::getConexion();

// Instanciamos la aplicación.
$app = new \Slim\App();

// Get container
$container = $app->getContainer();

// Registramos las vistas
$container['view'] = new \Slim\Views\PhpRenderer('../src/templates/');

// Registramos los mensajes flash
$container['flash'] = new \Slim\Flash\Messages();

// Definimos rutas de la aplicación
$app->get('/', function (Request $request, Response $response) {
    $response->getBody()->write('Ejemplo de API REST con Slimframework.<br/><a href="https://manuais.iessanclemente.net/index.php/Sl
    return $response;
})->setName('root');

// Creación del grupo de rutas de la API.
$app->group('/api', function () use ($app) {
    // Versionado de la API
    $app->group('/v1', function () use ($app) {
        $app->get('/usuarios', 'obtenerUsuarios');
        $app->get('/usuarios/{id}', 'obtenerUsuario');
        $app->post('/usuarios/create', 'crearUsuario');
        $app->put('/usuarios/{id}', 'actualizarUsuario');
        $app->delete('/usuarios/{id}', 'eliminarUsuario');
    });
});

/*
Otra forma programando el código dentro de la ruta:
$app->get('/usuarios', function() use ($pdo)
{
// Si necesitamos acceder a alguna variable global en el framework
// Tenemos que pasarla con use($variable) en la cabecera de la función.
```

```
// Va a devolver un objeto JSON con los datos de usuarios.
$stmt = $pdo->prepare("select * from usuarios");
$stmt->execute();

// Almacenamos los resultados en un array asociativo.
$resultados = $stmt->fetchAll(PDO::FETCH_ASSOC);

// Devolvemos ese array asociativo como un string JSON.
echo json_encode($resultados);
});
 */

function obtenerUsuarios(Request $request, Response $response)
{
    global $pdo;
    // Si necesitamos acceder a alguna variable global en el framework
    // Tenemos que pasarla con use($variable) en la cabecera de la función.
    // Va a devolver un objeto JSON con los datos de usuarios.
    try {

        // Preparamos la consulta a la tabla.
        $stmt = $pdo->prepare("select * from usuarios");
        $stmt->execute();
        // Almacenamos los resultados en un array asociativo.
        $resultados = $stmt->fetchAll(PDO::FETCH_ASSOC);
        // Devolvemos ese array asociativo como un string JSON.
        return $response->withJson($resultados, 200);
    } catch (PDOException $e) {
        $datos = array('status' => 'error', 'data' => $e->getMessage());
        return $response->withJson($datos, 500);
    }
}


function obtenerUsuario(Request $request, Response $response)
{
    global $pdo;
    // Si necesitamos acceder a alguna variable global en el framework
    // Tenemos que pasarla con use($variable) en la cabecera de la función.
    // Va a devolver un objeto JSON con los datos de usuarios.

    try {
        // Preparamos la consulta a la tabla.
        $stmt = $pdo->prepare("select * from usuarios where id=?");
        $id = $request->getAttribute('id');
        $stmt->bindParam(1, $id);
        $stmt->execute();

        if ($stmt->rowCount() != 0) {
            // Almacenamos los resultados en un array asociativo.
            $resultados = $stmt->fetchAll(PDO::FETCH_ASSOC);
            // Devolvemos ese array asociativo como un JSON con Status 200

            return $response->withJson($resultados, 200);
        } else {
            $datos = array('status' => 'error', 'data' => "No se ha encontrado el usuario con ID: $id.");
            return $response->withJson($datos, 404);
        }
    } catch (PDOException $e) {
        $datos = array('status' => 'error', 'data' => $e->getMessage());
        return $response->withJson($datos, 500);
    }
}

function crearUsuario(Request $request, Response $response)
{
    global $pdo;

    // Si necesitamos acceder a alguna variable global en el framework
    // Tenemos que pasarla con use($variable) en la cabecera de la función.
    // Va a devolver un objeto JSON con los datos de usuarios.

    $campos = $request->getParsedBody();
```

```
    try {
        // Preparamos la consulta a la tabla.
        $stmt = $pdo->prepare("insert into usuarios(nombre,apellidos,sueldo,edad) values(?,?,?,?)");
        $stmt->bindParam(1, $campos['nombre']);
        $stmt->bindParam(2, $campos['apellidos']);
        $stmt->bindParam(3, $campos['sueldo']);
        $stmt->bindParam(4, $campos['edad']);
        $stmt->execute();

        $datos = array('status' => 'ok', 'data' => 'Usuario dado de alta correctamente.');
        return $response->withJson($datos, 200);
    } catch (PDOException $e) {
        $datos = array('status' => 'error', 'data' => $e->getMessage());
        return $response->withJson($datos, 500);
    }
}

function actualizarUsuario(Request $request, Response $response)
{
    global $pdo;

    // Si necesitamos acceder a alguna variable global en el framework
    // Tenemos que pasarla con use($variable) en la cabecera de la función.
    // Va a devolver un objeto JSON con los datos de usuarios.

    $campos = $request->getParsedBody();

    try {
        // Preparamos la consulta a la tabla.
        $id = $request->getAttribute('id');

        $stmt = $pdo->prepare("select * from usuarios where id=?");
        $stmt->bindParam(1, $id);
        $stmt->execute();
        if ($stmt->rowCount() != 0) {

            $stmt = $pdo->prepare("update usuarios set nombre=?,apellidos=?,sueldo=?,edad=? where id=?");
            $stmt->bindParam(1, $campos['nombre']);
            $stmt->bindParam(2, $campos['apellidos']);
            $stmt->bindParam(3, $campos['sueldo']);
            $stmt->bindParam(4, $campos['edad']);
            $stmt->bindParam(5, $id);
            $stmt->execute();

            // Devolvemos ese array asociativo como un JSON con Status 200
            $datos = array('status' => 'ok', 'data' => 'Actualizado correctamente');
            return $response->withJson($datos, 200);
        } else {
            $datos = array('status' => 'error', 'data' => "No se ha encontrado el usuario con ID: $id.");
            return $response->withJson($datos, 404);
        }
    } catch (PDOException $e) {
        $datos = array('status' => 'error', 'data' => $e->getMessage());
        return $response->withJson($datos, 500);
    }
}

function eliminarUsuario(Request $request, Response $response)
{
    global $pdo;

    // Si necesitamos acceder a alguna variable global en el framework
    // Tenemos que pasarla con use($variable) en la cabecera de la función.
    // Va a devolver un objeto JSON con los datos de usuarios.

    try {
        // Preparamos la consulta a la tabla.
        $id = $request->getAttribute('id');

        $stmt = $pdo->prepare("select * from usuarios where id=?");
        $stmt->bindParam(1, $id);
        $stmt->execute();
```

```
        if ($stmt->rowCount() != 0) {

            $stmt = $pdo->prepare("delete from usuarios where id=?");
            $stmt->bindParam(1, $id);
            $stmt->execute();

            // Devolvemos ese array asociativo como un JSON con Status 204 No Content
            $datos = array('status' => 'ok', 'data' => 'Usuario borrado correctamente');
            return $response->withJson($datos, 204);
        } else {
            $datos = array('status' => 'error', 'data' => "No se ha encontrado el usuario con ID: $id.");
            return $response->withJson($datos, 404);
        }
    } catch (PDOException $e) {
        $datos = array('status' => 'error', 'data' => $e->getMessage());
        return $response->withJson($datos, 500);
    }
}



////////////////////////////////////////////////////////////////////////////////////////////
// A PARTIR DE AQUÍ ES UN EJEMPLO DE USO DE SLIM FRAMEWORK PARA HACER PARTES DE UNA APLICACIÓN.
////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//
// EJEMPLO DE USO DEL SLIM FRAMEWORK PARA GENERAR UNA APLICACIÓN.
//
//
//
// Ésto no formaría parte de la API REST. Ésto sería un ejemplo de aplicación
// que podemos generar con el framework Slim.
// Aquí se muestra un ejemplo de como se generaría una página utilizando vistas.
////////////////////////////////////////////////////////////////////////////

$app->get('/listadousuarios', function (Request $request, Response $response) {
    global $pdo;
    // Va a devolver un objeto JSON con los datos de usuarios.

    // Preparamos la consulta a la tabla.
    $stmt = $pdo->prepare("select * from usuarios");

    // Ejecutamos la stmt (si fuera necesario se le pasan parámetros).
    $stmt->execute();

    $response = $this->view->render($response, 'listadousuarios.php', ['resultados' => $stmt->fetchAll(PDO::FETCH_ASSOC)]);
    return $response;
});


// Cuando accedamos a /nuevousuario se mostrará un formulario de alta.
$app->get('/nuevousuario', function (Request $request, Response $response) {
    $response = $this->view->render($response, 'nuevousuario.php', ['mensajes' => $this->flash->getMessages()]);
    return $response;
})->setName('nuevousuario');


// Ruta que recibe los datos del formulario
$app->post('/nuevousuario', function (Request $request, Response $response) {
    global $pdo;

    // Si se reciben por GET $request->getQueryParams()
    // Si se reciben por POST $request->getParsedBody()
    $datosForm = $request->getParsedBody();
    // Preparamos la consulta de insert.
    $stmt = $pdo->prepare("insert into usuarios(nombre,apellidos,sueldo,edad)
                values (?,?,?,?)");
    $stmt->bindParam(1, $datosForm['nombre']);
    $stmt->bindParam(2, $datosForm['apellidos']);
    $stmt->bindParam(3, $datosForm['sueldo']);
    $stmt->bindParam(4, $datosForm['edad']);

    $estado = $stmt->execute();
```

```
    if ($estado)
        $this->flash->addMessage('mensaje', 'Usuario insertado correctamente.');
    else
        $this->flash->addMessage('error', 'Se ha producido un error al guardar datos.');

    // Redireccionamos al formulario original para mostrar
    // los mensajes Flash.,
    return $response->withRedirect('nuevousuario');
});



// Ejecutamos la aplicación para que funcionen las rutas.
$app->run();
```

FICHERO proyecto3/templates/listadousuarios.php:

```
<!doctype html>
<html lang="es">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css">
    <title>Listado de Usuarios</title>
</head>

<body>
    <h2>Listado de usuarios</h2>
    <ul>
        <?php
        // Aquí recibimos la variable $resultados
        // Que es un array de una posición que contiene en dicha posición otro array con todas las filas
        foreach ($resultados as $clave => $valor) {
            echo '<li>ID: ' . $valor['id'] . ' --> ' . $valor['nombre'] . '' . $valor['apellidos'] . ' --> Sueldo: ' . $valor['sueldo
        }
        ?>
    </ul>
</body>

</html>
```

FICHERO proyecto3/templates/nuevousuario.php:

```
<!doctype html>
<html lang="es">

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css">
    <title>Alta de usuarios en la aplicación</title>
</head>

<body>
    <div class="container">
        <form action="" method="POST" role="form" style="margin:0 auto;max-width:600px;padding:15px;">
            <legend>Alta de Usuarios</legend>
            <?php
            if (isset($mensajes['error'])) : ?>
                <span class="label label-danger"><?php echo $mensajes['error'][0] ?></span>
            <?php endif; ?>

            <div class="form-group">
                <label for="nombre">Nombre</label>
                <input type="text" class="form-control" id="nombre" name="nombre" placeholder="Introduzca nombre">
                <label for="apellidos">Apellidos</label>
                <input type="text" class="form-control" id="apellidos" name="apellidos" placeholder="Introduzca apellidos">
                <label for="apellidos">Sueldo</label>
                <input type="text" class="form-control" id="sueldo" name="sueldo" />
```

```
                <label for="apellidos">Edad</label>
                <input type="text" class="form-control" id="edad" name="edad" />
            </div>

            <div class="form-group" style="height:20px;">
                <?php
                if (isset($mensajes['mensaje'])) : ?>
                    <span class="label label-success"><?php echo $mensajes['mensaje'][0] ?></span>
                <?php endif; ?>
            </div>

            <button type="submit" class="btn btn-primary">Guardar</button>
        </form>
    </div>
</body>

</html>
```

FICHERO proyecto3/basedatos.php:

```
<?php
require_once __DIR__ . '/config.php';

class Basedatos
{

    // Propiedad estática dónde almacenaremos la referencia de la conexión
    // a la base de datos mariadb.
    private static $conexion = false;

    private function __construct()
    {
        try {
            $cadenaConexion = "mysql:host=" . DB_SERVIDOR . ";port=" . DB_PUERTO . ";dbname=" . DB_BASEDATOS . ";charset=utf8";
            self::$conexion = new PDO($cadenaConexion, DB_USUARIO, DB_PASSWORD);
            self::$conexion->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
            self::$conexion->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
        } catch (PDOException $e) {
            die("Error conectando a servidor de base de datos: " . $e->getMessage());
        }
    }


    public static function getConexion()
    {
        // Comprobamos si existe una conexión.
        if (!self::$conexion) {
            new self;
            // otra opción:
            // self::__construct();
        }

        return self::$conexion;
    }
}
```

FICHERO proyecto3/config.php:

```
<?php
define('DB_SERVIDOR', 'localhost');
define('DB_PUERTO', '3306');
define('DB_BASEDATOS', 'slimframework');
define('DB_USUARIO', 'slimframework');
define('DB_PASSWORD', 'abc123..');
```

Para probar la aplicación podemos ejecutar el servidor web desde el terminal de Linux:

```
# Desde dentro de la carpeta proyecto3

php -S 0.0.0.0:8080 -t public
```

```
# Para probar las rutas y URL iremos a:
http://veiga.dynu.net:8080/
```

Puedes descargarte los ficheros fuentes de ejemplo junto con Slim framework aqui: Archivo:REST Slimframework.zip

2.5 Extensión Advanced REST Client de Google Chrome

En el ejemplo anterior hemos visto que para probar la API REST hemos hecho un formulario con peticiones AJAX utilizando las diferentes acciones disponibles.

Una forma más sencilla de comprobar si nuestra API REST funciona correctamente es instalando una extensión de Google Chrome llamada **"Advanced REST Client"** la cuál nos permite simular peticiones de cualquier tipo a la URL que le indiquemos.

Instalad la extensión para Google Chrome Advanced REST Client desde aquí.

Video en Youtube de ejemplo de uso de Advanced REST Client:

2.6 Estándar a usar a la hora de devolver datos en formato JSON desde una API REST

En la siguiente URL podréis encontrar amplia información del estándar a seguir a la hora de devolver datos en formato JSON desde una API REST:

http://jsonapi.org/

2.7 Otros frameworks para PHP

Aquí tenéis una URL con enlaces a otros frameworks de PHP más potentes, pero a su vez con una curva de aprendizaje mayor:

http://www.genbetadev.com/frameworks/un-punado-de-frameworks-php-que-te-haran-la-vida-mas-simple

Mi recomendación particular es **LARAVEL**:

- Página oficial del framework Laravel
- Video tutoriales sobre Laravel

Veiga (discusión) 19:17 26 feb 2020 (CET)