

Bien en esta ocasión traigo un API Rest construido en PHP Con MySQL.

El cual consiste en CRUD para una tabla llamada: Usuarios, que consultará o todos los usuarios o un solo usuario, ingresará, actualizará y eliminará un usuario.

Vamos inicialmente a crear nuestra Base de Datos para probar el API REST, esta BD está muy sencilla pues solo tiene una tabla con tres columnas pero por ahora es solo lo que requerimos para probar. La BD yo la llamé: ApiRest\_DB, tú la puedes nombrar como quieras, pero deberás luego modificar la cadena de conexión en la clase dispuesta para ello que se encuentra alojada en la capa DAL. Crearemos una estructura llamada: Usuarios y tres procedimientos almacenados, para gestionar todo directamente en la BD y agilizar las respuestas solicitadas al API, pues si bien el json es ligeramente más eficiente que el xml, todavía los servicios web pecan de ser algo lentos, y una forma de ayudarlo es trabajando con procedimientos almacenados, que trabajan directamente sobre la BD y son muy ventajosos debido al funcionamiento que tienen. En realidad un PA, otorga velocidad a las demandas que hacen los usuarios sobre los datos en concreto, puesto que buscan exactamente lo que éste necesita, y prácticamente da una respuesta inmediata, rápida y eficiente, mejorando así el rendimiento en las aplicaciones, empleando un reducido uso de la red entre clientes y servidores, y brindando seguridad y mantenimiento centralizado además.

```
CREATE DATABASE ApiRest_DB CHARACTER SET utf8 COLLATE utf8_general_ci;
```

```
CREATE TABLE `usuarios` (  
  `Usuario_Id` int(4) NOT NULL AUTO_INCREMENT,  
  `Nombres` varchar(50) NOT NULL,  
  `Apellidos` varchar(50) NOT NULL,  
  PRIMARY KEY (`Usuario_Id`)  
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
INSERT INTO USUARIOS (NOMBRES, APELLIDOS) VALUES ('Lebron', 'James');  
INSERT INTO USUARIOS (NOMBRES, APELLIDOS) VALUES ('Stephen', 'Curry');  
INSERT INTO USUARIOS (NOMBRES, APELLIDOS) VALUES ('James', 'Harden');
```

```
CREATE PROCEDURE apirest_db.`spr_Listados`(  
  IN p_Usuario_Id int(4))  
BEGIN  
  IF (p_Usuario_Id <> 0) THEN  
    SELECT * FROM USUARIOS WHERE USUARIO_ID = p_Usuario_Id;  
  ELSE  
    SELECT * FROM USUARIOS;  
  END IF;  
END;
```

```

CREATE PROCEDURE apiREST_db.`spr_DUusuario`(
IN p_Usuario_Id int(4),
OUT p_Resultado tinyint(1))
BEGIN
    DECLARE v_DatoExiste tinyint(1) DEFAULT 0;
    DECLARE exit HANDLER
    FOR SQLEXCEPTION
    BEGIN
        SET p_Resultado = -1;
        ROLLBACK;
    END;

    SET p_Resultado = 0;
    SELECT COUNT(*) INTO v_DatoExiste FROM USUARIOS WHERE USUARIO_ID
= p_Usuario_Id;
    IF (v_DatoExiste > 0) THEN
        START TRANSACTION;
        DELETE FROM USUARIOS
        WHERE USUARIO_ID = p_Usuario_Id;

        COMMIT;
    ELSE
        SET p_Resultado = -2;
    END IF;
END;

```

```

CREATE PROCEDURE apiREST_db.`spr_IUUsuarios`(
IN p_Usuario_Id INT (4),
IN p_Nombres varchar(50),
IN p_Apellidos varchar(50),
OUT p_Resultado tinyint(1))
BEGIN
    DECLARE v_DatoExiste tinyint(1) DEFAULT 0;
    DECLARE exit HANDLER
    FOR SQLEXCEPTION
    BEGIN
        SET p_RESULTADO = -1;
        ROLLBACK;
    END;

    SET p_Resultado = 0;
    IF (p_Usuario_Id = 0) THEN
        START TRANSACTION;
        INSERT INTO USUARIOS (NOMBRES,APELLIDOS) VALUES (p_Nombres,
p_Apellidos);
        COMMIT;
    ELSE
        SELECT COUNT(*) INTO v_DatoExiste FROM USUARIOS WHERE
USUARIO_ID = p_Usuario_Id;
        IF (v_DatoExiste > 0) THEN
            START TRANSACTION;
            UPDATE USUARIOS SET
            NOMBRES = p_Nombres,
            APELLIDOS = p_Apellidos
            WHERE USUARIO_ID = p_Usuario_Id;
            COMMIT;
        ELSE
            SET p_RESULTADO = -2;
        END IF;
    END IF;
END;

```

Una vez que hayas creado la BD, procederemos a crear el API en PHP, yo trabajé con PHP 7, y Xampp 7.3.1, utilizando el Ide: NetBeans 8.2.

Bien ahora te detallo como quedó configurado el proyecto, para que se entienda la arquitectura y distribución de directorios y carpetas, yo utilicé el patrón MVC donde quedaron 3 paquetes a saber:

**BO - Capa Objetos de Negocio:** Acá queda alojado nuestro único objeto de negocio llamado: Usuarios, que es la clase que representa esta estructura en BD, y que solo tiene sus atributos privados con sus respectivos get y set.

**BLL - Capa Lógica de Negocio:** En esta alojamos toda la lógica propia del mundo que se pretende modelar, donde quedarán nuestros controladores. En realidad es una capa que sirve de enlace entre las vistas y los modelos (BO), respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación. Sin embargo, su responsabilidad no es manipular directamente datos, ni mostrar ningún tipo de salida, sino servir de puente entre modelos y vistas, acá en esta capa quedan tres clases:

- **AccesoDatosFactory:** Aquí emplearemos el patrón de Creación: Factory Method al cual le pasaremos como argumento la Interfaz IAccesoDatos, e invocaremos el método: CrearControlador alojado en la clase Funciones que nos devolverá, ya la clase AccesoDatos instanciada.
- **Controlador:** Como su nombre lo indica es el controlador manager general de todo el sistema. Todo tiene que pasar por esta clase.
- **Funciones:** Esta es una clase transversal, desde donde se instancia el Factory Method.
- **TokenForceAPI:** Es el servicio como tal, y es quien recibe las peticiones, del usuario, el cual solo tiene por ahora 5 métodos.

**DAL - Capa Acceso a Datos:** Es la capa que contiene todos los mecanismos de acceso a nuestra BD, acá en esta capa quedan tres clases a saber:

- **Conexion:** Clase que conecta con el proveedor de Base de Datos-MySQL.
- **IAccesoDatos:** Interfaz que contiene una colección de métodos abstractos y solo los expone.
- **AccesoDatos:** Implementa la interfaz con todos sus métodos.

**Importante:** En la raíz del proyecto se encuentra el .htaccess que es el fichero de configuración de navegación y enrutamiento de las URL, que es llamado desde el index. Este archivo: no se deja subir a Github, allí se subió como archivo de texto, debe quedar solamente con estas tres líneas:

```
RewriteEngine On
```

```
RewriteRule ^([a-zA-Z_-]*)$ index.php?action=$1
```

```
RewriteRule ^([a-zA-Z_-]*)/([0-9]+) index.php?action=$1&id=$2 [L,QSA]
```

## Funcionamiento

El funcionamiento es simple: la capa de presentación pregunta a la BLL por algún objeto, ésta a su vez puede opcionalmente desarrollar alguna validación, y después llamar al DAL, para que esta conecte con la base de datos y le pregunte por un registro específico, cuando ese registro es encontrado, éste es retornado de la base de datos al DAL, quien empaqueta los datos de la base de datos en un objeto personalizado y lo retorna al BLL, para finalmente retornarlo a la capa de presentación, donde podrá ser visualizado por el cliente en formato json.

Tanto la capa lógica de negocio como la capa de acceso de datos consiguen una referencia a los objetos en el BO. Además, la capa de negocio consigue una referencia a la capa de acceso de datos para toda la interacción de datos.

Los objetos del negocio se colocan en una capa diferente para evitar referencias circulares entre la capa del negocio y la de datos.

Es importante destacar que en la DAL habrá una interface llamada `IAccesoDatos`, que será nuestra puerta de entrada, allí no hay implementación de ningún método, solo se exponen, quien use dicha interface es quien los debe implementar, para nuestro caso el DAL (`AccesoDatos`).

Nos apoyaremos en el patrón de creación: `Factory Method` que nos ayudará a la hora de instanciar la clase DAL, que debe implementar todos los métodos de la interface `IAccesoDatos`, lo que nos ahorrará trabajo ya que el conjunto de clases creadas pueden cambiar dinámicamente.

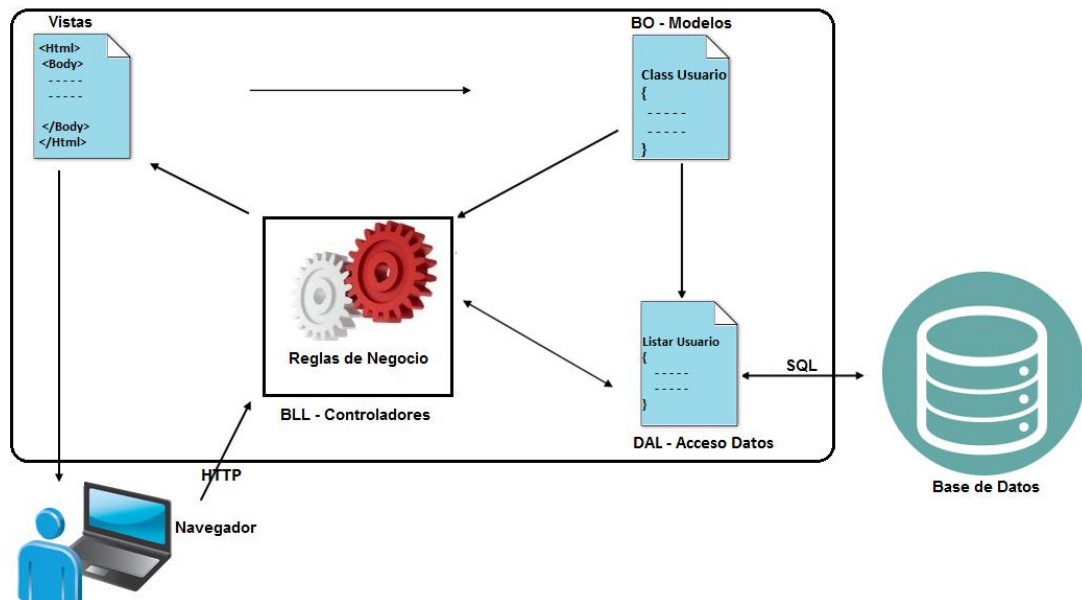
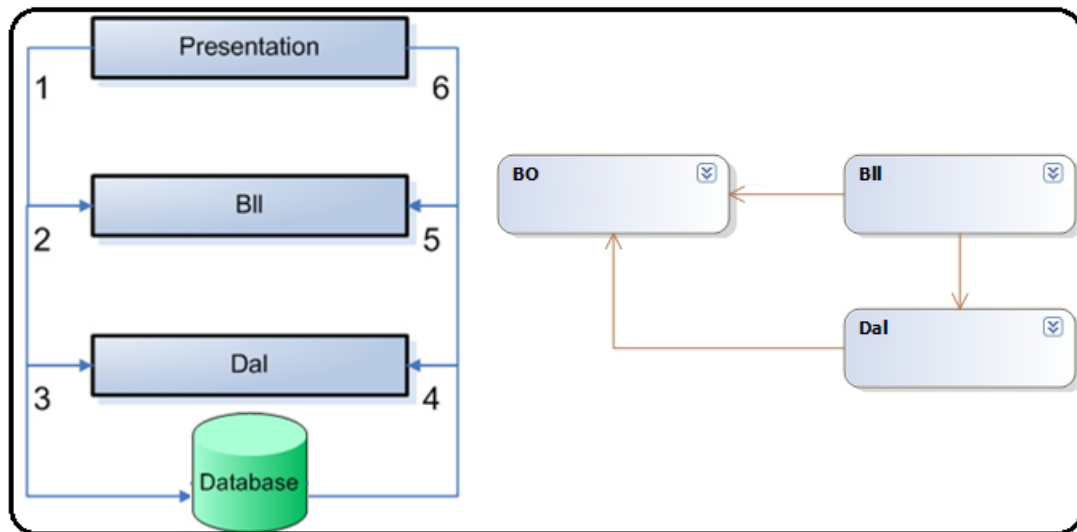
Nunca se llega directamente al DAL, siempre todo deberá pasar primero por el Controlador que será nuestro manager, encargado de orquestrar, controlar y definir que método del DAL invocar, para que sea ésta última quien devuelva los resultados esperados, y dar así respuesta a las diferentes peticiones de la vista, requeridas por el usuario.

El DAL tiene por cada objeto del BO, métodos para obtener listados completos, o un solo ítem o registro, y por supuesto los demás métodos básicos del CRUD, para creación, actualización y eliminación de registros.

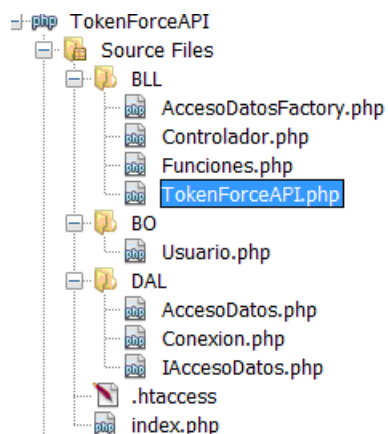
La finalidad de separar los sistemas en capas, no es otra que facilitar su posterior mantenimiento, donde cada capa expone servicios que otras aplicaciones o capas pueden consumir, lo que se traduce en una simplicidad conceptual, de alta cohesión y bajo acoplamiento, facilitando así su reutilización, y MVC nos ayuda con esa tarea, este patrón de arquitectura es definitivamente una belleza, por la forma como trabaja.

Predominando una “organización jerárquica tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior.” Garlan & Shaw.

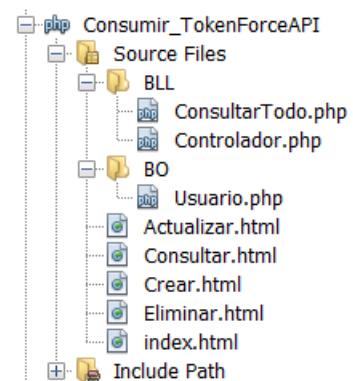
## ESQUEMA DE FUNCIONAMIENTO



### Detalle Proyecto API



### Detalle Proyecto Consumo API



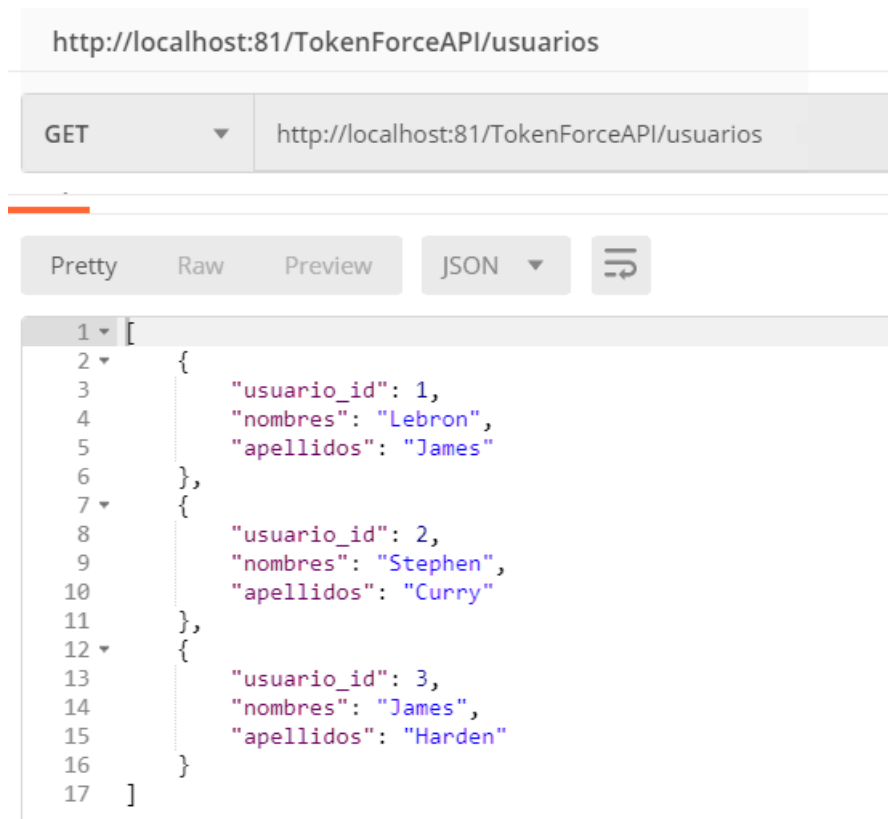
## CONSUMO API CON POSTMAN

Para indicar el path: `http://<host>:<puerto>/<nombre aplicación>/ usuarios`

### Obtener Todos los Usuarios

#### GET

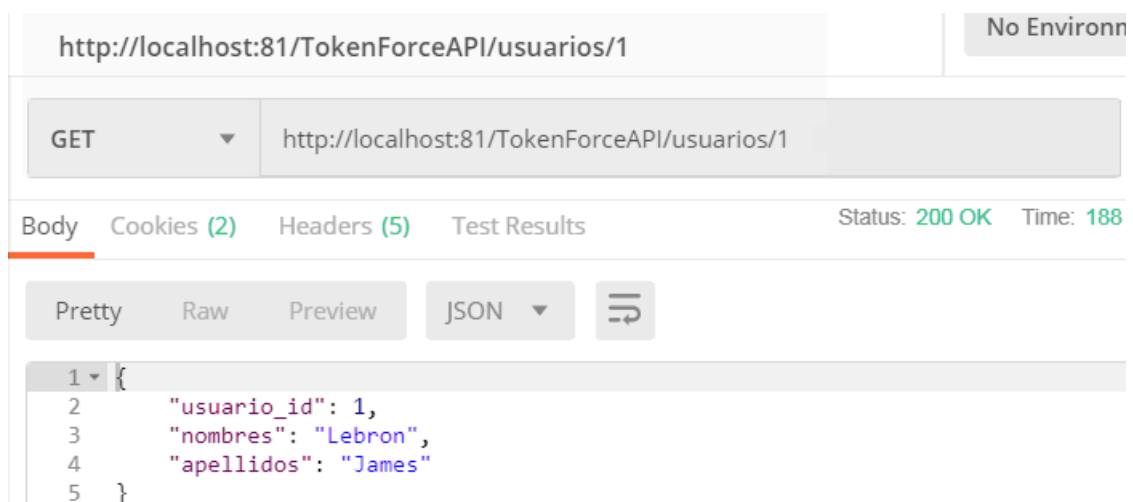
<http://localhost:81/TokenForceAPI/usuarios>



### Obtener Un Usuario

#### GET

<http://localhost:81/TokenForceAPI/usuarios/1>



## Agregar Un Usuario

### POST

<http://localhost:81/TokenForceAPI/usuarios>

The screenshot shows a REST client interface. At the top, a POST request is configured for the URL `http://localhost:81/TokenForceAPI/usuarios. The request body is a JSON object: { "usuario_id": 0, "nombres": "Russell", "apellidos": "West" }. The response status is 200 OK, with a time of 1129 ms and a size of 301 B. The response body is displayed in a Pretty view, showing a JSON object: { "status": "Exito", "message": "El nuevo registro ha sido grabado" }.`

```
POST http://localhost:81/TokenForceAPI/usuarios
```

```
{ "usuario_id": 0, "nombres": "Russell", "apellidos": "West" }
```

Status: 200 OK Time: 1129 ms Size: 301 B

```
{ "status": "Exito", "message": "El nuevo registro ha sido grabado" }
```

## Verificando el Insert en BD

The screenshot shows a SQL query `select * from USUARIOS` being executed in a database client. The results are displayed in a table with columns `Usuario_Id`, `Nombres`, and `Apellidos`. The table contains four rows of data, including the newly inserted user.

| Usuario_Id * | Nombres * | Apellidos * |
|--------------|-----------|-------------|
| 1            | Lebron    | James       |
| 2            | Stephen   | Curry       |
| 3            | James     | Harden      |
| 4            | Russell   | West        |

## Actualizar Un Usuario

### PUT

<http://localhost:81/TokenForceAPI/usuarios>

Para la prueba del PUT cambiaremos el apellido de este registro "West" por "Westbrook", que es el correcto apellido de este famoso jugador de baloncesto

The screenshot shows a REST client interface with a PUT request to `http://localhost:81/TokenForceAPI/usuarios/4`. The request body is a JSON object: `{"usuario_id":4,"nombres":"Russell","apellidos":"Westbrook"}`. The response status is 200 OK, and the response body is: `{"status": "Exito", "message": "El registro ha sido actualizado"}`.

PUT `http://localhost:81/TokenForceAPI/usuarios/4`

Params Authorization Headers (1) Body Pre-request Script Tests

none form-data x-www-form-urlencoded raw binary JSON (applicati

```
1 { "usuario_id":4,
2   "nombres":"Russell",
3   "apellidos":"Westbrook"
4 }
```

body Cookies (2) Headers (5) Test Results Status: 200 OK Time: 9

Pretty Raw Preview Auto

```
1 {
2   "status": "Exito",
3   "message": "El registro ha sido actualizado"
4 }
```

## Verificando el Update en BD

The screenshot shows a SQL query `select * from USUARIOS` executed in a database client. The results are displayed in a table with columns: Usuario\_Id, Nombres, and Apellidos.

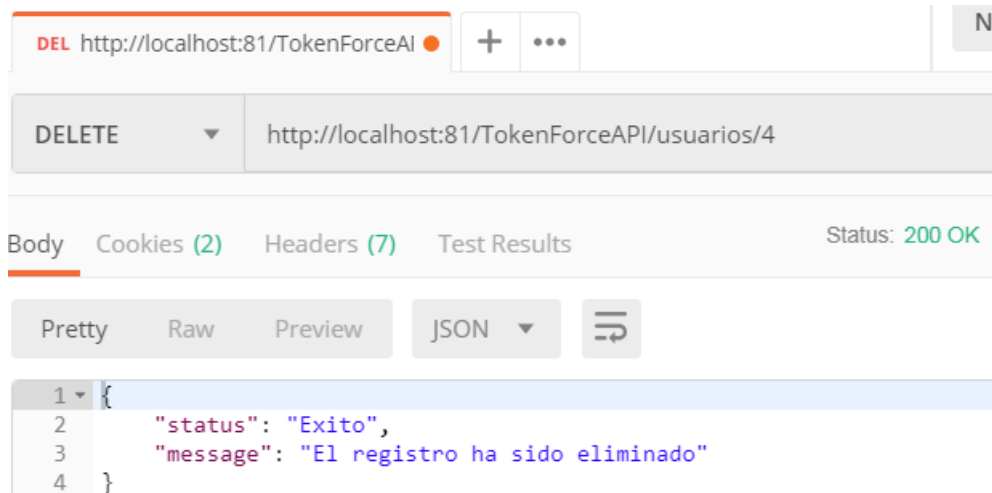
| Usuario_Id * | Nombres * | Apellidos * |
|--------------|-----------|-------------|
| 1            | Lebron    | James       |
| 2            | Stephen   | Curry       |
| 3            | James     | Harden      |
| 4            | Russell   | Westbrook   |



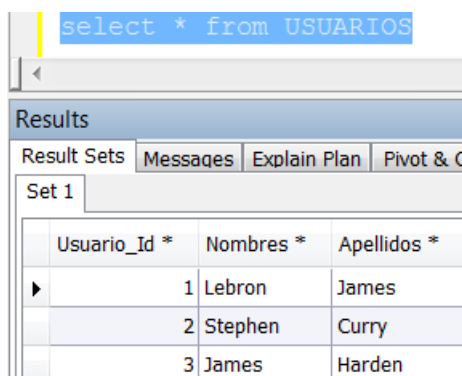
## Eliminar Un Usuario

### DELETE

<http://localhost:81/TokenForceAPI/usuarios/4>



## Verificando el Delete en BD



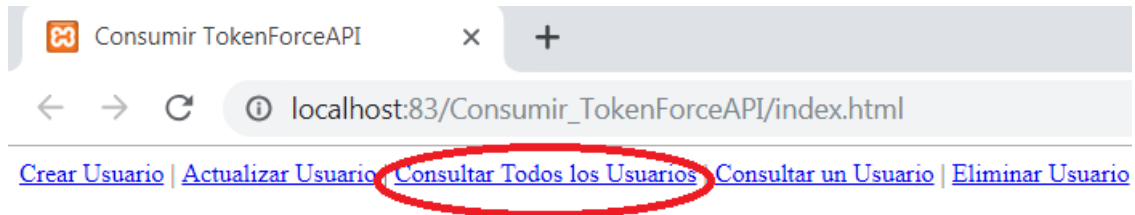
Es importante mencionar que el mensaje de respuesta está sujeto a lo que devuelva el procedimiento almacenado, para el caso de los procedimientos de inserción y borrado, siempre devolverá un 0, si la transacción fue exitosa, en caso contrario es que el registro que se intentó actualizar o borrar no existe, y devolverá un -2, en otro caso devuelve cualquier otro número asociado al error, y visualiza el mensaje: "Error, Se ha producido un error accedendo la base de datos".

## CONSUMO API MEDIANTE CLIENTE

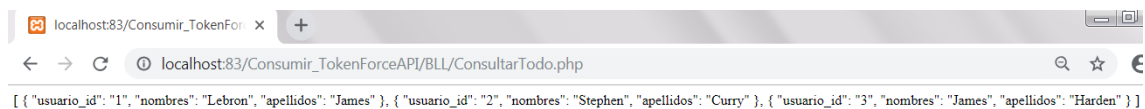
Por último, si no deseas probar con programas externos, puedes efectuarlo desde el cliente: Consumir\_TokenForceAPI, el cual también se adjunta en el repositorio.

Antes que nada, la presentación de este cliente está fea, pero es solo para poder probar el API.

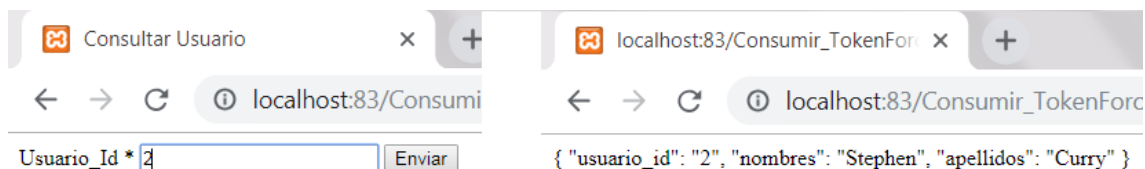
Estando en el menú, seleccionamos la opción: Consultar Todos los Usuarios



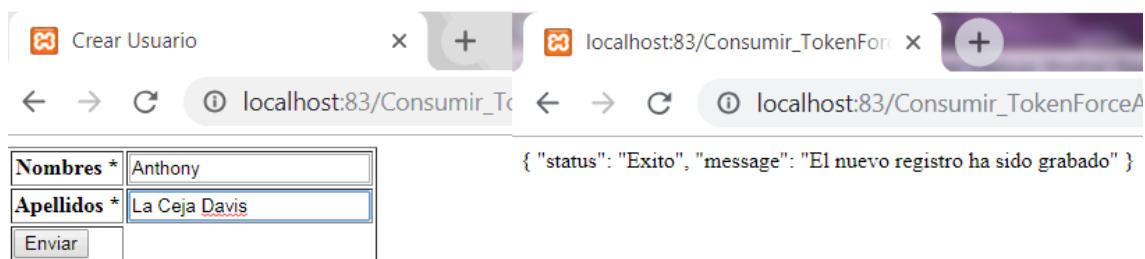
Respuesta Obtenida: Nos muestra los tres registros existentes en BD.



Seleccionando opción: Consultar un Usuario (Se debe ingresar un ID existente desde luego, para el ejemplo consultaremos el ID=2 de "Stephen Curry")



Seleccionando opción: Crear Usuario



Consultando ese usuario que se acaba de crear



Seleccionando opción: Actualizar un Usuario, vamos a modificar los apellidos

|                                       |         |
|---------------------------------------|---------|
| Usuario_id *                          | 4       |
| Nombres *                             | Anthony |
| Apellidos *                           | Davis   |
| <input type="button" value="Enviar"/> |         |

```
{ "status": "Exito", "message": "El registro ha sido actualizado" }
```

Consultando ese usuario que se acaba de actualizar

```
{ "usuario_id": "4", "nombres": "Anthony", "apellidos": "Davis" }
```

Seleccionando opción: Eliminar un Usuario

|              |   |                                       |
|--------------|---|---------------------------------------|
| Usuario_Id * | 4 | <input type="button" value="Enviar"/> |
|--------------|---|---------------------------------------|

```
{ "status": "Exito", "message": "El registro ha sido eliminado" }
```

Consultando el listado para verificar que eliminó el usuario

```
[ { "usuario_id": "1", "nombres": "Lebron", "apellidos": "James" }, { "usuario_id": "2", "nombres": "Stephen", "apellidos": "Curry" }, { "usuario_id": "3", "nombres": "James", "apellidos": "Harden" } ]
```

Puedes descargar de Github en el siguiente enlace:

[https://github.com/JCorreal/API\\_REST\\_PHP](https://github.com/JCorreal/API_REST_PHP)

## PORQUÉ REST

El lanzamiento del REST, como protocolo de intercambio y manipulación de datos en los servicios de Internet, cambió por completo el desarrollo de software, hoy casi toda empresa o aplicación, dispone de una API REST para intercambio de información, entre sistemas informáticos.

Si bien SOAP con su tradicional XML fue un estándar ampliamente difundido, pero lo pesado del XML, dado que tiene que ser parseado a un árbol DOM, y resolver espacios de nombre antes de poder empezar a procesar el documento, ha hecho que haya ido perdiendo fuerza ante REST.

REST por su parte cuenta con una serie de bondades como un manejo de sistema de capas: arquitectura jerárquica entre los componentes. Cada una de estas capas lleva a cabo una funcionalidad dentro del sistema REST.

Protocolo cliente/servidor sin estado: cada petición HTTP contiene toda la información necesaria para ejecutarla, lo que permite que ni cliente ni servidor necesiten recordar ningún estado previo para satisfacerla.

Interfaz uniforme: para la transferencia de datos en un sistema REST, este aplica acciones concretas (POST, GET, PUT y DELETE) sobre los recursos, siempre y cuando estén identificados con una URI.

Separación entre el cliente y el servidor: el protocolo REST separa totalmente la interfaz de usuario del servidor y el almacenamiento de datos. Eso tiene algunas ventajas cuando se hacen desarrollos. Por ejemplo, mejora la portabilidad de la interfaz a otro tipo de plataformas, aumenta la escalabilidad de los proyectos y permite que los distintos componentes de los desarrollos se puedan evolucionar de forma independiente.

Visibilidad, fiabilidad y escalabilidad. La separación entre cliente y servidor tiene una ventaja evidente y es que cualquier equipo de desarrollo puede escalar el producto sin excesivos problemas. Se puede migrar a otros servidores o realizar todo tipo de cambios en la base de datos, siempre y cuando los datos de cada una de las peticiones se envíen de forma correcta. Esta separación facilita tener en servidores distintos el front y el back y eso convierte a las aplicaciones en productos más flexibles a la hora de trabajar.

La API REST siempre es independiente del tipo de plataformas o lenguajes: la API REST siempre se adapta al tipo de sintaxis o plataformas con las que se estén trabajando, lo que ofrece una gran libertad a la hora de cambiar o probar nuevos entornos dentro del desarrollo. Con una API REST se pueden tener servidores PHP, Java, Python o Node.js. Lo único que es indispensable es que las respuestas a las peticiones se hagan siempre en el lenguaje de intercambio de información usado, normalmente XML o JSON.

Bien este ejemplo estuvo muy sencillo, la idea es que cada quien lo adapte y expanda de acuerdo a las necesidades. Cualquier duda no olviden escribir a: [correal\\_95@hotmail.com](mailto:correal_95@hotmail.com)