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 bastante lentooos.

```
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_DUsuario`(
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_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_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 carpetas en el proyecto, 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 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 enlace entre los modelos y las vistas, acá en este paquete 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 este paquete quedan tres clases a saber:

- Conexion: Clase que conecta con el proveedor de Base de Datos-MySql.

- IAccesoDatos: Interfaz que solo expone los métodos que implementa el acceso a datos.

- 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.

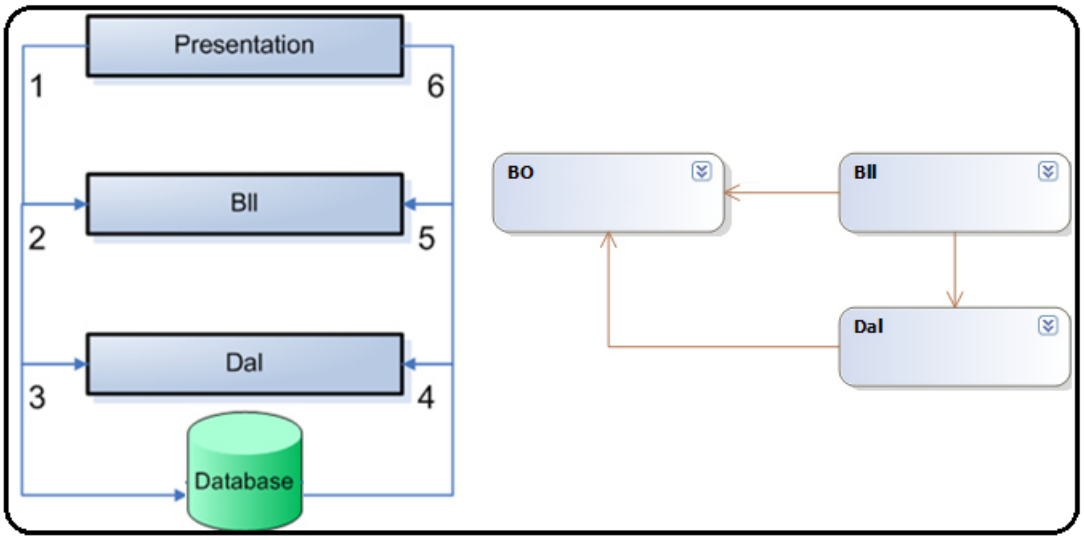
Nunca se llega directamente al DAL, siempre todo deberá pasar primero por el Controlador que será nuestro manager, encargado de orquestar, controlar y definir que método del DAL invocar, para que sea ésta última quien devuelva los resultados esperados, y dar así respuesta a las diferentes peticiones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

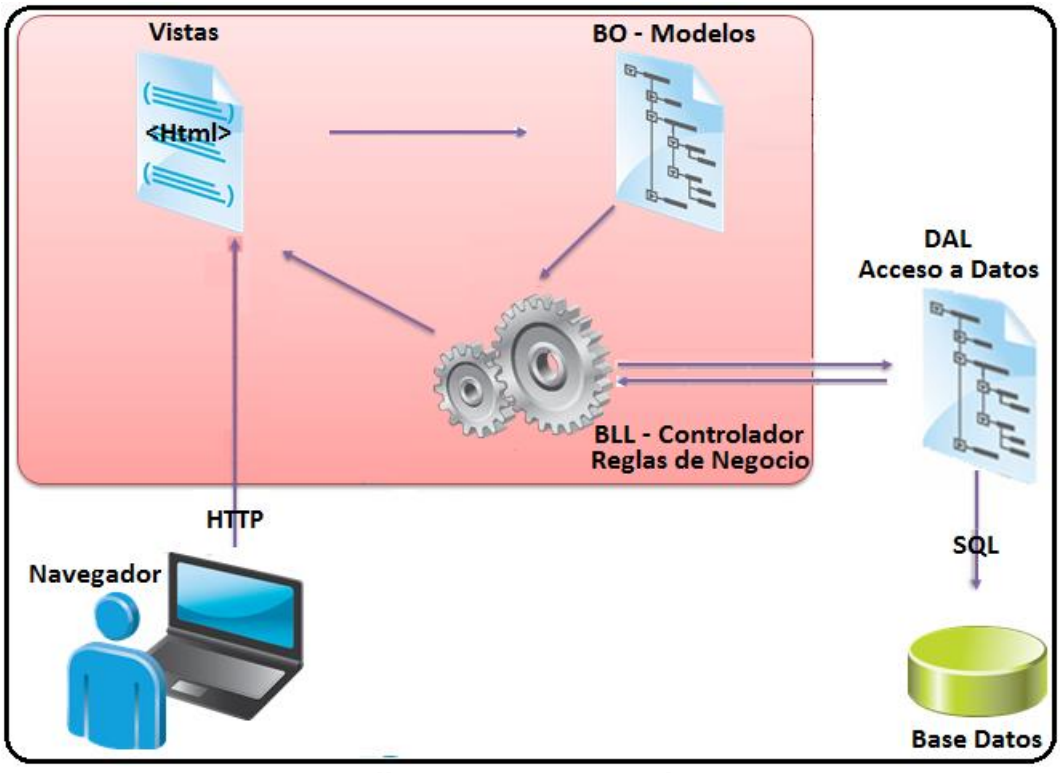
El DAL tiene por cada objeto del BO, métodos para obtener listados completos, o un solo ítem o registro, y por supuesto los demás métodos básicos del CRUD, para creación, actualización y eliminación de registros.

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.

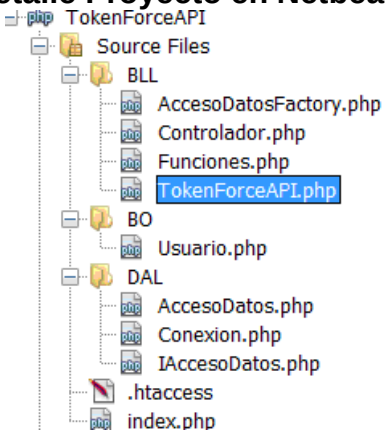
Donde predomina una “organización jerárquica tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior.” Garlan & Shaw.

ESQUEMA DE FUNCIONAMIENTO





Detalle Proyecto en Netbeans

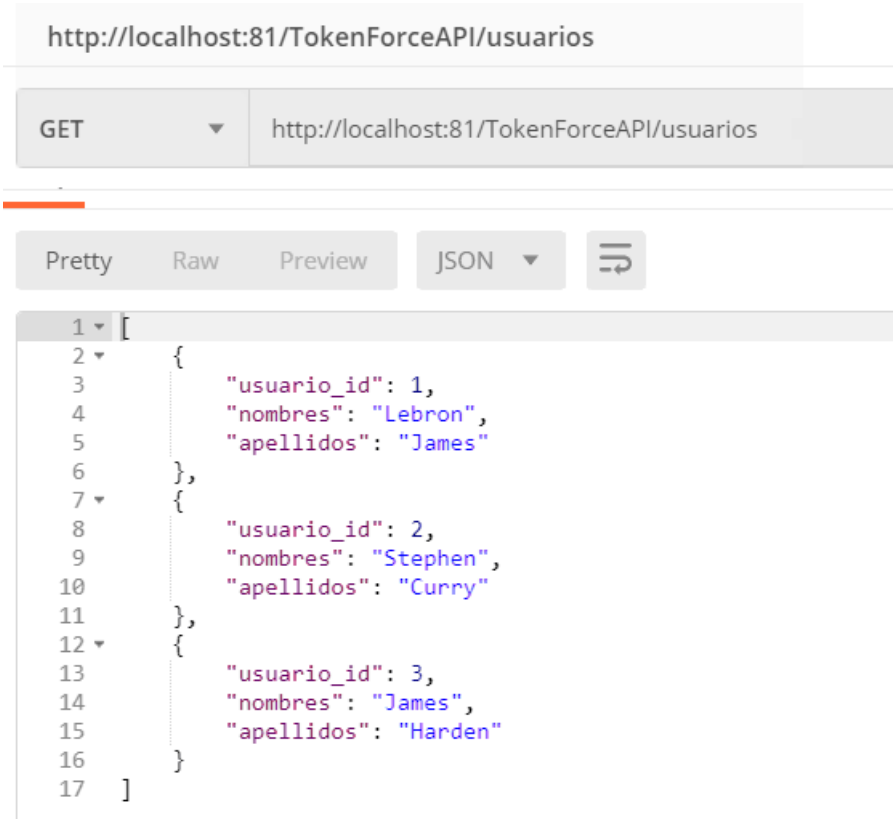


PRUEBAS API CON POSTMAN

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

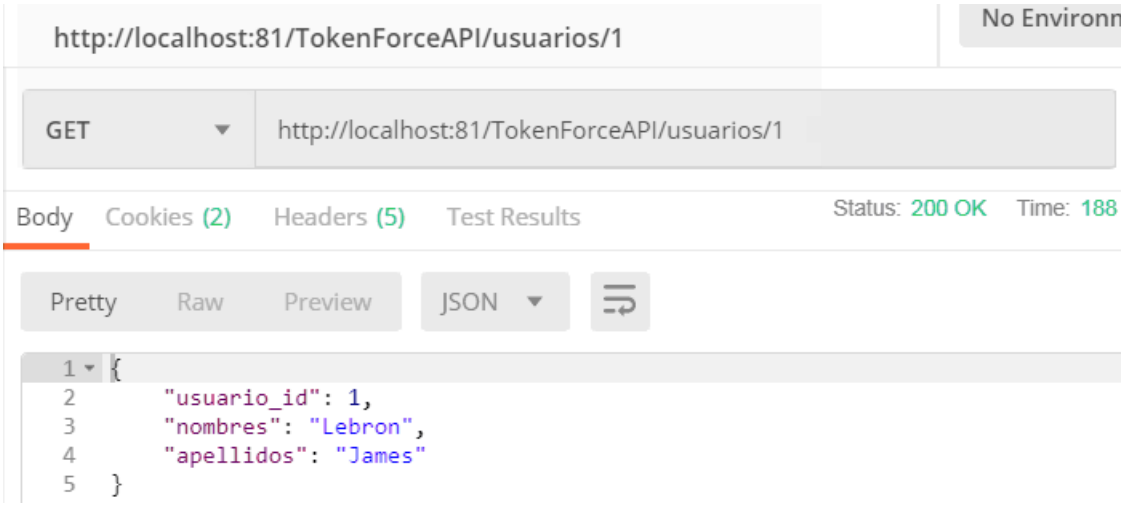
Obtener Todos los Usuarios

GET
http://localhost:81/TokenForceAPI/usuarios



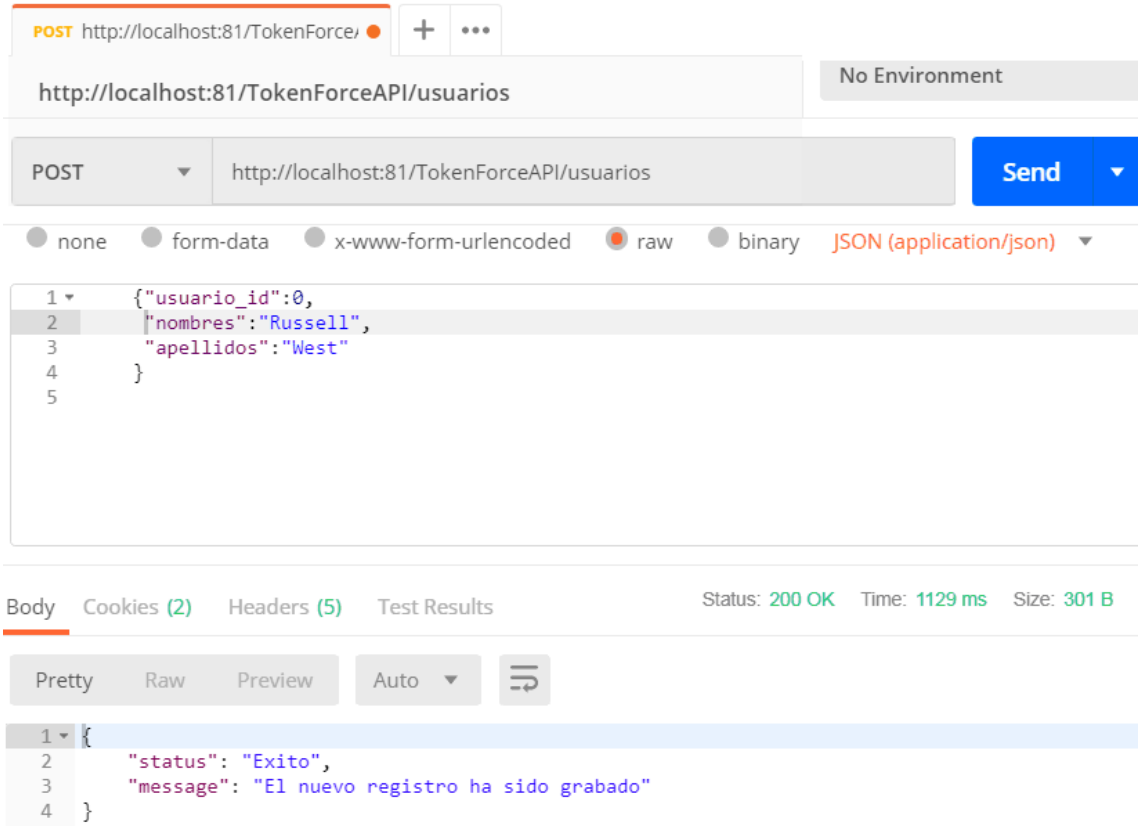
Obtener Un Usuario

GET
http://localhost:81/TokenForceAPI/usuarios/1

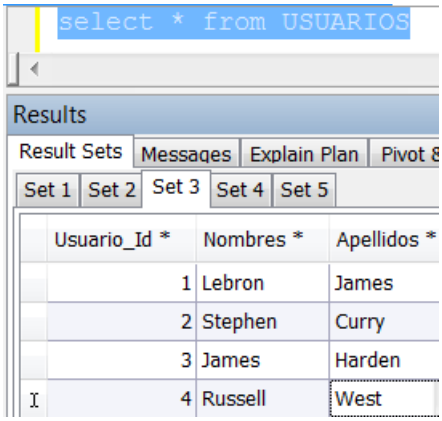


Agregar Un Usuario

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



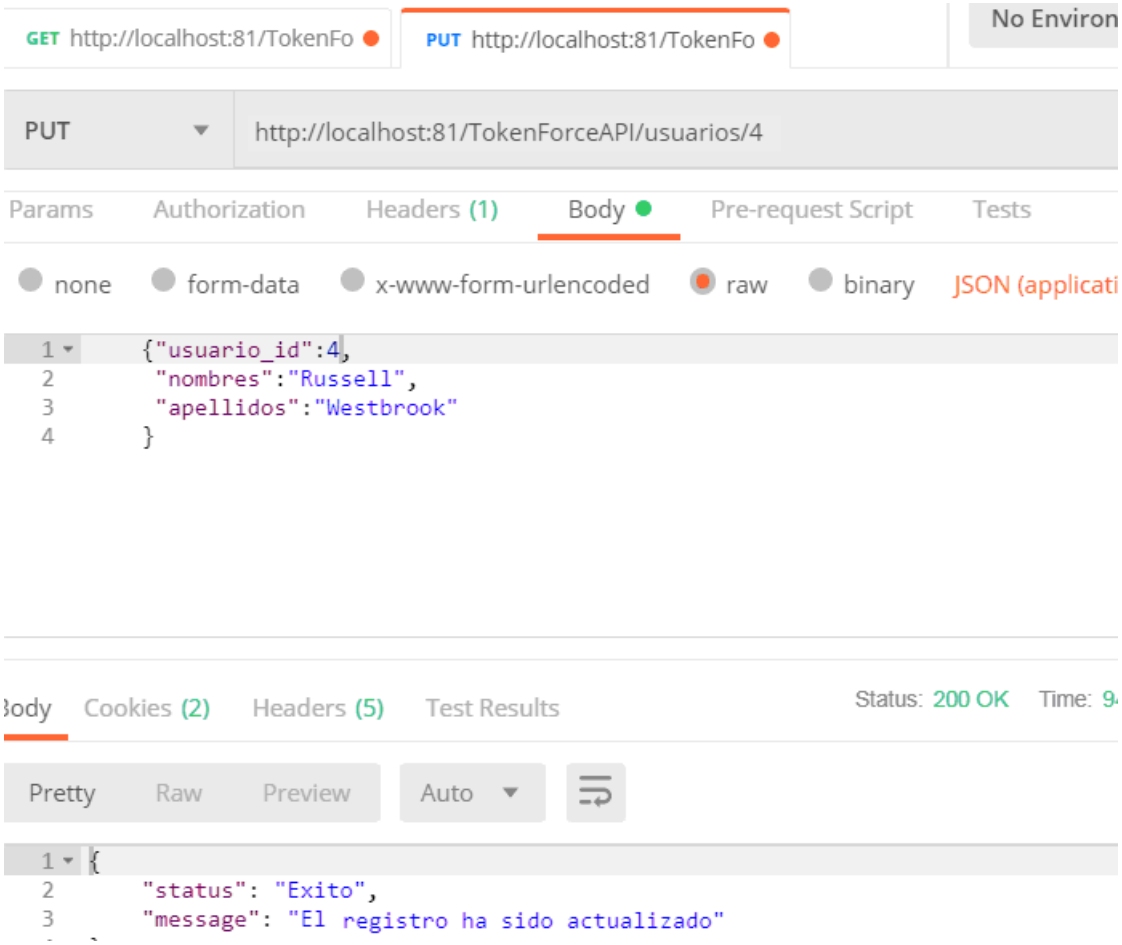
Verificando el Insert en BD



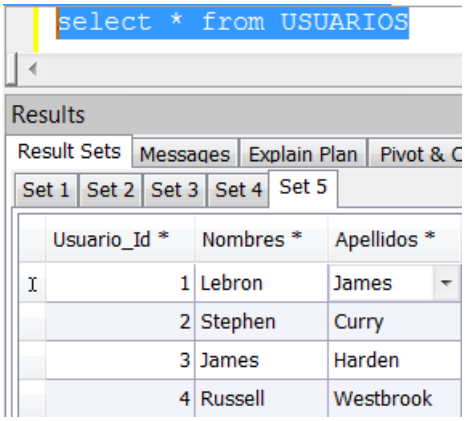
Actualizar Un Usuario

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

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

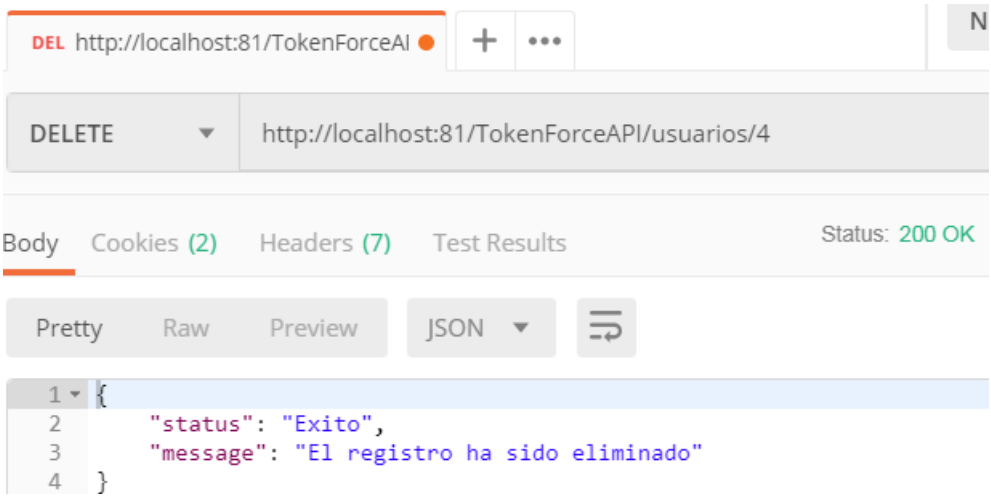


Verificando el Update en BD

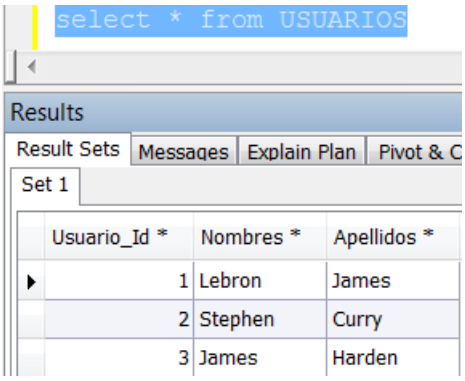


Eliminar Un Usuario

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 accesando la base de datos".

Puedes descargar de Github en el siguiente enlace:

https://github.com/JCorreal/API_REST_PHP