Bien, en esta ocasión traigo un API Rest construido en Java con MySQL, la cual consiste en CRUD para una tabla de registro de Usuarios.

Vamos inicialmente a crear nuestra Base de Datos para probar el API REST, ésta 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, se puede renombrar con el nombre que se desee, pero deberás luego modificar la cadena de conexión, en la clase dispuesta para ello, y 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…

Una vez hayas creado la BD, procederemos a crear el API en Java, yo trabajé con el JDK: 1.8, empleando el Ide: NetBeans 8.2. Además debes tener en cuenta las librerías:

- jersey-media-json-jackson-2.0-m07-1
- com.fasterxml.jackson.annotations
- com.fasterxml.jackson.databind

Y por supuesto el MySQL JDBC Driver, sin estas tres librerías y el driver Mysql no te funcionará, así que asegúrate descargarlos, de cualquier forma yo he subido a github (Mirar enlace al final del documento), la carpeta build que genera NetBeans y dentro de esta, queda la carpeta libs con todas las librerías.

Bien ahora te detallo como quedó configurado el proyecto, para que se entienda la arquitectura y distribución de carpetas en el proyecto, yo utilicé MVC donde quedaron 4 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 puente entre modelos y vistas, acá en este paquete quedan tres clases a saber:

- 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.

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.

Api - Configuración del Proyecto: Es la capa que tiene la configuración inherente dentro del aplicativo, acá en este paquete quedan tres clases a saber:

- ApplicationConfig: Extiende de Application y lo que se busca es que el contenedor web, localice todas nuestras clases y las publique como servicios web, comportándose para este caso como un servicio Rest.
- MyJacksonJsonProvider: Es la librería jackson empleada como base para el servicio, se puede utilizar cualquier otra del agrado.
- RestfulService: Es el servicio como tal, y es quien recibe las peticiones, el cual solo tiene por ahora 5 métodos.

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 interfaz llamada IAccesoDatos, que será nuestra puerta de entrada, allí no hay implementación de ningún método, solo se exponen, quien use dicha interfaz 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.

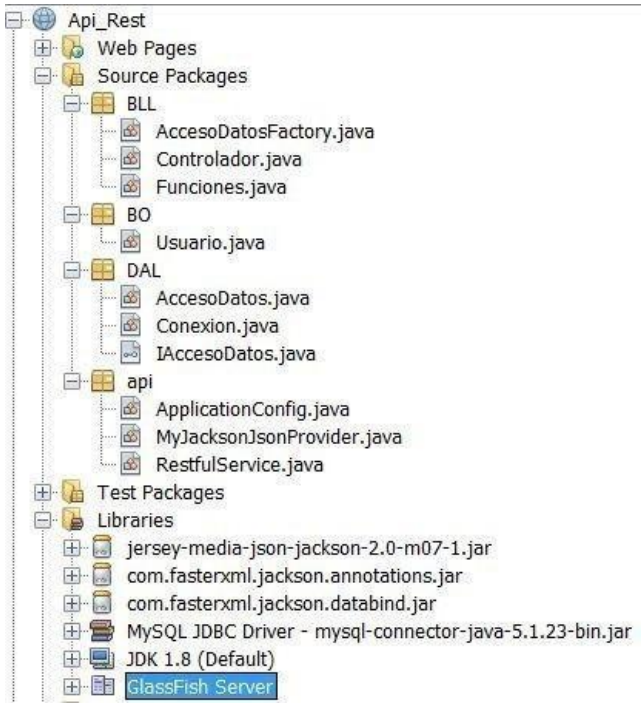
Nunca se llega directamente al DAL, siempre todo deberá pasar primero por el Controlador que será nuestro manager, encargado de orquestar, controlar y definir que método del DAL invocar, para que sea ésta última quien devuelva los resultados esperados, y dar así respuesta a las diferentes peticiones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

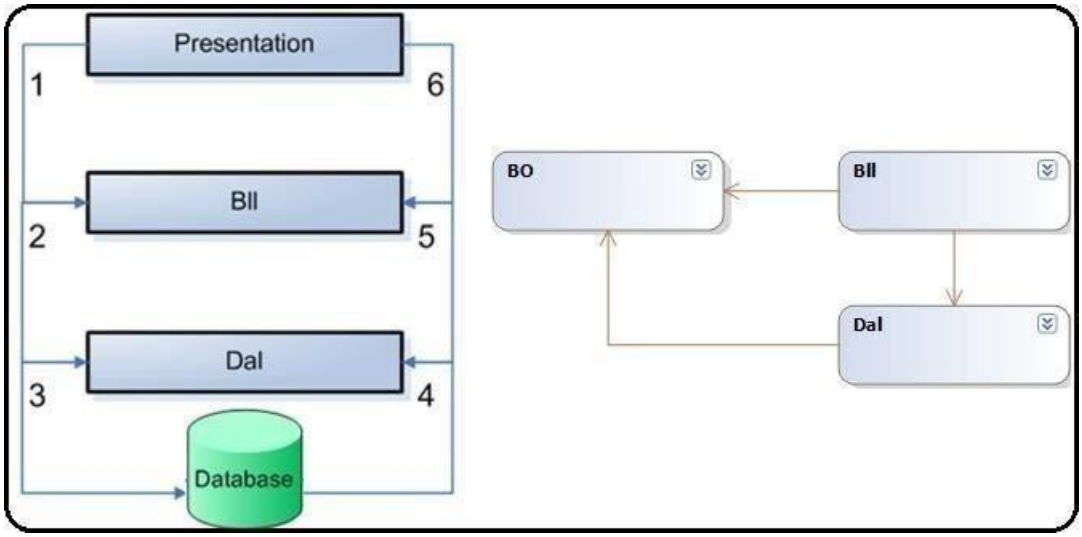
La finalidad de separar los sistemas en capas, no es otra que facilitar su posterior mantenimiento, donde cada capa expone servicios que otras aplicaciones o capas pueden consumir, lo que se traduce en una simplicidad conceptual, de alta cohesión y bajo acoplamiento, facilitando así su reutilización, 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.

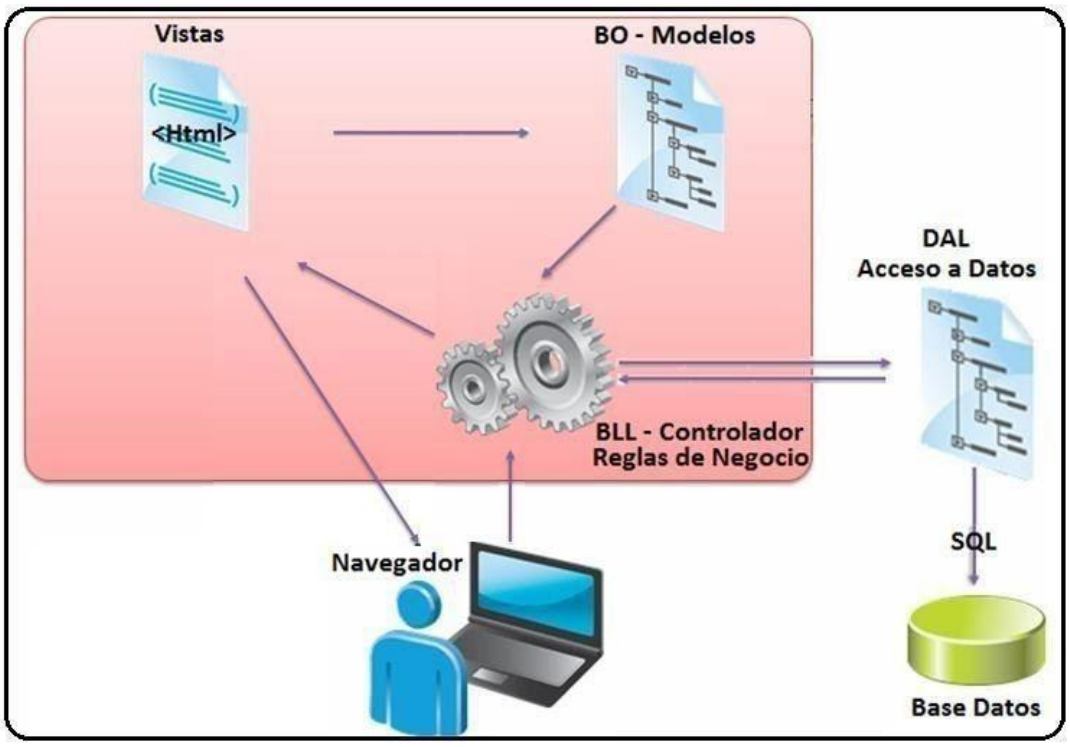
Detalle Proyecto en Netbeans



ESQUEMA DE FUNCIONAMIENTO



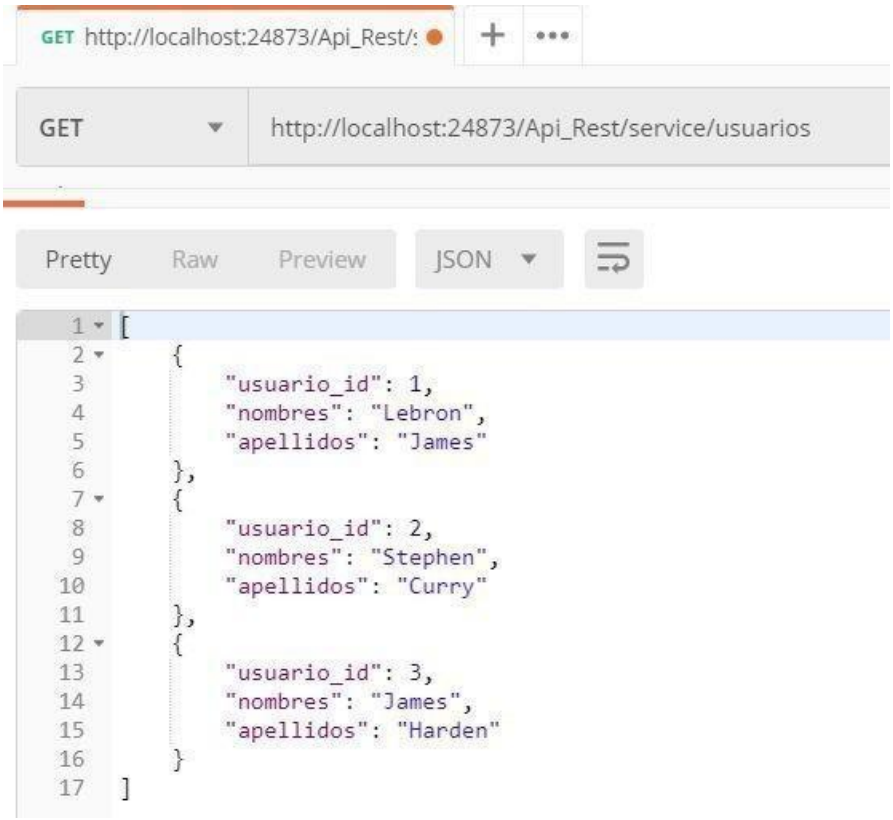
Funcionamiento MVC



PRUEBAS API CON POSTMAN

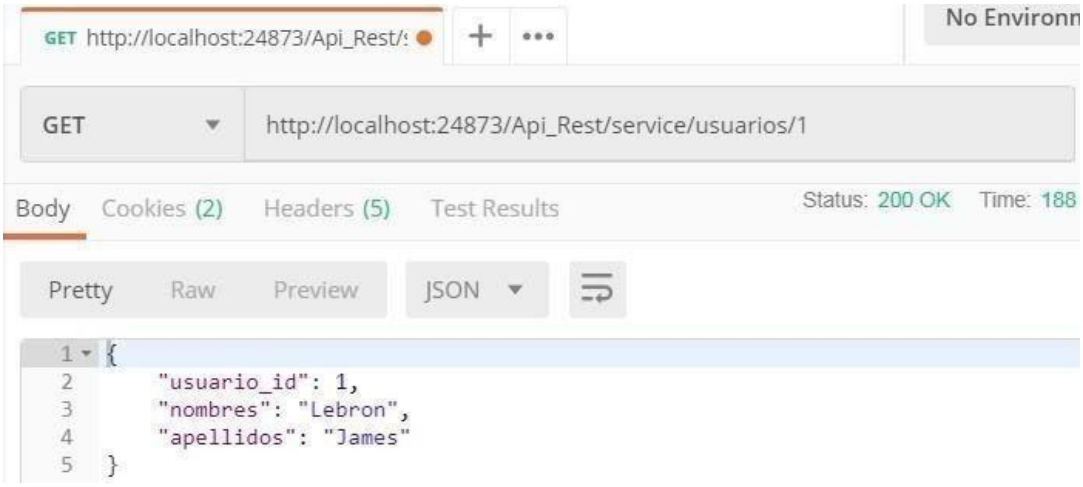
Obtener Todos los Usuarios GET

http://localhost:8080/Api_Rest/service/usuarios



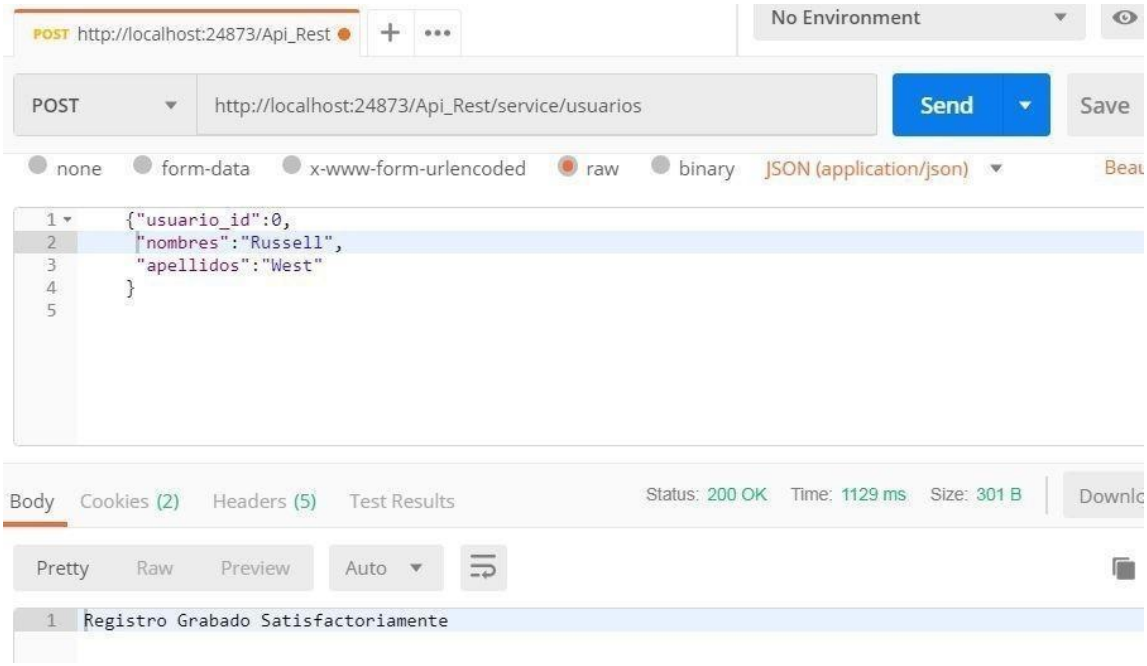
Obtener Un Usuario GET

http://localhost:8080/Api_Rest/service/usuarios/1

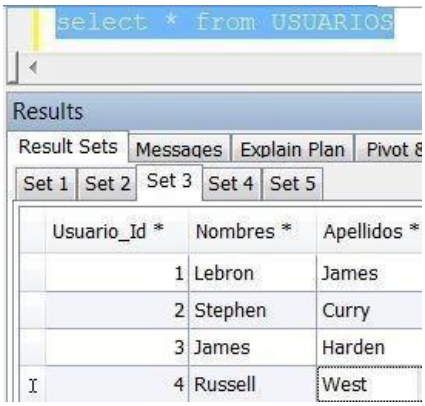


Agregar Un Usuario

POST
http://localhost:8080/Api_Rest/service/usuarios



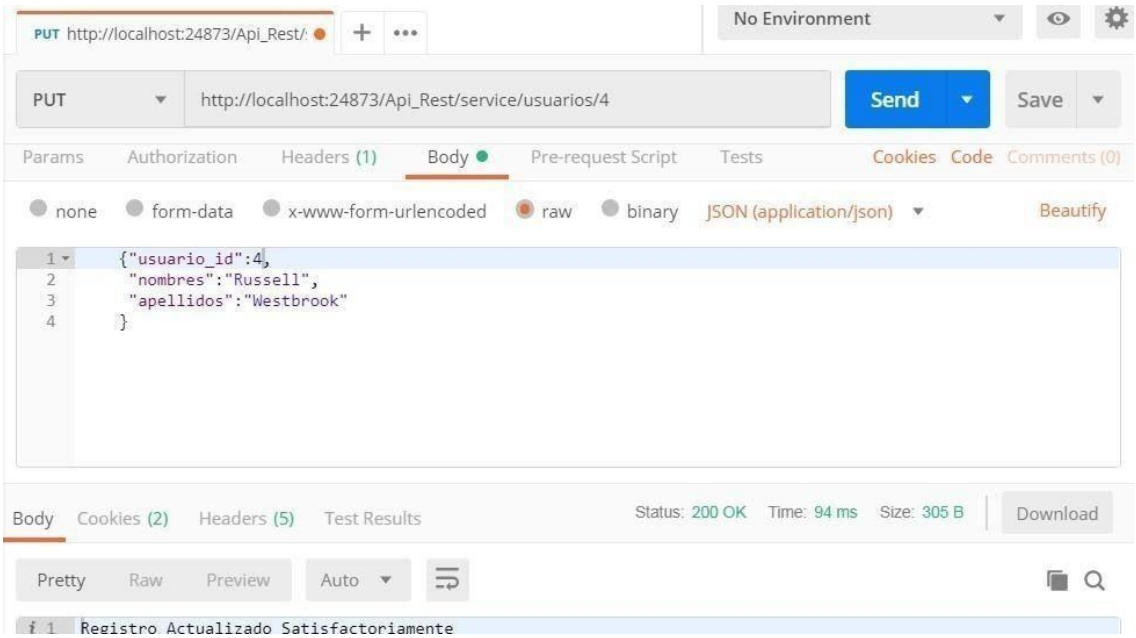
Verificando el Insert en BD



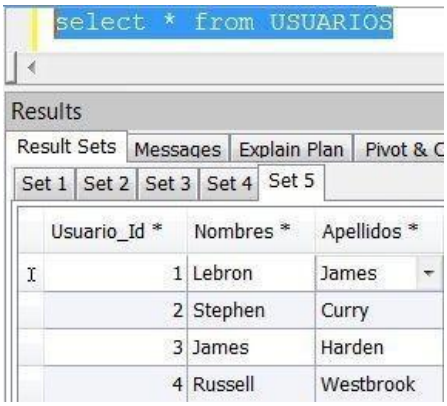
Actualizar Un Usuario

PUT
http://localhost:8080/Api_Rest/service/usuarios/4

Cambiaremos el Apellido de este registro "West" por "Westbrook", que es el correcto apellido de este famoso jugador de baloncesto

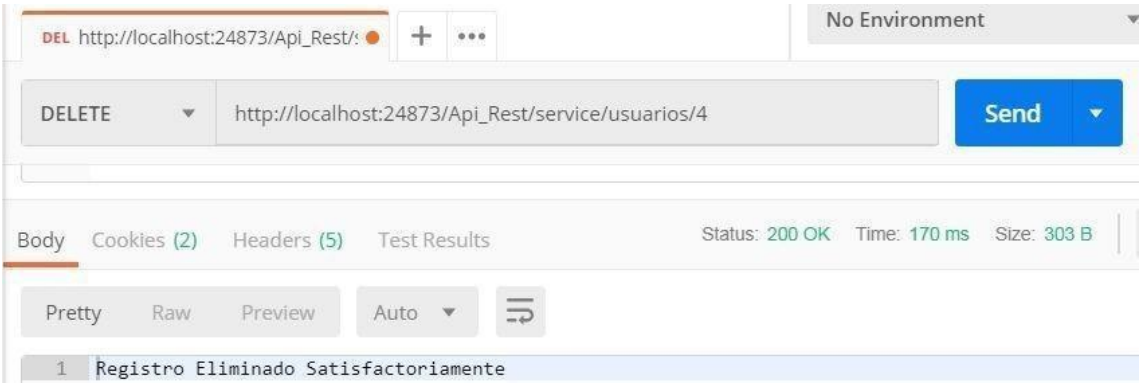


Verificando el Update en BD

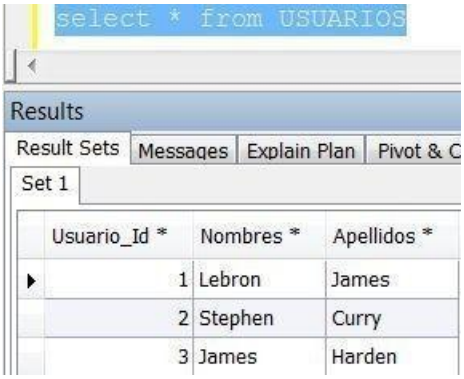


Eliminar Un Usuario

DELETE
http://localhost:8080/Api_Rest/service/usuarios/4



Verificando el Delete en BD

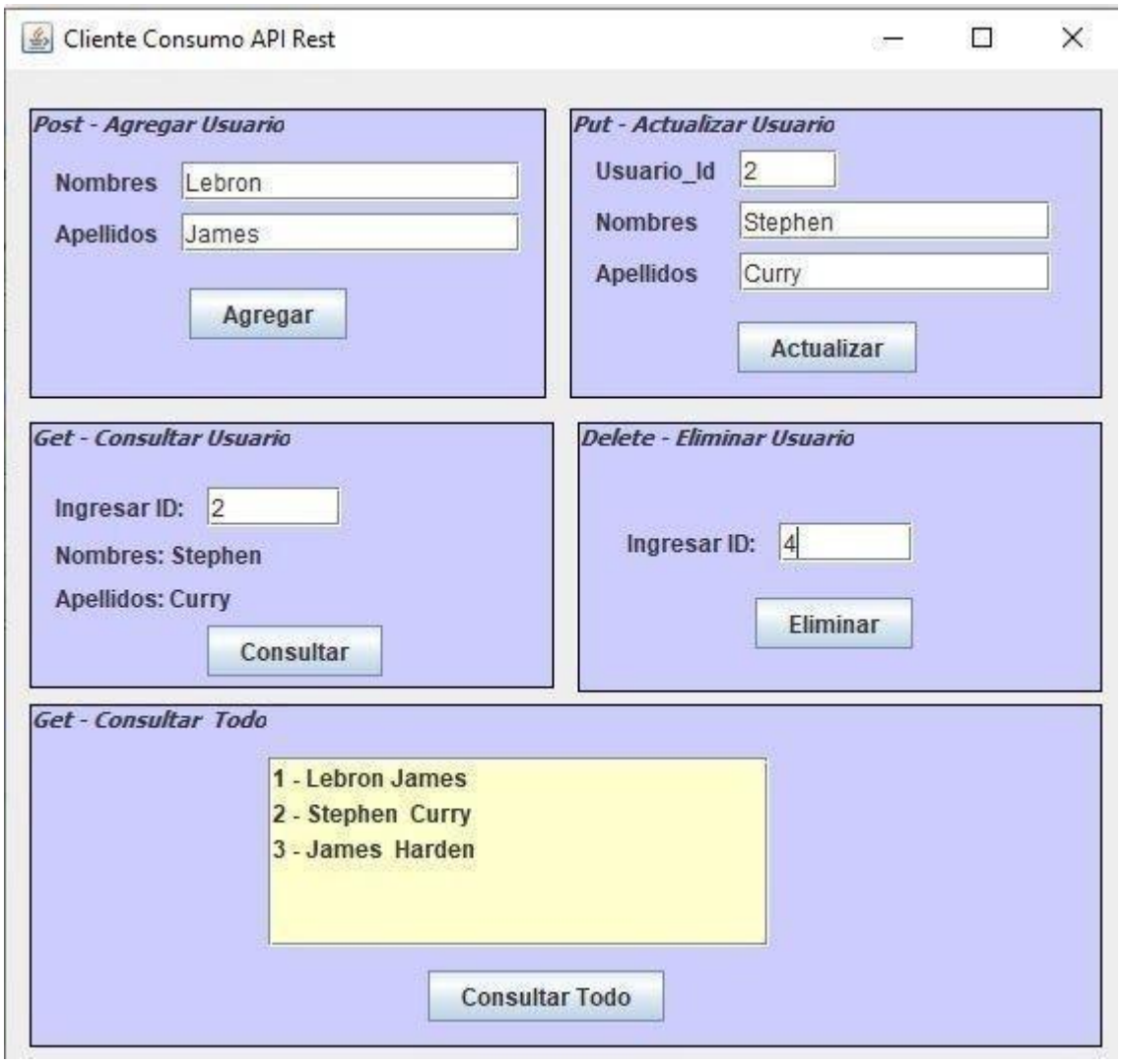


Es importante mencionar que el mensaje de respuesta está sujeto a lo que devuelva el procedimiento almacenado, para el caso de los procedimientos de inserción y borrado, siempre devolverá un 0, si la transacción fue exitosa, en caso contrario es que el registro que se intentó actualizar o borrar no existe, 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".

Ahora bien, si no deseas probar con software externo como Postman u otro cualquiera, también lo puedes hacer con un cliente propio que construí, del cual adjunto fuente.

En realidad son dos clientes en versiones desktop y web.

Cliente_Api_Rest_Desktop



Cliente_Api_Rest_Web

No creo que amerite mucha explicación, pero básicamente cada opción exige que se ingresen los datos (Usuario_ID o Nombres y Apellidos según sea el caso) y listo.

Estas interfaces de los clientes no están para nada bonitas, es solo para probar y consumir el API Rest. Cada quien que las mejore y optimice, pues finalmente el objetivo de un cliente que consuma el servicio, busca en esencia es eso, consumir y explotar todas las funcionalidades que exponga el servicio, y presentar los datos a los usuarios de forma más entendible para ellos, no es mostrarlos como los devuelve el servicio en JSON, sino recorrer y extraer de ese JSON la información, y visualizarla en cajas de texto o tablas, según sea la cantidad de datos que venga.

Lo que sí es **importante** tener en cuenta es que deben descargar la librería GSON de Google, pues los clientes fueron construidos con dicha librería, que bien puede utilizarse otra cualquiera, yo elegí esta porque es de amplio uso y muy fácil de implementar.

Finalmente, esta no es ni la mejor, ni la única forma de hacer las cosas, los microservicios en Java, se pueden construir apoyándose en muchos frameworks, y la persistencia igual se puede realizar de muchas maneras, yo elegí los procedimientos almacenados para gestionar todo en el servidor, pero eres libre de hacerlo como te convenga según las necesidades.

Puedes descargar de Github en el siguiente enlace:

https://github.com/JCorreal/Api_Rest_Java_Jersey

Y cualquier duda o sugerencia pueden escribirme por acá en Github