

API REST PYTHON Y MARIADB

Bien, debo confesar que aún no he trabajado mucho con python, quizás por falta de tiempo, quizás por pereza, quizás por muchas otras razones, pero una vez lo he tratado, me parece bastante sencillo, me gusta su organizada exigencia en la tabulación, porque ello ayuda enormemente en la organización y legibilidad del código, facilitando el mantenimiento de las aplicaciones, aparte de que soporta parcialmente la POO, contribuye a reutilizar código y facilita ampliamente el aprovechamiento de la herencia.

Sin embargo, hay ciertas cosas que se suelen hacer en el mundo java de una forma muy particular, que por momentos al hacerlo en python crean algo de confusión, como los atributos de clase para los objetos de negocio, se extrañan los get y set, pero independiente de ello el lenguaje es fácil y divertido de trabajar.

Bien, luego de mis impresiones iniciales al universo Python, vamos al grano.

API Rest construido en Python con MariaDB, el cual consiste en CRUD para una tabla de registro de Usuarios, que realizará todas las operaciones del CRUD como tal.

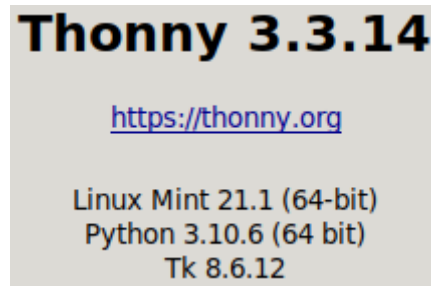
CONTENIDO DOCUMENTO

1. Creación Base Datos MariaDB
2. Configuración Proyecto
3. Funcionamiento
4. Esquema de Funcionamiento
5. Pruebas Consumo Api con Programas Externos
6. Por qué API REST

1. Creación Base Datos MariaDB

Vamos inicialmente a crear nuestra Base de Datos para probar el API REST, de la cual se adjuntan sus respectivos scripts, esta BD es 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 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.

La versión de MariaDB que yo instalé es: 10.4.28, una vez que hayas creado la BD, procederemos a crear el API en Python, yo trabajé con Python 3.10.6, utilizando el Ide Thonny 3.3.14 sobre sistema operativo Linux Mint 21.1 de 64 bit



2. Configuración Proyecto

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 quedan 3 capas a saber:

BO - Capa Objetos de Negocio: Acá queda alojado nuestro único objeto de negocio llamado: Usuario, que es la clase que representa esta estructura en BD.

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.

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.
- **IAccesoDatos:** Interfaz que contiene una colección de métodos abstractos y sólo los expone.
- **AccesoDatos:** Implementa la interfaz con todos sus métodos.

Main: Es el servicio como tal, y quien recibe las peticiones del usuario, tiene 5 métodos.

3. 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 consulte por un registro específico, cuando ese registro es encontrado, este es retornado de la base de datos al DAL, quien empaqueta los datos en un objeto personalizado y lo retorna al BLL, para finalmente presentarlo 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.

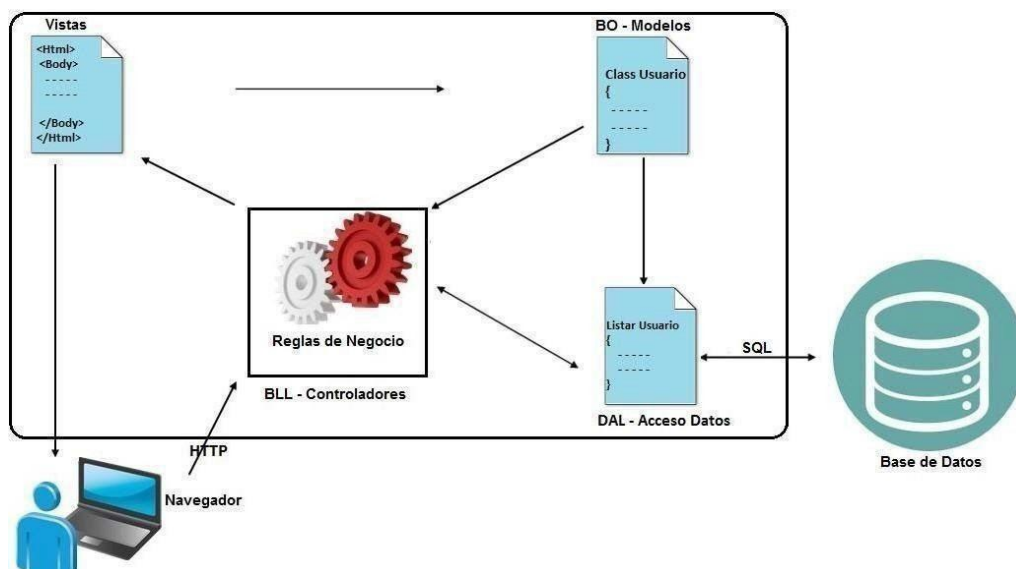
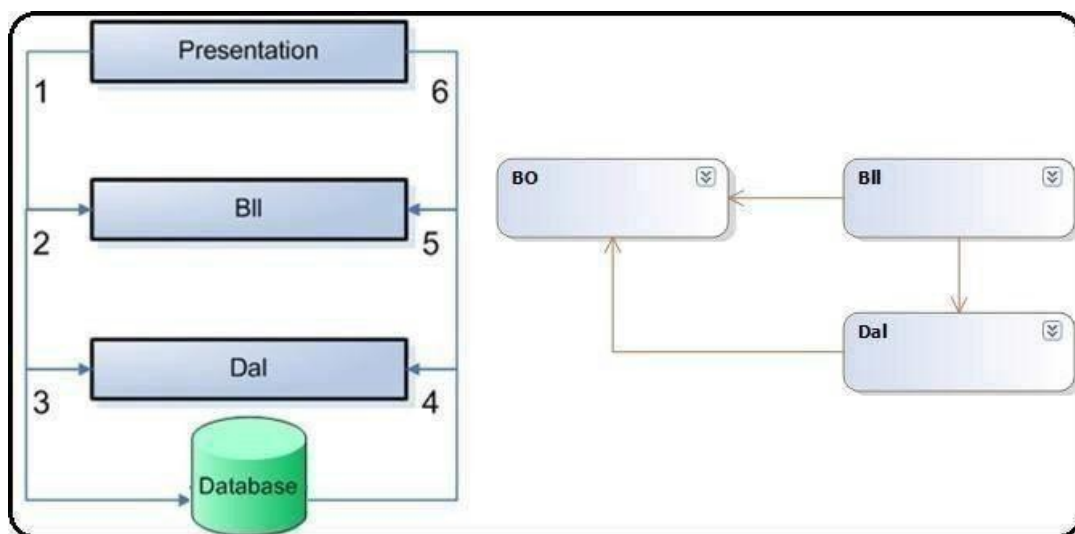
Nunca se llega directamente al DAL, siempre todo deberá pasar primero por el Controlador que será nuestro manager, encargado de orquestar, controlar y definir que método del DAL invocar, para que sea ésta última quien devuelva los resultados esperados, y dar así respuesta a las diferentes peticiones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

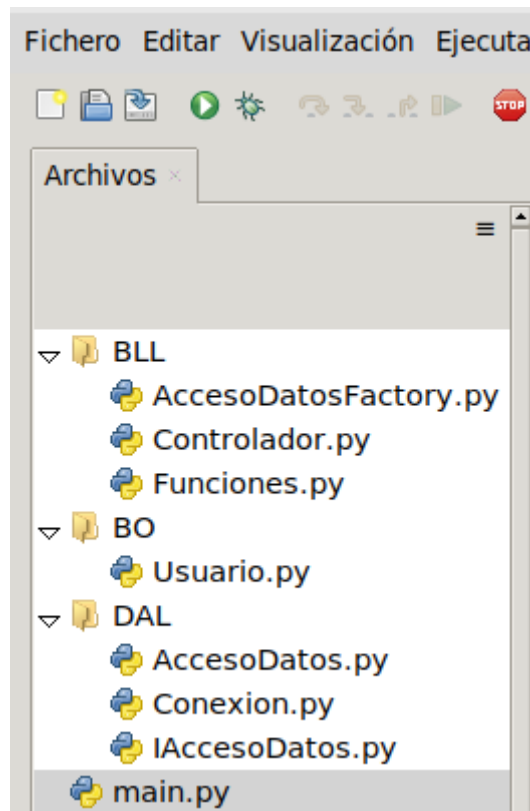
La finalidad de separar los sistemas en capas, no es otra que facilitar su posterior mantenimiento, donde cada capa expone servicios que otras aplicaciones o capas pueden consumir, lo que se traduce en una simplicidad conceptual, de alta cohesión y bajo acoplamiento, facilitando así su reutilización,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

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.

4. Esquema de Funcionamiento



Estructura Proyecto en Thonny



5. Pruebas Consumo API con Programas Externos

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

Mientras no se requiera enviar en el body información, basta con probar en cualquier navegador y debería responder ante el Get y Delete, ya los demás si necesariamente hay que buscar apoyo en algún programa que lo permita, para probar este API podemos bien recurrir a varios programas diseñados para tal función, inicialmente lo haremos con postman e insomnia, que son bastante populares y fáciles de usar.

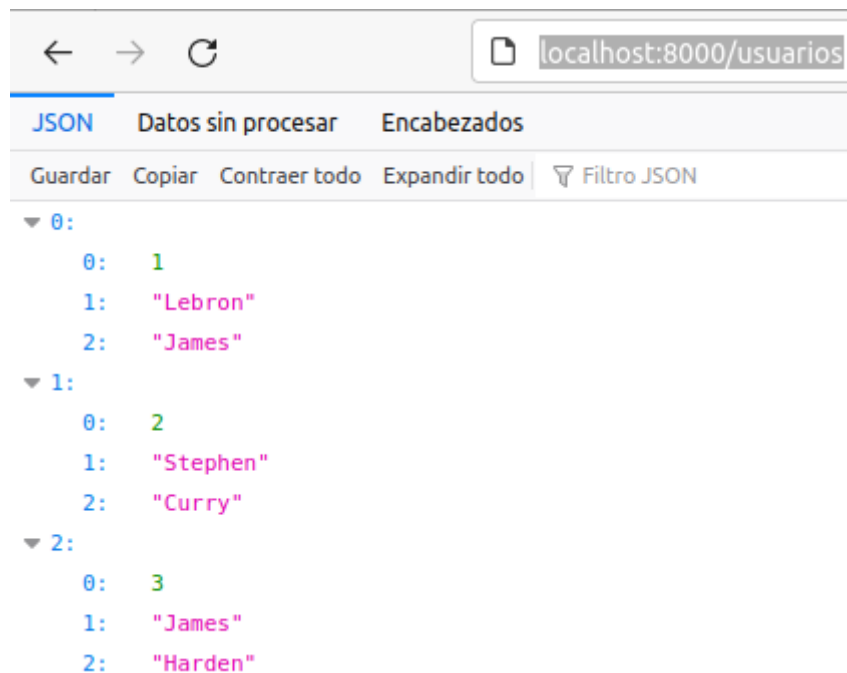
Para probar el API puedes configurar las siguientes peticiones dependiendo de la ruta de tu proyecto:

1. Obtener Usuarios : GET : <http://localhost/usuarios>
2. Obtener un Usuario : GET : <http://localhost/usuario/1>
3. Nuevo Usuario : POST: <http://localhost/usuario>
4. Actualizar Usuario : PUT : <http://localhost/usuario>
5. Eliminar Usuario : DELETE : <http://localhost/usuario/1>

Para los casos de obtener un usuario y eliminar un usuario, es válido colocar el id implicado al final y/o colocar al final `?id=X`, donde X es el id del usuario

Obtener Todos Los Usuarios GET

<http://localhost:8000/usuarios>

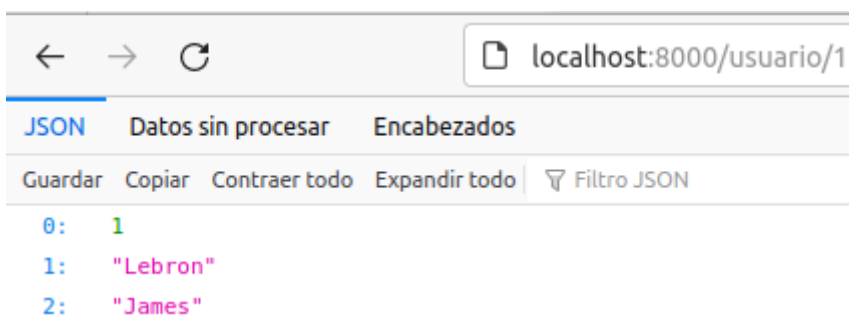


The screenshot shows a web browser with the address bar displaying `localhost:8000/usuarios`. The browser's developer tools are open, showing the JSON response for the GET request. The response is an array of three objects, each representing a user. The first object has an id of 1, name 'Lebron', and surname 'James'. The second object has an id of 2, name 'Stephen', and surname 'Curry'. The third object has an id of 3, name 'James', and surname 'Harden'.

```
{
  "0": {
    "id": 1,
    "name": "Lebron",
    "surname": "James"
  },
  "1": {
    "id": 2,
    "name": "Stephen",
    "surname": "Curry"
  },
  "2": {
    "id": 3,
    "name": "James",
    "surname": "Harden"
  }
}
```

Obtener Un Usuario GET

<http://localhost:8000/usuarios/1>



The screenshot shows a web browser with the address bar displaying `localhost:8000/usuario/1`. The browser's developer tools are open, showing the JSON response for the GET request. The response is a single object representing the user with id 1, name 'Lebron', and surname 'James'.

```
{
  "id": 1,
  "name": "Lebron",
  "surname": "James"
}
```

Agregar Un Usuario POST

<http://localhost/usuario>

POST

http://localhost:8000/usuario

Params

Authorization

Headers (9)

Body

Pre-request Script

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

JSON

1

2

3

4

5

.....

....."usuario_id":0,

....."nombres":"Russell",

....."apellidos":"West"

.....

body

Cookies

Headers (8)

Test Results

Status: 200 OK

Pretty

Raw

Preview

Visualize

JSON

1

2

3

4

5

{

"Codigo": 200,

"Estado": "Exito",

"Mensaje": "El nuevo registro ha sido grabado"

}

Verificando el Insert en BD

123	Usuario_Id	ABC	Nombres	ABC	Apellidos
	1		Lebron		James
	2		Stephen		Curry
	3		James		Harden
	4		Russell		West

Actualizar Un Usuario PUT

<http://localhost/usuario>

The screenshot shows a REST client interface. At the top, the method is set to **PUT** and the URL is `http://localhost:8000/usuario`. Below the URL bar, there are tabs for **Params**, **Authorization**, **Headers (9)**, **Body** (selected), **Pre-request Script**, **Tests**, and **Settings**. Under the **Body** tab, there are radio buttons for **none**, **form-data**, **x-www-form-urlencoded**, **raw**, **binary**, and **JSON** (selected). The body content is a JSON object:

```
1 {
2   "usuario_id": 4,
3   "nombres": "Russell",
4   "apellidos": "Westbrook"
5 }
```

Below the body, there are tabs for **Body** (selected), **Cookies**, **Headers (8)**, and **Test Results**. On the right, the status is **Status: 200 OK**. Under the **Body** tab, there are buttons for **Pretty** (selected), **Raw**, **Preview**, **Visualize**, **JSON**, and a refresh icon. The response body is a JSON object:

```
1 {
2   "Codigo": 200,
3   "Estado": "Exito",
4   "Mensaje": "El registro ha sido actualizado"
5 }
```

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

123 Usuario_Id	ABC Nombres	ABC Apellidos
1	Lebron	James
2	Stephen	Curry
3	James	Harden
4	Russell	Westbrook

Eliminar Un Usuario DELETE

<http://localhost/usuario/4>

DELETE

Params Authorization Headers (7) **Body** Pre-request Script Tests Settings

☒ none ☐ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary

This request does not have a body

body Cookies Headers (8) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON

```
1 {
2   "Codigo": 200,
3   "Estado": "Exito",
4   "Mensaje": "El registro ha sido eliminado"
5 }
```

Verificando el Delete en BD

123 Usuario_Id	ABC Nombres	ABC Apellidos
1	Lebron	James
2	Stephen	Curry
3	James	Harden

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 accedando la base de datos".

6. Por qué API 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 con arquitectura jerárquica entre los componentes, donde cada una de estas capas lleva a cabo una funcionalidad dentro del sistema.

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.

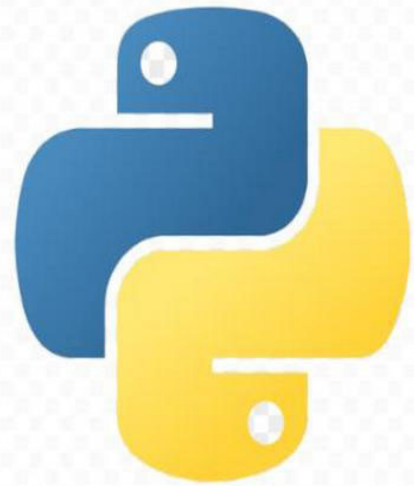
Interfaz uniforme: para la transferencia de datos en un sistema REST, este aplica acciones concretas (POST, GET, PUT y DELETE) sobre los recursos, siempre y cuando estén identificados con una URI.

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: ya que esta 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.

Finalmente, esta no es ni la mejor, ni la única forma de hacer las cosas, los microservicios, se pueden construir apoyándose en varios 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 convenga según las necesidades.



python