

API REST PHP Y MARIADB

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

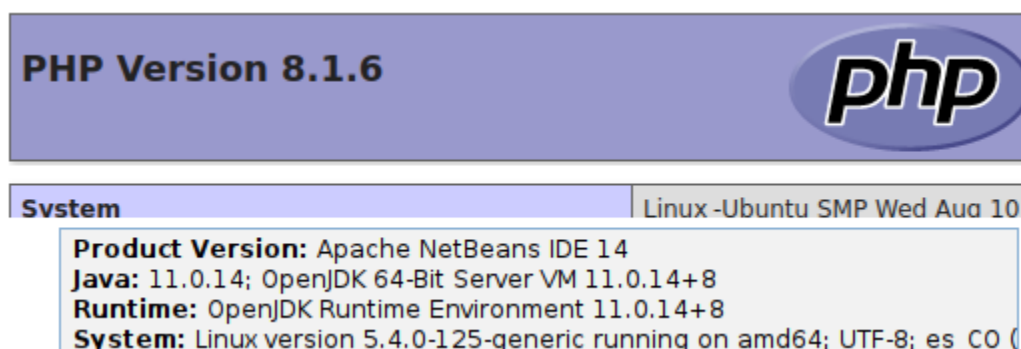
CONTENIDO DOCUMENTO

1. Creación Base Datos MariaDB
2. Configuración Proyecto
3. Funcionamiento
4. Esquema de Funcionamiento
5. Pruebas Consumo Api con Postman
6. Pruebas Consumo Api con Cliente Propio
7. Porqué 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 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 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: Ver 15.1 Distrib 10.3.34-MariaDB, Una vez que hayas creado la BD, procederemos a crear el API en PHP, yo trabajé con PHP 8.1 y Xampp 8.1.6, utilizando el Ide: NetBeans 14 sobre sistema operativo Linux.



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 paquetes 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, 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.
- **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 se subió a Github como txt, 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]
```

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 pregunte por un registro específico, cuando ese registro es encontrado, éste 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.

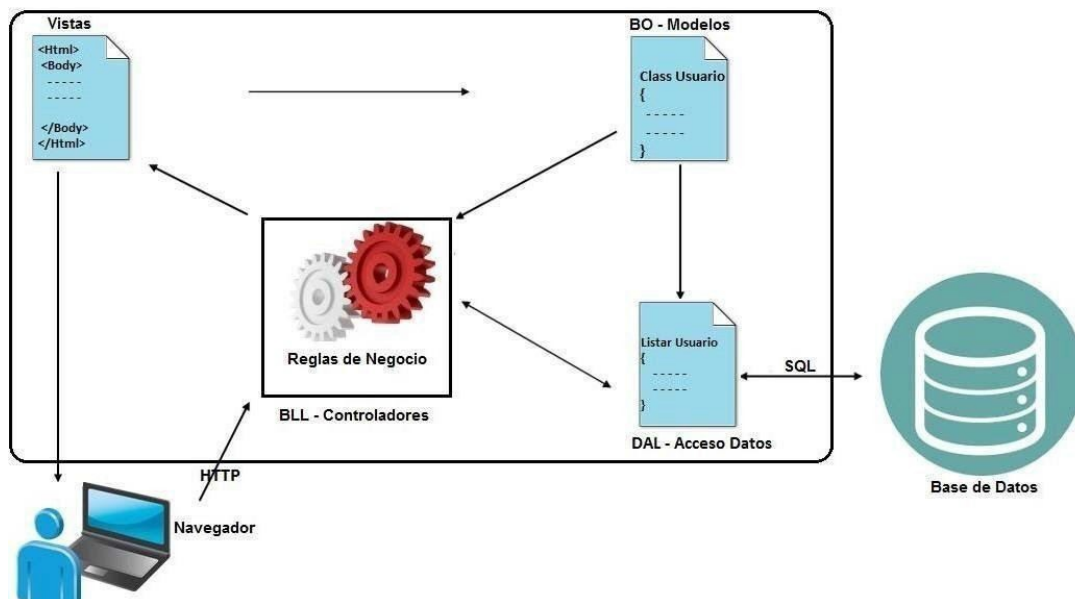
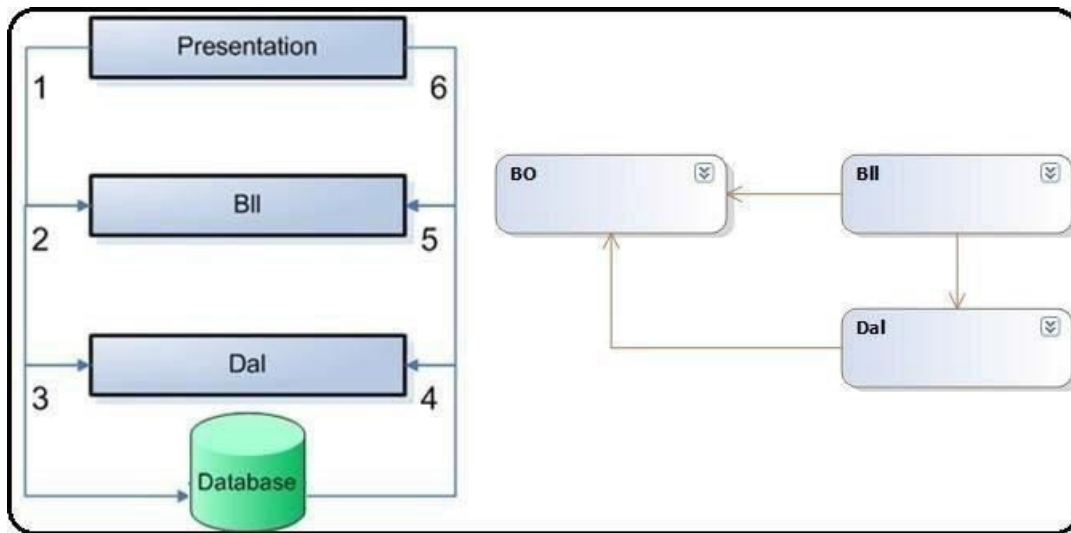
Nunca se llega directamente al DAL, siempre todo deberá pasar primero por el Controlador que será nuestro manager, encargado de orquestar, controlar y definir que método del DAL invocar, para que sea ésta última quien devuelva los resultados esperados, y dar así respuesta a las diferentes peticiones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

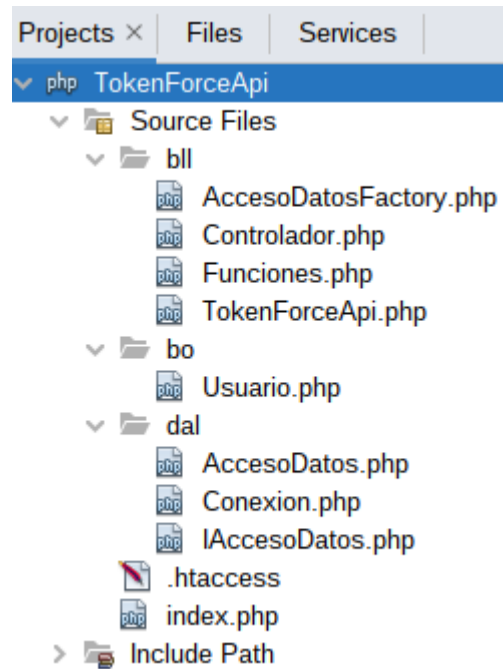
La finalidad de separar los sistemas en capas, no es otra que facilitar su posterior mantenimiento, donde cada capa expone servicios que otras aplicaciones o capas pueden consumir, lo que se traduce en una simplicidad conceptual, de alta cohesión y bajo acoplamiento, facilitando así su reutilización,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

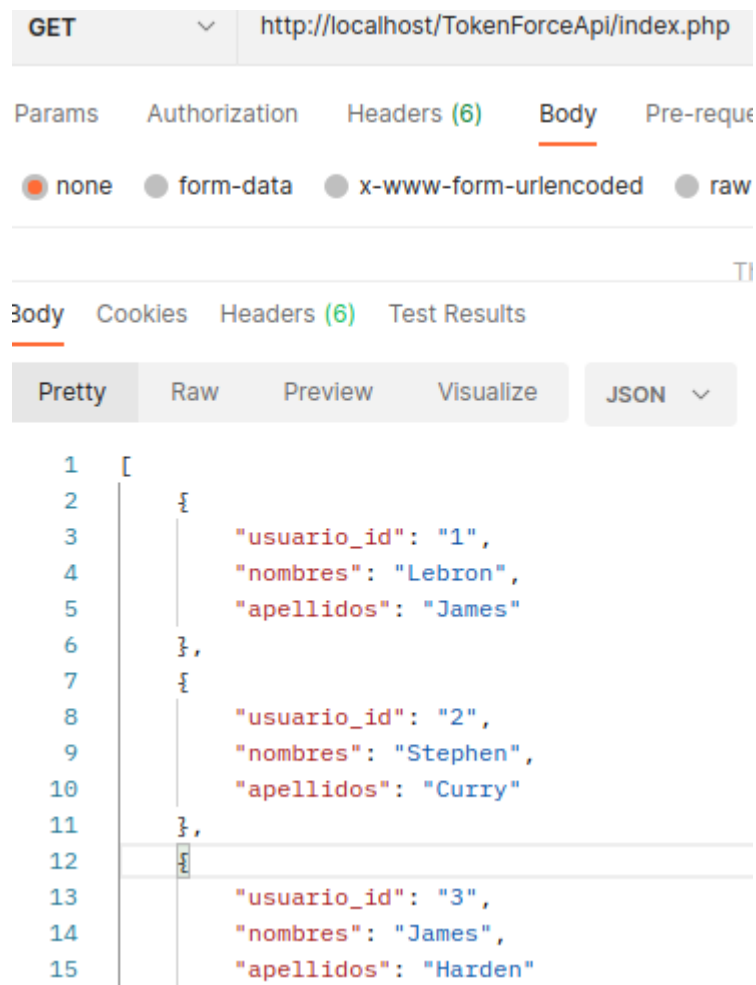


5. Pruebas Consumo API con Postman

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

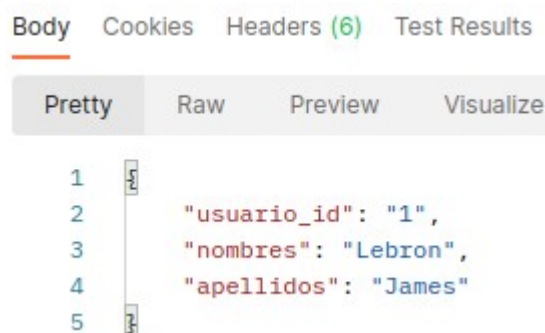
Obtener Todos Los Usuarios GET

<https://localhost/TokenForceApi/index.php>



Obtener Un Usuario GET

<http://localhost/TokenForceApi/index.php/usuarios/1>



Agregar Un Usuario POST

<http://localhost/TokenForceApi/index.php>

The screenshot shows a REST client interface. At the top, the method is set to **POST** and the URL is `http://localhost/TokenForceApi/index.php`. Below this, there are tabs for **Params**, **Authorization**, **Headers (8)**, **Body** (which is selected), and **Pre-request Script**. Under the **Body** tab, there are radio buttons for **none**, **form-data**, **x-www-form-urlencoded**, **raw** (which is selected), and **binary**. The request body is a JSON object:

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

. Below the request, there are tabs for **Body** (selected), **Cookies**, **Headers (6)**, and **Test Results**. Under the **Body** tab, there are buttons for **Pretty**, **Raw**, **Preview**, **Visualize**, and a **JSON** dropdown menu. The response body is a JSON object:

```
{  "status": "Exito",  "message": "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/TokenForceApi>

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. At the top, the method is set to 'POST' and the URL is 'http://localhost/TokenForceApi/index.php'. Below this, the 'Body' tab is selected, showing a JSON payload:

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

. The 'raw' radio button is selected. Below the body, the 'Body' tab is selected in the response section, showing a JSON response:

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

. The 'Pretty' radio button is selected for the response.

```
1 {
2   "usuario_id": "0",
3   "nombres": "Russell",
4   "apellidos": "Westbrook"
5 }
```

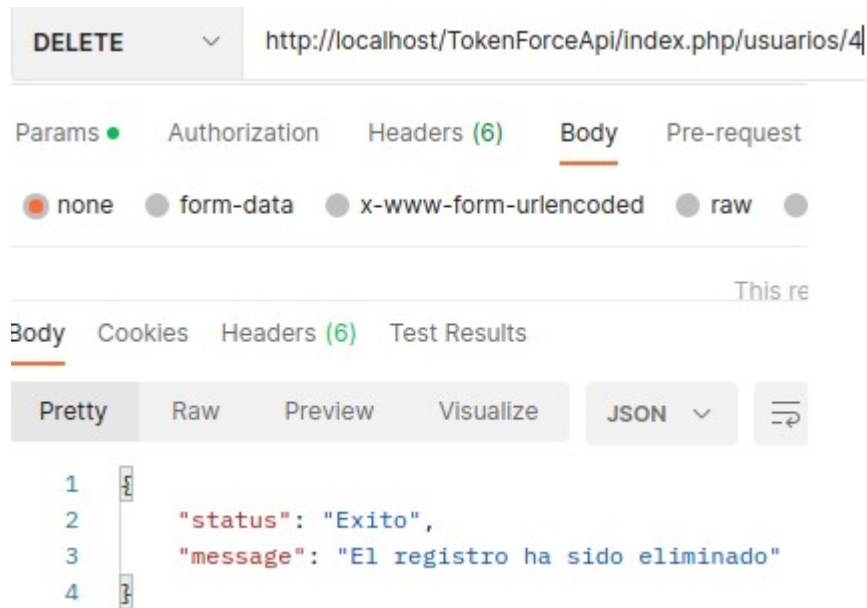
```
1 {
2   "status": "Exito",
3   "message": "El nuevo registro ha sido grabado"
4 }
```

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/TokenForceApi/usuarios/4>



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 accedendo la base de datos".

6. Pruebas Consumo API con Cliente Propio

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.

Consultar Todos los Usuarios - GET

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



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

ID	Nombres	Apellidos
1	Lebron	James
2	Stephen	Curry
3	James	Harden

Consultar un Usuario - GET

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

Usuario_Id *

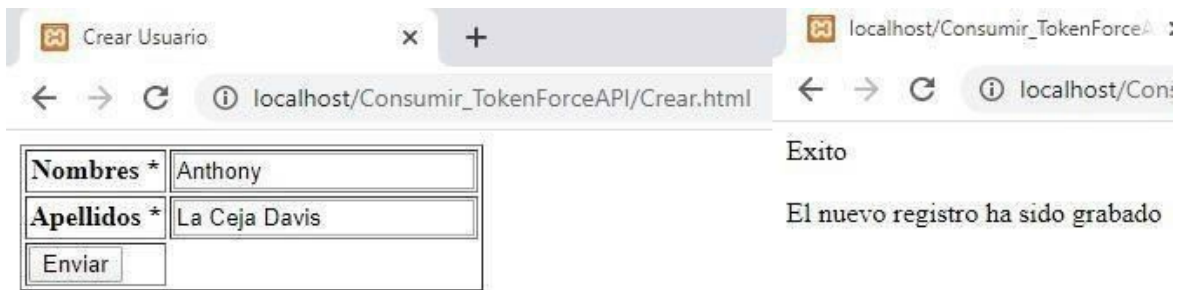
← → ↻ ⓘ localhost/Consum

Nombres:

Apellidos:

Agregar Un Usuario - POST

Seleccionando opción: Crear Usuario



The screenshot shows a web browser with two tabs. The active tab is titled 'Crear Usuario' and shows a form with the following fields:

Nombres *	Anthony
Apellidos *	La Ceja Davis
<input type="button" value="Enviar"/>	

To the right of the form, a success message is displayed:

Exito
El nuevo registro ha sido grabado

Consultando ese usuario que se acaba de crear



The screenshot shows a web browser with a form to look up a user by ID:

Usuario_Id *

Below the form, the retrieved user information is displayed:

Nombres:
Apellidos:

Actualizar Un Usuario - PUT

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



The screenshot shows a web browser with two tabs. The active tab is titled 'Actualizar Usuario' and shows a form with the following fields:

Usuario_Id *	5
Nombres *	Anthony
Apellidos *	Davis
<input type="button" value="Enviar"/>	

To the right of the form, a success message is displayed:

Exito
El registro ha sido actualizado

Eliminar Un Usuario - DELETE

Seleccionando opción: Eliminar un Usuario



The screenshot shows a web browser with a form to delete a user by ID:

Usuario_Id *

Below the form, a success message is displayed:

Exito
El registro ha sido eliminado

7. Porqué 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: arquitectura jerárquica entre los componentes. 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 en PHP, 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 convenga según las necesidades.

Pero, Cuidado con esto

En las operaciones de Get y Delete, se notará que se envió el parámetro del ID de usuario en la url.

Eso no se debe hacer, pues es un error grave, ya que un atacante estará viendo ese valor, y por ahí estamos dejando la puerta abierta para que hagan daños.

Se debe enviar como JSON en el cuerpo igual que se hizo con el Post y Put, y/o si se decide enviarlo en la url, por lo menos debe estar encriptado.