Tutorial CRUD con Javalin

1. Objetivo del Artefacto

El objetivo de este documento es construir un servidor API RESTful funcional utilizando Javalin. Nos centraremos exclusivamente en la capa de API: definir rutas (endpoints), gestionar peticiones HTTP (GET, POST, PUT, DELETE) y manejar la serialización/deserialización de datos JSON.
En esta primera fase, **simularemos (mock)** toda la lógica de negocio y persistencia. No se utilizará ninguna base de datos. Esto permite comprender y probar el funcionamiento de Javalin de forma aislada.

2. Requisitos y Dependencias

- IntelliJ IDEA (o IDE de preferencia)
- JDK 17 o superior
- Maven

Dependencias del pom.xml

Para esta fase, solo necesitamos dos dependencias. Notar la **ausencia** de sqlite-jdbc.

```
<dependencies>
    <!-- El micro-framework web. Gestiona las rutas y peticiones HTTP. -->
    <dependency>
        <groupId>io.javalin</groupId>
        <artifactId>javalin</artifactId>
        <version>6.1.3</version>
    </dependency>

    <!-- Librería para serializar (Java -> JSON) y deserializar (JSON -> Java). -->
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.10.1</version>
    </dependency>
</dependencies>
```

3. Estructura del Proyecto

A. Modelo de Datos (POJO)

Necesitamos una clase Java que represente la estructura de datos que esperamos recibir

(Payload) y enviar.

```
// src/main/java/User.java
public class User {
    // Nota: los campos pueden ser públicos para GSON o privados con getters/setters.
    public int id;
    public String name;
    public String email;

    // Constructor para crear instancias desde Java (ej. en mocks)
    public User(int id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }

    // Constructor vacío que GSON necesita para deserializar JSON a objetos User
    public User() {
    }
}
```

B. Servidor API (Main.java)

Esta es la clase principal que inicia el servidor y define los *handlers* (controladores) para cada ruta.

```
// src/main/java/Main.java
import com.google.gson.Gson;
import io.javalin.Javalin;
import io.javalin.http.Context;
import java.util.List;
import java.util.Map;
import java.util.Random;

public class Main {

    // Instancia de GSON para manejar JSON
    private static final Gson gson = new Gson();
    // Un generador de IDs falsos para la simulación
    private static final Random random = new Random();

    public static void main(String[] args) {
        Javalin app = Javalin.create().start(7000);
        System.out.println("Servidor API (Mock) iniciado en http://localhost:7000");
```

```
    // Definición de Endpoints
    app.post("/users", Main::crearUsuario);
    app.get("/users", Main::obtenerTodosLosUsuarios);
    app.get("/users/:id", Main::obtenerUsuarioPorId);
    app.put("/users/:id", Main::actualizarUsuario);
    app.delete("/users/:id", Main::eliminarUsuario);
}

// --- Implementación de los Handlers (Controladores) ---

/**
 * CREATE (Mock)
 * Simula la creación de un usuario.
 * Lee el JSON del body, lo convierte a User, y devuelve un mensaje de éxito.
 */
private static void crearUsuario(Context ctx) {
    // 1. Deserializar el JSON del body a un objeto User
    User newUser = gson.fromJson(ctx.body(), User.class);

    // 2. Simulación de Lógica:
    // En un caso real, aquí iría la lógica de BD (INSERT).
    int simulatedId = random.nextInt(1000) + 1; // Genera un ID falso

    // 3. Respuesta al cliente
    ctx.status(201); // 201 = Created
    ctx.json(Map.of(
        "status", "creado (simulación)",
        "usuario", newUser.name,
        "idAsignado", simulatedId
    ));
}

/**
 * READ ALL (Mock)
 * Simula la devolución de una lista de usuarios.
 */
private static void obtenerTodosLosUsuarios(Context ctx) {
    // 1. Simulación de Lógica:
    // En un caso real, aquí iría la lógica de BD (SELECT *)
    List<User> mockUsers = List.of(
        new User(101, "Ana (Mock)", "ana.mock@example.com"),
        new User(102, "Luis (Mock)", "luis.mock@example.com")
    );
```

```
    // 2. Respuesta al cliente
    ctx.status(200); // 200 = OK
    ctx.json(mockUsers);
}

/**
 * READ ONE (Mock)
 * Simula la búsqueda de un usuario por su ID.
 */
private static void obtenerUsuarioPorId(Context ctx) {
    // 1. Obtener el parámetro de la URL
    String id = ctx.pathParam("id");

    // 2. Simulación de Lógica:
    // En un caso real, aquí iría (SELECT ... WHERE id = ?)
    User mockUser = new User(
        Integer.parseInt(id), // Usa el ID de la URL
        "Usuario Simulado " + id,
        "user." + id + "@example.com"
    );

    // 3. Respuesta al cliente
    ctx.status(200);
    ctx.json(mockUser);
}

/**
 * UPDATE (Mock)
 * Simula la actualización de un usuario.
 */
private static void actualizarUsuario(Context ctx) {
    // 1. Obtener el parámetro de la URL
    String id = ctx.pathParam("id");

    // 2. Deserializar el JSON del body
    User updatedUser = gson.fromJson(ctx.body(), User.class);

    // 3. Simulación de Lógica:
    // En un caso real, aquí iría (UPDATE ... WHERE id = ?)

    // 4. Respuesta al cliente
    ctx.status(200);
```

```
        ctx.json(Map.of(
            "status", "actualizado (simulación)",
            "id", id,
            "nuevoNombre", updatedUser.name
        ));
    }

    /**
     * DELETE (Mock)
     * Simula la eliminación de un usuario.
     */
    private static void eliminarUsuario(Context ctx) {
        // 1. Obtener el parámetro de la URL
        String id = ctx.pathParam("id");

        // 2. Simulación de Lógica:
        // En un caso real, aquí iría (DELETE ... WHERE id = ?)

        // 3. Respuesta al cliente
        ctx.status(200); // También es común 204 (No Content)
        ctx.json(Map.of(
            "status", "eliminado (simulación)",
            "id", id
        ));
    }
}
```

4. Pruebas de la API (Mock) con cURL

Inicie la clase Main.java. El servidor estará escuchando en localhost:7000. Abra una terminal para probar los endpoints.

A. Probar CREATE (POST)

```
curl -X POST http://localhost:7000/users \
    -H "Content-Type: application/json" \
    -d '{"name":"Carlos", "email":"carlos@gmail.com"}'

# Respuesta esperada (el ID será aleatorio):
# {
#   "status": "creado (simulación)",
#   "usuario": "Carlos",
#   "idAsignado": 734
```

```
# }
```

B. Probar READ ALL (GET)

```
curl http://localhost:7000/users

# Respuesta esperada:
# [
#   {"id":101,"name":"Ana (Mock)","email":"ana.mock@example.com"},
#   {"id":102,"name":"Luis (Mock)","email":"luis.mock@example.com"}
# ]
```

C. Probar READ ONE (GET)

```
curl http://localhost:7000/users/42

# Respuesta esperada:
# {
#   "id": 42,
#   "name": "Usuario Simulado 42",
#   "email": "user.42@example.com"
# }
```

D. Probar UPDATE (PUT)

```
curl -X PUT http://localhost:7000/users/42 \
    -H "Content-Type: application/json" \
    -d '{"name":"Carlos Actualizado", "email":"carlos.nuevo@gmail.com"}'

# Respuesta esperada:
# {
#   "status": "actualizado (simulación)",
#   "id": "42",
#   "nuevoNombre": "Carlos Actualizado"
# }
```

E. Probar DELETE (DELETE)

```
curl -X DELETE http://localhost:7000/users/42

# Respuesta esperada:
# {
```

```
#   "status": "eliminado (simulación)",
#   "id": "42"
# }
```

5. Conclusión y Siguientes Pasos

Hemos implementado con éxito un servidor API REST con Javalin. El servidor define los cinco endpoints estándar de un CRUD (/users) y gestiona correctamente las peticiones HTTP, incluyendo la deserialización de *payloads* JSON y la serialización de respuestas.
Toda la lógica de negocio está actualmente simulada (*mocked*). Los *handlers* devuelven datos estáticos o procesan la entrada sin persistirla.
El siguiente paso lógico (Parte 2) será reemplazar estas simulaciones por operaciones reales de base de datos, introduciendo sqlite-jdbc para la persistencia de datos.