Tutorial REST con Javelin.

Módulo 1: Fundamentos Teóricos - ¿Qué es REST?

REST (REpresentational State Transfer) no es un protocolo ni un estándar, sino un **estilo arquitectónico** para sistemas distribuidos. Fue definido por Roy Fielding en su disertación doctoral (2000) al analizar las propiedades que hacían la Web (HTTP) tan escalable y resiliente.
Si un sistema cumple con las siguientes 6 restricciones, puede considerarse "RESTful".

1. Arquitectura Cliente-Servidor

Separación total de incumbencias. El cliente gestiona la UI y el estado de la interfaz; el servidor gestiona la lógica de negocio y el almacenamiento. Pueden evolucionar de forma independiente.

2. Sin Estado (Stateless)

La restricción más importante. Cada petición (request) del cliente debe contener **toda la información necesaria** para ser procesada. El servidor no debe almacenar ningún contexto de sesión del cliente entre peticiones.

- **Impacto:** Escalabilidad horizontal (cualquier servidor puede atender cualquier petición), fiabilidad (si un servidor cae, la petición se reintenta en otro) y visibilidad (la petición es atómica).

3. Capacidad de Caché (Cacheable)

Las respuestas deben autodefinirse como cacheables o no cacheables. Esto permite al cliente o a los intermediarios (proxies, CDNs) reutilizar respuestas, mejorando drásticamente el rendimiento percibido. GET es inherentemente cacheable; POST no lo es.

4. Sistema de Capas (Layered System)

El cliente no sabe (ni debe saber) si se comunica con el servidor final, un balanceador de carga, un proxy inverso o un API Gateway. Esto permite modularizar la arquitectura (seguridad, balanceo) sin impactar al cliente.

5. Interfaz Uniforme (Uniform Interface)

El núcleo de la práctica de REST. Es la restricción que define la interacción. Se subdivide en:

- **a. Identificación de Recursos:** Los recursos (ej. un usuario, un producto) se identifican unívocamente mediante URIs.
    - *Correcto:* https://api.empresa.com/usuarios/123
    - *Incorrecto (RPC-style):* https://api.empresa.com/getUsuario?id=123
- **b. Manipulación de Recursos a través de Representaciones:** El cliente interactúa

con una *representación* del recurso (ej. un JSON o XML), no con el recurso en sí (ej. el objeto en la BBDD).

- **c. Mensajes Autodescriptivos:** Cada mensaje (petición/respuesta) contiene metadatos para ser entendido.
    - *Petición:* POST /usuarios, Content-Type: application/json, Body: {"nombre": ...}.
    - *Respuesta:* 201 Created, Location: /usuarios/456.
- **d. HATEOAS (Hypermedia as the Engine of Application State):** La respuesta del servidor debe incluir hipervínculos que guíen al cliente sobre las siguientes acciones posibles.
    - Esta es la restricción menos implementada en la práctica. Muchas APIs "RESTful" son en realidad "APIs HTTP/JSON" que cumplen todo excepto HATEOAS. Se omite por pragmatismo, ya que acopla al cliente a la estructura de enlaces definida por el servidor, lo cual puede añadir complejidad en el *parsing* del cliente.

6. Código bajo Demanda (Code-On-Demand) (Opcional)

El servidor puede enviar lógica ejecutable (ej. JavaScript) al cliente. Es lo que permite las *Single Page Applications* (SPAs).

Módulo 2: Configuración del Proyecto (Maven)

Usaremos Maven para gestionar las dependencias.
pom.xml (extracto de dependencias):

```
<dependencies>
  <!-- El servidor web ligero (Jetty embebido) -->
  <dependency>
    <groupId>io.javalin</groupId>
    <artifactId>javalin</artifactId>
    <version>6.1.3</version>
  </dependency>

  <!-- El motor de JSON (POJO <-> JSON) -->
  <!-- Javalin lo autoconfigura si está presente -->
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.17.1</version>
  </dependency>

  <!-- Logging (Binding simple para SLF4J) -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>2.0.13</version>
```

```
    </dependency>
</dependencies>
```

(Dependencias):

- **Javalin:** Es un micro-framework. A diferencia de Spring, no utiliza *reflection* masiva ni inyección de dependencias. Simplemente mapea rutas HTTP a *handlers* de Java. Es, en esencia, un contenedor de *servlets* (Jetty) muy simplificado.
- **Jackson-Databind:** Javalin detecta automáticamente que Jackson está en el *classpath* y lo usa para toda la serialización/deserialización de JSON (ctx.json() y ctx.bodyAsClass()).
- **slf4j-simple:** Javalin usa la *fachada* (facade) SLF4J para loguear. slf4j-simple es una *implementación* (binding) que simplemente vuelca todos los logs a System.err.

Módulo 3: Creación de Clases (POJOs y Servicio Mock)

Definimos el modelo de datos y simulamos la capa de persistencia.

Paso 1: El Modelo (POJO)

Este es el "Recurso" en términos de REST.
src/main/java/com/ejemplo/Usuario.java

```
package com.ejemplo;

public class Usuario {

    private String id;
    private String nombre;
    private int edad;

    // Constructor vacío OBLIGATORIO para Jackson (deserialización POST)
    public Usuario() {
    }

    // Constructor para nuestro mock
    public Usuario(String id, String nombre, int edad) {
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
    }

    // Getters y Setters OBLIGATORIOS para Jackson (serialización GET)
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
```

```
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }
    public int getEdad() { return edad; }
    public void setEdad(int edad) { this.edad = edad; }
}
```

(POJOs y Jackson):
Jackson (y otras librerías de binding como JAXB) usan reflexión (reflection) para instanciar objetos y poblar sus campos.

1. Para **Deserializar** (JSON -> Java), Jackson necesita llamar al **constructor vacío** (new Usuario()).
2. Luego, usa los **métodos set...()** (siguiendo la convención JavaBeans) para asignar los valores de los campos del JSON ("nombre" -> setNombre(...)).
3. Para **Serializar** (Java -> JSON), Jackson usa los **métodos get...()** para leer los valores y construir el JSON.

Paso 2: El Servicio Mock (Fake)

Simulamos la base de datos con un Singleton y un HashMap en memoria.
src/main/java/com/ejemplo/MockUsuarioService.java

```
package com.ejemplo;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Esto simula nuestra Base de Datos (ej. un Repositorio).
 * Es un Singleton simple para mantener el estado en memoria.
 */
public class MockUsuarioService {

    // Simulación de una tabla de base de datos (Clave = ID, Valor = Objeto)
    private Map<String, Usuario> baseDeDatos = new HashMap<>();

    // Simulación de una secuencia auto-incremental de BBDD
    // Usamos AtomicInteger para seguridad en concurrencia (thread-safety)
    private AtomicInteger sequence = new AtomicInteger(0);

    // Singleton pattern
    private static final MockUsuarioService instancia = new MockUsuarioService();

    // Datos iniciales (mock)
    private MockUsuarioService() {
```

```
        create(new Usuario(null, "Ana", 30));
        create(new Usuario(null, "Beto", 45));
    }

    public static MockUsuarioService getInstancia() {
        return instancia;
    }

    // --- Métodos CRUD Mockeados ---

    public Collection<Usuario> getAll() {
        return baseDeDatos.values();
    }

    public Usuario getById(String id) {
        return baseDeDatos.get(id);
    }

    public Usuario create(Usuario usuario) {
        String nuevoId = String.valueOf(sequence.incrementAndGet());
        usuario.setId(nuevoId);
        baseDeDatos.put(nuevoId, usuario);
        return usuario;
    }
}
```

Nota del Doctorado (Mock vs. Fake):
Mencionaste Mockito. Es crucial la distinción terminológica (Gerard Meszaros):

- **Mock (usado con Mockito):** Es un objeto para *pruebas unitarias*. No tiene lógica real. Se usa para *verificar el comportamiento* (ej. verify(mockService, times(1)).create(any())).
- **Fake (lo que hicimos):** Es una *implementación funcional* pero simplista (ej. usando un HashMap en lugar de SQL). No es para producción, pero es ideal para *pruebas de integración* o para ejecutar la aplicación en desarrollo sin una BBDD real.

Módulo 4: Creando la API REST con Javalin

Conectamos el servidor HTTP (Javalin) a nuestra lógica *fake*.

src/main/java/com/ejemplo/ApiRest.java

```
package com.ejemplo;

import io.javalin.Javalin;
import io.javalin.http.Context;
import io.javalin.http.HttpStatus;
import java.util.Map;
import java.util.Collection;

public class ApiRest {

    // Obtenemos la instancia de nuestro servicio mockeado
    private static MockUsuarioService servicio = MockUsuarioService.getInstancia();

    public static void main(String[] args) {

        // 1. Crear y configurar el servidor Javalin
        Javalin app = Javalin.create(config -> {
            config.http.defaultContentType = "application/json"; // Default a JSON
            config.routing.contextPath = "/api/v1"; // Prefijo para todas las rutas
        }).start(7070); // Iniciar en el puerto 7070

        System.out.println("Servidor Javalin iniciado en http://localhost:7070/api/v1");

        // 2. Definir los Endpoints REST (Rutas)
        // Se mapea (Verbo, Path) -> Handler

        // REST: GET /api/v1/usuarios
        app.get("/usuarios", ApiRest::getAllUsuarios);

        // REST: GET /api/v1/usuarios/{id}
        app.get("/usuarios/{id}", ApiRest::getUsuarioById);

        // REST: POST /api/v1/usuarios
        app.post("/usuarios", ApiRest::createUsuario);
    }

    /**
     * Este es el "Handler" o "Controlador" para GET /usuarios
     */
    private static void getAllUsuarios(Context ctx) {
        // 1. Llama al servicio
        Collection<Usuario> usuarios = servicio.getAll();
```

```
    // 2. Javalin serializa la Colección a un array JSON automáticamente
    ctx.json(usuarios);

    // 3. Javalin establece status 200 OK por defecto
}

/**
 * Handler para GET /usuarios/{id}
 */
private static void getUsuarioById(Context ctx) {
    // 1. Extraer el 'path parameter'
    String id = ctx.pathParam("id");

    // 2. Llama al servicio
    Usuario usuario = servicio.getById(id);

    // 3. Manejo de respuesta
    if (usuario != null) {
        ctx.json(usuario); // 200 OK (implícito)
    } else {
        // Recurso no encontrado. Clave en REST: usar códigos HTTP correctos.
        ctx.status(HttpStatus.NOT_FOUND); // 404
        ctx.jsonMap(Map.of("error", "Usuario no encontrado con id: " + id));
    }
}

/**
 * Handler para POST /usuarios
 */
private static void createUsuario(Context ctx) {
    try {
        // 1. Deserializar el Body JSON a nuestro objeto Java (POJO)
        Usuario nuevoUsuario = ctx.bodyAsClass(Usuario.class);

        // 2. Llama al servicio para crearlo
        Usuario creado = servicio.create(nuevoUsuario);

        // 3. Respuesta REST correcta: 201 Created
        ctx.status(HttpStatus.CREATED); // 201
        ctx.json(creado);

    } catch (Exception e) {
```

```
            // Si el JSON está mal formado (ej. "edad" como string)
            ctx.status(HttpStatus.BAD_REQUEST); // 400
            ctx.jsonMap(Map.of("error", "JSON de entrada inválido", "detalle", e.getMessage()));
        }
    }
}
```

El objeto Context ctx es el corazón de Javalin. Es una implementación del patrón Facade.

- Abstrae y simplifica los complejos objetos HttpServletRequest y HttpServletResponse del API de Servlets de Java.
- ctx.pathParam("id") es un atajo para request.getPathInfo() + parsing.
- ctx.bodyAsClass(T.class) es un atajo para request.getInputStream() + ObjectMapper.readValue().
- ctx.json(obj) es un atajo para response.setContentType("application/json") + response.getWriter().write(...) + ObjectMapper.writeValueAsString().
- ctx.status(HttpStatus.XXX) es un atajo para response.setStatus(...).

Módulo 5: Pruebas con curl (La Práctica)

Abre una terminal y ejecuta la clase ApiRest.java. Luego, ejecuta los siguientes comandos curl.

Prueba 1: GET (Obtener todos los usuarios)

Solicitamos la colección completa de recursos usuarios.

```
# -v (verbose) nos muestra los headers de petición y respuesta
curl -v http://localhost:7070/api/v1/usuarios
```

Respuesta esperada:

```
< HTTP/1.1 200 OK
< Content-Type: application/json
...
[
 {"id":"1","nombre":"Ana","edad":30},
 {"id":"2","nombre":"Beto","edad":45}
]
```

- **Análisis REST:** Petición GET idempotente. El servidor responde 200 OK (éxito) y la representación JSON de la colección.

Prueba 2: GET (Recurso no encontrado)

Probamos el manejo de errores (nuestro 404 Not Found).

```
# -i (include) para ver solo el header de estado
```

curl -i http://localhost:7070/api/v1/usuarios/99

Respuesta esperada:
HTTP/1.1 404 Not Found
Content-Type: application/json
...
{"error":"Usuario no encontrado con id: 99"}

- **Análisis REST:** Perfecto. No devolvemos un 200 OK con un error en el body. Usamos la semántica de HTTP (404) para comunicar el estado de la aplicación.

Prueba 3: POST (Crear un nuevo usuario)

Enviamos una nueva representación de recurso para su creación.
curl -v -X POST http://localhost:7070/api/v1/usuarios \
-H "Content-Type: application/json" \
-d '{"nombre":"Carlos", "edad":25}'

- -X POST: Especifica el *verbo* (método) HTTP.
- -H "...": Define el *header* Content-Type. Es **fundamental** (Mensaje Autodescriptivo de REST).
- -d '...': El *data* (body) de la petición.

Respuesta esperada:
> POST /api/v1/usuarios HTTP/1.1
> Content-Type: application/json
...
< HTTP/1.1 201 Created
< Content-Type: application/json
...
{"id":"3","nombre":"Carlos","edad":25}

- **Análisis REST:** El servidor respondió con 201 Created (el estado semánticamente correcto para un POST exitoso que crea un recurso) y nos devolvió la representación del recurso recién creado.

Prueba 4: Verificación del POST

Si volvemos a ejecutar la Prueba 1, el estado del servidor (nuestro *Fake*) debe haber cambiado.
curl http://localhost:7070/api/v1/usuarios

Respuesta esperada:
[
 {"id":"1","nombre":"Ana","edad":30},
 {"id":"2","nombre":"Beto","edad":45},

{"id":"3","nombre":"Carlos","edad":25}
]

- **Análisis REST:** El sistema funciona. El estado fue transferido (POST) y ahora está disponible para ser recuperado (GET).

Módulo 6: Siguientes Pasos (Retos)

1. **Implementar PUT (Actualización):**
    - Añadir app.put("/usuarios/{id}", ...).
    - El *handler* debe usar ctx.bodyAsClass(Usuario.class) y ctx.pathParam("id").
    - Semántica REST: PUT es idempotente y reemplaza el recurso completo.
    - Debe devolver 200 OK (si actualiza) o 204 No Content.
2. **Implementar DELETE (Borrado):**
    - Añadir app.delete("/usuarios/{id}", ...).
    - Debe devolver 204 No Content (éxito, pero sin cuerpo de respuesta).
3. **Manejo de Excepciones (Exception Handling):**
    - Crear una excepción custom (ej. UsuarioNotFoundException).
    - Lanzarla desde el servicio.
    - Capturarla en Javalin usando app.exception(UsuarioNotFoundException.class, (e, ctx) -> { ... }). Esto centraliza el manejo de errores.