Fundamentos de Arquitectura REST e Implementación Práctica con Javalin

Java 23 + Javalin 6.3.0

1. Introducción y Objetivos

El propósito de este documento es establecer una base teórica sólida sobre la arquitectura REST (Representational State Transfer) y aplicarla en la construcción de una API funcional utilizando **Java 23** y el framework **Javalin 6.3.0**.

A diferencia de frameworks "opinados" y pesados como Spring Boot, Javalin ofrece una **abstracción ligera** sobre la capa de servlets (Jetty), permitiendo un control granular del ciclo de vida de la petición HTTP con una configuración mínima.

¿Por qué Javalin?

- **Simplicidad**: API intuitiva con expresiones lambda y method references
- **Ligereza**: Sin dependencias innecesarias, solo lo esencial
- **Rendimiento**: Basado en Jetty, uno de los servidores embebidos más rápidos
- **Productividad**: Menor boilerplate comparado con Spring Boot
- **Moderno**: Aprovecha las características de Java moderno (Records, Pattern Matching, etc.)

2. Marco Teórico: Arquitectura REST

REST no es un protocolo, sino un **estilo arquitectónico** definido por Roy Fielding en su tesis doctoral (2000). Para que un servicio sea considerado RESTful, debe cumplir con restricciones específicas.

2.1 Restricciones Fundamentales de REST

1. Interfaz Uniforme

La interacción entre cliente y servidor se realiza a través de una interfaz estandarizada. En el contexto de HTTP, esto significa usar los verbos (métodos) correctamente:

| Método HTTP | Propósito | Idempotente | Seguro | ||--|-|| | **GET** | Recuperar una representación de un recurso | ✓ | ✓ | | **POST** | Crear un recurso nuevo o procesar datos | ✕ | ✕ | | **PUT** | Reemplazar completamente un recurso existente | ✓ | ✕ | | **PATCH** | Actualizar parcialmente un recurso | ✕ | ✕ | | **DELETE** | Eliminar un recurso | ✓ | ✕ |

Conceptos clave:

- **Idempotente**: Ejecutar la operación múltiples veces produce el mismo resultado que ejecutarla una vez
- **Seguro**: No modifica el estado del servidor (solo lectura)

2. Sin Estado (Stateless)

Cada petición del cliente al servidor debe contener **toda la información necesaria** para entender y procesar dicha petición. El servidor no debe guardar el contexto de sesión del cliente entre peticiones.

Implicaciones:

- No hay sesiones del lado del servidor
- Cada request incluye credenciales/tokens
- Mejora la escalabilidad (no hay afinidad de sesión)

3. Sistema de Recursos

Todo es un **recurso** identificable unívocamente mediante URIs:

- `/users` → Colección de usuarios
- `/users/42` → Usuario con ID 42
- `/users/42/orders` → Órdenes del usuario 42

4. Representaciones

Los recursos pueden tener múltiples representaciones (JSON, XML, HTML). El cliente y servidor negocian el formato mediante headers:

```
Accept: application/json
Content-Type: application/json
```

3. Configuración del Entorno Maven

3.1 Análisis del `pom.xml`

El fallo común en tutoriales introductorios es **omitir la capa de logging**. Javalin requiere una implementación de la interfaz SLF4J. Sin ella, el servidor embebido Jetty no iniciará correctamente.

```
<properties>
    <maven.compiler.source>23</maven.compiler.source>
    <maven.compiler.target>23</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <!-- Versiones Centralizadas -->
    <javalin.version>6.3.0</javalin.version>
    <slf4j.version>2.0.16</slf4j.version>
    <gson.version>2.11.0</gson.version>
</properties>
```

3.2 Dependencias Críticas

1. Javalin Core (Framework Web)

```
<dependency>
    <groupId>io.javalin</groupId>
    <artifactId>javalin</artifactId>
    <version>6.3.0</version>
</dependency>
```

Proporciona la abstracción sobre Jetty y el DSL para definir rutas.

2. SLF4J Simple (Logging - OBLIGATORIO)

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>2.0.16</version>
</dependency>
```

¿Por qué es crítico?

- Jetty (motor interno) usa SLF4J para logging
- Sin una implementación, obtendrás `ClassNotFoundException`
- Alternativas: logback, log4j2 (más configurables pero más pesadas)

3. GSON (Serialización JSON)

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.11.0</version>
</dependency>
```

Convierte objetos Java ↔ JSON de forma transparente mediante reflexión.

4. Implementación del Modelo de Dominio

4.1 Clase `User.java`

```
/**
 * Representa un recurso de Usuario en el sistema.
 * POJO (Plain Old Java Object) simple.
 */
public class User {
    public int id;
    public String name;
    public String email;
```

```
    // Constructor vacío requerido por GSON para reflexión
    public User() { }

    public User(int id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }
}
```

4.2 Alternativa Moderna: Java Records

En **Java 14+** podríamos usar `record` para reducir boilerplate:

```
public record User(int id, String name, String email) { }
```

Ventajas:

- Inmutable por defecto
- Implementa automáticamente `equals()`, `hashCode()`, `toString()`
- Menos código

Desventaja para este tutorial:

- GSON requiere configuración adicional para deserializar records
- Menos explícito para fines didácticos

5. Implementación del Servidor REST

5.1 Estructura de `Main.java`

```
public class Main {
    // Singleton de Gson para manejo de JSON
    private static final Gson gson = new Gson();

    // Base de datos en memoria (Volátil) para simulación
    private static final List<User> users = new ArrayList<>();

    public static void main(String[] args) {
        // Datos iniciales
        users.add(new User(1, "Alice", "alice@university.edu"));
        users.add(new User(2, "Bob", "bob@university.edu"));

        // Configuración del servidor
        Javalin app = Javalin.create(config -> {
            config.router.contextPath = "/";
        }).start(7001);

        System.out.println("Servidor corriendo en http://localhost:7001");
```

```
        // Definición de rutas
        app.post("/users", Main::createUser);
        app.get("/users", Main::getAllUsers);
        app.get("/users/{id}", Main::getUserById);
        app.put("/users/{id}", Main::updateUser);
        app.delete("/users/{id}", Main::deleteUser);
    }
}
```

5.2 Análisis de Componentes

Base de Datos en Memoria

```
private static final List<User> users = new ArrayList<>();
```

- **Volátil**: Los datos se pierden al reiniciar el servidor
- **No apta para producción**: Solo fines didácticos
- **Alternativa real**: PostgreSQL, MySQL, MongoDB con JPA/Hibernate

Singleton de GSON

```
private static final Gson gson = new Gson();
```

- Reutilizable en toda la aplicación
- Evita crear instancias en cada request (mejor rendimiento)

Method References

```
app.post("/users", Main::createUser);
```

En lugar de:

```
app.post("/users", ctx -> Main.createUser(ctx));
```

Ventaja: Código más limpio y funcional (Java 8+)

6. Implementación de Endpoints

6.1 POST `/users` - Crear Usuario

```
private static void createUser(Context ctx) {
    // 1. Deserialización: JSON → Java Object
    User newUser = gson.fromJson(ctx.body(), User.class);

    // 2. Lógica de negocio (generación de ID)
    newUser.id = ThreadLocalRandom.current().nextInt(100, 999);
    users.add(newUser);

    // 3. Respuesta: 201 Created
    ctx.status(HttpStatus.CREATED);
    ctx.json(Map.of(
        "message", "Usuario creado exitosamente",
        "id", newUser.id
    ));
}
```

Análisis Técnico:

1. `ctx.body()`: Obtiene el cuerpo raw de la petición HTTP
2. `gson.fromJson()`: Deserialización mediante reflexión
3. `ThreadLocalRandom`: Generador de números aleatorios thread-safe (Java 7+)
4. `HttpStatus.CREATED`: Constante para código 201 (mejor práctica vs. números mágicos)
5. `Map.of()`: Factory method de Java 9+ para crear mapas inmutables

Códigos de Estado HTTP:

- **201 Created**: Recurso creado exitosamente
- **Location header** (opcional): URI del nuevo recurso creado

6.2 GET `/users` - Listar Todos los Usuarios

```
private static void getAllUsers(Context ctx) {
    ctx.status(HttpStatus.OK);
    ctx.json(users);
}
```

Análisis:

- `ctx.json(users)`: Javalin serializa automáticamente la lista a JSON
- **200 OK**: Operación exitosa con cuerpo de respuesta

Mejoras Productivas:

```
// Paginación
int page = ctx.queryParam("page", Integer.class, "0").get();
int size = ctx.queryParam("size", Integer.class, "10").get();
```

```
// Filtrado
String nameFilter = ctx.queryParam("name");
```

6.3 GET `/users/{id}` - Obtener Usuario por ID

```
private static void getUserById(Context ctx) {
    int id = Integer.parseInt(ctx.pathParam("id"));

    // Búsqueda funcional (Java Stream API)
    Optional<User> match = users.stream()
            .filter(u -> u.id == id)
            .findFirst();

    if (match.isPresent()) {
        ctx.status(HttpStatus.OK);
        ctx.json(match.get());
    } else {
        ctx.status(HttpStatus.NOT_FOUND);
        ctx.json(Map.of("error", "Usuario no encontrado"));
    }
}
```

Análisis de Java Streams:

1. `stream()`: Convierte la colección en un stream procesable
2. `filter()`: Operación intermedia (lazy evaluation)
3. `findFirst()`: Operación terminal que retorna `Optional<User>`

Manejo de Errores:

- **404 Not Found**: Recurso no existe
- Respuesta JSON consistente incluso en errores

Alternativa con Pattern Matching (Java 21+):

```
switch (match) {
    case Optional<User> u when u.isPresent() -> {
        ctx.status(HttpStatus.OK);
        ctx.json(u.get());
    }
    case Optional<User> _ -> {
        ctx.status(HttpStatus.NOT_FOUND);
        ctx.json(Map.of("error", "Usuario no encontrado"));
    }
}
```

6.4 PUT `/users/{id}` - Actualizar Usuario

```
private static void updateUser(Context ctx) {
    int id = Integer.parseInt(ctx.pathParam("id"));
    User incomeData = gson.fromJson(ctx.body(), User.class);

    Optional<User> match = users.stream()
            .filter(u -> u.id == id)
            .findFirst();

    if (match.isPresent()) {
        User user = match.get();
        user.name = incomeData.name;
        user.email = incomeData.email;

        ctx.status(HttpStatus.OK);
        ctx.json(user);
    } else {
        ctx.status(HttpStatus.NOT_FOUND);
        ctx.json(Map.of("error", "Usuario no encontrado para
actualizar"));
    }
}
```

Diferencia PUT vs PATCH:

- **PUT**: Reemplazo completo del recurso (idempotente)
- **PATCH**: Actualización parcial (no necesariamente idempotente)

Este código implementa PUT pero con comportamiento de PATCH (solo actualiza campos enviados).

6.5 DELETE `/users/{id}` - Eliminar Usuario

```
private static void deleteUser(Context ctx) {
    int id = Integer.parseInt(ctx.pathParam("id"));

    boolean removed = users.removeIf(u -> u.id == id);

    if (removed) {
        ctx.status(HttpStatus.OK);
        ctx.json(Map.of("message", "Usuario eliminado"));
    } else {
        ctx.status(HttpStatus.NOT_FOUND);
        ctx.json(Map.of("error", "Usuario no existe"));
    }
}
```

Opciones de Respuesta:

- **200 OK**: Con mensaje de confirmación
- **204 No Content**: Sin cuerpo de respuesta (más RESTful)

```
ctx.status(HttpStatus.NO_CONTENT); // Alternativa preferida
```

7. Verificación Empírica del Sistema

7.1 Iniciar el Servidor

```
# Compilar el proyecto
mvn clean compile

# Ejecutar el servidor
mvn exec:java -Dexec.mainClass="Main"
```

Salida esperada:

```
[Main.main()] INFO io.javalin.Javalin - Starting Javalin ...
[Main.main()] INFO io.javalin.Javalin - Javalin started in 129ms \o/
[Main.main()] INFO io.javalin.Javalin - Listening on
http://localhost:7001/
Servidor corriendo en http://localhost:7001
```

7.2 Pruebas con `curl`

1. Listar Usuarios Iniciales

```
curl -X GET http://localhost:7001/users
```

Respuesta esperada:

```
[
  {
    "id": 1,
    "name": "Alice",
    "email": "alice@university.edu"
  },
  {
    "id": 2,
    "name": "Bob",
    "email": "bob@university.edu"
  }
]
```

2. Crear Nuevo Usuario

```
curl -X POST http://localhost:7001/users \
     -H "Content-Type: application/json" \
     -d '{"name": "PhD Student", "email": "researcher@university.edu"}'
```

Respuesta esperada:

```
{
  "message": "Usuario creado exitosamente",
  "id": 547
}
```

(El ID será aleatorio entre 100-999)

3. Obtener Usuario Específico

```
curl -X GET http://localhost:7001/users/1
```

Respuesta esperada:

```
{
  "id": 1,
  "name": "Alice",
  "email": "alice@university.edu"
}
```

4. Actualizar Usuario

```
curl -X PUT http://localhost:7001/users/1 \
     -H "Content-Type: application/json" \
     -d '{"name": "Alice Johnson", "email": "alice.j@university.edu"}'
```

Respuesta esperada:

```
{
  "id": 1,
  "name": "Alice Johnson",
```

```
  "email": "alice.j@university.edu"
}
```

5. Eliminar Usuario

```
curl -X DELETE http://localhost:7001/users/2
```

Respuesta esperada:

```
{
  "message": "Usuario eliminado"
}
```

6. Manejo de Errores (404)

```
curl -X GET http://localhost:7001/users/999
```

Respuesta esperada:

```
{
  "error": "Usuario no encontrado"
}
```

8. Mejores Prácticas y Extensiones

8.1 Validación de Datos

```
private static void createUser(Context ctx) {
    User newUser = gson.fromJson(ctx.body(), User.class);

    // Validación básica
    if (newUser.name == null || newUser.name.isBlank()) {
        ctx.status(HttpStatus.BAD_REQUEST);
        ctx.json(Map.of("error", "El nombre es obligatorio"));
        return;
    }

    if (newUser.email == null || !newUser.email.matches("^[\\w-
\\.]+@[\\w-]+\\.[a-z]{2,}$")) {
        ctx.status(HttpStatus.BAD_REQUEST);
        ctx.json(Map.of("error", "Email inválido"));
        return;
```

```
    }

    // ... resto del código
}
```

8.2 Manejo Global de Excepciones

```
app.exception(JsonSyntaxException.class, (e, ctx) -> {
    ctx.status(HttpStatus.BAD_REQUEST);
    ctx.json(Map.of("error", "JSON inválido: " + e.getMessage()));
});

app.exception(NumberFormatException.class, (e, ctx) -> {
    ctx.status(HttpStatus.BAD_REQUEST);
    ctx.json(Map.of("error", "ID debe ser un número"));
});
```

8.3 CORS (Cross-Origin Resource Sharing)

Para permitir peticiones desde el frontend:

```
Javalin app = Javalin.create(config -> {
    config.bundledPlugins.enableCors(cors -> {
        cors.addRule(it -> {
            it.anyHost();
        });
    });
}).start(7001);
```

8.4 Logging Personalizado

```
app.before(ctx -> {
    System.out.println("[" + ctx.method() + "] " + ctx.path());
});

app.after(ctx -> {
    System.out.println("Status: " + ctx.status());
});
```

8.5 Conexión a Base de Datos Real

```
// Ejemplo conceptual con JDBC
import java.sql.*;
```

```java
public class UserRepository {
    private Connection conn;

    public List<User> findAll() throws SQLException {
        List<User> users = new ArrayList<>();
        String sql = "SELECT * FROM users";

        try (Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {
                users.add(new User(
                    rs.getInt("id"),
                    rs.getString("name"),
                    rs.getString("email")
                ));
            }
        }
        return users;
    }
}
```

9. Comparación con Spring Boot

Aspecto	Javalin	Spring Boot
Configuración	~50 líneas	~200+ líneas (annotations, configs)
Dependencias	3 básicas	20+ transitivas
Curva de aprendizaje	Baja (1-2 días)	Alta (1-2 semanas)
Tamaño del JAR	~2 MB	~30-50 MB
Tiempo de arranque	~100ms	~2-5s
Casos de uso	Microservicios, APIs ligeras	Aplicaciones empresariales
Ecosistema	Limitado	Muy extenso

10. Conceptos Avanzados

10.1 Códigos de Estado HTTP Completos

Código	Nombre	Uso
200	OK	Petición exitosa con respuesta
201	Created	Recurso creado exitosamente
204	No Content	Éxito sin cuerpo de respuesta
400	Bad Request	Datos inválidos del cliente
401	Unauthorized	Autenticación requerida
403	Forbidden	Sin permisos suficientes
404	Not Found	Recurso no existe
409	Conflict	Conflicto (ej: email duplicado)
500	Internal Server Error	Error del servidor

10.2 Headers HTTP Importantes

```
// Content Negotiation
ctx.header("Content-Type", "application/json; charset=utf-8");

// Cache Control
ctx.header("Cache-Control", "no-cache, no-store, must-revalidate");

// CORS
ctx.header("Access-Control-Allow-Origin", "*");

// Seguridad
ctx.header("X-Content-Type-Options", "nosniff");
ctx.header("X-Frame-Options", "DENY");
```

10.3 Autenticación JWT (Conceptual)

```
app.before("/users/*", ctx -> {
    String token = ctx.header("Authorization");

    if (token == null || !token.startsWith("Bearer ")) {
        ctx.status(HttpStatus.UNAUTHORIZED);
        ctx.json(Map.of("error", "Token requerido"));
        ctx.result(); // Detiene la ejecución
        return;
    }

    // Validar JWT
    String jwt = token.substring(7);
    if (!JWTValidator.isValid(jwt)) {
        ctx.status(HttpStatus.FORBIDDEN);
        ctx.json(Map.of("error", "Token inválido"));
        ctx.result();
    }
});
```

11. Estructura de Proyecto Profesional

```
javalin-rest-tutorial/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   ├── com/example/
│   │   │   │   ├── Main.java              # Punto de entrada
│   │   │   │   ├── controllers/
│   │   │   │   │   └── UserController.java
│   │   │   │   ├── models/
```

```
│   │   │   │   │   └── User.java
│   │   │   │   ├── repositories/
│   │   │   │   │   └── UserRepository.java
│   │   │   │   ├── services/
│   │   │   │   │   └── UserService.java
│   │   │   │   └── utils/
│   │   │   │       └── JsonUtil.java
│   │   └── resources/
│   │       └── application.properties
│   └── test/
│       └── java/
│           └── com/example/
│               └── UserControllerTest.java
├── pom.xml
└── README.md
```

12. Conclusiones

Conceptos Clave Aprendidos:

1. **REST es un estilo arquitectónico**, no un protocolo
2. **HTTP proporciona la semántica** para implementar REST
3. **Javalin ofrece simplicidad** sin sacrificar control
4. **Java moderno** (Streams, Lambdas, Records) mejora la expresividad
5. **Los códigos de estado HTTP** comunican el resultado de la operación

Próximos Pasos:

- Implementar base de datos real (PostgreSQL + JDBC/JPA)
- Añadir validación con Bean Validation (JSR-380)
- Implementar autenticación con JWT
- Escribir tests unitarios con JUnit 5
- Añadir documentación con OpenAPI/Swagger
- Desplegar en contenedor Docker