PRÁCTICA DE DESARROLLO DE APIs REST EN JAVA CON JAVALIN Y SPARK

FUNDAMENTOS DE FRAMEWORKS WEB LIGEROS EN JAVA

¿Qué es un framework web ligero?

Un framework web ligero es una herramienta de desarrollo que proporciona las funcionalidades esenciales para crear aplicaciones web y APIs sin la complejidad ni el overhead de frameworks empresariales como Spring Boot o Java EE. Estos frameworks se caracterizan por:

- **Minimalismo**: código reducido y sin "magia" (*anotaciones excesivas, reflexión innecesaria*)
- **Rápida configuración**: servidor web listo en pocas líneas de código
- **Bajo consumo de recursos**: ideal para microservicios y contenedores
- **Curva de aprendizaje suave**: API intuitiva y directa

Conceptos clave

- **Routing:** mapeo entre URLs y funciones que manejan las peticiones
- **Context/Request/Response:** objetos que encapsulan información HTTP
- **Handler:** función lambda o método que procesa una petición
- **JSON Serialization:** conversión automática entre objetos Java y JSON
- **HTTP Methods:** GET, POST, PUT, DELETE, PATCH para operaciones CRUD

JAVALIN VS SPARK: COMPARATIVA INICIAL

Javalin

- Framework moderno (*primera versión en 2017*)
- Construido sobre Jetty
- Soporte nativo para Kotlin y Java
- Documentación actualizada y activa comunidad
- Sintaxis más expresiva y moderna

Spark

- Framework consolidado (*primera versión en 2011*)
- Inspirado en Sinatra (*Ruby*)
- API estable y probada en producción
- Sintaxis minimalista y directa
- Amplia base de usuarios

JAVALIN - FRAMEWORK WEB MODERNO

Descripción técnica

Javalin es un framework web ligero que se ejecuta sobre Jetty, uno de los servidores web más estables y eficientes del ecosistema Java. A diferencia de otros frameworks, Javalin no utiliza anotaciones ni reflexión, optando por una API funcional basada en lambdas de Java 8+.

Arquitectura interna

Aplicación Javalin → Jetty Server → Sistema Operativo

↓

Context (*Request/Response*)

↓

Handler (*Lambda/Método*)

El flujo de una petición en Javalin:

1. Cliente HTTP realiza petición
2. Jetty recibe y parsea la petición
3. Javalin crea un objeto Context
4. Se ejecuta el handler correspondiente a la ruta
5. El handler modifica el Context (*response*)
6. Javalin serializa la respuesta
7. Jetty envía la respuesta al cliente

Configuración con Maven

Para utilizar Javalin, primero debemos añadir la dependencia en nuestro archivo 'pom.xml':

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>javalin-examples</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <!-- Javalin -->
    <dependency>
      <groupId>io.javalin</groupId>
      <artifactId>javalin</artifactId>
      <version>5.6.3</version>
    </dependency>

    <!-- SLF4J Simple -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
      <version>2.0.9</version>
    </dependency>

    <!-- Jackson para JSON -->
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.15.2</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

```
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
        </plugin>
    </plugins>
  </build>
</project>
```

Métodos principales de Javalin

1. Creación de la aplicación

```
// Aplicación básica
Javalin app = Javalin.create().start(7070);
// Con configuración personalizada
Javalin app = Javalin.create(config -> {
  config.http.asyncTimeout = 10000L;
  config.http.maxRequestSize = 1000000L;
  config.staticFiles.add("/public");
}).start(7070);
```

2. Definición de rutas HTTP

```
// GET - Obtener recurso
app.get("/usuarios", ctx -> {
  ctx.result("Lista de usuarios");
});
// POST - Crear recurso
app.post("/usuarios", ctx -> {
  ctx.result("Usuario creado");
});
// PUT - Actualizar recurso completo
app.put("/usuarios/{id}", ctx -> {
  String id = ctx.pathParam("id");
  ctx.result("Usuario " + id + " actualizado");
});
// DELETE - Eliminar recurso
```

```
app.delete("/usuarios/{id}", ctx -> {
    String id = ctx.pathParam("id");
    ctx.result("Usuario " + id + " eliminado");
});
// PATCH - Actualizar recurso parcialmente
app.patch("/usuarios/{id}", ctx -> {
    String id = ctx.pathParam("id");
    ctx.result("Usuario " + id + " modificado parcialmente");
});
```

3. Manejo del objeto Context

El objeto 'Context' es el núcleo de Javalin, proporcionando acceso a request y response:

```
app.get("/ejemplo", ctx -> {
    // Parámetros de ruta
    String id = ctx.pathParam("id");
    // Parámetros de query (?nombre=Juan&edad=25)
    String nombre = ctx.queryParam("nombre");
    Integer edad = ctx.queryParam("edad", Integer.class);
    // Headers
    String auth = ctx.header("Authorization");
    // Body como String
    String body = ctx.body();
    // Body como objeto (deserialización automática)
    Usuario usuario = ctx.bodyAsClass(Usuario.class);
    // Respuesta como texto
    ctx.result("Respuesta en texto plano");
    // Respuesta como JSON (serialización automática)
    ctx.json(usuario);
    // Código de estado
    ctx.status(201);
    // Headers de respuesta
```

```
    ctx.header("Content-Type", "application/json");
});
```

4. Manejo de JSON

Javalin utiliza Jackson por defecto para serialización/deserialización:

```
// Clase modelo
public class Producto {
    private int id;
    private String nombre;
    private double precio;
    // Constructores, getters y setters
    public Producto() {}
    public Producto(int id, String nombre, double precio) {
        this.id = id;
        this.nombre = nombre;
        this.precio = precio;
    }
    // Getters y setters...
}
// Enviar JSON
app.get("/productos/{id}", ctx -> {
    int id = ctx.pathParam("id", Integer.class);
    Producto producto = new Producto(id, "Laptop", 999.99);
    ctx.json(producto); // Automáticamente serializa a JSON
});
// Recibir JSON
app.post("/productos", ctx -> {
    Producto producto = ctx.bodyAsClass(Producto.class);
    // Procesar el producto...
    ctx.status(201).json(producto);
});
```

Limitaciones importantes

- **Thread safety**: los handlers deben ser thread-safe si comparten estado mutable
- **Dependencia de Jetty**: está acoplado al servidor Jetty embebido
- **Configuración de Jackson**: es global para toda la aplicación
- **Manejo de errores**: requiere configuración explícita de exception handlers

Cuando usar Javalin

- **APIs REST ligeras y microservicios**
- **Aplicaciones donde el rendimiento es crítico**
- **Proyectos educativos para aprender desarrollo web**
- **Prototipado rápido de servicios web**
- **Evitar para**: aplicaciones empresariales complejas que requieren características avanzadas de frameworks como Spring

SPARK - FRAMEWORK WEB MINIMALISTA

Descripción técnica

Spark Framework es un micro framework web para Java inspirado en Sinatra (Ruby). Se centra en proporcionar una API extremadamente simple y expresiva para crear servicios web y APIs REST. Spark también utiliza Jetty como servidor embebido, pero con una sintaxis aún más minimalista que Javalin.

Arquitectura interna

Aplicación Spark → Jetty Server → Sistema Operativo

↓

Request + Response (*parámetros*)

↓

Handler (*Lambda con return*)

El flujo de una petición en Spark:

1. Cliente HTTP realiza petición
2. Jetty recibe y parsea la petición
3. Spark identifica la ruta y ejecuta el handler
4. El handler retorna directamente el contenido de la respuesta
5. Spark envía la respuesta al cliente

Configuración con Maven

Dependencias necesarias en 'pom.xml':

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>spark-examples</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <!-- Spark Framework -->
    <dependency>
      <groupId>com.sparkjava</groupId>
      <artifactId>spark-core</artifactId>
      <version>2.9.4</version>
    </dependency>

    <!-- SLF4J API -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.36</version>
    </dependency>

    <!-- SLF4J Simple Implementation (EL IMPORTANTE) -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
```

```
        <version>1.7.36</version>
    </dependency>

    <!-- Gson para JSON -->
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.10.1</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
        </plugin>
    </plugins>
  </build>
</project>
```

Métodos principales de Spark

1. Configuración inicial

```
import static spark.Spark.*;

public class Main {

  public static void main(String[] args) {

    // Configurar puerto

    port(4567);

    // Configurar archivos estáticos

    staticFileLocation("/public");

    // Configurar número de threads

    threadPool(8);

  }

}
```

2. Definición de rutas HTTP

```
import static spark.Spark.*;

public class Main {
  public static void main(String[] args) {
    // GET - Obtener recurso
    get("/usuarios", (req, res) -> {
      return "Lista de usuarios";
    });
    // POST - Crear recurso
    post("/usuarios", (req, res) -> {
      res.status(201);
      return "Usuario creado";
    });
    // PUT - Actualizar recurso
    put("/usuarios/:id", (req, res) -> {
      String id = req.params(":id");
      return "Usuario " + id + " actualizado";
    });
    // DELETE - Eliminar recurso
    delete("/usuarios/:id", (req, res) -> {
      String id = req.params(":id");
      res.status(204);
      return "";
    });
    // PATCH - Actualizar parcialmente
    patch("/usuarios/:id", (req, res) -> {
      String id = req.params(":id");
      return "Usuario " + id + " modificado";
    });
  }
}
```

3. Manejo de Request y Response

```
get("/ejemplo/:id", (req, res) -> {
  // Parámetros de ruta
  String id = req.params(":id");
  // Parámetros de query (?nombre=Juan)
  String nombre = req.queryParams("nombre");
  // Headers
  String auth = req.headers("Authorization");
  // Body
  String body = req.body();
  // Configurar respuesta
  res.status(200);
  res.type("application/json");
  res.header("Custom-Header", "valor");
  return "Respuesta";
});
```

4. Manejo de JSON con Gson

Spark no tiene serialización JSON integrada, por lo que usamos Gson mediante un ResponseTransformer:

```
import com.google.gson.Gson;
import spark.ResponseTransformer;
public class JsonTransformer implements ResponseTransformer {
  private Gson gson = new Gson();
  @Override
  public String render(Object model) {
    return gson.toJson(model);
  }
}
```

Uso del transformer:

```
import com.google.gson.Gson;
import static spark.Spark.*;
public class Main {
  public static void main(String[] args) {
    Gson gson = new Gson();
    // Usando ResponseTransformer
    get("/productos/:id", (req, res) -> {
      int id = Integer.parseInt(req.params(":id"));
      Producto producto = new Producto(id, "Laptop", 999.99);
      return producto;
    }, new JsonTransformer());
    // O usando método reference de Java 8
    get("/productos", (req, res) -> {
      return new Producto(1, "Mouse", 25.50);
    }, gson::toJson);
    // Recibir JSON (deserialización manual)
    post("/productos", (req, res) -> {
      Producto producto = gson.fromJson(req.body(), Producto.class);
      res.status(201);
      res.type("application/json");
      return producto;
    }, gson::toJson);
  }
}
```

Consideraciones de rendimiento

- **Sin buffer automático**: Spark envía respuestas directamente
- **Thread pool configurable**: ajustar según carga esperada
- **Importaciones estáticas**: mejoran legibilidad pero pueden causar conflictos
- **Gestión manual de JSON**: más control pero más código

Casos de uso específicos

- Microservicios ultra-ligeros
- APIs REST simples con pocas rutas
- Servicios internos de backend
- Prototipos y MVPs rápidos

POSTMAN - PRUEBAS DE APIs

Descripción

Postman es una herramienta esencial para probar, documentar y desarrollar APIs REST. Permite enviar peticiones HTTP de cualquier tipo, inspeccionar respuestas, automatizar tests y colaborar en equipo.

Conceptos fundamentales

- **Request**: petición HTTP con método, URL, headers y body
- **Response**: respuesta del servidor con status, headers y body
- **Collection**: conjunto organizado de requests relacionadas
- **Environment**: conjunto de variables reutilizables
- **Test**: script JavaScript que valida respuestas

Operaciones básicas en Postman

1. GET Request

Method: GET

URL: http://localhost:7070/productos/1

Headers:

Content-Type: application/json

Response:

{

"id": 1,

"nombre": "Laptop",

"precio": 999.99

}

2. POST Request

Method: POST

URL: http://localhost:7070/productos

Headers:

Content-Type: application/json

Body (raw JSON):

{

"nombre": "Teclado Mecánico",

"precio": 89.99

}

Response:

Status: 201 Created

{

"id": 2,

"nombre": "Teclado Mecánico",

"precio": 89.99

}

3. PUT Request

Method: PUT

URL: http://localhost:7070/productos/2

Headers:

Content-Type: application/json

Body (raw JSON):

{

"id": 2,

"nombre": "Teclado Mecánico RGB",

"precio": 129.99

}

4. DELETE Request

Method: DELETE

URL: http://localhost:7070/productos/2

Response:

Status: 204 No Content

Tests automatizados en Postman

Postman permite escribir tests en JavaScript para validar respuestas:

```
// Test: Verificar código de estado
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
// Test: Verificar que la respuesta es JSON
pm.test("Response is JSON", function () {
    pm.response.to.be.json;
});
// Test: Verificar estructura de datos
pm.test("Response has correct structure", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property('id');
    pm.expect(jsonData).to.have.property('nombre');
    pm.expect(jsonData).to.have.property('precio');
});
// Test: Verificar valor específico
pm.test("Product name is correct", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.nombre).to.eql("Laptop");
});
// Test: Verificar tiempo de respuesta
pm.test("Response time is less than 200ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(200);
});
```

Variables de entorno

Las variables permiten reutilizar valores en múltiples requests:

Variables de entorno "Development":

base_url: http://localhost:7070

api_version: v1

Uso en requests:

URL: {{base_url}}/{{api_version}}/productos

Variables de entorno "Production":

base_url: https://api.midominio.com

api_version: v1

Buenas prácticas con Postman

- **Organizar en Collections**: agrupar requests relacionadas
- **Usar variables**: evitar hardcodear URLs y valores
- **Escribir tests**: automatizar validación de respuestas
- **Documentar requests**: añadir descripciones claras
- **Guardar ejemplos**: documentar responses esperadas

MEJORES PRÁCTICAS Y PATRONES

1. Estructura de proyecto recomendada y separación de responsabilidades

- src/main/java/com/example/Main.java
- src/main/java/com/example/models/Producto.java
- src/main/java/com/example/controllers/ProductoController.java
- src/main/java/com/example/services/ProductoService.java
- src/main/java/com/example/util/JsonUtil.java

2. Manejo robusto de excepciones

```
// Javalin
app.exception(IllegalArgumentException.class, (e, ctx) -> {
    ctx.status(400).json(Map.of("error", e.getMessage()));
});
app.exception(Exception.class, (e, ctx) -> {
    ctx.status(500).json(Map.of("error", "Error interno del servidor"));
});
// Spark
exception(IllegalArgumentException.class, (e, req, res) -> {
    res.status(400);
    res.type("application/json");
    res.body("{\"error\": \"" + e.getMessage() + "\"}");
});
exception(Exception.class, (e, req, res) -> {
    res.status(500);
    res.type("application/json");
    res.body("{\"error\": \"Error interno del servidor\"}");
});
```

3. Validación de datos

```
public static void create(Context ctx) {
    Producto producto = ctx.bodyAsClass(Producto.class);
    // Validaciones
    if (producto.getNombre() == null || producto.getNombre().trim().isEmpty()) {
        ctx.status(400).json(Map.of("error", "El nombre es obligatorio"));
        return;
    }
    if (producto.getPrecio() <= 0) {
        ctx.status(400).json(Map.of("error", "El precio debe ser mayor a 0"));
        return;
    }
    Producto creado = service.crear(producto);
    ctx.status(201).json(creado);
}
```

4. CORS (*Cross-Origin Resource Sharing*)

```
// Javalin
app.before(ctx -> {
    ctx.header("Access-Control-Allow-Origin", "*");
    ctx.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS");
    ctx.header("Access-Control-Allow-Headers", "Content-Type, Authorization");
});
app.options("/*", ctx -> {
    ctx.status(200);
});
// Spark
before((req, res) -> {
    res.header("Access-Control-Allow-Origin", "*");
    res.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS");
    res.header("Access-Control-Allow-Headers", "Content-Type, Authorization");
});
options("/*", (req, res) -> {
    return "OK";
});
```

EJEMPLOS DE INTRODUCCIÓN

Ejemplo 1: API REST básica con Javalin

Descripción: Aplicación simple que proporciona un endpoint GET para obtener un saludo. El servidor se inicia en el puerto 7070 y responde con un mensaje de texto plano.

Resultado: Al acceder a 'http://localhost:7070/' se muestra "¡Hola desde Javalin!".

Ejemplo 2: API REST básica con Spark

Descripción: Implementación equivalente usando Spark Framework. Utiliza importaciones estáticas para una sintaxis más concisa.

Resultado: Al acceder a 'http://localhost:4567/' se muestra "¡Hola desde Spark!".

Ejemplo 3: Manejo de parámetros de ruta con Javalin

Descripción: Demostración de cómo capturar parámetros de la URL. El parámetro '{nombre}' se extrae usando 'pathParam()' y se utiliza en la respuesta personalizada.

Resultado: 'http://localhost:7070/saludo/Juan' devuelve "¡Hola, Juan!".

Ejemplo 4: Respuesta JSON con Javalin

Descripción: Ejemplo de serialización automática de objetos Java a JSON. Javalin utiliza Jackson para convertir el objeto 'Map' en una respuesta JSON sin código adicional.

Resultado: 'http://localhost:7070/usuario' devuelve:

```
{
 "id": 1,
 "nombre": "Ana García",
 "edad": 28
}
```

Ejemplo 5: CRUD completo con Spark y Gson

Descripción: API REST completa con operaciones CRUD (*Create, Read, Update, Delete*). Utiliza un 'HashMap' como almacenamiento en memoria y Gson para transformar objetos a JSON. Incluye manejo de códigos de estado HTTP apropiados.

Resultado: API REST funcional con endpoints para gestionar productos. Prueba con Postman:

- GET 'http://localhost:4567/productos' - Lista todos
- POST 'http://localhost:4567/productos' con body JSON - Crea nuevo
- PUT 'http://localhost:4567/productos/1' con body JSON - Actualiza
- DELETE 'http://localhost:4567/productos/1' - Elimina

EJERCICIOS OBLIGATORIOS

Ejercicio 1: API de Gestión de Tareas

Objetivo: Crear una API REST completa para gestionar tareas (*To-Do List*) con Javalin, implementando todas las operaciones CRUD y validaciones básicas.

Firmas de funciones

```
/*
 * Obtiene todas las tareas
 * @param ctx contexto de Javalin con request/response
 */
public static void obtenerTodas(Context ctx)
/*
 * Obtiene una tarea por ID
 * @param ctx contexto de Javalin con parámetro {id}
 */
public static void obtenerPorId(Context ctx)
/*
 * Crea una nueva tarea
 * @param ctx contexto de Javalin con body JSON
 */
public static void crear(Context ctx)
/*
 * Actualiza una tarea existente
 * @param ctx contexto de Javalin con parámetro {id} y body JSON
 */
public static void actualizar(Context ctx)
/*
 * Elimina una tarea
 * @param ctx contexto de Javalin con parámetro {id}
 */
public static void eliminar(Context ctx)
```

```
/*
 * Marca una tarea como completada
 * @param ctx contexto de Javalin con parámetro {id}
 */
public static void marcarCompletada(Context ctx)
```

Casos de uso

Crear tarea (*POST*)

POST http://localhost:7070/tareas

Body:

```
{
  "titulo": "Estudiar APIs REST",
  "descripcion": "Completar ejercicios de Javalin y Spark"
}
```

Response (*201*):

```
{
  "id": 1,
  "titulo": "Estudiar APIs REST",
  "descripcion": "Completar ejercicios de Javalin y Spark",
  "completada": false,
  "fechaCreacion": "2025-11-24T16:30:00"
}
```

Obtener todas (*GET*)

GET http://localhost:7070/tareas

Response (200):

```
[
 {
  "id": 1,
  "titulo": "Estudiar APIs REST",
  "descripcion": "Completar ejercicios de Javalin y Spark",
  "completada": false,
  "fechaCreacion": "2025-11-24T16:30:00"
 }
]
```

Marcar completada (*PATCH*)

PATCH http://localhost:7070/tareas/1/completar

Response (*200*):

```
{
 "id": 1,
 "titulo": "Estudiar APIs REST",
 "descripcion": "Completar ejercicios de Javalin y Spark",
 "completada": true,
 "fechaCreacion": "2025-11-24T16:30:00"
}
```

Buenas prácticas

- Validar que el título no esté vacío
- Retornar 404 si la tarea no existe
- Usar códigos de estado HTTP apropiados (*200, 201, 204, 404, 400*)
- Separar lógica en controlador y servicio
- Manejar excepciones con handlers globales

Ver archivos para solución:

- 'Tarea.java' (*modelo*)
- 'TareaService.java' (*lógica de negocio*)
- 'TareaController.java' (*controlador*)
- 'MainEjercicio1.java' (*punto de entrada*)

Ejercicio 2: API de Biblioteca con Spark

Objetivo: Implementar una API REST para gestionar una biblioteca de libros usando Spark Framework y Gson, con filtrado y búsqueda.

Firmas de funciones

```
/*
 * Registra todas las rutas de la API
 */
public static void configurarRutas()
/*
 * Obtiene todos los libros o filtra por autor
 * @param req request de Spark
 * @param res response de Spark
 * @return lista de libros en JSON
 */
public static Object obtenerLibros(Request req, Response res)
/*
 * Obtiene un libro específico por ISBN
 * @param req request de Spark con parámetro :isbn
```

```
 * @param res response de Spark
 * @return libro en JSON o error 404
 */
public static Object obtenerLibroPorIsbn(Request req, Response res)
/*
 * Crea un nuevo libro
 * @param req request de Spark con body JSON
 * @param res response de Spark
 * @return libro creado en JSON
 */
public static Object crearLibro(Request req, Response res)
/*
 * Busca libros por título (búsqueda parcial)
 * @param req request de Spark con query param ?q=
 * @param res response de Spark
 * @return lista de libros que coinciden
 */
public static Object buscarLibros(Request req, Response res)
```

Casos de uso

Crear libro

POST http://localhost:4567/libros

Body:

```
{
  "isbn": "978-0134685991",
  "titulo": "Effective Java",
  "autor": "Joshua Bloch",
  "añoPublicacion": 2018
}
```

Response (*201*):

```
{
 "isbn": "978-0134685991",
 "titulo": "Effective Java",
 "autor": "Joshua Bloch",
 "añoPublicacion": 2018,
 "disponible": true
}
```

Buscar libros

GET http://localhost:4567/libros/buscar?q=Java

Response (*200*):

```
[
 {
  "isbn": "978-0134685991",
  "titulo": "Effective Java",
  "autor": "Joshua Bloch",
  "añoPublicacion": 2018,
  "disponible": true
 }
]
```

Filtrar por autor

GET http://localhost:4567/libros?autor=Joshua Bloch

Response (*200*):

```
[
 {
  "isbn": "978-0134685991",
  "titulo": "Effective Java",
  "autor": "Joshua Bloch",
  "añoPublicacion": 2018,
  "disponible": true
 }
]
```

Buenas prácticas

- Usar ISBN como identificador único
- Implementar búsqueda case-insensitive
- Validar formato de ISBN básico
- Usar HashMap para almacenamiento en memoria
- Aplicar ResponseTransformer para todas las respuestas JSON

Ver archivos para solución:

- 'Libro.java' (*modelo*)
- 'JsonTransformer.java' (*transformer*)
- 'BibliotecaController.java' (*controlador*)
- 'MainEjercicio2.java' (*punto de entrada*)

Ejercicio 3: API de Autenticación Básica

Objetivo: Crear un sistema básico de autenticación con registro de usuarios, login y endpoints protegidos usando Javalin.

Firmas de funciones

```
/*
 * Registra un nuevo usuario
 * @param ctx contexto con body {username, password, email}
 */
public static void registrar(Context ctx)

/*
 * Realiza login y genera token simple
 * @param ctx contexto con body {username, password}
 */
public static void login(Context ctx)

/*
 * Obtiene perfil del usuario autenticado
 * @param ctx contexto con header Authorization
 */
public static void obtenerPerfil(Context ctx)

/*
 * Valida token de autenticación
 * @param token token a validar
 * @return username del usuario o null si inválido
 */
public static String validarToken(String token)

/*
 * Handler que verifica autenticación antes de ejecutar endpoint
 * @param ctx contexto de Javalin
 */
public static void verificarAutenticacion(Context ctx)
```

Casos de uso

Registro

POST http://localhost:7070/auth/registrar

Body:

```
{
 "username": "usuario1",
 "password": "mipassword123",
 "email": "usuario1@example.com"
}
```

Response (*201*):

```
{
 "mensaje": "Usuario registrado exitosamente",
 "username": "usuario1"
}
```

Login

POST http://localhost:7070/auth/login

Body:

```
{
 "username": "usuario1",
 "password": "mipassword123"
}
```

Response (*200*):

```
{
 "token": "usuario1_1732462800000",
 "username": "usuario1"
}
```

Acceso a endpoint protegido

GET http://localhost:7070/perfil

Headers:

Authorization: usuario1_1732462800000

Response (*200*):

```
{
 "username": "usuario1",
 "email": "usuario1@example.com",
 "fechaRegistro": "2025-11-24T16:30:00"
}
```

Acceso sin token

GET http://localhost:7070/perfil

Response (*401*):

```
{
 "error": "No autorizado. Token requerido"
}
```

Buenas prácticas

- Usar 'before()' handler para verificar autenticación
- Generar tokens simples (username + timestamp en este ejercicio)
- Validar que username no esté duplicado
- NO almacenar contraseñas en texto plano (nota educativa)
- Retornar errores claros (401 Unauthorized, 403 Forbidden)

EJERCICIOS OPCIONALES

Ejercicio Opcional 1: API de Blog con Comentarios

Objetivo: Crear una API REST de blog con posts y comentarios anidados, utilizando Javalin y relaciones entre entidades.

Ejemplo de uso

POST http://localhost:7070/posts

Body:

```
{
 "titulo": "Aprendiendo APIs REST",
 "contenido": "Las APIs REST son fundamentales...",
 "autor": "Juan Pérez"
}
```

Response (*201*):

```
{
 "id": 1,
 "titulo": "Aprendiendo APIs REST",
 "contenido": "Las APIs REST son fundamentales...",
 "autor": "Juan Pérez",
 "fechaPublicacion": "2025-11-24T16:30:00",
 "comentarios": []
}
```

POST http://localhost:7070/posts/1/comentarios

Body:

```
{
 "autor": "Ana García",
 "contenido": "Excelente post!"
}
```

Response (*201*):

```
{
 "id": 1,
 "autor": "Ana García",
 "contenido": "Excelente post!",
 "fecha": "2025-11-24T16:35:00"
}
```

Resultado esperado: API funcional con endpoints para posts y comentarios, demostrando relaciones uno-a-muchos y serialización JSON de estructuras complejas.

Ejercicio Opcional 2: API de Reservas con Validación de Fechas

Objetivo: Implementar sistema de reservas con Spark que valide disponibilidad de fechas y gestione conflictos.

Ejemplo de uso

POST http://localhost:4567/reservas

Body:

```
{
 "recurso": "Sala de Reuniones A",
 "fecha": "2025-12-01",
 "horaInicio": "10:00",
 "horaFin": "12:00",
 "nombreUsuario": "María López"
}
```

Response (*201*):


```
{
 "id": 1,
 "recurso": "Sala de Reuniones A",
 "fecha": "2025-12-01",
 "horaInicio": "10:00",
 "horaFin": "12:00",
 "nombreUsuario": "María López",
 "estado": "CONFIRMADA"
}
```


POST http://localhost:4567/reservas (conflicto)

Body:


```
{
 "recurso": "Sala de Reuniones A",
 "fecha": "2025-12-01",
 "horaInicio": "11:00",
 "horaFin": "13:00",
 "nombreUsuario": "Pedro Ruiz"
}
```


Response (409):


```
{
 "error": "Conflicto de horario",
 "detalle": "La sala ya está reservada de 10:00 a 12:00"
}
```


Resultado esperado: Sistema que detecta solapamiento de reservas, valida fechas futuras y maneja códigos de estado HTTP apropiados (409 Conflict).

Ejercicio Opcional 3: API de Estadísticas en Tiempo Real

Objetivo: Crear endpoints que calculen estadísticas dinámicas sobre datos almacenados, usando Javalin con agregaciones.

Ejemplo de uso

POST http://localhost:7070/ventas

Body:

```
{
 "producto": "Laptop",
 "cantidad": 2,
 "precioUnitario": 999.99
}
```

GET http://localhost:7070/estadisticas?fecha_inicio=2025-11-01&fecha_fin=2025-11-30

Response (200):

```
{
 "totalVentas": 15998.50,
 "numeroTransacciones": 25,
 "productoMasVendido": "Laptop",
 "ventaPromedio": 639.94
}
```

Resultado esperado: API que procesa datos en memoria, calcula métricas agregadas y responde con estadísticas formateadas en JSON.

FORMATO DE ENTREGA

Requisitos del código

- **JavaDoc**: todas las clases y métodos públicos deben tener documentación JavaDoc
- **Comentarios**: lógica compleja debe estar comentada
- **Nombres descriptivos**: variables y métodos con nombres claros en español
- **Manejo de excepciones**: usar exception handlers globales de Javalin/Spark
- **Códigos HTTP**: usar códigos de estado apropiados (*200, 201, 204, 400, 404, 500*)
- **Separación de responsabilidades**: controlador, servicio, modelo en clases separadas
- **Formato JSON**: todas las respuestas deben ser JSON válido

Compilación y ejecución

Compilar el proyecto

```
mvn clean package
```

Ejecutar la aplicación

```
java -jar target/nombre-proyecto-1.0-SNAPSHOT.jar
```

O ejecutar directamente con Maven

```
mvn exec:java -Dexec.mainClass="com.example.Main"
```

Criterios de evaluación

1. **Funcionalidad (*40%*)**: La API cumple todos los requisitos
2. **Código limpio (*25%*)**: Organización, nombres, comentarios
3. **Buenas prácticas (*20%*)**: Manejo errores, validaciones, códigos HTTP
4. **Documentación (*15%*)**: JavaDoc, README, comentarios