Ingeniería de Pruebas para APIs REST con JUnit 5 y Javalin

1. Introducción y Estrategia de Testing

En el desarrollo de software profesional, la verificación del comportamiento de una API REST no se realiza manualmente (via Postman o cURL), sino mediante **pruebas automatizadas**. Dado que Javalin es un framework ligero ("un-opinionated"), la estrategia más efectiva para asegurar la calidad no es el *unit testing* aislado de métodos (mockear el objeto Context), sino las **Pruebas de Integración de Componentes**.
Esta estrategia consiste en levantar una instancia real (pero efímera) del servidor Javalin durante la ejecución de los tests, permitiendo realizar peticiones HTTP reales contra los endpoints. Esto garantiza que:

1. El enrutamiento (routing) es correcto.
2. La serialización/deserialización (JSON <-> Java Object) funciona.
3. Los códigos de estado HTTP son los adecuados.

A diferencia de frameworks pesados (como Spring Boot), Javalin arranca en milisegundos, lo que hace viable esta estrategia sin penalizar el tiempo de ejecución del pipeline de CI/CD.

2. Configuración del Entorno (Maven)

Para implementar un ciclo de pruebas riguroso, necesitamos añadir bibliotecas especializadas en el pom.xml. No basta con JUnit; necesitamos herramientas para realizar aserciones semánticas y un cliente HTTP de prueba.

2.1 Dependencias Requeridas

Añada el siguiente bloque a la sección <dependencies> de su pom.xml.

```
<!-- JUnit 5: El motor de ejecución de pruebas -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.1</version>
    <scope>test</scope>
</dependency>

<!-- AssertJ: Librería para aserciones fluidas y legibles -->
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.25.1</version>
    <scope>test</scope>
```

```
</dependency>

<!-- Unirest: Cliente HTTP minimalista para simular al consumidor de la API -->
<dependency>
  <groupId>com.konghq</groupId>
  <artifactId>unirest-java-core</artifactId>
  <version>4.2.9</version>
  <scope>test</scope>
</dependency>

<!-- Unirest Object Mapping (Gson): Para que Unirest entienda nuestros objetos -->
<dependency>
  <groupId>com.konghq</groupId>
  <artifactId>unirest-objectmapper-gson</artifactId>
  <version>4.2.9</version>
  <scope>test</scope>
</dependency>
```

Justificación técnica:

- **JUnit 5 (Jupiter):** Estándar actual. Permite control granular del ciclo de vida.
- **AssertJ:** Permite escribir assertThat(x).isEqualTo(y) en lugar del críptico assertEquals(y, x). Mejora drásticamente la legibilidad ante fallos.
- **Unirest:** Simplifica la sintaxis de las peticiones HTTP en los tests, evitando el boilerplate de java.net.http.HttpClient para pruebas rápidas.

3. Ciclo de Vida de la Prueba (Lifecycle)

El concepto crítico en pruebas de integración es el manejo del estado. Debemos garantizar el **aislamiento** y el **determinismo**.

3.1 Anotaciones Clave de JUnit 5

Anotación	**Propósito**	**Equivalente Legacy (JUnit 4)**	**Uso en Javalin**
@BeforeAll	Ejecutar una vez antes de todos los tests de la clase.	setUpClass	**Arrancar el servidor.**
@AfterAll	Ejecutar una vez al finalizar todos los tests.	tearDownClass	**Detener el servidor.**
@BeforeEach	Ejecutar antes de *cada* método de prueba individual.	setUp	**Limpiar la base de datos/lista.**
@AfterEach	Ejecutar después de	tearDown	Limpieza específica

	cada método.		(raro).

3.2 El problema del puerto

Nunca se debe hardcodear el puerto (ej. 7001) en los tests. Si el puerto está ocupado (por ejemplo, si Jenkins está corriendo dos tests en paralelo), la prueba fallará.

- **Solución:** Configurar Javalin para usar el puerto 0. Esto instruye al sistema operativo a asignar un puerto aleatorio disponible.

4. Implementación de la Suite de Pruebas

A continuación, se presenta la implementación de la clase UserIntegrationTest.java. Esta clase debe ubicarse en src/test/java/.

4.1 Preparación (Refactorización necesaria en Main)

Para que la aplicación sea comprobable, debemos poder acceder a la instancia de Javalin o, al menos, controlar su ciclo de vida y el acceso a los datos.
Asumiremos que en Main.java hemos refactorizado la lista de usuarios para que sea accesible (p.ej. mediante un getter estático o movida a un Repository separado) y que el método de inicio devuelve la instancia de Javalin.
Nota: Para este ejemplo, asumiremos que se ha añadido un método Main.stop() y un método Main.cleanData() en su clase principal para facilitar el testing, o que estamos instanciando la app directamente en el test. Lo más limpio es instanciar la app dentro del test.

4.2 Código Completo del Test

```
import io.javalin.Javalin;
import io.javalin.http.HttpStatus;
import kong.unirest.core.HttpResponse;
import kong.unirest.core.Unirest;
import org.junit.jupiter.api.*;

import java.util.List;

// Import estático para aserciones fluidas
import static org.assertj.core.api.Assertions.assertThat;

@DisplayName("Pruebas de Integración - Recurso Usuarios")
class UserIntegrationTest {

    private static Javalin app;
    private static String BASE_URL;

    // ========================================
```

```
// CICLO DE VIDA (Lifecycle)
// ========================================

@BeforeAll
static void setUpAll() {
    // 1. Iniciamos la aplicación.
    // Nota: En un entorno real, la lógica de creación de 'app' debería estar
    // en un método separado de Main para ser reutilizada aquí sin duplicar código.
    app = Javalin.create(/* config igual que en Main */)
            .post("/users", Main::createUser)
            .get("/users", Main::getAllUsers)
            .get("/users/{id}", Main::getUserById)
            .put("/users/{id}", Main::updateUser)
            .delete("/users/{id}", Main::deleteUser)
            .start(0); // Puerto 0 = Puerto aleatorio

    // 2. Capturamos el puerto real asignado por el SO
    int randomPort = app.port();
    BASE_URL = "http://localhost:" + randomPort + "/users";

    // 3. Configurar Unirest (opcional, para serialización automática)
    // Por simplicidad usaremos Strings o Mapeo manual en este ejemplo.
}

@AfterAll
static void tearDownAll() {
    // Detenemos el servidor para liberar el puerto y recursos
    if (app != null) {
        app.stop();
    }
}

@BeforeEach
void setUp() {
    // CRÍTICO: Limpiar el estado antes de CADA test.
    // Esto garantiza la independencia de las pruebas (Idempotencia).
    // Debemos exponer un método en Main para limpiar la lista o acceder a ella.
    Main.users.clear();

    // Añadimos un dato semilla (seed data) para pruebas de lectura
    Main.users.add(new Main.User(1, "Seed User", "seed@test.com"));
}
```

```
// ========================================
// CASOS DE PRUEBA (Test Cases)
// ========================================

@Test
@DisplayName("GET /users - Debe retornar todos los usuarios existentes")
void shouldReturnAllUsers() {
    // Arrange (Preparación ya hecha en setUp)

    // Act (Acción)
    HttpResponse<String> response = Unirest.get(BASE_URL).asString();

    // Assert (Verificación)
    assertThat(response.getStatus()).isEqualTo(HttpStatus.OK.getCode());
    assertThat(response.getBody()).contains("Seed User");
    assertThat(response.getBody()).contains("seed@test.com");
}

@Test
@DisplayName("POST /users - Debe crear un usuario y retornar 201 Created")
void shouldCreateUser() {
    // Arrange
    String newUserJson = """
        {
            "name": "Integration Test",
            "email": "integration@test.com"
        }
        """;

    // Act
    HttpResponse<String> response = Unirest.post(BASE_URL)
            .header("Content-Type", "application/json")
            .body(newUserJson)
            .asString();

    // Assert
    assertThat(response.getStatus()).isEqualTo(HttpStatus.CREATED.getCode());
    assertThat(response.getBody()).contains("Integration Test");

    // Verificación adicional: ¿Realmente se guardó en memoria?
    assertThat(Main.users).hasSize(2); // 1 semilla + 1 nuevo
}
```

```
    @Test
    @DisplayName("GET /users/{id} - Debe retornar 404 si el usuario no existe")
    void shouldReturn404ForNonExistentUser() {
        // Act
        HttpResponse<String> response = Unirest.get(BASE_URL + "/9999").asString();

        // Assert
        assertThat(response.getStatus()).isEqualTo(HttpStatus.NOT_FOUND.getCode());
        assertThat(response.getBody()).contains("Usuario no encontrado");
    }

    @Test
    @DisplayName("DELETE /users/{id} - Debe eliminar un usuario existente")
    void shouldDeleteUser() {
        // Act
        HttpResponse<String> response = Unirest.delete(BASE_URL + "/1").asString();

        // Assert
        assertThat(response.getStatus()).isEqualTo(HttpStatus.OK.getCode());

        // Verificar efecto colateral (Side Effect)
        assertThat(Main.users).isEmpty();
    }
}
```

5. Análisis de Patrones Utilizados

5.1 Patrón AAA (Arrange-Act-Assert)

Cada método de prueba debe estructurarse claramente en tres bloques:

1. **Arrange (Organizar):** Preparar los datos (@BeforeEach o al inicio del método). Ejemplo: Definir el JSON a enviar.
2. **Act (Actuar):** Ejecutar la acción contra el sistema bajo prueba. Ejemplo: Unirest.post(...).
3. **Assert (Aseverar):** Validar que el resultado es el esperado. Ejemplo: assertThat(...).

5.2 Determinismo y "Flakiness"

Un test es "flaky" si a veces pasa y a veces falla sin cambios en el código. La causa número uno es la **contaminación de estado**.

- **Incorrecto:** Crear un usuario en el Test A y esperar que exista en el Test B.
- **Correcto:** Usar @BeforeEach para limpiar la lista Main.users.clear(). Cada test asume que empieza desde un estado conocido (tabula rasa).

5.3 Pruebas de Caja Negra (Black Box)

Observe que en las aserciones validamos response.getStatus() y response.getBody(). Estamos probando la API **desde fuera**, tal como lo haría un cliente real (Frontend o Microservicio). Esto es superior a probar el método Main.createUser aisladamente porque valida toda la cadena HTTP.

6. Ejecución de las Pruebas

Para ejecutar las pruebas desde la terminal, utilizaremos Maven Surefire Plugin (integrado por defecto).

```
# Ejecutar solo los tests
mvn test

# Si desea ver la salida detallada en consola
mvn test -Dsurefire.useFile=false
```

Una ejecución exitosa mostrará:

```
[INFO] Running UserIntegrationTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.542 s - in
UserIntegrationTest
[INFO] BUILD SUCCESS
```