**PRUEBA TÉCNICA BACKEND JAVA**

Hola, nos emociona saber que te interesa ser parte de nuestro equipo.

Agradecemos sinceramente tu tiempo y disposición para participar.

¡Éxitos en el desarrollo de la prueba!

**Nombre completo:**

*(Por favor actualiza también el nombre del archivo con tu nombre)*

**Fecha en la que se realiza la prueba**:

Esta prueba está diseñada para completarse en aproximadamente 2 horas. Se concentra en los conocimientos y habilidades obligatorios del rol.

**CUESTIONARIO TEÓRICO**

**(20 minutos)**

Responde brevemente (100–280 párrafos cada uno):

1. Arquitectura de Microservicios / APIs Rest
   * ¿Qué ventajas aporta una arquitectura basada en microservicios frente a una aplicación monolítica?
   * Explica brevemente qué es OpenAPI/Swagger y para qué se utiliza en la construcción de APIs.
2. Java & Spring Framework
   * Enumera tres diferencias clave entre Java 8 y Java 11 (por ejemplo: características del lenguaje, cambios en el garbage collector, módulos, etc.).
   * Explica en qué consiste Spring Boot y nombra dos de sus principales ventajas para desarrollar servicios backend.
3. Testing en Java (JUnit, Mockito)
   * ¿Por qué es importante usar Mockito (o similar) cuando pruebas componentes de Spring? Da un ejemplo rápido de cómo “mockear” un repositorio JPA para probar un servicio.
   * ¿Qué es MockMVC y cómo ayuda a probar controladores REST en Spring?
4. Prometheus & Grafana
   * ¿Cuál es el flujo básico para exponer métricas de una aplicación Spring Boot a Prometheus y luego visualizarlas en Grafana?
   * Menciona dos métricas comunes que convendría instrumentar en un microservicio (por ejemplo: latencia de petición, contadores de errores, uso de memoria, etc.).
5. Liquibase/Flyway
   * Describe brevemente cómo funciona Liquibase (o Flyway) para gestionar migraciones de base de datos.
   * ¿Qué sucede si varias instancias del mismo microservicio intentan aplicar la misma migración al mismo tiempo? ¿Cómo lo evita Liquibase/Flyway?
6. GIT y Metodologías Ágiles
   * Explica los pasos para crear una rama feature/nueva desde develop, hacer commits mientras trabajas y luego rebasar esos cambios sobre develop antes de enviar el Pull Request.
   * En Scrum, ¿qué rol o ceremonia encajaría mejor para revisar la arquitectura de un nuevo microservicio antes de implementarlo? (menciona al menos uno).

**EJERCICIO PRÁCTICO EN JAVA/SPRING**

**(60 minutos)**

Servicio REST de Gestión de Usuarios

1. Objetivo Funcional:  
   Crea un pequeño microservicio en Spring Boot que exponga un API REST para gestionar usuarios. Debe permitir:
   * Crear un usuario (POST /users), con campos:

{

"username": "string",

"email": "string",

"active": boolean

}

Listar todos los usuarios (GET /users).

Obtener un usuario por ID (GET /users/{id}).

Eliminar un usuario (DELETE /users/{id}).

No se requiere editar (PUT/PATCH), solo las operaciones anteriores.

1. Enfoque técnico:
   * Proyecto Java 11 (o 8):
     + Crea un proyecto Maven con dependencias de Spring Boot (spring-boot-starter-web, spring-boot-starter-data-jpa, h2 como base en memoria para simplificar).
   * Definir una entidad JPA User con campos:

@Entity

public class User {

@Id @GeneratedValue

private Long id;

private String username;

private String email;

private boolean active;

// getters/setters...

}

* + Usar un repositorio JPA (UserRepository extends JpaRepository<User, Long>) para persistir en H2 (en memoria).
  + Crear un servicio (UserService) que encapsule la lógica de negocio (p.ej., saveUser, getAllUsers, getUserById, deleteUser).
  + Implementar un controlador REST (UserController) con los endpoints mencionados.
  + Configurar en application.properties H2 con consola habilitada (spring.h2.console.enabled=true) y base de datos en memoria.
  + Incluir un application.yml o application.properties sencillo que defina el puerto (server.port=8080 por defecto) y la URL de H2.

1. Estructura de carpetas (Maven):

backend-user-service/

├─ src/

│ ├─ main/

│ │ ├─ java/com/ditech/backend/

│ │ │ ├─ model/

│ │ │ │ └─ User.java

│ │ │ ├─ repository/

│ │ │ │ └─ UserRepository.java

│ │ │ ├─ service/

│ │ │ │ └─ UserService.java

│ │ │ └─ controller/

│ │ │ └─ UserController.java

│ │ └─ resources/

│ │ ├─ application.properties

│ │ └─ data.sql (opcional: insertar datos iniciales)

│ └─ test/

│ └─ java/com/ditech/backend/

│ ├─ service/

│ │ └─ UserServiceTest.java

│ └─ controller/

│ └─ UserControllerTest.java

├─ pom.xml

└─ README.md

1. Instrucciones de Entrega:
   * Debe bastar con ejecutar mvn spring-boot:run para levantar la aplicación en http://localhost:8080.
   * Probar manualmente los endpoints con curl o Postman.
   * Incorporar comentarios breves en el código (modelos, servicio, controlador) explicando la responsabilidad de cada clase/método.
   * Documentar en el README.md los pasos para ejecutar, probar y detener la aplicación.

**TESTS UNITARIOS Y DE INTEGRACIÓN**

**(30 minutos)**

A partir del microservicio de gestión de usuarios:

1. Configurar Tests Unitarios en JUnit + Mockito:
   * Crea un archivo UserServiceTest.java donde:
     + “Mockees” (con @Mock) el UserRepository.
   * Escribe dos tests mínimos para UserService:
     + 1. saveUser(): al pasar un objeto User con username y email, verificar que repository.save(...) se invoca una vez y retorna el objeto con ID asignado.
       2. getUserById(): cuando el repositorio devuelve un Optional.empty(), el servicio lanza una excepción (p. ej., EntityNotFoundException).
   * Usa @ExtendWith(MockitoExtension.class) y @InjectMocks para inyectar el servicio con el repositorio “mockeado”.
2. Test de Integración de Controlador (MockMVC):
   * En UserControllerTest.java, configura un test con @WebMvcTest(UserController.class) y @MockBean para el UserService.
   * Escribe dos tests básicos (usando MockMVC) para /users:

GET /users: cuando userService.getAllUsers() retorna una lista con un usuario, el controlador responde estado HTTP 200 y JSON con la lista.

POST /users: cuando envías un JSON válido, el controlador devuelve HTTP 201 con el usuario creado (simulando userService.saveUser(...)).

* + Usa MockMvcBuilders o la configuración estándar de Spring para inyectar MockMVC y verificar respuestas JSON.

Ejecuta estos tests basta con lanzar mvn test. Asegúrate de que pasen sin errores.

**PRUEBA FUNCIONAL**

**(10 minutos)**

Objetivo: Verificar manualmente que tu microservicio arranca y responde correctamente.

1. Levanta la aplicación con mvn spring-boot:run.

Con curl (o HTTPie), ejecuta:

⧫ Crear usuario

curl -i -X POST http://localhost:8080/users \

-H "Content-Type: application/json" \

-d '{"username":"juan","email":"juan@ditech.es","active":true}'

⧫ Listar usuarios

curl -i http://localhost:8080/users

⧫ Obtener usuario por ID

curl -i http://localhost:8080/users/1

⧫ Eliminar usuario

curl -i -X DELETE http://localhost:8080/users/1

Verifica que los códigos de estado (201, 200, 204/200) y las respuestas JSON sean los esperados.

Si todo funciona, pasa al siguiente punto.

**MONITORING BÁSICO**

**(10 minutos)**

Añade un endpoint de métricas para Prometheus:

**Dependencia**: en pom.xml, agrega:

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-registry-prometheus</artifactId>

</dependency>

**Configuración mínima**: en application.properties:

management.metrics.export.prometheus.enabled=true

management.endpoints.web.exposure.include=prometheus

management.endpoint.prometheus.enabled=true

1. Levanta de nuevo la app y accede a http://localhost:8080/actuator/prometheus.
2. Verifica que se expongan métricas básicas de JVM y HTTP.

No es necesario configurar Grafana; basta con comprobar que el endpoint responde texto con métricas Prometheus (p.ej., jvm\_memory\_used\_bytes).