**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:**

*Jorge Alejandro Aguirre Gutierrez*

**Fecha en la que se realiza la prueba**: 08 de Julio del 2025

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?

Son varias ventajas que puede aportar la arquitectura de microservicios, sin embargo todo depende de las características del proyecto.

Algunas ventajas pueden ser:

* Escalamiento vertical, donde también es posible aplicar escalamiento horizontal
* Despliegues mas rápidos: Ya que permite hacer cambios en distintas aplicaciones que tienen un tamaño mas reducido.
* Es posible tener un ecosistema de arquitectura de microservicios desarrollados en diferentes lenguajes de programación
* Tolerancia a fallos: ya que a pesar de que surja un fallo en alguno de los microservicios, los demás por el uso de la escalabilidad puede atender las peticiones entrantes.
* Tambien es posible aplicar mejores mantenimientos a largo plazo, ya que al tener aplicativos independientes, con un tamaño mas reducido la mantenibilidad es mas flexible.
  + Explica brevemente qué es OpenAPI/Swagger y para qué se utiliza en la construcción de APIs.

Básicamente se usa para describir y documentar las funcionalidades de una API, actualmente es posible generar automáticamente una documentación con referencia al código desarrollado haciendo uso de la librería swagger, Swagger facilita la interpretación de las funcionalidades con terceros.

1. 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.).

* En Java 8 hace uso de Parallel GC por defecto, mientras que en Java 11hace uso de GC G1 que mejora el uso de memoria.
* Java 11 añadio nuevos métodos para el manejo de Strings como isBlank(), lines(), strip().
* Java 11 elimino en definitiva paquetes obsoletos como java.se.ee, webstart y Applet.
  + Explica en qué consiste Spring Boot y nombra dos de sus principales ventajas para desarrollar servicios backend.

Spring Boot es un framework para Java que permite tener las siguientes ventajas y/o herramientas:

* Desarrollo rápido gracias a una pre configuración.
* Un servidor embebido como lo es Tomcat.
* Es posible tener una estructura mínima rápidamente para proceder a la ejecución de la API

1. 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.

Con la librería mockito en Spring es posible lo siguiente:

* Es posible probar las funcionalidades del Backend sin necesidad de hacer uso de la base de datos ni de servicios de terceros.
* Es posible simular comportamientos exitosos o errores haciendo uso de algo llamado “Mock”
* Al no hacer implementaciones de pruebas reales no hay perdida de la integración de los datos en el aplicativo.

Algunos pasos generales para hacer uso de Mockito son los siguientes:

* Hacer uso del @Mock para los repositorios de datos.
* Hacer uso de @InjectMock si queremos simular el comportamiento de un service o de un use case.
* Configurar al mock para devolver un resultado ficticio.
  + ¿Qué es MockMVC y cómo ayuda a probar controladores REST en Spring?

MockMVC básicamente es una herramienta que nos permite:

* Simular peticiones GET, POST, PUT, DELETE, entre otras.
* Es possible verificar la respuesta a la petición simulada, como también ver el código de estado, y los headers.
* Tambien como se hace una prueba desde el controlador es posible medir si los parámetros de entrada se están recibiendo correctamente.

1. Prometheus & Grafana
   * ¿Cuál es el flujo básico para exponer métricas de una aplicación Spring Boot a Prometheus y luego visualizarlas en Grafana?

Los pasos básicos para tener las métricas de prometheus y poderlas visualizar en grafana son:

* Agregar la dependencia actuator y la dependencia de prometheus en Maven o en gradle según corresponda.
* Agregar las configuraciones del endpoint de prometheus para las métricas básicas, también es posible agregar métricas mas customizadas.
* Instalar prometheus y modificar el prometheus.yml para que empiece a escuchar las métricas.
* Instalar grafana crear un datasource que apunte hacia nuestro aplicativo con prometheus, crear un dashboard para visualizar las métricas.
  + Menciona dos métricas comunes que convendría instrumentar en un microservicio (por ejemplo: latencia de petición, contadores de errores, uso de memoria, etc.).

Las dos métricas mas importantes para instrumentar con prometheus pueden ser:

* Request Latency: Esta métrica básicamente nos permite medir el tiempo que demora el procesamiento de una petición entrante, ayuda a detectar si hay problemas de rendmiento en el aplicativo.
* Error Rate: Esta métrica básicamente nos permite obtener el numero de peticiones y la información subyacente de la misma con los códigos de estado de error (4xx, 5xx, etc)

1. Liquibase/Flyway
   * Describe brevemente cómo funciona Liquibase (o Flyway) para gestionar migraciones de base de datos.

Los pasos básicos de funcionamiento de la herramienta liquibase para la gestión de cambios en base de datos son los siguientes:

* Creacion del changelogs: Es un archivo que puede ser escrito en yaml, json o sql en donde se puede definir los cambios a la base de datos, cada uno de estos cambios son denominados como changeSets
* Despues de tener el changelogs creado a través de liquibase se puede ejecutar la actualización con: liquibase update.
* Es posible revertir los cambios en la base de datos con el comando rollback
* En la base de datos, liquibase crea una tabla llamada databasechangelog de referencia para llevar el seguimiento de los changelogs ejecutados.
  + ¿Qué sucede si varias instancias del mismo microservicio intentan aplicar la misma migración al mismo tiempo? ¿Cómo lo evita Liquibase/Flyway?

Con liquibase se tiene una tabla llamada databasechangeloglock que permite antes de ejecutar una migración bloquear la base de datos para que solo se aplica una migración en ese preciso tiempo.

1. 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 general los pasos pueden ser los siguientes:

* git clone <repo>
* git checkout develop
* git pull origin develop
* git checkout -b feature/nueva
* git add .
* git commit -m “message”
* git rebase develop
* git push origin feature/nueva
  + En Scrum, ¿qué rol o ceremonia encajaría mejor para revisar la arquitectura de un nuevo microservicio antes de implementarlo? (menciona al menos uno).

Según lineamientos de scrum se debería hacer la revisión de la arquitectura en la reunión denominada Planning, sin embargo se ha adoptado una reunión llamada “Refinamiento” en donde también es posible revisar artquitecturas de nuevos proyectos a desarrollar.

**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).

**PRUEBA TECNICA**

Se entrega repositorio con la prueba técnica en el siguiente link:

<https://github.com/alejoved/user-management-ditech>

Tambien es posible ver en mi portafolio otros proyectos desarrollados en lenguajes como FastAPI, .NET, Spring boot, NestJS en el siguiente link:

<https://github.com/alejoved>

Certificaciones actuales: <https://www.credly.com/users/jorge-alejandro-aguirre-gutierrez>