**Prueba teórica:**

**Front-end:**

**a. En una aplicación bancaria que muestra el saldo de las cuentas, ¿cómo gestionarías el estado y la autenticación en el front-end, garantizando la seguridad y la coherencia de los datos?**

Autenticación y autorización con JWT, Asegurando que solo usuarios autenticados puedan enviar solicitudes y que solo administradores puedan aprobar o rechazar préstamos.

Y la gestión del estado, manteniendo los datos consistentes y seguros. Por ejemplo,

con angular: **Servicios inyectables o NgRx**

NgRx para estado global (préstamos, usuarios, saldo, etc.) o servicios simples si la aplicacion es pequeña

o con React, React Query o Redux Toolkit

**React Query para manejar fetch, cache y revalidación o mas completo redux toolkit.**

**SPRING BOOT**

1. **Que pasa en una aplicación internamente cuando usas la anotación** @SpringBootApplication **y cómo afecta el arranque de una aplicación?**

Es una anotación @SpringBootApplication, combina varias anotaciones fundamentales de Spring internamente marca la clase como fuente de definiciones de beans para el contexto de Spring, permite que Spring Boot configure automáticamente beans según las dependencias presentes en el classpath y escanea paquetes desde la ubicación de la clase anotada hacia abajo, buscando clases anotadas.

afecta en el arranque realizando auto-configuración según dependencias detectadas y escaneando los componentes y registra beans en el contexto, basicamente, con una sola anotación configuras y arrancas una aplicacion completa de Spring Boot sin necesidad de XML ni mucha configuración manual.

**b. ¿Cómo funciona el ciclo de vida de un beans en Spring y como podrías**

**intervenir en él?**

Spring crea una instancia del bean, inyecta todas las dependencias marcadas con @Autowired, @Value, @Resource, etc, Se llama a métodos anotados con @PostConstruct y si el bean está listo se puede inyectar en otros componentes pero antes de destruir el bean, Spring llama a métodos @PreDestroy o a Disposable.

Se puede intervenir usando:

@PostConstruct / @PreDestroy → hooks simples

InitializingBean / DisposableBean → interfaz para lógica de init/destroy.

BeanPostProcessor → intercepta todos los beans antes/después de la inicialización.

BeanFactoryPostProcessor / ApplicationContextInitializer → modificar definición de beans antes de instanciarlos.

**c. ¿Cómo personalizarías el comportamiento de la auto-configuración en Spring Boot sin romper la filosofía de 'convención sobre configuración'?**

1. Propiedades en application.properties o application.yml, sobrescribir valores de auto-configuración de manera declarativa.

**server.port=8081**

**spring.datasource.url=jdbc:mysql://localhost:3306/mi\_db**

2. Creando beans propios

**Spring Boot usará tu bean personalizado en lugar del auto-configurado, gracias a la anotación @ConditionalOnMissingBean.**

3. Excluir auto-configuración específica, si no quieres un auto-configuration particular:

**@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})**

**public class MiAplicacion {}**

4. O Usar @Conditional o @ConditionalOnProperty

**Personalizar solo cuando ciertas propiedades estén presentes o activas.**

**Prueba Práctica:**

# ****Explicación visual****

**Incluye:**

1. **Entidades y dominios:**
   * Prestamo con estados (PENDIENTE, APROBADO, RECHAZADO)
   * Usuario con rol (ROLE\_USER / ROLE\_ADMIN)
   * Excepciones personalizadas (BusinessException, NotFoundException)
2. **DTOs y validaciones:**
   * PrestamoRequestDTO, PrestamoResponseDTO
   * UsuarioRequestDTO, UsuarioResponseDTO
   * Validaciones con Hibernate Validator (@NotNull, @Min, @Email, etc.)
3. **Repositorios JPA:**
   * PrestamoRepository, UsuarioRepository
4. **Servicios / Use Cases:**
   * PrestamoService y UsuarioService con lógica de negocio
   * Transacciones en aprobaciones/rechazos de préstamos
   * Cache EhCache para reducir consultas repetitivas:

Separar la configuración de caché de la lógica de negocio

**Esto sigue buenas prácticas de arquitectura limpia / hexagonal.**

Permite usar @Cacheable en los servicios

1. **Controladores REST / WebFlux:**
   * PrestamoController y UsuarioController
   * Endpoints reactivos (Mono / Flux) donde aplica
2. **Seguridad Spring Security:**
   * JWT token-based authentication
   * CustomUserDetailsService consultando usuarios de la base de datos
   * Filtro JWT y configuración de roles
3. **Base de datos MySQL:**
   * Creación automática de la base (prestamosdb) si no existe
   * Tablas usuarios y prestamos generadas automáticamente por JPA
   * Usuarios predeterminados (usuario@test.com / [admin@test.com](mailto:admin@test.com))
4. **Tests unitarios y de integración:**
   * PrestamoServiceTest y UsuarioServiceTest con JUnit 5
   * Tests de endpoints REST con @WebFluxTest
   * Validación de reglas de negocio y seguridad
5. **Archivos de configuración:**
   * application.properties
   * ehcache.xml

# ****Arquitectura limpia y hexagonal****

com.example.sistemaprestamos

* service

1. PrestamoService.java
2. UsuarioService.java

* Domain

1. Prestamo.java
2. PrestamoStatus.java
3. Usuario.java

4. exception/

* 1. BusinessException.java
  2. NotFoundException.java
* Infrastructure

1. repository/
   1. PrestamoRepository.java
   2. UsuarioRepository.java
2. cache/
3. security/
   1. JwtUtils.java
   2. JwtAuthFilter.java
   3. CustomUserDetailsService.java
   4. SecurityConfig.java

* controller

1. PrestamoController.java
2. UsuarioController.java

* dto

1. PrestamoRequestDTO.java
2. PrestamoResponseDTO.java
3. UsuarioRequestDTO.java
4. UsuarioResponseDTO.java