1. Optimización en sistemas financieros:

a. Imagina que trabajas en una aplicación bancaria que debe procesar transacciones en tiempo real. ¿Qué estrategias aplicarías para mejorar el rendimiento y la escalabilidad? Considera aspectos como concurrencia, caché y patrones de diseño.

**No bloquear**: usar procesamiento reactivo para partes IO-bound (WebFlux, R2DBC) o hilos con pools bien dimensionados para tareas CPU/IO.

**Particionado y sharding**: particionar cuentas/transacciones por clave (accountId % N) para evitar hot-spots y permitir paralelismo independiente.

**Caché**: cache reads con *cache-aside* (Redis, Caffeine) para datos poco volátiles (catálogos, límites). Evitar inconsistencia con TTL corto o invalidación por eventos.

**Batching y bulk**: agrupar escrituras/consumos (por ej. pagos masivos) para reducir overhead DB/IO.

**Patrones resilientes**: circuit breaker, bulkhead, retry con backoff, timeouts.

**Colas/eventos**: uso de colas (Kafka/Rabbit) para desacoplar y absorber ráfagas; procesamiento idempotente.

**Observabilidad y tuning**: métricas, tracing distribuido y APM; perf tune en DB (indices, queries), connection pooling, GC y hardware adecuados.

**Diseño**: CQRS para separar lectura/escritura, comandos idempotentes, comandos síncronos cortos y trabajo asíncrono para lo pesado.

2. Seguridad en APIs financieras:

a. Explica cómo protegerías una API que maneja información sensible de cuentas bancarias contra ataques como inyección SQL, CSRF, XSS y otros ataques comunes.

**Autenticación/Autorización**: OAuth2/OIDC (Keycloak), scopes/roles, least privilege.

**Inyección SQL**: nunca concatenar SQL; usar parámetros/prepared statements o query builders/ORMs; validar y normalizar inputs.

**CSRF**: para apps que usan cookies, habilitar protección CSRF (tokens); para APIs basadas en tokens (Bearer) es menos relevante pero asegurar CORS.

**XSS**: escapar/sanitizar todo contenido que vuelve al cliente; CSP headers; no poner HTML desde el servidor sin sanitizar.

**Transporte & datos**: TLS obligatorio, HSTS, certificados válidos; cifrado at-rest (DB, backups) y en tránsito.

**Rate limiting & bot protection**: limitar por IP/client, WAF y detección anómala.

**Validación y filtrado**: whitelist inputs, size limits, content-type checks.

**Gestión de secretos**: vaults (HashiCorp/Vault), no secrets en código.

3. Transacciones en sistemas distribuidos:

a. En un sistema bancario distribuido, ¿cómo implementarías la consistencia y el manejo de errores en una API que procesa transferencias entre cuentas en diferentes servicios?

**Evitar 2PC si es posible**: 2PC es complejo y bloqueante; preferir **Saga** (orquestada o coreografiada) con compensaciones.

**Outbox pattern**: escribir evento en la misma transacción local (DB) y luego publicar desde outbox para garantizar entrega (idempotencia).

**Idempotencia**: usar claves de idempotencia para evitar doble gasto (client-provided id o generated).

**Timeouts y retries**: límites y políticas de retry; circuit breakers.

**Compensaciones**: diseñar pasos compensatorios claros (revertir reserva, reembolso).

**Consistencia eventual**: documentar SLAs de consistencia y exponer estados (PENDING, COMPLETED, FAILED).

**Observabilidad**: tracing distribuido para seguir transferencias y diagnosticar fallos; colas DLQ para errores manuales.

4. Pruebas unitarias y de integración:

a. Describe cómo diseñarías una suite de pruebas que asegure la correcta operación de una API bancaria, considerando tanto pruebas unitarias como de integración. ¿Qué herramientas utilizarías?

**Unit tests**: JUnit 5 + Mockito (o Mockk), probar lógica de negocio aislada, validaciones, mappers. Para reactive: Reactor Test (StepVerifier).

**Integration tests**: Spring Boot Test / WebTestClient (WebFlux). Usar **Testcontainers** para DB real (MySQL/Postgres), Kafka, Redis: pruebas con infra real.

**Contract tests**: Pact o Spring Cloud Contract para asegurar contratos entre servicios.

**End-to-end**: pruebas E2E con Cypress / Playwright (frontend).

**Fixtures / data builders**: para crear objetos de prueba claros.

**CI**: ejecutar tests en pipeline, separar suites rápidas (unit) y lentas (integration).

**Cobertura & calidad**: métricas, pero no sacrificar diseño por 100% coverage.

5. 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 segura**: usar httpOnly, Secure cookies para tokens o Authorization header con Access+Refresh tokens (Refresh rotativo). Evitar localStorage para tokens sensibles.

**Gestión de estado server-state**: usar React Query / SWR para cache, revalidación automática y stale-while-revalidate; garantiza coherencia y sincronización.

**Estado local**: Redux/Context solo para UI local (filters, modals). Mantener la fuente de verdad en el servidor.

**Actualizaciones en tiempo real**: WebSocket/Server-Sent Events para push de saldo o reconciliaciones periódicas (polling corto si no).

**Optimistic updates**: solo en UI no crítica; revertir on error.

**CORS & CSRF**: configurar backend para aceptar solo orígenes autorizados; si usas cookies, proteger con CSRF token.

**Hardening UI**: Content Security Policy, escape de datos, evitar inline scripts.

**UX de seguridad**: mostrar estados PENDING/STALE y botones para refresh, y confirmar acciones sensibles.

6. SPRING BOOT

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

Es una meta-anotación que combina:

- @Configuration (clase de configuración),
- @EnableAutoConfiguration (activa la auto-configuración basada en classpath y propiedades),
- @ComponentScan (escaneo de componentes en el paquete).

Al arrancar, SpringApplication.run(...) crea el ApplicationContext, aplica auto-configuraciones, registra beans por convención y prepara el entorno (profiles, properties, logging).

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

**Fases**: instanciación → inyección de dependencias → BeanPostProcessor.postProcessBeforeInitialization → @PostConstruct / afterPropertiesSet() →

BeanPostProcessor.postProcessAfterInitialization → uso → @PreDestroy / DisposableBean.destroy() al cerrar.

**Intervenir**: usar @PostConstruct/@PreDestroy, implementar InitializingBean/DisposableBean, registrar BeanPostProcessor para modificar beans, BeanFactoryPostProcessor/EnvironmentPostProcessor antes de crear beans, o usar SmartLifecycle para controlar start/stop.

c. ¿Cómo personalizarías el comportamiento de la auto-configuración en Spring

Boot sin romper la filosofía de 'convención sobre configuración'?

**Propiedades**: sobrescribir con application.properties/application.yml

**Conditional beans**: crear beans con condiciones (@ConditionalOnMissingBean, @ConditionalOnProperty) para que se apliquen solo cuando convenga.

**Excluir auto-configuración**: @SpringBootApplication(exclude = {...}) o spring.autoconfigure.exclude en properties para desactivar módulos concretos.

**Custom AutoConfiguration**: crear una @Configuration con @AutoConfigureBefore/After y empaquetarla como starter si necesitas integrarlo como extensión; mantener Conditional para no romper defaults.