TALLER PRÁCTICO

Análisis, Evaluación y Propuesta de Refactorización de Arquitectura Monolítica

Proyecto: Sistema de Encuestas "Espagueti"

Asignatura: Arquitectura de Software

1. Introducción

La refactorización del sistema monolítico 'Espagueti de Encuestas' busca mejorar su escalabilidad, mantenibilidad y seguridad mediante la adopción de una arquitectura bien estructurada que permita una futura migración a una arquitectura de microservicios si el sistema crece de manera significativa. Esta propuesta detalla cómo iniciar con un monolito bien organizado y modular, con la posibilidad de separar el sistema en microservicios en etapas futuras cuando sea necesario.

2. Análisis de Anti-Patrones y Propuestas de Mejora

2.1 Anti-patrón: Todo en un Controlador

Problema: El controlador EncuestaController.java tiene múltiples responsabilidades.

Solución: Implementación de una arquitectura en capas (Controller → Service → Repository), que mejora la mantenibilidad y permite pruebas unitarias más sencillas.

2.2 Anti-patrón: SQL Injection

Problema: Uso de concatenación SQL, lo que es vulnerable a SQL Injection.

Solución: Aplicación del Repository Pattern utilizando JPA y Spring Data para gestionar las consultas a la base de datos de manera más segura.

2.3 Anti-patrón: Configuración Hardcodeada

Problema: URL de la API hardcodeada en el frontend.

Solución: Implementación de una capa de servicio en Angular y externalización de la configuración en el archivo environment.ts.

2.4 Anti-patrón: Manejo de Errores Inadecuado

Problema: Uso de printStackTrace() y retorno de null en caso de errores.

Solución: Creación de un Global Exception Handler con @ControllerAdvice para una gestión centralizada y clara de los errores.

3. Propuesta de Patrones de Diseño

3.1 Patrones de Diseño Propuestos

Layered Architecture (MVC) para separar las responsabilidades en Controller → Service → Repository.

Repository Pattern con Spring Data para evitar el uso de SQL directo.

DTO Pattern para mejorar el tipado y la claridad en las respuestas de la API.

3.2 Implementación de Patrones de Diseño

Controller

```
@RestController
@RequestMapping("/api/surveys")
public class EncuestaController {
    private final EncuestaService encuestaService;

    public EncuestaController(EncuestaService encuestaService) {
        this.encuestaService = encuestaService;
    }

    @PostMapping
    public ResponseEntity<EncuestaDTO> crear(@RequestBody EncuestaDTO
dto) {
        return ResponseEntity.ok(encuestaService.crearEncuesta(dto));
    }
}
```

Service

```
@Service
public class EncuestaService {
    private final EncuestaRepository repository;

    public EncuestaService(EncuestaRepository repository) {
        this.repository = repository;
```

```
    }

    public EncuestaDTO crearEncuesta(EncuestaDTO dto) {
        validarTexto(dto.getTexto());
        return repository.save(dto);
    }
}
```

4. Propuesta de Arquitectura de Microservicios

En esta fase, proponemos iniciar con un monolito bien estructurado, modular y escalable, que pueda evolucionar hacia una arquitectura de microservicios cuando sea necesario. Esta estrategia ofrece un balance entre simplicidad y flexibilidad, permitiendo que el sistema crezca de manera controlada.

4.1 Microservicios Propuestos

survey-service: Gestiona las encuestas (CRUD).

voting-service: Registra y cuenta los votos.

API Gateway: Maneja las peticiones de los clientes, enrutamiento y control de acceso.

4.2 Componentes Adicionales a Considerar

Service Discovery (Eureka o Consul) para el descubrimiento de servicios.

Config Server para la centralización de la configuración.

Message Broker (RabbitMQ o Kafka) para la comunicación entre microservicios.

Circuit Breaker (Resilience4j) para mejorar la resiliencia del sistema.

Centralized Logging (ELK Stack) para la recolección y análisis de logs.

5. ADR (Architecture Decision Records)

ADR-001: Adoptar Arquitectura en Capas

Contexto: El controlador actual maneja demasiadas responsabilidades.

Decisión: Adoptar una arquitectura en capas (Controller → Service → Repository).

Consecuencia: Mejora en la mantenibilidad, pero mayor cantidad de clases y complejidad inicial.

ADR-002: Usar JPA y Spring Data

Contexto: El uso de SQL directo es vulnerable a inyecciones SQL.

Decisión: Implementar JPA con Spring Data.

Consecuencia: Mayor seguridad y facilidad de mantenimiento del código.

ADR-003: Descomponer en Microservicios

Contexto: El sistema monolítico es difícil de escalar y mantener.

Decisión: Separar en microservicios: survey-service, voting-service, API Gateway.

Consecuencia: Aumento de la escalabilidad y flexibilidad, pero mayor complejidad operativa.

6. Conclusión

La propuesta de refactorización presentada aborda las principales deficiencias del sistema monolítico actual y proporciona una arquitectura moderna basada en un monolito modular, que tiene la posibilidad de escalar hacia una arquitectura de microservicios cuando sea necesario. Esta refactorización mejora la seguridad, mantenibilidad y escalabilidad del sistema, alineándose con las mejores prácticas de la industria.