**Integración de Arquitectura Cliente-Servicio y Pipeline CI/CD con Control de Calidad en Spring Boot: Caso MovieCards**

Hernan Andres Henao Camelo  
Universidad de Alcala de Henares  
Curso: Integración Continua en el desarrollo agil  
22 de febrero de 2026

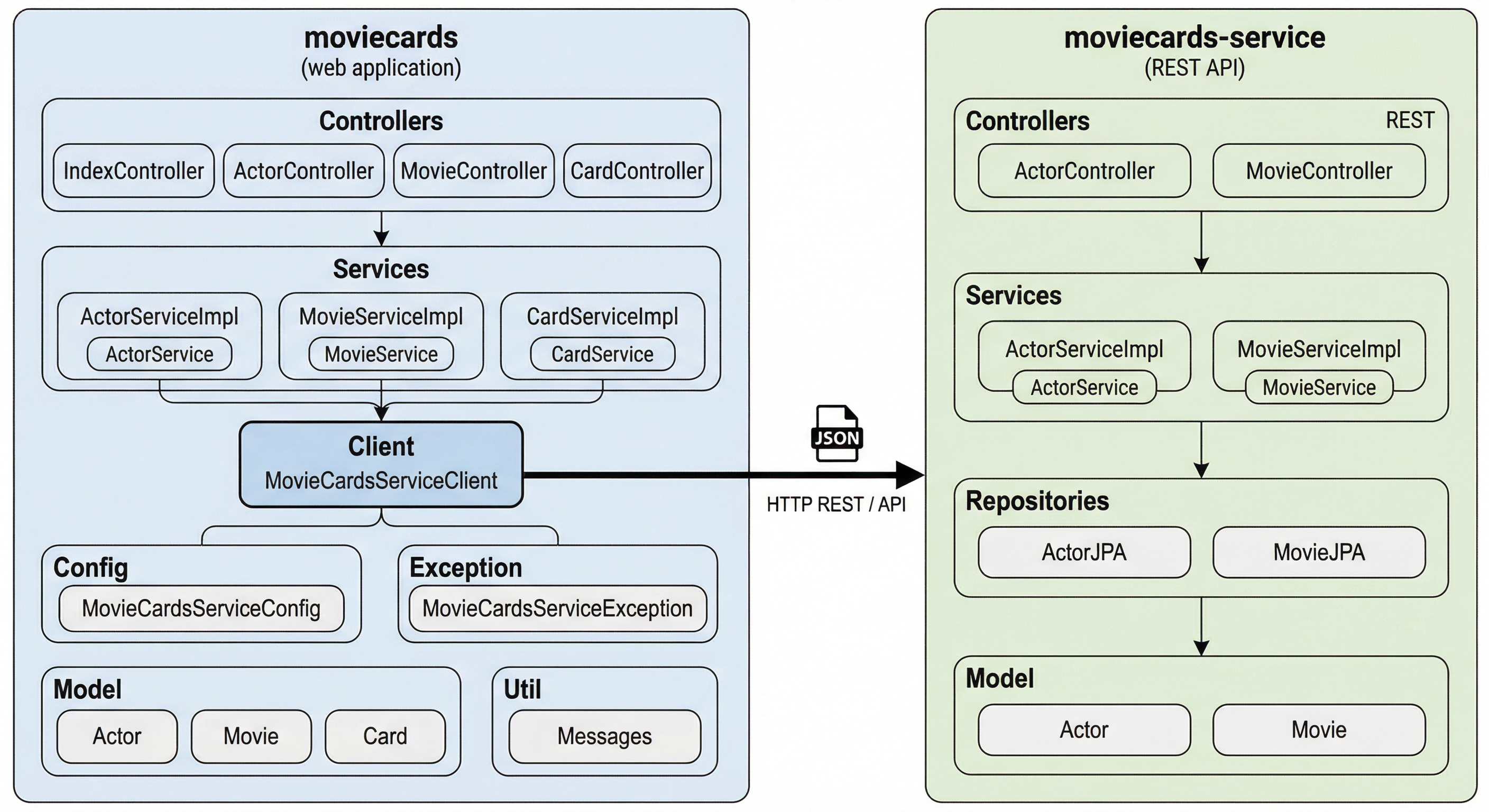
# Resumen

Este documento presenta la sustentación técnica de los cambios realizados en los proyectos MovieCards y MovieCards-Service. Se describe la evolución desde una arquitectura con persistencia JPA local hacia un modelo cliente-servidor desacoplado basado en micro servicios, así como la implementación de un pipeline completo de Integración Continua y Despliegue Continuo (CI/CD). El flujo incorpora pruebas automatizadas, análisis estático con SonarQube, control de compuertas de calidad basadas en issues críticos y despliegue en Microsoft Azure en entornos de preproducción y producción con aprobación manual.

**Palabras clave:** Integración Continua, Microservicios, Spring Boot, SonarQube, GitHub Actions, Azure.

# 1. Introducción y Contexto

El trabajo desarrollado integra dos proyectos relacionados: MovieCards, aplicación web desarrollada en Spring Boot, y MovieCards-Service, micro servicio que expone una API REST para la gestión de datos. El objetivo principal fue desacoplar la capa de persistencia de la aplicación web y consolidar una arquitectura cliente-servidor, fortalecida por un pipeline CI/CD robusto y controlado por métricas de calidad.



# 2. Proyecto MovieCards – Aplicación Web

MovieCards constituye la capa visible para el usuario final. Está desarrollada con Java 11, Spring Boot 2.6.7, Thymeleaf como motor de plantillas y MySQL en producción (H2 en pruebas). Inicialmente utilizaba repositorios JPA locales. Sin embargo, fue re factorizada para consumir exclusivamente el micro servicio MovieCards-Service mediante un cliente REST centralizado.

Se implementó la clase MovieCardsServiceClient para encapsular todas las llamadas HTTP al API, evitando duplicación de literales mediante el uso de constantes y permitiendo configurar la URL del servicio desplegado en Azure en la propiedad moviecards.service.url del application.properties.

En el modelo Actor se incorporó el atributo deadDate (fecha de fallecimiento), manteniendo coherencia con el backend. Este campo es opcional y fue integrado tanto en el formulario como en la vista de listado.

Las pruebas unitarias mockean el cliente REST para aislar la lógica de negocio. Las pruebas end-to-end con Selenium se ejecutan bajo el perfil test, permitiendo simular respuestas cuando el servicio no está disponible y garantizando estabilidad en el pipeline.

# 3. Proyecto MovieCards-Service – API Backend

MovieCards-Service actúa como backend de datos en la solución. Está desarrollado con Spring Boot y Spring Data JPA, utilizando H2 en memoria por defecto. Expone endpoints REST para listar, crear y relacionar actores y películas.

Su pipeline CI es más simple: build, test y despliegue en Azure con aprobación manual en rama master. Este diseño permitió la independencia entre frontend y backend, favoreciendo escalabilidad y mantenimiento.

# 4. Integración Cliente-Servicio

La integración entre ambos proyectos se diseñó bajo principios de mantenibilidad y claridad. La clase MovieCardsServiceConfig configura el RestTemplate y propiedades como timeout y URL base.

Se creó una excepción dedicada, MovieCardsServiceException, que encapsula información de diagnóstico como URL invocada, código HTTP y cuerpo de respuesta, mejorando la trazabilidad en logs.

# 5. Pipeline CI/CD de MovieCards

El workflow de GitHub Actions está compuesto por múltiples jobs encadenados: Build, Test, QA, Stage y Deploy para el despliegue de ambos proyectos.

El job Test ejecuta pruebas unitarias, de integración y E2E con Selenium. El job QA realiza análisis con SonarQube en un runner self-hosted llamado mi-runner.

Posteriormente se consulta la API de SonarQube para contar issues críticos. Si el número supera el umbral configurable SONAR\_MAX\_CRITICAL\_ISSUES (por defecto 5), el despliegue queda bloqueado.

El despliegue en preproducción (Stage) ocurre únicamente si se cumplen las condiciones de calidad. El despliegue en producción requiere además aprobación manual.

# 6. Calidad de Código y SonarQube

Se configuró sonar-project.properties para excluir código de pruebas y modelos del análisis de cobertura. Se aplicaron refactorizaciones para eliminar code smells, literales duplicados y mejorar manejo de excepciones.

El pipeline consulta correctamente la API de SonarQube verificando código HTTP 200 y parseando el total de issues críticos mediante herramientas como jq o grep/sed cuando sea necesario.

# 7. Conclusiones

* El proyecto evolucionó hacia una arquitectura desacoplada basada en micro servicios, respaldada por un pipeline CI/CD robusto con control automatizado de calidad. La integración de pruebas, análisis estático y despliegue condicionado mejora significativamente la confiabilidad del sistema y evidencia la aplicación práctica de los principios DevOps.
* La separación entre moviecards (interfaz y lógica de presentación) y moviecards-service (API y persistencia) permite escalar, mantener y desplegar cada parte por separado, y reutilizar la API en otros clientes (móvil, otro front-end).
* La centralización de las llamadas HTTP en MovieCardsServiceClient y la configuración de la URL por propiedades facilita cambiar de entorno (local, staging, Azure) sin recompilar y cumple buenas prácticas de configuración externa.
* El pipeline de moviecards automatiza comprobaciones y reduce el riesgo de desplegar código sin test o con demasiados issues críticos; el criterio de deploy\_allowed basado en SonarQube hace que la calidad sea una condición explícita del despliegue.
* Tener un entorno Staging (moviecards-henao-pre) permite validar la integración con moviecards-service en Azure antes de producción y la aprobación manual para producción añade un control humano sobre el proceso de release.
* La estrategia combina pruebas unitarias (mocks del cliente REST), pruebas que ya no dependen de JPA local y E2E con Selenium. Ahora bien, el perfil test con respuestas mock cuando el servicio no está disponible permite que el pipeline sea estable, aunque el servicio externo no sea accesible en CI.
* Integrar SonarQube en el flujo y usar exclusiones/reglas multicriterio (modelos como DTOs, controladores) muestra cómo alinear análisis estático con las decisiones de diseño sin ignorar la calidad. Por otra parte, la corrección del conteo de críticos (evitar el 999 por defecto) hace que la puerta de despliegue sea fiable.
* SonarQube Community no soporta análisis por ramas, por lo que se eliminaron parámetros de rama y se adaptó la consulta de issues. En entornos self-hosted sin jq se usó grep/sed para parsear la API y garantizar el mismo comportamiento.