



INSTITUTO POLITÉCNICO NACIONAL

Escuela Superior de Cómputo

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Practica 7:

Microservicios

Alumno:

Cardoso Osorio Atl Yosafat

Profesor:

Chadwick Carreto Arellano

Fecha:

12/04/2025

INSTITUTO POLITÉCNICO NACIONAL



Índice de contenido

Antecedentes.....	5
Microservicios	5
Diferencias entre Arquitectura Monolítica y Arquitectura de Microservicios	6
Características de la Arquitectura de Microservicios	8
Relevancia en sistemas distribuidos modernos	10
Ventajas de los Microservicios.....	10
Desventajas y Desafíos de los Microservicios	11
Casos de Uso y Aplicaciones de Microservicios en Sistemas Distribuidos	12
Netflix – Escalabilidad y Resiliencia.....	12
Amazon – Escalado Independiente y Autonomía de Equipos.....	13
Spotify – Independencia Funcional y Desarrollo Ágil	14
Airbnb – Escalabilidad Dinámica y Observabilidad	14
Servicios Web RESTful	14
Métodos HTTP Utilizados en una API REST	16
Diferencias entre REST y SOAP.....	17
Protocolos de comunicación.....	18
Planteamiento del problema	20
Propuesta de solución	20
Materiales y métodos empleados.....	21
Materiales	21
Métodos	22
Desarrollo	25
Instalación de Maven.....	27
Configuración de la Base de Datos	27
Eurekaserver	28
Serviciocompras	29
Servicioboletos	33
Apigateway.....	37
Cliente.....	39
Instrucciones para ejecutar el proyecto	46

Resultados.....	49
Pruebas en Postman	49
Prueba de obtención de la cartelera de películas	50
Prueba de consulta de disponibilidad de boletos	50
Prueba de compra de boletos	51
Pruebas desde el Frontend	52
Visualización de la Cartelera de Películas.....	52
Proceso de Compra de Boletos	53
Pruebas de Validación del Sistema.....	55
Conclusión	56
Referencias	57

Índice de figuras

Figura 1. Estructura de un microservicio	6
Figura 2. Arquitectura del microservicio de Netflix	13
Figura 3. Arquitectura de microservicio de Amazon	14
Figura 4. Estructura básica de un servicio web REST	16
Figura 5. Ejemplo del uso de métodos HTTP	17
Figura 6. Estructura básica de un servicio web SOAP	18
Figura 7. Diagrama de protocolos de comunicación	19
Figura 8. Proyectos creados	26
Figura 9. Estructura y creación del proyecto con Spring Boot	26
Figura 10. Verificación de la instalación de Apache Maven	27
Figura 11. Creación de la base de datos Cine	27
Figura 12. Tablas creadas en la base de datos	28
Figura 13. Estructura del proyecto eurekaserver	28
Figura 14. Clase principal del servidor Eureka	29
Figura 15. Configuración del servidor Eureka	29
Figura 16. Estructura del proyecto serviciocompras	30
Figura 17. Configuración del servicio compras	30
Figura 18. Controlador del servicio compras	31
Figura 19. Clase del servicio del servicio compras	31
Figura 20. Modelo del servicio compras	32
Figura 21. CorsConfig del servicio compras	33
Figura 22. Estructura del proyecto servicioboletos	33
Figura 23. Modelo de datos del servicio boletos	34

Figura 24. Repositorio del servicio boletos.....	34
Figura 25. Lógica del servicio boletos	35
Figura 26. Clase ProjectorServicio del servicio boletos	35
Figura 27. Controlador del servicio boletos	36
Figura 28. Configuración del servicio boletos.....	37
Figura 29. Estructura del proyecto Apigateway	37
Figura 30. Configuración del Apigateway	38
Figura 31. Políticas CORS globales	38
Figura 32. Filtro implementado en el apigateway	39
Figura 33. Estructura del servicio del Cliente web.....	40
Figura 34. Clase FrontendController para la interfaz de usuario	40
Figura 35. Frontend del servicio Cliente	41
Figura 36. Fragmento del código style.css.....	42
Figura 37. Conexión con el APIGATEWAY	42
Figura 38. Función asíncrona para obtener la cartelera	43
Figura 39. Función para mostrar las películas en el frontend.....	43
Figura 40. Función asíncrona para comprar Boletos.....	44
Figura 41. Solicitud para realizar compras	44
Figura 42. Función para Refrescar disponibilidad e iniciar cartelera	45
Figura 43. Ejecución del proyecto EurekaServer	46
Figura 44. Ejecución del proyecto servicioboletos.....	46
Figura 45. Ejecución del proyecto serviciocompras	47
Figura 46. Ejecución del proyecto Apigateway.....	48
Figura 47. Ejecución del proyecto Cine.....	48
Figura 48. Frontend del servicio Cliente	48
Figura 49. Servicios registrados vistos desde el panel de Eureka.	49
Figura 50. Prueba con Postman de obtención de la cartelera	50
Figura 51. Prueba con Postman de obtención del stock específico de una película	51
Figura 52. Prueba con Postman al realizar una compra	51
Figura 53. Actualización de la disponibilidad al realizar una compra en la BD.	52
Figura 54. Prueba con Postman de fallo al realizar una compra.....	52
Figura 55. Despliegue de la cartelera de películas en la página	53
Figura 56. Realización de una compra exitosa desde el sitio web	54
Figura 59. Bitácora en donde se comprueba la conexión exitosa del front	54
Figura 60 Registro exitoso en terminal.....	55
Figura 57. Solicitud al usuario de que seleccione al menos una película.....	55
Figura 58. Solicitud al usuario para que ingrese su nombre al realizar una compra. .	55

Antecedentes

La arquitectura de microservicios ha revolucionado el desarrollo de aplicaciones distribuidas al fomentar la creación de sistemas compuestos por componentes pequeños, autónomos y que se pueden desplegar de manera independiente. Esta evolución surge como una respuesta a las limitaciones de los enfoques monolíticos, especialmente en lo que respecta a la escalabilidad, el mantenimiento y el despliegue continuo. Al dividir una aplicación en servicios independientes, cada uno responsable de una función específica, se facilita el desarrollo colaborativo, la adopción de diversas tecnologías por servicio y una mayor agilidad frente a cambios en los requisitos del negocio. Esta arquitectura emplea protocolos ligeros y estándares comunes, como HTTP/REST y mensajería asíncrona, para asegurar una comunicación eficaz entre los servicios, favoreciendo la interoperabilidad, la resiliencia y la escalabilidad horizontal en entornos modernos como la nube.

Microservicios

La arquitectura de microservicios es un enfoque de diseño de software que organiza una aplicación como un conjunto de servicios pequeños, independientes y desplegables de manera autónoma. Cada microservicio está diseñado para abordar una función específica dentro del sistema global, ejecutándose en su propio proceso y comunicándose con otros servicios a través de protocolos ligeros, como HTTP con APIs RESTful o mediante mensajería asíncrona.

Una de las principales ventajas de los microservicios es su capacidad para ser desarrollados, probados, desplegados y escalados de forma independiente. Esto permite a las organizaciones adoptar un enfoque más ágil y modular en el desarrollo de software, facilitando la integración y el despliegue continuo (CI/CD). Además, cada microservicio puede ser implementado utilizando diferentes tecnologías o lenguajes de programación, siempre que cumpla con el contrato de comunicación acordado.

Los microservicios representan una evolución hacia una arquitectura más modular, centrada en la separación de responsabilidades y la autonomía de cada componente del sistema. Este enfoque no surgió de manera repentina, sino que se desarrolló como respuesta a las limitaciones de los sistemas monolíticos y la creciente demanda de escalabilidad, mantenibilidad y rapidez en el desarrollo de software moderno. Su origen puede rastrearse en prácticas arquitectónicas previas, como la arquitectura orientada a servicios (SOA), que promovía la creación de aplicaciones basadas en servicios reutilizables y débilmente acoplados.

A principios de la década de 2000, empresas como Amazon y Netflix comenzaron a enfrentar dificultades al escalar sus arquitecturas monolíticas, tales como tiempos de despliegue largos, complejidades al hacer cambios en partes específicas del sistema sin afectar el resto, y problemas de escalabilidad horizontal. En respuesta a estos desafíos, comenzaron a

descomponer sus aplicaciones en servicios más pequeños y autónomos. Esta práctica, al principio no estandarizada, ganó popularidad, y hacia 2011, el término "microservicios" empezó a utilizarse de manera formal.

Para 2014, los microservicios se consolidaron como un paradigma arquitectónico claramente definido, impulsado por publicaciones, conferencias y libros técnicos que establecieron sus principios clave. Actualmente, la arquitectura de microservicios es ampliamente adoptada por empresas de todos los tamaños, respaldada por tecnologías como Docker, Kubernetes, Spring Boot y plataformas en la nube, que facilitan su implementación. A continuación, se presenta un ejemplo de la estructura de un microservicio.

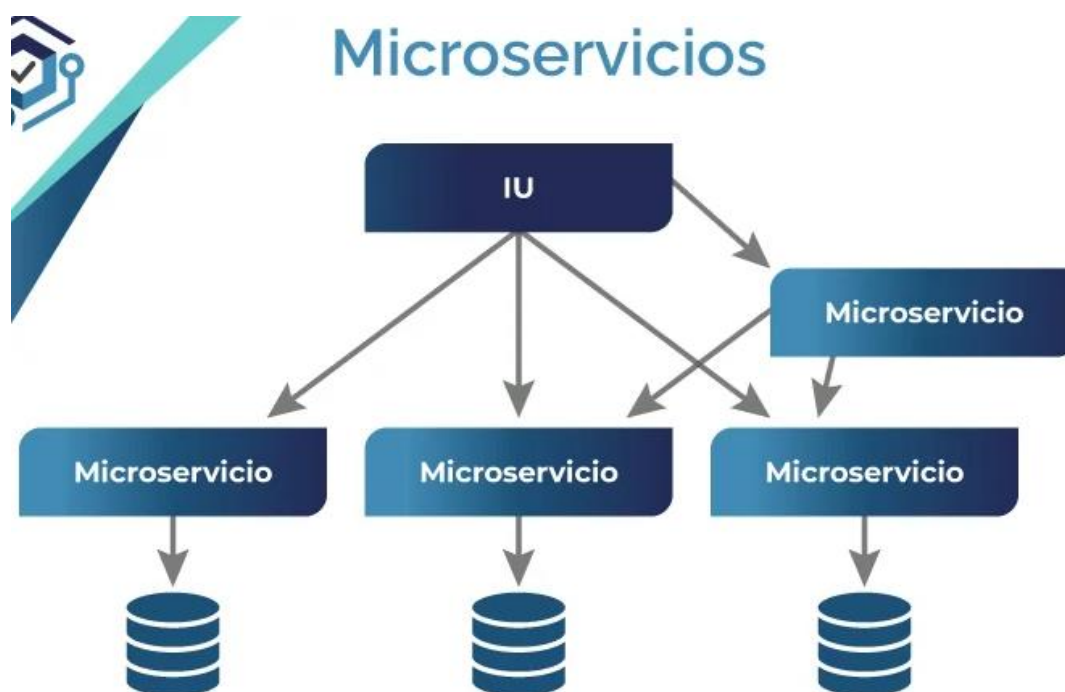


Figura 1. Estructura de un microservicio

Diferencias entre Arquitectura Monolítica y Arquitectura de Microservicios

La arquitectura monolítica representa un enfoque tradicional en el diseño de aplicaciones, en el que todos los componentes —como la lógica de negocio, la capa de presentación, el acceso a datos y otros módulos funcionales— están integrados en un único bloque de software que se despliega y ejecuta como una sola unidad. Este modelo es más simple de desarrollar y administrar en sus etapas iniciales, pero presenta limitaciones significativas a medida que el sistema crece en complejidad y demanda operativa.

En contraste, la arquitectura de microservicios propone una descomposición funcional del sistema, dividiendo la aplicación en múltiples servicios pequeños, autónomos e

independientes, cada uno enfocado en una funcionalidad específica del dominio de negocio. Estos servicios se comunican entre sí mediante protocolos ligeros, como HTTP/REST o sistemas de mensajería asincrónica, y pueden tener sus propias bases de datos o almacenamiento dedicado.

Característica	Arquitectura Monolítica	Arquitectura de Microservicios
Estructura	Aplicación única con todos los módulos integrados	Conjunto de servicios independientes y especializados
Despliegue	Se implementa como una sola unidad	Cada microservicio se despliega de forma autónoma
Escalabilidad	Escalamiento vertical (más recursos a un único sistema)	Escalamiento horizontal por servicio según demanda
Desarrollo	Todos los equipos trabajan sobre el mismo repositorio	Equipos pueden trabajar en paralelo con servicios aislados
Tecnologías	Restringido a un stack tecnológico común	Libre elección de lenguajes y frameworks por servicio
Tolerancia a fallos	Un fallo puede afectar a toda la aplicación	Fallos suelen estar contenidos en servicios individuales
Tiempos de despliegue	Lento, ya que requiere pruebas y empaquetado integral	Rápido, favorece CI/CD y despliegue frecuente
Mantenibilidad	Se vuelve más difícil conforme crece el código	Mantenimiento más sencillo por su modularidad
Escalamiento organizacional	Limitado, difícil dividir trabajo en equipos autónomos	Facilita organización en equipos DevOps o squads por servicio

Una de las principales desventajas del enfoque monolítico es que cualquier cambio, por pequeño que sea, requiere volver a compilar y desplegar toda la aplicación. Esto puede generar cuellos de botella en el desarrollo, dificultar la incorporación de nuevas funcionalidades y elevar el riesgo de introducir errores en partes no relacionadas del sistema. Además, las aplicaciones monolíticas suelen ser difíciles de escalar eficientemente, ya que para mejorar el rendimiento de una funcionalidad específica se necesita escalar toda la aplicación, lo que implica un mayor consumo de recursos.

Por otro lado, la arquitectura de microservicios permite a las organizaciones dividir su código en módulos bien definidos, facilitando la implementación de metodologías ágiles y DevOps. Cada equipo puede responsabilizarse de un microservicio, gestionando de manera independiente su ciclo de vida, desde el desarrollo y pruebas, hasta el despliegue y monitoreo. Esta independencia promueve la innovación, reduce los tiempos de entrega y permite una adaptación más rápida a los cambios del negocio.

Otro beneficio clave es la posibilidad de usar diferentes lenguajes de programación o tecnologías en cada servicio, lo que otorga mayor flexibilidad para elegir la herramienta adecuada para cada necesidad específica. Esto contrasta con los entornos monolíticos, donde la tecnología elegida al inicio suele limitar las opciones a lo largo del tiempo.

En términos de resiliencia, los microservicios también ofrecen ventajas. Al estar desacoplados, un fallo en uno de ellos no necesariamente compromete el funcionamiento de toda la aplicación. Además, se pueden implementar mecanismos como circuit breakers, fallbacks o reintentos para gestionar fallos y mantener la disponibilidad del sistema.

Características de la Arquitectura de Microservicios

En el marco de los sistemas distribuidos, la arquitectura de microservicios representa una transformación profunda frente a los modelos monolíticos tradicionales. Este enfoque propone la construcción de aplicaciones mediante un conjunto de servicios pequeños, cohesivos y autónomos, que cooperan entre sí a través de redes de comunicación. Cada microservicio encapsula una funcionalidad específica del dominio del negocio, y está diseñado para operar de forma independiente, permitiendo una mayor flexibilidad, escalabilidad y tolerancia a fallos.

A continuación, se describen las principales características que definen esta arquitectura:

Autonomía de servicios

Cada microservicio posee su propio ciclo de vida, lo que implica que puede ser desarrollado, desplegado, monitoreado y escalado de forma independiente al resto del sistema. Esta autonomía permite una gestión más eficiente del desarrollo y facilita la actualización continua sin afectar la disponibilidad general.

Bajo acoplamiento y alta cohesión

Los microservicios están diseñados para reducir al mínimo las dependencias entre ellos. Esta propiedad de desacoplamiento asegura que los cambios internos en un servicio no impacten el comportamiento de otros, permitiendo que los equipos trabajen en paralelo sin generar conflictos. Además, cada servicio mantiene una alta cohesión, enfocándose en una única responsabilidad o conjunto de funcionalidades relacionadas.

Escalabilidad granular

Una de las ventajas fundamentales de esta arquitectura es la capacidad de escalar únicamente los componentes que lo requieran. Esto se traduce en un uso más eficiente de los recursos de infraestructura, ya que no es necesario escalar toda la aplicación cuando solo una parte enfrenta alta demanda.

Especialización funcional

Cada microservicio está centrado en una funcionalidad del negocio (por ejemplo, autenticación, procesamiento de pagos, gestión de inventario), lo que facilita la comprensión del sistema, mejora la mantenibilidad del código y fomenta la reutilización de componentes.

Comunicación basada en red

La interacción entre microservicios se realiza a través de mecanismos de comunicación sobre red, como HTTP/REST, gRPC, o mediante soluciones de mensajería asincrónica como RabbitMQ, Apache Kafka o MQTT. Esta comunicación distribuida permite que los servicios residan en diferentes ubicaciones físicas o entornos de ejecución (on-premise, nube, contenedores, etc.).

Despliegue y entrega continua

El diseño independiente de los servicios permite su integración en pipelines de CI/CD (Integración y Entrega Continua), facilitando despliegues frecuentes, pruebas automatizadas y actualizaciones sin tiempos de inactividad prolongados.

Tolerancia a fallos

Dado que los microservicios operan de forma distribuida, la falla de un componente no necesariamente compromete la integridad del sistema completo. Esta característica fortalece la resiliencia del sistema, especialmente cuando se incorporan técnicas como circuit breakers, fallbacks y retry patterns.

Persistencia distribuida

Cada microservicio puede gestionar su propia base de datos o esquema de almacenamiento, lo cual refuerza su independencia. Este enfoque, sin embargo, plantea desafíos adicionales en cuanto a la consistencia eventual, la gestión de transacciones distribuidas y la sincronización de datos.

Poliglotismo tecnológico

La arquitectura de microservicios permite utilizar diferentes tecnologías, lenguajes de programación o motores de base de datos para cada servicio. Esta característica —conocida

como heterogeneidad tecnológica— ofrece flexibilidad para adoptar las herramientas más adecuadas a los requerimientos de cada componente.

Observabilidad y monitoreo avanzado

Debido a la complejidad inherente de los sistemas distribuidos, los microservicios deben contar con herramientas robustas de observabilidad, que incluyan monitoreo, trazabilidad de solicitudes (tracing), métricas personalizadas y manejo de logs centralizado. Tecnologías como Prometheus, Grafana, ELK Stack y OpenTelemetry son comúnmente utilizadas para tal fin.

Relevancia en sistemas distribuidos modernos

Estas características hacen que la arquitectura de microservicios sea especialmente apropiada para aplicaciones modernas que requieren escalar rápidamente, adaptarse a cambios frecuentes y mantenerse operativas en entornos de alta disponibilidad. En el contexto de la computación en la nube, contenedores (como Docker) y orquestadores (como Kubernetes), los microservicios facilitan el despliegue automatizado, la recuperación ante fallos y la implementación continua en infraestructuras dinámicas.

La arquitectura de microservicios no solo redefine la forma en que se construyen las aplicaciones, sino también cómo se organizan los equipos de desarrollo, fomentando modelos colaborativos como DevOps, SRE (Site Reliability Engineering) y arquitecturas orientadas a dominio (Domain-Driven Design - DDD).

Ventajas de los Microservicios

Modularidad y separación de responsabilidades

La arquitectura de microservicios se basa en el principio de alta cohesión y bajo acoplamiento. Cada servicio está orientado a una única responsabilidad del negocio, lo que promueve un diseño más limpio, estructurado y comprensible. Esta modularidad permite aplicar prácticas de diseño como Domain-Driven Design (DDD) y facilita el mantenimiento a largo plazo del sistema.

Escalabilidad granular

Una de las mayores fortalezas de los microservicios es su capacidad para escalar de manera independiente. En lugar de escalar toda la aplicación, es posible escalar únicamente los servicios con mayor carga o demanda, lo que mejora la eficiencia en el uso de recursos y permite reducir costos en infraestructura.

Desarrollo paralelo y mayor productividad

La independencia de los servicios permite que múltiples equipos trabajen en paralelo sobre distintos componentes sin generar conflictos, lo cual acelera significativamente el ciclo de desarrollo. Esta estructura organizacional también facilita la especialización de los equipos por dominio funcional.

Facilidad de mantenimiento y evolución

Gracias al desacoplamiento, los cambios o actualizaciones en un microservicio no afectan directamente al resto del sistema. Esto permite implementar mejoras, corregir errores o introducir nuevas funcionalidades de forma continua y con menor riesgo.

Flexibilidad tecnológica

Cada microservicio puede ser desarrollado utilizando el lenguaje de programación, framework, motor de base de datos o sistema operativo más adecuado a sus requerimientos, lo que se conoce como poliglotismo tecnológico. Esta flexibilidad permite adoptar nuevas tecnologías sin comprometer el resto del sistema.

Despliegue independiente y continuo

Los microservicios se integran de forma natural con prácticas DevOps y herramientas de CI/CD (Integración y Entrega Continua), lo que permite realizar despliegues frecuentes, automatizados y controlados, incluso en producción.

Mayor resiliencia

La arquitectura distribuida de los microservicios permite aislar fallos, de modo que si un componente falla, no compromete necesariamente la estabilidad del sistema completo. Esto incrementa la disponibilidad del sistema y facilita la implementación de estrategias de recuperación automática (failover, circuit breaker, retry).

Desventajas y Desafíos de los Microservicios

Aumento en la complejidad de infraestructura

La gestión de múltiples servicios distribuidos requiere una infraestructura más compleja, que incluya herramientas para balanceo de carga, descubrimiento de servicios (service discovery), monitoreo, trazabilidad y orquestación. Tecnologías como Kubernetes, Istio, Consul y Prometheus suelen ser necesarias para manejar esta complejidad.

Complejidad en la comunicación entre servicios

La interacción entre microservicios, generalmente a través de redes (REST, gRPC, mensajería asíncrona), introduce desafíos relacionados con latencias, fallos de red, serialización de datos y control de versiones de APIs. Además, pueden surgir problemas como el backpressure, duplicación de mensajes o inconsistencia en los tiempos de respuesta.

Gestión de datos distribuida

Cada microservicio puede mantener su propio almacenamiento, lo que mejora su independencia, pero complica la sincronización de información y la gestión de transacciones distribuidas (por ejemplo, cuando se requiere consistencia fuerte entre múltiples servicios). Para enfrentar esto, se suelen aplicar patrones como sagas o event sourcing.

Sobrecarga operativa

A medida que aumenta el número de servicios, también lo hacen las tareas asociadas a su despliegue, configuración, monitoreo, versionado y documentación. Esto requiere personal con experiencia en operación de sistemas distribuidos y un enfoque más robusto de automatización.

Complejidad en las pruebas

Las pruebas funcionales e integradas se vuelven más exigentes, ya que es necesario considerar múltiples interacciones, flujos distribuidos, y posibles fallos de red o servicios intermedios. Las pruebas deben contemplar no solo el comportamiento aislado de los servicios, sino también su interoperabilidad.

Seguridad distribuida

Asegurar un sistema de microservicios implica implementar mecanismos de autenticación, autorización y encriptación en cada punto de entrada. También se deben considerar vulnerabilidades en la comunicación entre servicios, ataques de intermediarios (man-in-the-middle) o uso indebido de APIs internas. Es común el uso de herramientas como OAuth 2.0, JWT, mTLS y API Gateways para mitigar estos riesgos.

Casos de Uso y Aplicaciones de Microservicios en Sistemas Distribuidos

Las arquitecturas de microservicios han transformado el desarrollo y operación de sistemas distribuidos en múltiples industrias. Este enfoque consiste en descomponer una aplicación compleja en un conjunto de servicios pequeños, independientes y especializados. Cada microservicio ejecuta una funcionalidad específica y se comunica con otros a través de interfaces bien definidas, usualmente mediante protocolos ligeros como HTTP/REST o sistemas de mensajería asíncrona como Kafka, RabbitMQ o gRPC. Esta independencia facilita el desarrollo, despliegue y mantenimiento continuo, convirtiéndolos en una herramienta clave en entornos de alta disponibilidad y rápida evolución tecnológica.

Netflix – Escalabilidad y Resiliencia

Uno de los casos más emblemáticos de implementación de microservicios es Netflix, que migró de una arquitectura monolítica a una compuesta por cientos de servicios independientes. Esta decisión fue motivada por la necesidad de escalar su plataforma ante el

crecimiento exponencial de usuarios y la demanda de nuevas funcionalidades. Servicios como reproducción de contenido, autenticación, recomendaciones, monitoreo y facturación fueron desacoplados y desplegados como componentes individuales.

Netflix desarrolló herramientas internas como Eureka para el descubrimiento de servicios, Hystrix para tolerancia a fallos y Ribbon para balanceo de carga, construyendo así una infraestructura altamente resiliente y adaptable, capaz de operar sin interrupciones incluso ante fallos parciales.

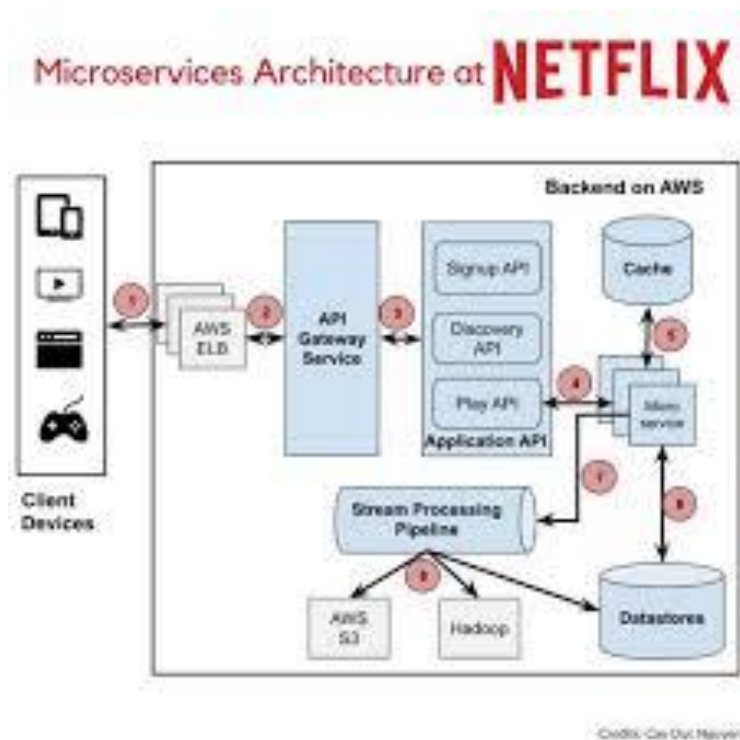


Figura 2. Arquitectura del microservicio de Netflix

Amazon – Escalado Independiente y Autonomía de Equipos

Amazon también adoptó esta arquitectura tanto en su plataforma de comercio electrónico como en su nube pública (AWS). Su sistema se estructura alrededor de múltiples microservicios que gestionan dominios funcionales como inventario, pagos, recomendaciones, logística y atención al cliente.

Cada equipo de desarrollo es responsable de sus propios servicios bajo el principio “You build it, you run it”, fomentando una responsabilidad integral sobre todo el ciclo de vida del software. Esta estrategia ha sido clave para lograr despliegues múltiples al día sin afectar la disponibilidad del sistema global, permitiendo una innovación constante y segura.



Figura 3. Arquitectura de microservicio de Amazon

Spotify – Independencia Funcional y Desarrollo Ágil

Spotify organiza su plataforma mediante microservicios enfocados en funciones específicas como reproducción de música, descubrimiento de artistas, gestión de playlists, perfiles de usuario y recomendaciones.

Esta segmentación permite que distintos equipos trabajen de forma autónoma sobre áreas específicas del sistema. Su adopción de microservicios les ha permitido escalar eficientemente su infraestructura y lanzar nuevas funcionalidades de forma continua, integrando prácticas de DevOps y CI/CD.

Airbnb – Escalabilidad Dinámica y Observabilidad

Airbnb migró su backend monolítico en Ruby on Rails hacia una arquitectura de microservicios construida en tecnologías como Node.js y Java. Esta transición permitió separar funcionalidades como gestión de reservas, validación de usuarios y mensajería entre huéspedes y anfitriones.

Para facilitar la comunicación y observabilidad entre los microservicios, Airbnb implementó una malla de servicios (service mesh) utilizando herramientas como Istio y Envoy, que proporcionan capacidades de seguridad, monitoreo, enrutamiento dinámico y resiliencia de forma transparente.

Servicios Web RESTful

Los servicios web RESTful han ganado una enorme popularidad en el desarrollo de aplicaciones distribuidas debido a su simplicidad, escalabilidad y eficiencia en la comunicación entre sistemas. REST (Representational State Transfer) es un estilo de arquitectura que se basa en el uso de los principios y convenciones de la web para facilitar la interacción entre clientes y servidores mediante operaciones bien definidas.

Este modelo de servicios web se ha convertido en la opción preferida para el diseño de APIs (Application Programming Interfaces) en aplicaciones modernas, especialmente en sistemas basados en la nube, aplicaciones móviles e Internet de las Cosas (IoT).

Entonces, REST es un conjunto de restricciones arquitectónicas para el desarrollo de servicios web que se fundamenta en la simplicidad y el aprovechamiento de tecnologías web ya existentes, como HTTP. Fue propuesto por Roy Fielding en el año 2000 en su tesis doctoral como un modelo de comunicación escalable y distribuido.

Para que un servicio web se considere RESTful, debe cumplir con ciertos principios fundamentales:

- **Modelo Cliente-Servidor:** Se mantiene una separación entre el cliente, que realiza las solicitudes, y el servidor, que responde con los recursos requeridos. Esto permite independencia en el desarrollo de ambos componentes.
- **Interfaz Uniforme:** REST define un conjunto estandarizado de operaciones para interactuar con los recursos del sistema, utilizando métodos HTTP como GET, POST, PUT y DELETE.
- **Uso de Recursos Identificables:** Cada recurso en un servicio RESTful es representado por una URL única, lo que facilita su acceso y manipulación.
- **Comunicación Sin Estado:** Cada solicitud del cliente debe contener toda la información necesaria para ser procesada por el servidor, sin depender de un estado previo en la comunicación.
- **Caché:** REST permite el uso de mecanismos de caché para mejorar el rendimiento y reducir la carga del servidor, almacenando temporalmente respuestas a solicitudes recurrentes.
- **Sistema en Capas:** La arquitectura REST permite la existencia de múltiples capas en la comunicación (como balanceadores de carga y proxies) sin que esto afecte la funcionalidad del cliente o del servidor.
- **Soporte para HATEOAS (Hypermedia as the Engine of Application State):** Este principio sugiere que un cliente debe poder descubrir dinámicamente las acciones disponibles a través de enlaces dentro de las respuestas del servicio web.

Gracias a estos principios, REST permite la creación de servicios web escalables, modulares y fáciles de mantener, adecuados para entornos de alta concurrencia. En la siguiente Figura podemos observar la estructura básica de un servicio web con REST.

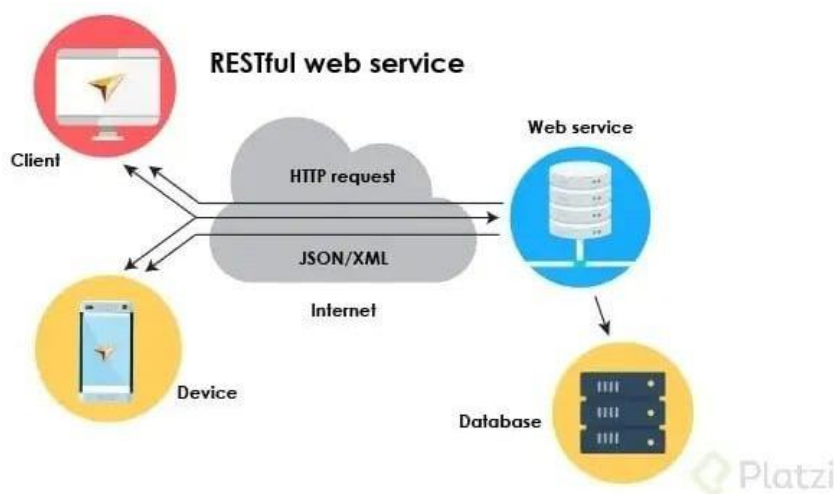


Figura 4. Estructura básica de un servicio web REST

Métodos HTTP Utilizados en una API REST

En REST, los recursos se manipulan utilizando los métodos estándar de HTTP. Cada uno de estos métodos tiene un propósito específico y define una acción sobre un recurso representado por una URL.

1. GET (Obtener Datos)

Este método se utiliza para recuperar información de un recurso sin modificarlo. Es un método seguro, lo que significa que no importa cuántas veces se realice la misma solicitud, siempre se obtendrá el mismo resultado.

2. POST (Crear un Nuevo Recurso)

Se utiliza para enviar datos al servidor y crear un nuevo recurso. A diferencia de GET, POST no es idempotente; si se envía la misma solicitud varias veces, se pueden generar múltiples recursos.

3. PUT (Actualizar un Recurso Existente)

PUT se usa para actualizar un recurso completo en el servidor. Es idempotente, lo que significa que aplicar la misma solicitud varias veces producirá el mismo resultado.

4. DELETE (Eliminar un Recurso)

Este método elimina un recurso identificado por una URL. Es idempotente, lo que significa que, si se llama varias veces a la misma URL, el resultado será el mismo (el recurso ya no existirá).

A continuación, se presenta un pequeño ejemplo de los métodos HTTP.

/books		
GET	/books	Lists all the books in the database
DELETE	/books/{bookId}	Deletes a book based on their id
POST	/books	Creates a Book
PUT	/books/{bookId}	Method to update a book
GET	/books/{bookId}	Retrieves a book based on their id

Figura 5. Ejemplo del uso de métodos HTTP

En general, los servicios web RESTful han transformado la manera en que las aplicaciones distribuidas intercambian datos en la web, gracias a su simplicidad, flexibilidad y eficiencia. Su arquitectura basada en HTTP y en la manipulación de recursos mediante métodos estandarizados permite diseñar APIs escalables y fácilmente integrables en distintos sistemas.

Diferencias entre REST y SOAP

SOAP (Simple Object Access Protocol) es otro modelo de comunicación ampliamente utilizado en servicios web, especialmente en entornos empresariales que requieren seguridad y transacciones más complejas. A continuación, se presentan las principales diferencias entre REST y SOAP en distintos aspectos:

Aspecto	REST	SOAP
Protocolo	Se basa en HTTP.	Puede utilizar múltiples protocolos (HTTP, SMTP, TCP, etc.).
Formato de datos	JSON, XML, HTML, texto plano, entre otros.	Exclusivamente XML.
Interoperabilidad	Alta interoperabilidad entre distintos sistemas y lenguajes.	Requiere herramientas específicas para interpretar XML y SOAP.
Complejidad	Simple, fácil de implementar y ligero.	Más complejo debido a su estructura basada en XML y WSDL.

Seguridad	Puede usar autenticación mediante OAuth, JWT y HTTPS.	Usa WS-Security para seguridad avanzada y control granular.
Estado	Sin estado (cada solicitud es independiente).	Puede ser con o sin estado (stateful/stateless).
Velocidad y rendimiento	Rápido debido a su estructura ligera.	Más lento debido al procesamiento de XML.

En términos generales, como podemos ver, REST es la opción preferida para la mayoría de las aplicaciones modernas debido a su simplicidad y compatibilidad con la web, mientras que SOAP se utiliza en sistemas que requieren operaciones más seguras y estructuradas. En mi caso, yo preferí utilizar REST para el desarrollo de esta práctica. Sin embargo, en la siguiente Figura podemos observar la estructura de un servicio web con SOAP.

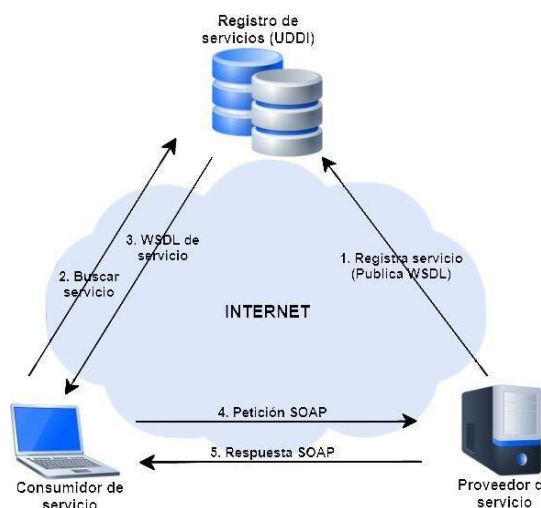


Figura 6. Estructura básica de un servicio web SOAP

Protocolos de comunicación

Los sistemas cliente-servidor dependen en gran medida de los protocolos de comunicación para facilitar la transferencia de datos entre dispositivos. Estos protocolos establecen las reglas y los procedimientos para la transmisión de información, asegurando que la comunicación sea eficiente, confiable y segura. Algunos de los protocolos más importantes aplicados en estos sistemas:

TCP/IP (Transmission Control Protocol/Internet Protocol): Este protocolo es fundamental en Internet y en los sistemas cliente-servidor. TCP garantiza una conexión confiable al dividir los datos en paquetes y asegurarse de que se entreguen correctamente, mientras que IP se encarga de direccionar los paquetes a través de la red.

HTTP (Hypertext Transfer Protocol): Ampliamente utilizado en la web, HTTP define cómo se comunican los navegadores web y los servidores. Establece un formato para las

solicitudes y respuestas, permitiendo la transferencia de recursos como páginas web, imágenes y otros archivos.

FTP (File Transfer Protocol): Especializado en la transferencia de archivos, FTP facilita la carga y descarga de archivos entre un cliente y un servidor. Proporciona un método seguro y controlado para gestionar archivos remotos.

SMTP (Simple Mail Transfer Protocol): Esencial para el envío de correos electrónicos, SMTP define cómo se transmiten los mensajes de correo entre servidores. Asegura que los correos se entreguen de manera confiable y eficiente.

SSH (Secure Shell): Utilizado para acceder de forma segura a servidores remotos, SSH cifra la comunicación entre el cliente y el servidor, protegiendo los datos sensibles durante la transmisión.

Estos protocolos son fundamentales para el funcionamiento efectivo de los sistemas cliente-servidor, ya que permiten la comunicación fluida y segura entre dispositivos. Desde la transferencia de archivos hasta el intercambio de correos electrónicos, estos protocolos juegan un papel crucial en numerosos aspectos de la interacción en línea. Su implementación adecuada garantiza una experiencia de usuario óptima y protege la integridad de los datos en todo momento. En general los protocolos de comunicación son el corazón de los sistemas cliente-servidor, proporcionando el marco necesario para la transmisión confiable de información en esta era, sin embargo, para el desarrollo de esta práctica únicamente nos enfocamos en el protocolo FTP y TCP.

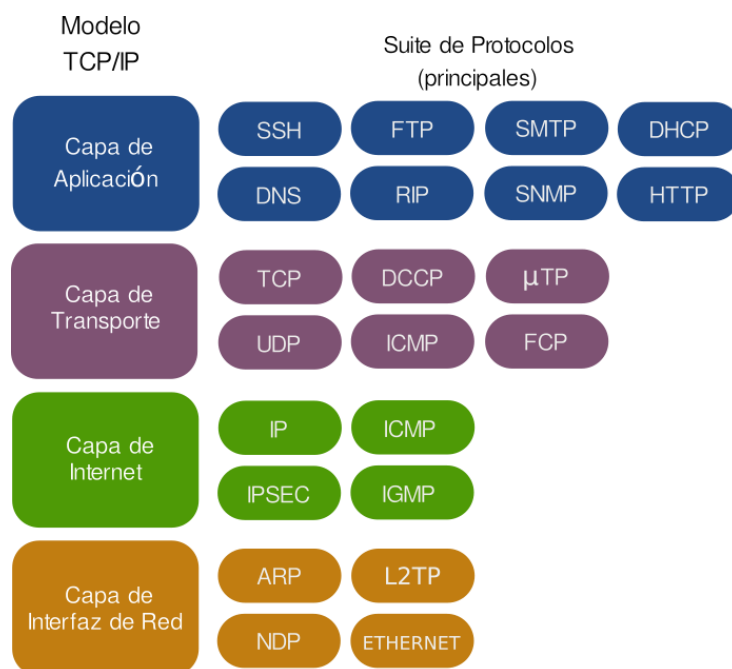


Figura 7. Diagrama de protocolos de comunicación

Planteamiento del problema

En la actualidad, las empresas dedicadas a la venta de productos y servicios enfrentan el desafío de ofrecer plataformas digitales que sean escalables, eficientes y altamente disponibles. Este reto es aún mayor en sectores como el del entretenimiento, donde la experiencia del usuario y la capacidad de respuesta del sistema son fundamentales. En particular, los cines requieren soluciones tecnológicas que les permitan gestionar de forma eficaz la cartelera, el stock de boletos disponibles por función, y el proceso de compra, todo ello de manera simultánea para múltiples usuarios. Los sistemas monolíticos tradicionales presentan limitaciones importantes en este tipo de entornos, ya que su estructura rígida dificulta la escalabilidad, reduce la tolerancia a fallos y complica la implementación de cambios sin afectar el sistema completo.

En este contexto, surge la necesidad de diseñar un sistema distribuido basado en microservicios que permita simular la operación digital de un cine moderno. Se busca resolver problemas como la gestión dinámica del stock de boletos en múltiples servidores, la distribución de la carga de usuarios concurrentes, y la comunicación eficiente entre los distintos componentes del sistema. Además, es indispensable ofrecer una interfaz amigable al usuario final, que permita seleccionar funciones, elegir boletos y realizar compras de manera intuitiva y rápida.

La implementación de un sistema de este tipo no solo permite demostrar el funcionamiento de un entorno distribuido, sino también resalta las ventajas de la arquitectura de microservicios en cuanto a escalabilidad, mantenimiento, reutilización y disponibilidad. Este enfoque responde a las necesidades reales de empresas que buscan modernizar sus operaciones, mejorar la experiencia de usuario y estar preparadas para escenarios de alta demanda.

El presente trabajo busca desarrollar y evaluar la efectividad del sistema implementado, analizando su comportamiento en términos de consistencia, eficiencia y tolerancia a fallos. Al finalizar la práctica, espero tener una mejor comprensión de cómo funcionan los microservicios en escenarios del mundo real y cómo pueden aplicarse para resolver problemas en entornos distribuidos.

Propuesta de solución

Para abordar esta problemática, se propone una aplicación distribuida basada en una arquitectura de microservicios, diseñada para simular el funcionamiento de un cine digital moderno. Esta arquitectura se compone de servicios desacoplados y autónomos que se comunican entre sí de forma eficiente a través de un API Gateway, lo que permite una mejor escalabilidad, mantenibilidad y capacidad de respuesta frente a un alto volumen de usuarios concurrentes. Cada microservicio cumple una función específica dentro del sistema,

asegurando la separación de responsabilidades y facilitando la actualización y despliegue de cada módulo de manera independiente.

El sistema contempla dos servicios principales: el servicio de inventario, encargado de gestionar el stock de boletos disponibles para las distintas funciones del cine, y el servicio de compras, responsable de registrar y procesar las solicitudes de compra realizadas por los clientes. Estos servicios se comunican mediante llamadas RESTful, usando JSON como formato de intercambio de datos, lo cual garantiza una interoperabilidad clara y eficaz entre los componentes. Para centralizar las peticiones del cliente y canalizarlas adecuadamente hacia los servicios internos, se incorpora un API Gateway que actúa como punto de entrada del sistema, encargándose de enrutar, autenticar y balancear las solicitudes.

A fin de asegurar una alta disponibilidad y distribuir la carga entre varios servidores, se implementa también un balanceador de carga utilizando Java RMI, que distribuye dinámicamente las peticiones entre múltiples instancias del servicio de inventario. Esto permite al sistema atender múltiples solicitudes simultáneamente y mejorar el tiempo de respuesta, evitando cuellos de botella y aumentando la tolerancia a fallos.

En la capa del cliente, se diseña una interfaz web amigable usando HTML, CSS y JavaScript, que permite a los usuarios consultar los boletos disponibles, seleccionar la cantidad deseada y realizar sus compras ingresando su nombre. Esta interfaz se conecta directamente con el API Gateway, facilitando una experiencia fluida para el usuario final y una comunicación clara con los microservicios del backend.

La práctica propuesta no solo resuelve los problemas asociados a sistemas monolíticos en entornos de alta demanda, sino que también permite demostrar de forma práctica los principios de los sistemas distribuidos, tales como la escalabilidad horizontal, la independencia de servicios, el balanceo de carga y la comunicación eficiente entre procesos distribuidos. Además, esta práctica también nos ofrece una base sólida para futuras extensiones, como recomendaciones de películas o autenticación de usuarios.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollarla se implementó la arquitectura de Microservicios para resolver el problema de la venta de boletos y gestión se emplearon las siguientes herramientas y métodos:

Materiales

1. Lenguaje de programación: Java

El lenguaje de programación Java fue elegido debido a su robustez y amplia compatibilidad con aplicaciones distribuidas.

2. Entorno de desarrollo: Visual Studio Code (VS Code)

Fue seleccionado como entorno de desarrollo por su flexibilidad, amplia disponibilidad de extensiones y facilidad de uso. Además, cuenta con herramientas integradas para la depuración y la administración de proyectos en Java. Así como también es bastante cómodo trabajar en él.

3. *JDK (Java Development Kit):*

El JDK fue utilizado para compilar y ejecutar el código Java. Su compatibilidad con las bibliotecas y la implementación de concurrencia lo convierten en una herramienta esencial para esta práctica.

4. *Framework: Spring Boot*

Fue seleccionado para la creación de los servicios, ya que es un framework de desarrollo basado en Java que facilita la construcción de aplicaciones backend de manera rápida y eficiente. Spring Boot permite crear aplicaciones con configuración mínima y soporta el desarrollo de servicios RESTful a través de controladores y servicios bien estructurados, integrando componentes como la programación de tareas y la gestión de base de datos.

5. *Gestión de Dependencias: Maven*

Se empleó como herramienta para la gestión de dependencias y la construcción del proyecto. Maven facilita la configuración de bibliotecas necesarias y la automatización de tareas relacionadas con el ciclo de vida del proyecto, como la compilación, pruebas y empaquetado.

6. *Sistema Operativo: Windows*

La práctica se realizó en un sistema operativo Windows debido a su amplia compatibilidad con Visual Studio Code y el JDK.

Métodos

Para la implementación del sistema distribuido basado en microservicios, se llevaron a cabo diversas metodologías y técnicas que permitieron desarrollar una solución escalable, modular, eficiente y tolerante a fallos. El desarrollo se centró en una arquitectura cliente-servidor distribuida, con separación de responsabilidades mediante microservicios independientes, comunicación RESTful, mecanismos de control de concurrencia, balanceo de carga con Java RMI y una interfaz web intuitiva para el usuario final. A continuación, se detallan los métodos utilizados:

Arquitectura Cliente-Servidor Distribuida

La solución se construyó bajo el paradigma de arquitectura distribuida cliente-servidor. En este modelo, el cliente se encarga de la interacción directa con el usuario a través de una interfaz web, mientras que los servidores procesan las solicitudes de manera remota y

ejecutan las operaciones correspondientes. El cliente se comunica con el sistema mediante un API Gateway, el cual actúa como punto de entrada centralizado para dirigir las solicitudes al microservicio correspondiente. Esta separación de responsabilidades permite que cada parte del sistema evolucione y escale de manera independiente.

El servidor está compuesto por distintos microservicios especializados que se comunican entre sí para cumplir con las funcionalidades del sistema, como la verificación de disponibilidad de boletos, la gestión de compras, el registro de historial de ventas y la actualización del inventario. Esta estructura modular facilita el mantenimiento del sistema, así como la incorporación de nuevas funcionalidades sin afectar el funcionamiento global.

Implementación de Microservicios con Spring Boot

Cada componente del sistema fue desarrollado como un microservicio independiente utilizando Spring Boot, un framework ligero y robusto para el desarrollo de aplicaciones Java. Entre los microservicios implementados destacan:

Servicio de Inventario: Responsable de llevar el control de los boletos disponibles para distintos eventos.

Servicio de Compras: Encargado de procesar las solicitudes de compra de boletos y de comunicar el resultado al usuario.

Servicio de Historial de Transacciones: (opcional) Registro de las compras realizadas, útil para auditorías o visualización posterior por parte del cliente.

Cada microservicio expone su funcionalidad mediante una API RESTful, permitiendo que otras partes del sistema, incluyendo el cliente, puedan realizar solicitudes mediante HTTP de forma sencilla y estandarizada.

Comunicación RESTful y Serialización de Datos

La interacción entre los componentes del sistema se realizó utilizando el protocolo HTTP a través de endpoints REST. Se utilizaron los métodos GET para la recuperación de información (por ejemplo, consultar boletos disponibles) y POST para la ejecución de acciones (por ejemplo, comprar boletos). Esta comunicación RESTful permite una fácil integración entre microservicios e incluso con sistemas externos en el futuro.

Los datos intercambiados entre servicios y hacia el cliente fueron estructurados en formato JSON, debido a su simplicidad, ligereza y compatibilidad con la mayoría de lenguajes de programación y frameworks modernos. Esto facilitó tanto el desarrollo del frontend como el procesamiento de las solicitudes por parte del backend.

Control de Concurrencia y Acceso Seguro al Inventario

Uno de los principales retos en un sistema distribuido que maneja operaciones simultáneas es garantizar la consistencia de los datos, especialmente cuando múltiples usuarios intentan comprar boletos al mismo tiempo. Para abordar este problema, se implementaron mecanismos de sincronización en el microservicio de inventario.

Se utilizaron métodos sincronizados y bloqueos explícitos para asegurar que las operaciones de lectura y escritura sobre la cantidad de boletos disponibles se realicen de forma atómica. De esta manera, se evita la ocurrencia de condiciones de carrera, donde dos procesos podrían leer el mismo estado y realizar actualizaciones incorrectas, lo que resultaría en inconsistencias como la venta de boletos inexistentes.

Este control también garantiza que las transacciones se procesen en orden y que cada compra refleje correctamente el estado actual del inventario, manteniendo la integridad del sistema.

Balanceo de Carga con Java RMI

Para asegurar la escalabilidad y disponibilidad del sistema, se implementó un balanceador de carga utilizando Java RMI (Remote Method Invocation). Este componente se encargó de distribuir las solicitudes entrantes entre múltiples instancias del microservicio de inventario.

El balanceador se diseñó para registrar múltiples servidores disponibles, y seleccionar de forma dinámica el más adecuado para atender cada solicitud. Se implementaron estrategias sencillas como round-robin o selección aleatoria para el reparto de carga. Además, se contempló la posibilidad de detectar la caída de una instancia del servidor y redirigir la petición a otra disponible, aumentando así la tolerancia a fallos del sistema.

El uso de RMI permitió simular un entorno distribuido realista, donde los clientes pueden acceder a múltiples nodos que comparten la carga de trabajo y ofrecen redundancia frente a fallos individuales.

Interfaz Web para el Usuario Final

Para brindar una experiencia de usuario accesible y moderna, se desarrolló una interfaz web utilizando HTML, CSS y JavaScript. Esta interfaz permite a los usuarios realizar acciones como consultar disponibilidad de boletos, ingresar sus datos personales y completar la compra de boletos de forma intuitiva.

La interfaz web se comunica con el API Gateway del sistema a través de solicitudes HTTP asíncronas (AJAX), lo cual permite una navegación fluida sin recargar la página. Además, se incluyeron validaciones básicas del lado del cliente para garantizar que los datos ingresados sean correctos antes de ser enviados al servidor.

Gracias a esta integración con el backend distribuido, el usuario puede interactuar con un sistema robusto sin notar la complejidad de los procesos que ocurren en segundo plano, lo

cual es uno de los objetivos clave del uso de microservicios: abstraer la complejidad técnica manteniendo una interfaz sencilla.

Desarrollo

En esta práctica se implementó un sistema distribuido para un microservicio de venta de boletos de cine, utilizando el framework Spring Boot y el ecosistema de Spring Cloud. La práctica estuvo compuesta por cinco módulos, cada uno con una función específica dentro de la arquitectura general del proyecto. Estos módulos incluyeron el servidor Eureka (eureka-server), el servicio API Gateway (apigateway), el servicio de administración de boletos (servicioboletos) y el frontend para el cliente (Cliente).

Primeramente, tenemos el servidor Eureka el cual actúa como un service discovery, lo que significa que permite a los diferentes microservicios del sistema registrarse y descubrirse entre sí dinámicamente. Esto es fundamental en arquitecturas de microservicios, ya que permite escalar y mantener los servicios de forma independiente sin necesidad de configurar rutas fijas entre ellos. En este módulo, se utilizó la dependencia `spring-cloud-starter-netflix-eureka-server` y se configuró para que los demás servicios puedan registrarse correctamente.

Después tenemos el API Gateway el cual, por su parte, funcionó como punto de entrada principal al sistema. Su función fue enrutar las solicitudes del cliente a los servicios correspondientes. Para ello, se empleó la dependencia `spring-cloud-starter-gateway`, la cual facilita la creación de filtros, rutas personalizadas, y manejo centralizado de autenticación o control de tráfico. Este gateway también se registró en el servidor Eureka como cliente, utilizando la dependencia `spring-cloud-starter-netflix-eureka-client`.

Posteriormente se realizó el módulo Cliente que desarrolle como una aplicación frontend utilizando también Spring Boot. A pesar de que normalmente un frontend se implementa con tecnologías como Angular, React o Vue, en este caso se optó por construir un cliente simple en Java para mantener la arquitectura uniforme y facilitar pruebas con los servicios. Este módulo también se registró en Eureka y permitió interactuar con los servicios backend a través del API Gateway.

Finalmente, el módulo servicioboletos se encargó de la lógica de negocio relacionada con la gestión de boletos. Incluyó funcionalidades como el registro, consulta y eliminación de boletos. Este servicio se integró con una base de datos, se registró en Eureka y fue expuesto mediante el gateway para que otros módulos pudieran acceder a él. Además, se añadió el starter de actuator para monitorear el estado del servicio y verificar su correcta operación.

apigateway	15/04/2025 01:58 p. m.
Cliente	15/04/2025 02:42 p. m.
eureka-server	15/04/2025 01:58 p. m.
servicioboletos	16/04/2025 09:18 p. m.
serviciocompras	16/04/2025 09:19 p. m.
servicioventas	16/04/2025 01:24 a. m.

Figura 8. Proyectos creados

Para la gestión de dependencias y versiones, todos los módulos utilizaron spring-boot-starter-parent como padre del proyecto, junto con la importación de spring-cloud-dependencies en la sección dependencyManagement, asegurando compatibilidad entre las versiones de los componentes utilizados. Las versiones utilizadas fueron Spring Boot 3.4.4 y Spring Cloud 2024.0.1, lo cual garantizó el uso de funcionalidades modernas y soporte para Java 17.

Para realizar lo antes descrito se tuvo que realizar mediante Spring Initializr (<https://start.spring.io>). Una vez configuradas las opciones y seleccionadas las dependencias, se descargaron los archivos ZIP generados por Spring Initializr. Luego, se descomprimió y se abrió en Visual Studio Code, donde se comenzó la implementación del microservicio, siguiendo una estructura modular y organizada para facilitar su escalabilidad y mantenimiento.

The screenshot shows the Spring Initializr web interface with the following configuration details:

- Project:**
 - ☐ Gradle - Groovy
 - ☐ Gradle - Kotlin
 - ☒ **Java**
 - ☐ Kotlin
 - ☐ Groovy
- Language:**
 - ☒ **Java**
 - ☐ Kotlin
 - ☐ Groovy
- Spring Boot:**
 - ☐ 3.5.0 (SNAPSHOT)
 - ☐ 3.5.0 (M3)
 - ☐ 3.4.5 (SNAPSHOT)
 - ☒ **3.4.4**
 - ☐ 3.3.11 (SNAPSHOT)
 - ☐ 3.3.10
- Project Metadata:**
 - Group:
 - Artifact:
 - Name:
 - Description:
 - Package name:
 - Packaging: ☒ **Jar** ☐ War
 - Java: ☐ 24 ☐ 21 ☒ **17**
- Dependencies:**
 - Spring Web** WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Figura 9. Estructura y creación del proyecto con Spring Boot

Instalación de Maven

Posteriormente se tuvo que instalar Maven para lo cual primero es necesario descargarlo desde la página oficial de Apache Maven, a la cual se puede acceder en el siguiente enlace: <https://maven.apache.org/download.cgi> En esta página, se debe seleccionar y descargar la versión más reciente del archivo binario en formato ZIP.

Una vez descargado el archivo, se procede a extraer su contenido en una ubicación deseada del sistema, por ejemplo, en la ruta C:\Users\user\Desktop\apache-maven-3.9.9-bin\apache-maven-3.9.9. Posteriormente, es necesario configurar la variable de entorno MAVEN_HOME, apuntando a la carpeta donde se extrajo Maven. Además, se debe agregar la carpeta bin de Maven a la variable Path del sistema, lo que permitirá ejecutar comandos de Maven desde la terminal sin necesidad de especificar la ruta completa.

Para verificar que la instalación se realizó correctamente, se abre una terminal y se ejecuta el comando mvn -version. Si la configuración es correcta, el sistema mostrará la versión instalada de Maven junto con la versión de Java detectada.

```
PS C:\Users\Atl1God\Desktop\ESCOM ATL\OCTAVO SEMESTRE\SISTEMAS DISTRIBUIDOS\Practica 6\ServicioWebCine> mvn -version
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfc9d97d260186937)
Maven home: C:\Users\Atl1God\Documents\apache-maven-3.9.9
Java version: 23.0.2, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-23
Default locale: es_MX, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

Figura 10. Verificación de la instalación de Apache Maven

Configuración de la Base de Datos

Primeramente, tuvimos que instalar MySQL, para posteriormente crear una base de datos llamada cine y posteriormente crear las tablas necesarias para poder realizar el sistema adecuadamente.

```
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 55
Server version: 8.0.41 MySQL Community Server - GPL

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE DATABASE cine;
Query OK, 1 row affected (0.01 sec)
```

Figura 11. Creación de la base de datos Cine

<pre>mysql> CREATE TABLE boletos (-> id_boleto INT AUTO_INCREMENT PRIMARY KEY, -> id_evento INT NOT NULL, -> tipo VARCHAR(50) NOT NULL, -> precio DECIMAL(8,2) NOT NULL, -> total_disponibles INT NOT NULL, -> FOREIGN KEY (id_evento) REFERENCES eventos(id_eve nto) ->); Query OK, 0 rows affected (0.03 sec)</pre>	<pre>mysql> CREATE TABLE compras (-> id_compra INT AUTO_INCREMENT PRIMARY KEY, -> id_usuario INT NOT NULL, -> fecha_compra TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -> total_pagado DECIMAL(8,2) NOT NULL, -> FOREIGN KEY (id_usuario) REFERENCES usuarios(id_u suario) ->); Query OK, 0 rows affected (0.03 sec)</pre>
---	--

Figura 12. Tablas creadas en la base de datos

Una vez completados estos pasos, se procedió con el desarrollo del backend, lo cual se detalla en las siguientes secciones.

Eurekaserver

En primera instancia tenemos el servidor eurekaserver, este se implementó utilizando Spring Boot y Spring Cloud Netflix Eureka. Este servidor cumple con la función de ser el registro central de microservicios, es decir, un punto donde los distintos servicios del sistema pueden registrarse y descubrirse entre sí de forma dinámica. Esto es fundamental en nuestra arquitectura basada en microservicios, ya que permite que los servicios se comuniquen sin necesidad de conocer las direcciones IP o puertos específicos de los demás componentes.

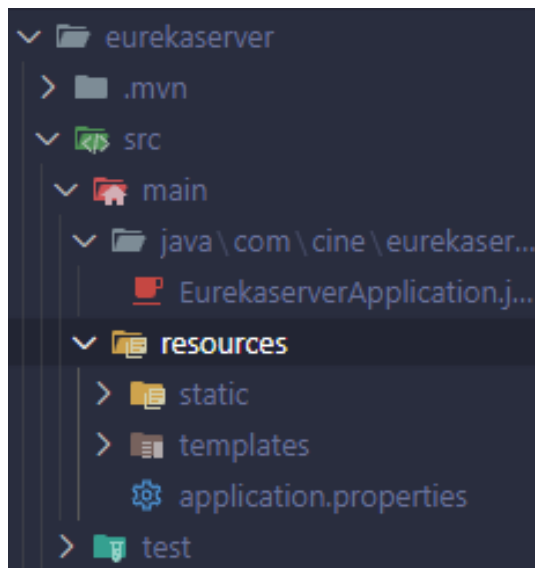


Figura 13. Estructura del proyecto eurekaserver

El archivo EurekaserverApplication.java contiene la clase principal del proyecto. Esta clase está anotada con @SpringBootApplication, ya que es una aplicación de Spring Boot, y con @EnableEurekaServer, el cual habilita el comportamiento de servidor Eureka. Esto significa que, al ejecutar esta clase, se levanta un servidor web en el puerto especificado y comienza a

funcionar como un centro de registro de servicios. La ejecución se inicia con el método main, que llama a `SpringApplication.run()` para lanzar la aplicación.

```
package com.cine.eurekaserver;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@EnableEurekaServer
public class EurekaserverApplication {

    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(primarySource:EurekaserverApplication.class, args);
    }
}
```

Figura 14. Clase principal del servidor Eureka

Por otro lado, el archivo `application.properties` contiene la configuración básica del servidor Eureka. En donde definimos el nombre del servicio como `eurekaserver` mediante la propiedad `spring.application.name`, y se especifica que se ejecutará en el puerto 8761. Además, se configuran dos propiedades importantes: `eureka.client.register-with-eureka=false` y `eureka.client.fetch-registry=false`. Estas líneas indican que este servidor no actuará como un cliente de Eureka (es decir, no se registrará a sí mismo ni consultará el registro), ya que su propósito es únicamente administrar el registro para los demás servicios del sistema.

```
spring.application.name=eurekaserver
server.port=8761

#Eureka server configuracion
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Figura 15. Configuración del servidor Eureka

Serviciocompras

En el desarrollo del microservicio `serviciocompras`, implementé una solución orientada a la gestión de la compra de boletos para funciones de cine, dentro de una arquitectura basada en microservicios. Este módulo se encarga de procesar las solicitudes de compra enviadas por los clientes, validar la disponibilidad de boletos mediante una consulta al servicio cartelera, y registrar la compra en caso de que existan suficientes lugares disponibles.

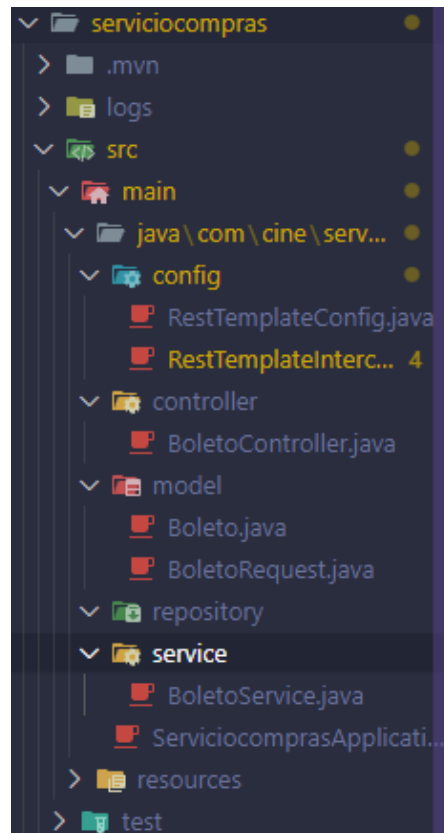


Figura 16. Estructura del proyecto serviciocompras

La clase principal del proyecto se encuentra en el archivo `ServiciocomprasApplication.java`. Esta clase, anotada con `@SpringBootApplication`, permite iniciar el microservicio en el puerto 8082. En el archivo `application.properties` configuré parámetros esenciales, como el nombre del servicio (`spring.application.name=serviciocompras`), la conexión con el servidor Eureka para el descubrimiento de servicios, así como la conexión a la base de datos MySQL. También se definió una configuración para guardar los logs generados por este servicio en el archivo `compra-service.log`, lo cual es útil para rastrear y depurar el comportamiento del sistema.

```
spring.application.name=serviciocompras
# Configuración de Eureka
server.port=8082
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
# Configuración de Spring Cloud Config
#Conexión a la base de datos
spring.datasource.url=jdbc:mysql://localhost:3306/cine
spring.datasource.username=root
spring.datasource.password=Atl1God$
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
# Configuración de JPA
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
# Guardar logs en archivo
logging.file.name=logs/serviciocompras.log
logging.level.root=INFO
logging.level.com.panaderia=DEBUG
```

Figura 17. Configuración del servicio compras

El componente encargado de atender las solicitudes HTTP es el `BoletoController`. En este controlador se define el endpoint `/boletos`, que escucha peticiones POST. Cuando se recibe una solicitud con los datos de compra (cliente, película y número de boletos), estos se envían al servicio `BoletoService` para ser procesados. El controlador es simple, ya que delega toda la lógica al servicio correspondiente, siguiendo el principio de responsabilidad única.

```
@RestController
@RequestMapping("/boletos")
public class BoletoController {

    private final BoletoService boletoService;

    @Autowired
    public BoletoController(BoletoService boletoService) {
        this.boletoService = boletoService;
    }

    @PostMapping
    public String comprarBoletos(@RequestBody BoletoRequest boletoRequest) {
        return boletoService.procesarCompra(boletoRequest);
    }
}
```

Figura 18. Controlador del servicio compras

Dentro del paquete `service`, la clase `BoletoService` implementa la lógica principal del microservicio. A través de la clase `RestTemplate`, se realizan llamadas HTTP al microservicio `cartelera`, que contiene la información sobre la disponibilidad de boletos. Primero se consulta cuántos boletos hay disponibles para una película; si hay suficientes, se ejecuta una llamada PUT para registrar la compra, y luego se vuelve a consultar la cantidad disponible para mostrarla al usuario. Toda esta información se almacena en logs usando `SLF4J`, lo que me permite mantener trazabilidad sobre las acciones realizadas por los clientes.

```
public String procesarCompra(BoletoRequest boletoRequest) {
    StringBuilder resultado = new StringBuilder(str:"Resultados de compra:\n");

    for (Boleto boleto : boletoRequest.getBoletos()) {
        String pelicula = boleto.getPelicula();
        int boletosDeseados = boleto.getCantidad();

        // Obtener disponibilidad actual desde cartelera-service
        Integer boletosDisponibles = restTemplate.getForObject(
            CARTELERA_URL + "/disponibles/" + pelicula,
            Integer.class
        );

        if (boletosDisponibles != null && boletosDisponibles >= boletosDeseados) {
            // Realizar la compra
            restTemplate.put(
                CARTELERA_URL + "/comprar/" + pelicula + "/" + boletosDeseados,
                request:null
            );

            // Consultar boletos restantes después de la compra
            Integer boletosRestantes = restTemplate.getForObject(
                CARTELERA_URL + "/disponibles/" + pelicula,
                Integer.class
            );
        }
    }
}
```

Figura 19. Clase del servicio del servicio compras

Para permitir que las peticiones HTTP entre microservicios contengan información de seguimiento (por ejemplo, un ID de correlación), se configuró un interceptor personalizado llamado `RestTemplateInterceptor`. Este interceptor captura el encabezado `X-Correlation-ID` de las solicitudes entrantes y lo añade a todas las llamadas salientes hechas por el `RestTemplate`. Esta configuración se define en la clase `RestTemplateConfig`, ubicada en el paquete `config`. Así se facilita el monitoreo y seguimiento de las peticiones a lo largo del sistema distribuido.

En cuanto al modelo de datos, dentro del paquete `model` se encuentran dos clases: `Boleto`, que representa un boleto con los atributos `pelicula` y `cantidad`, y `BoletoRequest`, que agrupa una lista de boletos junto con el nombre del cliente. Estas clases se utilizan para mapear automáticamente los datos JSON que se reciben desde el frontend o desde otros servicios.

```
public class Boleto {
    private String pelicula;
    private int cantidad;

    public Boleto() {}

    public String getPelicula() {
        return pelicula;
    }

    public void setPelicula(String pelicula) {
        this.pelicula = pelicula;
    }

    public int getCantidad() {
        return cantidad;
    }

    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }
}
```

Figura 20. Modelo del servicio compras

También se configuró la política de CORS (Cross-Origin Resource Sharing) a través de la clase `CorsConfig`. Esta clase permite que el servicio acepte solicitudes desde cualquier origen (`allowedOrigins("*")`) y que soporte métodos HTTP como GET, POST, PUT y DELETE. Esta configuración es especialmente útil ya que nuestro frontend está alojado en otro dominio o puerto, ya que se basó en otras aplicaciones modernas con arquitectura de microservicios.


```

@Configuration
public class CorsConfig {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(@NonNull CorsRegistry registry) {
                registry.addMapping(pathPattern: "**")
                    .allowedOrigins(...origins:"**")
                    .allowedMethods(...methods:"GET", "POST", "PUT", "DELETE");
            }
        };
    }
}

```

Figura 21. CorsConfig del servicio compras.

Servicioboletos

En este proyecto desarrollé un servicio web RESTful utilizando Spring Boot que simula la gestión de boletos para películas en cartelera. El servicio permite consultar la lista de películas disponibles, verificar cuántos boletos quedan para una película específica y realizar compras de boletos. Además, el sistema está conectado a una base de datos MySQL donde se almacena la información del inventario de boletos.

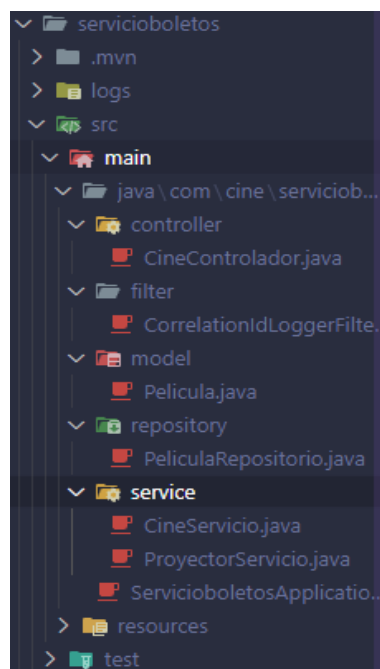


Figura 22. Estructura del proyecto servicioboletos

Comencé creando el modelo de datos, representado por la clase `Pelicula`, que está mapeada a una tabla llamada `inventario` dentro de la base de datos. Esta clase contiene atributos como `id`, `titulo` y `boletosDisponibles`, y usa anotaciones de JPA para que Spring pueda gestionar automáticamente la persistencia de los datos.

```

@Entity
@Table(name = "inventario")
public class Pelicula {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String titulo;
    private int boletosDisponibles;

    // Getters y Setters
    public Long getId() { return id; }
    public String getTitulo() { return titulo; }
    public void setTitulo(String titulo) { this.titulo = titulo; }

    public int getBoletosDisponibles() { return boletosDisponibles; }
    public void setBoletosDisponibles(int boletosDisponibles) { this.b
}

```

Figura 23. Modelo de datos del servicio boletos

Luego implementé el repositorio PeliculaRepositorio, el cual extiende JpaRepository y me permite acceder de forma sencilla a la base de datos. Agregué un método personalizado findByTitulo para poder buscar una película por su nombre, lo cual es fundamental para la lógica del sistema.

```

package com.cine.servicioboletos.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.cine.servicioboletos.model.Pelicula;

public interface PeliculaRepositorio extends JpaRepository<Pelicula, Long> {
    Pelicula findByTitulo(String titulo);
}

```

Figura 24. Repositorio del servicio boletos

La lógica del negocio está en la clase CineServicio. Aquí definí métodos para obtener todas las películas, verificar los boletos disponibles y comprar boletos. También agregué un método llamado reponerBoletos que permite aumentar la cantidad de boletos para una película en caso de que se necesite reabastecer.

```

@Service
public class CineServicio {

    @Autowired
    private PeliculaRepositorio peliculaRepositorio;

    public List<Pelicula> obtenerPeliculas() {
        return peliculaRepositorio.findAll();
    }

    public int obtenerBoletosDisponibles(String titulo) {
        Pelicula pelicula = peliculaRepositorio.findByTitulo(titulo);
        if (pelicula != null) {
            return pelicula.getBoletosDisponibles();
        } else {
            return 0;
        }
    }

    public boolean comprarBoletos(String titulo, int cantidad) {
        Pelicula pelicula = peliculaRepositorio.findByTitulo(titulo);
        if (pelicula != null && pelicula.getBoletosDisponibles() >= cantidad) {
            pelicula.setBoletosDisponibles(pelicula.getBoletosDisponibles() - cantidad);
            peliculaRepositorio.save(pelicula);
            return true;
        } else {
            return false;
        }
    }
}

```

Figura 25. Lógica del servicio boletos

Para darle más funcionalidad al sistema, creé un servicio adicional llamado `ProyectorServicio`, que utiliza anotaciones de Spring para programar tareas automáticas. En este caso, cada 100 segundos (aunque originalmente eran 10 minutos), se reabastecen 50 boletos para una lista predeterminada de películas. Esto se realiza con la ayuda de la anotación `@Scheduled`, que automatiza el proceso de reposición y registra la acción mediante logs.

```

@Service
public class ProyectorServicio {

    private static final Logger logger = LoggerFactory.getLogger(ProyectorServicio.class);

    private final CineServicio cineServicio;

    public ProyectorServicio(CineServicio cineServicio) {
        this.cineServicio = cineServicio;
    }

    // Reponer boletos cada 10 minutos (100000 ms)
    @Scheduled(fixedRate = 100000)
    public void reponerBoletosPeliculas() {
        List<String> peliculas = Arrays.asList(
            "El Padrino", "Titanic", "Inception", "Interstellar",
            "Avengers", "Matrix", "Parasite"
        );

        for (String titulo : peliculas) {
            cineServicio.reponerBoletos(titulo, cantidad:50);
            int disponibles = cineServicio.obtenerBoletosDisponibles(titulo);
            logger.info(format:"Proyector reabastecido - Se añadieron 50 boletos a la película: %s", titulo, disponibles);
        }
    }
}

```

Figura 26. Clase `ProyectorServicio` del servicio boletos

El controlador CineControlador expone los endpoints REST para interactuar con el sistema. Por ejemplo, /cartelera/peliculas devuelve todas las películas disponibles, /cartelera/boletos/{titulo} permite consultar cuántos boletos quedan, y /cartelera/comprar/{titulo}/{cantidad} permite realizar una compra. Las respuestas son en formato JSON y la interacción es sencilla tanto para pruebas con herramientas como Postman, como para integraciones con otras aplicaciones.

```
@RestController
@RequestMapping("/cartelera")
public class CineControlador {

    @Autowired
    private CineServicio cineServicio;

    @GetMapping("/peliculas")
    public List<Película> obtenerPeliculas() {
        return cineServicio.obtenerPeliculas();
    }

    @GetMapping("/boletos/{titulo}")
    public int obtenerBoletosDisponibles(@PathVariable String titulo) {
        return cineServicio.obtenerBoletosDisponibles(titulo);
    }

    @PutMapping("/comprar/{titulo}/{cantidad}")
    public String comprarBoletos(@PathVariable String titulo, @PathVariable int cantidad) {
        boolean exito = cineServicio.comprarBoletos(titulo, cantidad);
        return exito ? "Compra realizada exitosamente" : "No hay suficientes boletos disponi
    }
}
```

Figura 27. Controlador del servicio boletos

Además, implementé un filtro HTTP personalizado (CorrelationIdLoggerFilter) que intercepta todas las solicitudes entrantes para registrar el encabezado X-Correlation-ID si está presente. Esto es útil para rastrear peticiones específicas en los logs y mejorar el diagnóstico o la trazabilidad de solicitudes en entornos distribuidos.

Finalmente, toda la aplicación está configurada para levantarse en el puerto 8081 y está registrada en un servidor Eureka como parte de una arquitectura de microservicios. La conexión a la base de datos MySQL se configura desde el archivo application.properties, donde también habilité el logeo de consultas SQL y la escritura automática de logs a un archivo.

```

spring.application.name=servicioboletos
server.port=8081
# Configuración de Eureka
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
# Configuración de la base de datos
spring.datasource.url=jdbc:mysql://localhost:3306/cine
spring.datasource.username=root
spring.datasource.password=Atl1God$
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
# Configuración de JPA
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
# Guardar logs en archivo
logging.file.name=logs/servicioboletos.log
logging.level.root=INFO
logging.level.com.panaderia=DEBUG

```

Figura 28. Configuración del servicio boletos

Apigateway

Durante el desarrollo del proyecto del sistema de cine, implementé un microservicio llamado apigateway, el cual cumple la función de servir como punto de entrada único a todos los servicios internos registrados en Eureka. Este gateway permite gestionar de forma eficiente las peticiones que llegan al sistema, enrutar dinámicamente a los microservicios correspondientes y aplicar filtros globales como parte de una arquitectura basada en Spring Cloud Gateway.

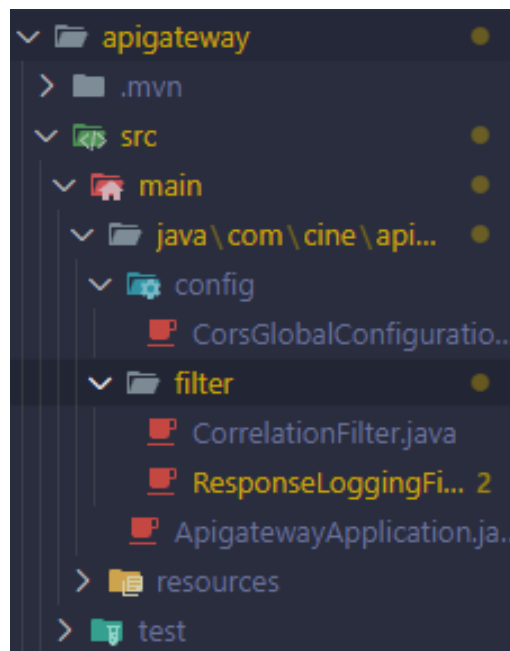


Figura 29. Estructura del proyecto Apigateway

Para comenzar, la clase principal `ApigatewayApplication` es la responsable de arrancar el gateway mediante la anotación `@SpringBootApplication`. En el archivo de propiedades, configuré el nombre del servicio como `apigateway`, establecí el puerto 8083 como el punto de escucha, y activé el registro con Eureka para permitir la detección automática de otros servicios. Además, definí rutas dinámicas para redirigir las solicitudes entrantes hacia los servicios `servicioboletos`, `serviciocompras` y `servicioventas`, cada uno filtrando rutas específicas (`/boletos/**`, `/compras/**` y `/ventas/**`, respectivamente).

```
spring.application.name=apigateway
server.port=8083
# Registro en Eureka
# eureka.client.service-url.defaultZone=http://localhost:8761/eureka
# Lo uso para usarlo con docker-compose
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.instance.prefer-ip-address=true
# Rutas dinámicas usando el nombre registrado en Eureka
spring.cloud.gateway.routes[0].id=boletos
spring.cloud.gateway.routes[0].uri=lb://servicioboletos
spring.cloud.gateway.routes[0].predicates=Path=/inventario/**

spring.cloud.gateway.routes[1].id=compras
spring.cloud.gateway.routes[1].uri=lb://serviciocompras
spring.cloud.gateway.routes[1].predicates=Path=/compras/**
```

Figura 30. Configuración del Apigateway

Uno de los aspectos clave fue la configuración de CORS. Implementé una clase llamada `CorsGlobalConfiguration` que define una política de CORS global para permitir el acceso desde `http://localhost:8084`, lo cual es útil cuando se trabaja con una interfaz gráfica o cliente web que se ejecuta en otro puerto. Esta configuración permite cualquier cabecera y método, y además activa el uso de credenciales en las solicitudes.

```
@Configuration
public class CorsGlobalConfiguration {

    @Bean
    public CorsWebFilter corsWebFilter() {
        CorsConfiguration config = new CorsConfiguration();
        config.addAllowedOrigin(origin:"http://localhost:8084");
        config.addAllowedHeader(allowedHeader:"*");
        config.addAllowedMethod(method:"*");
        config.setAllowCredentials(allowCredentials:true);

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration(path:"/**", config);

        return new CorsWebFilter(source);
    }
}
```

Figura 31. Políticas CORS globales

Adicionalmente, desarrollé dos filtros globales personalizados. El primero es el `CorrelationFilter`, que se encarga de generar un identificador único (X-Correlation-ID) para cada solicitud entrante. Este ID es útil para rastrear las solicitudes a lo largo del sistema distribuido, facilitando así el monitoreo y la depuración. Cada vez que se recibe una petición, este filtro añade el identificador al encabezado y lo registra en los logs.

El segundo filtro es el `ResponseLoggingFilter`, el cual intercepta todas las respuestas que el gateway envía a los clientes. Su función principal es registrar el contenido de dichas respuestas en la consola. Esto me permitió ver de forma clara qué datos regresaban los servicios, y resultó muy útil durante la etapa de pruebas para detectar errores o verificar que la respuesta fuera la esperada. El filtro utiliza un `ServerHttpResponseDecorator` para envolver la respuesta y leer su contenido antes de enviarlo al cliente.

```
@Component
public class ResponseLoggingFilter implements GlobalFilter, Ordered {

    private static final Logger log = LoggerFactory.getLogger(clazz:ResponseLoggingFilter.class);

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, org.springframework.cloud.gateway.filter
        ServerHttpResponse originalResponse = exchange.getResponse();
        ServerHttpRequest request = exchange.getRequest();

        ServerHttpResponseDecorator decoratedResponse = new ServerHttpResponseDecorator(originalResponse);

        @Override
        public Mono<Void> writeWith(org.reactivestreams.Publisher<? extends DataBuffer> body) {
            if (body instanceof Flux) {
                Flux<? extends DataBuffer> fluxBody = (Flux<? extends DataBuffer>) body;

                return super.writeWith(fluxBody.map(dataBuffer -> {
                    byte[] content = new byte[dataBuffer.readableByteCount()];
                    dataBuffer.read(content);
                    DataBufferUtils.release(dataBuffer);

                    String responseBody = new String(content, StandardCharsets.UTF_8);

                    log.info(format: " 📄 Respuesta al cliente: [{ } {}] -> { }",
                        request.getMethod(), request.getURI(), responseBody);

                    return new DefaultDataBufferFactory().wrap(content);
                }));
            }
        }
    }
}
```

Figura 32. Filtro implementado en el apigateway

En conjunto, este apigateway me permitió centralizar la entrada al sistema de cine, gestionar la comunicación entre microservicios de forma eficiente y facilitar la depuración con filtros personalizados. También preparé el entorno para que fuera compatible tanto en ejecución local como en despliegue con Docker, asegurando así su escalabilidad y fácil mantenimiento.

Cliente

Como parte del desarrollo de un sistema web distribuido para la compra de boletos de cine, implementé un cliente web que se comunica con un API Gateway para consumir los servicios del backend. Este cliente está compuesto por un controlador en Spring Boot, archivos HTML, CSS y JavaScript ubicados en la carpeta static, que es servida como contenido estático por el servidor de Spring.

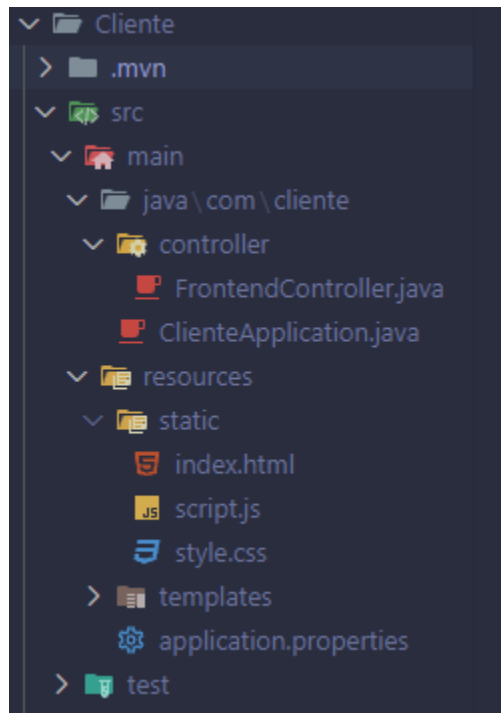


Figura 33. Estructura del servicio del Cliente web

El archivo `FrontendController.java`, dentro del paquete `com.cliente.controller`, tiene la única función de redirigir las peticiones que llegan a la raíz (/) del servidor hacia el archivo `index.html` ubicado en la carpeta `static`. Esto se logra mediante la anotación `@GetMapping("/")` y el método `index()` que retorna `"forward:/index.html"`. Esta técnica es muy útil cuando se manejan aplicaciones web frontend sin usar plantillas de Spring como Thymeleaf, ya que nos permite servir directamente un frontend estático como si fuera una SPA (Single Page Application).

```
package com.cliente.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class FrontendController {

    @GetMapping("/")
    public String index() {
        return "forward:/index.html"; // Esto redirige a "
    }
}
```

Figura 34. Clase `FrontendController` para la interfaz de usuario

El archivo index.html es la interfaz principal que ve el usuario. En ella se presenta una bienvenida al sitio "CineBoleto", permitiendo visualizar la cartelera actual, ingresar el nombre del cliente, y seleccionar la cantidad de boletos que desea comprar por cada película. Esta vista se apoya en hojas de estilo (style.css) para lograr un diseño moderno y visualmente atractivo, utilizando tipografías de Google Fonts, un fondo degradado, y componentes con sombras y bordes redondeados para dar una experiencia más amigable al usuario.

The screenshot displays the 'CineBoleto' web application interface. At the top, a red banner contains the text '¡Bienvenido a CineBoleto!' and a subtitle 'Compra tus boletos fácilmente y sin hacer filas'. Below this, a section titled 'Cartelera:' lists eight movies with their available ticket counts and a corresponding input field for the number of tickets to purchase. The movies listed are Avengers: Endgame (8 tickets), El Padrino (20 tickets), Titanic (20 tickets), Inception (20 tickets), Interestelar (20 tickets), Avengers (20 tickets), Matrix (20 tickets), and Parasite (20 tickets). At the bottom, a section titled 'Datos de la Compra:' includes a text input field for the user's name and a purple button labeled 'Comprar Boletos'.

Movimiento	Boletos disponibles	Cantidad a comprar
Avengers: Endgame	8	0
El Padrino	20	0
Titanic	20	0
Inception	20	0
Interestelar	20	0
Avengers	20	0
Matrix	20	0
Parasite	20	0

Datos de la Compra:

Ingresa tu nombre

[Comprar Boletos](#)

Figura 35. Frontend del servicio Cliente

Finalmente, el archivo style.css se encarga de toda la parte visual. Define estilos globales como la fuente, colores de fondo, márgenes y espaciado. También personaliza los elementos del formulario, las tarjetas de películas, botones, y el mensaje de confirmación. La combinación de colores, fuentes y sombras ayuda a que la aplicación tenga una apariencia profesional y moderna, sin perder la simplicidad.

```

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

body {
  font-family: 'Montserrat', sans-serif;
  background: linear-gradient(to right,
    color: #2c3e50;
  min-height: 100vh;
}

.container {
  max-width: 900px;
  margin: 0 auto;
  padding: 30px 20px;
}

```

Figura 36. Fragmento del código style.css

En el archivo script.js desarrollé la lógica del cliente para conectar la interfaz de usuario con el backend mediante una API REST. Lo primero que hice fue definir la constante API_URL con el valor `http://localhost:8083`, ya que en ese puerto se encuentra corriendo el API Gateway, el cual se encarga de enrutar las solicitudes al microservicio correspondiente.

```
const API_URL = "http://localhost:8083"; // API Gateway
```

Figura 37. Conexión con el API GATEWAY

Después, creé una función asíncrona llamada `obtenerPelículas()`, que se encarga de hacer una solicitud GET al endpoint `/inventario/peliculas` para obtener la lista de películas disponibles en cartelera, junto con la cantidad de boletos disponibles para cada una. Esta información se recupera usando `fetch` y, si la solicitud es exitosa (`response.ok`), se convierte la respuesta en formato JSON y se pasa a la función `mostrarPelículas(peliculas)`, la cual es la encargada de actualizar visualmente el HTML y mostrar dinámicamente la cartelera.

```
// Obtener la cartelera y la disponibilidad de boletos
async function obtenerPelículas() {
  try {
    const response = await fetch(`${API_URL}/inventario/películas`);
    if (response.ok) {
      const películas = await response.json();
      mostrarPelículas(películas);
    } else {
      document.getElementById("películas").innerText = "Error al ob
    }
  } catch (error) {
    document.getElementById("películas").innerText = "No se pudo con
  }
}
```

Figura 38. Función asíncrona para obtener la cartelera

La función `mostrarPelículas(películas)` recorre cada objeto de la lista de películas que se recibe y por cada uno crea un nuevo div dentro del contenedor `#películas`. Dentro de este div, se genera un label con el título de la película y los boletos disponibles, además de un input tipo number para que el usuario pueda seleccionar cuántos boletos desea comprar. Este número está limitado por el máximo de boletos disponibles que indica el backend.

```
// Mostrar las películas en el frontend
function mostrarPelículas(películas) {
  const películasDiv = document.getElementById("películas");
  películasDiv.innerHTML = ''; // Limpiar el contenedor antes

  películas.forEach((película) => {
    const películaDiv = document.createElement("div");
    películaDiv.classList.add("producto");

    películaDiv.innerHTML = `
      <label for="cantidad-${película.id}">${película.títu
      <input type="number" id="cantidad-${película.id}" m
    `;

    películasDiv.appendChild(películaDiv);
  });
}
```

Figura 39. Función para mostrar las películas en el frontend

Luego implementé la función `comprarBoletos()`, que se ejecuta cuando el usuario hace clic en el botón "🛒 Comprar Boletos". Primero valida que el campo del nombre del cliente (`#nombreCliente`) no esté vacío. Si está vacío, se muestra un mensaje pidiendo que se introduzca un nombre. Si hay un nombre, se crea un arreglo llamado `compras`, el cual se va llenando con los títulos de las películas seleccionadas y la cantidad de boletos que el usuario

eligió. Para cada película, uso su input asociado para leer la cantidad seleccionada y, si es mayor a cero, la agrego al arreglo.

```
// Realizar la compra de boletos
async function comprarBoletos() {
    const nombre = document.getElementById("nombreCliente").value;

    if (!nombre) {
        document.getElementById("mensaje").innerText = "Por favor, ingresa tu nombre.";
        return;
    }

    const compras = [];
    const peliculas = document.querySelectorAll("#peliculas div");

    peliculas.forEach((peliculaDiv) => {
        const peliculaId = peliculaDiv.querySelector("input").id.split("-")[1];
        const cantidad = document.getElementById(`cantidad-${peliculaId}`).value;

        if (cantidad > 0) {
            compras.push({
                titulo: peliculaDiv.querySelector("label").innerText.split(" - ")[0].trim(),
                cantidad: parseInt(cantidad),
            });
        }
    });
};
```

Figura 40. Función asíncrona para comprar Boletos

Posteriormente, por cada compra dentro del arreglo compras, se hace una solicitud PUT al backend con la ruta `/inventario/comprar/{titulo}/{cantidad}`, que se construye dinámicamente según la película y la cantidad de boletos. Si alguna de estas solicitudes falla, se muestra un mensaje de error indicando que no se pudo procesar la compra de esa película.

```
const data = {
    nombreCliente: nombre,
    boletos: compras,
};

for (const compra of compras) {
    const response = await fetch(`${API_URL}/inventario/comprar/${encodeURIComponent(compra.titulo)}/${compra.cantidad}`, {
        method: "PUT",
    });

    if (!response.ok) {
        document.getElementById("mensaje").innerText = "Error al procesar la compra por la película " + compra.titulo;
        return;
    }
}
```

Figura 41. Solicitud para realizar compras

Finalmente, si todas las compras fueron exitosas, se vuelve a ejecutar la función `obtenerPeliculas()` para refrescar la disponibilidad de boletos y reflejar los cambios en

pantalla, permitiendo así que el usuario vea en tiempo real cuántos boletos quedan después de su compra.

```
// Refrescar disponibilidad
obtenerPelículas();
}

// Inicializar cartelera
obtenerPelículas();
```

Figura 42. Función para Refrescar disponibilidad e iniciar cartelera

Gracias al uso de fetch y la sintaxis async/await, el archivo script.js permite manejar las respuestas del servidor de manera asincrónica, manteniendo una interfaz fluida, sin recargar la página ni bloquear la interacción. El sistema también está preparado para responder ante errores de conexión o fallos en el backend, lo que mejora la experiencia de usuario con mensajes claros y acciones automáticas como la recarga del stock actualizado tras cada compra.

Nota: Es importante destacar que la estructura generada por Spring Boot incluye múltiples archivos adicionales que son fundamentales para el correcto funcionamiento de la aplicación. Entre estos archivos se encuentran configuraciones clave del framework, archivos de pruebas automatizadas (tests), y componentes necesarios para el manejo de seguridad, dependencias, y entorno de ejecución. Estos elementos son generados automáticamente al crear un proyecto nuevo con Spring Boot, con el fin de facilitar el desarrollo y garantizar una base sólida y funcional para la aplicación.

Durante el desarrollo del sistema distribuido para la gestión de boletos de cine, se presentaron algunos inconvenientes técnicos al momento de instalar las dependencias y generar el archivo .jar del proyecto. En particular, se detectaron errores durante el proceso de compilación relacionados con los archivos de prueba generados por defecto, como la clase ApplicationTests. Estos errores impedían la correcta construcción del proyecto y el empaquetado de los microservicios.

Para poder avanzar con el empaquetado y despliegue de los microservicios, fue necesario deshabilitar temporalmente estos archivos de prueba. Esta decisión permitió completar satisfactoriamente el proceso de build del proyecto y continuar con la ejecución y pruebas funcionales del sistema distribuido de cine, incluyendo la comunicación entre el cliente web, el API Gateway y los servicios encargados del manejo de cartelera y compras de boletos.

No obstante, en este documento no se profundiza en la explicación de estos archivos adicionales, ya que su objetivo principal es brindar soporte al entorno de desarrollo, facilitar pruebas automatizadas y mantener configuraciones predeterminadas del framework. El

enfoque de esta explicación se ha centrado en los archivos, clases y funcionalidades que tienen un impacto directo en el comportamiento y funcionamiento del microservicio de cine, especialmente en lo relacionado con la compra de boletos y la visualización dinámica de la cartelera.

Instrucciones para ejecutar el proyecto

Para poder ejecutarlo una vez que tengamos todos nuestros archivos `application.properties` configurados correctamente como se mostro anteriormente tendremos primeramente que iniciar el servicio EurekaServer es por ello que nos vamos a la ruta del proyecto y ejecutamos el comando `./mvnw spring-boot:run`, lo cual hará que se inicie como se muestra en la siguiente imagen:

```
PS C:\Users\Atl1God\Desktop\ESCOM ATL\OCTAVO SEMESTRE\SISTEMAS DISTRIBUIDOS\Practica 7\eurekaser
ver> ./mvnw spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.cine:eurekaServer >-----
[INFO] Building eurekaServer 0.0.1-SNAPSHOT
[INFO]    from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot:3.4.4:run (default-cli) > test-compile @ eurekaServer >>>
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ eurekaServer ---
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO] Copying 0 resource from src\main\resources to target\classes
[INFO]
[INFO] --- compiler:3.13.0:compile (default-compile) @ eurekaServer ---
```

Figura 43. Ejecución del proyecto EurekaServer

Posteriormente procederemos a ejecutar los servicios boletos y compras, para que los pueda registrar eureka server, a continuación, se muestra la ejecución de dichos servicios:

```
PS C:\Users\Atl1God\Desktop\ESCOM ATL\OCTAVO SEMESTRE\SISTEMAS DISTRIBUIDOS\Practica 7\serviciob
oletos> ./mvnw spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.cine:servicioboletos >-----
[INFO] Building servicioboletos 0.0.1-SNAPSHOT
[INFO]    from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot:3.4.4:run (default-cli) > test-compile @ servicioboletos >>>
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ servicioboletos ---
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO] Copying 0 resource from src\main\resources to target\classes
```

Figura 44. Ejecución del proyecto servicioboletos

```

PS C:\Users\Atl1God\Desktop\ESCOM ATL\OCTAVO SEMESTRE\SISTEMAS DISTRIBUIDOS\Practica 7\serviciocompras> ./mvnw spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.cine:serviciocompras >-----
[INFO] Building serviciocompras 0.0.1-SNAPSHOT
[INFO]    from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot:3.4.4:run (default-cli) > test-compile @ serviciocompras >>>
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ serviciocompras ---
[INFO] Copying 1 resource from src\main\resources to target\classes
[INFO] Copying 0 resource from src\main\resources to target\classes

```

Figura 45. Ejecución del proyecto serviciocompras

Por último, ejecutamos el API Gateway que es un componente fundamental en una arquitectura de microservicios, ya que actúa como un punto único de entrada para todas las solicitudes que realizan los clientes hacia los distintos servicios que conforman una aplicación. En lugar de que el cliente tenga que conocer la dirección y el puerto de cada microservicio (como el de boletos o compras), todas las peticiones pasan primero por el gateway, que se encarga de redirigirlas al microservicio correspondiente. Esto simplifica enormemente la comunicación entre el cliente y el backend.

Una de las funciones más importantes del API Gateway es el enrutamiento de peticiones. A través de configuraciones definidas, el gateway detecta qué ruta está solicitando el cliente (por ejemplo, /boletos/) y reenvía esa solicitud al microservicio adecuado (como servicioboletos). Este enrutamiento se hace de forma dinámica utilizando servicios de descubrimiento como Eureka, lo cual también permite que el gateway realice balanceo de carga entre múltiples instancias de un mismo microservicio, distribuyendo equitativamente las peticiones.

El gateway también permite aplicar filtros personalizados, que pueden ser utilizados para registrar logs, modificar encabezados HTTP, validar datos o controlar el acceso a determinadas rutas. Asimismo, proporciona mecanismos de manejo de errores, capturando fallas de los servicios internos y devolviendo mensajes de error más claros o incluso redirigiendo a rutas alternativas (fallbacks).

También, el API Gateway es útil para configurar políticas como CORS (que controla qué dominios pueden consumir la API) y limitación de peticiones (rate limiting), ayudando a proteger los servicios de posibles abusos o sobrecargas. A continuación, se muestra la ejecución del apigateway.

```

PS C:\Users\At11God\Desktop\ESCOM ATL\OCTAVO SEMESTRE\SISTEMAS DISTRIBUIDOS\Practica 7\apigateway
y> ./mvnw spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.cine:apigateway >-----
[INFO] Building apigateway 0.0.1-SNAPSHOT
[INFO] from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot:3.4.4:run (default-cli) > test-compile @ apigateway >>>
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ apigateway ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO]
[INFO] --- compiler:3.13.0:compile (default-compile) @ apigateway ---

```

Figura 46. Ejecución del proyecto Apigateway

Por último, ejecutamos nuestro Proyecto Cine, el cual es nuestro frontend en este caso, aunque si no se tiene el frontend se pueden realizar peticiones desde postman o insomnia, a continuación, se muestra la ejecución del frontend y el cómo se ve:

```

PS C:\Users\At11God\Desktop\ESCOM ATL\OCTAVO SEMESTRE\SISTEMAS DISTRIBUIDOS\Practica 7\Cliente>
./mvnw spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.ciente:Cliente >-----
[INFO] Building Cliente 0.0.1-SNAPSHOT
[INFO] from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot:3.4.4:run (default-cli) > test-compile @ Cliente >>>
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ Cliente ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 3 resources from src/main/resources to target/classes
[INFO]

```

Figura 47. Ejecución del proyecto Cine

Figura 48. Frontend del servicio Cliente

Resultados

En el proceso de desarrollo del sistema distribuido de cine implementado mediante microservicios ha demostrado un funcionamiento correcto, estable y eficiente. La arquitectura construida con Spring Boot, Eureka, MySQL y un cliente web responsivo permitió validar que los servicios se comunican eficazmente y que el sistema es capaz de escalar de manera modular según la demanda.

A través del API Gateway, las solicitudes realizadas desde el cliente web se enrutan correctamente hacia los microservicios internos, manteniendo una capa de abstracción, seguridad y separación de responsabilidades entre el frontend y la lógica del backend. Esta configuración facilita tanto el mantenimiento como la extensión del sistema.

Durante las pruebas, se constató que cada microservicio cumple adecuadamente con su función específica. El servicio de inventario permitió obtener la cartelera de películas junto con la disponibilidad actual de boletos, mientras que el servicio de compras gestionó correctamente la solicitud del usuario, registrando la operación y actualizando el stock de entradas de forma coherente.

El servidor Eureka operó como descubridor de servicios, facilitando la interacción entre los distintos módulos de manera automática. Se verificó que, al iniciar los microservicios, todos los servicios se registraban correctamente en el panel de Eureka (<http://localhost:8761>), lo cual evidencia un entorno funcional, distribuido y coordinado.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
APIGATEWAY	n/a (1)	(1)	UP (1) - host.docker.internal:apigateway:8083
SERVICIOBOLETOS	n/a (1)	(1)	UP (1) - host.docker.internal:servicioboletos:8081
SERVICIOCOMPRAS	n/a (1)	(1)	UP (1) - host.docker.internal:serviciocompras:8082

Figura 49. Servicios registrados vistos desde el panel de Eureka.

Pruebas en Postman

Se llevaron a cabo diversas pruebas utilizando Postman para verificar el funcionamiento de la API REST que opera a través del API Gateway (ubicado en <http://localhost:8083>). A continuación, se describen los principales puntos evaluados durante las pruebas:

Prueba de obtención de la cartelera de películas

La primera prueba consistió en realizar una solicitud GET /inventario/peliculas. Esta solicitud tiene como objetivo obtener la lista de películas disponibles junto con la cantidad de boletos restantes. La respuesta fue un arreglo en formato JSON, que contenía los títulos de las películas junto con sus cantidades de boletos disponibles. La información obtenida coincidió con los datos almacenados en la base de datos MySQL, lo que demostró que la conexión entre el microservicio de inventario y la base de datos se realizó correctamente.

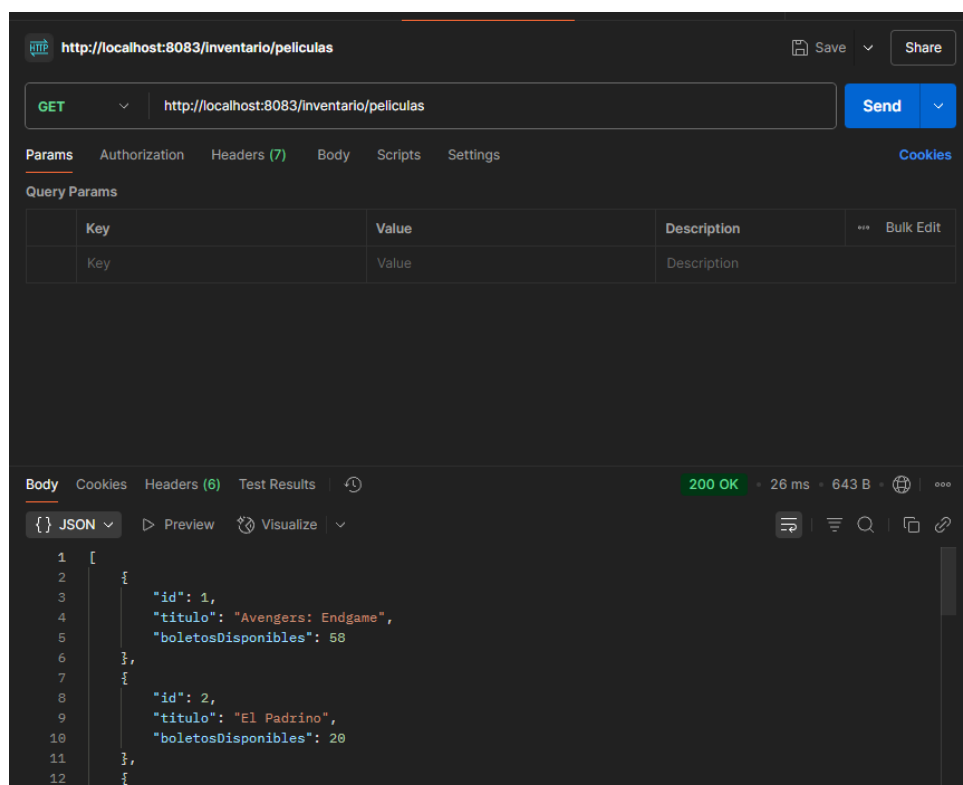


Figura 50. Prueba con Postman de obtención de la cartelera

Prueba de consulta de disponibilidad de boletos

Se realizaron solicitudes GET /boletos/{titulo} para obtener la cantidad de boletos disponibles de una película específica. El sistema respondió correctamente proporcionando la cantidad exacta de boletos disponibles, lo que permitió verificar que la función de consulta de stock operaba adecuadamente.

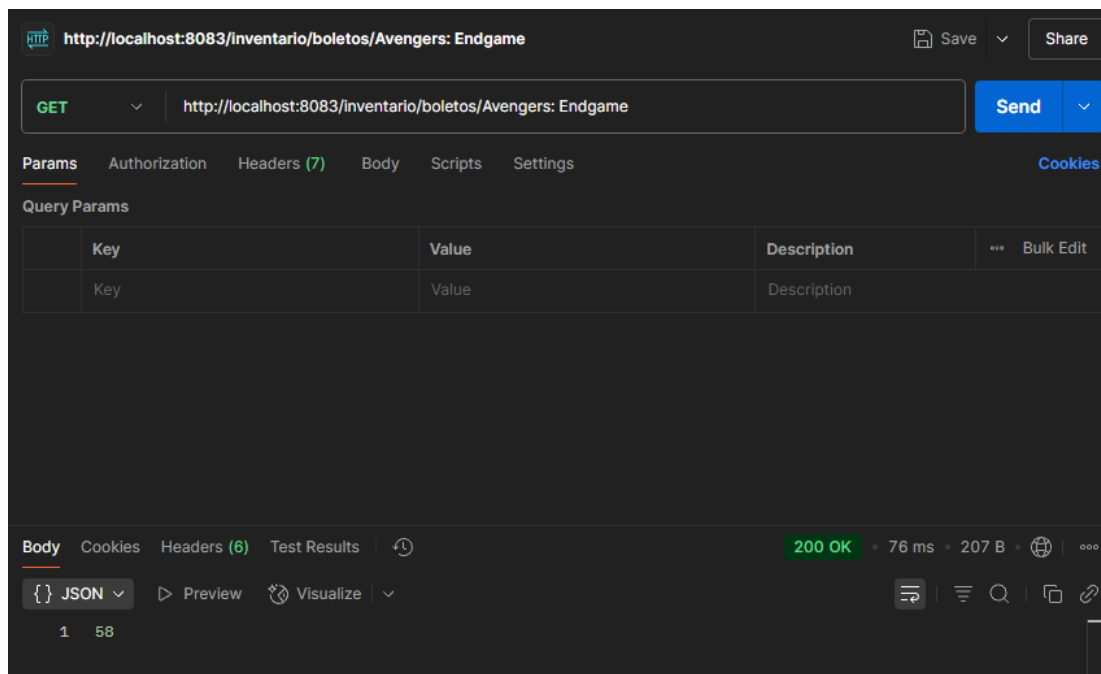


Figura 51. Prueba con Postman de obtención del stock específico de una película

Prueba de compra de boletos

Otra prueba crucial consistió en enviar solicitudes PUT /compras, donde se enviaba un listado de las películas seleccionadas junto con la cantidad de boletos solicitados. En los casos en los que la cantidad de boletos solicitados era válida y el stock era suficiente, el sistema respondía con un mensaje de confirmación y actualizaba el inventario en tiempo real. Si la cantidad solicitada excedía la disponibilidad de boletos, el sistema devolvía un mensaje de error indicando la insuficiencia de disponibilidad.

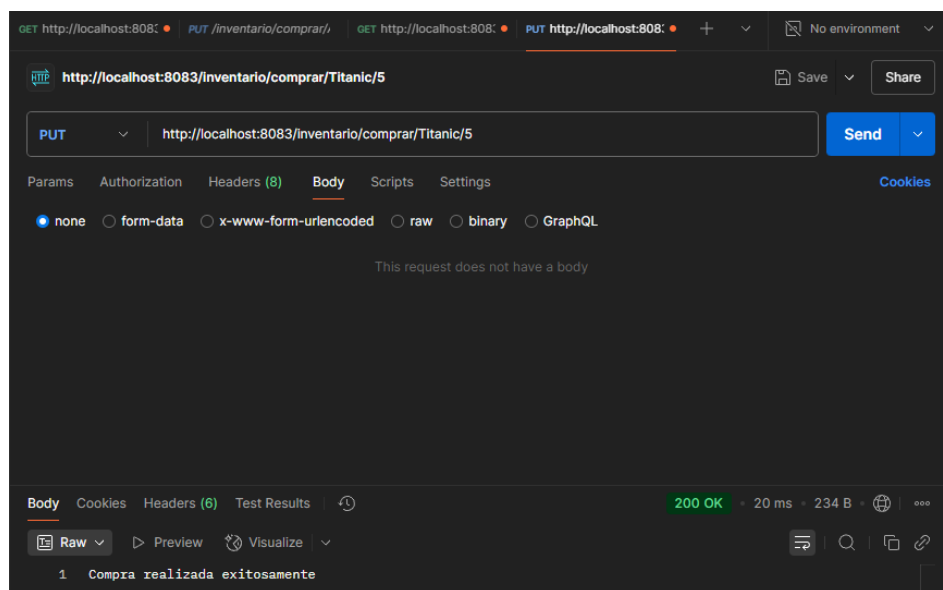


Figura 52. Prueba con Postman al realizar una compra

```
mysql> select * from inventario;
```

id	boletos_disponibles	titulo
1	56	Avengers: Endgame
2	20	El Padrino
3	45	Titanic
4	40	Inception
5	20	Interestelar
6	16	Avengers
7	20	Matrix
8	15	Parasite

```
8 rows in set (0.00 sec)
```

Figura 53. Actualización de la disponibilidad al realizar una compra en la BD.

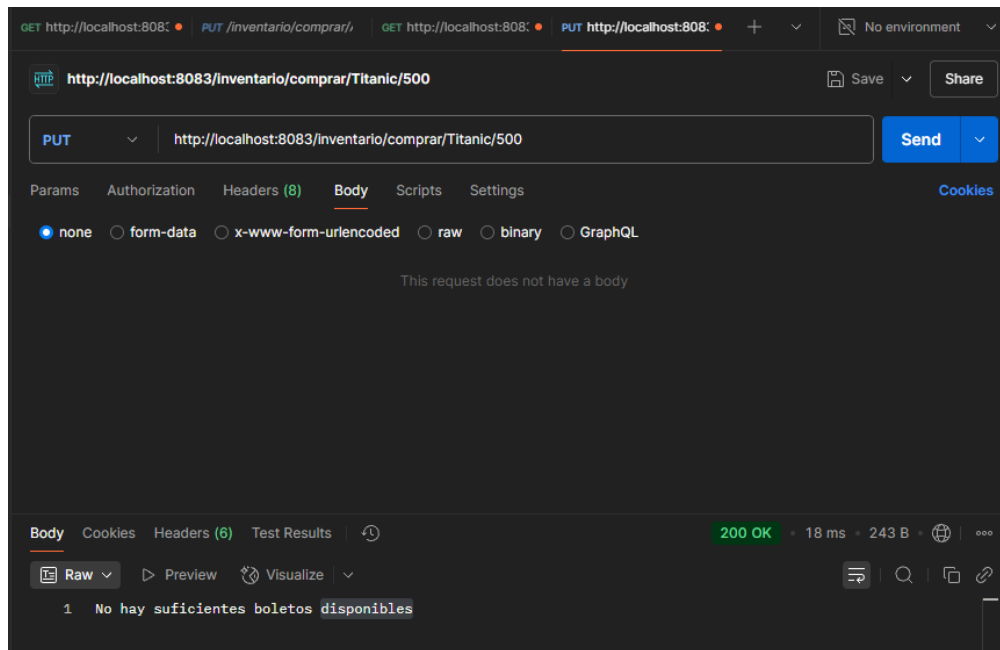


Figura 54. Prueba con Postman de fallo al realizar una compra

Pruebas desde el Frontend

Visualización de la Cartelera de Películas

Al abrir la página web en un navegador, la lista de películas y su respectiva disponibilidad de boletos se cargó correctamente en la interfaz, confirmando que la comunicación entre el frontend y el backend se estableció de forma exitosa. Esta información se obtuvo a través de una solicitud GET al endpoint correspondiente, expuesto por el API Gateway (en este caso, `http://localhost:8083/inventario/peliculas`). La solicitud fue realizada desde el archivo `script.js` del frontend, lo que permitió obtener dinámicamente la lista de películas disponibles en el cine y mostrarla en la página web.

The screenshot shows the CineBoleto website interface. At the top, there is a header with the logo and the text "¡Bienvenido a CineBoleto!" and "Compra tus boletos fácilmente y sin hacer filas". Below the header, there is a section titled "Cartelera:" with a list of movies and their available ticket counts. Each movie has a corresponding input field to select the number of tickets. At the bottom, there is a "Datos de la Compra:" section with a text input field for the user's name and a "Comprar Boletos" button. A green message at the bottom says "Selecciona al menos una película."

Movimiento	Boletos disponibles	Cantidad seleccionada
Avengers: Endgame	56	0
El Padrino	20	0
Titanic	40	0
Inception	40	0
Interestelar	20	0
Avengers	16	0
Matrix	16	0
Parasite	12	0

Datos de la Compra:

Ingresa tu nombre


Selecciona al menos una película.

Figura 55. Despliegue de la cartelera de películas en la página


Proceso de Compra de Boletos

Para validar la funcionalidad de compra, se probó el proceso ingresando un nombre de cliente y seleccionando diferentes cantidades de boletos para varias películas desde la interfaz. Al presionar el botón "Realizar Compra", el frontend envió una solicitud POST al backend a través del API Gateway (por ejemplo, <http://localhost:8083/compras>), incluyendo los detalles de la compra. Dado que la solicitud fue válida y el stock de boletos era suficiente, el backend procesó la compra correctamente y devolvió una respuesta de confirmación. Esta respuesta fue mostrada en la interfaz web, informando al usuario que la compra fue realizada con éxito.

Además, el stock de boletos se actualizó automáticamente en la base de datos y, tras la compra, la interfaz volvió a hacer una solicitud GET para obtener la lista actualizada de películas y reflejar los cambios en tiempo real.

 **Cartelera:**

Avengers: Endgame - Boletos disponibles: 49	<input type="text" value="0"/>
El Padrino - Boletos disponibles: 14	<input type="text" value="0"/>
Titanic - Boletos disponibles: 32	<input type="text" value="0"/>
Inception - Boletos disponibles: 30	<input type="text" value="0"/>
Interstellar - Boletos disponibles: 15	<input type="text" value="0"/>
Avengers - Boletos disponibles: 6	<input type="text" value="0"/>
Matrix - Boletos disponibles: 10	<input type="text" value="0"/>
Parasite - Boletos disponibles: 8	<input type="text" value="0"/>

 **Datos de la Compra:**

Compra exitosa para las películas: Avengers: Endgame, El Padrino, Titanic, Inception, Interstellar, Avengers, Matrix, Parasite.

Figura 56. Realización de una compra exitosa desde el sitio web

```
[
  {
    "cliente": "atl",
    "cantidad": 20
  },
  {
    "cliente": "Kelly ",
    "cantidad": 20
  },
  {
    "cliente": "Yosafat",
    "cantidad": 10
  },
  {
    "cliente": "ATL YOSFAT",
    "cantidad": 20
  }
]
```

Figura 57. Bitácora en donde se comprueba la conexión exitosa del front

```

: Compra exitosa: Cliente=Kelly compr| 20 boletos. Boletos restantes=80
: Compra exitosa: Cliente=Yosafat compr| 10 boletos. Boletos restantes=70

: Compra exitosa: Cliente=ATL YOSFAT compr| 20 boletos. Boletos restantes=70

```

Figura 58 Registro exitoso en terminal

Pruebas de Validación del Sistema

En cuanto a las pruebas de validación, se implementaron algunas restricciones en la interfaz para asegurar que los usuarios ingresen correctamente los datos. Si un usuario solo ingresaba su nombre y no seleccionaba ninguna película, el sistema le indicaba que debía seleccionar al menos una película para realizar la compra. Por otro lado, si un usuario seleccionaba productos, pero no ingresaba su nombre, se le solicitaba que introdujera su nombre antes de proceder con la compra. Estas validaciones fueron implementadas para garantizar que las compras solo se realicen cuando se han completado todos los campos obligatorios.

Matrix - Boletos disponibles: 10

Parasite - Boletos disponibles: 8

Datos de la Compra:

Selecciona al menos una película.

Figura 59. Solicitud al usuario de que seleccione al menos una película

Parasite - Boletos disponibles: 8

Datos de la Compra:

Por favor, ingresa tu nombre.

Figura 60. Solicitud al usuario para que ingrese su nombre al realizar una compra.

Las pruebas realizadas confirmaron que el sistema de cine cumple con las funcionalidades esperadas. Tanto el backend como el frontend mantuvieron una comunicación fluida y estable, lo que permitió una gestión eficiente de las compras y el inventario de boletos. El diseño basado en microservicios permitió una distribución eficiente de los componentes del sistema, facilitando su mantenimiento y escalabilidad. Con estos resultados, se valida que la solución propuesta es efectiva para la administración distribuida y en tiempo real de la venta de boletos en el cine.

Conclusión

Desarrollar este proyecto basado en microservicios para la materia de Sistemas Distribuidos fue una experiencia bastante enriquecedora, pero también compleja y retadora. A lo largo del desarrollo, pude comprender a profundidad los beneficios y desafíos que implica trabajar con una arquitectura distribuida, especialmente cuando se busca escalar y mantener el modularidad de una aplicación.

Uno de los aspectos que más complicaciones me generó fue la conexión y el manejo de la base de datos. Al trabajar con múltiples microservicios, cada uno con su responsabilidad y, en algunos casos, con su propia instancia de datos, surgieron conflictos de sincronización, duplicación de información y errores de conexión, sobre todo al momento de integrarlos todos a través del API Gateway. Me tomó tiempo entender cómo aislar correctamente los datos y asegurarme de que cada servicio tuviera acceso únicamente a lo que le correspondía.

Además, la estructuración de los microservicios en sí no fue una tarea sencilla. A pesar de haber investigado previamente sobre patrones de diseño y ejemplos, al momento de la práctica surgieron dudas constantes sobre la mejor forma de dividir la lógica del sistema, cómo establecer la comunicación entre servicios, cómo mantener la consistencia en los datos compartidos y cómo asegurar la tolerancia a fallos. El diseño inicial cambió varias veces conforme fui detectando cuellos de botella, problemas de redundancia o simplemente errores en el planteamiento.

Sin embargo, a pesar de todas las dificultades, considero que fue una experiencia muy valiosa. Me permitió poner en práctica conceptos importantes como la comunicación REST, el uso de RMI para el balanceo de carga, el control de concurrencia y la implementación de una interfaz cliente funcional. También me dio una visión más realista de los retos que enfrentan los sistemas modernos en producción, donde la escalabilidad, la disponibilidad y la organización del código son claves.

En lo personal, realizar este proyecto me pareció muy interesante. Aunque en momentos me sentí frustrado por los errores y la complejidad técnica, al final logré consolidar una solución funcional que refleja lo aprendido durante el curso. Esta experiencia me motivó a seguir explorando este tipo de arquitecturas, ya que considero que dominar los microservicios es fundamental para el desarrollo de software actual y futuro.

Referencias

- Microsoft Learn, "Conceptos de Sistemas Distribuidos," 2022. [En línea]. Available: <https://learn.microsoft.com/es-es/azure/architecture/patterns/> [Último acceso: 01 Abril 2025].
- Spring, "Introducción a Spring Boot," 2024. [En línea]. Available: <https://spring.io/projects/spring-boot> . [Último acceso: 01 Abril 2025].
- Silberschatz, P. Galvin y G. Gagne, *Operating System Concepts*, 10ª ed., Wiley, 2018.
- Apache Maven, "Descarga e instalación de Maven," 2024. [En línea]. Available: <https://maven.apache.org/download.cgi> . [Último acceso: 01 Abril 2025].
- ResearchGate, "Arquitectura típica de un Servicio Web," 2024. [En línea]. Available: https://www.researchgate.net/figure/Figura-212-Arquitectura-tipica-de-u-ServicioWeb_fig7_305403120. [Último acceso: 01 Abril 2025].
- Disrupción Tecnológica, "Arquitectura de Servicios Web," 2024. [En línea]. Available: <https://www.disrupciontecnologica.com/arquitectura-de-servicios-web/> [Último acceso: 01 Abril 2025].
- Spring Cloud, "Guía de Spring Cloud Netflix Eureka," 2024. [En línea]. Disponible en: https://cloud.spring.io/spring-cloudnetflix/multi/multi__service_discovery_eureka_clients.html . [Último acceso: 15 abril 2025].
- Docker Docs, "Getting Started with Docker," 2024. [En línea]. Disponible en: <https://docs.docker.com/get-started/> . [Último acceso: 15 abril 2025].
- Apache Maven, "Descarga e instalación de Maven," 2024. [En línea]. Disponible en: <https://maven.apache.org/download.cgi> . [Último acceso: 12 abril 2025].
- Postman, "Guía de uso de Postman para API REST," 2024. [En línea]. Disponible en: <https://learning.postman.com/docs/getting-started/introduction/> . [Último acceso: 14 abril 2025].
- Platzi, "¿Qué significa REST y qué es una API RESTful?" 2024. [En línea]. Disponible en: <https://platzi.com/clases/1638-api-rest/21611-que-significa-rest-y-que-es-una-api-restful/>. [Último acceso: 11 abril 2025].
- Mozilla Developer Network (MDN), "Uso de Fetch API," 2024. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Web/API/Fetch_API . [Último acceso: 16 abril 2025].
- Silberschatz, P. Galvin y G. Gagne, *Operating System Concepts*, 10ª ed., Wiley, 2018.
- ResearchGate, "Arquitectura típica de un Servicio Web," 2024. [En línea]. Disponible en: https://www.researchgate.net/figure/Figura-212-Arquitectura-tipica-de-un-ServicioWeb_fig7_305403120 . [Último acceso: 14 abril 2025].

- *Disrupción Tecnológica*, "Arquitectura de Servicios Web," 2024. [En línea]. Disponible en: <https://www.disrupciontecnologica.com/arquitectura-de-servicios-web/>. [Último acceso: 10 abril 2025].
- *LinkedIn – Midudev*, "Esta es la arquitectura de Netflix y su evolución," 2024. [En línea]. Disponible en: https://es.linkedin.com/posts/midudev_esta-es-la-arquitectura-de-netflix-y-su-activity-7141082778275684352-T--r. [Último acceso: 09 abril 2025].
- *Chakray*, "Experiencia con Arquitecturas de Microservicios," 2024. [En línea]. Disponible en: <https://chakray.com/es/experiencia/microservicios/>. [Último acceso: 08 abril 2025].
- *Fazt Code (YouTube)*, "¿Qué es una arquitectura de microservicios? ¿Cómo funciona?", 2024. [En línea]. Disponible en: <https://www.youtube.com/watch?v=8g1GKGd3ens>. [Último acceso: 07 abril 2025].
- *Universidad Veracruzana – E. Meneses*, "Introducción a los Sistemas Distribuidos," 2020. [En línea]. Disponible en: <https://www.uv.mx/personal/ermeneses/files/2020/09/Clase1-Introduccion.pdf>. [Último acceso: 15 abril 2025].

Para este apartado, a continuación, se mencionan los elementos técnicos relevantes para la implementación del sistema distribuido desarrollado en esta práctica, dato a que son demasiados y el anexar el código fuente de cada uno sería complejo y tedioso se decidió que agregar una carpeta con todos los archivos en la carpeta adjunta en el drive titulada “Practica 7_Microservicios.zip”, organizada por microservicio y componentes.

- 1) **Anexo A = Microservicio de Boletos:** Contiene los archivos CineController.java, CineService.java, Película.java, PelículaRepository.java y application.properties.
- 2) **Anexo B = Microservicio de Compras:** Incluye BoletoController.java, BoletoService.java, BoletoRepository.java y application.properties.
- 3) **Anexo C = Microservicio de Cliente:** Con su respectiva lógica de frontend, archivo index.html junto con los estilos, scripts utilizados y configuración.
- 4) **Anexo D = Eureka Server y API Gateway:** Configuración básica y application.properties.

Para consultar el código completo, por favor revise la carpeta “Practica 7_Microservicios.zip” incluida en este drive.