INSTITUTO POLITÉCTICO NACIONAL

Escuela Superior de Cómputo

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Practica 6:

Servicio Web

Alumno:

Cardoso Osorio Atl Yosafat

Profesor:

Chadwick Carreto Arellano

Fecha:

02/04/2025

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Índice de contenido

Índice de figuras

Antecedentes

Los servicios web han revolucionado la forma en que las aplicaciones distribuidas se comunican en entornos heterogéneos. Su desarrollo responde a la necesidad de intercambiar datos y ejecutar procesos de manera interoperable a través de redes, principalmente en internet. A través de protocolos estándar, permiten que diferentes sistemas, sin importar su plataforma o lenguaje de programación, se integren y colaboren eficazmente.

Servicios Web

Un servicio web es un conjunto de funcionalidades accesibles a través de internet o redes privadas que permiten la interacción entre distintas aplicaciones o sistemas mediante el uso de protocolos y formatos de comunicación estandarizados. A diferencia de los métodos tradicionales de comunicación entre aplicaciones, que suelen requerir compatibilidad en términos de plataforma y lenguaje de programación, los servicios web se diseñan para ser independientes de la implementación tecnológica. Esto los convierte en una solución altamente flexible para la integración de sistemas distribuidos.

Los servicios web operan sobre protocolos estándar ampliamente adoptados, como HTTP (Hypertext Transfer Protocol) y SOAP (Simple Object Access Protocol), y emplean formatos estructurados de intercambio de información, como JSON (JavaScript Object Notation) y XML (Extensible Markup Language). Estos formatos permiten representar datos de una manera estructurada y legible tanto para humanos como para máquinas, facilitando la comunicación entre aplicaciones heterogéneas.

Desde el punto de vista de su arquitectura, los servicios web se pueden clasificar en distintos paradigmas, siendo los más representativos REST (Representational State Transfer) y WS- (Web Services Architecture):

- ***RESTful Web Services:*** Son servicios web basados en la arquitectura REST, que utiliza operaciones estándar de HTTP, como GET, POST, PUT y DELETE, para manipular recursos representados en formatos como JSON o XML. REST se ha convertido en la opción preferida para la mayoría de las aplicaciones modernas debido a su simplicidad, escalabilidad y eficiencia en la transmisión de datos.
- ***WS- (Web Services Architecture):*** Se basa en una serie de estándares definidos por organismos como el W3C (World Wide Web Consortium) y la OASIS (Organization for the Advancement of Structured Information Standards), entre los que se encuentran SOAP, WSDL (Web Services Description Language) y UDDI (Universal Description, Discovery, and Integration). Este modelo es utilizado en entornos empresariales que requieren mayor seguridad y control en la comunicación de datos.

En términos generales, un servicio web actúa como una interfaz remota que expone ciertas operaciones o funcionalidades para ser consumidas por otros sistemas, sin necesidad de que estos compartan el mismo entorno de desarrollo o infraestructura. Cuando una aplicación

cliente necesita acceder a un servicio web, realiza una solicitud a un servidor, que a su vez procesa la petición, ejecuta las operaciones necesarias y devuelve una respuesta estructurada, generalmente en formato JSON o XML.

El propósito principal de los servicios web es promover la interoperabilidad y la comunicación eficiente entre sistemas distribuidos, sin importar el lenguaje de programación o plataforma en la que fueron implementados. Gracias a esto, las organizaciones pueden integrar aplicaciones heterogéneas, optimizar flujos de trabajo y mejorar la accesibilidad a la información de manera estructurada y segura.

Características de los Servicios Web

Los servicios web presentan varias características clave que los hacen fundamentales en el contexto de los sistemas distribuidos:

- **Interoperabilidad**: Al utilizar estándares abiertos (como HTTP, XML, JSON y SOAP), los servicios web pueden ser consumidos por aplicaciones desarrolladas en distintos lenguajes y ejecutadas en diferentes plataformas.
- **Comunicación basada en protocolos estándar**: Emplean protocolos como HTTP, REST, SOAP y WebSockets para la transmisión de datos, asegurando compatibilidad con una amplia variedad de sistemas.
- **Independencia de la plataforma y del lenguaje de programación**: Pueden ser desarrollados en cualquier lenguaje y ejecutados en cualquier infraestructura sin necesidad de adaptación.
- **Arquitectura distribuida**: Permiten la descentralización del procesamiento y almacenamiento de datos, favoreciendo la escalabilidad y flexibilidad de las aplicaciones.
- **Uso de descriptores de servicios**: Tecnologías como WSDL (Web Services Description Language) permiten definir formalmente las operaciones, parámetros y formatos de los datos intercambiados.
- **Seguridad y control de acceso**: Se pueden implementar protocolos como OAuth, JWT, SSL/TLS y WS-Security para garantizar la autenticación, autorización y cifrado de los datos transmitidos.

Ventajas y Aplicaciones de los servicios web en Sistemas Distribuidos

Los servicios web han impulsado el desarrollo de aplicaciones más modulares, escalables y mantenibles. Algunas de sus principales ventajas en sistemas distribuidos incluyen:

- Facilidad de integración: Permiten conectar aplicaciones heterogéneas sin necesidad de modificar su lógica interna.

- Eficiencia en la comunicación: Al usar estándares abiertos, se optimiza la transmisión de datos entre sistemas distribuidos.
- Reutilización de componentes: Facilitan la reutilización de servicios, reduciendo costos y tiempos de desarrollo.
- Escalabilidad: Su arquitectura distribuida permite manejar grandes volúmenes de datos y usuarios concurrentes.
- Flexibilidad: Pueden ser consumidos por múltiples dispositivos, desde aplicaciones móviles hasta sistemas empresariales complejos.

Entre sus aplicaciones más comunes se encuentran:

1) Sistemas empresariales: Integración de plataformas ERP y CRM con otros servicios internos y externos.
2) Aplicaciones móviles: Sincronización de datos con servidores en la nube.
3) E-commerce: Interacción con pasarelas de pago, proveedores de logística y sistemas de gestión de inventario.
4) Internet de las cosas (IoT): Comunicación entre dispositivos inteligentes a través de APIs web.
5) Servicios en la nube: Infraestructura como servicio (IaaS), plataformas como servicio (PaaS) y software como servicio (SaaS).

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.

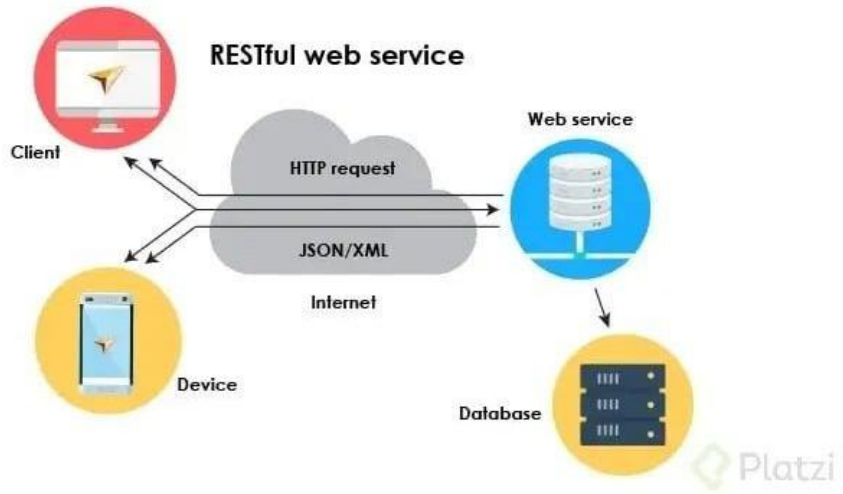

Figura 1. Estructura básica de un servicio web REST

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.

Figura 2. 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.

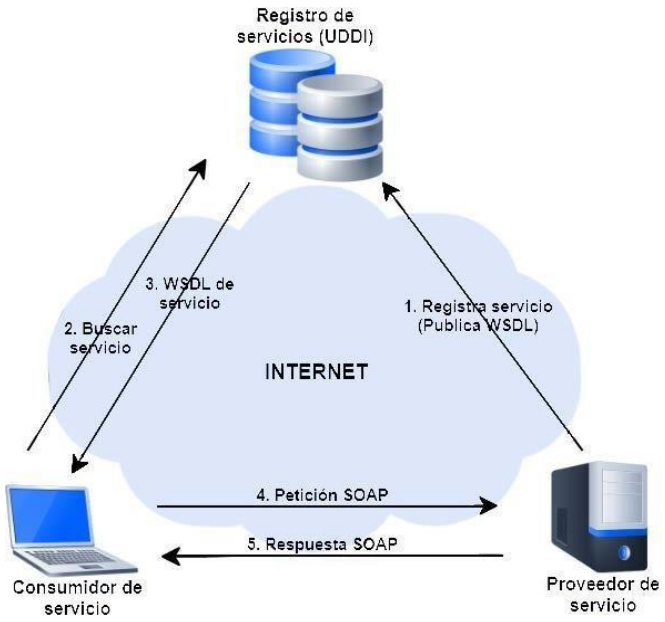

Figura 3. Estructura básica de un servicio web SOAP

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.

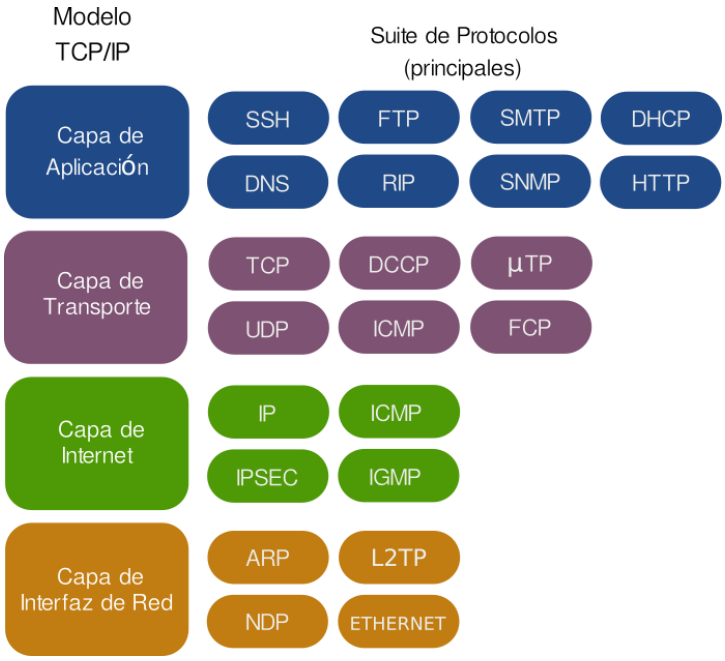

Figura 4. Diagrama de protocolos de comunicación

Planteamiento del problema

En la actualidad, la venta de boletos en línea para eventos y funciones de cine es un proceso fundamental para la optimización de recursos y la comodidad de los clientes. Sin embargo, gestionar la disponibilidad de boletos en tiempo real y garantizar la integridad de las transacciones en un entorno distribuido presenta diversos desafíos, especialmente en términos de concurrencia, escalabilidad y manejo de errores.

El sistema desarrollado en esta práctica busca simular un servicio web distribuido para la administración de boletos en un cine. Este servicio está diseñado para gestionar la compra de boletos en tiempo real, garantizar la correcta actualización del inventario y registrar las transacciones realizadas por los clientes.

Entre los principales problemas que se aborda en los servicios web es la gestión de concurrencia, ya que varios clientes pueden intentar comprar boletos al mismo tiempo, lo que puede generar inconsistencias en la disponibilidad si no se implementan mecanismos adecuados de sincronización. Otra de las problemáticas que enfrentan son la escalabilidad del servicio, debido a que a medida que aumenta la demanda, el sistema debe ser capaz de manejar múltiples solicitudes concurrentes sin degradar su rendimiento. El manejo de excepciones y errores también es fundamental para gestionar adecuadamente errores como solicitudes inválidas o intentos de compra superiores a la cantidad disponible de boletos.

Por otro lado, la seguridad y restricciones de acceso son importantes, es por ello por lo que implementamos controles para evitar manipulaciones en las transacciones y garantizar el correcto funcionamiento del servicio. Así como también se busca la automatización de proceso, por lo que se requiere la generación automática de boletos en intervalos definidos para simular un sistema en producción.

Esta práctica busca principalmente el resolver un ejemplo práctico de cómo un sistema distribuido puede implementarse para un servicio, pero con desafíos en términos de concurrencia, comunicación de red y gestión de recursos compartidos.

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

Propuesta de solución

Para abordar los desafíos identificados en la gestión de boletos para un cine en un entorno distribuido, se propone el desarrollo e implementación de un servicio web basado en una arquitectura RESTful utilizando el framework Spring Boot. Este servicio permitirá la gestión eficiente de la disponibilidad, compra y generación automática de boletos mediante el uso de

controladores REST, servicios de negocio y almacenamiento de datos en estructuras adecuadas.

En primer lugar, se implementará un controlador REST (CineController) que permitirá la interacción con los clientes a través de endpoints HTTP. Este controlador gestionará las operaciones fundamentales como la consulta de boletos disponibles, la compra de boletos por parte de los usuarios y la adición de nuevos boletos al sistema. Además, se incluirán validaciones para evitar inconsistencias en las solicitudes, como compras con cantidades inválidas o sin especificar el nombre del cliente.

Para garantizar la correcta administración de los datos y el acceso concurrente, se diseñará un servicio de negocio (CineService) que manejará la lógica principal de la aplicación. Este servicio será responsable de actualizar la cantidad de boletos disponibles, registrar las ventas y generar boletos automáticamente en intervalos predefinidos mediante el uso de tareas programadas (@Scheduled). Estas tareas garantizarán la reposición de boletos sin intervención manual, asegurando que la disponibilidad de boletos se mantenga en niveles adecuados.

Dado que el sistema operará en un entorno distribuido, es fundamental implementar mecanismos de sincronización y concurrencia para evitar condiciones de carrera y asegurar la integridad de los datos. Se utilizarán métodos synchronized para garantizar que las operaciones críticas, como la compra de boletos y la actualización de inventario, se ejecuten de manera segura sin generar inconsistencias en el número de boletos disponibles.

Para mejorar la seguridad y estabilidad del servicio, se incorporará un manejador global de excepciones (GlobalExceptionHandler) que capturará y responderá apropiadamente a errores como solicitudes mal formuladas o intentos de compra que excedan la cantidad de boletos disponibles. Esto permitirá proporcionar respuestas adecuadas al cliente sin comprometer la estabilidad del servicio.

Además, se implementará una configuración de CORS (Cross-Origin Resource Sharing) para permitir el acceso a la API desde múltiples clientes, garantizando compatibilidad con aplicaciones web y móviles sin restricciones de origen. Esto facilitará la escalabilidad del servicio y permitirá su integración con otras plataformas.

Con esta solución, el sistema proporcionará un servicio web robusto, seguro y escalable para la gestión de boletos en un cine, optimizando el manejo de concurrencia, la automatización de tareas y la validación de solicitudes. La implementación de esta arquitectura distribuida permitirá garantizar un servicio confiable y eficiente, adaptándose a las necesidades de los usuarios y a la disponibilidad de recursos en el entorno distribuido.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollar esta práctica con Objetos Distribuidos para resolver el problema de la venta de boletos 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. Java proporciona una API nativa para la implementación de RMI (Remote Method Invocation), lo que facilita la creación de aplicaciones cliente-servidor que pueden comunicarse a través de una red. Además, las características de Java para el manejo de hilos y concurrencia fueron fundamentales para gestionar múltiples solicitudes de clientes de manera eficiente.

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 del servicio web, 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 empleo 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 servicios web RESTful, se emplearon los siguientes métodos y técnicas:

Arquitectura Cliente-Servidor Distribuida

Se implementó un sistema cliente-servidor basado en servicios web REST mediante Spring Boot. El servidor expone una serie de endpoints que permiten a los clientes realizar operaciones remotas, como consultar la disponibilidad de boletos, realizar compras y visualizar el historial de ventas. Los clientes interactúan con el sistema mediante solicitudes HTTP utilizando los métodos GET y POST.

Concurrencia en el Acceso al Inventario

Para gestionar múltiples solicitudes simultáneas y evitar condiciones de carrera, el acceso a la disponibilidad de boletos se controló mediante el uso de métodos sincronizados en el backend. Estas operaciones garantizan que las compras se procesen de manera atómica, evitando inconsistencias en el número de boletos disponibles.

Automatización del Reabastecimiento de Boletos

Se implementó un proceso automatizado para aumentar la disponibilidad de boletos de manera periódica. Utilizando la anotación @Scheduled de Spring Boot, el sistema genera boletos automáticamente cada 60 segundos, simulando la reposición de entradas en el cine sin intervención manual.

Manejo de Solicitudes Concurrentes

El sistema está diseñado para procesar múltiples solicitudes concurrentes de clientes mediante la arquitectura asincrónica de Spring Boot. Como servicio web RESTful, el servidor puede atender varias peticiones simultáneamente, asegurando una respuesta rápida y evitando bloqueos en el sistema.

Pruebas de API

Se utilizó Postman y el HTML para probar los diferentes endpoints del servicio web. Esto permitió verificar que los métodos de la API (como la consulta de boletos disponibles, la compra de boletos y la reposición automática) funcionaran correctamente bajo diferentes condiciones. Las pruebas incluyeron intentos de compra con stock suficiente, solicitudes con stock insuficiente y validaciones de datos de entrada.

Registro de Actividades y Monitoreo

Para monitorear el comportamiento del sistema y facilitar la depuración, se implementó un sistema de registro de eventos en Spring Boot, utilizando la librería SLF4J. Esto permitió registrar las compras realizadas, los cambios en la disponibilidad de boletos y las actualizaciones automáticas del inventario. Los registros facilitan la identificación de errores y mejoran la gestión del sistema.

Con estos métodos y herramientas, se logró desarrollar un sistema distribuido eficiente para la gestión de boletos de cine, garantizando la integridad de los datos y una interacción fluida entre el servidor y los clientes.

Interfaz de Usuario (Frontend)

El sistema también incluye una interfaz de usuario diseñada con HTML, CSS y JavaScript, que permite a los usuarios interactuar con el servicio web de forma sencilla e intuitiva.

- **HTML: Estructura de la Página**

Se utilizó HTML para definir la estructura de la página web. En el archivo Cineusuario.html, se creó una interfaz que permite a los usuarios ver la cantidad de boletos disponibles y realizar compras mediante un formulario interactivo.

- **CSS: Estilo y Diseño**

Mediante el archivo style.css, se aplicaron estilos visuales a la interfaz. Se personalizaron los botones, los campos de entrada y la disposición de los elementos, asegurando una experiencia de usuario clara y accesible.

- **JavaScript: Lógica de Interacción y Comunicación con el Backend**

La funcionalidad de la interfaz se implementó en script.js, donde se estableció la comunicación con el backend utilizando fetch API.

- **Consulta de Boletos Disponibles:**

  Cuando el usuario accede a la página, la función obtenerBoletos() realiza una solicitud GET al endpoint /cine/boletos, obteniendo el número de boletos disponibles y mostrándolo en la interfaz.

- **Compra de Boletos:**

  Cuando el usuario ingresa su nombre y selecciona la cantidad de boletos a comprar, la función comprarBoletos() envía los datos al backend mediante una solicitud POST. Si la compra es exitosa, el backend responde con un mensaje de confirmación.

- **Actualización Dinámica:**

Después de cada compra, la función obtenerBoletos() se ejecuta nuevamente para reflejar los cambios en la disponibilidad en la interfaz.

Conexión con el Backend

El frontend se comunica con el backend Spring Boot a través de la URL base http://localhost:8080/cine. Las solicitudes GET y POST permiten la consulta y compra de boletos en tiempo real, asegurando una experiencia de usuario fluida y eficiente.

Este sistema permite a los clientes visualizar la disponibilidad en tiempo real, realizar compras de manera sencilla y recibir actualizaciones sobre el stock de boletos, mejorando así la experiencia del usuario.

Desarrollo

En esta práctica se implementó un sistema distribuido para un servicio web de venta de boletos de cine, utilizando el framework Spring Boot. Para ello, se generó un proyecto con Spring Initializr, asegurándonos de incluir las dependencias necesarias para la creación de un servicio web REST.

Primeramente, se realizó la clase CineRMI, la cual representa La interfaz remota define los métodos que pueden ser invocados por el cliente de forma remota. En nuestro caso, la interfaz tiene métodos para obtener boletos disponibles, comprar boletos, y registrar la conexión y desconexión de clientes.

Para estructurar el backend del sistema distribuido del cine, se utilizó Spring Boot como marco principal para desarrollar un servicio web RESTful. El primer paso fue acceder a Spring Initializr (https://start.spring.io) para generar la configuración inicial del proyecto. Se seleccionaron las siguientes opciones: Maven como sistema de construcción, Java como lenguaje de programación y Spring Boot 3.4.4 como versión base. Además, se estableció la estructura del paquete con el grupo com.servicioweb, el artefacto cine y el nombre del proyecto como cine , asegurando así una organización clara del código. El paquete raíz quedó definido como com.servicioweb.cine y se eligió Jar como formato de empaquetado, con Java 17 como versión del lenguaje.

Posteriormente, se seleccionaron las dependencias necesarias para el desarrollo del backend. Se incluyó Spring Web para exponer los servicios REST y permitir la comunicación con clientes externos.

Una vez configuradas las opciones y seleccionadas las dependencias, se descargó el archivo ZIP generado por Spring Initializr. Luego, se descomprimió y se abrió en Visual Studio Code, donde se comenzó la implementación del servicio web, siguiendo una estructura modular y organizada para facilitar su escalabilidad y mantenimiento.

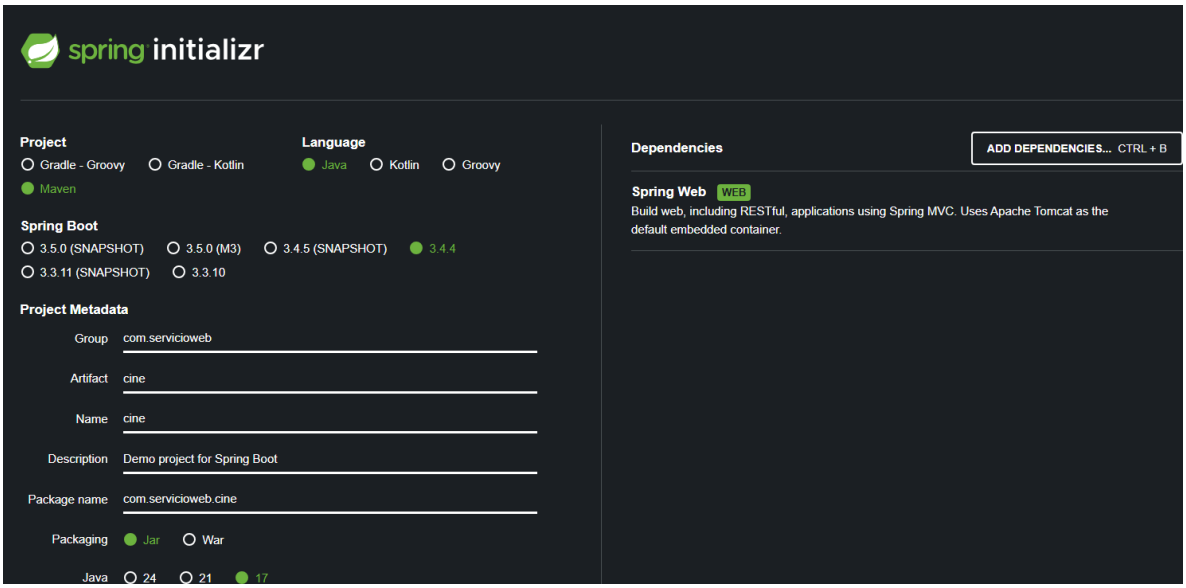

Figura 5. Estructura y creación del proyecto con Spring Boot

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\Pract
ica 6\ServicioWebCine> mvn -version
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
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 6. Verificación de la instalación de Apache Maven

Una vez completados estos pasos, se procedió con el desarrollo del backend, lo cual se detalla en la siguiente sección.

CineController.java

Ahora en la parte del desarrollo del backend del servicio web, primeramente, tenemos el código CineController.java, el cual contiene lo siguiente:

Paquete e Importaciones

En este código primeramente tenemos la declaración del paquete com.servicioweb.cine, que organiza las clases en un espacio de nombres específico dentro del proyecto. A continuación, se importan varias clases necesarias para la implementación del controlador. Entre ellas se incluyen las clases RestController, RequestMapping, PostMapping, GetMapping de Spring, que son esenciales para la creación de servicios web RESTful, así como la clase Boleto, que es un modelo que representa un boleto de cine.

```
package com.servicioweb.cine;

import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.servicioweb.cine.model.Boleto;
```

Figura 7. Paquetes e importaciones de CineController.java.

Anotaciones y Clase Controladora

En la primera línea dentro de la clase CineController, encontramos la anotación @CrossOrigin(origins = "*"). Esta anotación permite que el servicio reciba solicitudes de cualquier origen, lo cual es útil cuando el servicio se consume desde un cliente web o una aplicación móvil que puede estar alojado en un dominio diferente, en nuestro caso este cliente web será nuestra página realizada en HTML. La notación * indica que se permite el acceso desde cualquier origen sin restricciones.

Posteriormente, la clase CineController está marcada con la anotación @RestController, lo que indica que es un controlador REST y que los métodos dentro de la clase se encargarán de gestionar las solicitudes HTTP y devolver respuestas adecuadas. Esta anotación combina @Controller y @ResponseBody, lo que implica que los métodos en la clase devolverán datos (generalmente en formato JSON) y no vistas.

La anotación @RequestMapping("/cine") especifica que todas las rutas definidas dentro de esta clase estarán bajo el prefijo /cine, lo que nos proporciona una forma de organizar y estructurar las URL del servicio.

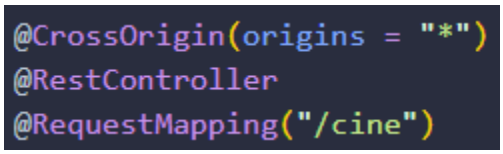

Figura 8. Anotaciones y clase controladora

Atributo cineService y Constructor

Dentro de la clase CineController, definí un atributo privado cineService, que es de tipo CineService. Este atributo se inicializa mediante un constructor que recibe un objeto CineService. El propósito de este atributo es delegar las operaciones lógicas de negocio relacionadas con la gestión de boletos al servicio CineService, lo que ayuda a mantener una separación clara entre la lógica de presentación (el controlador) y la lógica de negocio (el servicio).

Métodos del Controlador

1. agregarBoletos

El primer método es el encargado de agregar boletos al inventario disponible en el cine. Se marca con la anotación @PostMapping("/agregar"), lo que significa que este método se invoca cuando se recibe una solicitud HTTP POST a la ruta /cine/agregar. Este método recibe un parámetro cantidad a través de la anotación @RequestParam, lo que indica que el valor de la cantidad se pasa como un parámetro en la solicitud HTTP. Al ejecutarse, el método invoca al servicio cineService.agregarBoletos(cantidad) para aumentar el número de boletos disponibles. Finalmente, devuelve un mensaje de éxito indicando cuántos boletos se agregaron.

2. obtenerBoletosDisponibles

Este método es un simple getter que devuelve la cantidad actual de boletos disponibles. Está marcado con @GetMapping("/boletos"), lo que significa que este método maneja las solicitudes HTTP GET a la ruta /cine/boletos. Al ejecutarse, el método llama al servicio cineService.obtenerBoletosDisponibles() para obtener el número de boletos disponibles y lo devuelve como respuesta.

```java
public class CineController {
    private final CineService cineService;

    public CineController(CineService cineService) {
        this.cineService = cineService;
    }

    @PostMapping("/agregar")
    public String agregarBoletos(@RequestParam int cantidad) {
        cineService.agregarBoletos(cantidad);
        return cantidad + " boletos agregados exitosamente.";
    }

    @GetMapping("/boletos")
    public int obtenerBoletosDisponibles() {
        return cineService.obtenerBoletosDisponibles();
    }
```

Figura 9. Clase CineController y sus métodos parte 1

3. comprarBoletos

Este método se encarga de gestionar las compras de boletos. Está marcado con la anotación @PostMapping("/comprar"), lo que indica que se invoca cuando se recibe una solicitud HTTP POST a la ruta /cine/comprar. El método recibe dos parámetros a través de @RequestParam: cliente (el nombre del cliente que realiza la compra) y cantidad (el número de boletos que desea comprar). Antes de proceder con la compra, el método valida ambos parámetros. Si el nombre del cliente es nulo o está vacío, lanza una excepción IllegalArgumentException con el mensaje "El nombre del cliente no puede estar vacío". De manera similar, si la cantidad de boletos solicitada es menor o igual a cero, lanza otra excepción con el mensaje "La cantidad de boletos debe ser mayor a cero". Si ambas validaciones pasan, el método invoca el servicio cineService.comprarBoleto(cliente, cantidad) para realizar la compra y devuelve un mensaje con el resultado, que puede ser exitoso o indicar que no hay suficientes boletos disponibles.

4. obtenerVentas

Este método devuelve una lista de objetos Boleto que representan todas las compras realizadas. Está marcado con @GetMapping("/ventas"), lo que indica que maneja las solicitudes HTTP GET a la ruta /cine/ventas. Al ejecutarse, el método llama al servicio cineService.obtenerVentas() para obtener todas las ventas registradas y devuelve esa lista como respuesta.

```java
@PostMapping("/comprar")
public String comprarBoletos(@RequestParam String cliente, @RequestParam int cantidad) {
    if (cliente == null || cliente.isBlank()) {
        throw new IllegalArgumentException(s:"El nombre del cliente no puede estar vacío.");
    }
    if (cantidad <= 0) {
        throw new IllegalArgumentException(s:"La cantidad de boletos debe ser mayor a cero.");
    }
    return cineService.comprarBoleto(cliente, cantidad);
}

@GetMapping("/ventas")
public List<Boleto> obtenerVentas() {
    return cineService.obtenerVentas();
```

Figura 10. Clase CineController y sus métodos parte 2

CineService.java

Posteriormente tenemos el archivo Cine Service.java el cual se encarga de manejar las operaciones relacionadas con la venta y la gestión de boletos de cine. La clase CineService contiene la lógica de negocio, que incluye la adición de boletos al inventario, la compra de boletos por parte de los clientes, la generación automática de boletos y la obtención del historial de ventas.

Paquete e Importaciones

Primeramente tenemos la declaración del paquete com.servicioweb.cine, lo que indica que esta clase pertenece a la estructura organizativa del proyecto, específicamente a los servicios relacionados con el cine. Luego, se importan varias clases necesarias para la implementación:

- *Boleto:* Esta clase es un modelo que representa un boleto de cine y se usa para registrar las compras de los clientes.
- *Service:* Esta anotación de Spring indica que la clase es un servicio, lo que significa que contiene la lógica de negocio.
- *Scheduled:* Esta anotación permite ejecutar métodos de manera periódica, en este caso, para generar boletos automáticamente cada cierto tiempo.
- *Logger y LoggerFactory:* Estas clases son de la biblioteca SLF4J y se utilizan para generar logs, lo cual es muy útil en aplicaciones distribuidas para hacer un seguimiento de las actividades y problemas que ocurren durante la ejecución.

```
package com.servicioweb.cine;
import com.servicioweb.cine.model.Boleto;
import org.springframework.stereotype.Service;
import org.springframework.scheduling.annotation.Scheduled;
import java.util.ArrayList;
import java.util.List;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

Figura 11. Paquetes e importaciones del archivo CineService.java

Atributos

Dentro de la clase, se definen dos atributos principales:

1. **boletosDisponibles**: Este es un contador que representa la cantidad de boletos disponibles para la venta. Se inicializa con un valor de 30 boletos disponibles.

2. **ventas**: Es una lista de objetos Boleto que registra todas las ventas realizadas, es decir, cada vez que un cliente compra boletos, se añade un objeto Boleto a esta lista.

Métodos de la Clase CineService

1. agregarBoletos

Este método es responsable de agregar boletos al inventario de boletos disponibles. Está marcado con la palabra clave synchronized, lo que significa que es un método que solo puede ser ejecutado por un hilo a la vez, lo que garantiza que no haya problemas de concurrencia cuando múltiples hilos intentan modificar el valor de boletosDisponibles simultáneamente. El método recibe un parámetro cantidad que indica cuántos boletos se deben agregar. Luego, simplemente aumenta el valor de boletosDisponibles por la cantidad recibida.

2. generarBoletosAutomaticos

Este método es invocado automáticamente gracias a la anotación @Scheduled(fixedRate = 60000). La anotación fixedRate indica que este método se ejecutará de manera periódica cada 60,000 milisegundos (es decir, cada minuto). Su propósito es generar boletos automáticamente y agregarlos al inventario disponible. En este caso, cada vez que se ejecuta el método, genera 10 boletos y aumenta la cantidad de boletos disponibles en ese valor. Además, imprime un mensaje en la consola informando cuántos boletos fueron generados.

La sincronización de este método asegura que la operación sea segura en un entorno multihilo, ya que podría haber múltiples hilos accediendo y modificando la cantidad de boletos al mismo tiempo.

3. obtenerBoletosDisponibles

Este método es muy sencillo: simplemente retorna el número de boletos disponibles. Está marcado como synchronized para asegurar que el valor de boletosDisponibles no cambie mientras se está leyendo, lo que es importante en un entorno concurrente para evitar que se obtenga un valor inconsistente.

```java
@Service
public class CineService {
    private static final Logger logger = LoggerFactory.getLogger(clazz:CineService.class);
    private int boletosDisponibles = 30;
    private List<Boleto> ventas = new ArrayList<>();

        public synchronized void agregarBoletos(int cantidad) {
            boletosDisponibles += cantidad;
        }

        @Scheduled(fixedRate = 60000)
        public synchronized void generarBoletosAutomaticos() {
            int boletosGenerados = 10;
            boletosDisponibles += boletosGenerados;
            System.out.println("Generados " + boletosGenerados + " boletos automáticamente.");
        }
    public synchronized int obtenerBoletosDisponibles() {
        return boletosDisponibles;
    }
```

Figura 12. Clase CineService primera parte

4. comprarBoleto

Este método es crucial, ya que se encarga de procesar la compra de boletos. Acepta dos parámetros: cliente (el nombre del cliente que está comprando los boletos) y cantidad (el número de boletos que desea comprar). El primer paso del método es verificar si la cantidad de boletos solicitada es mayor que la cantidad de boletos disponibles. Si no hay suficientes boletos, el método utiliza el logger para generar una advertencia (con el nivel de log warn) que indica que la compra no se pudo realizar debido a la falta de boletos, y retorna un mensaje indicando que no hay suficientes boletos disponibles.

Si hay suficientes boletos, el método procede a restar la cantidad de boletos comprados de boletosDisponibles, y luego añade un objeto Boleto a la lista ventas para registrar la compra. También utiliza el logger para generar un mensaje de información (con el nivel de log info) indicando que la compra fue exitosa, mencionando el cliente, la cantidad de boletos comprados y los boletos restantes.

Finalmente, el método retorna un mensaje confirmando la compra exitosa, con el número de boletos restantes.

5. obtenerVentas

Este método es un getter que devuelve la lista de ventas realizadas, es decir, todos los boletos que han sido comprados. No requiere ninguna lógica adicional, solo devuelve el contenido de la lista ventas.

```
public synchronized String comprarBoleto(String cliente, int cantidad) {
    if (boletosDisponibles < cantidad) {
        logger.warn(format:"Compra fallida: Cliente={} intentó comprar {} boletos. Boletos disponibles={}", cliente, cantidad, boletosDisponibles);
        return "No hay suficientes boletos disponibles.";
    }
    boletosDisponibles -= cantidad;
    ventas.add(new Boleto(cliente, cantidad));
    logger.info(format:"Compra exitosa: Cliente={} compró {} boletos. Boletos restantes={}", cliente, cantidad, boletosDisponibles);
    return "Compra exitosa. Boletos restantes: " + boletosDisponibles;
}

public List<Boleto> obtenerVentas() {
    return ventas;
}
```

Figura 13. Clase CineService segunda parte

Uso de Logger

En esta clase, se hace uso de SLF4J (a través de Logger y LoggerFactory) para registrar eventos importantes dentro del servicio. El uso de logs es muy útil en aplicaciones distribuidas, ya que permite rastrear el comportamiento del sistema, identificar errores y analizar el rendimiento de las operaciones. En este caso, se utilizan tres niveles de log: info para registros de operaciones exitosas, warn para advertencias cuando hay problemas, y error en caso de fallos graves.

Sincronización

Es importante destacar que todos los métodos que modifican o leen el estado compartido de la aplicación (como boletosDisponibles y ventas) están marcados con la palabra clave synchronized. Esto garantiza que solo un hilo pueda acceder a estas secciones del código al mismo tiempo, evitando problemas de concurrencia que podrían ocurrir si múltiples hilos intentaran modificar el número de boletos disponibles simultáneamente.

Boleto.java

Después de realizar lo anterior tenemos el código Boleto.java, el cual corresponde a la clase Boleto, que representa un boleto de cine en el contexto de un servicio web para la venta y gestión de boletos. La clase es simple pero importante, ya que almacena información relevante sobre los boletos comprados por los usuarios, como el nombre del cliente y la cantidad de boletos adquiridos. Esta clase será utilizada dentro de la lógica de negocio para gestionar las ventas y operaciones relacionadas con los boletos.

La clase Boleto está ubicada en el paquete com.servicioweb.cine.model, lo que indica que esta es una clase modelo, una estructura que representa los datos que manejaremos en el

sistema. Esta clase es esencial porque actúa como el contenedor de los datos que el servicio web va a manejar para registrar las compras de boletos por parte de los usuarios.

Atributos

Dentro de la clase, se definen dos atributos principales:

private String cliente; Este atributo almacena el nombre del cliente que realiza la compra de boletos. Es de tipo String, lo que significa que puede almacenar cualquier secuencia de caracteres. Es un campo muy importante porque permite asociar las compras de boletos con los usuarios. Dado que un cliente puede comprar múltiples boletos en diferentes momentos, este atributo se usa para identificar al comprador de manera única en cada transacción.

private int cantidad; Este atributo almacena la cantidad de boletos que un cliente ha comprado. Es de tipo int (entero), ya que la cantidad de boletos será siempre un número entero. Este valor es esencial para llevar un registro preciso de cuántos boletos se venden y cómo afecta esto al inventario de boletos disponibles.

Ambos atributos son privados, lo que significa que no se pueden acceder directamente desde fuera de la clase. Esta es una buena práctica de encapsulamiento, que ayuda a mantener el control sobre cómo se manipulan estos datos. Los métodos públicos (getters y setters) permiten modificar o acceder a estos atributos de manera controlada.

Constructores

Constructor sin parámetros (public Boleto() {}) Este es un constructor por defecto, que no recibe ningún argumento. Es útil cuando se necesita crear un objeto de la clase Boleto sin proporcionar valores iniciales en el momento de la creación. Este constructor puede ser utilizado por frameworks o bibliotecas que requieren crear instancias de clases sin inicializar atributos de manera explícita, o en situaciones donde el objeto Boleto se llena de datos después de su creación.

Constructor con parámetros (public Boleto(String cliente, int cantidad)) Este constructor recibe dos parámetros: cliente (de tipo String) y cantidad (de tipo int). Es el constructor principal de la clase, y se utiliza para crear un objeto Boleto ya inicializado con los datos de un cliente y la cantidad de boletos que ha comprado. Este constructor es útil cuando se conoce la información de la compra en el momento de crear el objeto, por ejemplo, cuando un cliente realiza una compra a través de una solicitud HTTP en el servicio web.

Métodos Getters y Setters

Los métodos getters y setters proporcionan una manera de acceder y modificar los atributos de la clase Boleto.

public String getCliente() Este método es un getter para el atributo cliente. Permite obtener el valor del nombre del cliente. Dado que el atributo cliente es privado, no se puede acceder directamente desde fuera de la clase, pero este método público permite leer su valor. Este método es importante para acceder al nombre del cliente cuando se registran las compras de boletos, por ejemplo, al mostrar un historial de ventas o al verificar los detalles de una compra.

public void setCliente(String cliente) Este método es un setter para el atributo cliente. Permite modificar el nombre del cliente después de que el objeto haya sido creado. El método recibe un argumento de tipo String, que es el nuevo valor que se asignará a cliente. Este método es útil cuando se necesita cambiar el nombre del cliente por alguna razón, aunque en el contexto de un sistema de compras de boletos, el nombre del cliente probablemente no cambiará después de realizar la compra.

*public int getCantidad()*Este método es otro getter, pero en este caso para el atributo cantidad. Permite obtener el número de boletos que fueron comprados. Es útil cuando se necesita conocer cuántos boletos han sido adquiridos, por ejemplo, al consultar el historial de ventas o al verificar la información de una compra.

public void setCantidad(int cantidad) Este método es un setter para el atributo cantidad. Permite modificar la cantidad de boletos que fueron comprados. El método recibe un argumento de tipo int, que es el nuevo valor que se asignará a cantidad. Aunque en el contexto de compras de boletos este valor probablemente no cambie una vez que se ha realizado la compra, este método proporciona flexibilidad para modificarlo si fuera necesario en un caso especial.

```java
package com.servicioweb.cine.model;

public class Boleto {
    private String cliente;
    private int cantidad;

    public Boleto() {}

    public Boleto(String cliente, int cantidad) {
        this.cliente = cliente;
        this.cantidad = cantidad;
    }

    public String getCliente() {
        return cliente;
    }

    public void setCliente(String cliente) {
        this.cliente = cliente;
    }

    public int getCantidad() {
        return cantidad;
    }

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

Figura 14. Clase Boleto

ServicioWebCineApplication.java

En esta parte de la practica se realizó el archivo ServicioWebCineApplication.java que es la clase principal de la aplicación Spring Boot llamada que gestiona la venta de boletos de cine a través de un servicio web. Utiliza la anotación @SpringBootApplication para configurar automáticamente la aplicación, iniciando el servidor web embebido y configurando los componentes necesarios. Además, la anotación @EnableScheduling habilita la ejecución de tareas programadas, como la generación automática de boletos, lo que permite realizar tareas periódicas de manera automática.

El método principal main arranca la aplicación, configurando y ejecutando el ciclo de vida de Spring Boot. Esta clase facilita la creación de la infraestructura del servicio web y la

automatización de procesos, como la actualización del inventario de boletos, haciendo que la aplicación sea más eficiente y fácil de mantener.

```java
package com.servicioweb.cine;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableScheduling;

@SpringBootApplication
@EnableScheduling  // Habilita la ejecución de tareas programadas

public class ServicioWebCineApplication {

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

Figura 15. Clase principal ServicioWebAplication de la aplicación

CorsConfig.java

Me encontré con la necesidad de permitir que diferentes aplicaciones web en distintos dominios pudieran comunicarse entre sí sin problemas, ya que implemente un HTML, es por eso que para lograr esto, utilicé una configuración de CORS (Cross-Origin Resource Sharing) en Spring Framework, ya que es una de las herramientas más comunes para gestionar este tipo de interacciones entre aplicaciones distribuidas.

La clase que implementé se llama CorsConfig y está anotada con @Configuration, lo que indica que es una clase de configuración dentro de Spring. Dentro de esta clase, creé un bean de tipo WebMvcConfigurer, que permite personalizar la configuración del manejo de las solicitudes HTTP. En el método addCorsMappings, especifiqué que todas las rutas de la aplicación (/**) deben permitir solicitudes desde cualquier origen, utilizando. allowedOrigins("*"). Esto significa que no hay restricción de dominios, permitiendo que cualquier aplicación pueda hacer peticiones a nuestro servicio web.

Además, definí que los métodos HTTP permitidos son GET, POST, PUT y DELETE, lo que cubre las operaciones básicas para interactuar con recursos a través de la web. Esta configuración fue fundamental para asegurar que las aplicaciones distribuidas, que podrían estar ubicadas en diferentes dominios, pudieran comunicarse de forma segura y eficiente sin bloquearse por restricciones de CORS. Esto es algo muy común cuando estamos trabajando con arquitecturas de microservicios o servicios web que necesitan interactuar entre diferentes entornos.

```java
package com.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class CorsConfig {

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

Figura 16. Implementación de la clase CorsConfig

CineUsuario.html

Para realizar las pruebas desarrollé una interfaz sencilla para la compra de boletos de cine utilizando HTML, CSS y JavaScript. La idea principal era permitir que los usuarios ingresaran su nombre y la cantidad de boletos que deseaban comprar, y que esta información se enviara a un servicio web para procesar la compra.

El código HTML define una estructura básica con un formulario donde los usuarios pueden ingresar su nombre y la cantidad de boletos. Los estilos CSS mejoran la apariencia de la interfaz, haciéndola más atractiva y adaptable a diferentes dispositivos. Se utilizan colores llamativos y efectos en los botones para mejorar la experiencia del usuario.

En la parte de JavaScript, implementé la función comprarBoletos(), que se encarga de capturar los datos ingresados por el usuario y enviarlos a un servicio web mediante una petición fetch(). La petición se realiza a http://localhost:8888/cine/comprar, utilizando el método POST, lo que indica que estamos enviando datos al servidor para procesar la compra. Si la compra es exitosa, el mensaje de respuesta del servidor se muestra en la página, y los campos del formulario se limpian automáticamente.

```
<script>
  function comprarBoletos() {
    const cliente = document.getElementById("cliente").value;
    const cantidad = document.getElementById("cantidad").value;

    if (!cliente || !cantidad) {
      document.getElementById('resultado').innerText = 'Por favor, ingresa todos los datos.';
      return;
    }

    fetch(`http://localhost:8888/cine/comprar?cliente=${cliente}&cantidad=${cantidad}`, {
      method: 'POST',
    })
    .then(response => response.text())
    .then(data => {
      document.getElementById('resultado').innerText = data;

      // Limpiar los campos de entrada después de la compra
      document.getElementById("cliente").value = '';
      document.getElementById("cantidad").value = '';
    })
    .catch(error => {
      document.getElementById('resultado').innerText = 'Error: ' + error;
    });
  }
</script>
```

Figura 17. Implementación del Script en la pagina HTML

Una de las principales ventajas de este enfoque es que permite la integración con un backend distribuido que gestiona la venta de boletos en tiempo real. Además, al usar JavaScript y fetch(), evitamos recargar la página, logrando una experiencia más fluida para el usuario. Este ejercicio me ayudó a comprender cómo conectar una interfaz web con un servicio distribuido y manejar las respuestas del servidor de manera dinámica.

Figura 18. Vista para usuario final

Resultados

En nuestro sistema web de gestión de boletos para cine, desarrollado con Spring Boot ha demostrado un funcionamiento estable y eficiente, permitiendo la administración del inventario de boletos y la compra de entradas de manera concurrente. A través de la implementación de una API REST, los clientes pueden interactuar con el sistema en tiempo real, consultar la disponibilidad de boletos y realizar compras sin inconvenientes.

Durante las pruebas realizadas, se verificó que el sistema maneja correctamente múltiples solicitudes simultáneas sin afectar la integridad de los datos. Cada compra realizada impacta de inmediato en la disponibilidad de boletos, registrándose en la lista de ventas y reflejando los cambios en el backend. La API responde adecuadamente a cada petición, validando que los clientes no intenten adquirir más boletos de los disponibles y asegurando la coherencia en las transacciones.

Además, la tarea programada de generación automática de boletos funciona correctamente, agregando boletos al inventario en los intervalos definidos (cada 60 segundos). Se comprobó que el sistema gestiona compras incluso cuando el stock es insuficiente, notificando al usuario sobre la falta de boletos disponibles y evitando inconsistencias en la base de datos.

El servidor, ejecutado en el puerto 8080, registra cada solicitud recibida mediante logs detallados, mostrando información sobre las compras exitosas y advertencias en caso de intentos fallidos. A continuación, se presentan capturas de pantalla de las pruebas realizadas con Postman y el frontend, evidenciando el correcto funcionamiento del sistema.

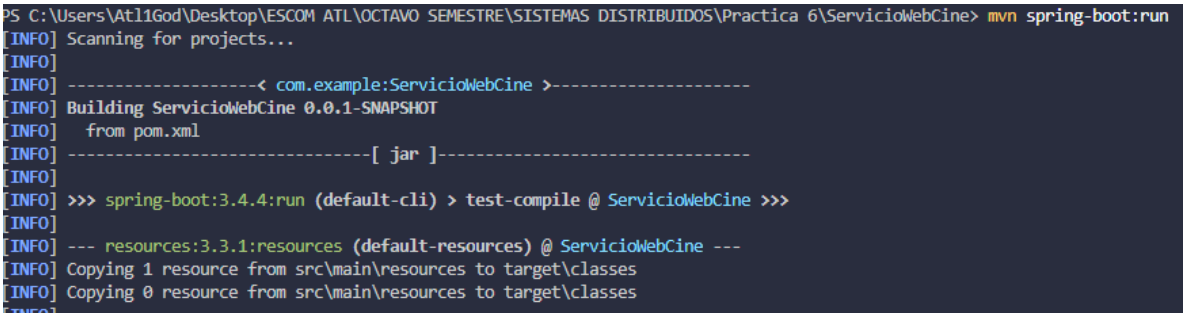

Figura 19. Ejecución del proyecto

Pruebas con Postman

En primer lugar, se probó el método **GET** para obtener la cantidad de boletos disponibles en el cine. Para ello, se realizó una solicitud a la URL: http://localhost:8080/cine/boletos

La respuesta en formato **JSON** contenía el número exacto de boletos disponibles en ese momento, confirmando que el sistema responde correctamente y que los datos reflejan el estado actual del inventario.

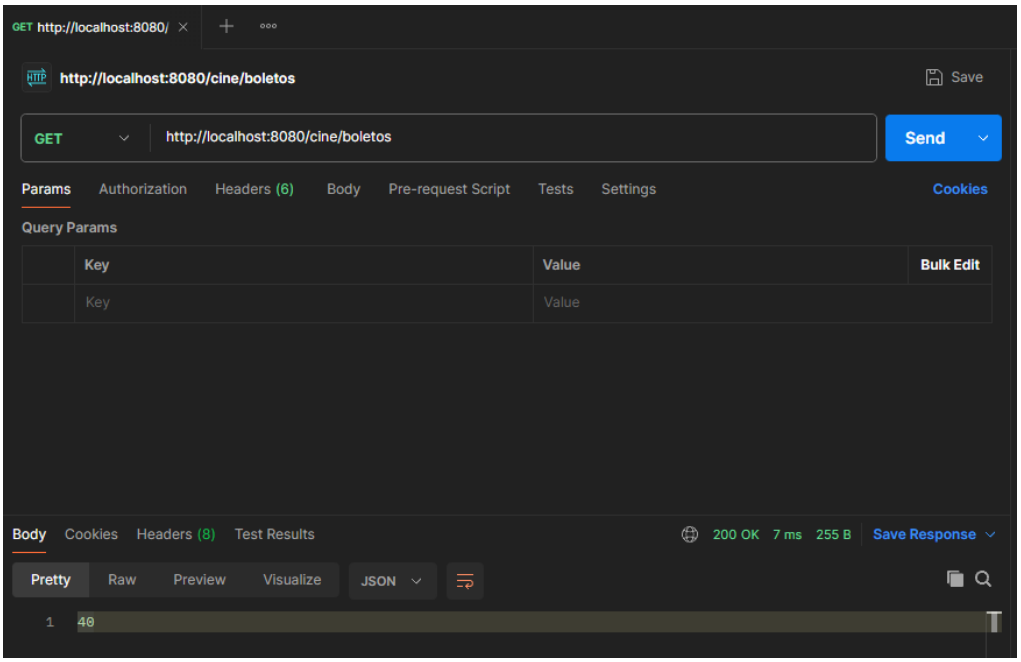

Figura 20. Prueba del método GET, para obtener el total de boletos

Posteriormente, se probó el método **POST**, que permite realizar la compra de boletos enviando el nombre del cliente junto con la cantidad deseada. Se envió una solicitud a: http://localhost:8080/cine/comprar

La respuesta del servidor confirmó que la compra se realizó con éxito y se verificó que el sistema descontó correctamente la cantidad adquirida del stock total.

Además, en la terminal del servidor se registró un **log** con la compra efectuada, mostrando el nombre del cliente, la cantidad de boletos comprados y los boletos restantes.

Asimismo, se comprobó que, si la cantidad de boletos solicitada superaba el stock disponible, el sistema devolvía un mensaje de error, evitando así transacciones inválidas. En este caso, se recibió la siguiente respuesta del servidor:

Para comprobar el correcto funcionamiento de la tarea programada de generación de boletos, se monitoreó el sistema durante varios minutos. Se verificó que cada 60 segundos se agregaban automáticamente 10 boletos al inventario, recibiendo en la terminal del servidor el siguiente mensaje:

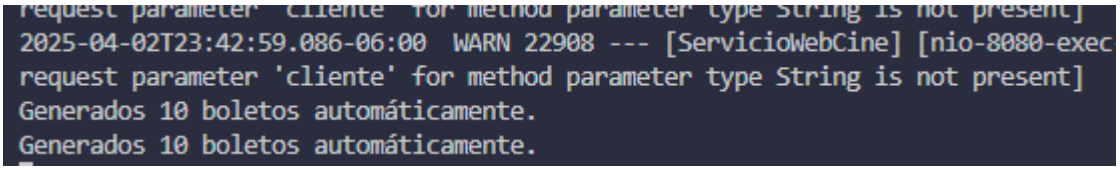

Figura 21. Generación de boletos

Después de la generación automática de boletos, se realizó nuevamente una solicitud **GET** para confirmar que el inventario reflejaba los cambios esperados. Se constató que el sistema mantiene una gestión precisa de la disponibilidad de boletos, actualizando correctamente el stock tras cada operación de compra y reabastecimiento.

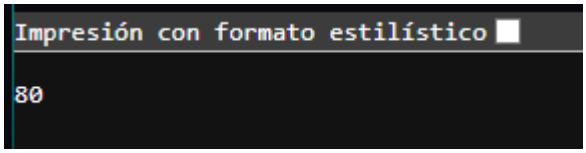

Figura 22. Boletos actualizados

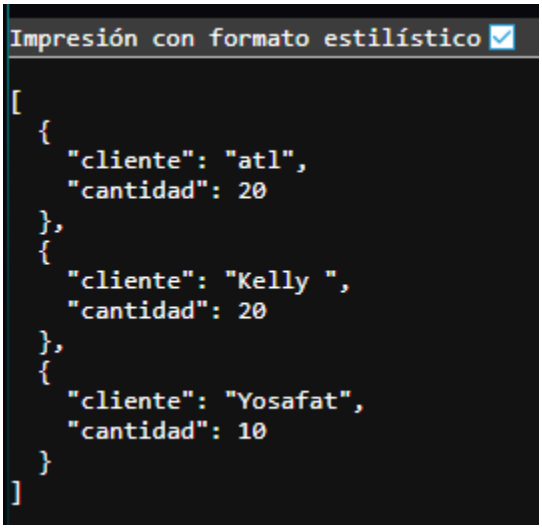

Figura 23. Bitácora de Clientes y sus compras

Para validar la comunicación entre el frontend y el backend, se realizó una prueba ejecutando la interfaz gráfica diseñada para el sistema de cine. Al abrir la página en un navegador, se mostró correctamente la cantidad de boletos disponibles, lo que confirmó que la integración entre la API REST y la interfaz web se estableció con éxito.

Figura 24. Frontend prueba

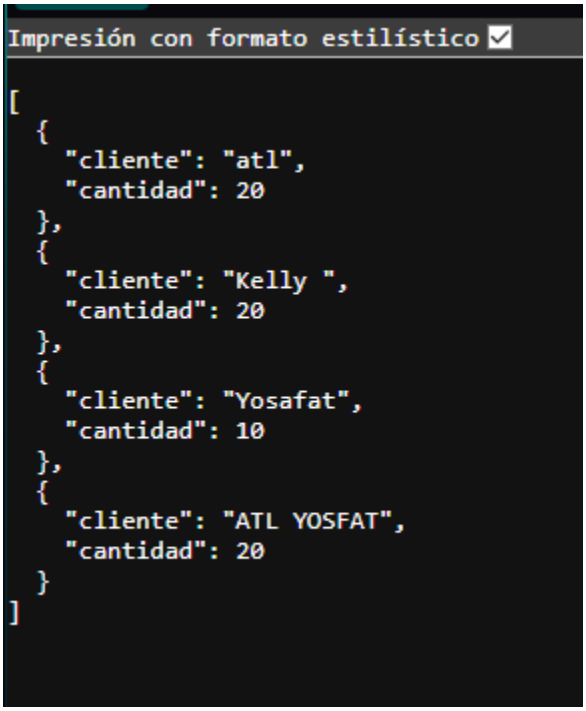

Figura 25. Bitácora en donde se comprueba la conexión exitosa del front

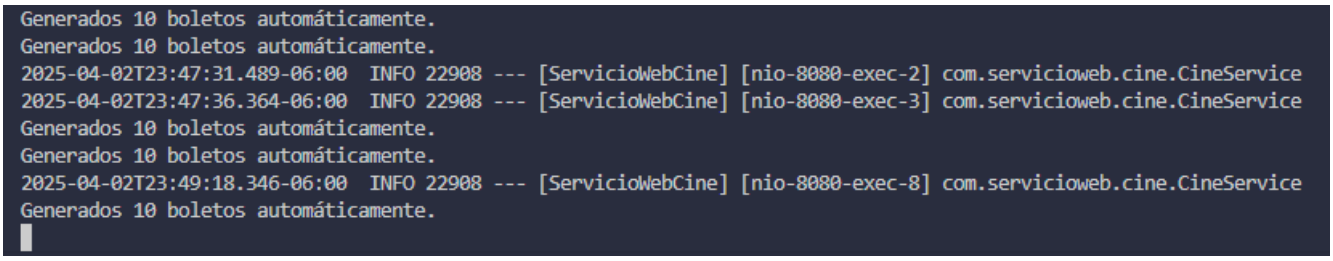

Figura 26. Registro en la terminal

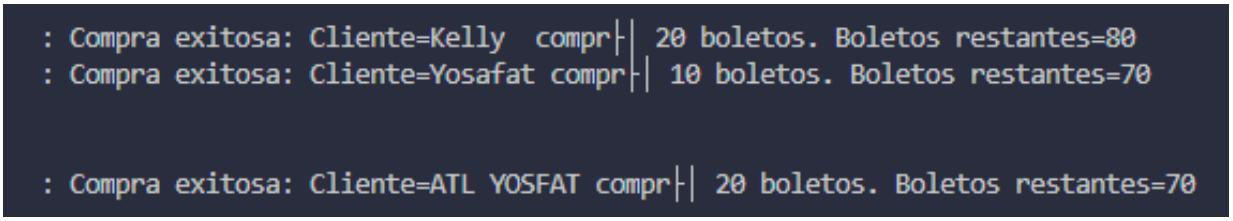

Figura 27 Registro exitoso en terminal

Conclusión

La implementación del servicio web para la gestión de boletos de cine permitió afianzar conocimientos fundamentales sobre los sistemas distribuidos y su aplicación en entornos reales. A lo largo del desarrollo de la práctica, se trabajó con múltiples conceptos clave, como la concurrencia, la escalabilidad, la comunicación cliente-servidor mediante una API REST y el manejo de excepciones, elementos esenciales para la construcción de aplicaciones distribuidas robustas y eficientes.

Uno de los aspectos más relevantes de esta práctica fue la implementación de un servicio web basado en Spring Boot, lo que facilitó la estructuración del código y la exposición de endpoints para la interacción con los clientes. Se desarrolló un controlador (CineController) que proporciona operaciones clave como la compra de boletos, la consulta de disponibilidad y el registro de ventas. Para asegurar la correcta gestión de los datos, se utilizó un servicio (CineService) con métodos sincronizados que garantizan la integridad y consistencia de la información, evitando problemas de concurrencia en un entorno distribuido donde múltiples usuarios pueden acceder al sistema simultáneamente.

Otro punto destacable fue la automatización de procesos mediante la anotación @Scheduled, la cual permitió la generación automática de boletos en intervalos de tiempo definidos. Este mecanismo representa un caso real de actualización de recursos sin intervención manual, optimizando la disponibilidad de boletos y mejorando la eficiencia del sistema.

En términos de seguridad y accesibilidad, se implementó una configuración CORS (Cross-Origin Resource Sharing), la cual permite que el servicio sea consumido desde cualquier origen sin restricciones, facilitando su integración con distintas aplicaciones web o móviles. Además, se incorporó un manejador global de excepciones (GlobalExceptionHandler), que

permite capturar errores y devolver respuestas HTTP adecuadas, mejorando la experiencia del usuario al proporcionar mensajes claros sobre posibles fallos en la solicitud.

El uso de Spring Boot y su arquitectura modular facilitó la organización del código y su mantenibilidad, permitiendo separar responsabilidades en diferentes componentes. Este enfoque no solo mejora la claridad del sistema, sino que también facilita futuras expansiones, como la integración con bases de datos o la escalabilidad en entornos cloud.

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].*

Anexos

Código Fuente CorsConfig.java

```
package com.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.web.servlet.config.annotation.CorsRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer
;

@Configuration
public class CorsConfig {

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

Código Fuente Boleto.java

```
package com.servicioweb.cine.model;

public class Boleto {
    private String cliente;
    private int cantidad;

    public Boleto() {}
```

```
    public Boleto(String cliente, int cantidad) {
        this.cliente = cliente;
        this.cantidad = cantidad;
    }

    public String getCliente() {
        return cliente;
    }

    public void setCliente(String cliente) {
        this.cliente = cliente;
    }

    public int getCantidad() {
        return cantidad;
    }

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

Código Fuente CineController.java

```
package com.servicioweb.cine;

import org.springframework.web.bind.annotation.*;
import java.util.List;
import com.servicioweb.cine.model.Boleto;

@CrossOrigin(origins = "*") // Permitir solicitudes desde
cualquier origen
@RestController
@RequestMapping("/cine")
public class CineController {
    private final CineService cineService;
```

```
    public CineController(CineService cineService) {
        this.cineService = cineService;
    }

    @PostMapping("/agregar")
    public String agregarBoletos(@RequestParam int cantidad) {
        cineService.agregarBoletos(cantidad);
        return cantidad + " boletos agregados exitosamente.";
    }

    @GetMapping("/boletos")
    public int obtenerBoletosDisponibles() {
        return cineService.obtenerBoletosDisponibles();
    }

    @PostMapping("/comprar")
    public String comprarBoletos(@RequestParam String cliente,
@RequestParam int cantidad) {
        if (cliente == null || cliente.isBlank()) {
            throw new IllegalArgumentException("El nombre del
cliente no puede estar vacío.");
        }
        if (cantidad <= 0) {
            throw new IllegalArgumentException("La cantidad de
boletos debe ser mayor a cero.");
        }
        return cineService.comprarBoleto(cliente, cantidad);
    }

    @GetMapping("/ventas")
    public List<Boleto> obtenerVentas() {
        return cineService.obtenerVentas();
    }
}
```

Código Fuente CineService.java

```
package com.servicioweb.cine;
```

```java
import com.servicioweb.cine.model.Boleto;
import org.springframework.stereotype.Service;
import org.springframework.scheduling.annotation.Scheduled;
import java.util.ArrayList;
import java.util.List;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Service
public class CineService {
    private static final Logger logger =
LoggerFactory.getLogger(CineService.class);
    private int boletosDisponibles = 30;
    private List<Boleto> ventas = new ArrayList<>();

        public synchronized void agregarBoletos(int cantidad) {
            boletosDisponibles += cantidad;
        }

        @Scheduled(fixedRate = 60000)
        public synchronized void generarBoletosAutomaticos() {
            int boletosGenerados = 10;
            boletosDisponibles += boletosGenerados;
            System.out.println("Generados " + boletosGenerados + "
boletos automáticamente.");
        }
    public synchronized int obtenerBoletosDisponibles() {
        return boletosDisponibles;
    }

    public synchronized String comprarBoleto(String cliente, int
cantidad) {
        if (boletosDisponibles < cantidad) {
            logger.warn("Compra fallida: Cliente={} intentó
comprar {} boletos. Boletos disponibles={}", cliente, cantidad,
boletosDisponibles);
            return "No hay suficientes boletos disponibles.";
        }
        boletosDisponibles -= cantidad;
```

```
        ventas.add(new Boleto(cliente, cantidad));
        logger.info("Compra exitosa: Cliente={} compró {} boletos.
Boletos restantes={}", cliente, cantidad, boletosDisponibles);
        return "Compra exitosa. Boletos restantes: " +
boletosDisponibles;
    }

    public List<Boleto> obtenerVentas() {
        return ventas;
    }
}
```

Código Fuente ServicioWebCineApplication.java

```
package com.servicioweb.cine;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableScheduling;

@SpringBootApplication
@EnableScheduling  // Habilita la ejecución de tareas programadas

public class ServicioWebCineApplication {

    public static void main(String[] args) {
        SpringApplication.run(ServicioWebCineApplication.class,
args);
    }

}
```

Código Fuente GoblaExceptionHandler.java

```
package com.servicioweb.cine.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
```

```
import org.springframework.web.bind.annotation.ExceptionHandler;
import
org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String>
handleIllegalArgumentException(IllegalArgumentException ex) {
        return new ResponseEntity<>(ex.getMessage(),
HttpStatus.BAD_REQUEST);
    }
}
```

Código Fuente CineUsuario.html

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
  <title>Compra de Boletos - Cine</title>
  <style>
    body {
      font-family: 'Arial', sans-serif;
      background-color: #f4f4f4;
      margin: 0;
      padding: 0;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      flex-direction: column;
    }
```

```css
h1 {
  color: #2c3e50;
  font-size: 2.5rem;
  margin-bottom: 20px;
}

h3 {
  font-size: 1.2rem;
  color: #27ae60;
  margin-top: 20px;
}
label {
  font-size: 1rem;
  color: #34495e;
  margin-bottom: 10px;
  display: inline-block;
}

input[type="text"], input[type="number"] {
  width: 250px;
  padding: 10px;
  margin: 10px 0;
  border: 2px solid #bdc3c7;
  border-radius: 5px;
  font-size: 1rem;
  background-color: #ecf0f1;
}

input[type="text"]:focus, input[type="number"]:focus {
  outline: none;
  border-color: #3498db;
  background-color: #ffffff;
}

button {
  padding: 10px 20px;
  background-color: #3498db;
  color: #fff;
```

```
      border: none;
      border-radius: 5px;
      font-size: 1rem;
      cursor: pointer;
      transition: background-color 0.3s;
    }

    button:hover {
      background-color: #2980b9;
    }
    #resultado {
      font-size: 1.1rem;
      color: #e74c3c;
      text-align: center;
      margin-top: 20px;
    }
    @media (max-width: 600px) {
      input[type="text"], input[type="number"] {
        width: 100%;
      }

      button {
        width: 100%;
      }
    }
  </style>
</head>
<body>
  <h1>Compra de Boletos</h1>

  <label for="cliente">Cliente:</label>
  <input type="text" id="cliente" placeholder="Ingresa tu nombre"
/><br />

  <label for="cantidad">Cantidad de Boletos:</label>
  <input type="number" id="cantidad" placeholder="Cantidad de
boletos" /><br />

  <button onclick="comprarBoletos()">Comprar</button>
```

```
  <h3 id="resultado"></h3>

  <script>
    function comprarBoletos() {
      const cliente = document.getElementById("cliente").value;
      const cantidad = document.getElementById("cantidad").value;

      if (!cliente || !cantidad) {
        document.getElementById('resultado').innerText = 'Por
favor, ingresa todos los datos.';
        return;
      }

      fetch(`http://localhost:8080/cine/comprar?cliente=${cliente}
&cantidad=${cantidad}`, {
        method: 'POST',
      })
      .then(response => response.text())
      .then(data => {
        document.getElementById('resultado').innerText = data;

        // Limpiar los campos de entrada después de la compra
        document.getElementById("cliente").value = '';
        document.getElementById("cantidad").value = '';
      })
      .catch(error => {
        document.getElementById('resultado').innerText = 'Error: '
+ error;
      });
    }
  </script>
</body>
</html>
```