



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Actividad:

Practica 7 – Microservicios

Integrante:

Hernández Vázquez Jorge Daniel

Profesor:

Chadwick Carreto Arellano

Fecha de entrega:

16/04/2025

INSTITUTO POLITÉCNICO NACIONAL



Índice de contenido

Antecedente.....	1
Microservicios.....	1
Diferencias con la Arquitectura Monolítica	2
Características de los Microservicios	3
Ventajas y desventajas de los Microservicios	4
Casos de Uso y Aplicaciones de Microservicios en Sistemas Distribuidos	5
Servicios RESTful.....	8
Planteamiento del problema	13
Propuesta de solución.....	14
Materiales y métodos empleados	16
Desarrollo de la solución.....	19
Creación de proyectos en Spring Boot.....	19
Instalación de Maven	22
Microservicio: inventario-service	23
Microservicio: compra-service.....	28
Microservicio: gateway-service	34
Microservicio: eureka-server.....	38
Microservicio: cliente-service (Frontend).....	39
Construcción y despliegue con Docker.....	46
Instrucciones para ejecutar el proyecto	51
Resultados	54
Conclusión.....	63
Bibliografía	65
Anexos.....	66

Índice de Figuras

Figura 1	2
Figura 2	5
Figura 3	9
Figura 4	10
Figura 5	13
Figura 6	20
Figura 7	21

Figura 8	21
Figura 9	22
Figura 10	22
Figura 11	22
Figura 12	23
Figura 13	23
Figura 14	24
Figura 15	24
Figura 16	25
Figura 17	26
Figura 18	26
Figura 19	27
Figura 20	28
Figura 21	29
Figura 22	30
Figura 23	31
Figura 24	32
Figura 25	34
Figura 26	35
Figura 27	35
Figura 28	35
Figura 29	36
Figura 30	37
Figura 31	37
Figura 32	38
Figura 33	38
Figura 34	39
Figura 35	40
Figura 36	40
Figura 37	41
Figura 38	42
Figura 39	43
Figura 40	43
Figura 41	45
Figura 42	47

Figura 43	47
Figura 43	54
Figura 44	55
Figura 45	55
Figura 46	56
Figura 47	56
Figura 48	57
Figura 49	57
Figura 50	58
Figura 51	58
Figura 52	58
Figura 53	59
Figura 54	60
Figura 55	60
Figura 56	61
Figura 57	61
Figura 58	62
Figura 59	62

Antecedente

La arquitectura de microservicios ha transformado el desarrollo de aplicaciones distribuidas al promover la creación de sistemas compuestos por componentes pequeños, autónomos y desplegables de forma independiente. Esta evolución surge como respuesta a las limitaciones de los enfoques monolíticos, especialmente en términos de escalabilidad, mantenimiento y despliegue continuo. Al dividir una aplicación en servicios independientes, cada uno encargado de una funcionalidad específica, se facilita el desarrollo colaborativo, la adopción de diferentes tecnologías por servicio y una respuesta más ágil ante cambios en los requerimientos del negocio. Esta arquitectura aprovecha protocolos ligeros y estándares comunes, como HTTP/REST y mensajería asíncrona, para lograr una comunicación efectiva entre servicios, promoviendo la interoperabilidad, la resiliencia y la escalabilidad horizontal en entornos modernos como la nube.

Microservicios

La arquitectura de microservicios es un estilo de diseño de software que estructura una aplicación como un conjunto de servicios pequeños, autónomos y desplegables de manera independiente. Cada microservicio está diseñado para cumplir con una funcionalidad específica del sistema global, ejecutándose en su propio proceso y comunicándose con otros microservicios mediante protocolos ligeros, generalmente HTTP con APIs RESTful o mediante sistemas de mensajería asíncrona.

Una característica fundamental 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 continua y el despliegue continuo (CI/CD). Además, cada microservicio puede estar desarrollado con diferentes tecnologías o lenguajes de programación, siempre y cuando cumpla con el contrato de comunicación establecido.

Los microservicios representan una evolución hacia una arquitectura más modular, enfocada en la separación de responsabilidades y la autonomía de cada componente del sistema.

El concepto de microservicios no surgió de manera repentina, sino que evolucionó como respuesta a las limitaciones de los sistemas monolíticos y las necesidades cambiantes de escalabilidad, mantenibilidad y velocidad en el desarrollo de software moderno. Sus orígenes pueden rastrearse hacia prácticas arquitectónicas más antiguas como la arquitectura orientada a servicios (SOA), que promovía la construcción de aplicaciones basadas en servicios reusables y loosely coupled (débilmente acoplados).

Durante la década de 2000, compañías como Amazon y Netflix comenzaron a enfrentar retos significativos al escalar sus arquitecturas monolíticas. Estos retos incluían tiempos largos de despliegue, dificultades para realizar cambios en partes específicas del sistema sin afectar otras, y problemas de escalabilidad horizontal. En respuesta, iniciaron la descomposición de sus aplicaciones en servicios más pequeños y autónomos. Esta práctica, inicialmente no estandarizada, fue ganando popularidad y, eventualmente, el término "microservicios" comenzó a utilizarse formalmente alrededor de 2011.

En 2014, los microservicios empezaron a consolidarse como un paradigma arquitectónico claramente definido, gracias a publicaciones, conferencias y libros técnicos que establecieron sus principios fundamentales. Hoy en día, la arquitectura de microservicios es ampliamente adoptada por empresas de todos los tamaños, apoyada por tecnologías como Docker, Kubernetes, Spring Boot, y plataformas

en la nube que facilitan su implementación. En la siguiente Figura podemos ver un ejemplo de la estructura de un microservicio.

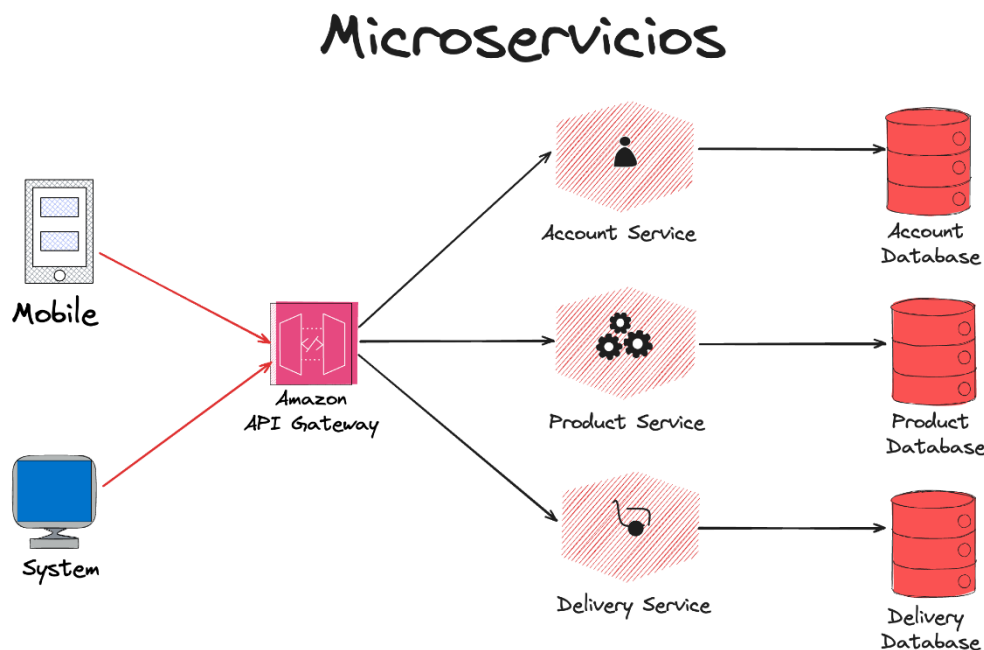


Figura 1. Ejemplo de estructura de un microservicio.

Diferencias con la Arquitectura Monolítica

La arquitectura monolítica es un modelo tradicional de diseño de aplicaciones en el que todos los componentes funcionales están agrupados y ejecutados como una sola unidad o proceso. En contraste, la arquitectura de microservicios promueve la descomposición de la aplicación en múltiples servicios independientes, cada uno con su propia lógica y base de datos (opcionalmente).

Principales diferencias:

Característica	Arquitectura Monolítica	Arquitectura de Microservicios
Estructura	Una sola aplicación grande	Conjunto de servicios independientes
Despliegue	Se despliega como una sola unidad	Cada servicio se despliega por separado
Escalabilidad	Escalado vertical de toda la aplicación	Escalado horizontal específico por servicio
Desarrollo	Equipos trabajan sobre el mismo código base	Equipos pueden trabajar de forma independiente
Tecnologías	Limitado a un stack común	Puede usar diferentes lenguajes o tecnologías
Fallos	Un error puede afectar toda la aplicación	Fallos suelen estar aislados a servicios individuales
Tiempos de despliegue	Lento, complejo, requiere pruebas integrales	Rápido y frecuente, favorece CI/CD

Mantenibilidad	Difícil conforme crece la base de código	Más fácil gracias a la separación de responsabilidades
-----------------------	--	--

Estas diferencias hacen que la arquitectura de microservicios sea especialmente atractiva para sistemas que requieren alta escalabilidad, modularidad y adaptabilidad en entornos de desarrollo ágil y despliegue continuo.

Características de los Microservicios

Y en el contexto de los sistemas distribuidos, la arquitectura de microservicios representa una evolución significativa frente a los enfoques monolíticos tradicionales. Su diseño se basa en la descomposición de una aplicación en múltiples servicios pequeños, independientes y especializados, que se comunican entre sí a través de redes. Este enfoque permite construir sistemas más flexibles, escalables y resistentes a fallos, adaptándose mejor a entornos dinámicos y exigentes.

A continuación, se presentan las principales características que definen a los microservicios dentro de un sistema distribuido:

- **Autonomía:** Cada microservicio funciona de manera independiente, con su propio ciclo de vida, lógica de negocio y almacenamiento de datos. Esto facilita el mantenimiento, la escalabilidad y la resiliencia del sistema.
- **Desacoplamiento:** Los servicios están diseñados para minimizar dependencias entre sí. Este bajo acoplamiento permite que los cambios en un servicio no afecten a los demás, promoviendo una evolución más ágil del sistema.
- **Escalabilidad independiente:** Los microservicios pueden escalarse de manera individual según la demanda de cada uno, lo que mejora el uso eficiente de recursos en sistemas distribuidos.
- **Especialización funcional:** Cada servicio aborda una única función o dominio del negocio, lo que mejora la claridad del código, la modularidad y la reutilización.
- **Comunicación por red:** Los microservicios interactúan entre sí a través de protocolos ligeros como HTTP/REST o mediante sistemas de mensajería asíncrona (como RabbitMQ o Kafka), lo que habilita la distribución física de los servicios.
- **Despliegue independiente:** Al ser servicios autónomos, pueden desarrollarse, probarse y desplegarse por separado, lo que facilita prácticas como la integración y entrega continua (CI/CD).
- **Tolerancia a fallos:** Gracias a su distribución, el fallo de un microservicio no implica el colapso del sistema completo. Esto incrementa la disponibilidad y robustez general.
- **Persistencia distribuida:** Cada microservicio puede gestionar su propia base de datos, lo que refuerza su independencia, pero también plantea desafíos de sincronización y consistencia.
- **Tecnología heterogénea:** En sistemas distribuidos, los microservicios pueden desarrollarse en distintos lenguajes o entornos, lo que permite elegir la tecnología más adecuada para cada caso particular.

Estas características hacen de los microservicios una arquitectura ideal para sistemas distribuidos modernos, orientados a la escalabilidad, flexibilidad y capacidad de evolución continua en entornos de alta demanda.

Ventajas y desventajas de los Microservicios

Como ya hemos visto, la arquitectura de microservicios se ha consolidado como una de las principales alternativas para el desarrollo de sistemas modernos, especialmente aquellos que requieren alta escalabilidad, flexibilidad y mantenibilidad. Este enfoque propone dividir una aplicación en servicios pequeños, independientes y desplegables de manera autónoma, que se comunican entre sí para conformar un sistema completo.

Adoptar esta arquitectura conlleva una transformación tanto técnica como organizacional, permitiendo a los equipos de desarrollo adaptarse mejor a entornos cambiantes, implementar nuevas funcionalidades con mayor rapidez y escalar sus soluciones de forma eficiente. Sin embargo, también implica enfrentar desafíos asociados a la complejidad operativa y la gestión distribuida.

A continuación, se presentan los principales beneficios y retos que implica la adopción de una arquitectura basada en microservicios:

Beneficios técnicos y organizacionales

- **Modularidad:** Favorece el diseño basado en dominios, permitiendo construir software como un conjunto de componentes altamente cohesionados.
- **Escalabilidad específica:** Solo se escalan los servicios necesarios, optimizando recursos y costos.
- **Facilidad para el mantenimiento:** Gracias al desacoplamiento, los errores se aíslan fácilmente, y los cambios se aplican sin afectar el sistema completo.
- **Desarrollo paralelo:** Equipos distintos pueden trabajar simultáneamente en distintos microservicios, acelerando el desarrollo.
- **Elección tecnológica flexible:** Permite adoptar tecnologías diferentes para cada servicio según su necesidad particular.
- **Despliegue ágil y continuo:** Integración con prácticas DevOps para CI/CD eficientes.
- **Resiliencia:** Un fallo en un servicio no colapsa todo el sistema, mejorando la disponibilidad.

Retos en su implementación y mantenimiento

- **Complejidad de la infraestructura:** Se requiere una infraestructura robusta para gestionar múltiples servicios, incluyendo balanceo de carga, discovery, monitoreo, etc.
- **Comunicación entre servicios:** La necesidad de gestionar protocolos, formatos y latencias entre servicios puede añadir complejidad.
- **Gestión de datos distribuida:** Cada microservicio suele tener su propia base de datos, lo cual complica la consistencia transaccional.
- **Mayor sobrecarga operativa:** Supervisar, desplegar y versionar múltiples servicios exige herramientas adicionales y personal capacitado.

- **Pruebas complejas:** Las pruebas integradas deben considerar múltiples puntos de fallo y dependencia.
- **Seguridad distribuida:** Garantizar la autenticación, autorización y encriptación en múltiples puntos es más desafiante.

Casos de Uso y Aplicaciones de Microservicios en Sistemas Distribuidos

Las arquitecturas de microservicios han sido adoptadas por muchas empresas líderes en tecnología, particularmente aquellas que necesitan escalar rápidamente y manejar una enorme cantidad de usuarios y datos distribuidos. A continuación, se presentan algunos de los casos más destacados:

Netflix

Netflix es uno de los pioneros y principales referentes del uso de microservicios a gran escala. En sus inicios, la plataforma funcionaba sobre una arquitectura monolítica que se volvió insostenible ante el crecimiento exponencial de usuarios y la necesidad de desplegar nuevas funciones constantemente. Para abordar estos desafíos, Netflix migró a una arquitectura basada en cientos de microservicios, cada uno encargado de funcionalidades específicas como recomendaciones, reproducción de contenido, facturación, autenticación, etc.

Cada microservicio opera de forma independiente y se comunica a través de APIs y colas de mensajería. Además, emplean herramientas propias como **Eureka** para descubrimiento de servicios, **Ribbon** para balanceo de carga, y **Hystrix** para tolerancia a fallos, contribuyendo a una arquitectura resiliente y escalable. En la siguiente figura podemos observar la arquitectura que utiliza Netflix.

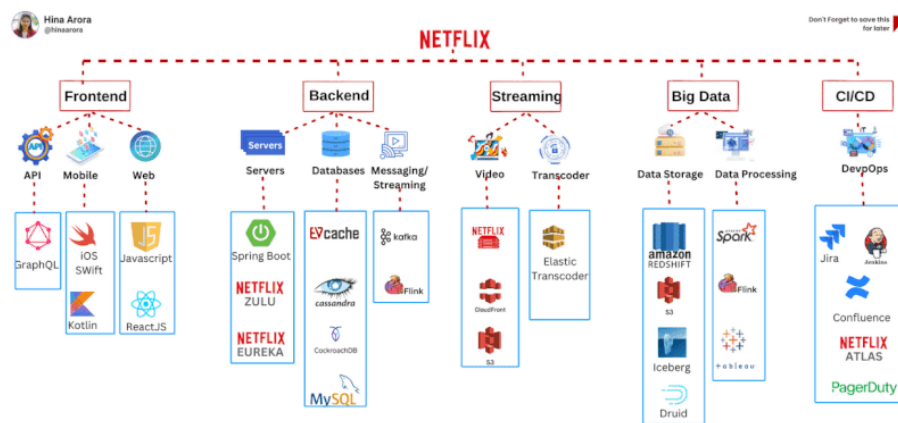


Figura 2. Arquitectura utilizada por Netflix.

Amazon

Amazon también adoptó los microservicios para responder a las crecientes demandas de su plataforma. Su sistema está segmentado en servicios independientes que manejan operaciones como gestión de productos, procesamiento de pagos, control de inventario, recomendaciones

personalizadas, etc. Esta descomposición permitió a Amazon realizar despliegues diarios (e incluso múltiples veces al día), manteniendo alta disponibilidad y estabilidad.

Cada equipo es responsable de sus propios microservicios, desde el desarrollo hasta la operación, aplicando la filosofía de "You build it, you run it".

Otros ejemplos:

- **Uber:** Utiliza microservicios para separar funciones como geolocalización, pagos, historial de viajes y mensajería.
- **Spotify:** Divide sus servicios por funcionalidades como reproducción de música, descubrimiento de artistas, gestión de usuarios, entre otros.
- **LinkedIn:** Usa microservicios para escalar partes críticas como el feed de noticias, recomendaciones y mensajería profesional.

Estos casos demuestran cómo los microservicios son clave para operar a gran escala, permitiendo una evolución constante, despliegue continuo y una mayor resistencia a fallos.

Es por ello por lo que, los microservicios se ajustan de forma ideal a entornos de **alta demanda** y metodologías de **desarrollo ágil** por diversas razones:

Alta demanda:

- **Escalabilidad horizontal:** Permite replicar únicamente los servicios más demandados, lo que mejora el rendimiento sin desperdiciar recursos.
- **Despliegues sin interrupciones:** Se pueden actualizar servicios específicos sin afectar la disponibilidad del sistema completo.
- **Balanceo de carga eficiente:** Es posible dirigir el tráfico a instancias saludables de servicios individuales.
- **Resiliencia ante fallos:** La independencia entre servicios limita el impacto de errores o cuellos de botella.

Desarrollo ágil:

- **Iteración rápida:** Equipos pequeños y multidisciplinarios pueden desarrollar, probar y lanzar nuevas funcionalidades de forma rápida y autónoma.
- **Integración con DevOps y CI/CD:** Facilita la automatización del ciclo de vida del software, desde la construcción hasta el monitoreo en producción.
- **Pruebas y validaciones modulares:** Es más fácil implementar pruebas unitarias e integración a nivel de servicio.

Gracias a estas capacidades, los microservicios se han convertido en una base tecnológica clave para empresas que adoptan estrategias de transformación digital, innovación continua y despliegue acelerado.

Comparación entre Modelos Arquitectónicos en Sistemas Distribuidos

A continuación, se presenta una tabla comparativa que destaca las diferencias más relevantes entre estos modelos que hemos realizado durante las practicas:

Característica	Cliente-Servidor	Multicliente-Multiservidor	Objetos Distribuidos (RMI)	Servicios Web	SOA	Microservicios
Estructura	1 servidor - 1 o varios clientes	Varios servidores y múltiples clientes	Invocación remota de objetos	Servicios accesibles vía HTTP	Servicios organizados por dominio	Servicios pequeños y autónomos
Comunicación	Socket TCP/IP	Socket TCP/IP	RMI (Java, serialización)	HTTP, SOAP o REST	SOAP sobre HTTP	REST, gRPC, mensajería
Acoplamiento	Alto	Alto	Medio (orientado a objetos)	Bajo	Medio	Muy bajo
Escalabilidad	Limitada	Mejorada por concurrencia	Limitada a la JVM	Alta (REST)	Media (dependiente del ESB)	Alta, escalado específico por servicio
Lenguajes compatibles	Generalmente homogéneo	Homogéneo	Java	Multilenguaje	Multilenguaje	Totalmente heterogéneo
Distribución geográfica	Limitada	Limitada	Moderada	Amplia (web)	Amplia	Totalmente distribuida
Gestión de estados	Dependiente del servidor	Con manejo concurrente	Por objeto	Generalmente sin estado	Opcional	Preferentemente sin estado
Reutilización de componentes	Baja	Baja	Media	Alta	Alta	Muy alta
Despliegue	Centralizado	Centralizado	Necesita compatibilidad Java	Fácil mediante contenedores	Requiere middleware (ESB)	Independiente por servicio
Tolerancia a fallos	Baja	Media	Baja	Alta (dependiendo del diseño)	Media (punto único en ESB)	Alta (fallo aislado por servicio)
Apropiado para	Aplicaciones pequeñas	Aplicaciones concurrentes	Sistemas Java distribuidos	Integración entre plataformas	Integración empresarial	Sistemas modernos y escalables
Complejidad de implementación	Baja	Media	Media/Alta	Media	Alta	Alta (infraestructura avanzada)

Esta evolución refleja una transición natural en el desarrollo de sistemas distribuidos:

- **Cliente-servidor** y **multicliente-multiservidor** sentaron las bases para la conexión en red y la atención a múltiples usuarios, aunque con un enfoque monolítico.
- **Objetos distribuidos** agregaron la capacidad de invocar métodos remotamente, facilitando la construcción de aplicaciones distribuidas orientadas a objetos, aunque limitadas a entornos homogéneos como Java.
- **Servicios Web** introdujeron la interoperabilidad entre plataformas, reduciendo el acoplamiento y promoviendo el uso de protocolos estandarizados.
- **SOA** formalizó la integración entre servicios empresariales a través de un bus central de servicios, aunque con limitaciones en escalabilidad y acoplamiento.
- Finalmente, **microservicios** perfeccionan estos modelos al ofrecer una arquitectura más modular, escalable, desacoplada y apta para despliegue ágil y continuo.

Servicios RESTful

Los servicios 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:

1. **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.
2. **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**.
3. **Uso de Recursos Identificables:** Cada recurso en un servicio RESTful es representado por una URL única, lo que facilita su acceso y manipulación.
4. **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.
5. **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.

6. **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.
7. **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.

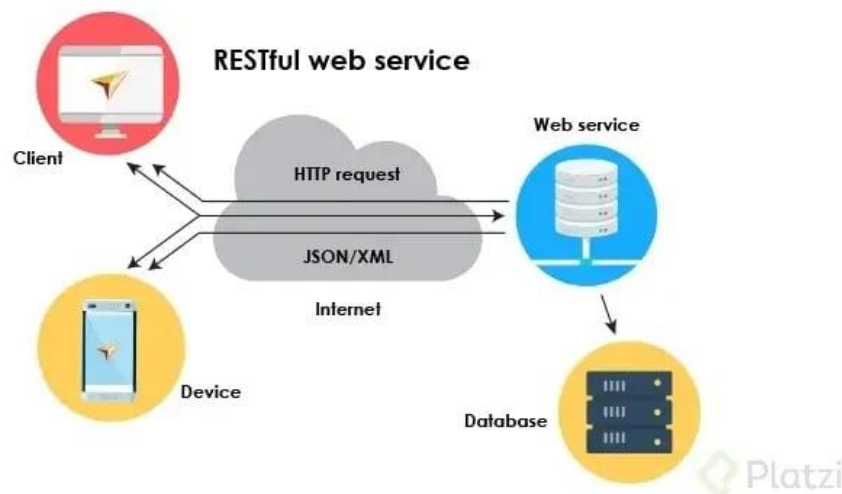


Figura 3. Estructura básica de un servicio REST.

Transición desde SOA hacia Microservicios

La arquitectura orientada a servicios (SOA) sentó las bases para el enfoque modular que posteriormente evolucionó en los microservicios. Ambos paradigmas comparten la idea de descomponer sistemas grandes en servicios reutilizables, pero se diferencian en su implementación y grado de acoplamiento.

Diferencias clave:

- **SOA** depende frecuentemente de un **Enterprise Service Bus (ESB)** para la orquestación de servicios, lo cual introduce un punto de fallo centralizado y un fuerte acoplamiento técnico.
- **Microservicios**, por el contrario, evitan la orquestación centralizada, favoreciendo la **coreografía distribuida** y el **despliegue independiente**.
- Mientras **SOA** suele usar **SOAP** como protocolo estándar de comunicación, los microservicios se inclinan por **REST**, **gRPC**, o mensajería basada en eventos.

La transición de SOA a microservicios se suele dar de forma progresiva, y consiste en:

1. **Identificación de servicios clave** en la arquitectura SOA.

2. **Desacoplamiento gradual del ESB**, usando API Gateways y patrones de mensajería.
3. **Migración de servicios a contenedores** (Docker) y orquestadores (Kubernetes).
4. **Refactorización para despliegue y pruebas independientes**.

Este proceso permite a las organizaciones modernizar sus sistemas sin reescribir todo el código, beneficiándose de las ventajas de los microservicios mientras mitigan los riesgos. En la siguiente figura observamos una estructura básica de cada arquitectura.

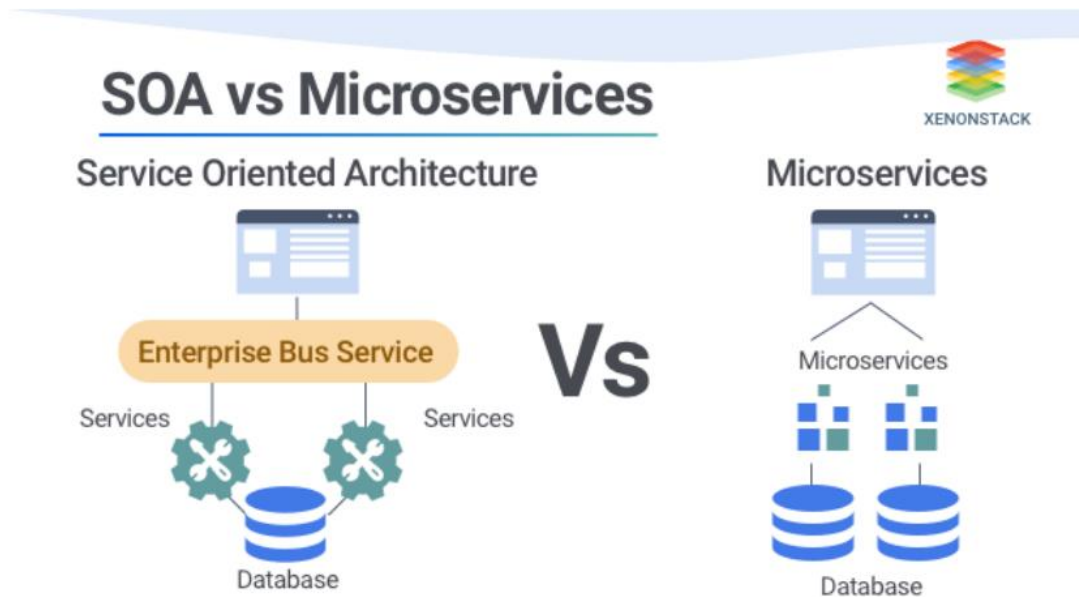


Figura 4. SOA vs Microservicios.

Tecnologías Utilizadas en el Desarrollo de Microservicios

Spring Boot

Spring Boot es un framework de desarrollo basado en Java que facilita la creación de aplicaciones web, especialmente servicios RESTful y microservicios. Forma parte del ecosistema Spring, un conjunto de herramientas y bibliotecas diseñadas para el desarrollo de aplicaciones empresariales en Java.

Spring Boot se creó para simplificar y agilizar la configuración de aplicaciones basadas en Spring, eliminando gran parte de la complejidad relacionada con la configuración manual y la gestión de dependencias. Permite crear aplicaciones con una estructura preconfigurada, minimizando la cantidad de código necesario para iniciar un proyecto.

En el contexto de servicios web REST, Spring Boot es ampliamente utilizado debido a sus características avanzadas de integración con el protocolo HTTP, su capacidad de manejar grandes volúmenes de tráfico y su compatibilidad con bases de datos, seguridad y otras tecnologías modernas.

Entre las razones por las cuales Spring Boot es ideal para desarrollar servicios web REST, destacan:

1. **Configuración automática (Auto Configuration):** Spring Boot detecta automáticamente los componentes necesarios y los configura sin intervención manual.

2. **Integración con Spring Web:** Facilita la creación de controladores REST y la gestión de rutas HTTP.
3. **Compatibilidad con JSON:** Usa la biblioteca **Jackson** para serializar y deserializar objetos Java en JSON, lo que facilita la comunicación entre cliente y servidor.
4. **Soporte para Microservicios:** Es la tecnología principal en arquitecturas basadas en microservicios, permitiendo la creación de sistemas escalables y desacoplados.
5. **Facilidad para implementar seguridad:** Se integra con **Spring Security** para autenticar y autorizar peticiones REST de manera sencilla.

Spring Cloud

Spring Cloud es una colección de herramientas que complementan Spring Boot para el desarrollo de sistemas distribuidos. Proporciona soluciones para los problemas comunes que enfrentan los microservicios, tales como descubrimiento de servicios, configuración centralizada, balanceo de carga, tolerancia a fallos, entre otros.

Funcionalidades destacadas:

- Configuración distribuida (Spring Cloud Config)
- Registro y descubrimiento de servicios (Eureka)
- Enrutamiento y gestión de tráfico (Spring Cloud Gateway)
- Balanceo de carga (con LoadBalancer o Ribbon)
- Circuit breakers (Hystrix, Resilience4j)
- Tracing y observabilidad (Sleuth + Zipkin)

Spring Cloud permite la creación de microservicios robustos que pueden escalar y comunicarse eficientemente en un entorno distribuido.

Eureka (Service Discovery)

Eureka es un servicio de descubrimiento desarrollado por Netflix e integrado con Spring Cloud. Su propósito es permitir que los microservicios se registren automáticamente y puedan ser descubiertos por otros servicios en tiempo de ejecución.

Características clave:

- Los servicios cliente se registran al arrancar.
- Permite localizar otros servicios dinámicamente sin necesidad de conocer direcciones IP o puertos.
- Facilita el balanceo de carga al distribuir las peticiones entre múltiples instancias.
- Ayuda a eliminar la necesidad de configuración estática de direcciones, facilitando el despliegue en entornos dinámicos como contenedores.

El uso de Eureka es fundamental en arquitecturas con alta dinámica y escalabilidad.

Spring Cloud Gateway (API Gateway)

Spring Cloud Gateway actúa como una puerta de entrada única para todas las peticiones que llegan a los microservicios. Es un **API Gateway** moderno basado en WebFlux que reemplaza herramientas como Zuul y proporciona una forma eficiente y reactiva de enrutar, filtrar, y gestionar las solicitudes entrantes.

Funcionalidades principales:

- **Ruteo dinámico:** Redirige peticiones a diferentes microservicios según su URI.
- **Filtros:** Añade filtros personalizados para logging, autenticación, validaciones, etc.
- **Balanceo de carga:** Integra con Eureka para enrutar hacia instancias disponibles.
- **Seguridad centralizada:** Implementación de políticas comunes como autenticación JWT o control de acceso.

El API Gateway representa un patrón esencial en microservicios al permitir control y centralización del tráfico.

REST (Representational State Transfer)

REST es un estilo arquitectónico que define un conjunto de restricciones para crear servicios web escalables y fácilmente mantenibles. En el contexto de microservicios, es ampliamente adoptado debido a su simplicidad, compatibilidad con HTTP y legibilidad.

Características en microservicios:

- Comunicación a través de métodos HTTP estándar: GET, POST, PUT, DELETE, etc.
- Intercambio de datos en formatos livianos como JSON.
- Orientado a recursos, cada entidad tiene una URL única.
- Stateless: No mantiene estado entre peticiones, lo que lo hace más escalable.

REST permite que los microservicios interactúen de manera eficiente, con APIs bien definidas y documentables.

Docker

Docker es una plataforma de contenedores que permite empaquetar aplicaciones y sus dependencias en una unidad portable y reproducible llamada contenedor. En arquitecturas de microservicios, Docker resulta crucial para el despliegue consistente y escalable.

Beneficios clave:

- **Portabilidad:** Un servicio empaquetado en Docker se puede ejecutar en cualquier entorno que soporte el motor de Docker.
- **Aislamiento:** Cada microservicio corre en su propio contenedor, aislado de los demás.
- **Escalabilidad:** Los contenedores pueden replicarse o eliminarse rápidamente según la carga.

- **Compatibilidad con orquestadores:** Como Kubernetes o Docker Swarm, que permiten una gestión avanzada de los servicios.

El uso de Docker en conjunto con Spring Boot y Spring Cloud permite desarrollar e implementar una arquitectura de microservicios completamente modular, desplegable y mantenible.

Estructura básica de un Microservicio con Spring Boot

Cada microservicio en esta práctica está diseñado como una aplicación Spring Boot independiente, siguiendo una estructura de paquetes estándar que permite organizar claramente la lógica de negocio, los controladores REST, los modelos de datos, los servicios y los repositorios. Esta organización facilita el mantenimiento, la escalabilidad y el entendimiento del código por parte de otros desarrolladores.

A continuación, se describe la estructura general utilizada en los microservicios implementados:

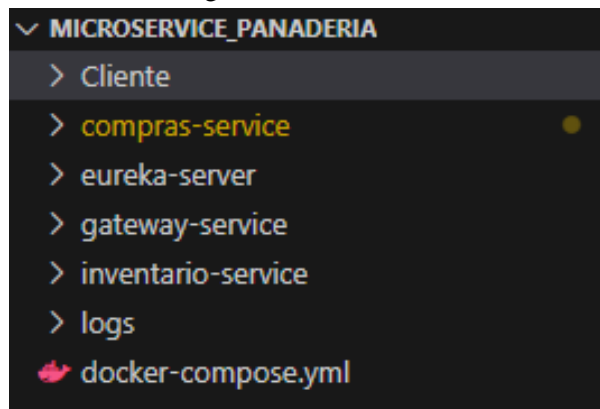


Figura 5. Estructura de mi practica para microservicios.

Cada carpeta es un microservicio, el cual tiene la estructura de una aplicación spring boot y se ejecutan de manera independiente. Spring Boot no solo permite desarrollar microservicios de manera rápida y eficiente, sino que, además, al integrarse fácilmente con herramientas de orquestación, descubrimiento de servicios, balanceo de carga y monitoreo, se adapta perfectamente a los requerimientos de una arquitectura de sistemas distribuidos.

El enfoque modular que promueve Spring Boot facilita la creación de microservicios escalables, independientes y reutilizables, optimizando el desarrollo y mantenimiento de aplicaciones distribuidas modernas.

Planteamiento del problema

En prácticas anteriores se desarrollaron aplicaciones distribuidas de forma monolítica o con una arquitectura cliente-servidor simple, lo cual limitaba la capacidad de escalar, mantener y extender los sistemas. A medida que los sistemas crecen en complejidad, se vuelve necesario adoptar arquitecturas que permitan una mayor modularidad, independencia entre componentes y facilidad de despliegue. En este contexto, surge la necesidad de diseñar una arquitectura basada en microservicios, donde cada módulo del sistema sea autónomo, especializado en una función específica y pueda comunicarse de manera eficiente con los demás servicios.

El reto principal de esta práctica consistió en construir una aplicación distribuida que simulara el funcionamiento de una panadería, dividiendo su lógica en múltiples servicios independientes:

inventario, compras, cliente, gateway y un servidor de descubrimiento (Eureka). Cada servicio debía ser capaz de ejecutarse por separado, contar con su propia configuración y, en algunos casos, su propia base de datos, lo que implicaba resolver problemas comunes de los sistemas distribuidos como:

- La comunicación entre microservicios mediante peticiones HTTP (REST) y el uso de RestTemplate.
- La autodescubribilidad de los servicios a través de Eureka, permitiendo que los servicios se registren dinámicamente y se comuniquen sin necesidad de conocer direcciones IP fijas.
- La exposición centralizada de los servicios a través de un API Gateway, el cual actúa como punto de entrada único para el sistema, manejando rutas, CORS y filtros personalizados.
- El manejo de interacción con el cliente final mediante un frontend desarrollado en el módulo Cliente, el cual consume los servicios del backend.
- La persistencia de datos en MySQL, especialmente en los servicios compras-service e inventario-service, gestionando operaciones CRUD de forma distribuida.

Este planteamiento implicó también el uso de tecnologías como Spring Boot, Docker para contenerización de los servicios, y Docker Compose para la orquestación del sistema completo, con el objetivo de levantar todos los servicios de manera integrada. Así, se abordaron los desafíos clave de los sistemas distribuidos: desacoplamiento, escalabilidad, descubrimiento de servicios y tolerancia a fallos.

Propuesta de solución

Para abordar el problema planteado, se propone el desarrollo de una arquitectura basada en **microservicios** utilizando **Spring Boot**, **Eureka**, **API Gateway**, **Docker** y **MySQL**. La solución simula el funcionamiento de una panadería digital distribuida, permitiendo gestionar productos, inventario, compras y la interacción con el cliente final de manera separada, pero coordinada.

El sistema estará compuesto por los siguientes componentes principales:

1. Microservicios con Spring Boot

Cada módulo del sistema será implementado como un microservicio independiente utilizando Spring Boot. En esta práctica se desarrollaron los siguientes servicios:

- **inventario-service**: Encargado de gestionar los productos disponibles en la panadería, su stock y la reposición automática.
- **compras-service**: Procesa las compras realizadas por los clientes, actualizando el inventario y registrando las transacciones.
- **Cliente**: Simula un cliente que consume los servicios ofrecidos por la panadería (Frontend).
- **gateway-service**: Actúa como punto de entrada único para el sistema, redirigiendo las solicitudes al microservicio correspondiente.
- **eureka-server**: Permite el descubrimiento de servicios, facilitando la comunicación dinámica entre ellos sin necesidad de direcciones fijas.

Cada microservicio tiene su propia lógica de negocio, configuración y puede escalar de manera independiente.

2. Comunicación entre servicios y API Gateway

La comunicación entre los microservicios se realiza mediante HTTP REST usando RestTemplate. El **API Gateway**, configurado con Spring Cloud Gateway, enruta las peticiones a los servicios internos de acuerdo con el path de la solicitud. Esto permite:

- Simplificar el acceso al sistema desde el cliente final.
- Aplicar configuraciones comunes como manejo de CORS, logging o autenticación (en casos futuros).
- Separar la lógica de enrutamiento de la lógica de negocio.

Ejemplos de algunas rutas expuestas:

- GET /inventario/productos: Lista todos los productos disponibles.
- POST /compras: Realiza una compra con una lista de productos.

3. Base de datos MySQL para persistencia de información

Los microservicios inventario-service y compras-service utilizan MySQL para gestionar de forma persistente la información crítica del sistema:

- El **inventario** mantiene el stock actualizado de cada producto.
- Las **compras** se registran en una tabla con la información del producto adquirido y la fecha de la transacción.

Cada servicio cuenta con su propia base de datos, permitiendo el **aislamiento de datos**, uno de los principios clave de la arquitectura de microservicios.

4. Automatización del inventario con tareas programadas

El servicio de inventario cuenta con una tarea automática implementada con @Scheduled en Spring Boot, que simula el horneado de pan. Esta tarea incrementa periódicamente el stock de cada producto (por ejemplo, cada 30 segundos), garantizando que la panadería tenga productos disponibles sin necesidad de intervención manual.

5. Descubrimiento dinámico con Eureka

Todos los microservicios (excepto el servidor Eureka) se registran automáticamente en el **Eureka Server**. Esto permite que los servicios se comuniquen entre sí utilizando únicamente el nombre lógico del servicio, sin importar la IP o el puerto en el que estén corriendo. Así se facilita el escalado y la resiliencia del sistema.

6. Contenerización y despliegue con Docker

Cada microservicio fue contenerizado utilizando Docker, permitiendo que puedan ejecutarse de forma aislada. Mediante **Docker Compose**, se definió un entorno de despliegue unificado que levanta todos los servicios junto con sus bases de datos respectivas. Esto asegura que el sistema completo pueda ejecutarse de forma local o en la nube de manera consistente y reproducible.

7. Monitoreo y registros del sistema

Cada microservicio genera logs que permiten monitorear las operaciones realizadas (consultas, compras, errores, actualizaciones de stock, etc.). Esto resulta útil para depurar el sistema, revisar su funcionamiento y detectar posibles fallos.

Con esta solución se construye un sistema distribuido funcional y modular que cumple con los principios de los microservicios. Se exploran aspectos fundamentales como el descubrimiento de servicios, la comunicación REST, la persistencia distribuida, la automatización de tareas, y el despliegue con contenedores. Todo ello permite simular una panadería que puede crecer, mantenerse y escalar fácilmente con base en la demanda del sistema.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollar la simulación del sistema distribuido basado en Microservicios, se utilizaron las siguientes herramientas y metodologías:

Materiales (Herramientas utilizadas)

1. **Lenguaje de programación: Java (JDK 21):** Se utilizó Java por su solidez y amplio soporte en el desarrollo de aplicaciones empresariales distribuidas. En particular, la versión JDK 21 proporciona mejoras de rendimiento, nuevas funcionalidades del lenguaje y compatibilidad con herramientas modernas del ecosistema Spring.
2. **Framework: Spring Boot:** Spring Boot fue la base para construir cada uno de los microservicios. Gracias a su enfoque de configuración mínima y su soporte nativo para REST, facilita el desarrollo de servicios independientes, escalables y mantenibles. Se aprovecharon módulos clave como Spring Web, Spring Data JPA y Spring Cloud.
3. **Spring Cloud y Eureka Server:** Se integró **Eureka Server** como solución de descubrimiento de servicios. Spring Cloud permitió que los microservicios se registraran automáticamente en el servidor Eureka, facilitando la comunicación entre servicios mediante nombres lógicos en lugar de direcciones IP o puertos fijos.
4. **Spring Cloud Gateway:** Se utilizó como **API Gateway** para centralizar las solicitudes entrantes desde los clientes y dirigirlos al microservicio correspondiente. Esta herramienta simplificó la gestión de rutas y la integración entre frontend y backend.
5. **Sistema de gestión de base de datos: MySQL:** Cada microservicio que requería persistencia (inventario) se conectó a una base de datos MySQL independiente. Esto permitió aislar datos por dominio, siguiendo buenas prácticas en arquitectura de microservicios. Se usó Spring Data JPA para interactuar con la bases de datos de forma eficiente y sencilla.
6. **Herramienta de construcción y dependencias: Maven:** Maven facilitó la gestión de bibliotecas necesarias en cada microservicio, así como la automatización del empaquetado y construcción del proyecto.
7. **Herramienta de contenedores: Docker:** Docker se utilizó para contenerizar cada microservicio y su bases de datos. Se definieron Dockerfile y un docker-compose.yml para desplegar el sistema completo de manera orquestada y replicable.

8. **Herramienta de pruebas: Postman:** Postman se utilizó para probar individualmente los endpoints de los microservicios a través del API Gateway. Permite realizar solicitudes HTTP y observar respuestas, facilitando la verificación de la lógica implementada.
9. **Entorno de desarrollo: Visual Studio Code (VSCode):** VSCode fue el editor de código empleado durante el desarrollo. Con extensiones para Java, Docker y Spring Boot, ofreció un entorno ágil para escribir, depurar y administrar múltiples proyectos simultáneamente.

Métodos Empleados

Para la implementación del sistema distribuido basado en microservicios, se emplearon los siguientes métodos y técnicas:

A continuación, se describen los principales métodos utilizados:

1. **Arquitectura basada en microservicios:** Se diseñó el sistema como un conjunto de microservicios independientes, cada uno con una única responsabilidad. Esto mejora la escalabilidad, el aislamiento de fallos y la capacidad de actualización individual de los servicios.
2. **Descubrimiento de servicios con Eureka:** Los microservicios se registran en el servidor Eureka al iniciar, y consultan otros servicios mediante su nombre lógico. Esto permite una comunicación flexible y desacoplada, eliminando la necesidad de conocer direcciones IP estáticas.
3. **Gestión de rutas con Spring Cloud Gateway:** Se definieron rutas personalizadas en el API Gateway para canalizar solicitudes hacia los microservicios internos. Esto permite a los clientes acceder al sistema a través de una única puerta de entrada (localhost:8083) mientras se mantiene la separación de responsabilidades en el backend.
4. **Persistencia de datos con MySQL y Spring Data JPA:** Los servicios inventario-service y compras-service utilizan JPA para mapear entidades a la tabla de la base de datos y realizar operaciones CRUD.
5. **Automatización del inventario con tareas programadas:** El inventario-service incluye una tarea automática (@Scheduled) que simula la reposición de productos en la panadería cada cierto tiempo. Este método asegura que el sistema mantenga stock disponible de forma regular.
6. **Control de concurrencia y consistencia de datos:** Para evitar errores en compras simultáneas, se implementaron operaciones transaccionales en compras-service. Antes de completar una compra, se verifica el stock disponible y se actualiza de manera atómica. En caso de falta de stock, se devuelve una respuesta adecuada al cliente.
7. **Contenerización y despliegue con Docker Compose:** Todo el sistema fue empaquetado en contenedores y gestionado con Docker Compose. Esto facilitó la ejecución simultánea de todos los servicios en un solo entorno, reduciendo la complejidad del despliegue y eliminando problemas de configuración local.
8. **Pruebas de endpoints con Postman:** Se probaron operaciones como la consulta de inventario, la simulación de compras y la verificación de reabastecimiento automático. Las

pruebas incluyeron condiciones normales, errores esperados (como intentos de compra sin stock) y pruebas concurrentes.

9. **Monitoreo básico con logs:** Cada microservicio genera logs usando SLF4J. Se registran eventos clave como el procesamiento de compras, actualizaciones de inventario y errores de comunicación. Esto permitió observar el comportamiento del sistema durante su ejecución.

Esta combinación de herramientas modernas y métodos bien estructurados me permitió construir un sistema distribuido modular y robusto, simulando la operación de una panadería desde una perspectiva orientada a microservicios. La integración de contenedores, descubrimiento de servicios y automatización de procesos nos refleja por lo visto, prácticas reales en sistemas de producción actuales.

Interfaz de Usuario (Cliente_Frontend)

El sistema incluye una interfaz de usuario implementada con tecnologías web básicas (HTML, CSS y JavaScript), que permite a los clientes interactuar con el sistema distribuido de forma amigable. Esta interfaz se conecta directamente al **API Gateway** para consultar productos disponibles, realizar compras y obtener actualizaciones del inventario en tiempo real.

1. HTML: Estructura de la Página

Se empleó HTML para definir la estructura de la aplicación web en el archivo index.html. La interfaz presenta una sección de bienvenida, un listado de productos obtenidos desde el microservicio de inventario, y un formulario de compra que permite al usuario ingresar su nombre, seleccionar productos y especificar cantidades. Cada elemento fue organizado para asegurar claridad y facilidad de uso.

2. CSS: Estilo y Diseño

El archivo style.css contiene los estilos aplicados a la interfaz. Se personalizaron botones, campos de entrada y se aplicó una disposición visual que mejora la estética general de la aplicación. Los estilos fueron definidos para mantener una experiencia visual limpia y coherente, facilitando la navegación del usuario.

3. JavaScript: Lógica de Interacción y Comunicación con el Backend

La lógica principal se encuentra en script.js, que controla la interacción entre el cliente y el backend distribuido. Se utiliza la **Fetch API** para realizar solicitudes HTTP a través del **API Gateway** expuesto en `http://localhost:8083`, lo que permite mantener una sola puerta de entrada al sistema completo.

- **obtenerProductos():** Se ejecuta al cargar la página, realizando una solicitud GET al gateway, el cual redirige la petición al microservicio de inventario. Los datos obtenidos (lista de productos y stock disponible) se muestran dinámicamente en la interfaz.
- **comprarProductos():** Esta función recopila el nombre del cliente y las cantidades seleccionadas por producto. Luego, se hace una solicitud POST a través del API Gateway, el cual reenvía la solicitud al microservicio de compras. La respuesta se muestra al usuario como confirmación o advertencia.
- **actualización dinámica del stock:** Tras una compra, se vuelve a ejecutar automáticamente obtenerProductos() para mantener la visualización del inventario actualizada.

4. Conexión con el Backend

Toda la comunicación se realiza a través del **API Gateway (Spring Cloud Gateway)**, cuya URL base es `http://localhost:8083`. Las solicitudes se enrutan de manera transparente a los microservicios correspondientes según la configuración definida en el gateway. Esto desacopla completamente al cliente de la ubicación real de cada servicio, permitiendo una mayor flexibilidad y escalabilidad del sistema.

Esta interfaz permite a los usuarios realizar compras de forma sencilla, conocer en tiempo real el estado del inventario y observar cambios inmediatos en la disponibilidad de productos. Gracias a su integración con el sistema de microservicios a través del gateway, el cliente interactúa con un backend robusto, distribuido y preparado para escalabilidad horizontal. Todos los códigos correspondientes a esta interfaz se presentan en el apartado de **Anexos**.

Desarrollo de la solución

Para la construcción del sistema distribuido basado en microservicios, se utilizó Spring Boot como base para cada uno de los servicios que componen el sistema (cliente, compras, inventario, gateway y Eureka Server). La creación de cada proyecto se realizó a través de [Spring Initializr](#), configurando los parámetros iniciales y seleccionando las dependencias necesarias de acuerdo al rol de cada microservicio.

Creación de proyectos en Spring Boot

Cada microservicio fue generado individualmente siguiendo la siguiente configuración básica:

- **Project:** Maven
- **Language:** Java
- **Spring Boot Version:** 3.3.4
- **Packaging:** Jar
- **Java Version:** 21
- **IDE:** Visual Studio Code

Dependiendo del microservicio, se añadieron diferentes dependencias:

- **Para todos los servicios (excepto el frontend):**
 - Spring Web
 - Spring Boot DevTools
 - Spring Boot Actuator
- **Para los microservicios que se comunican entre sí:**
 - Spring Cloud (para comunicación REST interna)
 - Eureka Client (para registro y descubrimiento de servicios)
- **Para el API Gateway:**

- Spring Cloud Gateway
- Eureka Discovery Client
- **Para el Eureka Server:**
 - Eureka Server
- **Para el Inventario y Compras:**
 - Spring Data JPA
 - MySQL Driver

Una vez descargado cada proyecto, fueron descomprimidos y abiertos en Visual Studio Code para su desarrollo.

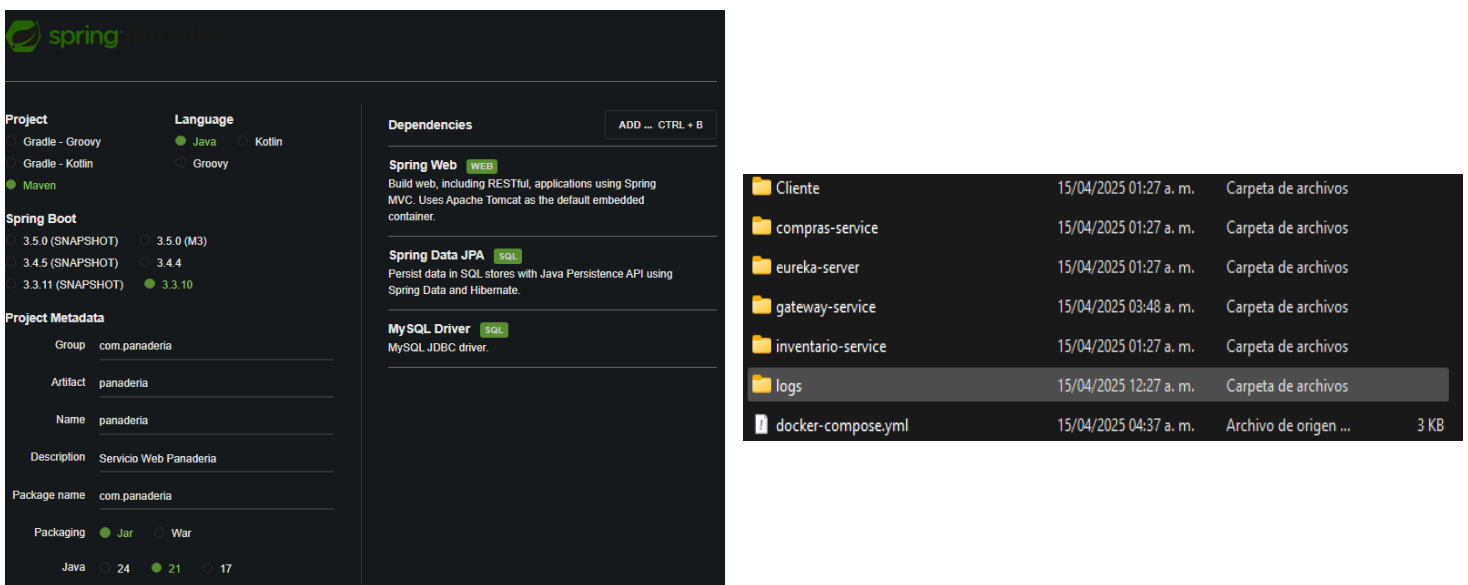


Figura 6. Estructura y creación de los proyectos con Spring Boot.

Configuración de la Base de Datos

La base de datos utilizada para los microservicios de compras e inventario fue **MySQL**, ejecutada dentro de un **contenedor Docker**. Para ello, se creó y levantó un contenedor con la imagen oficial de MySQL, y posteriormente se accedió a la terminal del contenedor para crear manualmente la base de datos correspondiente. Este enfoque permitió encapsular el entorno de la base de datos, facilitando su despliegue e integración con los servicios de backend. Y la estructura de esta, fue la siguiente.

Cabe destacar que para crear la tabla e insertar los datos iniciales, fue necesario ingresar manualmente al contenedor utilizando el siguiente comando:


```

PS C:\Users\user> docker exec -it mysql bash
bash-5.1# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 117
Server version: 8.4.5 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> |

```

Figura 7. Comandos utilizados para acceder al contenedor de mysql.

Dentro del entorno interactivo de MySQL, se creó la base de datos, las tablas necesarias y se insertaron los datos requeridos para las pruebas iniciales de los microservicios. Este procedimiento aseguró un entorno controlado y reproducible para el desarrollo y la ejecución del sistema distribuido.

```

Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 25
Server version: 8.4.5 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 TABLE inventario (
->     id INT PRIMARY KEY AUTO_INCREMENT,
->     producto VARCHAR(50) NOT NULL,
->     stock INT NOT NULL
-> );|

```

Figura 8. Código SQL para la creación de la tabla inventario.

Y posteriormente, respecto a la base de datos, antes de desarrollar la lógica del backend, fue necesario definir la estructura de datos. Para ello, se utilizó la misma base de datos que las practicas anteriores, sin embargo, ahora se añadieron más productos, utilizando las siguientes instrucciones SQL:

```

INSERT INTO inventario (producto, stock) VALUES ('Pan de trigo', 20);
INSERT INTO inventario (producto, stock) VALUES ('Pan de centeno', 15);
INSERT INTO inventario (producto, stock) VALUES ('Pan integral', 30);
INSERT INTO inventario (producto, stock) VALUES ('Pan de avena', 10);
INSERT INTO inventario (producto, stock) VALUES ('Pan de maíz', 25);
INSERT INTO inventario (producto, stock) VALUES ('Donas', 25);

```

Figura 9. Comandos utilizados para añadir más productos en la BD.

Y finalmente, en mi caso, fue necesario realizar la instalación de **Maven**, ya que es una herramienta fundamental para la gestión de dependencias y la compilación del proyecto en **Spring Boot**.

Instalación de Maven

Para instalar Maven, seguimos los siguientes pasos:

1. Descargar Maven

- Accedemos a la página oficial de Apache Maven: <https://maven.apache.org/download.cgi>.
- Descargamos la versión más reciente del archivo binario en formato ZIP.

2. Configurar Maven en el sistema

- Extraemos el contenido del archivo ZIP en una ubicación deseada (por ejemplo, C:\Users\user\Desktop\apache-maven-3.9.9-bin\apache-maven-3.9.9).
- Configuramos la variable de entorno MAVEN_HOME, apuntando a la ruta donde se extrajo Maven.

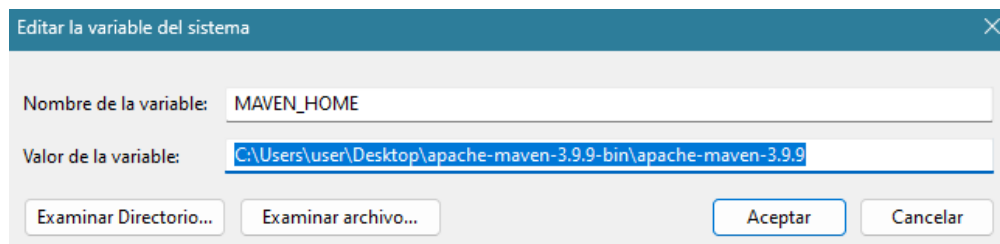


Figura 10. Configuración de la variable de entorno MAVEN_HOME.

- Agregamos la carpeta bin de Maven a la variable Path del sistema para poder ejecutar comandos desde la terminal.

```
%MAVEN_HOME%\bin
```

Figura 11. Añadimos la carpeta bin ahora a la variable Path del sistema.

3. Verificar la instalación

- Abrimos una terminal y ejecutamos el siguiente comando:

```

PS C:\Users\user> mvn -version
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
Maven home: C:\Users\user\Desktop\apache-maven-3.9.9-bin\apache-maven-3.9.9
Java version: 21.0.6, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-21
Default locale: es_MX, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"

```

Figura 12. Verificación de la instalación de Apache Maven.

Como se puede observar, la instalación fue exitosa y en la Figura anterior podemos encontrar la versión de Maven junto con la versión de Java detectada en el sistema.

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

A continuación, se detallan los componentes principales de la solución y su implementación:

Microservicio: inventario-service

Este microservicio tiene como objetivo la gestión del inventario de la panadería, permitiendo consultar productos y verificar su stock. Además, cuenta con un sistema de reabastecimiento automático mediante tareas programadas.

A continuación, se describen los principales componentes de su implementación:

1. PanaderiaControlador.java

Este archivo actúa como controlador REST. Es responsable de exponer los endpoints HTTP relacionados con la consulta y actualización del inventario.

- **@RestController:** Indica que esta clase responderá a solicitudes HTTP con datos, no vistas.
- **@RequestMapping("/inventario"):** Define la ruta base para este controlador.
- **@Autowired:** Inyecta automáticamente una instancia del servicio PanaderiaServicio.

Métodos principales del controlador:

- **GET /inventario/productos:** Devuelve la lista de todos los productos disponibles.

```

// Obtener lista de todos los productos disponibles
@GetMapping("/productos")
public List<Pan> obtenerProductos() {
    return panaderiaServicio.obtenerProductos();
}

```

Figura 13. Obtiene la lista de todos los productos disponibles.

- **GET /inventario/stock/{producto}:** Devuelve el stock actual de un producto específico, usando la variable de ruta {producto}.

```
// Ver el stock actual de un producto específico
@GetMapping("/stock/{producto}")
public int obtenerStock(@PathVariable String producto) {
    return panaderiaServicio.obtenerStock(producto);
}
```

Figura 14. Muestra el stock actual de un producto en específico.

2. PanaderiaServicio.java

Este componente implementa la lógica de negocio del microservicio. Interactúa directamente con el repositorio de datos para obtener, modificar o actualizar productos.

- **obtenerProductos():** Retorna todos los productos registrados.
- **obtenerStock(String producto):** Busca un producto por nombre y retorna su stock.
- **realizarCompra(String producto, int cantidad):** Verifica si hay suficiente stock y, si lo hay, lo descuenta.
- **abastecerStock(String producto, int cantidad):** Aumenta el stock del producto (usado por el horno automático).

```
@Service
public class PanaderiaServicio {

    @Autowired
    private PanRepositorio panRepositorio;

    // Obtener todos los productos
    public List<Pan> obtenerProductos() {
        return panRepositorio.findAll();
    }

    // Obtener el stock de un producto específico
    public int obtenerStock(String producto) {
        Pan pan = panRepositorio.findByProducto(producto);
        if (pan != null) {
            return pan.getStock();
        } else {
            return 0;
        }
    }

    // Realizar una compra y actualizar el stock
    public boolean realizarCompra(String producto, int cantidad) {
        Pan pan = panRepositorio.findByProducto(producto);
        if (pan != null && pan.getStock() >= cantidad) {
            pan.setStock(pan.getStock() - cantidad);
            panRepositorio.save(pan); // Actualizar en la base de datos
            return true;
        } else {
            return false; // No hay suficiente stock o el producto no existe
        }
    }

    // Abastecer el stock de un producto específico (sumando al stock actual)
    public void abastecerStock(String producto, int cantidad) {
        Pan pan = panRepositorio.findByProducto(producto);
        if (pan != null) {
            pan.setStock(pan.getStock() + cantidad); // Sumar la cantidad al stock actual
            panRepositorio.save(pan); // Guardar los cambios
        }
    }
}
```

Figura 15. Bloque de código de PanaderiaServicio.

3. HornoServicio.java

Este servicio simula un horno automático que abastece de forma periódica todos los productos disponibles. Utiliza tareas programadas para ejecutarse cada cierto tiempo.

- **@Scheduled(fixedRate = 100000):** Ejecuta el método `abastecerStockPanes()` cada 100 segundos.
- **@EnableScheduling:** Se debe agregar en la clase principal del microservicio (donde está la anotación `@SpringBootApplication`) para habilitar las tareas programadas.

```
public HornoServicio(PanaderiaServicio panaderiaServicio) {
    this.panaderiaServicio = panaderiaServicio;
}

// Se ejecuta cada hora (100000 ms = 100 segundos)
@Scheduled(fixedRate = 100000)
public void abastecerStockPanes() {
    List<String> productos = Arrays.asList(
        ...a:"Pan", "Pan de trigo", "Pan de centeno", "Pan integral",
        "Pan de avena", "Pan de maíz", "Donas"
    );

    for (String producto : productos) {
        panaderiaServicio.abastecerStock(producto, cantidad:20);
        int stockActual = panaderiaServicio.obtenerStock(producto);
        logger.info("Horno abastecido - Se añadieron {} unidades de '{}'. Stock actual: {}",
            50, producto, stockActual);
    }
}

@SpringBootApplication
@EnableScheduling
public class InventarioServiceApplication {

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

Figura 16. Función para reabastecer el stock disponible y anotación `@EnableScheduling`

4. Pan.java

Entidad JPA que representa la tabla inventario en la base de datos.

- **@Entity:** Marca la clase como una entidad persistente.
- **@Table(name = "inventario"):** Indica el nombre de la tabla correspondiente.
- Incluye campos como `id`, `producto` y `stock`.

```

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

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

    private String producto;
    private int stock;

    // Getters y Setters
    public Long getId() { return id; }
    public String getProducto() { return producto; }
    public int getStock() { return stock; }
    public void setStock(int stock) { this.stock = stock; }
}

```

Figura 17. Clase Pan.java que representa la tabla inventario,

5. PanRepositorio.java

Interfaz que extiende JpaRepository, lo cual permite acceder a operaciones CRUD automáticas sobre la entidad Pan.

- Incluye un método personalizado:

```

public interface PanRepositorio extends JpaRepository<Pan, Long> {
    Pan findByProducto(String producto);
}

```

Figura 18. Metodo para consultar el stock de un producto por su nombre.

6. application.properties

Este archivo contiene la configuración esencial del microservicio, la cual abarca los siguientes aspectos:

- **Puerto del servidor:** Define el puerto en el que se ejecuta el microservicio dentro del contenedor o entorno local.
- **Conexión a la base de datos MySQL en Docker:** Se especifican las credenciales, la URL del host y el nombre de la base de datos que reside en un contenedor Docker, permitiendo una integración directa y encapsulada.
- **Configuración de JPA e Hibernate:** Se incluyen parámetros como el dialecto de MySQL, la estrategia de actualización del esquema (ddl-auto) y opciones de log para facilitar la depuración durante el desarrollo.

- **Integración con Eureka (modo Docker):** Permite el registro automático del microservicio dentro del servidor Eureka, favoreciendo el descubrimiento dinámico de servicios dentro del entorno distribuido.
- **Logs:** Se configuran los niveles de logging para monitorear adecuadamente el comportamiento del microservicio y facilitar la trazabilidad de errores.

Este microservicio constituye una parte esencial del sistema distribuido, ya que se encarga de mantener la consistencia del inventario ante solicitudes provenientes de distintos servicios. Su diseño modular y desacoplado no solo facilita la escalabilidad y el mantenimiento, sino que, gracias a su integración con Eureka y la ejecución en contenedores Docker, permite una orquestación eficiente dentro de una arquitectura basada en microservicios.

```
spring.application.name=inventario-service
server.port=8081

#spring.datasource.url=jdbc:mysql://localhost:3306/panaderia //Descomentar para ejecutarlo de manera local
# Lo uso para usarlo con docker-compose
spring.datasource.url=jdbc:mysql://mysql:3306/panaderia
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

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/inventario-service.log
logging.level.root=INFO
logging.level.com.panaderia=DEBUG

# eureka.client.service-url.defaultZone=http://localhost:8761/eureka //Descomentar para ejecutarlo de manera local
# Lo uso para usarlo con docker-compose
eureka.client.service-url.defaultZone=http://eureka-service:8761/eureka
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.instance.prefer-ip-address=true
```

Figura 19. application.properties de inventario-service

Como se puede observar en la configuración del archivo application.properties, cada microservicio debe especificar la URL del servidor Eureka para poder registrarse correctamente. Esto se logra mediante la propiedad eureka.client.service-url.defaultZone, la cual, en un entorno de Docker, debe apuntar al nombre del servicio declarado en docker-compose (por ejemplo, http://eureka-service:8761/eureka). Asimismo, se deben habilitar los parámetros eureka.client.register-with-eureka=true y eureka.client.fetch-registry=true, que permiten el registro del microservicio en el servidor Eureka y la obtención del registro de otros servicios disponibles, respectivamente. Esta configuración es crucial para habilitar el descubrimiento dinámico de servicios dentro de una arquitectura basada en microservicios.

Microservicio: compra-service

El microservicio compra-service se encarga de procesar solicitudes de compra de productos por parte de los clientes. Este microservicio actúa como cliente del microservicio inventario-service, al que consulta y actualiza el stock disponible de productos. Implementa balanceo de carga usando RestTemplate con anotaciones de Spring Cloud para permitir la comunicación con múltiples instancias del servicio de inventario si están registradas en Eureka.

A continuación se describe su arquitectura e implementación:

1. CompraController.java

Este controlador expone un endpoint para recibir solicitudes de compra:

- **@RestController:** Define la clase como un controlador REST.
- **@RequestMapping("/compras"):** Ruta base del microservicio.
- **@PostMapping:** Permite recibir solicitudes HTTP POST con los datos de la compra en el cuerpo.

```
@RestController
@RequestMapping("/compras")
public class CompraController {

    private final CompraService compraService;

    @Autowired    Unnecessary `@Autowired` annotation
    public CompraController(CompraService compraService) {
        this.compraService = compraService;
    }

    @PostMapping
    public String realizarCompra(@RequestBody CompraRequest compraRequest) {
        return compraService.procesarCompra(compraRequest);
    }
}
```

Figura 20. Clase para recibir solicitudes de compra.

Este método delega la lógica a CompraService, quien se encarga del procesamiento.

2. CompraService.java

Este componente representa el núcleo lógico del microservicio de compras, encargándose de orquestar el proceso de adquisición de productos. La clase CompraService emplea un RestTemplate para comunicarse con el microservicio de inventario y un LoadBalancerClient para seleccionar dinámicamente instancias disponibles de inventario-service registradas en Eureka, implementando así un balanceo de carga manual.

El flujo de ejecución inicia con la recepción de una solicitud de compra, que puede involucrar múltiples productos. Para cada uno, se consulta el stock disponible mediante una petición HTTP GET. Si la cantidad deseada está disponible, se procede a ejecutar una operación PUT sobre el inventario para realizar la compra. En caso contrario, se registra un mensaje de advertencia. Durante todo el

proceso, se generan logs detallados que reportan tanto el estado previo como el estado resultante del stock, además de indicar el nombre del cliente y los productos adquiridos.

Entre las características destacadas de esta clase se encuentran:

- **Balanceo de carga manual:** mediante `LoadBalancerClient`, se selecciona dinámicamente una instancia del microservicio de inventario.
- **Comunicación interservicios:** se realizan llamadas REST para consultar y modificar el inventario.
- **Gestión de errores y validación:** se valida que el stock sea suficiente antes de realizar la compra.
- **Registro exhaustivo:** el uso de SLF4J permite generar logs informativos y de advertencia según el resultado de cada transacción.

Este diseño promueve la comunicación desacoplada entre microservicios, permitiendo que la lógica de negocio se mantenga distribuida y escalable en un entorno de arquitectura orientada a servicios.

```
public class CompraService {

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

    private final RestTemplate restTemplate;
    private final LoadBalancerClient loadBalancerClient; // Inyectar loadBalancerClient

    private final String INVENTARIO_URL = "http://inventario-service/inventario";

    public CompraService(RestTemplate restTemplate, LoadBalancerClient loadBalancerClient) {
        this.restTemplate = restTemplate;
        this.loadBalancerClient = loadBalancerClient;
    }

    public String procesarCompra(CompraRequest compraRequest) {
        StringBuilder resultado = new StringBuilder("Resultados de compra:\n");

        for (Compra compra : compraRequest.getCompras()) {
            String producto = compra.getProducto();
            int cantidadDeseada = compra.getCantidad();

            // log de selección del servidor por Ribbon
            ServiceInstance instance = loadBalancerClient.choose(serviceId:"inventario-service");
            if (instance != null) {
                logger.info("Load balancer for 'inventario-service' chose server: {}", instance.getHost(), instance.getPort());
            } else {
                logger.error("No available instances for 'inventario-service' found!");
            }

            // Obtener stock actual desde el inventario-service
            Integer stockActual = restTemplate.getForObject(
                INVENTARIO_URL + "/stock/" + producto,
                Integer.class
            );

            if (stockActual != null && stockActual >= cantidadDeseada) {
                // Realizar la compra
                restTemplate.put(
                    INVENTARIO_URL + "/comprar/" + producto + "/" + cantidadDeseada,
                    request:null
                );

                // Consultar el nuevo stock después de la compra
                Integer stockRestante = restTemplate.getForObject(
                    INVENTARIO_URL + "/stock/" + producto,
                    Integer.class
                );

                logger.info("Cliente '{}' compró {} unidades de '{}'. Stock previo: {}, stock restante: {}",
                    compraRequest.getNombreCliente(), cantidadDeseada, producto, stockActual, stockRestante);

                resultado.append("Compra de ").append(cantidadDeseada)
                    .append(" ").append(producto).append(" realizada con éxito. ")
                    .append("Stock restante: ").append(stockRestante).append("\n");
            } else {
                logger.warn("Cliente '{}' intentó comprar {} unidades de '{}', pero el stock era de {}.",
                    compraRequest.getNombreCliente(), cantidadDeseada, producto,
                    stockActual != null ? stockActual : "desconocido");

                resultado.append("Stock insuficiente para ").append(producto).append("\n");
            }
        }
    }
}
```

Figura 21. Bloque de código de `CompraService`.

3. Modelos de Datos

- **Compra.java:** Representa una compra individual de un producto con dos campos: producto y cantidad.

Y en el contexto de la panadería, la clase Compra.java tiene un papel fundamental en el manejo de las transacciones de los clientes. Esta clase actúa como un modelo que encapsula la información necesaria para representar una compra específica, como el producto que el cliente desea adquirir y la cantidad de ese producto.

A continuación, se presenta el código de la clase Compra.java:

```
package com.panaderia.model;

public class Compra {
    private String producto;
    private int cantidad;

    // Constructor vacío
    public Compra() {}

    // Getters y Setters
    public String getProducto() {
        return producto;
    }

    public void setProducto(String producto) {
        this.producto = producto;
    }

    public int getCantidad() {
        return cantidad;
    }

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

Figura 22. Código de la clase Compra.java.

- **Atributos:** La clase tiene dos atributos:
 - **producto:** Un String que representa el nombre del producto que el cliente desea comprar.
 - **cantidad:** Un int que indica la cantidad del producto que se está comprando.
- **Constructor:** Tiene un constructor vacío, lo que permite que Spring lo utilice para instanciar objetos automáticamente cuando se reciba una solicitud, por ejemplo, en el cuerpo de una solicitud HTTP (como en el caso de un @RequestBody en un controlador).
- **Getters y Setters:** Son métodos utilizados para obtener y establecer los valores de los atributos producto y cantidad. Esto sigue el principio de encapsulamiento, permitiendo que el acceso a los atributos se controle a través de estos métodos.

La clase `Compra` es crucial para el proceso de la compra en la panadería, ya que permite encapsular los datos sobre qué producto se está comprando y cuántas unidades de ese producto se desean adquirir.

- **CompraRequest.java:** Agrupa una lista de compras realizadas por un cliente, junto con su nombre.

La clase `CompraRequest.java` juega un papel crucial en la interacción entre el cliente y el backend, ya que encapsula los detalles de una compra realizada por un cliente. Esta clase se utiliza para recibir los datos en el cuerpo de una solicitud HTTP (por ejemplo, en una solicitud POST) que contiene la información del cliente y de los productos que desea comprar.

A continuación, se presenta el código de la clase `CompraRequest.java`:

```
package com.panaderia.model;

import java.util.List;

public class CompraRequest {
    private String nombreCliente;
    private List<Compra> compras;

    // Getters y Setters
    public String getNombreCliente() {
        return nombreCliente;
    }

    public void setNombreCliente(String nombreCliente) {
        this.nombreCliente = nombreCliente;
    }

    public List<Compra> getCompras() {
        return compras;
    }

    public void setCompras(List<Compra> compras) {
        this.compras = compras;
    }
}
```

Figura 23. Código de la clase `CompraRequest.java`.

- **Atributos:**
 - **nombreCliente:** Un `String` que representa el nombre del cliente que está realizando la compra.
 - **compras:** Una lista de objetos `Compra`, que contiene los productos y cantidades que el cliente desea comprar.
- **Getters y Setters:**
 - **getNombreCliente() y setNombreCliente():** Métodos para acceder y modificar el nombre del cliente.

- **getCompras() y setCompras():** Métodos para acceder y modificar la lista de compras, que contiene los productos y las cantidades que el cliente desea adquirir.

La clase `CompraRequest.java` es utilizada para recibir los datos completos de la compra en una solicitud HTTP, permitiendo que el backend procese la compra correctamente y actualice el inventario en función de las solicitudes del cliente.

Estos modelos son usados como estructura del JSON que se recibe en la petición POST del controlador.

4. `RestTemplateConfig.java`

Esta clase define la configuración del cliente HTTP utilizado por el microservicio de compras para comunicarse con otros servicios REST, en particular con el microservicio de inventario. Se trata de una clase anotada con `@Configuration`, lo que indica que forma parte del contexto de configuración de Spring.

El método `restTemplate()` expone un bean de tipo `RestTemplate` con dos características clave:

- **@LoadBalanced:** Esta anotación permite que el `RestTemplate` utilice nombres lógicos de los servicios registrados en Eureka (como `inventario-service`) en lugar de URLs físicas o direcciones IP. Gracias a ello, las llamadas REST pueden beneficiarse del balanceo de carga y la detección dinámica de instancias de servicio.
- **Interceptor personalizado:** Se aplica un interceptor (`RestTemplateInterceptor`) que permite agregar funcionalidades como el logueo de peticiones salientes, trazabilidad de llamadas o manipulación de cabeceras. Esto es útil para monitoreo, depuración o integración con herramientas de observabilidad.

En conjunto, esta configuración fortalece la capacidad del microservicio para operar de forma eficiente dentro de una arquitectura distribuida, asegurando que las llamadas HTTP sean resilientes, trazables y orientadas a servicios dinámicos registrados en Eureka.

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.setInterceptors(List.of(restTemplateInterceptor));
    return restTemplate;
}
```

Figura 24. Configuración del `RestTemplate`.

5. `RestTemplateInterceptor.java`

Este componente personalizado implementa la interfaz `ClientHttpRequestInterceptor` y forma parte de la configuración avanzada del cliente HTTP (`RestTemplate`) en el microservicio de compras. Su propósito es interceptar las solicitudes HTTP salientes realizadas mediante `RestTemplate` para añadir una cabecera personalizada denominada **X-Correlation-ID**, utilizada comúnmente para fines de **trazabilidad** en sistemas distribuidos.

Características clave:

- **Propagación de contexto:** El interceptor busca en la petición HTTP entrante el encabezado X-Correlation-ID (si existe), lo cual permite mantener un identificador único entre múltiples servicios en una misma transacción o flujo de llamadas. Esta técnica es fundamental para realizar trazabilidad en arquitecturas de microservicios.
- **Integración con Spring Context:** Utiliza RequestContextHolder para acceder al contexto de la petición actual y extraer el identificador de correlación desde las cabeceras de la solicitud original.
- **Transparencia:** Si el encabezado no está presente, el interceptor no lo agrega, asegurando que la lógica sea no intrusiva y mantenga el comportamiento predeterminado.

6. application.properties

Este archivo contiene la configuración necesaria para la correcta ejecución del microservicio:

La configuración del archivo application.properties del microservicio de compras sigue una estructura muy similar a la del microservicio de inventario, ya que ambos forman parte de una misma arquitectura distribuida y comparten dependencias comunes, como la base de datos MySQL, Eureka para el descubrimiento de servicios y JPA para el manejo de persistencia.

Elementos destacados:

- **Nombre del servicio y puerto:** Se define el identificador lógico del microservicio como compra-service, y se expone en el puerto 8082, lo cual evita colisiones con otros servicios como inventario-service.
- **Conexión a base de datos (MySQL en Docker):** Utiliza la misma base de datos panaderia, alojada en un contenedor Docker, con credenciales estándar. Esta configuración garantiza la centralización de la información de compras e inventario.
- **Persistencia con JPA y Hibernate:** Se activa la actualización automática del esquema de base de datos (ddl-auto=update) y se especifica el dialecto de MySQL 8, además de mostrar las consultas SQL en consola.
- **Logs y trazabilidad:** Se configura la salida de logs a un archivo específico del servicio (logs/compra-service.log) y se habilitan niveles de log detallados para los componentes relacionados con el balanceo de carga y el descubrimiento de servicios. Esto es especialmente útil para depurar la interacción entre compra-service y inventario-service.
- **Integración con Eureka (modo Docker):** Al igual que en el microservicio de inventario, se especifica la URL del servidor Eureka en su versión Dockerizada (eureka-service:8761/eureka), y se habilita el registro y la obtención del registro de servicios, permitiendo que este microservicio pueda descubrir dinámicamente a otros dentro del ecosistema.

Esta configuración homogénea asegura que ambos microservicios puedan comunicarse fluidamente en un entorno orquestado, al tiempo que se mantienen aislados y fácilmente escalables.

```

spring.application.name=compra-service
server.port=8082

#spring.datasource.url=jdbc:mysql://localhost:3306/panaderia //Descomentar para ejecutarlo de manera local
# Lo uso para usarlo con docker-compose
spring.datasource.url=jdbc:mysql://mysql:3306/panaderia
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

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/compra-service.log
logging.level.root=INFO
logging.level.com.panaderia=DEBUG

# eureka.client.service-url.defaultZone=http://localhost:8761/eureka
# Lo uso para usarlo con docker-compose
eureka.client.service-url.defaultZone=http://eureka-service:8761/eureka
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.instance.prefer-ip-address=true

# Activa logging detallado para el load balancer
logging.level.org.springframework.cloud.loadbalancer=DEBUG
logging.level.org.springframework.cloud.client=DEBUG
# Activa logging detallado para el cliente Eureka
logging.level.com.netflix.client=DEBUG
logging.level.com.netflix.discovery=DEBUG
logging.level.org.springframework.cloud.netflix.ribbon=DEBUG

```

Figura 25. application.properties de compra-service.

Este microservicio cumple un rol fundamental como orquestador de compras dentro del sistema distribuido. Su capacidad de comunicarse de forma dinámica con el microservicio de inventario y tomar decisiones basadas en el estado actual del stock le otorgan flexibilidad y robustez. Además, gracias al uso de Eureka y balanceo de carga, se garantiza una alta disponibilidad del sistema en un entorno de microservicios.

Microservicio: gateway-service

El microservicio gateway-service actúa como punto de entrada único (API Gateway) al sistema distribuido de la panadería. Su función principal es enrutar las solicitudes HTTP hacia los microservicios correspondientes (inventario-service y compra-service), basándose en rutas definidas. Además, implementa filtros personalizados que añaden funcionalidad de trazabilidad y logging, mejorando la observabilidad del sistema.

1. Configuración de rutas en application.properties

Las rutas se definen de forma dinámica utilizando Spring Cloud Gateway y el balanceador de carga basado en Eureka (lb://):

```
# Rutas dinámicas usando el nombre registrado en Eureka
spring.cloud.gateway.routes[0].id=inventario-service
spring.cloud.gateway.routes[0].uri=lb://inventario-service
spring.cloud.gateway.routes[0].predicates[0]=Path=/inventario/**

spring.cloud.gateway.routes[1].id=compra-service
spring.cloud.gateway.routes[1].uri=lb://compra-service
spring.cloud.gateway.routes[1].predicates[0]=Path=/compras/**
```

Figura 26. Rutas dinámicas hacia los microservicios de inventario y compras.

Esto permite enrutar peticiones que inician con `/inventario` o `/compras` hacia los microservicios correspondientes, sin necesidad de conocer sus direcciones físicas.

2. Registro con Eureka

El API Gateway se registra como cliente en **Eureka**:

```
# Registro en Eureka
# eureka.client.service-url.defaultZone=http://localhost:8761/eureka //Descomentar para ejecutarlo de manera local
# Lo uso para usarlo con docker-compose
eureka.client.service-url.defaultZone=http://eureka-service:8761/eureka
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.instance.prefer-ip-address=true
```

Figura 27. Registro del API Gateway en Eureka.

Esto permite que el gateway descubra dinámicamente a los servicios a los que debe enrutar las peticiones.

3. Filtro Global de Correlación (CorrelationFilter.java)

Este filtro genera un ID de correlación único (UUID) para cada solicitud que entra al sistema, y lo agrega como cabecera `X-Correlation-ID`. Esto es útil para trazabilidad de solicitudes a través de los distintos microservicios.

```
@Component
public class CorrelationFilter implements GlobalFilter {

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

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        String correlationId = UUID.randomUUID().toString(); // Generar un ID único

        // Agregar encabezado a la solicitud
        exchange.getRequest().mutate()
            .header(headerName:"X-Correlation-ID", correlationId)
            .build();

        // Log del ID de correlación
        logger.info("Encabezado X-Correlation-ID agregado: {}", correlationId);

        return chain.filter(exchange);
    }
}
```

Figura 28. Filtro Global de Correlación.

- El ID generado puede ser utilizado por otros microservicios (si se propaga) para registrar eventos relacionados con la misma solicitud.
- Se hace un log informativo con el valor del ID generado.

Esta estrategia de correlación es fundamental para entornos de microservicios, ya que permite analizar el flujo completo de una solicitud, desde su entrada al gateway hasta su procesamiento por los distintos servicios, incluso en escenarios de múltiples instancias o balanceo de carga.

4. Filtro Global de Logging de Respuesta (ResponseLoggingFilter.java)

Este filtro intercepta todas las respuestas que pasan por el gateway y loggea el cuerpo completo de cada respuesta:

```
// Log personalizado del cuerpo de la respuesta
log.info("Respuesta de {} {} -> {}", request.getMethod(), request.getURI(), responseBody);
```

Figura 29. Log personalizado de respuesta.

Esto es especialmente útil para debugging y monitoreo, ya que permite ver exactamente qué datos regresan los servicios sin modificar su lógica interna.

- Usa ServerHttpResponseDecorator para interceptar y leer el cuerpo de la respuesta de forma reactiva (usando Flux y Mono).
- Tiene prioridad alta en la cadena de filtros (orden = -2).

5. Configuración de CORS (CorsGlobalConfiguration.java)

Este archivo define la configuración global de CORS (Cross-Origin Resource Sharing) para el gateway-service. Su propósito es permitir que aplicaciones web externas, como el frontend que se ejecuta en `http://localhost:8084`, puedan realizar peticiones al gateway sin ser bloqueadas por las políticas de seguridad del navegador.

Características clave:

- **Origen permitido:** Se especifica explícitamente el origen `http://localhost:8084` como permitido. Esto es útil cuando se está desarrollando el frontend por separado del backend.
- **Métodos y cabeceras permitidas:** Se habilita cualquier tipo de cabecera (*) y método HTTP (*), permitiendo una interacción completa desde el cliente con los distintos endpoints del backend.
- **Credenciales habilitadas:** La opción `setAllowCredentials(true)` permite que las cookies u otros datos de autenticación se incluyan en las peticiones, si es necesario.
- **Aplicación global:** La configuración se aplica a todas las rutas (/**) del gateway, gracias a `UrlBasedCorsConfigurationSource`.

Esta configuración asegura que aplicaciones web externas puedan interactuar sin bloqueos del navegador debido a políticas de seguridad.


```

@Configuration
public class CorsGlobalConfiguration {

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

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

        return new CorsWebFilter(source);
    }
}

```

Figura 30. Clase de configuración global de CORS.

Y cabe decir que, gracias a esta configuración global de CORS implementada en el gateway-service, **ya no es necesario anotar manualmente cada controlador o endpoint de los microservicios con @CrossOrigin**. El gateway se encarga de gestionar las políticas de acceso entre dominios, permitiendo así una interacción fluida entre el frontend y los microservicios backend (como inventario-service y compra-service) sin necesidad de configuraciones adicionales en cada servicio.

Esto no solo simplifica el código de los controladores, sino que centraliza el control de acceso, facilitando el mantenimiento y la escalabilidad del sistema distribuido.

6. Configuración de Logging (logback.xml)

Incluye una configuración específica para mostrar logs detallados en la consola:

- Se establece un formato de log personalizado.
- Se activa el nivel DEBUG específicamente para el paquete del API Gateway:

```

<configuration>

    <!-- Appender que escribe los logs a la consola -->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!-- Definimos el formato del log -->
            <pattern>%d{yyyy-MM-dd HH:mm:ss} - [%thread] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <!-- Root logger que enviará los logs a la consola -->
    <root level="INFO">
        <appender-ref ref="console" />
    </root>

    <!-- Logger específico para el API Gateway -->
    <logger name="org.springframework.cloud.gateway" level="DEBUG"/>

</configuration>

```

Figura 31. Archivo de configuración para logs más detallados.

Esto facilita el rastreo de los eventos dentro del Gateway durante el desarrollo.

El gateway-service cumple un papel esencial al centralizar el acceso a los microservicios del sistema distribuido. Al integrar **balanceo de carga**, **descubrimiento de servicios**, **trazabilidad de solicitudes** y **logging detallado**, este componente no solo simplifica la arquitectura externa, sino que también mejora considerablemente la capacidad de monitoreo y mantenimiento del sistema completo.

Microservicio: eureka-server

El eureka-server cumple la función de **registro y descubrimiento de servicios** dentro de la arquitectura de microservicios. Este componente es esencial para que los demás microservicios (como compra-service, inventario-service, y gateway-service) puedan localizarse entre sí de forma dinámica, sin necesidad de conocer sus direcciones IP o puertos específicos.

1. Configuración del servidor Eureka (application.properties)

Se asigna el nombre eureka-server a la aplicación y se configura para que escuche en el puerto 8761, el cual es el estándar para servidores Eureka.

Además, se especifica que este componente **no debe registrarse a sí mismo** ni intentar descubrir otros servicios:

```
spring.application.name=eureka-server
server.port=8761

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

Figura 32. Configuración del servidor Eureka.

Cabe decir que como se mencionó, anteriormente, es necesario implementar para los microservicios que se quieren registrar Eureka Client y para el servidor Eureka Server, y para activar en eureka-server dicha funcionalidad, el proyecto debe incluir la dependencia de **Spring Cloud Netflix Eureka Server** y usar la anotación `@EnableEurekaServer` en la clase principal:

```
@SpringBootApplication
@EnableEurekaServer

public class EurekaServerApplication {

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

Figura 33. Anotación `@EnableEurekaServer` en la clase principal.

2. Interfaz Web de Eureka

Al ejecutar el servidor de Eureka, ya sea de forma individual o como parte del entorno completo de contenedores, este expone una interfaz web accesible desde <http://localhost:8761>. A través de esta interfaz, es posible:

- Visualizar los microservicios actualmente registrados y disponibles en el sistema.
- Monitorear en tiempo real el estado del sistema de descubrimiento.
- Verificar el momento en que un microservicio se registra o se da de baja.

Esta consola web es una herramienta esencial para el diagnóstico y monitoreo de la arquitectura distribuida. Además, la integración del eureka-server en el sistema elimina la necesidad de configurar rutas fijas entre servicios, ya que permite una comunicación dinámica y basada en descubrimiento, lo que a su vez favorece la escalabilidad, tolerancia a fallos y mantenimiento del sistema distribuido.

spring Eureka

HOME

LAST 1000 SINCE STARTUP

System Status

Environment

test

Current time

2025-04-16T07:46:49 +0000

Data center

default

Uptime

00:00

Lease expiration enabled

false

Renews threshold

1

Renews (last min)

0

DS Replicas

localhost

Instances currently registered with Eureka

Application

AMIs

Availability Zones

Status

No instances available

General Info

Name

Value

total-avail-memory

96mb

num-of-cpus

12

current-memory-usage

54mb (56%)

server-up-time

00:00

registered-replicas

http://localhost:8761/eureka/

Figura 34. Interfaz Web de Eureka para el diagnóstico y monitoreo de los microservicios.

Cabe decir que, en el apartado de **Instances currently registered with Eureka**, se podrán observar los microservicios registrados, como **inventario-service**, **compra-service**, **gateway-service**, entre otros, una vez que se inicien. Esto permite confirmar que cada microservicio se ha registrado correctamente y está disponible para la comunicación en el sistema distribuido.

Microservicio: cliente-service (Frontend)

El cliente-service actúa como la interfaz de usuario del sistema distribuido. Está desarrollado como una aplicación Spring Boot que sirve una interfaz web estática, compuesta por archivos HTML, CSS y JavaScript.

1. Arquitectura del cliente

- Utiliza Spring Boot para alojar los archivos estáticos y exponerlos desde la raíz (\).
- Contiene un controlador básico (**FrontendController**) que direcciona todas las solicitudes raíz hacia index.html.

```

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 "index.html" en la carpeta static/
    }
}

```

Figura 35. Bloque de código que direcciona todas las solicitudes hacia index.html.

2. Estructura de archivos estáticos

- **index.html:** Página principal con campos para seleccionar productos y realizar compras.

El archivo index.html es el punto de entrada principal para la interfaz de usuario del frontend de la panadería. En este archivo, se define la estructura básica del documento HTML, y se enlazan los estilos y scripts que gestionarán la apariencia y el comportamiento dinámico de la página.

Estructura del HTML

Estructura básica de la página HTML: La estructura comienza con las etiquetas estándar de HTML5, donde se definen el lenguaje (lang="es") y el conjunto de caracteres (UTF-8), así como la configuración para la visualización en dispositivos móviles mediante la etiqueta meta viewport.

```

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>¡Tu Panadería Favorita!</title>
    <link rel="stylesheet" href="style.css">
</head>

```

Figura 36. Etiquetas básicas de HTML.

Cuerpo del documento: Dentro de la etiqueta <body>, se define la estructura visual de la página, que incluye un encabezado con el título y una pequeña descripción, seguido de una sección principal donde se mostrarán los productos disponibles y los campos para realizar la compra.

En la sección <div id="productos"> se cargarán los productos disponibles en la panadería, lo que se hace dinámicamente utilizando JavaScript (sección que será gestionada por el archivo script.js). De igual forma se incluye un campo de texto donde el cliente podrá ingresar su nombre y un botón que ejecutará la función comprarPan(), la cual se definirá en el archivo JavaScript para gestionar la compra.

Posteriormente, después de realizar alguna compra, se mostrará un mensaje con el resultado de la operación (si la compra fue exitosa o si ocurrió algún problema). Finalmente, el archivo style.css se

se enlaza dentro de la etiqueta <head> para aplicar los estilos a los elementos HTML. Lo anterior lo podemos observar en la siguiente Figura:

```
<body>
  <div class="container">
    <header>
      <h1>¡Bienvenido a La Panadería del Barrio!</h1>
      <p>¡El mejor pan recién horneado, directo a tu hogar!</p>
    </header>

    <div class="content">
      <h2>Selecciona tus panes favoritos:</h2>
      <div id="productos">
        <!-- Los productos serán cargados aquí mediante JavaScript -->
      </div>

      <h3>Datos de Compra:</h3>
      <input type="text" id="nombrecliente" placeholder="Ingresa tu nombre" class="input-field">

      <button onclick="comprarPan()" class="btn">Realizar Compra</button>

      <p id="mensaje"></p>
    </div>
  </div>

  <script src="script.js"></script>
</body>
</html>
```

Figura 37. Cuerpo del documento HTML.

En general, este archivo HTML ofrece una interfaz sencilla donde los usuarios pueden ver los productos disponibles, ingresar su nombre y realizar compras a través de una interacción con JavaScript. Los productos se cargarán dinámicamente, y las compras se gestionarán en el backend, mostrando los resultados al usuario.

- **style.css:** Hoja de estilo para dar formato y diseño visual a la página.

El archivo style.css se encarga del diseño y la apariencia visual de la interfaz de la panadería. Su propósito es mejorar la experiencia del usuario al proporcionar un diseño limpio, moderno y fácil de navegar.

Principales características del diseño:

- **Diseño responsivo:** La página se adapta a diferentes tamaños de pantalla gracias a propiedades como max-width y box-sizing.
- **Colores y tipografía:** Se utiliza una paleta de colores suaves y fuentes legibles para mejorar la estética y la accesibilidad.
- **Organización visual:** Se definen estilos para el encabezado, los productos, los formularios y los botones, asegurando una presentación ordenada y agradable.
- **Interactividad:** Los botones cambian de color cuando el usuario pasa el cursor sobre ellos, lo que mejora la experiencia de navegación.

En general, este archivo mejora la presentación del contenido sin afectar la funcionalidad del sistema.

- **script.js:** Este archivo JavaScript contiene la lógica principal de interacción entre el frontend y los microservicios del sistema, a través del API Gateway. Su propósito es facilitar la

comunicación con los servicios de inventario y de compra, así como reflejar los cambios de manera dinámica en la interfaz de usuario. Las principales funciones que implementa son:

- **Consulta de productos disponibles:** Al cargar la página, se realiza una petición HTTP GET al endpoint `/inventario/productos` expuesto por el `inventario-service` (accesible mediante el Gateway en `http://localhost:8083`). Esto permite obtener el listado de productos disponibles junto con su stock actual. La respuesta se procesa para mostrarse de forma dinámica en la interfaz HTML.
- **Gestión de compras:** El usuario puede seleccionar cantidades de productos y registrar su nombre. Al confirmar la compra, se construye un objeto JSON que representa la transacción y se envía mediante una petición POST al endpoint `/compras`, el cual redirige al `compra-service`. El microservicio correspondiente se encarga de validar y procesar la compra.
- **Actualización del stock posterior a la compra:** Una vez realizada una compra, se vuelve a invocar la función `obtenerProductos()` para refrescar los datos de stock en la interfaz. Esto asegura que el usuario siempre vea información actualizada respecto a los productos disponibles.

A continuación, se explican en detalle las funciones contenidas en este archivo:

Definición de la URL de la API

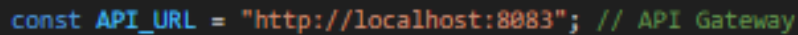
A screenshot of a code editor showing a single line of JavaScript code: `const API_URL = "http://localhost:8083"; // API Gateway`. The text is white on a dark background.

Figura 38. URL del API Gateway.

Esta constante almacena la URL base de la API REST del backend, en este caso, el **API Gateway**. Se utiliza en las solicitudes para obtener productos desde el `inventario-service` y para enviar las compras al `compra-service`.

Función `obtenerProductos()`

Esta función obtiene la lista de productos disponibles en la panadería y su respectivo stock actual.

- `fetch(`${API_URL}/inventario/productos)` → Realiza una solicitud HTTP **GET** al endpoint del inventario.
- Si la respuesta es exitosa (`response.ok`) → Convierte la respuesta a JSON y llama a la función `mostrarProductos()` para renderizarlos.
- Si ocurre un error o el servidor no responde → Muestra un mensaje de error al usuario en pantalla.

Esta función también se invoca automáticamente al cargar la página, asegurando que el usuario siempre vea los productos disponibles desde el inicio.

```
// Obtener el stock disponible y la lista de productos
async function obtenerProductos() {
  try {
    const response = await fetch(`${API_URL}/inventario/productos`); // Solicitar al inventario-service
    if (response.ok) {
      const productos = await response.json();
      mostrarProductos(productos);
    } else {
      document.getElementById("stock").innerText = "Error al obtener productos";
    }
  } catch (error) {
    document.getElementById("stock").innerText = "No se pudo conectar al servidor";
  }
}
```

Figura 39. Función para obtener el stock y lista de productos disponibles.

Función mostrarProductos(productos)

Esta función despliega los productos obtenidos del backend en la interfaz de usuario.

1. Se accede al contenedor con id #productos.
2. Se limpia el contenido previo para evitar duplicación de datos.
3. Se recorre el arreglo de productos recibido.
4. Por cada producto, se genera dinámicamente un bloque HTML que incluye:
 - Nombre del producto.
 - Stock disponible.
 - Un campo numérico donde el usuario puede indicar cuántas unidades desea comprar.
5. Finalmente, cada producto es añadido al DOM.

Esta implementación asegura que los productos se presenten de manera clara, con controles intuitivos para seleccionar cantidades.

```
// Mostrar los productos en el frontend
function mostrarProductos(productos) {
  const productosDiv = document.getElementById("productos");
  productosDiv.innerHTML = ''; // Limpiar el contenedor de productos antes de agregar nuevos

  productos.forEach((producto) => {
    const productoDiv = document.createElement("div");
    productoDiv.classList.add("producto");

    productoDiv.innerHTML = `
      <label for="cantidad-${producto.id}">${producto.producto} - Stock: ${producto.stock}</label>
      <input type="number" id="cantidad-${producto.id}" min="0" max="${producto.stock}" value="0" />
    `;

    productosDiv.appendChild(productoDiv);
  });
}
```

Figura 40. Función para mostrar los productos en el frontend.

Función comprarPan()

Esta función permite al usuario realizar una compra al sistema:

1. **Validación del nombre del cliente:**
 - Si el campo está vacío, se muestra un mensaje solicitando al usuario que lo complete.
2. **Recopilación de productos seleccionados:**
 - Se recorren los productos listados en la interfaz.
 - Se verifica qué productos tienen una cantidad mayor a cero.
 - Se almacenan en un arreglo compras, cada uno con su nombre y cantidad.
3. **Validación de productos seleccionados:**
 - Si no se ha elegido ningún producto, se muestra una advertencia en pantalla.
4. **Construcción del objeto JSON:**
 - Se arma un objeto data con el nombre del cliente y el arreglo de compras.
5. **Envío de la solicitud al backend:**
 - Se realiza una petición HTTP **POST** al endpoint /compras.
 - Si la compra es exitosa, se muestra el mensaje de respuesta del servidor.
 - Si ocurre un error, se notifica al usuario.
 - Finalmente, se actualiza el stock en la interfaz llamando de nuevo a obtenerProductos().

Comunicación eficiente con la API

Gracias al uso de fetch y la sintaxis async/await, el archivo script.js permite manejar las respuestas del servidor de manera asíncrona, 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.


```

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

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

  const compras = [];

  // Recopilar los productos y sus cantidades seleccionadas
  const productos = document.querySelectorAll("#productos div");
  productos.forEach((productoDiv) => {
    const productoId = productoDiv.querySelector("input").id.split("-")[1];
    const cantidad = document.getElementById(`cantidad-${productoId}`).value;

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

  if (compras.length === 0) {
    document.getElementById("mensaje").innerText = "Por favor, selecciona al menos un producto.";
    return;
  }

  // Preparar el JSON para el envío
  const data = {
    nombreCliente: nombre,
    compras: compras,
  };

  try {
    const response = await fetch(`${API_URL}/compras`, { // Solicitar al compra-service
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(data), // Enviar el JSON correctamente formateado
    });

    if (response.ok) {
      const mensaje = await response.text();
      document.getElementById("mensaje").innerText = mensaje;
    } else {
      document.getElementById("mensaje").innerText = "Error al realizar la compra";
    }
  } catch (error) {
    document.getElementById("mensaje").innerText = "No se pudo conectar al servidor";
  }

  // Actualizar el stock después de la compra
  obtenerProductos();
}

// Inicializar con los productos disponibles
obtenerProductos();

```

Figura 41. Función para comprar pan y comunicación con la API.

NOTA:

Es importante destacar que la estructura generada por Spring Boot incluye varios archivos adicionales que son fundamentales para el funcionamiento de la aplicación, como los archivos de configuración de **Spring Security**, los archivos de pruebas (**tests**), y otros componentes internos del framework. Estos archivos son creados automáticamente por Spring Boot para asegurar el correcto funcionamiento y manejo de la aplicación.

Durante el desarrollo del presente proyecto, se presentaron algunos inconvenientes al momento de instalar las dependencias y generar el archivo .jar del proyecto. En particular, surgieron errores relacionados con los archivos de prueba generados automáticamente (ApplicationTests), los cuales impedían la correcta compilación. Para poder continuar con el proceso de empaquetado e instalación de dependencias, fue necesario **deshabilitar temporalmente estos archivos de prueba**. Esta acción

permitió completar exitosamente el build del proyecto y continuar con la ejecución de los microservicios.

Sin embargo, en este documento no se profundiza en la explicación de estos archivos adicionales, ya que su función es principalmente facilitar el entorno de desarrollo, las pruebas y la configuración predeterminada de los proyectos. El enfoque de esta explicación se ha centrado en los archivos y clases que tienen un impacto directo en la funcionalidad principal del microservicio de panadería.

Construcción y despliegue con Docker

Para facilitar el despliegue y la gestión de los servicios que componen este sistema distribuido, se utilizó **Docker** para contenerizar cada microservicio, y **Docker Compose** para orquestar su ejecución conjunta.

Creación de imágenes Docker

Cada microservicio del sistema (incluyendo inventario-service, compras-service, gateway-service, cliente y eureka-server) cuenta con su propio archivo **Dockerfile**, el cual define las instrucciones necesarias para construir su respectiva imagen de Docker.

Es importante señalar que **cada Dockerfile debe ubicarse en la raíz del directorio de su correspondiente microservicio**, es decir, en la carpeta principal de cada uno (por ejemplo, ./inventario-service/Dockerfile, ./compras-service/Dockerfile, etc.). Esta estructura permite que las rutas relativas utilizadas dentro del Dockerfile funcionen correctamente y que la construcción de la imagen se realice con base en el contenido de esa carpeta específica.

Esta organización facilita la construcción independiente de imágenes para cada componente del sistema, favoreciendo la modularidad, reutilización y despliegue consistente de los servicios en un entorno contenerizado.

Dockerfile base para microservicios Spring Boot

Para los microservicios que corren en Spring Boot (inventario, compras, gateway, eureka-server), se empleó la siguiente estructura:

```
# Usar una imagen base de OpenJDK
FROM openjdk:21-jdk-slim

# Establecer el directorio de trabajo dentro del contenedor
WORKDIR /app

# Copiar el archivo JAR compilado al contenedor
COPY target/<nombre-del-servicio>-*.*.jar app.jar

# Exponer el puerto del servicio
EXPOSE <puerto-del-servicio>

# Ejecutar la aplicación Spring Boot
ENTRYPOINT ["java", "-jar", "app.jar"]
```

El archivo .jar correspondiente se genera a través del comando mvn clean install dentro del proyecto antes de construir la imagen, y se encuentra en la carpeta target/. Para ello al ejecutar dicho comando, la compilación no debe tener errores, se debe ver de la siguiente manera:

```
[INFO] Building jar: c:\Users\user\Desktop\Microservice_Panaderia\gateway-service\target\gateway-service-0.0.1-SNAPSHOT.jar
[INFO] --- spring-boot:3.4.4:repackage (repackage) @ gateway-service ---
[INFO] Replacing main artifact c:\Users\user\Desktop\Microservice_Panaderia\gateway-service\target\gateway-service-0.0.1-SNAPSHOT.jar with repackaged archive, ad
[INFO] The original artifact has been renamed to c:\Users\user\Desktop\Microservice_Panaderia\gateway-service\target\gateway-service-0.0.1-SNAPSHOT.jar.original
[INFO] --- install:3.1.4:install (default-install) @ gateway-service ---
[INFO] Installing c:\Users\user\Desktop\Microservice_Panaderia\gateway-service\pom.xml to C:\Users\user\.m2\repository\gateway-service\gateway-service\0.0.1-SNAP
[INFO] Installing c:\Users\user\Desktop\Microservice_Panaderia\gateway-service\target\gateway-service-0.0.1-SNAPSHOT.jar to C:\Users\user\.m2\repository\gateway-
vice-0.0.1-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 27.620 s
[INFO] Finished at: 2025-04-16T02:24:15-06:00
[INFO] -----
PS C:\Users\user\Desktop\Microservice_Panaderia>
```

Figura 42. Ejecución de mvn clean install y compilación sin errores.

Al hacer lo anterior, en la carpeta “**Target**”, encontraremos el .jar, como se ve a continuación.

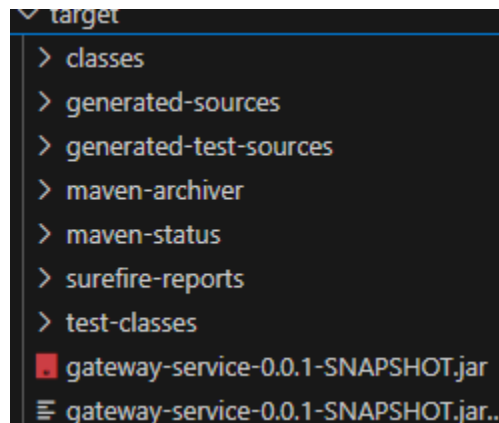


Figura 43. Archivo .jar obtenido.

Dockerfile para el frontend (cliente-service)

Para servir la interfaz web del sistema de panadería, se utiliza un contenedor basado en Apache. Esta solución permite desplegar el frontend como una aplicación estática de forma simple y eficiente.

A continuación, se presenta el contenido del Dockerfile correspondiente al frontend:

```
# Usar la imagen oficial de Apache
FROM httpd:alpine

# Copiar archivos estáticos al directorio raíz de Apache
COPY src/main/resources/static/ /usr/local/apache2/htdocs/

# Exponer el puerto por el que Apache sirve el contenido
EXPOSE 8084

# Cambiar el puerto por defecto de Apache (80) al 8084
```

```
RUN sed -i 's/Listen 80/Listen 8084/'  
/usr/local/apache2/conf/httpd.conf
```

Es importante mencionar que este Dockerfile debe ubicarse en la raíz de la carpeta del frontend (cliente-service) para que Docker pueda construir la imagen correctamente, accediendo al contenido estático de la aplicación.

Orquestación con Docker Compose

Para ejecutar todos los servicios del sistema de panadería de forma coordinada y eficiente, se utilizó el archivo docker-compose.yml. Este archivo permite definir y gestionar múltiples contenedores que conforman el sistema, incluyendo sus puertos, variables de entorno, volúmenes, redes y relaciones de dependencia entre ellos.

Servicios definidos:

- **mysql:** Contenedor que ejecuta la base de datos MySQL 8, utilizada por los microservicios.
- **eureka-service:** Servidor de descubrimiento que permite registrar y localizar los microservicios de manera dinámica.
- **inventario-service y compras-service:** Microservicios encargados de la gestión del stock y el procesamiento de compras respectivamente.
- **api-gateway:** Servicio que funciona como puerta de enlace, centralizando las peticiones del cliente y redirigiéndolas a los microservicios correspondientes.
- **frontend:** Aplicación web del cliente, servida mediante un contenedor Apache.

Fragmento clave del docker-compose.yml:

```
services:  
  
  mysql:  
    image: mysql:8  
    container_name: mysql  
    restart: always  
    environment:  
      MYSQL_ROOT_PASSWORD: root  
      MYSQL_DATABASE: panaderia  
    ports:  
      - "3307:3306"  
    volumes:  
      - mysql_data:/var/lib/mysql  
    networks:  
      - panaderia-net  
    healthcheck:  
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]  
      interval: 10s
```

```
    timeout: 5s
    retries: 5
    start_period: 10s

eureka-service:
  image: eureka-server-image
  build: ./eureka-server
  ports:
    - "8761:8761"
  networks:
    - panaderia-net
  depends_on:
    mysql:
      condition: service_healthy

inventario-service:
  image: inventario-service-image
  build: ./inventario-service
  ports:
    - "8081:8081"
  depends_on:
    eureka-service:
      condition: service_started
    mysql:
      condition: service_healthy
  environment:
    SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/panaderia
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: root

  networks:
    - panaderia-net

compras-service:
  image: compras-service-image
  build: ./compras-service
  ports:
    - "8082:8082"
  depends_on:
    eureka-service:
      condition: service_started
    mysql:
      condition: service_healthy
  environment:
    SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/panaderia
```

```

    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: root
networks:
  - panaderia-net

api-gateway:
  image: api-gateway-image
  build: ./gateway-service
  ports:
    - "8083:8083"
  depends_on:
    inventario-service:
      condition: service_started
    compras-service:
      condition: service_started
    eureka-service:
      condition: service_started
  networks:
    - panaderia-net

frontend:
  image: frontend-image
  build: ./Cliente
  ports:
    - "8084:8084"
  depends_on:
    api-gateway:
      condition: service_started
  networks:
    - panaderia-net

volumes:
  mysql_data:

networks:
  panaderia-net:
    name: panaderia-net
    driver: bridge

```

Redes y persistencia

- Se definió una red personalizada (panaderia-net) para permitir la comunicación entre contenedores.

- Se utilizó un volumen Docker (mysql_data) para persistir los datos de MySQL incluso después de reiniciar el contenedor.

Cabe mencionar que, el archivo docker-compose.yml debe ubicarse en la raíz del proyecto principal, es decir, en el mismo nivel donde se encuentran todas las carpetas de los servicios (eureka-server, inventario-service, compras-service, gateway-service, Cliente, etc.).

Esto es fundamental para que Docker Compose pueda acceder correctamente a cada directorio y construir las imágenes de manera adecuada a través de las rutas relativas definidas en cada servicio.

Con esta estructura, bastará con ejecutar docker-compose up --build desde la raíz del proyecto para iniciar todos los servicios del sistema de panadería de forma automática y coordinada; pero básicamente con ello, ya tendremos todo listo y podremos generar nuestro contenedor con todos los servicios de la panadería, sin estar ejecutando servicio por servicio.

Instrucciones para ejecutar el proyecto

Para ejecutar este sistema distribuido de manera completa utilizando contenedores, es necesario contar con **Docker** y **Docker Compose** instalados en el equipo. A continuación, se describen los pasos necesarios para compilar, construir e iniciar todos los servicios.

1. Compilación de los proyectos

Antes de construir los contenedores Docker, es necesario compilar cada microservicio para generar los archivos .jar necesarios. Para ello, abrimos una terminal y nos ubicamos dentro de la carpeta principal del proyecto. En nuestro caso, la ruta es algo similar a:

```
PS C:\Users\user\Desktop\Microservice_Panaderia>
```

Una vez ubicados, ejecutamos el siguiente comando en cada carpeta de microservicio (eureka-server, inventario-service, compras-service, gateway-service):

mvn clean install

Este comando se encarga de limpiar compilaciones anteriores, descargar dependencias y generar el archivo .jar dentro de la carpeta target correspondiente. Este archivo será utilizado en el proceso de construcción de las imágenes Docker.

NOTA: Este apartado solo lo menciono en caso de que se quiera realizar de cero la construcción del contenedor con cada servicio. Sin embargo, para esta práctica, este proceso ya está hecho, así que podemos pasar al siguiente paso.

2. Construcción y despliegue de los contenedores

Una vez generados los .jar, podemos levantar todo el sistema usando Docker Compose. Desde la carpeta raíz del proyecto, ejecutamos el siguiente comando:

```
PS C:\Users\user\Desktop\Microservice_Panaderia> docker compose up --build
[+] Building 0.4s (1/2)
=> [eureka-service internal] load build definition from Dockerfile
=> => transferring dockerfile: 483B
=> [eureka-service internal] load metadata for docker.io/library/openjdk:21-jdk-slim
```

Este comando construirá todas las imágenes de los servicios definidos en el archivo docker-compose.yml, incluyendo la base de datos, servidor Eureka, microservicios, gateway y el frontend.

3. Verificación del despliegue

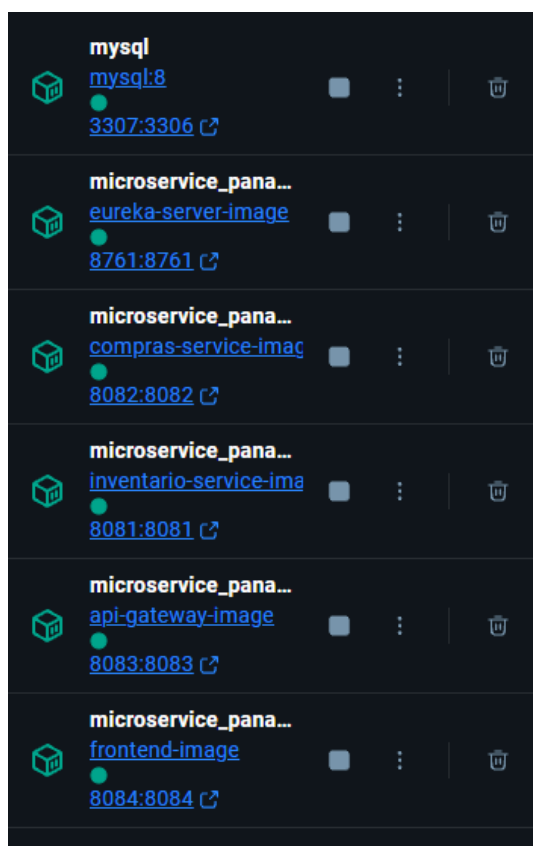
Si todos los servicios se construyen y levantan correctamente, el sistema estará distribuido y operativo. Los puertos asignados por cada servicio son los siguientes:

- **Frontend (Cliente):** <http://localhost:8084>
- **API Gateway:** <http://localhost:8083>
- **Eureka Server:** <http://localhost:8761>
- **Microservicios** (accesibles a través del gateway):
 - Inventario: /inventario/productos
 - Compras: /compras

Aunque de igual forma en Docker Desktop, podemos encontrar lo siguiente:



El contenedor en ejecución, y dentro de este, todos los servicios corriendo, especificados en el archivo docker-compose.yml.



4. Interacción con el sistema

Para probar el sistema, simplemente abrimos un navegador web y accedemos al frontend: <http://localhost:8084>

Desde esta interfaz gráfica, el usuario puede:

- Visualizar los productos disponibles y su stock.
- Ingresar su nombre como cliente.
- Seleccionar la cantidad de productos deseados.
- Realizar la compra con un solo clic.

Una vez realizada la compra, se actualiza automáticamente el stock disponible y se muestra un mensaje confirmando la operación.

También se puede utilizar herramientas como **Postman** para realizar pruebas directas a los endpoints expuestos por el API Gateway, lo cual es útil para verificar el comportamiento de los servicios de manera individual, sin embargo, eso ya lo veremos más a detalle en el apartado de **Resultados**.

The screenshot shows a web browser at localhost:8084 displaying the '¡BIENVENIDO A LA PANADERÍA DEL BARRIO!' application. The header includes the title and a tagline '¡El mejor pan recién horneado, directo a tu hogar!'. The main content area is titled 'Selecciona tus panes favoritos:' and lists eight items with their respective stock levels and a quantity selector (set to 0):

Producto	Stock	Cantidad
Pan	250	0
Pan de trigo	258	0
Pan de centeno	255	0
Pan integral	270	0
Pan de avena	250	0
Pan de maíz	265	0
Donas	262	0
Conchas	18	0

Below the list, there is a 'Datos de Compra:' section with a text input field labeled 'Ingresa tu nombre' and a blue 'Realizar Compra' button.

Resultados

El sistema distribuido de panadería implementado mediante microservicios ha mostrado un funcionamiento correcto, estable y eficiente. La arquitectura basada en Spring Boot, Eureka, MySQL, Docker y un cliente web permitió verificar que los servicios se comunican de forma efectiva y que el sistema es capaz de escalar modularmente. A través del API Gateway, las peticiones del cliente se enrutan correctamente a los microservicios correspondientes, manteniendo una capa de abstracción y seguridad entre el frontend y los servicios internos.

Durante las pruebas realizadas, se constató que cada microservicio responde adecuadamente según su responsabilidad. El servicio de inventario permitió consultar productos y su disponibilidad, mientras que el servicio de compras procesó adecuadamente las solicitudes de compra, actualizando el stock y registrando los eventos.

El servidor Eureka cumplió su función como descubridor de servicios, facilitando la comunicación entre los distintos módulos de manera automática. Se pudo comprobar que, al iniciar los contenedores, todos los servicios se registraban correctamente en el panel de Eureka (accesible desde <http://localhost:8761>), mostrando un entorno funcional distribuido y coordinado.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
COMPRA-SERVICE	n/a (1)	(1)	UP (1) - a7dda862a7b0:compra-service:8082
GATEWAY-SERVICE	n/a (1)	(1)	UP (1) - 04206c50ab53:gateway-service:8083
INVENTARIO-SERVICE	n/a (1)	(1)	UP (1) - 9c8a0c18e0af:inventario-service:8081

Figura 43. Servicios registrados en el panel de Eureka.

Pruebas con Postman

Se realizaron diversas pruebas utilizando **Postman** para verificar el comportamiento de la API REST a través del API Gateway (<http://localhost:8083>). Entre las principales pruebas destacan:

- **GET /inventario/productos:** Se utilizó para obtener la lista de productos disponibles. La respuesta fue un arreglo JSON con los id, nombres y cantidades disponibles de los productos registrados. Esta información coincidía con los datos almacenados en la base de datos MySQL, demostrando que la conexión entre el microservicio y la base de datos es exitosa.

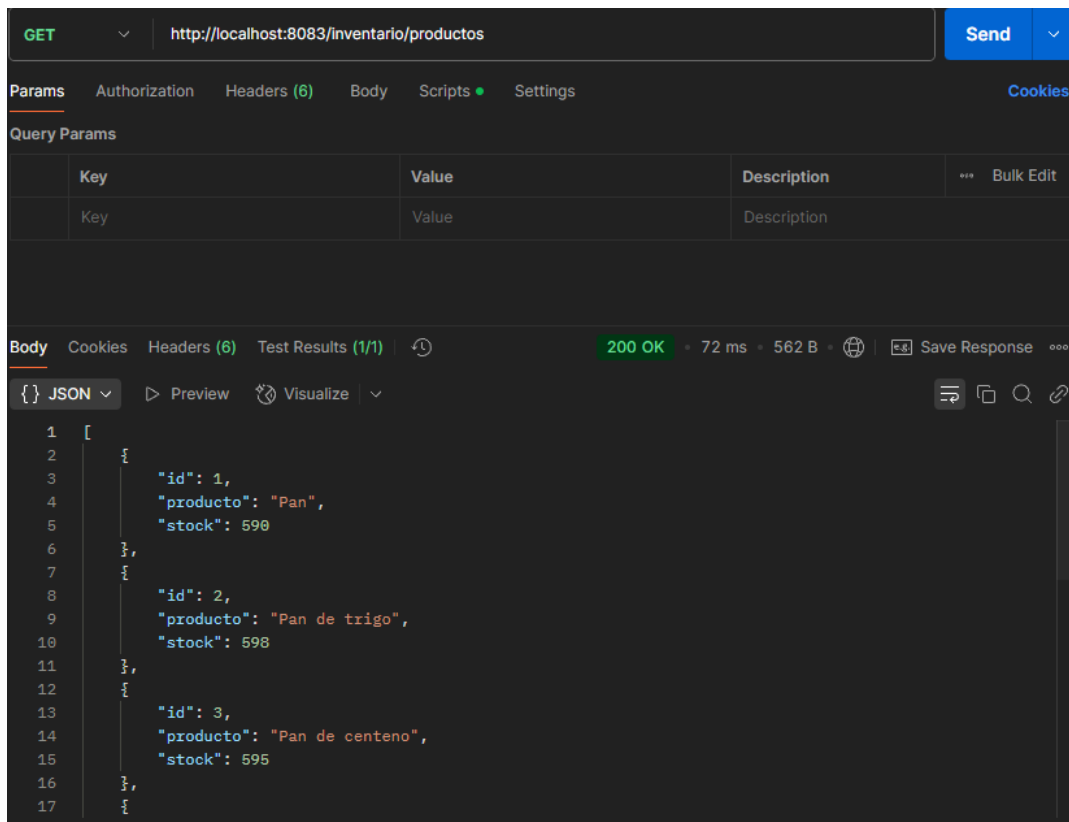


Figura 44. Prueba con Postman para obtener el listado de productos.

- **GET /stock/{producto}:** Se enviaron solicitudes, para obtener el stock de un producto específico.

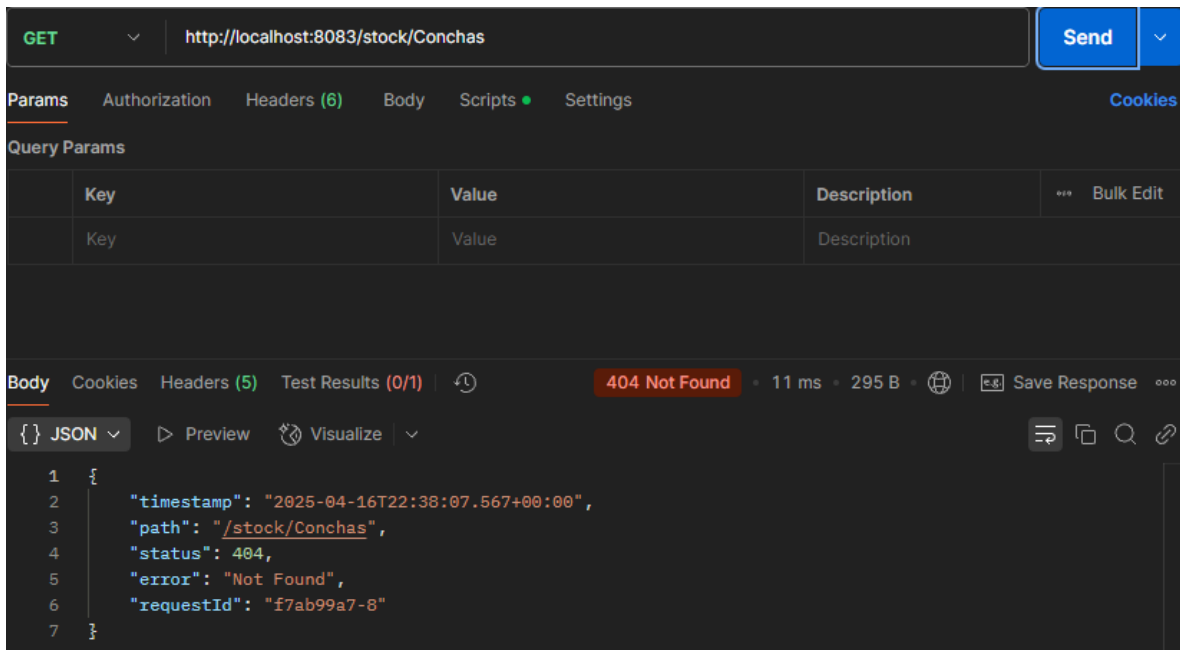


Figura 45. Prueba con Postman para obtener el stock específico de un producto.

- **POST /compras:** Se enviaron solicitudes con un cuerpo JSON que contenía el nombre del cliente y un listado de productos con sus cantidades. Cuando la compra era válida (es decir, el stock era suficiente), el sistema respondía con un mensaje de confirmación y se actualizaba automáticamente el inventario. En caso de que la cantidad solicitada superara el stock disponible, se devolvía un mensaje de error indicando la insuficiencia de productos. Esto demuestra que el sistema previene transacciones inválidas.

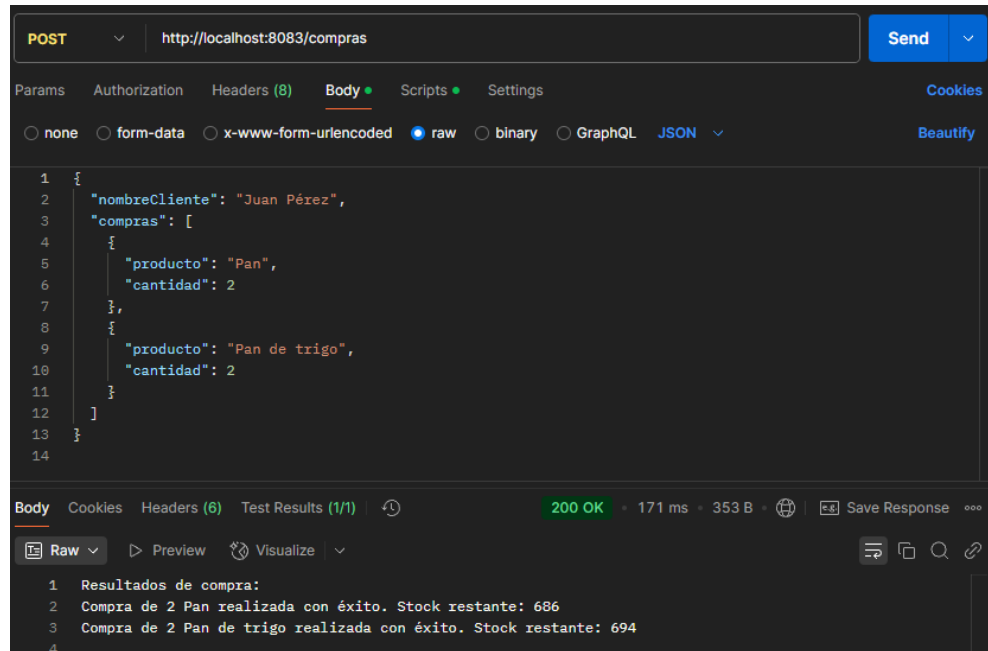


Figura 46. Prueba con Postman al realizar una compra.

Y como podemos observar en la siguiente figura, el stock se actualiza correctamente.

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

id	producto	stock
1	Pan	688
2	Pan de trigo	696
3	Pan de centeno	695
4	Pan integral	710
5	Pan de avena	690
6	Pan de maíz	705
7	Donas	702
8	Conchas	18

8 rows in set (0.01 sec)

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

id	producto	stock
1	Pan	686
2	Pan de trigo	694
3	Pan de centeno	695
4	Pan integral	710
5	Pan de avena	690
6	Pan de maíz	705
7	Donas	702
8	Conchas	18

Figura 47. Actualización del stock al realizar una compra en la BD.

```

2025-04-16 16:41:24 2025-04-16T22:41:24.258Z INFO 1 --- [compra-service] [nio-8082-exec-3] c.p.compra.service.CompraService
: Cliente 'Juan Pérez' compró 2 unidades de 'Pan'. Stock previo: 688, stock restante: 686
2025-04-16 16:41:24 2025-04-16T22:41:24.259Z INFO 1 --- [compra-service] [nio-8082-exec-3] c.p.compra.service.CompraService
: Load balancer for 'inventario-service' chose server: 172.18.0.4:8081
2025-04-16 16:41:24 2025-04-16T22:41:24.311Z INFO 1 --- [compra-service] [nio-8082-exec-3] c.p.compra.service.CompraService
: Cliente 'Juan Pérez' compró 2 unidades de 'Pan de trigo'. Stock previo: 696, stock restante: 694
2025-04-16 16:41:44 2025-04-16T22:41:44.843Z DEBUG 1 --- [compra-service] [beatExecutor-%d] com.netflix.discovery.DiscoveryClient
: DiscoveryClient_COMPRA-SERVICE/a7dda862a7b0:compra-service:8082 - Heartbeat status: 200

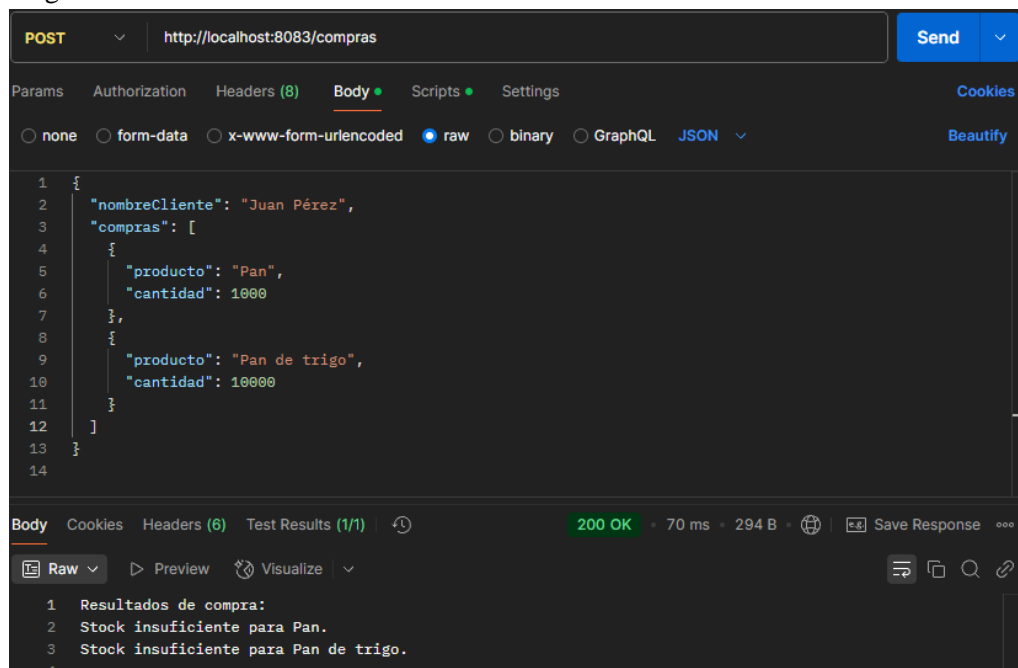
```

Figura 48. Logs de compra-service al realizar una compra.

Como se puede ver en la figura anterior, cada vez que se realiza una compra desde la interfaz, el microservicio compra-service registra en sus logs los detalles de la operación. Esto permite verificar que la solicitud fue correctamente procesada y que el stock fue actualizado en el sistema.

En este caso, al realizar la compra, se generarn las entradas que podemos ver en la Figura, además estos logs no solo confirman que la compra fue registrada correctamente, sino que también muestran cómo el servicio realiza peticiones al inventario-service a través del balanceador de carga para actualizar el stock. Además, se evidencian los mensajes de latido (*heartbeat*) enviados por el cliente Eureka, lo que indica que el microservicio continúa registrándose correctamente en el servidor de descubrimiento.

Y en caso de realizar una compra con un stock mayor al disponible, podemos ver lo siguiente junto con los logs :



```

inventario-service chose server: 172.18.0.4:8081
2025-04-16T23:01:51.343Z WARN 1 --- [compra-service] [nio-8082-exec-7] c.p.compra.service.CompraService : Cliente Juan Pérez intentó comprar 1000 unidades de 'Pan', pero el stock era de 846.
2025-04-16T23:01:51.344Z INFO 1 --- [compra-service] [nio-8082-exec-7] c.p.compra.service.CompraService : Load balancer for 'inventario-service' chose server: 172.18.0.4:8081
2025-04-16T23:01:51.354Z WARN 1 --- [compra-service] [nio-8082-exec-7] c.p.compra.service.CompraService : Cliente Juan Pérez intentó comprar 10000 unidades de 'Pan de trigo', pero el stock era de 874.

```

Figura 49. Prueba en Postman y Logs de compra-service al realizar una compra con un stock mayor.

- **Simulación de horneado automático:** Se verificó que, cada 100 segundos, se ejecuta una tarea programada en el microservicio de inventario que incrementa el stock de productos automáticamente, simulando la producción de pan. Estos eventos se reflejaron en los logs del contenedor, mostrando un mensaje cada vez que se realizaba la reposición.

```

inventario-service-1 | 2025-04-16T22:44:06.883Z INFO 1 --- [inventario-service] [ scheduling-1] com.panaderia.service.HornoServicio : Horno abastecido - Se añadieron 50 unidades de 'Pan de avena'. Stock
actual: 730
inventario-service-1 | Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
inventario-service-1 | Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.id=?
inventario-service-1 | Hibernate: update inventario set producto=?,stock=? where id=?
inventario-service-1 | Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
inventario-service-1 | 2025-04-16T22:44:06.913Z INFO 1 --- [inventario-service] [ scheduling-1] com.panaderia.service.HornoServicio : Horno abastecido - Se añadieron 50 unidades de 'Pan de maíz'. Stock
actual: 745
inventario-service-1 | Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
inventario-service-1 | Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.id=?
inventario-service-1 | Hibernate: update inventario set producto=?,stock=? where id=?
inventario-service-1 | Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
inventario-service-1 | 2025-04-16T22:44:06.945Z INFO 1 --- [inventario-service] [ scheduling-1] com.panaderia.service.HornoServicio : Horno abastecido - Se añadieron 50 unidades de 'Donas'. Stock actual
: 742

```

Figura 50. Logs en el microservicio de Inventario-Service del reabastecimiento del stock.

Y a contiinuacin podemos observar como en la Base de datos, de igual forma el stock se actualiza, en base a los productos colocados en la función de reabastecimiento.

```

mysql> select * from inventario;
+----+-----+-----+
| id | producto      | stock |
+----+-----+-----+
| 1  | Pan           | 726   |
| 2  | Pan de trigo  | 734   |
| 3  | Pan de centeno | 735   |
| 4  | Pan integral  | 750   |
| 5  | Pan de avena  | 730   |
| 6  | Pan de maíz   | 745   |
| 7  | Donas         | 742   |
| 8  | Conchas       | 18    |
+----+-----+-----+
8 rows in set (0.00 sec)

```

Figura 51. Prueba en la BD del stock reabastecido.

Y de igual forma al hacer una petición GET en Postman vemos reflejado el reabastecimiento del stock.

```

1  [
2    {
3      "id": 1,
4      "producto": "Pan",
5      "stock": 766
6    },
7    {
8      "id": 2,
9      "producto": "Pan de trigo",
10     "stock": 774
11   },
12   {
13     "id": 3,
14     "producto": "Pan de centeno",
15     "stock": 775
16   },
17   ]

```

Figura 52. Prueba en Postman del stock reabastecido.

Pruebas realizadas con el frontend

Al abrir la página en un navegador, se mostró correctamente la lista de productos con su respectivo stock, lo que confirma que la comunicación entre el frontend y el backend se estableció con éxito. Esta información se obtuvo mediante una petición GET al endpoint correspondiente expuesto por el **API Gateway** (en este caso, <http://localhost:8083/inventario/productos>). Desde el archivo `script.js` del frontend, se realiza esta solicitud para obtener dinámicamente la lista de productos disponibles en la panadería y cargarlos en la interfaz web.

¡BIENVENIDO A LA PANADERÍA DEL BARRIO!

¡El mejor pan recién horneado, directo a tu hogar!

Selecciona tus panes favoritos:

Pan - Stock: 826	0
Pan de trigo - Stock: 834	0
Pan de centeno - Stock: 835	0
Pan integral - Stock: 850	0
Pan de avena - Stock: 830	0
Pan de maíz - Stock: 845	0
Donas - Stock: 842	0
Conchas - Stock: 18	0

Datos de Compra:

Ingresa tu nombre

Figura 53. Despliegue de la lista de productos en la página.

Para probar la funcionalidad de compra, se ingresó un nombre de cliente y se seleccionaron diferentes cantidades de productos disponibles desde la interfaz web. 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>) con los datos correspondientes.

Dado que la solicitud fue válida, el backend procesó la compra correctamente, devolviendo una respuesta de confirmación que fue mostrada en la interfaz. Además, el stock de los productos se actualizó automáticamente, reflejando los cambios al volver a cargar la lista de productos disponibles mediante una nueva petición **GET**.

Selecciona tus panes favoritos:

Pan - Stock: 806	0
Pan de trigo - Stock: 834	0
Pan de centeno - Stock: 844	0
Pan integral - Stock: 820	0
Pan de avena - Stock: 820	0
Pan de maíz - Stock: 820	0
Donas - Stock: 820	0
Conchas - Stock: 8	0

Datos de Compra:

Elena Realizar Compra

Resultados de compra:
 Compra de 60 Pan realizada con éxito. Stock restante: 806
 Compra de 40 Pan de trigo realizada con éxito. Stock restante: 834
 Compra de 31 Pan de centeno realizada con éxito. Stock restante: 844
 Compra de 70 Pan integral realizada con éxito. Stock restante: 820
 Compra de 50 Pan de avena realizada con éxito. Stock restante: 820
 Compra de 65 Pan de maíz realizada con éxito. Stock restante: 820
 Compra de 62 Donas realizada con éxito. Stock restante: 820
 Compra de 10 Conchas realizada con éxito. Stock restante: 8

Figura 54. Realización de una compra desde el sitio web.

De igual forma como con Postman, en el servidor se reciben los siguientes mensajes:

```
2025-04-16T22:56:51.261Z INFO 1 --- [compra-service] [nio-8082-exec-5] c.p.compra.service.CompraService : Load balancer fo
r 'inventario-service' chose server: 172.18.0.4:8081
2025-04-16T22:56:51.302Z INFO 1 --- [compra-service] [nio-8082-exec-5] c.p.compra.service.CompraService : Cliente 'Elena'
compró 60 unidades de 'Pan'. Stock previo: 866, stock restante: 806
2025-04-16T22:56:51.303Z INFO 1 --- [compra-service] [nio-8082-exec-5] c.p.compra.service.CompraService : Load balancer fo
r 'inventario-service' chose server: 172.18.0.4:8081
2025-04-16T22:56:51.341Z INFO 1 --- [compra-service] [nio-8082-exec-5] c.p.compra.service.CompraService : Cliente 'Elena'
compró 40 unidades de 'Pan de trigo'. Stock previo: 874, stock restante: 834
2025-04-16T22:56:51.341Z INFO 1 --- [compra-service] [nio-8082-exec-5] c.p.compra.service.CompraService : Load balancer fo
r 'inventario-service' chose server: 172.18.0.4:8081
2025-04-16T22:56:51.377Z INFO 1 --- [compra-service] [nio-8082-exec-5] c.p.compra.service.CompraService : Cliente 'Elena'
compró 31 unidades de 'Pan de centeno'. Stock previo: 875, stock restante: 844
2025-04-16T22:56:51.377Z INFO 1 --- [compra-service] [nio-8082-exec-5] c.p.compra.service.CompraService : Load balancer fo
r 'inventario-service' chose server: 172.18.0.4:8081
```

Figura 55. Mensajes del servidor al realizar una compra.

Por otro lado, si se intenta comprar una cantidad mayor a la disponible, el sistema muestra un mensaje de error, indicando que el stock es insuficiente.

Selecciona tus panes favoritos:

Pan - Stock: 906	0
Pan de trigo - Stock: 934	0
Pan de centeno - Stock: 944	0
Pan integral - Stock: 920	0
Pan de avena - Stock: 920	0
Pan de maíz - Stock: 920	0
Donas - Stock: 920	0
Conchas - Stock: 8	0

Datos de Compra:

Resultados de compra:
 Stock insuficiente para Pan.
 Stock insuficiente para Conchas.

Figura 56. Intento de compra mayor al stock disponible.

Y de igual forma en el servidor se reciben mensajes de dichas compras fallidas.

```

2025-04-16T23:06:46.326Z WARN 1 --- [compra-service] [nio-8082-exec-9] c.p.compra.service.CompraService : Cliente 'Elena'
intentó comprar 1000 unidades de 'Pan', pero el stock era de 906.
2025-04-16T23:06:46.326Z INFO 1 --- [compra-service] [nio-8082-exec-9] c.p.compra.service.CompraService : Load balancer fo
r 'inventario-service' chose server: 172.18.0.4:8081
2025-04-16T23:06:46.334Z WARN 1 --- [compra-service] [nio-8082-exec-9] c.p.compra.service.CompraService : Cliente 'Elena'
intentó comprar 1000 unidades de 'Conchas', pero el stock era de 8.

```

Figura 57. Mensajes en el servidor de la compra fallida.

Tras una compra exitosa, el frontend volvía a solicitar la lista de productos al servidor mediante un nuevo **GET**, reflejando en tiempo real la actualización del stock como se puede observar en las figuras 53 y 54. También se observó que, pasados algunos minutos, los valores del stock aumentaban debido a la simulación del horneado automático, el cual ocurre cada 100 segundo. Este comportamiento fue validado revisando los mensajes en la consola del servidor, donde se registraba cada reposición de panes. Y pues en cada Figura mostrada, hemos visto como el stock va aumentando cada cierto tiempo.

Respecto a algunas pruebas de validación, en caso de que el usuario coloque únicamente su nombre y no realice alguna compra, se le indica que al menos debe seleccionar un producto, o en caso de seleccionar uno o mas productos, se le solicita al usuario que debe colocar un nombre.

Selecciona tus panes favoritos:

Pan - Stock: 926	0
Pan de trigo - Stock: 954	0
Pan de centeno - Stock: 964	0
Pan integral - Stock: 940	0
Pan de avena - Stock: 940	0
Pan de maíz - Stock: 940	0
Donas - Stock: 940	0
Conchas - Stock: 8	0

Datos de Compra:

Elena

Por favor, selecciona al menos un producto.

Figura 58. Solicitud al usuario de que seleccione al menos un producto.

Selecciona tus panes favoritos:

Pan - Stock: 926	0
Pan de trigo - Stock: 954	0
Pan de centeno - Stock: 964	0
Pan integral - Stock: 940	2
Pan de avena - Stock: 940	3
Pan de maíz - Stock: 940	4
Donas - Stock: 940	0
Conchas - Stock: 8	8

Datos de Compra:

Por favor, ingresa tu nombre.

Figura 59. Solicitud al usuario para que ingrese su nombre al realizar una compra.

Las pruebas realizadas demuestran que el sistema cumple satisfactoriamente con las funcionalidades esperadas. Tanto el backend como el frontend mantienen una comunicación fluida y estable. La gestión del inventario, el control de compras y la reposición automática de productos se realizaron sin inconvenientes. El diseño basado en microservicios permitió una distribución eficiente de los

componentes, facilitando su mantenimiento y escalabilidad. Con esto, se valida que la solución propuesta es efectiva para la administración distribuida de una panadería en tiempo real.

Conclusión

Esta práctica me permitió profundizar en el desarrollo de sistemas distribuidos mediante microservicios, una arquitectura ampliamente utilizada en aplicaciones modernas por su flexibilidad, escalabilidad y capacidad de mantenimiento. A lo largo de su desarrollo, adquirí una visión más clara sobre cómo dividir una aplicación en componentes independientes que se comunican entre sí, así como los beneficios y desafíos que esto implica en entornos reales.

Uno de los aprendizajes más importantes fue el uso de **Spring Boot** para la creación de microservicios RESTful. Comprendí la importancia de estructurar correctamente cada servicio, asignándole una responsabilidad específica (como compras, inventario o cliente), y cómo configurar controladores, servicios y repositorios para lograr una arquitectura sólida. La interacción con bases de datos utilizando **Spring Data JPA** y **MySQL** también fue clave para garantizar la persistencia y consistencia de los datos.

La implementación del frontend, aunque sencilla, me permitió comprender cómo una interfaz gráfica puede consumir datos desde múltiples servicios a través de un API Gateway, en este caso utilizando **Spring Cloud Gateway**. Pude reforzar conceptos como las peticiones HTTP, el manejo de respuestas del backend y la validación básica desde el cliente para mejorar la experiencia del usuario.

Uno de los desafíos más interesantes fue la incorporación de una **tarea programada para simular el horneado de productos**, lo que me permitió experimentar con procesos automáticos dentro de una arquitectura distribuida. Esta parte me ayudó a comprender cómo los microservicios también pueden incluir tareas internas que se ejecutan periódicamente, y la importancia de que estas tareas no interfieran con el resto del sistema.

Sin embargo, no todo fue sencillo. Al intentar contenerizar la aplicación usando **Docker**, surgieron algunos problemas al compilar el proyecto con `mvn clean install`, principalmente debido a que los tests definidos en clases como `ApplicationTest` no estaban correctamente configurados para un entorno distribuido o aislado. Esto provocaba errores durante la generación de los archivos `.jar`, lo que me obligó a investigar cómo desactivar los tests temporalmente o reestructurar las pruebas para que no afectaran el proceso de construcción. Estos inconvenientes me permitieron entender lo delicado que puede ser el paso de un entorno local a un entorno contenerizado, y la necesidad de que cada microservicio sea lo más independiente y desacoplado posible para facilitar su empaquetado y despliegue.

Además, gestionar múltiples servicios simultáneamente me ayudó a valorar la importancia de herramientas como **Eureka** para el descubrimiento dinámico de servicios, especialmente cuando las direcciones IP y puertos pueden cambiar durante la ejecución. También entendí que al escalar un sistema así, se deben considerar estrategias adicionales para el balanceo de carga, seguridad, y tolerancia a fallos, aspectos que son fundamentales en arquitecturas reales basadas en microservicios.

En general, esta práctica fue una experiencia enriquecedora que me permitió aplicar conocimientos teóricos sobre sistemas distribuidos en un entorno práctico y actual. Aprendí sobre la implementación de microservicios, la configuración de un API Gateway, el manejo de servicios REST, el uso de contenedores, la gestión de tareas programadas y la integración de frontend con backend. También experimenté de primera mano algunos de los problemas que pueden surgir al trabajar con proyectos

distribuidos y cómo resolverlos. Esta experiencia me dio una visión más completa sobre el desarrollo de software moderno y me preparó espero, para las siguientes prácticas.

Bibliografía

- Microsoft Learn, "Conceptos de Sistemas Distribuidos," 2022. [En línea]. Disponible en: <https://learn.microsoft.com/es-es/azure/architecture/patterns/>. [Último acceso: 01 abril 2025].
- Spring, "Introducción a Spring Boot," 2024. [En línea]. Disponible en: <https://spring.io/projects/spring-boot>. [Último acceso: 01 abril 2025].
- Spring Cloud, "Guía de Spring Cloud Netflix Eureka," 2024. [En línea]. Disponible en: https://cloud.spring.io/spring-cloud-netflix/multi/multi__service_discovery_eureka_clients.html. [Último acceso: 01 abril 2025].
- Docker Docs, "Getting Started with Docker," 2024. [En línea]. Disponible en: <https://docs.docker.com/get-started/>. [Último acceso: 01 abril 2025].
- Apache Maven, "Descarga e instalación de Maven," 2024. [En línea]. Disponible en: <https://maven.apache.org/download.cgi>. [Último acceso: 01 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: 01 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: 01 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: 01 abril 2025].
- A. 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-Servicio-Web_fig7_305403120. [Último acceso: 01 abril 2025].
- Disrupción Tecnológica, "Arquitectura de Servicios Web," 2024. [En línea]. Disponible en: <https://www.disrupciontecnologica.com/arquitectura-de-servicios-web/>. [Último acceso: 01 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: 01 abril 2025].
- Chakray, "Experiencia con Arquitecturas de Microservicios," 2024. [En línea]. Disponible en: <https://chakray.com/es/experiencia/microservicios/>. [Último acceso: 01 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: 01 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: 01 abril 2025].

Anexos

Respecto a este apartado, a continuación se mencionan los elementos técnicos relevantes para la implementación del sistema distribuido desarrollado en esta práctica. Todos los archivos se incluyen en la carpeta adjunta en el drive titulada “**Microservicios_Panaderia.zip**”, organizada por microservicio y componentes de los proyectos.

- **Anexo A – Microservicio de Inventario:** contiene los archivos `InventarioController.java`, `InventarioService.java`, `InventarioRepository.java` y `application.properties`.
- **Anexo B – Microservicio de Compras:** incluye `CompraController.java`, `CompraService.java`, `CompraRepository.java` y configuración.
- **Anexo C – Microservicio de Cliente:** con su respectiva lógica y configuración.
- **Anexo D – Eureka Server y API Gateway:** configuración básica y `application.properties`.
- **Anexo E – Frontend:** archivo `index.html` junto con los estilos y scripts utilizados.
- **Anexo F – Docker y configuración general:** incluye `docker-compose.yml`, `Dockerfile` y `pom.xml` de cada microservicio, además del `init.sql` para inicializar un script en la BD.

Para consultar el código completo, por favor revise la carpeta **Microservicios_Panaderia.zip** incluida en este drive.