



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Actividad:

Practica 6 – Servicios Web

Integrante:

Hernández Vázquez Jorge Daniel

Profesor:

Chadwick Carreto Arellano

Fecha de entrega:

02/04/2025

INSTITUTO POLITÉCNICO NACIONAL



Índice de contenido

Antecedente.....	1
Servicios Web	1
Características de los Servicios Web	2
Ventajas y Aplicaciones de los servicios web en Sistemas Distribuidos	2
Comparación entre algunos Modelos de Arquitectura	3
Servicios Web RESTful	4
Diferencias entre REST y SOAP	5
Métodos HTTP Utilizados en una API REST	6
Evolución de los Modelos de Comunicación en Sistemas Distribuidos	7
Tecnologías Utilizadas: Spring Boot.....	8
Ventajas de Spring Boot en el Desarrollo de APIs	8
Estructura básica de una API con Spring Boot	9
Planteamiento del problema	10
Propuesta de solución.....	10
Materiales y métodos empleados	12
Desarrollo de la solución.....	15
Instrucciones para ejecutar el proyecto	37
Resultados	39
Conclusión.....	49
Bibliografía	51
Anexos.....	52
Controller/PanaderiaControlador.java.....	52
Model/Compra.java.....	52
Model/CompraRequest.java.....	53
Model/Pan.java.....	53
Repository/PanRepository.java.....	54
Service/PanaderiaServicio.java	54
application.properties	56

Índice de Figuras

Figura 1	5
Figura 2	6
Figura 3	7

Figura 4	9
Figura 5	16
Figura 6	16
Figura 7	17
Figura 8	17
Figura 9	17
Figura 10	18
Figura 11	18
Figura 12	19
Figura 13	19
Figura 14	19
Figura 15	20
Figura 16	21
Figura 16	21
Figura 17	22
Figura 18	23
Figura 19	24
Figura 20	25
Figura 21	26
Figura 22	27
Figura 23	28
Figura 24	30
Figura 25	32
Figura 26	32
Figura 27	33
Figura 28	34
Figura 29	34
Figura 30	35
Figura 31	36
Figura 32	39
Figura 33	40
Figura 34	40
Figura 35	41
Figura 36	41
Figura 37	41

Figura 38	42
Figura 39	42
Figura 40	43
Figura 41	44
Figura 42	44
Figura 43	45
Figura 44	45
Figura 45	46
Figura 46	46
Figura 47	47
Figura 48	47
Figura 49	48
Figura 50	48

Antecedente

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

Servicios Web

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

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

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

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

En términos generales, un servicio web actúa como una interfaz remota que expone ciertas operaciones o funcionalidades para ser consumidas por otros sistemas, sin necesidad de que estos compartan el mismo entorno de desarrollo o infraestructura. Cuando una aplicación cliente necesita acceder a un servicio web, realiza una solicitud a un servidor, que a su vez procesa la petición, ejecuta las operaciones necesarias y devuelve una respuesta estructurada, generalmente en formato JSON o XML.

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

optimizar flujos de trabajo y mejorar la accesibilidad a la información de manera estructurada y segura.

Características de los Servicios Web

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

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

Ventajas y Aplicaciones de los servicios web en Sistemas Distribuidos

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

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

Entre sus aplicaciones más comunes se encuentran:

1. **Sistemas empresariales:** Integración de plataformas ERP y CRM con otros servicios internos y externos.

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

Comparación entre algunos Modelos de Arquitectura

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

Característica	Monolítica	Cliente-Servidor	Multiservidor/Multicliente	Objetos Distribuidos	Servicios Web
Estructura	Todo el sistema en una única aplicación	Un servidor central atiende múltiples clientes	Múltiples servidores gestionan múltiples clientes	Componentes distribuidos mediante objetos accesibles en red	Componentes distribuidos en red
Escalabilidad	Baja, requiere rediseño completo	Media, depende del servidor	Alta, se pueden agregar servidores según demanda	Alta, objetos pueden replicarse en múltiples nodos	Alta, basada en microservicios
Mantenimiento	Complejo, cualquier cambio afecta a todo el sistema	Medio, pero el servidor puede convertirse en un cuello de botella	Mayor facilidad para mantener servicios individuales	Puede ser complejo debido a la comunicación entre objetos	Sencillo, cada servicio puede actualizarse de forma independiente
Interoperabilidad	Baja, difícil integrar nuevas tecnologías	Media, limitada a las capacidades del servidor	Media-Alta, dependiendo de los protocolos utilizados	Media, depende del middleware utilizado (CORBA, RMI)	Alta, comunicación mediante estándares abiertos (HTTP, JSON, XML)
Ejemplo de uso	Aplicaciones de escritorio tradicionales	Aplicaciones web básicas con backend centralizado	Plataformas de streaming, videojuegos en línea	Sistemas de control distribuidos, aplicaciones empresariales en red	APIs, sistemas de microservicios en la nube

Entonces, como pudimos ver, los Objetos Distribuidos fueron una evolución natural del modelo cliente-servidor, ya que permitieron una comunicación más eficiente entre diferentes sistemas a través de tecnologías como Java RMI y CORBA. Sin embargo, su complejidad y dependencia de tecnologías

específicas dificultó su adopción masiva en comparación con los Servicios Web, que utilizan estándares abiertos y son más accesibles desde cualquier plataforma.

Los Servicios Web han logrado combinar las ventajas del modelo Multiservidor/Multicliente y Objetos Distribuidos, proporcionando escalabilidad y facilidad de mantenimiento sin las restricciones de middleware propietario.

Servicios Web RESTful

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

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

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

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

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.

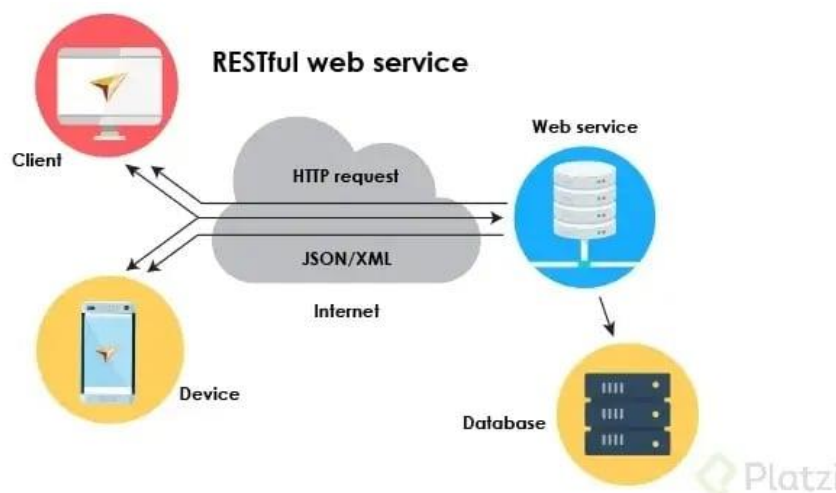


Figura 1. Estructura básica de un servicio web REST.

Diferencias entre REST y SOAP

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

Aspecto	REST	SOAP
Protocolo	Se basa en HTTP.	Puede utilizar múltiples protocolos (HTTP, SMTP, TCP, etc.).
Formato de datos	JSON, XML, HTML, texto plano, entre otros.	Exclusivamente XML.
Interoperabilidad	Alta interoperabilidad entre distintos sistemas y lenguajes.	Requiere herramientas específicas para interpretar XML y SOAP.
Complejidad	Simple, fácil de implementar y ligero.	Más complejo debido a su estructura basada en XML y WSDL.
Seguridad	Puede usar autenticación mediante OAuth, JWT y HTTPS.	Usa WS-Security para seguridad avanzada y control granular.
Estado	Sin estado (cada solicitud es independiente).	Puede ser con o sin estado (stateful/stateless).
Velocidad y rendimiento	Rápido debido a su estructura ligera.	Más lento debido al procesamiento de XML.

Casos de uso	Aplicaciones web, APIs públicas, microservicios, aplicaciones móviles.	Sistemas bancarios, telecomunicaciones, transacciones empresariales.
---------------------	--	--

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

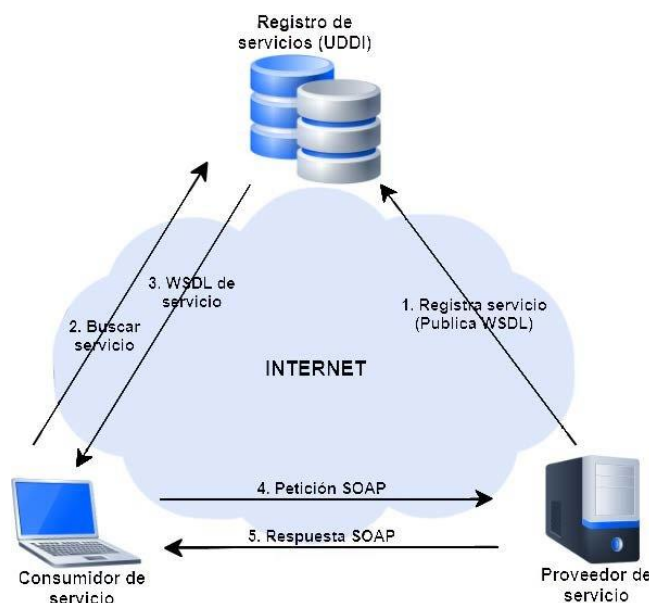


Figura 2. Estructura básica de un servicio web SOAP.

Métodos HTTP Utilizados en una API REST

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

1. GET (Obtener Datos)

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

2. POST (Crear un Nuevo Recurso)

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

3. PUT (Actualizar un Recurso Existente)

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

4. DELETE (Eliminar un Recurso)

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

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

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

Figura 3. Ejemplo del uso de métodos HTTP.

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

Si bien REST y SOAP tienen aplicaciones distintas, REST se ha convertido en la elección predominante para APIs modernas, microservicios y aplicaciones móviles. El uso correcto de los métodos HTTP en una API RESTful garantiza una comunicación eficiente entre clientes y servidores, optimizando el desempeño y la interoperabilidad en sistemas distribuidos.

Evolución de los Modelos de Comunicación en Sistemas Distribuidos

La evolución de la comunicación en sistemas distribuidos ha estado impulsada por la necesidad de mejorar la interoperabilidad, la escalabilidad y la facilidad de desarrollo.

Si bien RMI, CORBA y DCOM fueron fundamentales en la construcción de aplicaciones distribuidas en las décadas de 1990 y 2000, la aparición de **arquitecturas basadas en microservicios y servicios web** ha llevado a la adopción de nuevos enfoques, como:

- **REST (Representational State Transfer):** Utiliza HTTP como protocolo de comunicación y se ha convertido en el estándar de facto para la integración de sistemas distribuidos en la web.
- **gRPC:** Desarrollado por Google, permite la comunicación eficiente entre servicios utilizando **Protocol Buffers** en lugar de JSON o XML, lo que reduce el consumo de ancho de banda.
- **Arquitecturas basadas en eventos:** Tecnologías como **Kafka** y **RabbitMQ** permiten la comunicación asíncrona entre sistemas distribuidos, mejorando la escalabilidad y tolerancia a fallos.

A pesar del desplazamiento de tecnologías como RMI, CORBA y DCOM en favor de soluciones más modernas, estas siguen siendo relevantes en sistemas heredados y en entornos donde se requiere una fuerte integración con plataformas específicas. La elección del modelo de comunicación adecuado dependerá de los requisitos del sistema, incluyendo el rendimiento, la interoperabilidad y la facilidad de implementación.

Tecnologías Utilizadas: 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.

Ventajas de Spring Boot en el Desarrollo de APIs

Spring Boot aporta numerosas ventajas al desarrollo de APIs RESTful, lo que lo convierte en una de las tecnologías más utilizadas en el mundo empresarial.

1. Configuración rápida y sencilla

Spring Boot elimina la necesidad de escribir extensos archivos de configuración XML, reduciendo el tiempo de desarrollo y evitando errores de configuración.

2. Integración con librerías y frameworks populares

Se integra fácilmente con bases de datos SQL y NoSQL (MySQL, PostgreSQL, MongoDB, Redis, etc.), herramientas de monitoreo (Spring Actuator), mensajería (RabbitMQ, Kafka) y seguridad (OAuth2, JWT, LDAP).

3. Gestión eficiente de dependencias

Spring Boot usa Maven o Gradle para manejar automáticamente las dependencias necesarias del proyecto, evitando problemas de compatibilidad entre bibliotecas.

4. Creación de APIs REST en pocos pasos

Permite crear endpoints REST con una simple anotación `@RestController`, evitando la configuración manual de rutas y manejadores HTTP.

5. Servidor embebido

Incluye servidores como Tomcat, Jetty o Undertow, lo que significa que la aplicación puede ejecutarse sin necesidad de instalar un servidor web por separado.

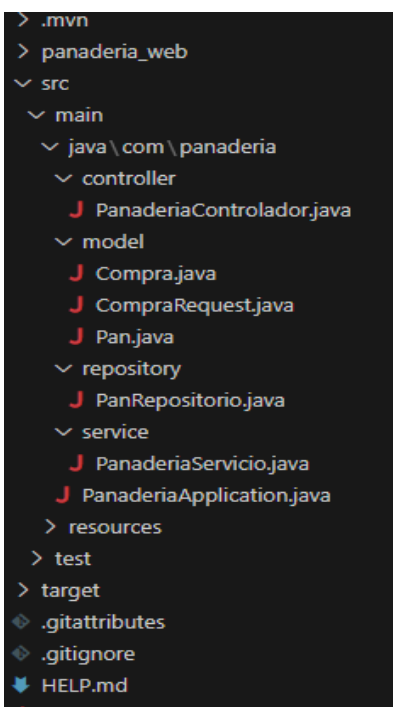
7. Escalabilidad y compatibilidad con la nube

Diseñado para funcionar en entornos en la nube como AWS, Azure o Google Cloud, permitiendo desplegar APIs REST en contenedores Docker y orquestadores como Kubernetes.

Estructura básica de una API con Spring Boot

La estructura típica de un proyecto Spring Boot sigue un modelo organizado en capas para separar la lógica de negocio, los controladores REST y la comunicación con la base de datos.

Y un ejemplo, sería el siguiente, el de mi practica:



```
> .mvn
> panaderia_web
  > src
    > main
      > java\com\panaderia
        > controller
          > PanaderiaControlador.java
        > model
          > Compra.java
          > CompraRequest.java
          > Pan.java
        > repository
          > PanRepositorio.java
        > service
          > PanaderiaServicio.java
          > PanaderiaApplication.java
      > resources
    > test
    > target
  > .gitattributes
  > .gitignore
  > HELP.md
```

Figura 4. Estructura de mi practica con Spring Boot.

Donde:

1. **Controller (controller/)**: Maneja las solicitudes HTTP y define los endpoints REST.
2. **Model (model/)**: Contiene las clases que representan las entidades de la base de datos.

3. **Repository (repository/)**: Interactúa con la base de datos utilizando Spring Data JPA.
4. **Service (service/)**: Contiene la lógica de negocio y operaciones sobre los datos.
5. **application.properties**: Archivo de configuración donde se definen los parámetros del proyecto.

En general, Spring Boot es una herramienta poderosa y flexible para el desarrollo de APIs RESTful en Java. Gracias a su capacidad de autoconfiguración, integración con múltiples tecnologías y facilidad de uso, se ha convertido en la opción preferida para empresas y desarrolladores.

El enfoque modular de Spring Boot permite la creación de aplicaciones escalables y de alto rendimiento, optimizando el desarrollo de servicios web en sistemas distribuidos.

Planteamiento del problema

El objetivo de esta práctica es desarrollar un servicio web, en mi caso haciendo uso de REST que permita la gestión distribuida del inventario de una panadería, facilitando la consulta de productos disponibles, la compra de pan y la actualización automática del stock. A través de esta implementación, se busca comprender los principios de los sistemas distribuidos y su aplicación en entornos web mediante tecnologías como **Spring Boot y MySQL**.

En un sistema distribuido, donde múltiples clientes pueden interactuar simultáneamente con los mismos datos, surge la necesidad de garantizar la consistencia y disponibilidad de la información. En este caso, al permitir que varios clientes realicen compras en tiempo real, es esencial evitar inconsistencias en el inventario, asegurando que las transacciones se realicen correctamente y que no se vendan productos fuera de stock. Para ello, el sistema debe implementar mecanismos que gestionen la concurrencia y aseguren la integridad de los datos en la base de datos.

Otro aspecto clave de esta práctica es la automatización de la producción de pan, simulada mediante un proceso interno que incrementará periódicamente el stock disponible. Esta funcionalidad se implementará utilizando una tarea programada en Spring Boot, lo que permitirá que el sistema actualice automáticamente el inventario sin necesidad de intervención manual.

A lo largo de la práctica, se explorarán conceptos fundamentales de los sistemas distribuidos, como la comunicación a través de servicios web REST, la manipulación de datos en un entorno multiusuario y la implementación de tareas programadas. La API desarrollada podrá integrarse con diferentes interfaces o aplicaciones clientes, facilitando la interacción con el sistema y proporcionando una solución escalable para la gestión del inventario de la panadería.

Esta práctica me permitirá comprender los desafíos asociados a la administración de datos en sistemas distribuidos, aplicando buenas prácticas de desarrollo en Spring Boot, el uso de bases de datos MySQL y la correcta gestión de transacciones concurrentes. Al finalizar la práctica, espero tener una mejor comprensión de cómo funcionan los servicios web en escenarios del mundo real y cómo pueden aplicarse para resolver problemas en entornos distribuidos.

Propuesta de solución

Para abordar el problema planteado, se propone el desarrollo de un servicio web RESTful utilizando **Spring Boot**, que permitirá la gestión del inventario de una panadería de manera distribuida. A través de este sistema, múltiples clientes podrán conectarse a una API para consultar productos, realizar compras y recibir actualizaciones sobre la disponibilidad del stock.

La solución estará compuesta por los siguientes componentes principales:

1. Backend con Spring Boot

El sistema será desarrollado utilizando **Spring Boot**, un framework de Java que facilita la creación de aplicaciones web escalables y eficientes. El backend se encargará de:

- **Exponer una API RESTful** que permitirá a los clientes realizar solicitudes HTTP para consultar productos, verificar stock y comprar pan.
- **Gestionar la lógica de negocio** mediante un servicio que procesará las compras, actualizará el stock y registrará las transacciones.
- **Automatizar la reposición del inventario** mediante una tarea programada (@Scheduled en Spring Boot), simulando el horneado de pan en intervalos regulares.
- **Registrar actividades del sistema** utilizando logs para monitorear las operaciones realizadas.

2. API RESTful y Comunicación con los Clientes

El acceso al sistema se realizará mediante una API REST, la cual ofrecerá los siguientes **endpoints**:

- GET /panaderia/productos: Devuelve la lista de productos disponibles en la panadería.
- GET /panaderia/stock/{producto}: Consulta el stock de un producto específico.
- POST /panaderia/comprar: Permite a un cliente realizar una compra enviando una lista de productos.

La API utilizará el formato **JSON** para la comunicación con los clientes, asegurando compatibilidad con distintos tipos de aplicaciones y dispositivos.

3. Base de Datos MySQL para la Gestión del Inventario

El sistema almacenará la información de los productos y sus existencias en una base de datos **MySQL**, la cual permitirá:

- **Consultar y actualizar el stock** en tiempo real, asegurando que la información sea precisa.
- **Ejecutar transacciones de manera segura**, evitando inconsistencias cuando varios clientes realicen compras simultáneamente.
- **Registrar las compras realizadas**, manteniendo un historial de ventas que facilite el monitoreo del sistema.

4. Mecanismos de Concurrencia y Seguridad en la Base de Datos

Para garantizar que las compras se procesen de manera correcta y sin interferencias entre múltiples clientes, se implementarán mecanismos de control de concurrencia:

- **Transacciones en MySQL:** Se asegurará que las operaciones de compra se realicen de manera atómica, evitando que dos clientes adquieran el mismo producto si hay pocas unidades en stock.

- **Bloqueo optimista y control de stock:** Se verificará la cantidad disponible antes de confirmar una compra, asegurando que el cliente solo pueda adquirir lo que realmente está disponible.

5. Automatización del Reabastecimiento del Inventario

Para simular la producción de pan en una panadería real, el sistema incluirá un proceso automático que incrementará el stock de productos en intervalos regulares. Esto se logrará mediante una **tarea programada en Spring Boot (@Scheduled)**, la cual ejecutará la actualización del inventario cada cinco minutos, agregando una cantidad predeterminada de cada tipo de pan.

6. Registro de Actividades y Monitoreo

El sistema implementará un mecanismo de **registro de eventos mediante logs**, lo que permitirá:

- Monitorear las transacciones realizadas, registrando cada compra y la cantidad de productos adquiridos.
- Controlar la ejecución de la tarea programada para asegurar que la reposición de stock se realice correctamente.
- Detectar posibles errores en el sistema, facilitando su depuración y mantenimiento.

Con esta solución, se desarrollará un sistema distribuido funcional y eficiente que permitirá la gestión del inventario de una panadería a través de un servicio web REST. La implementación con Spring Boot y MySQL garantizará una arquitectura escalable, mientras que los mecanismos de concurrencia asegurarán que las transacciones sean seguras y confiables. A través de esta práctica, se explorarán los fundamentos de los sistemas distribuidos basados en servicios web, incluyendo la comunicación cliente-servidor, la gestión de datos en entornos multiusuario y la automatización de procesos en una infraestructura web.

Materiales y métodos empleados

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

Materiales (Herramientas utilizadas):

1. **Lenguaje de programación:** Java Se utilizó Java debido a su robustez y fiabilidad en aplicaciones distribuidas, así como su excelente soporte para la creación de servicios web RESTful. Java proporciona un entorno adecuado para el desarrollo de aplicaciones de backend escalables y seguras, utilizando bibliotecas y frameworks ampliamente soportados.
2. **Framework:** Spring Boot Para la creación del servicio web, se empleó Spring Boot, un framework de desarrollo basado en Java que facilita la construcción de aplicaciones backend de manera rápida y eficiente. Spring Boot permite crear aplicaciones con configuración mínima y soporta el desarrollo de servicios RESTful a través de controladores y servicios bien estructurados, integrando componentes como la programación de tareas y la gestión de base de datos.
3. **Base de Datos:** MySQL fue utilizado como sistema de gestión de bases de datos para almacenar el inventario de productos y las transacciones realizadas por los clientes. MySQL es conocido por su fiabilidad, escalabilidad y facilidad de integración con aplicaciones

basadas en Java, lo que lo hace adecuado para la gestión de grandes volúmenes de datos en aplicaciones web.

4. **Herramienta de gestión de dependencias:** Maven se utilizó como herramienta para la gestión de dependencias y la construcción del proyecto. Maven facilita la configuración de bibliotecas necesarias y la automatización de tareas relacionadas con el ciclo de vida del proyecto, como la compilación, pruebas y empaquetado.
5. **Herramienta de pruebas de API:** Para probar los endpoints de la API RESTful, se utilizó Postman, una herramienta que permite realizar solicitudes HTTP y analizar las respuestas de la API de manera sencilla. Postman facilitó la prueba de los diferentes métodos del servicio web, como la consulta de productos, verificación del stock y realización de compras.
6. **Entorno de desarrollo: Visual Studio Code (VSCode)** Visual Studio Code fue el editor de código utilizado en el desarrollo de la práctica. Este editor es ligero, rápido y cuenta con diversas extensiones que facilitaron la programación en Java, la integración con Maven, y la depuración de código.
7. **JDK (Java Development Kit):** Se utilizó JDK 21 para compilar y ejecutar el código. Este entorno de desarrollo incluye las herramientas necesarias para compilar el código fuente, ejecutar las aplicaciones Java y realizar pruebas, asegurando la compatibilidad con las versiones más recientes del lenguaje.

Métodos Empleados

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

A continuación, se describen los principales métodos utilizados:

1. **Arquitectura Cliente Servidor Distribuida:** Se implementó un sistema **cliente-servidor** basado en **servicios web REST** mediante Spring Boot. El servidor expone una serie de endpoints que permiten a los clientes realizar operaciones remotas como consultar el inventario, verificar el stock y realizar compras. Los clientes interactúan con el sistema a través de solicitudes HTTP utilizando los métodos **GET**, **POST**.
2. **Concurrencia en el Acceso al Inventario:** Para manejar solicitudes simultáneas y evitar condiciones de carrera, el acceso al inventario se gestionó cuidadosamente. El backend utilizó **transacciones** para asegurar que los cambios en el stock se realizaran de manera atómica, evitando que varios clientes compren la misma unidad de un producto si no hay stock disponible. Las actualizaciones del inventario se realizaron mediante **operaciones controladas** que aseguran la consistencia de los datos.
3. **Automatización del Reabastecimiento del Inventario:** Se implementó un proceso automatizado para **restablecer el stock** de productos a intervalos regulares. Utilizando la anotación `@Scheduled` de Spring Boot, el sistema simula la producción continua de pan, agregando una cantidad predeterminada de productos al inventario cada 5 minutos. Esta tarea garantiza que el stock se mantenga actualizado sin intervención manual.
4. **Manejo de Solicitudes Concurrentes:** El sistema está diseñado para manejar múltiples solicitudes concurrentes de clientes mediante la arquitectura **sincrónica** de Spring Boot. Al ser un servicio web RESTful, el servidor puede procesar varias solicitudes al mismo tiempo,

manteniendo la eficiencia y asegurando que cada cliente reciba una respuesta adecuada en tiempo real.

5. **Pruebas de API:** Se utilizó **Postman** para probar los diferentes endpoints del servicio web. Esto permitió verificar que los métodos de la API (como la consulta de productos, la compra de pan y la actualización de stock) funcionaran correctamente bajo diversas condiciones. Las pruebas incluyeron solicitudes de compra con stock suficiente, intentos de compra con stock insuficiente y verificaciones de inventario.
6. **Registro de Actividades y Monitoreo:** Para monitorear el comportamiento del sistema y facilitar la depuración, se implementaron registros detallados en **Spring Boot** mediante la librería **SLF4J**. Esto permitió realizar un seguimiento de las operaciones del sistema, incluyendo las compras realizadas, los cambios en el stock y las actualizaciones automáticas del inventario. Los registros proporcionaron información clave para garantizar el correcto funcionamiento del sistema y detectar posibles errores.

Con estas herramientas y métodos, se logró desarrollar un sistema distribuido robusto que simula la gestión de inventario en una panadería utilizando servicios web RESTful, garantizando la integridad de los datos y permitiendo una interacción eficiente entre el servidor y los clientes.

Después de describir los materiales y métodos empleados, es importante mencionar la integración del frontend con el backend para proporcionar una experiencia completa al usuario. A continuación, se describe cómo se implementó la interfaz de usuario (UI) y la conexión con los servicios web:

Interfaz de Usuario (Frontend)

El sistema también incluye una interfaz de usuario diseñada con HTML, CSS y JavaScript, la cual permite a los usuarios interactuar con el servicio web de manera sencilla y accesible.

1. HTML: Estructura de la Página

Se utilizó HTML para construir la estructura de la página web. En el archivo `index.html`, se creó una página que presenta una bienvenida al usuario, muestra una lista de productos disponibles y permite realizar compras a través de un formulario interactivo. La interfaz permite ingresar el nombre del cliente y seleccionar los productos que desea comprar, indicando la cantidad deseada.

2. CSS: Estilo y Diseño

El archivo `style.css` fue utilizado para mejorar la apariencia de la interfaz. A través de CSS, se aplicaron estilos visuales para que la página fuera más atractiva y fácil de usar. Se personalizaron los botones, campos de entrada y la disposición general de los elementos para que la experiencia del usuario fuera intuitiva.

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

El archivo `script.js` contiene la lógica para interactuar con el servidor. A través de JavaScript, se realiza la comunicación con el backend utilizando **fetch API** para hacer solicitudes **GET** y **POST**. La función `obtenerProductos` consulta el stock disponible de los productos, mientras que la función `comprarPan` envía los datos de la compra al backend para su procesamiento. Además, se actualiza el stock en tiempo real en la interfaz después de que se realice una compra.

Detalles del flujo de trabajo:

- **Obtener Productos:** Cuando el cliente entra en la página, la función `obtenerProductos()` es llamada para cargar los productos disponibles desde el servidor. Esta función hace una solicitud **GET** al endpoint `/productos` del servicio backend, obteniendo los detalles del inventario y mostrándolos dinámicamente en la página.
 - **Realizar Compra:** Cuando el usuario selecciona los productos y hace clic en el botón de compra, la función `comprarPan()` recopila los datos ingresados (nombre del cliente y cantidades de productos) y los envía al backend a través de una solicitud **POST**. El servidor procesará la compra y responderá con un mensaje que se mostrará al usuario en la página.
 - **Actualización Dinámica:** Después de cada compra, el sistema vuelve a consultar los productos disponibles mediante `obtenerProductos()`, asegurando que el stock se mantenga actualizado en la interfaz del usuario.
4. **Conexión con el Backend:** El frontend se comunica con el **backend Spring Boot** a través de la URL base `http://localhost:8081/panaderia`, que está definida en la constante `API_URL` en el archivo JavaScript. El frontend hace solicitudes **GET** para obtener la lista de productos y **POST** para registrar las compras realizadas. La integración entre ambos componentes asegura una interacción fluida y eficiente entre el cliente y el servidor.

El sistema permite a los clientes visualizar la disponibilidad en tiempo real, realizar compras de manera sencilla y recibir actualizaciones sobre el estado del inventario, lo que mejora la experiencia del usuario (los códigos mencionados en el texto, los podemos ver en el apartado de Anexos).

Desarrollo de la solución

Para el desarrollo de la solución, primeramente, fue necesario generar un proyecto en Spring Boot utilizando Spring Initializr, asegurándonos de incluir las dependencias necesarias para la implementación de un servicio web REST con conexión a una base de datos MySQL. La solución implementada para la gestión distribuida del inventario en la panadería se basa en un modelo Cliente/Servidor con múltiples servidores sincronizados a través de una base de datos MySQL. Se ha diseñado un sistema que permite a varios clientes conectarse simultáneamente a distintos servidores, realizar compras y actualizar el inventario sin inconsistencias.

Creación del proyecto en Spring Boot

Para estructurar el backend de la panadería distribuida, seguí los siguientes pasos:

1. **Acceder a Spring Initializr** (<https://start.spring.io>) y configurar el proyecto con las siguientes opciones:
 - **Project:** Maven
 - **Language:** Java
 - **Spring Boot:** 3.3.10
 - **Group:** com.panaderia

- **Artifact:** panaderia
 - **Name:** panaderia
 - **Package Name:** com.panaderia
 - **Packaging:** Jar
 - **Java:** 21
2. **Seleccionar las dependencias necesarias:**
 - **Spring Web** (para exponer el servicio REST)
 - **Spring Data JPA** (para la gestión de la base de datos)
 - **MySQL Driver** (para la conexión con MySQL)
 3. **Descargar el proyecto ZIP**, descomprimirlo y abrirlo en **Visual Studio Code**.

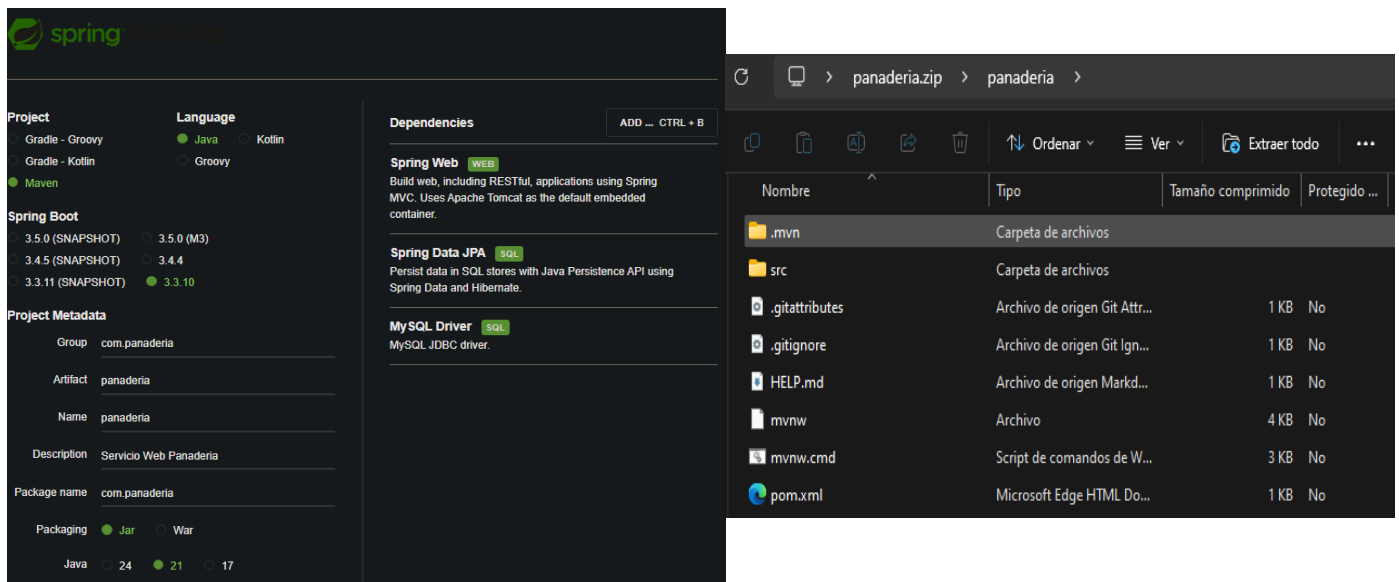


Figura 5. Estructura y creación del proyecto con Spring Boot.

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 6. 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.

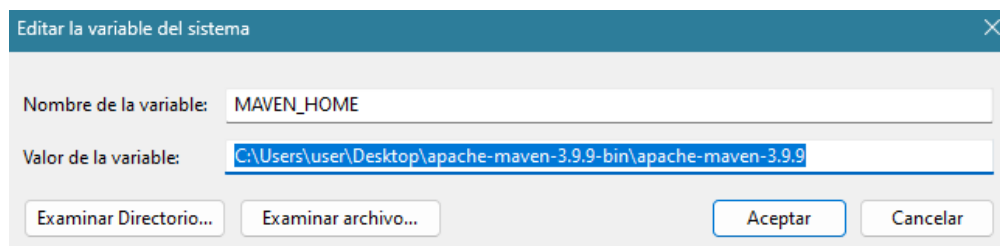


Figura 7. Configuración de la variable de entorno MAVEN_HOME.

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

```
%MAVEN_HOME%\bin
```

Figura 8. 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 (8e8579a9e76f7d015ee5ec7bfc9d97d260186937)
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 9. 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:

1. PanaderiaControlador.java

Este archivo contiene el controlador de la API REST en un proyecto Spring Boot. Un controlador en Spring Boot maneja las solicitudes HTTP y las mapea a métodos específicos, que a su vez procesan la solicitud y devuelven una respuesta. En este caso, el controlador de la panadería gestiona las solicitudes relacionadas con la compra de pan y la consulta del inventario.

En la siguiente figura podemos observar las librerías utilizadas y las anotaciones para este primer código:

```
package com.panaderia.controller;

import com.panaderia.model.CompraRequest; // Nueva clase para encapsular nombreCliente y compras
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import com.panaderia.service.PanaderiaServicio;
import com.panaderia.model.Pan;

import java.util.List;

@RestController
@RequestMapping("/panaderia")
@CrossOrigin(origins = "*")
```

Figura 10. Anotaciones.

- **@RestController:** Esta anotación indica que la clase es un controlador que responde a solicitudes HTTP, y que los métodos dentro de esta clase devuelven datos directamente (en lugar de vistas). Es una combinación de **@Controller** y **@ResponseBody**.
- **@RequestMapping("/panaderia"):** Esta anotación establece que todas las rutas dentro de este controlador comenzarán con `/panaderia`. Por ejemplo, si se llama a `/productos`, la URL completa será `/panaderia/productos`.
- **@CrossOrigin(origins = "*"):** Permite solicitudes de cualquier origen (en este caso, desde cualquier dominio). Es útil para permitir que el frontend, que podría estar en un dominio diferente, interactúe con el backend.

Posteriormente, se realiza la inyección de dependencias, como podemos ver en la siguiente figura:

```
@Autowired
private PanaderiaServicio panaderiaServicio;
```

Figura 11. Inyección de dependencias **@Autowired**.

- **@Autowired:** Spring gestiona la inyección de dependencias. Esta anotación asegura que **PanaderiaServicio** se inyecte automáticamente en el controlador. Es decir, Spring Boot proporciona una instancia de **PanaderiaServicio** para que podamos usar sus métodos en este controlador sin necesidad de crear una nueva instancia manualmente.

Después, se realizaron los métodos del controlador, en este caso tenemos:

Método 1: Obtener lista de productos disponibles

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

Figura 12. Método para obtener la lista de productos.

- `@GetMapping("/productos")`: Este método se invoca cuando se realiza una solicitud HTTP GET a `/panaderia/productos`. `@GetMapping` es una forma abreviada de usar `@RequestMapping` para las solicitudes GET.
- `public List<Pan> obtenerProductos()`: Este método llama al servicio `PanaderiaServicio` para obtener una lista de todos los productos disponibles (en este caso, panes). El servicio devuelve una lista de objetos `Pan`, que son los productos almacenados en la base de datos.

Método 2: Obtener stock de un producto específico

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

Figura 13. Método para obtener stock de un producto.

- `@GetMapping("/stock/{producto}")`: Este método maneja las solicitudes GET a `/panaderia/stock/{producto}`, donde `{producto}` es un parámetro dinámico que se obtiene de la URL.
- `@PathVariable String producto`: Este parámetro toma el valor de la parte dinámica de la URL (el nombre del producto) y lo pasa al método.
- `return panaderiaServicio.obtenerStock(producto);`: Llama al servicio `PanaderiaServicio` para obtener el stock del producto especificado y devuelve la cantidad disponible.

Método 3: Realizar una compra

```
// Realizar una compra
@PostMapping("/comprar")
public String comprarPan(@RequestBody CompraRequest request) {
    return panaderiaServicio.comprarPan(request.getNombreCliente(), request.getCompras());
}
```

Figura 14. Método para poder realizar compras.

- `@PostMapping("/comprar")`: Este método maneja las solicitudes HTTP POST a `/panaderia/comprar`. Se utiliza para procesar una compra de pan.

- **@RequestBody CompraRequest request:** La anotación **@RequestBody** indica que el cuerpo de la solicitud contiene un objeto JSON que se convertirá automáticamente en una instancia de **CompraRequest**. Esta clase encapsula el nombre del cliente y la lista de compras.
- **return panaderiaServicio.comprarPan(request.getNombreCliente(), request.getCompras());:** Este código llama al servicio **PanaderiaServicio** para realizar la compra, pasando el nombre del cliente y las compras solicitadas.

En general, este controlador define 3 rutas principales:

- **/productos:** Devuelve todos los productos disponibles en la panadería.
- **/stock/{producto}:** Devuelve el stock disponible de un producto específico.
- **/comprar:** Recibe los detalles de la compra (nombre del cliente y productos solicitados) y realiza la compra.

El controlador **PanaderiaControlador.java** es responsable de manejar las solicitudes HTTP relacionadas con los productos de la panadería, como obtener la lista de productos, verificar el stock y procesar las compras. Este controlador utiliza los servicios proporcionados por **PanaderiaServicio**, que implementan la lógica de negocio, y asegura que las operaciones se realicen correctamente, con la sincronización adecuada y la actualización del inventario.

2. PanaderiaServicio.java

Este archivo contiene la implementación de la lógica de negocio relacionada con las operaciones de la panadería, como la gestión del inventario, la realización de compras y la simulación del horneado de panes. Los métodos en este servicio interactúan con el repositorio de datos para obtener y actualizar los registros de productos, y también incluyen funcionalidades adicionales, como la programación de tareas. En la siguiente figura podemos observar las principales librerías y anotaciones utilizadas:

```
package com.panaderia.service;

import com.panaderia.model.Compra;
import com.panaderia.model.Pan;
import com.panaderia.repository.PanRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Service

// Método que simula el horneado de los panes cada 2 minutos
@Scheduled(fixedRate = 300000) // 300000 milisegundos = 5 minutos
```

Figura 15. Librerías y anotaciones utilizadas.

- **@Service:** Esta anotación marca la clase como un servicio de Spring. Los servicios son componentes de negocio que contienen la lógica de la aplicación. Al usar esta anotación,

Spring Boot puede gestionar el ciclo de vida de la clase y permitir la inyección de dependencias en otros componentes.

- **@Autowired**: Se utiliza para inyectar dependencias de manera automática. En este caso, se inyecta una instancia de **PanRepositorio**, que permite acceder a la base de datos para realizar operaciones sobre los panes.
- **@Scheduled(fixedRate = 300000)**: Esta anotación se usa para ejecutar el método `restablecerStock` de manera programada, en este caso, cada 5 minutos (300,000 milisegundos). Esto simula el horneado de panes, actualizando el stock de la panadería.

En las siguientes figuras tenemos los métodos del servicio, primeramente, vemos el método para obtener todos los productos:

```
// Obtener todos los productos
public List<Pan> obtenerProductos() {
    logger.info("Obteniendo lista completa de productos.");
    return panRepositorio.findAll();
}
```

Figura 16. Método para obtener los productos disponibles.

- **obtenerProductos()**: Este método devuelve todos los productos de la panadería al llamar al método `findAll` del repositorio **PanRepositorio**. Utiliza **logger.info** para registrar la acción en los logs.

Obtener el stock de un producto específico

```
// Obtener el stock de un producto específico
public int obtenerStock(String producto) {
    Pan pan = panRepositorio.findByProducto(producto);
    if (pan != null) {
        logger.info("Obteniendo stock de producto: '{}'. Stock disponible: {}", producto, pan.getStock());
        return pan.getStock();
    } else {
        logger.warn("Producto '{}' no encontrado en el inventario.", producto);
        return 0;
    }
}
```

Figura 16. Método para obtener el stock de un producto específico.

`obtenerStock(String producto)`: Este método busca un producto específico en la base de datos utilizando `findByProducto(producto)`. Si el producto se encuentra, devuelve la cantidad de stock disponible. Si no se encuentra, se registra un aviso en los logs y devuelve 0.

En la figura anterior podemos observar como se realiza la búsqueda del stock.

Realizar una compra

En la siguiente Figura podemos observar el método de como se realiza la compra de un producto, actualizando su stock y registrando las transacciones.

```

public String comprarPan(String nombreCliente, List<Compra> compras) {
    StringBuilder resumen = new StringBuilder();
    int totalCompra = 0;
    boolean compraExitosa = false;

    // Si no hay productos en el carrito, se devuelve un mensaje de error
    if (compras == null || compras.isEmpty()) {
        logger.warn("El cliente {} intentó realizar una compra sin productos seleccionados.", nombreCliente);
        return "No se han seleccionado productos para la compra.";
    }

    for (Compra compra : compras) {
        Pan pan = panRepositorio.findByProducto(compra.getProducto());

        if (pan != null) {
            int stockDisponible = pan.getStock();
            int cantidadSolicitada = compra.getCantidad();

            if (stockDisponible >= cantidadSolicitada) {
                // Si hay stock suficiente, se actualiza el stock
                int stockRestante = stockDisponible - cantidadSolicitada;
                pan.setStock(stockRestante);
                panRepositorio.save(pan);

                // Log de la compra realizada
                logger.info("Usuario: {} compró {} unidades de '{}'. Stock restante: {}",
                    nombreCliente, cantidadSolicitada, compra.getProducto(), stockRestante);

                // Añadir al resumen
                totalCompra += cantidadSolicitada;
                resumen.append(compra.getProducto())
                    .append(str: " ")
                    .append(cantidadSolicitada)
                    .append(str: " unidades compradas. Stock restante: ")
                    .append(stockRestante)
                    .append(str: "\n");

                compraExitosa = true;
            } else {
                // Si no hay suficiente stock
                logger.warn("Compra fallida para el cliente '{}'. Producto: '{}', cantidad solicitada: {}, stock disponible: {}",
                    nombreCliente, compra.getProducto(), cantidadSolicitada, stockDisponible);

                resumen.append(compra.getProducto())
                    .append(str: " Stock insuficiente o producto no encontrado.\n");
            }
        } else {
            // Producto no encontrado
            logger.error("Producto '{}' no encontrado en la base de datos para el cliente '{}'.", compra.getProducto(), nombreCliente);
            resumen.append(compra.getProducto())
                .append(str: " Producto no encontrado en el inventario.\n");
        }
    }

    // Resultado final de la compra
    if (compraExitosa) {
        resumen.append(str: "Compra exitosa. Total de productos comprados: ")
            .append(totalCompra);
    } else {
        resumen.append(str: "No se pudo realizar la compra. Verifique el stock o los productos seleccionados.");
    }

    // Log del resultado final de la compra
    logger.info("Compra realizada por {}. Total de productos comprados: {}", nombreCliente, totalCompra);
}

```

Figura 17. Método para realizar la compra de un producto.

- comprarPan(String nombreCliente, List<Compra> compras): Este es el método central para procesar una compra. Recibe el nombre del cliente y una lista de productos que el cliente desea comprar. Para cada producto, se valida si hay suficiente stock. Si el stock es suficiente, se actualiza el inventario; si no, se registra un error.

Lo siguiente, es el proceso para la simulación del horneado de panes (restablecer stock), el cual podemos ver en la siguiente Figura:

```

// Método que simula el horneado de los panes cada 2 minutos
@Scheduled(fixedRate = 300000) // 300000 milisegundos = 5 minutos
public void restablecerStock() {
    // Definir la cantidad de panes a hornear por tipo de pan
    Map<String, Integer> stockPorTipo = new HashMap<>();
    stockPorTipo.put(key:"Pan de avena", value:15);
    stockPorTipo.put(key:"Pan de trigo", value:20);
    stockPorTipo.put(key:"Pan de centeno", value:10);
    stockPorTipo.put(key:"Pan integral", value:25);
    // podemos añadir más tipos de panes según sea necesario
    stockPorTipo.put(key:"Pan de maíz", value:30);
    stockPorTipo.put(key:"Pan", value:10); // Pan común

    // Obtener todos los panes en inventario
    Iterable<Pan> panes = panRepositorio.findAll();

    for (Pan pan : panes) {
        Integer stockRestablecer = stockPorTipo.get(pan.getProducto());
        if (stockRestablecer != null) {
            // Restablecer el stock de cada pan según lo definido arriba
            pan.setStock(pan.getStock() + stockRestablecer); // Sumar el stock
            panRepositorio.save(pan); // Guardar el producto actualizado
            logger.info("Stock de {} restablecido. Nuevo stock: {}", pan.getProducto(), pan.getStock());
        }
    }
}

```

Figura 18. Método para restablecer el stock de panes de manera periodica.

- **restablecerStock():** Este método es ejecutado automáticamente cada 5 minutos gracias a la anotación `@Scheduled`. Simula el horneado de panes, restableciendo el stock de los diferentes tipos de pan según un valor predeterminado.

El servicio `PanaderiaServicio.java` contiene varios métodos importantes para la gestión del inventario y la realización de compras:

- **obtenerProductos():** Recupera todos los productos disponibles en la panadería.
- **obtenerStock(String producto):** Devuelve el stock de un producto específico.
- **comprarPan(String nombreCliente, List<Compra> compras):** Procesa la compra de un cliente, actualizando el stock y registrando la transacción.
- **restablecerStock():** Simula el horneado de panes, actualizando el stock de diferentes tipos de pan cada 5 minutos.

Este servicio juega un papel crucial en la gestión de la panadería, asegurando que las operaciones relacionadas con el stock y las compras se manejen correctamente. También proporciona una función programada que simula el proceso de producción de panes, restableciendo el stock de manera automática.

3. Compra.java

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 19. 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.

4. Pan.java

La clase Pan.java representa el modelo de los productos en el inventario de la panadería. Esta clase se utiliza para mapear los registros de productos en la base de datos, permitiendo realizar operaciones como la consulta de stock o la actualización de la cantidad disponible de un tipo de pan.

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

```

package com.panaderia.model;

import jakarta.persistence.*;

@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 20. Código de la clase Pan.java.

- **Anotaciones de JPA (Jakarta Persistence API):**
 - **@Entity:** Indica que esta clase es una entidad persistente que será mapeada a una tabla de la base de datos. En este caso, la entidad Pan corresponde a los productos en el inventario de la panadería.
 - **@Table(name = "inventario"):** Esta anotación especifica que los registros de la clase Pan estarán almacenados en la tabla inventario de la base de datos.
 - **@Id y @GeneratedValue(strategy = GenerationType.IDENTITY):** La anotación @Id marca el campo id como la clave primaria de la entidad. @GeneratedValue se usa para indicar que el valor de este campo se generará automáticamente cuando se inserte un nuevo producto en la base de datos, utilizando una estrategia de generación de identidad.
- **Atributos:**
 - **id:** Un identificador único para cada producto de pan en el inventario.
 - **producto:** El nombre del producto (por ejemplo, "Pan de avena", "Pan integral", etc.).
 - **stock:** La cantidad disponible de ese tipo de pan en el inventario.
- **Métodos:**
 - **getId(), getProducto(), getStock():** Métodos de acceso para obtener los valores de los atributos.
 - **setStock():** Método de acceso para establecer el valor de stock, utilizado cuando se realiza una compra y el inventario se actualiza.

La clase Pan.java es esencial para la gestión del inventario en la panadería, permitiendo el seguimiento de los productos disponibles y la actualización de su stock a medida que se realizan compras.

5. CompraRequest.java

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 21. 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.

En general, estas 3 clases **`Compra.java`**, **`Pan.java`** y **`CompraRequest.java`** forman una parte esencial de la arquitectura del sistema de la panadería. **`Compra.java`** define un único producto y su cantidad en la compra, **`Pan.java`** representa el producto en el inventario de la panadería y maneja la cantidad disponible, mientras que **`CompraRequest.java`** sirve como contenedor para los detalles completos de la compra, como el cliente y los productos solicitados. Juntas, estas clases facilitan la manipulación de las compras y el control del inventario, interactuando estrechamente con el resto del sistema para ofrecer una solución funcional de gestión de ventas y stock.

6. `PanRepositorio.java`

La clase `PanRepositorio.java` se encarga de manejar las operaciones relacionadas con la base de datos para la entidad `Pan`. Este repositorio es clave para la persistencia y recuperación de datos en el sistema de gestión de inventario de la panadería.

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

```
package com.panaderia.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.panaderia.model.Pan;

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

Figura 22. Código de la clase `PanRepositorio.java`.

- **`extends JpaRepository<Pan, Long>:`**
 - `JpaRepository` es una interfaz de Spring Data JPA que proporciona métodos CRUD (Crear, Leer, Actualizar, Eliminar) de manera automática. Al extender `JpaRepository`, `PanRepositorio` hereda estos métodos, lo que simplifica la interacción con la base de datos.
 - El tipo `Pan` es la entidad que estamos manejando, y `Long` es el tipo del identificador único de la entidad (es decir, el campo `id` en la clase `Pan`).
- **`findByProducto(String producto):`**
 - Este es un método personalizado que se utiliza para encontrar un `Pan` específico basado en el nombre del producto. Spring Data JPA genera automáticamente la implementación de este método debido a la convención de nomenclatura, en la que `findBy` seguido del nombre de un atributo (en este caso, `producto`) se traduce a una consulta de base de datos que busca un registro donde ese campo coincida con el valor proporcionado.

- Por ejemplo, si se busca un Pan con el nombre "Pan de avena", se devolverá el objeto Pan correspondiente con ese nombre de producto.

Función del repositorio en el sistema:

El repositorio PanRepository es fundamental para la comunicación entre la capa de servicio y la base de datos. Gracias a este repositorio, el servicio de la panadería puede realizar operaciones como:

- **Buscar un producto:** Usando métodos como findByProducto, el servicio puede consultar la base de datos para obtener información específica sobre un tipo de pan en el inventario.
- **Actualizar el stock:** Después de realizar una compra, el servicio puede actualizar el stock de un producto utilizando el método save() heredado de JpaRepository.

La interfaz PanRepository simplifica el acceso a los datos de la panadería al proporcionar métodos predefinidos para interactuar con la base de datos sin necesidad de escribir SQL manualmente, lo que mejora la eficiencia y reduce el código necesario para manejar las operaciones de la base de datos.

En general, el repositorio PanRepository.java es una parte integral de la capa de acceso a datos en el sistema de panadería. Aprovechando la potencia de Spring Data JPA, este repositorio proporciona métodos automáticos para interactuar con la base de datos y permite realizar operaciones eficientes sobre la entidad Pan. Esto facilita la gestión del inventario de la panadería y la implementación de funcionalidades como la consulta del stock y la actualización de la cantidad de productos disponibles.

7. application.properties

Respecto a estos archivos, ya vienen predefinidos y creados en la carpeta obtenida durante la creación del proyecto spring boot. En este caso, el archivo application.properties en un proyecto Spring Boot se utiliza para configurar varias propiedades del sistema, tales como la conexión a la base de datos, el comportamiento de las transacciones, los niveles de logging, y muchas otras configuraciones necesarias para que la aplicación funcione correctamente.

A continuación, se muestran las configuraciones que se encuentran en este archivo específico application.properties para la panadería:

```
# Mostrar los logs en la terminal
logging.level.com.panaderia.service.PanaderiaServicio=INFO
spring.application.name=panaderia
spring.datasource.url=jdbc:mysql://localhost:3306/panaderia?useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
server.port=8081
```

Figura 23. Configuración de application.properties.

Desglose de las configuraciones:

1. logging.level.com.panaderia.service.PanaderiaServicio=INFO:

- Esta propiedad establece el nivel de logs para la clase **PanaderiaServicio** en **INFO**, lo que significa que los mensajes de log de nivel **INFO** y superior (como **WARN** y **ERROR**) serán mostrados en la terminal cuando se ejecute la aplicación.
- Es útil para ver información detallada durante la ejecución del servicio de la panadería, como los procesos de compra, stock y otros eventos relevantes.

2. **spring.application.name=panaderia:**

- Define el nombre de la aplicación, en este caso, **panaderia**. Este valor es útil en entornos de producción o cuando se maneja múltiples aplicaciones, ya que permite identificar de forma clara cuál es el nombre del proyecto.

3. **Configuración de la base de datos:**

- **spring.datasource.url=jdbc:mysql://localhost:3306/panaderia?useSSL=false&serverTimezone=UTC:**
 - Especifica la URL de conexión a la base de datos **MySQL**, donde **localhost:3306** es la dirección del servidor de base de datos, **panaderia** es el nombre de la base de datos, y los parámetros adicionales **useSSL=false** y **serverTimezone=UTC** son configuraciones necesarias para evitar problemas de SSL y para establecer la zona horaria del servidor.
- **spring.datasource.username=root:**
 - Define el nombre de usuario para la conexión a la base de datos. En este caso, se está utilizando **root**.
- **spring.datasource.password=root:**
 - Especifica la contraseña para el acceso a la base de datos. En este caso, también es **root**.

4. **Configuración de Hibernate y JPA:**

- **spring.jpa.hibernate.ddl-auto=update:**
 - Esta propiedad controla el comportamiento de Hibernate respecto al esquema de la base de datos. **update** indica que Hibernate debe actualizar automáticamente el esquema de la base de datos si detecta cambios en las entidades del modelo. Es útil en entornos de desarrollo, pero no se recomienda para producción.
- **spring.jpa.show-sql=true:**
 - Habilita la visualización de las consultas SQL que Hibernate genera al interactuar con la base de datos. Esto puede ser útil para depurar o entender cómo se realizan las operaciones de base de datos en la aplicación.

5. **server.port=8081:**

- Configura el puerto en el que la aplicación Spring Boot se ejecutará. En este caso, el servidor estará escuchando en el puerto **8081**. Este valor obviamente puede modificarse según sea necesario.

En general, este archivo **application.properties** proporciona las configuraciones esenciales para el funcionamiento de la aplicación. Desde la gestión del logging hasta la configuración de la base de datos y la conexión con el servidor, este archivo centraliza las propiedades que controlan el comportamiento de la aplicación **Spring Boot**. En este caso, se ha configurado el nivel de logging para obtener visibilidad sobre el servicio de panadería, se ha establecido la conexión con la base de datos **MySQL** y se ha configurado el puerto para el servidor, asegurando que la aplicación funcione correctamente en su entorno de ejecución.

8. PanaderiaApplication.java y pom.xml

El archivo PanaderiaApplication.java es el punto de entrada de la aplicación en Spring Boot. Este archivo contiene el método main que lanza la aplicación Spring Boot. Además, se le han agregado algunas configuraciones para habilitar características específicas de Spring.

```
package com.panaderia;

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

@SpringBootApplication
@EnableScheduling // Habilitar la programación de tareas
public class PanaderiaApplication {

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

Figura 24. Código del punto de entrada de la aplicación .

- **@SpringBootApplication:**
 - Esta anotación es clave en cualquier aplicación Spring Boot, ya que combina varias anotaciones importantes: **@Configuration**, **@EnableAutoConfiguration**, y **@ComponentScan**. Juntas, estas anotaciones le indican a Spring que debe buscar configuraciones de la aplicación, habilitar la configuración automática y escanear los componentes en el paquete de la aplicación.
- **@EnableScheduling:**
 - Habilita la programación de tareas dentro de la aplicación. Esto permite que métodos anotados con **@Scheduled** (como el de restablecimiento de stock de la panadería) se ejecuten automáticamente en intervalos definidos. En el código de **PanaderiaServicio.java**, el método **restablecerStock()** usa esta funcionalidad para simular el horneado de pan cada 5 minutos.
- **public static void main(String[] args):**

- Este es el punto de inicio de la aplicación. Al ejecutar este método, Spring Boot inicia la aplicación y configura todo lo necesario para que funcione.

Este archivo es el encargado de arrancar la aplicación Spring Boot y configurar el entorno necesario para que los componentes y servicios de la panadería funcionen correctamente. La integración de **@EnableScheduling** permite que las tareas programadas se ejecuten de manera automática, asegurando la simulación del proceso de horneado de panes en intervalos regulares.

De igual forma se genera el archivo `pom.xml`, el cual es el archivo de configuración de Maven, el sistema de gestión de proyectos que utiliza Spring Boot para manejar dependencias, construcción del proyecto y otros aspectos relacionados.

Dicho archivo **pom.xml** es esencial para gestionar las dependencias y configuraciones del proyecto. En este caso, incluye las dependencias de Spring Boot necesarias para la creación de servicios web y el acceso a bases de datos, así como la configuración para ejecutar la aplicación en un entorno de desarrollo. Gracias a Maven, el proyecto puede manejar automáticamente las dependencias y facilitar el proceso de construcción y empaquetado.

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.

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 del proyecto. El enfoque de esta explicación se ha centrado en los archivos y clases que tienen un impacto directo en la funcionalidad principal de la aplicación de panadería.

Desarrollo del Frontend

Para complementar el servicio web desarrollado en Spring Boot, se implementó un frontend que permite a los usuarios interactuar con el sistema de panadería de manera sencilla y amigable. Esta interfaz fue desarrollada utilizando HTML, CSS y JavaScript, asegurando que los clientes puedan visualizar los productos disponibles, agregar productos al carrito y finalizar su compra de manera intuitiva.

El frontend se diseñó con el objetivo de ofrecer una experiencia de usuario clara y funcional, facilitando la comunicación con el servicio REST a través de peticiones HTTP (GET y POST) para obtener la lista de productos y registrar compras.

A continuación, se detallará el proceso de desarrollo del frontend, incluyendo la estructura de la interfaz, la implementación de estilos y la lógica de interacción con la API.

1. Index.html

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 25. 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 26. 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.

2. style.css

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.

3. script.js

El archivo script.js es el encargado de manejar la interacción entre el usuario y el sistema. Su función principal es comunicarse con la API del backend para obtener el stock de productos y gestionar las compras.

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

Definición de la URL de la API

```
const API_URL = "http://localhost:8081/panaderia";
```

Figura 27. Definición de la URL.

Esta constante almacena la URL base de la API REST del backend. Se utiliza en las solicitudes para obtener productos y realizar compras.

Función obtenerProductos()

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

- **fetch(\${API_URL}/productos)** → Realiza una solicitud GET al endpoint /productos del backend.
- **Si la respuesta es exitosa (response.ok)** → Convierte la respuesta a JSON y la pasa a la función mostrarProductos().

- **Si ocurre un error o el servidor no responde** → Muestra un mensaje en la interfaz informando al usuario.

En la siguiente Figura podemos observar lo mencionado anteriormente:

```
// Obtener el stock disponible y la lista de productos
async function obtenerProductos() {
  try {
    const response = await fetch(`${API_URL}/productos`);
    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 28. Función para obtener la lista de productos y su stock disponible.

Función mostrarProductos(productos)

Esta función muestra los productos obtenidos del backend en la interfaz.

1. Se obtiene el contenedor #productos del HTML.
2. Se vacía el contenido previo para evitar duplicados.
3. Se recorre la lista de productos y se genera dinámicamente un bloque de HTML por cada producto, que incluye:
 - Su nombre.
 - El stock disponible.
 - Un campo numérico para que el usuario ingrese la cantidad a comprar.
4. Cada producto se añade al contenedor #productos.

En la siguiente figura observamos el bloque de código de dicha función:

```
// Mostrar los productos disponibles en la interfaz
function mostrarProductos(productos) {
  const productosDiv = document.getElementById("productos");
  productosDiv.innerHTML = ""; // Limpiar la lista actual

  productos.forEach((producto) => {
    const productoDiv = document.createElement("div");
    productoDiv.innerHTML = `
      <label>
        ${producto.producto} - Stock: <span id="stock-${producto.id}">${producto.stock}</span>
        <input type="number" id="cantidad-${producto.id}" placeholder="Cantidad" min="1" max="${producto.stock}">
      </label>
      <br>
    `;
    productosDiv.appendChild(productoDiv);
  });
}
```

Figura 29. Función para mostrar los productos del backend en la interfaz.

Función comprarPan()

Esta función permite a los usuarios realizar una compra.

1. **Validación del nombre del cliente:** Si no ingresa su nombre, se muestra un mensaje de error.
2. **Recopilación de productos:**
 - Se recorre la lista de productos disponibles.
 - Se obtiene la cantidad ingresada por el usuario para cada producto.
 - Si la cantidad es mayor a 0, se guarda en un arreglo llamado compras.
3. **Validación de selección:** Si el usuario no ha seleccionado ningún producto, se muestra un mensaje de advertencia.
4. **Se construye un objeto data** con el nombre del cliente y la lista de compras para enviarlo al backend.

```
// 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,
  };
}
```

Figura 30. Función para que el usuario pueda realizar una compra.

Envío de la solicitud de compra al backend

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

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

  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();
}
```

Figura 31. Interacción entre el frontend y backend al realizar una compra.

- Se envía la solicitud POST al backend con la información de la compra en formato JSON.
- Si la compra se realiza con éxito, el mensaje de respuesta del servidor se muestra en la interfaz.
- Si ocurre un error, se informa al usuario.
- Se actualiza el stock de productos en la interfaz llamando a `obtenerProductos()`.

Posteriormente, al cargar la página se ejecuta `obtenerProductos()`, lo que permite mostrar los productos disponibles desde el inicio.

En general, el archivo `script.js` cumple un papel fundamental en la interacción del usuario con el sistema, permitiendo obtener productos y realizar compras de manera dinámica. Su correcta implementación garantiza una experiencia fluida y eficiente para los clientes de la panadería.

Con esta implementación, el servicio web de la panadería queda completamente desarrollado y funcional. El backend gestiona el inventario, procesa las compras y actualiza el stock de manera automática, mientras que el frontend permite a los usuarios visualizar los productos disponibles y realizar pedidos de forma intuitiva. Gracias a la comunicación entre ambos componentes a través de la API REST, el sistema garantiza una experiencia fluida y eficiente.

Instrucciones para ejecutar el proyecto

Para poder ejecutar este proyecto, es necesario abrir una terminal y ubicarse dentro de la carpeta donde se encuentran todos los archivos del backend. En mi caso, la ruta del proyecto es: **C:\Users\user\Desktop\panaderia**

1. Compilación y construcción del proyecto

El primer paso es compilar y construir el proyecto con Maven. Para ello, ejecutamos el siguiente comando en la terminal:

```
PS C:\Users\user\Desktop\panaderia> mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.panaderia:panaderia >-----
[INFO] Building panaderia 0.0.1-SNAPSHOT
[INFO]    from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- clean:3.3.2:clean (default-clean) @ panaderia ---
[INFO] Deleting C:\Users\user\Desktop\panaderia\target
```

Este comando se encarga de limpiar cualquier compilación previa, descargar las dependencias necesarias y generar el paquete del proyecto. Es un paso fundamental para asegurarnos de que el código está correctamente preparado para ejecutarse.

```
m\panaderia\panaderia\0.0.1-SNAPSHOT\panaderia-0.0.1-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 20.407 s
[INFO] Finished at: 2025-04-02T01:14:24-06:00
[INFO] -----
PS C:\Users\user\Desktop\panaderia>
```

2. Ejecución del servicio backend

Si la compilación se ha realizado sin errores, podemos proceder a ejecutar el servicio backend con el siguiente comando:

```
PS C:\Users\user\Desktop\panaderia> mvn spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.panaderia:panaderia >-----
[INFO] Building panaderia 0.0.1-SNAPSHOT
[INFO]    from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot:3.3.10:run (default-cli) > test-compile @ panaderia >>>
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ panaderia ---
```

Este comando inicia el servidor en el puerto 8081 (según lo configurado en application.properties). Una vez iniciado, el servicio estará disponible y listo para recibir peticiones HTTP.

```

2025-04-02T01:16:02.575-06:00 INFO 20012 --- [panaderia] [      main] j.LocalContainerEntityManagerFactoryBean
: Initialized JPA EntityManagerFactory for persistence unit 'default'
2025-04-02T01:16:03.066-06:00 WARN 20012 --- [panaderia] [      main] JpaBaseConfiguration$JpaWebConfiguration
: spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering
. Explicitly configure spring.jpa.open-in-view to disable this warning
2025-04-02T01:16:03.495-06:00 INFO 20012 --- [panaderia] [      main] o.s.b.w.embedded.tomcat.TomcatWebServer
: Tomcat started on port 8081 (http) with context path '/'
2025-04-02T01:16:03.516-06:00 INFO 20012 --- [panaderia] [      main] com.panaderia.PanaderiaApplication
: Started PanaderiaApplication in 6.091 seconds (process running for 6.677)

```

3. Pruebas con Postman o navegador

Una vez que el backend está en ejecución, podemos probar su funcionamiento de dos maneras:

- **Usando Postman:** Enviando solicitudes GET y POST a los endpoints de la API para verificar el funcionamiento del sistema.
- **Usando el frontend:** Para ello, es necesario abrir el archivo index.html.

4. Ejecución del Frontend

Para interactuar con el sistema de manera gráfica, debemos ejecutar el frontend. El archivo index.html se encuentra en la siguiente ruta: **C:\Users\user\Desktop\panaderia\panaderia_web**

Basta con hacer doble clic en el archivo index.html o abrirlo en un navegador web. Desde esta interfaz, podremos visualizar los productos disponibles, realizar compras y ver la actualización del stock en tiempo real.

Una vez abierto el frontend, el usuario puede seleccionar productos, indicar la cantidad deseada, ingresar su nombre y realizar la compra. Si la compra se procesa correctamente, el stock se actualizará y se mostrará un mensaje de confirmación. De esta manera, el sistema estará completamente operativo, permitiendo la gestión de productos en la panadería de manera distribuida.

The screenshot shows a web browser displaying the '¡BIENVENIDO A LA PANADERÍA DEL BARRIO!' page. The page has a light purple header with the text '¡El mejor pan recién horneado, directo a tu hogar!'. Below the header, there is a section titled 'Selecciona tus panes favoritos:' containing a list of bread products with their stock and a 'Cant' input field for quantity:

- Pan - Stock: 207 Cant
- Pan de trigo - Stock: 188 Cant
- Pan de centeno - Stock: 96 Cant
- Pan integral - Stock: 247 Cant
- Pan de avena - Stock: 130 Cant
- Pan de maíz - Stock: 292 Cant
- Donas - Stock: 13 Cant

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

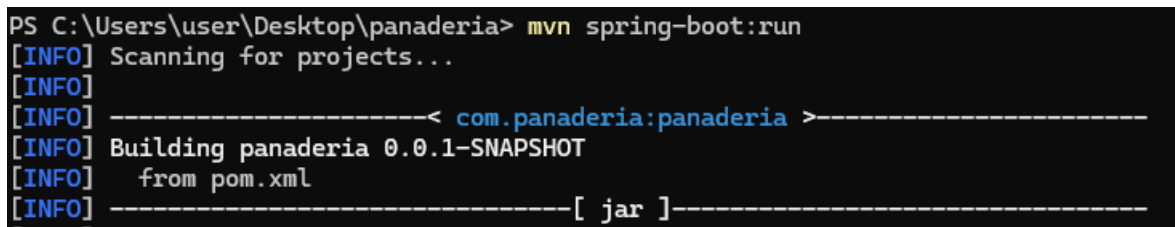
Resultados

El sistema web de panadería distribuido desarrollado con Spring Boot y MySQL ha demostrado un funcionamiento correcto y estable, permitiendo la gestión eficiente del inventario y la compra de productos de manera distribuida. A través de la implementación de una API REST, los clientes pueden interactuar con el sistema en tiempo real, consultar la disponibilidad de productos y realizar compras sin inconvenientes.

Durante las pruebas realizadas, se verificó que el sistema maneja correctamente las solicitudes de múltiples usuarios de manera concurrente. Cada compra realizada impacta directamente en la base de datos, actualizando el stock de manera inmediata y reflejando los cambios tanto en el backend como en la interfaz gráfica del frontend.

Asimismo, la tarea programada de reabastecimiento simulado del pan opera adecuadamente, asegurando que el inventario se incremente en los intervalos definidos. Se comprobó que el sistema es capaz de gestionar compras incluso cuando el stock es insuficiente, notificando al usuario sobre la falta de productos y evitando inconsistencias en la base de datos.

El servidor, ejecutado en el puerto 8081, registra cada solicitud recibida, mostrando mensajes en la terminal sobre la actividad del sistema. A continuación, se presentan capturas de pantalla de las pruebas realizadas para evidenciar su correcto funcionamiento, tanto con Postman como con el frontend.



```
PS C:\Users\user\Desktop\panaderia> mvn spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.panaderia:panaderia >-----
[INFO] Building panaderia 0.0.1-SNAPSHOT
[INFO]    from pom.xml
[INFO] -----[ jar ]-----
```

Figura 32. Ejecución del proyecto.

Para verificar el correcto funcionamiento del sistema, se realizaron pruebas con Postman utilizando los métodos **GET** y **POST** para interactuar con la API REST del backend.

En primer lugar, se probó el método **GET** para obtener la lista de productos disponibles en la panadería, mostrando su nombre y el stock actual. Para ello, se realizó una solicitud a la URL **http://localhost:8081/panaderia/productos**, obteniendo una respuesta en formato JSON con el listado de productos. Esta respuesta contenía información precisa sobre los productos y su stock, confirmando que el sistema responde correctamente y que los datos reflejan el estado actual de la base de datos.

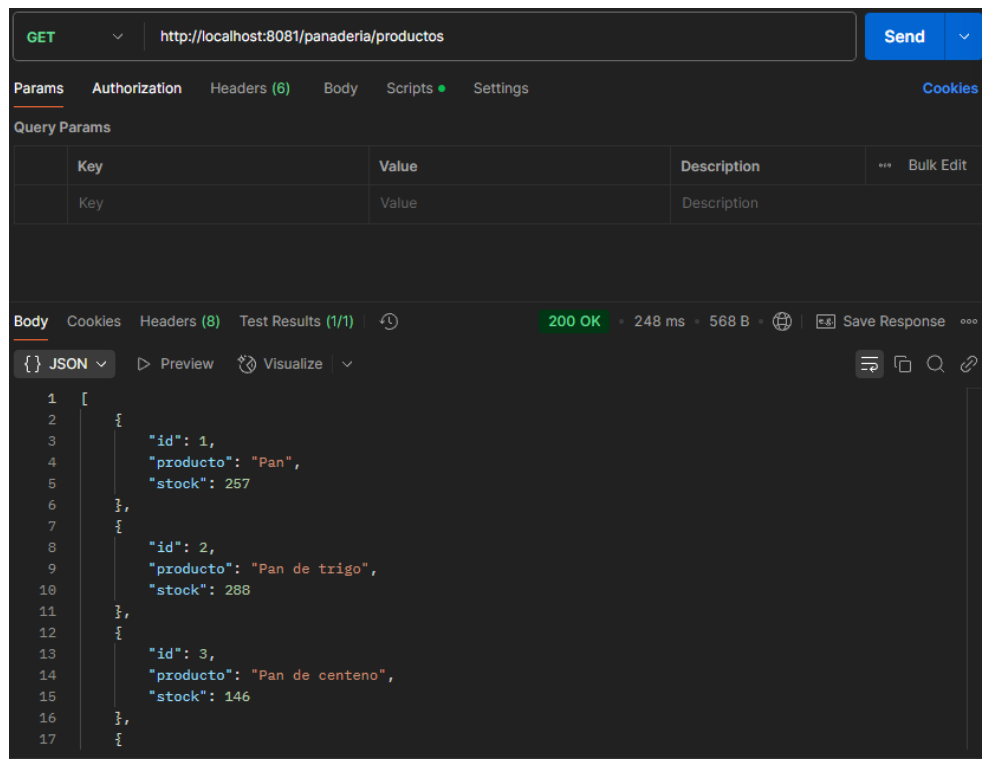


Figura 33. Prueba del método GET, para obtener el listado de productos.

Posteriormente, se probó el método **POST**, el cual permite realizar una compra enviando un nombre de cliente junto con la cantidad de productos deseados. Se envió una solicitud a `http://localhost:8081/panaderia/comprar` con un cuerpo en formato JSON que contenía el nombre del comprador y los productos seleccionados. La respuesta del servidor confirmó que la compra se realizó con éxito, y se verificó que el sistema descuenta correctamente la cantidad comprada del stock.

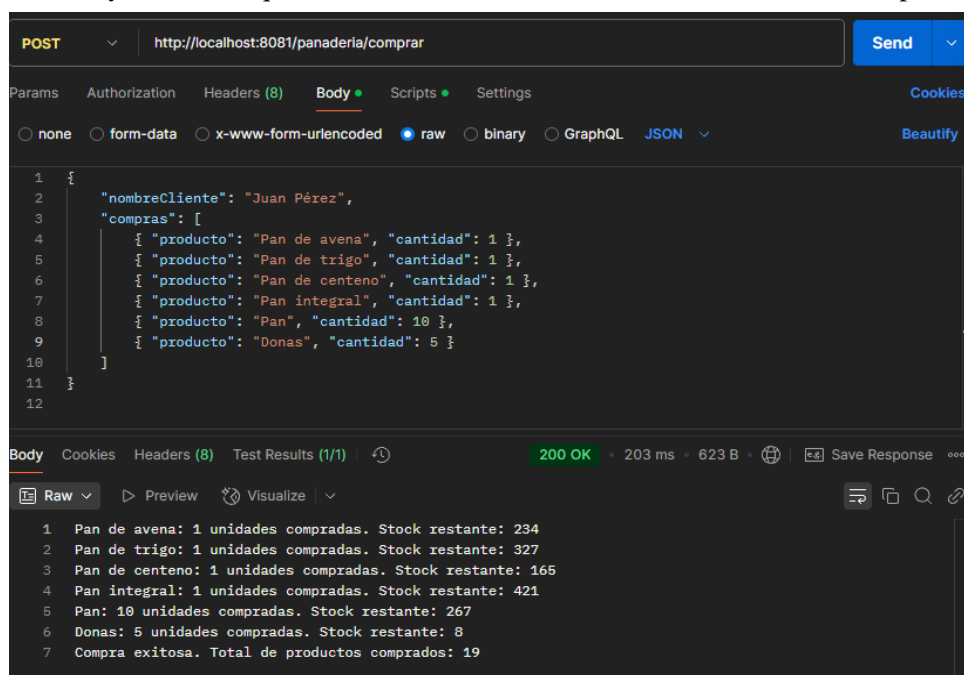


Figura 34. Prueba del método POST, para realizar una compra.

Y en el servidor obtenemos la siguiente respuesta o log.

```
2025-04-02T18:52:28.897-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio : Usuario: Juan P|@rez compr| 1 unidades de 'Pan de avena'. Stock restante: 234
Hibernate: select pl_0.id,pl_0.producto,pl_0.stock from inventario pl_0 where pl_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T18:52:28.903-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio : Usuario: Juan P|@rez compr| 1 unidades de 'Pan de trigo'. Stock restante: 327
Hibernate: select pl_0.id,pl_0.producto,pl_0.stock from inventario pl_0 where pl_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T18:52:28.930-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio : Usuario: Juan P|@rez compr| 1 unidades de 'Pan de centeno'. Stock restante: 165
Hibernate: select pl_0.id,pl_0.producto,pl_0.stock from inventario pl_0 where pl_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T18:52:28.944-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio : Usuario: Juan P|@rez compr| 1 unidades de 'Pan integral'. Stock restante: 421
Hibernate: select pl_0.id,pl_0.producto,pl_0.stock from inventario pl_0 where pl_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T18:52:28.957-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio : Usuario: Juan P|@rez compr| 10 unidades de 'Pan'. Stock restante: 267
Hibernate: select pl_0.id,pl_0.producto,pl_0.stock from inventario pl_0 where pl_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T18:52:28.971-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio : Usuario: Juan P|@rez compr| 5 unidades de 'Donas'. Stock restante: 8
2025-04-02T18:52:28.972-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio : Compra realizada por Juan P|@rez. Total de productos comprados: 19
```

Figura 35. Respuesta del servidor al cliente realizar una compra.

Además, se comprobó que, si la cantidad solicitada superaba el stock disponible, el sistema devolvía un mensaje indicando la falta de productos, evitando así transacciones inválidas.

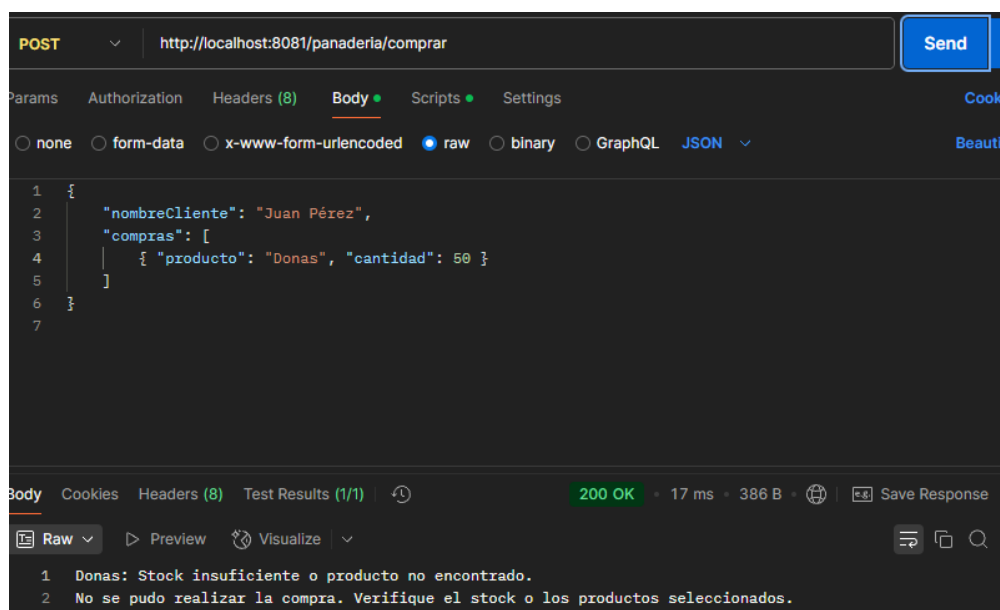


Figura 36. Prueba al comprar más del stock disponible.

Mensaje del servidor recibido al intentar realizar una compra donde el stock es menor a la cantidad solicitada por el cliente.

```
2025-04-02T19:11:20.261-06:00 WARN 25828 --- [panaderia] [nio-8081-exec-3] c.panaderia.service.PanaderiaServicio : Compra fallida para el cliente 'Juan P|@rez'. Producto: 'Donas', cantidad solicitada: 50, stock disponible: 8
2025-04-02T19:11:20.261-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-3] c.panaderia.service.PanaderiaServicio : Compra realizada por Juan P|@rez. Total de productos comprados: 0
```

Figura 37. Mensaje en el servidor de la compra fallida.

Finalmente, se realizó una nueva solicitud **GET** después de la compra para verificar que el stock se haya actualizado correctamente. Se constató que los valores reflejaban los cambios aplicados, asegurando que el sistema mantiene una gestión precisa del inventario. Aunque en este caso, los valores del stock aumentaron debido a la simulación del horno. Para ello, cada 5 minutos se simula que se hornearon nuevos panes y se actualiza el stock, además de recibir un mensaje en el servidor notificando dicha actualización.

```

2025-04-02T19:12:04.064-06:00 INFO 25828 --- [panaderia] [ scheduling-1] c.panaderia.service.PanaderiaServicio
: Stock de Pan restablecido. Nuevo stock: 307
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.id=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:12:04.076-06:00 INFO 25828 --- [panaderia] [ scheduling-1] c.panaderia.service.PanaderiaServicio
: Stock de Pan de trigo restablecido. Nuevo stock: 407
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.id=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:12:04.089-06:00 INFO 25828 --- [panaderia] [ scheduling-1] c.panaderia.service.PanaderiaServicio
: Stock de Pan de centeno restablecido. Nuevo stock: 205
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.id=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:12:04.101-06:00 INFO 25828 --- [panaderia] [ scheduling-1] c.panaderia.service.PanaderiaServicio
: Stock de Pan integral restablecido. Nuevo stock: 521
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.id=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:12:04.114-06:00 INFO 25828 --- [panaderia] [ scheduling-1] c.panaderia.service.PanaderiaServicio
: Stock de Pan de avena restablecido. Nuevo stock: 294
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.id=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:12:04.126-06:00 INFO 25828 --- [panaderia] [ scheduling-1] c.panaderia.service.PanaderiaServicio
: Stock de Pan de ma|iz restablecido. Nuevo stock: 622

```

Figura 38. Mensaje en el servidor de la actualización de stock, cada 5 min.

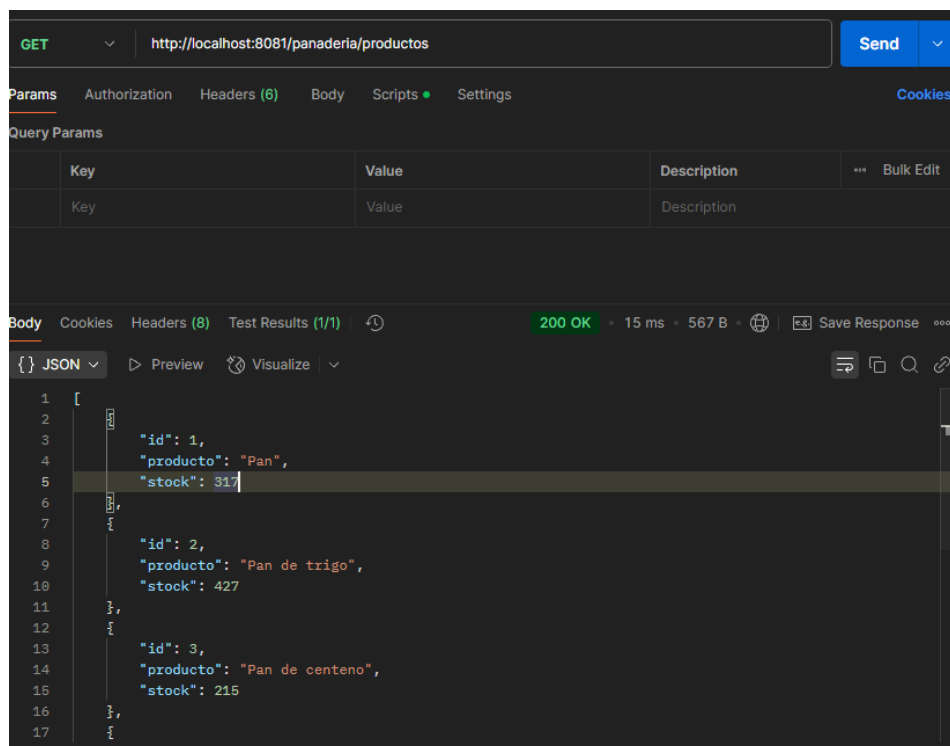


Figura 39. Actualización del stock, tras la simulación del horneado de nuevos panes.

Las pruebas realizadas con Postman confirmaron que la API REST de la panadería funciona correctamente. El sistema responde de manera adecuada a las solicitudes GET y POST, actualizando el inventario de forma precisa y notificando al usuario en caso de errores. Con esto, se valida que el servicio web desarrollado es completamente funcional y está listo para ser utilizado en un entorno distribuido.

Ahora, para poder comprobar el funcionamiento del **frontend**, se realizaron pruebas ejecutando el archivo index.html, el cual permite interactuar con el sistema de panadería a través de una interfaz gráfica sencilla y amigable para el usuario.

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 `http://localhost:8081/panaderia/productos`, cargando dinámicamente los productos en la interfaz.

¡BIENVENIDO A LA PANADERÍA DEL BARRIO!
¡El mejor pan recién horneado, directo a tu hogar!

Selecciona tus panes favoritos:

Pan - Stock: 327

Pan de trigo - Stock: 447

Pan de centeno - Stock: 225

Pan integral - Stock: 571

Pan de avena - Stock: 324

Pan de maíz - Stock: 682

Donas - Stock: 8

Datos de Compra:

Figura 40. 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. Al presionar el botón "Realizar Compra", se envió una solicitud **POST** al backend con los datos de la compra. Y como la compra fue valida, se mostró un mensaje de confirmación en pantalla y el stock de los productos se actualizó automáticamente.

Selecciona tus panes favoritos:

Pan - Stock: 300	<input type="text" value="Cant"/>
Pan de trigo - Stock: 400	<input type="text" value="Cant"/>
Pan de centeno - Stock: 200	<input type="text" value="Cant"/>
Pan integral - Stock: 500	<input type="text" value="Cant"/>
Pan de avena - Stock: 300	<input type="text" value="Cant"/>
Pan de maíz - Stock: 600	<input type="text" value="Cant"/>
Donas - Stock: 0	<input type="text" value="Cant"/>

Datos de Compra:

<input type="text" value="Elena"/>	<input type="button" value="Realizar Compra"/>
------------------------------------	--

Pan: 27 unidades compradas. Stock restante: 300
Pan de trigo: 47 unidades compradas. Stock restante: 400
Pan de centeno: 25 unidades compradas. Stock restante: 200
Pan integral: 71 unidades compradas. Stock restante: 500
Pan de avena: 24 unidades compradas. Stock restante: 300
Pan de maíz: 82 unidades compradas. Stock restante: 600
Donas: 8 unidades compradas. Stock restante: 0
Compra exitosa. Total de productos comprados: 284

Figura 41. Realización de una compra desde el sitio web.

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

```
2025-04-02T19:25:35.902-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-4] c.panaderia.service.PanaderiaServicio
: Usuario: Elena compr| 27 unidades de 'Pan'. Stock restante: 300
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:25:35.918-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-4] c.panaderia.service.PanaderiaServicio
: Usuario: Elena compr| 47 unidades de 'Pan de trigo'. Stock restante: 400
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:25:35.930-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-4] c.panaderia.service.PanaderiaServicio
: Usuario: Elena compr| 25 unidades de 'Pan de centeno'. Stock restante: 200
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:25:35.942-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-4] c.panaderia.service.PanaderiaServicio
: Usuario: Elena compr| 71 unidades de 'Pan integral'. Stock restante: 500
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:25:35.955-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-4] c.panaderia.service.PanaderiaServicio
: Usuario: Elena compr| 24 unidades de 'Pan de avena'. Stock restante: 300
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:25:35.974-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-4] c.panaderia.service.PanaderiaServicio
: Usuario: Elena compr| 82 unidades de 'Pan de maíz'. Stock restante: 600
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T19:25:35.988-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-4] c.panaderia.service.PanaderiaServicio
: Usuario: Elena compr| 8 unidades de 'Donas'. Stock restante: 0
```

Figura 42. 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: 310	<input type="text" value="Cant"/>
Pan de trigo - Stock: 420	<input type="text" value="Cant"/>
Pan de centeno - Stock: 210	<input type="text" value="Cant"/>
Pan integral - Stock: 525	<input type="text" value="Cant"/>
Pan de avena - Stock: 315	<input type="text" value="Cant"/>
Pan de maíz - Stock: 630	<input type="text" value="Cant"/>
Donas - Stock: 0	<input type="text" value="Cant"/>

Datos de Compra:

<input type="text" value="Elena"/>	<input type="button" value="Realizar Compra"/>
------------------------------------	--

Pan de avena: Stock insuficiente o producto no encontrado.
 Pan de maíz: Stock insuficiente o producto no encontrado.
 Donas: Stock insuficiente o producto no encontrado.
 No se pudo realizar la compra. Verifique el stock o los productos seleccionados.

Figura 43. Intento de compra mayor al stock disponible.

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

```
2025-04-02T19:29:30.172-06:00 WARN 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio
: Compra fallida para el cliente 'Elena'. Producto: 'Pan de avena', cantidad solicitada: 400, stock disponible: 31
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
2025-04-02T19:29:30.175-06:00 WARN 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio
: Compra fallida para el cliente 'Elena'. Producto: 'Pan de ma|jz', cantidad solicitada: 700, stock disponible: 63
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
2025-04-02T19:29:30.177-06:00 WARN 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio
: Compra fallida para el cliente 'Elena'. Producto: 'Donas', cantidad solicitada: 55, stock disponible: 0
2025-04-02T19:29:30.177-06:00 INFO 25828 --- [panaderia] [nio-8081-exec-8] c.panaderia.service.PanaderiaServicio
```

Figura 44. 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 40 y 41. También se observó que, pasados algunos minutos, los valores del stock aumentaban debido a la simulación del horneado automático, el cual ocurre cada 5 minutos. Este comportamiento fue validado revisando los mensajes en la consola del servidor, donde se registraba cada reposición de panes.

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: 320	<input type="text" value="Cant"/>
Pan de trigo - Stock: 440	<input type="text" value="Cant"/>
Pan de centeno - Stock: 220	<input type="text" value="Cant"/>
Pan integral - Stock: 550	<input type="text" value="Cant"/>
Pan de avena - Stock: 330	<input type="text" value="Cant"/>
Pan de maíz - Stock: 660	<input type="text" value="Cant"/>
Donas - Stock: 0	<input type="text" value="Cant"/>

Datos de Compra:

<input type="text" value="Elena"/>	<input type="button" value="Realizar Compra"/>
------------------------------------	--

Por favor, selecciona al menos un producto.

Figura 45. Solicitud al usuario de que seleccione al menos un producto.

Selecciona tus panes favoritos:

Pan - Stock: 320	<input type="text" value="Cant"/>
Pan de trigo - Stock: 440	<input type="text" value="2"/>
Pan de centeno - Stock: 220	<input type="text" value="3"/>
Pan integral - Stock: 550	<input type="text" value="21"/>
Pan de avena - Stock: 330	<input type="text" value="Cant"/>
Pan de maíz - Stock: 660	<input type="text" value="121"/>
Donas - Stock: 0	<input type="text" value="121"/>

Datos de Compra:

<input type="text" value="Ingresa tu nombre"/>	<input type="button" value="Realizar Compra"/>
--	--

Por favor, ingresa tu nombre.

Figura 46. Solicitud al usuario para que ingrese su nombre al realizar una compra.

En general, las pruebas demostraron que el frontend funciona correctamente, logrando una comunicación fluida con el backend y ofreciendo una experiencia de usuario intuitiva. La interfaz permite visualizar los productos disponibles, realizar compras y observar los cambios en el stock en tiempo real, cumpliendo con los objetivos planteados en la solución propuesta.

Y como se pudo observar, los resultados obtenidos durante las pruebas demuestran que el sistema funciona correctamente, permitiendo gestionar el stock de productos de panadería, realizar compras y actualizar el inventario en tiempo real. La simulación del horneado automático opera sin

inconvenientes, y tanto el backend como el frontend logran una comunicación efectiva. Esto valida la implementación de la solución propuesta y confirma el correcto funcionamiento del servicio web desarrollado

De último momento, se realizó una modificación en el proyecto para que fuera accesible desde varios dispositivos, no solo de manera local con localhost. Se añadieron los archivos index.html, style.css y script.js a la carpeta de recursos del proyecto ubicada en la ruta:

C:\Users\user\Desktop\panaderia\src\main\resources. Además, se actualizó el archivo script.js modificando la URL de la API de la siguiente manera:

```
const SERVER_IP = "192.168.100.12";  
const API_URL = `http://${SERVER_IP}:8081/panaderia`;
```

Figura 47. Modificación al archivo script.js.

Como se mencionó, se colocaron los archivos del frontend ahora en la carpeta “resources”, ya que esta carpeta es la ubicación predeterminada donde Spring Boot busca los archivos estáticos para ser servidos al cliente, como los archivos HTML, CSS y JavaScript.

Al colocar estos archivos dentro de la carpeta resources, Spring Boot puede acceder a ellos automáticamente y servirlos a través de las rutas correspondientes cuando se accede al proyecto desde un navegador. De esta manera, se garantiza que los recursos estáticos se carguen correctamente cuando se accede a la aplicación desde varios dispositivos dentro de la misma red local, no solo de manera local con localhost.

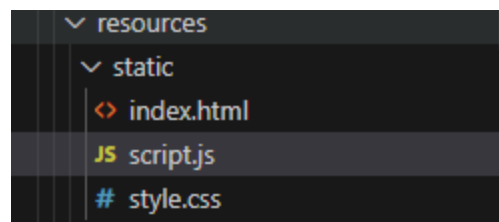


Figura 48. Estructura de la carpeta resources.

Aunque cabe decir que, para volver a hacerlo local, mediante localhost, únicamente habrá que cambiar el apartado de SERVER_IP = “192.168.100.12” por “localhost” en el archivo de script.js.

Como prueba, ahora al acceder desde un dispositivo móvil, podemos realizar compras, observar los productos disponibles y ver los resultados obtenidos desde el servidor, todo funcionando correctamente.

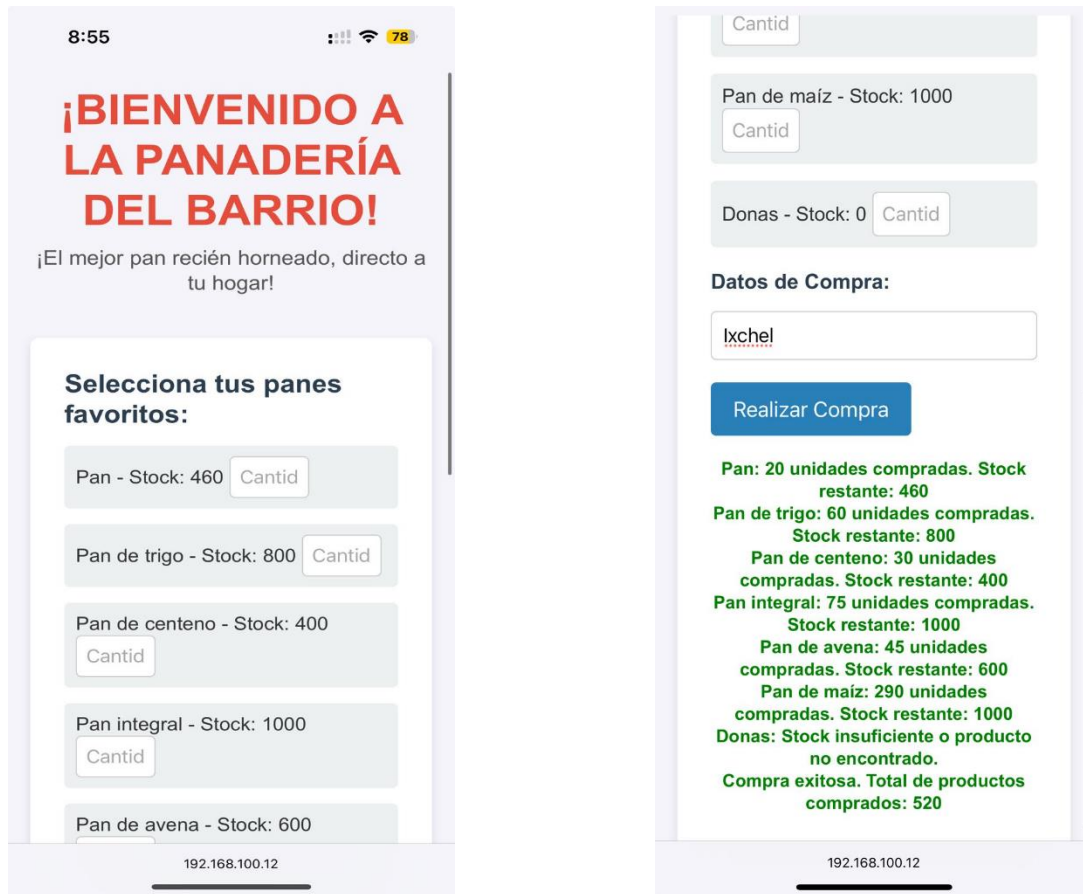


Figura 49. Prueba desde un dispositivo móvil.

```
2025-04-02T20:51:13.784-06:00 INFO 25356 --- [panaderia] [0.0-8081-exec-1] c.panaderia.service.PanaderiaServicio
: Usuario: Ixchel compr|| 20 unidades de 'Pan'. Stock restante: 460
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T20:51:13.799-06:00 INFO 25356 --- [panaderia] [0.0-8081-exec-1] c.panaderia.service.PanaderiaServicio
: Usuario: Ixchel compr|| 60 unidades de 'Pan de trigo'. Stock restante: 800
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T20:51:13.812-06:00 INFO 25356 --- [panaderia] [0.0-8081-exec-1] c.panaderia.service.PanaderiaServicio
: Usuario: Ixchel compr|| 30 unidades de 'Pan de centeno'. Stock restante: 400
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T20:51:13.824-06:00 INFO 25356 --- [panaderia] [0.0-8081-exec-1] c.panaderia.service.PanaderiaServicio
: Usuario: Ixchel compr|| 75 unidades de 'Pan integral'. Stock restante: 1000
Hibernate: select p1_0.id,p1_0.producto,p1_0.stock from inventario p1_0 where p1_0.producto=?
Hibernate: update inventario set producto=?,stock=? where id=?
2025-04-02T20:51:13.836-06:00 INFO 25356 --- [panaderia] [0.0-8081-exec-1] c.panaderia.service.PanaderiaServicio
: Usuario: Ixchel compr|| 45 unidades de 'Pan de avena'. Stock restante: 600
```

Figura 50. Mensajes desde el servidor al realizar una compra.

Finalmente, al acceder a la IP del servidor estando conectados a la misma red local, varios dispositivos pueden acceder a la aplicación, lo que permite que, tanto desde un móvil como desde una laptop, se pueda interactuar con la panadería, realizar compras y consultar los productos disponibles de manera fluida. Esto confirma que la configuración realizada ha sido exitosa y que la aplicación es accesible desde diferentes dispositivos dentro de la misma red.

Conclusión

Esta práctica me permitió profundizar en el desarrollo de sistemas web distribuidos, aplicando tecnologías modernas para la creación de una solución funcional y escalable. A lo largo de su implementación, adquirí un mejor entendimiento sobre el funcionamiento de los servicios web, su estructura y la forma en que los distintos componentes interactúan entre sí para brindar una experiencia fluida al usuario. Antes de esta práctica, mi conocimiento sobre la integración de backend y frontend era más teórico; sin embargo, al desarrollar este sistema, pude experimentar de primera mano los retos y soluciones que conlleva la implementación de una aplicación distribuida en un entorno real.

Uno de los aspectos más enriquecedores fue trabajar con Spring Boot para la creación del backend y la exposición de una API REST. Aprendí cómo estructurar correctamente un proyecto en este framework, definir controladores, servicios y repositorios, así como gestionar la base de datos con MySQL para garantizar la persistencia de la información. Además, comprendí la importancia de manejar adecuadamente las peticiones HTTP, asegurando que el servidor pueda recibir y responder a solicitudes de forma eficiente.

Por otro lado, la implementación del frontend utilizando HTML, CSS y JavaScript me permitió reforzar mis habilidades en el desarrollo de interfaces dinámicas. Pude comprobar cómo se pueden consumir servicios REST desde el cliente para obtener información en tiempo real y actualizar la interfaz de usuario de manera dinámica. También comprendí la importancia de validar los datos antes de enviarlos al servidor para evitar errores y mejorar la experiencia del usuario.

Uno de los desafíos más interesantes de esta práctica fue la simulación del horneado de panes, donde el sistema actualiza automáticamente el stock cada cinco minutos. Este mecanismo me ayudó a comprender cómo se pueden utilizar tareas programadas en Spring Boot para automatizar procesos dentro de una aplicación distribuida. Además, me permitió analizar la importancia de mantener la consistencia de los datos, asegurando que los cambios en el stock se reflejen correctamente tanto en el backend como en el frontend.

Otro punto clave fue la realización de pruebas utilizando Postman para verificar el correcto funcionamiento de los servicios web. A través de estas pruebas, pude analizar las respuestas del servidor, detectar posibles errores y asegurarme de que las peticiones GET y POST se manejaban adecuadamente. Esta etapa me permitió reforzar la importancia de probar cada componente antes de integrarlo completamente en la aplicación.

Además de lo aprendido con la implementación práctica, también complementé estos conocimientos con lo visto en clase, lo que me permitió comprender cómo los sistemas distribuidos han evolucionado con el paso del tiempo. En particular, pude notar cómo han pasado de modelos más tradicionales, como los sistemas basados en RMI o SOAP, hacia arquitecturas más flexibles y escalables, como las que utilizan servicios REST y microservicios. Esta evolución por lo visto ha permitido el desarrollo de aplicaciones más eficientes y modulares, facilitando la integración con otras tecnologías y mejorando la experiencia del usuario final.

En general, esta práctica para mí fue una experiencia enriquecedora que me permitió aplicar conocimientos teóricos en una práctica muy interesante. Aprendí sobre la estructura y desarrollo de sistemas web distribuidos, el manejo de API REST, la integración de frontend y backend, y la importancia de realizar pruebas para garantizar el correcto funcionamiento de una aplicación.

Además, me brindó una visión más clara sobre el desarrollo de software actual y la forma en que distintas tecnologías pueden integrarse para crear soluciones eficientes y escalables.

Bibliografía

- Microsoft Learn, "Conceptos de Sistemas Distribuidos," 2022. [En línea]. Available: <https://learn.microsoft.com/es-es/azure/architecture/patterns/>. [Último acceso: 01 Abril 2025].
- Spring, "Introducción a Spring Boot," 2024. [En línea]. Available: <https://spring.io/projects/spring-boot>. [Último acceso: 01 Abril 2025].
- A. Silberschatz, P. Galvin y G. Gagne, Operating System Concepts, 10ª ed., Wiley, 2018.
- Apache Maven, "Descarga e instalación de Maven," 2024. [En línea]. Available: <https://maven.apache.org/download.cgi>. [Último acceso: 01 Abril 2025].
- ResearchGate, "Arquitectura típica de un Servicio Web," 2024. [En línea]. Available: https://www.researchgate.net/figure/Figura-212-Arquitectura-tipica-de-un-Servicio-Web_fig7_305403120. [Último acceso: 01 Abril 2025].
- Disrupción Tecnológica, "Arquitectura de Servicios Web," 2024. [En línea]. Available: <https://www.disrupciontecnologica.com/arquitectura-de-servicios-web/>. [Último acceso: 01 Abril 2025].
- PHPPot, "Creación de un servicio web RESTful con PHP," 2024. [En línea]. Available: <https://phpspot.com/php/php-restful-web-service/>. [Último acceso: 01 Abril 2025].
- Platzi, "¿Qué significa REST y qué es una API RESTful?" 2024. [En línea]. Available: <https://platzi.com/clases/1638-api-rest/21611-que-significa-rest-y-que-es-una-api-restful/>. [Último acceso: 01 Abril 2025].
- IBM, "Servicios Web en WebSphere Application Server," 2024. [En línea]. Available: <https://www.ibm.com/docs/es/was/9.0.5?topic=services-web>. [Último acceso: 01 Abril 2025].
- Postman, "Guía de uso de Postman para API REST," 2024. [En línea]. Available: <https://learning.postman.com/docs/getting-started/introduction/>. [Último acceso: 01 Abril 2025]

Anexos

Controller/PanaderiaControlador.java

```
package com.panaderia.controller;

import com.panaderia.model.CompraRequest; // Nueva clase para encapsular nombreCliente y
compras

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import com.panaderia.service.PanaderiaServicio;
import com.panaderia.model.Pan;

import java.util.List;

@RestController
@RequestMapping("/panaderia")
@CrossOrigin(origins = "*")
public class PanaderiaControlador {

    @Autowired
    private PanaderiaServicio panaderiaServicio;

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

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

    // Realizar una compra
    @PostMapping("/comprar")
    public String comprarPan(@RequestBody CompraRequest request) {
        return panaderiaServicio.comprarPan(request.getNombreCliente(), request.getCompras());
    }
}
```

Model/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;
    }
}

```

Model/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;
    }
}

```

Model/Pan.java

```

package com.panaderia.model;

import jakarta.persistence.*;

@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; }
}

```

Repository/PanRepositorio.java

```

package com.panaderia.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.panaderia.model.Pan;

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

```

Service/PanaderiaServicio.java

```

package com.panaderia.service;

import com.panaderia.model.Compra;
import com.panaderia.model.Pan;
import com.panaderia.repository.PanRepositorio;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Service
public class PanaderiaServicio {

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

    @Autowired
    private PanRepositorio panRepositorio;

    // Obtener todos los productos
    public List<Pan> obtenerProductos() {
        logger.info("Obteniendo lista completa de productos.");
        return panRepositorio.findAll();
    }

    // Obtener el stock de un producto específico
    public int obtenerStock(String producto) {
        Pan pan = panRepositorio.findByProducto(producto);
        if (pan != null) {
            logger.info("Obteniendo stock de producto: '{}'. Stock disponible: {}", producto, pan.getStock());
            return pan.getStock();
        } else {

```

```

        logger.warn("Producto '{}' no encontrado en el inventario.", producto);
        return 0;
    }
}

// Realizar la compra
public String comprarPan(String nombreCliente, List<Compra> compras) {
    StringBuilder resumen = new StringBuilder();
    int totalCompra = 0;
    boolean compraExitosa = false;

    // Si no hay productos en el carrito, se devuelve un mensaje de error
    if (compras == null || compras.isEmpty()) {
        logger.warn("El cliente {} intentó realizar una compra sin productos
seleccionados.", nombreCliente);
        return "No se han seleccionado productos para la compra.";
    }

    for (Compra compra : compras) {
        Pan pan = panRepositorio.findByProducto(compra.getProducto());

        if (pan != null) {
            int stockDisponible = pan.getStock();
            int cantidadSolicitada = compra.getCantidad();

            if (stockDisponible >= cantidadSolicitada) {
                // Si hay stock suficiente, se actualiza el stock
                int stockRestante = stockDisponible - cantidadSolicitada;
                pan.setStock(stockRestante);
                panRepositorio.save(pan);

                // Log de la compra realizada
                logger.info("Usuario: {} compró {} unidades de '{}'. Stock restante: {}",
                    nombreCliente, cantidadSolicitada, compra.getProducto(),
stockRestante);

                // Añadir al resumen
                totalCompra += cantidadSolicitada;
                resumen.append(compra.getProducto())
                    .append(": ")
                    .append(cantidadSolicitada)
                    .append(" unidades compradas. Stock restante: ")
                    .append(stockRestante)
                    .append("\n");

                compraExitosa = true;
            } else {
                // Si no hay suficiente stock
                logger.warn("Compra fallida para el cliente '{}'. Producto: '{}', cantidad
solicitada: {}, stock disponible: {}",
                    nombreCliente, compra.getProducto(), cantidadSolicitada,
stockDisponible);

                resumen.append(compra.getProducto())
                    .append(": Stock insuficiente o producto no encontrado.\n");
            }
        } else {

```

```

        // Producto no encontrado
        logger.error("Producto '{}' no encontrado en la base de datos para el cliente '{}'.", compra.getProducto(), nombreCliente);
        resumen.append(compra.getProducto())
            .append(": Producto no encontrado en el inventario.\n");
    }
}

// Resultado final de la compra
if (compraExitosa) {
    resumen.append("Compra exitosa. Total de productos comprados: ")
        .append(totalCompra);
} else {
    resumen.append("No se pudo realizar la compra. Verifique el stock o los productos seleccionados.");
}

// Log del resultado final de la compra
logger.info("Compra realizada por {}. Total de productos comprados: {}", nombreCliente, totalCompra);

return resumen.toString();
}

// Método que simula el horneado de los panes cada 2 minutos
@Scheduled(fixedRate = 300000) // 300000 milisegundos = 5 minutos
public void restablecerStock() {
    // Definir la cantidad de panes a hornear por tipo de pan
    Map<String, Integer> stockPorTipo = new HashMap<>();
    stockPorTipo.put("Pan de avena", 15);
    stockPorTipo.put("Pan de trigo", 20);
    stockPorTipo.put("Pan de centeno", 10);
    stockPorTipo.put("Pan integral", 25);
    // podemos añadir más tipos de panes según sea necesario
    stockPorTipo.put("Pan de maíz", 30);
    stockPorTipo.put("Pan", 10); // Pan común

    // Obtener todos los panes en inventario
    Iterable<Pan> panes = panRepositorio.findAll();

    for (Pan pan : panes) {
        Integer stockRestablecer = stockPorTipo.get(pan.getProducto());
        if (stockRestablecer != null) {
            // Restablecer el stock de cada pan según lo definido arriba
            pan.setStock(pan.getStock() + stockRestablecer); // Sumar el stock
            panRepositorio.save(pan); // Guardar el producto actualizado
            logger.info("Stock de {} restablecido. Nuevo stock: {}", pan.getProducto(), pan.getStock());
        }
    }
}
}

```

application.properties

```

# Mostrar los logs en la terminal
logging.level.com.panaderia.service.PanaderiaServicio=INFO
spring.application.name=panaderia

```

```
spring.datasource.url=jdbc:mysql://localhost:3306/panaderia?useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
server.port=8081
```