INSTITUTO POLITÉCTICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Actividad:

Practica 8 – PWA

Integrante:

Hernández Vázquez Jorge Daniel

Profesor:

Chadwick Carreto Arellano

Fecha de entrega:

06/05/2025

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Índice de contenido

Índice de Figuras

Antecedente

El desarrollo de aplicaciones web ha evolucionado considerablemente en los últimos años, dando paso a enfoques más modernos y centrados en la experiencia del usuario. Entre estas innovaciones destacan las Progressive Web Apps (PWA), una tecnología que combina lo mejor del desarrollo web y móvil, permitiendo a los usuarios instalar aplicaciones directamente desde el navegador, sin necesidad de pasar por tiendas oficiales. Las PWA han surgido como respuesta a la necesidad de crear aplicaciones más rápidas, confiables y accesibles, incluso en condiciones de conectividad limitada. Su arquitectura basada en tecnologías estándar como HTML, CSS, JavaScript, y el uso de Service Workers, las convierte en una solución eficiente y adaptable para entornos multiplataforma.

Aplicaciones Web

Las aplicaciones web han evolucionado significativamente desde sus inicios en la década de los 90. Inicialmente, las páginas web eran estáticas, compuestas únicamente por documentos HTML que ofrecían contenido limitado sin interacción con el usuario. Con el paso del tiempo, surgieron tecnologías como JavaScript y CSS, que permitieron introducir interactividad y estilos visuales avanzados.

La aparición de **AJAX (Asynchronous JavaScript and XML)** marcó un punto de inflexión, al permitir la actualización de partes específicas de una página web sin necesidad de recargarla completamente. Esto dio lugar a lo que hoy se conoce como **Web 2.0**, caracterizada por aplicaciones más dinámicas e interactivas.

Posteriormente, con la llegada de **HTML5** y la adopción de nuevas APIs por parte de los navegadores, las aplicaciones web comenzaron a ofrecer funcionalidades similares a las de las aplicaciones de escritorio o móviles, tales como acceso al almacenamiento local, geolocalización y capacidades multimedia. Esta evolución ha sido clave en el surgimiento de las Progressive Web Apps (PWA), las cuales representan la última etapa en esta transformación.

A pesar de su evolución, las aplicaciones web tradicionales presentan diversas limitaciones en comparación con las aplicaciones nativas:

- **Dependencia constante de conexión a internet**: muchas aplicaciones web dejan de funcionar total o parcialmente en entornos offline.
- **Rendimiento limitado**: aunque ha mejorado con los años, el rendimiento de una aplicación web suele ser inferior al de una aplicación nativa, especialmente en dispositivos móviles o con funcionalidades avanzadas.
- **Sin acceso completo al hardware**: las aplicaciones web están restringidas por el entorno del navegador, lo que limita el acceso a sensores del dispositivo, sistema de archivos, Bluetooth, entre otros.
- **Experiencia de usuario limitada**: las interfaces web, aunque responsivas, a menudo no logran replicar completamente la fluidez y sensación nativa de una app instalada.

Estas limitaciones motivaron el desarrollo de tecnologías que pudieran combinar lo mejor del mundo web y móvil, dando origen a las Progressive Web Apps.

Rol de las tecnologías web (HTML, CSS, JavaScript, HTTP)

Las tecnologías web fundamentales siguen siendo la base sobre la cual se construyen tanto las aplicaciones web tradicionales como las PWAs:

- **HTML (HyperText Markup Language)**: proporciona la estructura básica del contenido. Su evolución (HTML5) ha permitido incorporar elementos semánticos, multimedia y formularios enriquecidos.
- **CSS (Cascading Style Sheets)**: define la presentación visual de la aplicación. Con CSS3, se introdujeron transiciones, animaciones y diseño responsivo.
- **JavaScript**: es el lenguaje de programación que dota de interactividad y lógica a las aplicaciones. Su ecosistema (frameworks, librerías, APIs) ha sido crucial para el desarrollo moderno del front-end.
- **HTTP (HyperText Transfer Protocol)**: es el protocolo de comunicación entre el cliente (navegador) y el servidor. Con la llegada de **HTTP/2** y **HTTPS**, se han mejorado aspectos de seguridad y rendimiento.

Estas tecnologías, al trabajar en conjunto, han hecho posible que las aplicaciones web modernas sean altamente funcionales, escalables y cada vez más cercanas a las experiencias nativas.

Progressive Web Apps (PWA)

Una Progressive Web App (PWA) es una aplicación web que utiliza tecnologías modernas para proporcionar una experiencia similar a la de una aplicación nativa en dispositivos móviles y de escritorio. Las PWAs combinan lo mejor de ambos mundos: la accesibilidad y universalidad de la web con la experiencia inmersiva de las aplicaciones instalables.

A diferencia de las aplicaciones tradicionales que deben descargarse desde una tienda (como Google Play o App Store), una PWA puede instalarse directamente desde el navegador, sin necesidad de pasar por procesos de publicación. Esta instalación permite que el usuario tenga un acceso directo a la aplicación desde su pantalla de inicio, con una interfaz que puede ejecutarse incluso sin conexión a internet.

El término “progresiva” hace referencia a su capacidad de adaptación: estas aplicaciones están diseñadas para funcionar correctamente en cualquier navegador moderno, y escalar su funcionalidad en función de las capacidades del dispositivo o del entorno de ejecución. Por ejemplo, en dispositivos con hardware limitado o conexión inestable, una PWA puede seguir siendo funcional gracias a mecanismos como el almacenamiento en caché y los service workers.

El concepto de Progressive Web App fue introducido formalmente por Google en el año 2015, como una respuesta a las limitaciones que enfrentaban tanto las aplicaciones web como las nativas. Antes de esto, los desarrolladores debían elegir entre crear una aplicación web (accesible y multiplataforma, pero con limitaciones técnicas) o una aplicación nativa (potente pero costosa y dependiente de cada plataforma).

Con el aumento del uso de dispositivos móviles y la necesidad de reducir los costos de desarrollo, surgió la necesidad de un enfoque unificado. Las tecnologías web habían alcanzado un punto de madurez suficiente como para permitir experiencias enriquecidas. Esto incluyó mejoras como el uso de service workers, notificaciones push, almacenamiento local, y APIs avanzadas del navegador.

Desde su introducción, el soporte para PWA ha crecido considerablemente. Navegadores como Chrome, Firefox, Edge y Safari han ido incorporando soporte para las funcionalidades necesarias. Además, empresas como Twitter, Pinterest y Starbucks han implementado PWAs con excelentes resultados, reduciendo tiempos de carga, mejorando la retención de usuarios y optimizando recursos.

La evolución de PWA se encuentra alineada con la visión de una web más rápida, segura y universal, donde los usuarios no dependan de una tienda de aplicaciones para acceder a experiencias completas desde cualquier dispositivo. En la siguiente Figura podemos observar la arquitectura básica de PWA.

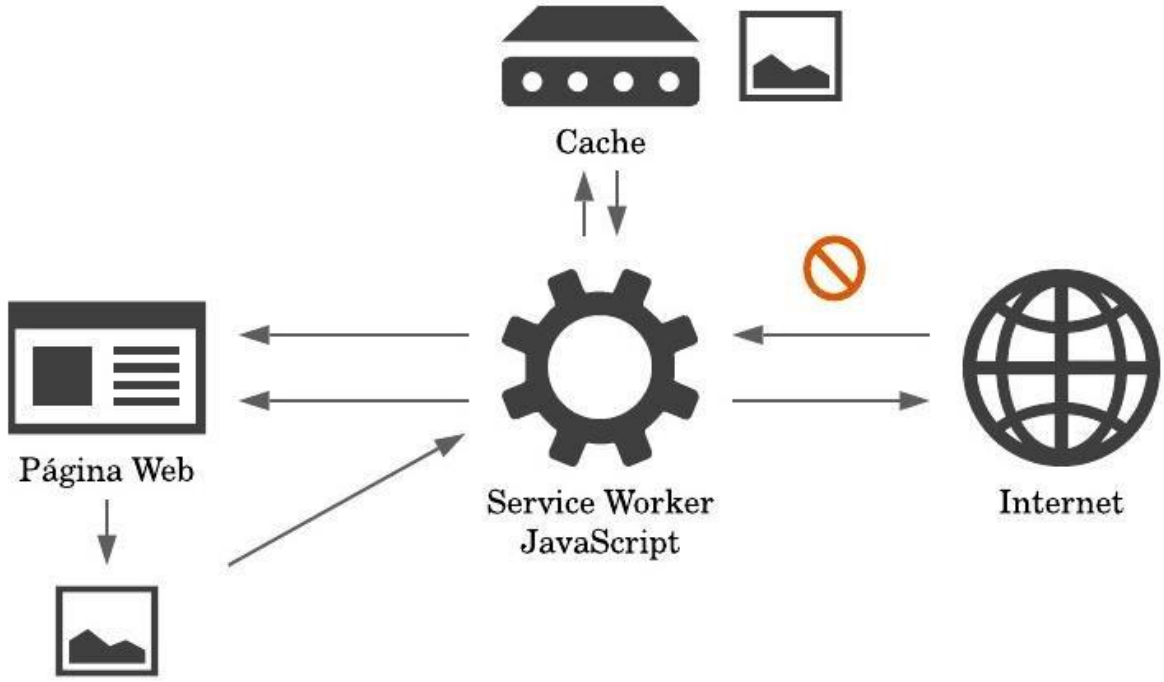


Figura 1. Arquitectura básica de PWA.

Características principales de una PWA

Las Progressive Web Apps se definen por una serie de características que las hacen destacar sobre las aplicaciones web tradicionales:

- **Responsividad (Responsive Design)**: Las PWAs están diseñadas para adaptarse automáticamente a cualquier tamaño de pantalla o tipo de dispositivo, ya sea móvil, tableta, laptop o pantalla grande.
- **Capacidad offline**: Mediante el uso de service workers, las PWAs pueden funcionar sin conexión a internet, almacenando en caché los recursos y datos necesarios para mantener la experiencia del usuario incluso cuando no hay conectividad.
- **Instalabilidad**: Una PWA puede instalarse fácilmente desde el navegador, sin pasar por una tienda de aplicaciones. Al instalarla, se comporta como una app nativa: aparece en el menú de aplicaciones, puede ejecutarse en una ventana propia y utilizar recursos locales del dispositivo.

- **Seguridad**: Las PWAs requieren servirse bajo HTTPS, lo que garantiza una conexión segura, cifrada y confiable. Esta característica es fundamental para proteger la integridad y privacidad del usuario.
- **Actualización progresiva**: A través de los service workers, las PWAs pueden gestionar sus propias actualizaciones de forma eficiente, descargando únicamente los recursos modificados.
- **Interactividad similar a una app nativa**: Las animaciones fluidas, las transiciones, el acceso al hardware y las notificaciones push hacen que la experiencia de usuario sea comparable a la de una aplicación móvil convencional.
- **Ligereza y rendimiento**: En comparación con las aplicaciones nativas, las PWAs suelen ser más ligeras, ya que no requieren bibliotecas pesadas ni múltiples versiones para diferentes sistemas operativos.

Diferencias entre PWA y aplicaciones nativas/híbridas

Entonces, para comprender mejor el lugar que ocupan las PWAs en el ecosistema de desarrollo, es importante compararlas con otros tipos de aplicaciones, como vemos en la siguiente tabla:

Característica	Aplicación Nativa	Aplicación Híbrida	Progressive Web App
Distribución	Tiendas de apps (Play Store, App Store, etc.)	Tiendas de apps	Navegador web, instalable desde sitio
Acceso al hardware	Completo	Parcial (depende del framework)	Parcial (en expansión)
Desarrollo multiplataforma	No, requiere código distinto por plataforma	Sí, con frameworks como Ionic, React Native o Flutter	Sí, con tecnologías web estándar
Actualizaciones	A través de la tienda, requiere aprobación	A través de la tienda o backend	Automáticas, controladas por el navegador
Funcionamiento offline	Sí	Sí (depende de la implementación)	Sí, mediante service workers
Rendimiento	Alto	Medio/alto	Medio (aunque en constante mejora)
Costos de desarrollo	Altos (por plataforma)	Moderados	Bajos

Como podemos ver en la comparación, las PWAs ofrecen una alternativa atractiva cuando se busca una solución multiplataforma que reduzca costos, mantenga buena experiencia de usuario y funcione

eficientemente tanto online como offline. Aunque aún tienen limitaciones frente al acceso completo al hardware, el desarrollo web moderno ha logrado reducir esa brecha de forma significativa.

Tecnologías Clave en Progressive Web Apps (PWA)

El funcionamiento y las ventajas de una PWA son posibles gracias a un conjunto de tecnologías modernas que operan en conjunto. A continuación se detallan las principales herramientas y componentes que permiten a las PWAs brindar una experiencia avanzada, segura y similar a la de una aplicación nativa.

Service Workers: qué son, cómo funcionan y su papel en el funcionamiento offline

Los Service Workers son una de las piezas fundamentales que hacen posible muchas de las capacidades avanzadas de las PWAs, como la navegación sin conexión, el almacenamiento en caché inteligente y las notificaciones push.

Un Service Worker es un script de JavaScript que actúa como intermediario entre la aplicación web, el navegador y la red (cuando está disponible). Opera en segundo plano, separado del hilo principal de ejecución de la página, y permite interceptar y controlar las solicitudes de red. Esto significa que el Service Worker puede decidir si una solicitud debe ser respondida desde la red, desde la caché o incluso generar una respuesta personalizada.

Funcionamiento general:

1. Registro: El Service Worker se registra desde el archivo principal de la aplicación.
2. Instalación: En esta fase, se pueden almacenar recursos en la caché.
3. Activación: Aquí se limpia la caché antigua y se finaliza la configuración.
4. Intercepción de eventos (fetch): Cada vez que se realiza una solicitud, el Service Worker puede manejarla.

Rol en el funcionamiento offline: El principal beneficio del Service Worker es su capacidad de mantener parte o toda la aplicación funcional sin conexión a internet. Al almacenar recursos estáticos (como HTML, CSS, JS, imágenes) en la caché, la aplicación puede seguir respondiendo al usuario aun cuando no tenga conectividad, lo cual representa una mejora significativa en la experiencia de usuario.

Manifest File: propósito y estructura

El **Web App Manifest** es un archivo en formato JSON que proporciona al navegador la información necesaria sobre cómo debería comportarse una PWA cuando se instala en un dispositivo. Es el archivo responsable de que la aplicación se vea y funcione como una app nativa desde la pantalla de inicio del dispositivo.

Propósitos principales:

- Permitir que la aplicación se instale como un ícono en el escritorio o pantalla de inicio.
- Definir los colores, temas, y orientación de pantalla.
- Especificar el nombre y la URL que se abrirá al iniciar la app.

Figura 2. Estructura básica de un manifest file.

Caching y almacenamiento local: tipos y estrategias

Uno de los aspectos clave de las PWAs es su capacidad para funcionar de forma rápida y confiable, incluso sin conexión. Para lograrlo, se utilizan distintas formas de almacenamiento en el lado del cliente.

Tipos de almacenamiento:

- **Cache Storage (caché de Service Workers):** Se utiliza para guardar archivos estáticos, como HTML, CSS, JS e imágenes. Es controlada por el Service Worker.
- **LocalStorage:** Almacena datos en pares clave-valor en el navegador. Es sincrónico y de capacidad limitada (~5MB).
- **SessionStorage:** Similar a LocalStorage, pero su duración está limitada a la sesión actual del navegador.
- **IndexedDB:** Base de datos en el navegador que permite almacenar grandes cantidades de datos estructurados. Ideal para aplicaciones complejas.

Estrategias comunes de caché:

- **Cache First:** Se consulta la caché antes de hacer una petición a la red. Ideal para recursos estáticos.

- **Network First:** Se intenta obtener la respuesta de la red, y si falla, se recurre a la caché. Útil para contenido dinámico.
- **Stale-while-revalidate:** Se sirve el recurso desde la caché mientras se actualiza en segundo plano.
- **Cache Only / Network Only:** Usadas en casos específicos donde se prefiere una sola fuente de datos.

Estas estrategias permiten adaptar el comportamiento de la aplicación según la conectividad y el tipo de recurso solicitado.

HTTPS y seguridad

La seguridad es un pilar esencial en el diseño de las Progressive Web Apps. Todas las PWAs deben ser servidas obligatoriamente sobre **HTTPS**, es decir, bajo un protocolo seguro con cifrado SSL/TLS.

Razones por las que HTTPS es obligatorio:

- **Protección del usuario:** Evita que terceros intercepten o manipulen la información intercambiada entre la app y el servidor.
- **Service Workers requieren HTTPS:** Como los Service Workers tienen la capacidad de interceptar solicitudes de red, podrían ser utilizados maliciosamente si no se restringe su uso a contextos seguros.
- **Integridad de la app:** Garantiza que los recursos cargados por el navegador no han sido modificados por atacantes en tránsito.

Además del uso de HTTPS, una buena PWA debe aplicar prácticas como:

- Validación de entradas para evitar ataques XSS o SQL Injection.
- Uso de políticas de seguridad de contenido (Content Security Policy - CSP).
- Uso de permisos explícitos para acceder a funcionalidades como ubicación o notificaciones.

Estas tecnologías trabajan en conjunto para dar forma a las funcionalidades modernas de las PWAs, permitiéndoles ser más rápidas, confiables, seguras e instalables, lo que representa un avance significativo en el desarrollo web.

PWA en el Contexto de Sistemas Distribuidos

Un **sistema distribuido** se define como un conjunto de computadoras independientes que aparecen ante el usuario como un sistema único y cohesionado. Estos sistemas trabajan coordinadamente, comparten recursos y cooperan para cumplir tareas complejas, todo mientras se ocultan los detalles de su distribución.

En este contexto, una **Progressive Web App (PWA)** puede considerarse parte integral de un sistema distribuido por varias razones técnicas y funcionales:

- **Interacción cliente-servidor**: Una PWA actúa como cliente dentro de una arquitectura distribuida, comunicándose con servidores remotos mediante APIs REST o GraphQL, normalmente expuestas a través de HTTP/HTTPS.

- **Almacenamiento y procesamiento distribuido**: Muchas PWAs se integran con servicios en la nube (como Firebase, AWS o servicios personalizados) que ofrecen bases de datos, autenticación, almacenamiento y funciones en la nube, distribuidos geográficamente.
- **Desacoplamiento de componentes**: En una solución moderna, una PWA puede consumir microservicios distribuidos, cada uno con responsabilidades específicas y desplegados en distintos entornos.
- **Tolerancia a fallos y replicación**: Gracias a técnicas de caché, funcionamiento offline y sincronización en segundo plano, las PWAs aumentan la disponibilidad, tolerancia a desconexiones y resiliencia del sistema en general.
- **Actualización independiente**: Las PWAs pueden recibir actualizaciones desde el backend sin necesidad de re-distribuir el cliente, lo cual es una ventaja común en sistemas distribuidos bien diseñados.

Por lo tanto, desde la perspectiva de arquitectura, una PWA moderna no solo participa en un sistema distribuido, sino que puede ser un nodo inteligente que aporta capacidades offline, caché y experiencia local a una solución distribuida globalmente.

Ventajas de usar PWA en entornos distribuidos

El uso de PWAs dentro de sistemas distribuidos aporta numerosas ventajas, tanto a nivel técnico como de experiencia de usuario:

1. Reducción de latencia y carga en servidores

Gracias al uso de Service Workers y almacenamiento local, muchas peticiones pueden ser atendidas directamente desde el cliente, lo que reduce el número de solicitudes al servidor y mejora la velocidad percibida.

2. Tolerancia a fallos y funcionamiento offline

En un sistema distribuido, la conectividad a todos los nodos no siempre está garantizada. Las PWAs permiten seguir operando localmente incluso cuando se pierde conexión con alguno de los servicios centrales, algo crítico en entornos donde la disponibilidad no siempre es constante.

3. Escalabilidad

Al centralizar la lógica del sistema en APIs distribuidas y dejar la capa de presentación (la PWA) en el cliente, se mejora la escalabilidad del sistema. Más usuarios pueden ser atendidos sin necesidad de escalar masivamente el backend.

4. Actualización progresiva y continua

El Service Worker permite que las actualizaciones del cliente se hagan de manera gradual y controlada, sin interrumpir al usuario. Esto se alinea con los principios de despliegue continuo típicos en arquitecturas distribuidas modernas.

5. Independencia de la plataforma

Las PWAs funcionan en múltiples dispositivos y sistemas operativos sin necesidad de múltiples desarrollos, lo que reduce la complejidad de mantener clientes para cada plataforma en un sistema distribuido heterogéneo.

6. Costos operativos más bajos

Al no depender de tiendas de aplicaciones y al poder operar con recursos locales, las PWAs permiten construir sistemas distribuidos con menor inversión en infraestructura y mantenimiento.

Ejemplos de casos de uso o aplicaciones reales

Numerosas empresas y organizaciones han implementado PWAs como parte de sistemas distribuidos, con excelentes resultados en rendimiento, disponibilidad y experiencia del usuario. Algunos ejemplos destacados incluyen:

- **Twitter Lite**: PWA que reemplaza a la aplicación nativa en muchos dispositivos. Funciona con bajo consumo de datos y excelente rendimiento incluso en redes 2G. Opera como frontend de un sistema altamente distribuido con múltiples APIs, servidores de contenido y almacenamiento en caché global.
- **Starbucks**: La PWA de Starbucks permite hacer pedidos incluso sin conexión. Se sincroniza automáticamente cuando se recupera la conectividad. Esto forma parte de una arquitectura distribuida que incluye bases de datos geográficamente replicadas y servicios en la nube.
- **Uber PWA**: Diseñada para ser funcional incluso con conexiones de red lentas, la PWA de Uber es extremadamente ligera (menor a 100KB) y se integra con servicios distribuidos de geolocalización, pagos y enrutamiento de viajes.
- **Google Maps Go**: Una versión ligera y rápida de Google Maps que usa tecnología PWA para ofrecer funcionalidades esenciales sin consumir demasiados recursos. Se integra con servidores distribuidos para datos de mapas, tráfico y búsqueda.
- **Pinterest**

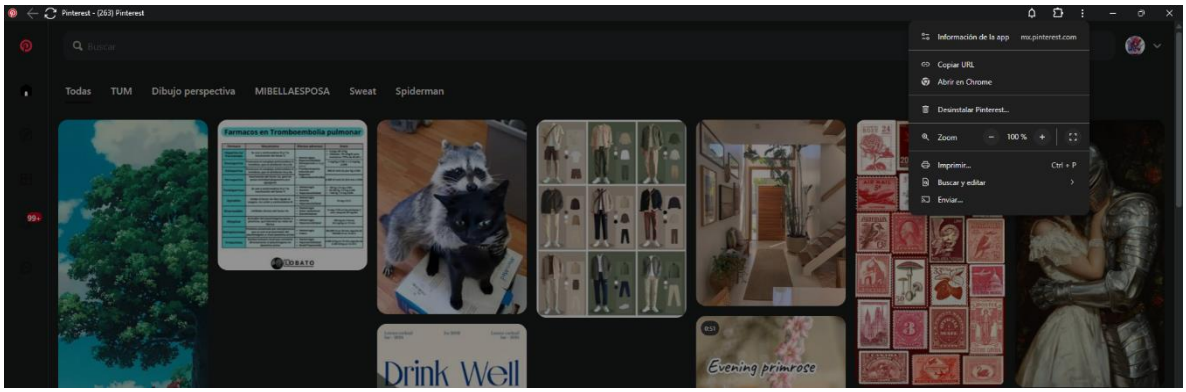


Figura 3. PWA de Pinterest.

Estos casos demuestran cómo las PWAs pueden desempeñar un papel clave en soluciones distribuidas, especialmente cuando se busca portabilidad, disponibilidad y bajo costo sin sacrificar experiencia de usuario.

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	PWA (Progressive Web App)
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	Cliente web avanzado que consume múltiples servicios distribuidos (APIs, bases de datos, notificaciones, etc.)
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	Alta, se adapta bien a arquitecturas escalables mediante servicios backend desacoplados y caching local
Mantenimiento	Complejo, cualquier cambio afecta a todo	Medio, pero el servidor puede ser cuello de botella	Mayor facilidad para mantener servicios individuales	Puede ser complejo por la comunicación entre objetos	Sencillo, cada servicio puede actualizarse de forma independiente	Sencillo, se actualiza en el cliente vía Service Workers sin pasar por tiendas de aplicaciones
Interoperabilidad	Baja, difícil integrar	Media, limitada a capacidad	Media-Alta, depende de los protocolos	Media, depende del	Alta, comunicación	Alta, usa tecnologías web

	nuevas tecnologías	es del servidor		middleware utilizado (CORBA, RMI)	mediante estándares abiertos (HTTP, JSON, XML)	estándares y puede integrarse con APIs REST, GraphQL, y servicios en la nube
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	APIs, sistemas de microservicios en la nube	Aplicaciones web modernas como Twitter Lite, Starbucks PWA, Uber Web, que operan online y offline desde navegador

Las Progressive Web Apps (PWAs) representan una evolución significativa dentro del ecosistema de aplicaciones distribuidas. Su diseño moderno y flexible permite combinar lo mejor de las aplicaciones web y móviles, aprovechando una arquitectura orientada a servicios que se alinea estrechamente con los principios de los sistemas distribuidos.

A diferencia de los modelos monolíticos o estrictamente cliente-servidor, las PWAs se integran perfectamente en entornos **multiservidor** o **basados en microservicios**, ya que pueden consumir múltiples APIs distribuidas y operar eficientemente incluso en condiciones de conectividad limitada. Esto es posible gracias al uso de tecnologías como **Service Workers, almacenamiento local y manifest files**, que les otorgan capacidades offline, responsividad e instalabilidad.

En comparación con otros modelos arquitectónicos más tradicionales, las PWAs ofrecen ventajas destacadas en **escalabilidad, mantenimiento e interoperabilidad**, gracias a su construcción sobre tecnologías web estándar y su independencia de plataformas o tiendas de aplicaciones. Además, su capacidad para actualizarse de forma automática y silenciosa en el cliente las convierte en una solución ágil y adaptable para sistemas distribuidos modernos.

En general, las PWAs no solo son una alternativa ligera y accesible para los usuarios finales, sino que también representan una **solución técnica eficiente y robusta** dentro de arquitecturas distribuidas, especialmente en escenarios donde la experiencia del usuario, la conectividad intermitente y la portabilidad son factores clave.

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, seria el siguiente, el de mi practica:

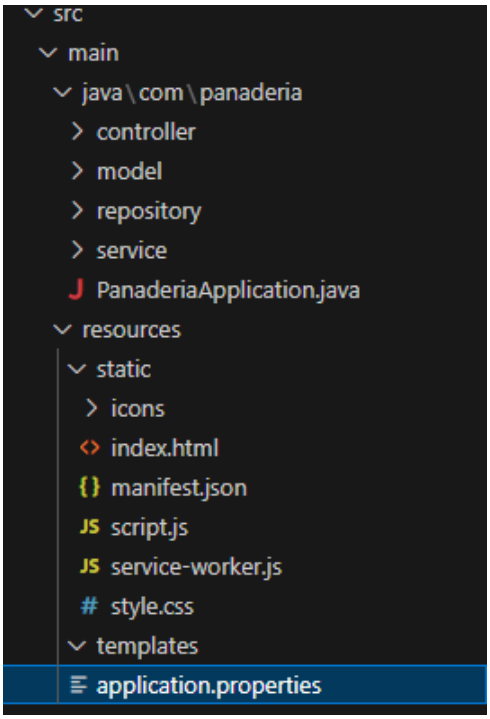


Figura 4. Estructura de mi practica con Spring Boot.

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 una aplicación web progresiva (PWA) para la gestión de una panadería, integrando tecnologías del lado del cliente como HTML, CSS y JavaScript, junto con un backend desarrollado en Spring Boot. Esta aplicación busca simular un entorno donde los usuarios pueden consultar productos disponibles, realizar compras en línea y recibir notificaciones en tiempo real sobre actualizaciones del inventario, todo en un entorno distribuido y concurrente.

En este contexto, surge el problema de cómo garantizar la integridad y consistencia de los datos cuando múltiples usuarios acceden simultáneamente al sistema. En una panadería real, el stock de productos es limitado y varía constantemente según las compras realizadas y la producción. Si varios clientes intentan comprar el mismo producto al mismo tiempo, sin una correcta sincronización, se corre el riesgo de que el sistema registre ventas de productos que ya no están disponibles, lo que generaría insatisfacción y errores en la gestión del inventario.

Además, en sistemas tradicionales, los usuarios no suelen recibir notificaciones inmediatas sobre cambios importantes en el sistema, como la reposición del stock o el agotamiento de un producto. Esto representa una limitación significativa cuando se busca ofrecer una experiencia fluida, moderna y centrada en el usuario. La ausencia de mecanismos de comunicación en tiempo real entre el servidor y los clientes puede dificultar la toma de decisiones al momento de comprar.

Otro aspecto que complica este escenario es la necesidad de mantener funcional la aplicación incluso en condiciones de conectividad limitada o intermitente, algo común entre usuarios que acceden desde dispositivos móviles. Las aplicaciones web tradicionales no están diseñadas para operar sin conexión, lo que puede limitar su utilidad en ciertos momentos críticos para el cliente.

Finalmente, la gestión automatizada de procesos internos, como la simulación del horneado de pan, plantea el desafío de actualizar periódicamente el inventario sin intervención manual y reflejar estos cambios de manera inmediata para todos los usuarios conectados. Resolver estas situaciones es crucial para simular adecuadamente el funcionamiento de una panadería moderna en un entorno digital, y representa el problema central que esta práctica busca abordar.

Propuesta de solución

Para abordar el problema planteado, se propone el desarrollo de una Progressive Web App (PWA) conectada a un backend distribuido implementado con Spring Boot. Esta solución permitirá a múltiples usuarios consultar productos disponibles en una panadería, realizar compras en línea y recibir notificaciones en tiempo real sobre cambios en el inventario. La propuesta busca integrar tecnologías modernas del lado del cliente y del servidor para simular una experiencia realista, eficiente y escalable en la gestión de un sistema distribuido.

La solución estará compuesta por los siguientes componentes principales:

1. Frontend como Aplicación Web Progresiva (PWA): El cliente será desarrollado como una aplicación web progresiva usando HTML, CSS y JavaScript. Sus principales funcionalidades serán:

- Mostrar los productos disponibles y su stock en tiempo real.
- Permitir al usuario seleccionar productos, indicar cantidades y realizar compras.
- Registrar al cliente mediante la introducción de su nombre.
- Mostrar notificaciones automáticas al usuario cuando el inventario se actualice.
- Funcionar parcialmente sin conexión mediante almacenamiento en caché con Service Workers.
- Tener un manifest.json que permita su instalación como aplicación en dispositivos móviles.

2. Backend con Spring Boot: El backend se desarrollará con Spring Boot y gestionará toda la lógica del sistema. Entre sus responsabilidades se incluyen:

- Exponer una API RESTful para el acceso a productos, compras y notificaciones.
- Procesar compras, actualizar el inventario y evitar ventas cuando no haya stock suficiente.
- Ejecutar una tarea programada para simular el horneado periódico del pan, reponiendo automáticamente el inventario.
- Emitir eventos en tiempo real mediante Server-Sent Events (SSE) para notificar a los clientes sobre actualizaciones del sistema.

3. Comunicación mediante API REST y SSEñ La aplicación contará con dos mecanismos principales de comunicación entre cliente y servidor:

- **API RESTful**, con los siguientes endpoints:
    - GET /panaderia/productos: Devuelve la lista actual de productos y su stock.
    - POST /panaderia/comprar: Permite enviar una compra compuesta por cliente y productos seleccionados.
- **Eventos SSE**:
    - A través del endpoint GET /panaderia/eventos, el cliente se suscribe para recibir notificaciones automáticas cuando el stock sea modificado.

4. Base de datos MySQL para persistencia: Toda la información se almacenará en una base de datos MySQL que permitirá:

- Consultar y actualizar productos y stock de forma segura.

5. Gestión de concurrencia y transacciones: Para asegurar que el sistema funcione correctamente en entornos concurrentes, se implementarán:

- **Transacciones atómicas** en las operaciones de compra para evitar inconsistencias.
- **Validaciones previas** al confirmar una compra, verificando el stock actualizado.

- Control de errores que impida a un cliente realizar una compra si los productos ya no están disponibles.

6. Automatización de la producción (stock): Mediante una tarea programada con @Scheduled en Spring Boot, se simulará el horneado de pan cada cierto intervalo (por ejemplo, cada 5 minutos), aumentando el stock de los productos automáticamente.

7. Experiencia sin conexión y actualización silenciosa: Gracias al uso de Service Workers, la aplicación será capaz de:

- Cargar archivos esenciales como HTML, CSS, JS e íconos incluso sin conexión.
- Actualizar en segundo plano los recursos cacheados cuando se detecte una nueva versión en el servidor.

Con esta solución, se logrará una aplicación web distribuida moderna y funcional que no solo permite a múltiples usuarios interactuar con el sistema de manera concurrente, sino que también ofrece una experiencia fluida, en tiempo real y resiliente a fallos de red. Esta práctica me permitirá profundizar en temas como desarrollo full-stack, arquitectura cliente-servidor, programación reactiva con eventos SSE, tareas programadas en backend y diseño de PWAs para distintos entornos.

Materiales y métodos empleados

Para llevar a cabo la práctica y desarrollar la simulación del sistema distribuido basado en PWA, 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)** Se utilizó VSCode como entorno principal para el desarrollo tanto del frontend como del backend. Sus extensiones facilitaron la programación en Java, HTML, CSS y JavaScript, además de permitir depuración eficiente del 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 implementar el sistema distribuido con funcionalidad en tiempo real y una interfaz interactiva, se aplicaron los siguientes métodos:

A continuación, se describen los principales métodos utilizados:

1. **Arquitectura cliente-servidor distribuida**: Se desarrolló una arquitectura distribuida basada en servicios web, donde el servidor Spring Boot expone endpoints REST y gestiona la lógica de negocio, mientras que el cliente PWA interactúa con el sistema a través de solicitudes HTTP y eventos SSE.

2. **Manejo de concurrencia en el inventario**: Las compras concurrentes fueron gestionadas mediante transacciones en la base de datos y control de stock desde el backend. Se verificó el stock antes de registrar la compra para evitar inconsistencias, utilizando métodos sincronizados y validaciones antes del commit.

3. **Simulación automatizada del horneado de pan**: Para mantener el inventario actualizado, se implementó una tarea programada (@Scheduled) en Spring Boot que incrementa periódicamente el stock de los productos. Este mecanismo simula el horneado automático de pan, ejecutándose, por ejemplo, cada 5 minutos.

4. **Notificaciones en tiempo real con Server-Sent Events (SSE)**: Se implementó un canal de comunicación unidireccional en tiempo real mediante SSE. Cada vez que el stock se actualiza (por compra), se emite un evento al que todos los clientes conectados están suscritos, permitiendo así la actualización dinámica del frontend sin necesidad de recargar la página.

5. **Desarrollo de una PWA como frontend**: La interfaz de usuario se diseñó como una aplicación web progresiva (PWA) utilizando HTML, CSS y JavaScript. Esta permite a los usuarios consultar productos, ingresar su nombre, seleccionar cantidades y realizar compras. También se integraron Service Workers y un archivo manifest.json para permitir su instalación como aplicación móvil y funcionamiento offline básico.

6. **Pruebas de integración y validación con Postman**: Cada endpoint del backend fue probado individualmente mediante Postman. Se verificó la correcta devolución de productos (GET), el registro de compras (POST) y la actualización del stock en condiciones normales y con escenarios de error (como stock insuficiente).

7. **Monitoreo y registros del sistema**: Se integró un sistema de logs mediante SLF4J para monitorear las operaciones clave del sistema, como las compras, actualizaciones de stock y ejecuciones de tareas programadas. Estos registros fueron fundamentales para detectar errores y validar el comportamiento del sistema durante las pruebas.

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

Interfaz de Usuario (Frontend)

La interfaz del sistema fue desarrollada como una PWA ligera e interactiva que permite una comunicación fluida con el backend y una experiencia optimizada para el usuario.

1. **HTML - Estructura de la aplicación:** El archivo index.html contiene la estructura básica de la aplicación. Incluye un formulario para ingresar el nombre del cliente, una lista de productos con sus cantidades y un botón para realizar la compra.
2. **CSS - Estilos visuales:** Se utilizó un archivo style.css para personalizar la apariencia de la aplicación, mejorando la disposición, los botones y la visualización de los productos para facilitar la usabilidad.
3. **JavaScript - Lógica y conexión con el backend:** El archivo script.js contiene funciones para:
    - Obtener la lista de productos (GET /productos)
    - Registrar una compra (POST /comprar)
    - Escuchar eventos SSE para actualizar el inventario en tiempo real sin recargar
    - Mostrar notificaciones visuales al usuario cuando se actualiza el stock
4. **Service Workers y manifest.json:** Se configuraron Service Workers para permitir el almacenamiento en caché de los recursos esenciales, y un archivo manifest.json que habilita la instalación como app móvil. Esto aporta resistencia ante fallos de red y mejora la experiencia del usuario.
5. **Conexión frontend-backend:** El frontend realiza solicitudes al backend usando la API definida en http://localhost:8081/panaderia. Los datos se comunican en formato JSON tanto para solicitudes como para respuestas, garantizando compatibilidad con múltiples dispositivos y navegadores.

Gracias al uso combinado de estas herramientas y metodologías, se logró simular de manera efectiva el comportamiento de un sistema distribuido moderno, que incluye interacción concurrente de múltiples clientes, automatización de procesos, comunicación en tiempo real y una experiencia web progresiva.

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**. Cabe destacar que el backend o API utilizada en esta práctica fue **reutilizada y adaptada** a partir del proyecto previamente desarrollado durante la práctica de **servicios web**, aprovechando su estructura modular y su lógica de negocio ya implementada.

La solución propuesta para la gestión distribuida del inventario en la panadería se basa en un modelo **Cliente/Servidor**, donde múltiples clientes pueden conectarse simultáneamente para consultar productos, realizar compras y recibir actualizaciones de stock en tiempo real. A nivel de backend, se diseñó un sistema capaz de sincronizar múltiples servidores a través de una base de datos **MySQL compartida**, lo que garantiza que el inventario se mantenga consistente incluso ante solicitudes concurrentes.

Por su parte, el **frontend fue desarrollado como una PWA (Progressive Web App)** moderna, que consume directamente la **API REST expuesta por el backend reutilizado**. Esta interfaz web permite a los usuarios interactuar con el sistema de forma fluida desde distintos dispositivos, incluso con capacidades offline básicas gracias al uso de **Service Workers**. Además, se implementó una funcionalidad de comunicación en tiempo real mediante **Server-Sent Events (SSE)**, que permite que los clientes reciban actualizaciones automáticas del inventario sin necesidad de recargar la página, brindando así una experiencia interactiva y responsiva.

Este enfoque híbrido —reutilización del backend existente y construcción de un frontend PWA sobre esa base— permitió optimizar los tiempos de desarrollo y lograr una solución coherente, escalable y centrada en la experiencia del usuario.

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**.

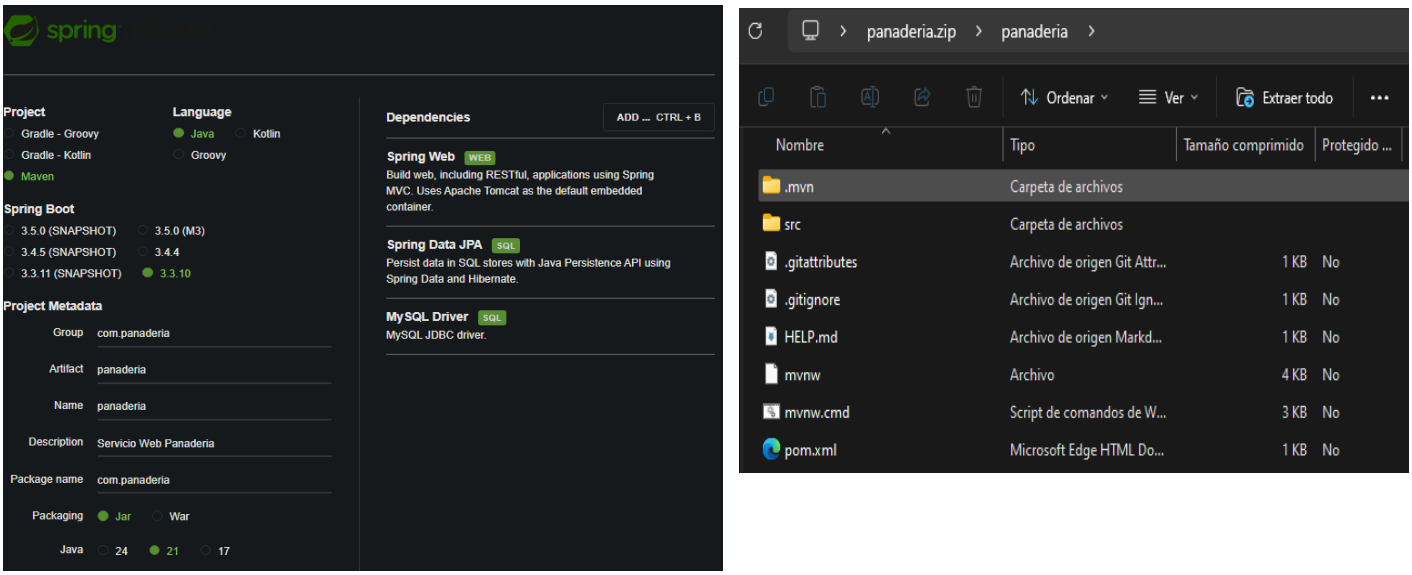


Figura 5. Estructura y creación del proyecto con Spring Boot.

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**.

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 principal del sistema desarrollado con **Spring Boot**, encargado de manejar todas las solicitudes HTTP del cliente relacionadas con la gestión de productos, compras

y actualizaciones en tiempo real. Este controlador es parte fundamental del patrón de arquitectura MVC, ya que actúa como intermediario entre el frontend (cliente) y la lógica de negocio contenida en el servicio.

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.

Comunicación en tiempo real con SSE (Server-Sent Events)

Una característica avanzada implementada en este controlador es la posibilidad de **comunicación en tiempo real** desde el servidor hacia los clientes, utilizando **Server-Sent Events (SSE)**. Esto permite que el cliente reciba notificaciones automáticas sin necesidad de hacer "polling" constante.

Método 4: Suscripción a eventos en tiempo real

- Este método devuelve un objeto SseEmitter que permite mantener una conexión abierta con el cliente.
- Cada cliente que accede a /panaderia/eventos se suscribe para recibir actualizaciones.

```
    // Endpoint SSE para que el frontend escuche actualizaciones en tiempo real
@GetMapping(value = "/eventos", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public SseEmitter emitirEventos() {
    SseEmitter emisor = new SseEmitter(Long.MAX_VALUE); // Emisor sin límite de tiempo
    emisores.add(emisor);

    emisor.onCompletion(() -> emisores.remove(emisor)); // Eliminar al completar
    emisor.onTimeout(() -> emisores.remove(emisor));    // Eliminar al expirar

    return emisor;
}
```

Figura 15. Método para emisión de eventos en tiempo real.

Método 5: Notificación de compra realizada

- Este método escucha eventos de tipo EventoCompraRealizada emitidos por el servicio.
- Cuando una compra se realiza, se genera un mensaje que es enviado a todos los clientes suscritos vía SSE.

```
// Método para escuchar el evento de compra realizada y notificar a los clientes
@EventListener
public void onCompraRealizada(EventoCompraRealizada evento) {
    String mensaje = evento.getMensaje();
    List<SseEmitter> emisoresAEliminar = new ArrayList<>();

    for (SseEmitter emisor : emisores) {
        try {
            emisor.send(SseEmitter.event().data(mensaje));
        } catch (IOException e) {
            // En caso de error, marcamos el emisor para ser eliminado
            emisoresAEliminar.add(emisor);
        }
    }

    // Eliminar los emisores que no pudieron enviar el mensaje
    emisores.removeAll(emisoresAEliminar);
}
```

Figura 16. Método para notificaciones de compras realizadas.

Método 6: Notificación de reposición de stock

- Similar al anterior, pero activado cuando se repone el inventario.
- Envía un mensaje de reposición a todos los clientes conectados.

```
// Método para escuchar eventos de reposición de stock y notificar a los clientes
@EventListener
public void onReposicionStock(EventoStockRepuesto evento) {
    String mensaje = evento.getMensaje();
    List<SseEmitter> emisoresAEliminar = new ArrayList<>();

    for (SseEmitter emisor : emisores) {
        try {
            emisor.send(SseEmitter.event().data(mensaje));
        } catch (IOException e) {
            emisoresAEliminar.add(emisor);
        }
    }

    emisores.removeAll(emisoresAEliminar);
}
```

Figura 17. Método para notificaciones de reposición de stock.

Este archivo como podemos ver representa el núcleo de interacción del sistema, ya que:

- Permite a los usuarios consultar productos y stock, y realizar compras a través de una interfaz REST.
- Integra notificaciones en tiempo real mediante SSE, manteniendo a los clientes informados sin recargar la página.
- Facilita la comunicación entre el frontend y el backend, lo cual es clave en la experiencia PWA.

Este controlador, combinado con el servicio y el modelo de datos, logra una solución eficiente, sincronizada y moderna para la gestión de inventario distribuido en una panadería.

2. PanaderiaServicio.java

Este archivo contiene la lógica de negocio de la panadería, encargándose de procesar las compras, consultar el stock y simular el horneado periódico de panes para reabastecer el inventario. También se encarga de enviar notificaciones en tiempo real al frontend usando eventos personalizados.

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.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
```

```
@Autowired
```

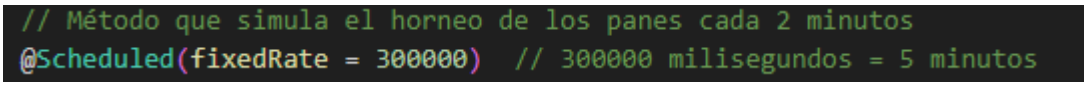


Figura 18. Librerías y anotaciones utilizadas.

- **@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:

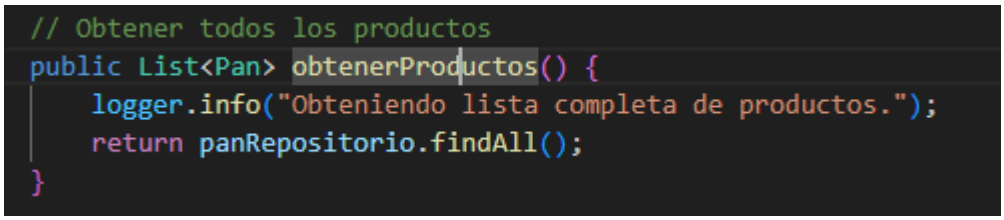


Figura 19. Método para obtener los productos dispoinibles.

- **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 especifico

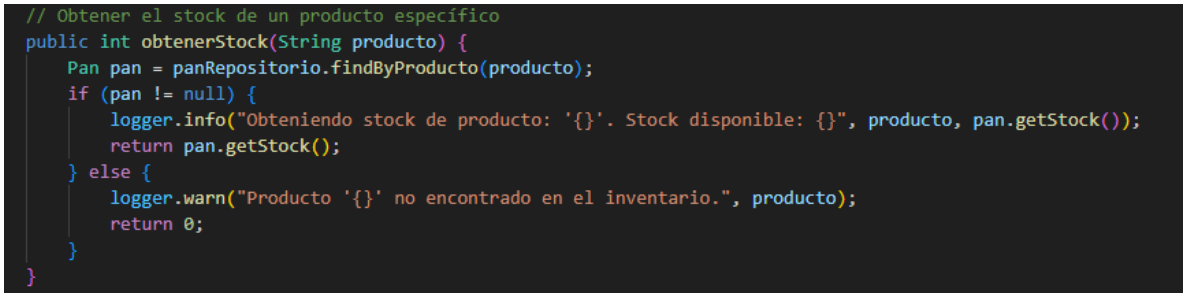


Figura 20. Método para obtener el stock de un producto especifico.

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 21. 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.

- actualizarStockYNotificar(String producto, int stockRestante): Este método encapsula la lógica para notificar al frontend que el stock ha cambiado, enviando un evento con un mensaje personalizado.

```java
// Método para notificar la actualización del stock
private void actualizarStockYNotificar(String producto, int stockRestante) {
    // Notificar a los clientes conectados que el stock ha cambiado
    String mensaje = "El stock de '" + producto + "' ha sido actualizado. Stock restante: " + stockRestante;
    eventPublisher.publishEvent(new EventoCompraRealizada(mensaje));
}
```

Figura 22. Método para notificar actualizaciones de stock.

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

```java
// Método programado para restablecer el stock de panes cada 5 minutos
@Scheduled(fixedRate = 300000) // Cada 5 minutos (300000 ms)
public void restablecerStock() {
    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);
    stockPorTipo.put(key:"Pan de maíz", value:30);
    stockPorTipo.put(key:"Pan", value:10);

    Iterable<Pan> panes = panRepositorio.findAll();

    for (Pan pan : panes) {
        Integer stockRestablecer = stockPorTipo.get(pan.getProducto());
        if (stockRestablecer != null) {
            pan.setStock(pan.getStock() + stockRestablecer);
            panRepositorio.save(pan);
            logger.info("Stock de {} restablecido. Nuevo stock: {}", pan.getProducto(), pan.getStock());

            // Publicar evento de reposición de stock
            actualizarStockYNotificar(pan.getProducto(), pan.getStock());
        }
    }

    // Notificar que el inventario ha sido repuesto
    eventPublisher.publishEvent(new EventoStockRepuesto(mensaje:"El horno ha repuesto el inventario de panes"));
}
```

Figura 23. 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.
- Al final, se envía un evento general indicando que el horno ha reabastecido el inventario (EventoStockRepuesto).

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 gestiona toda la lógica de negocio relacionada con productos, compras e inventario, y trabaja de forma conjunta con eventos Spring para emitir notificaciones al frontend en tiempo real, brindando una experiencia interactiva y actualizada para los usuarios.

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:

```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 24. 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:

```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 25. 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:

```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 26. 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:

```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 27. 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 PanRepositorio 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 PanRepositorio 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 PanRepositorio.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. Eventos y Publicación de Mensajes en Tiempo Real

Estos archivos complementan la lógica del servicio principal (PanaderiaServicio.java) al encargarse de la publicación y definición de eventos personalizados, permitiendo que otras partes del sistema (como el controlador SSE o el frontend PWA) reaccionen en tiempo real a cambios en el sistema, como compras realizadas o reabastecimiento de stock.

EventPublisher.java

- **@Component:** Esta anotación registra la clase como un componente de Spring, permitiendo su uso mediante inyección de dependencias.
- **ApplicationEventPublisher:** Interfaz de Spring para publicar eventos a los que otros componentes pueden suscribirse.
- **publicarEvento(String mensaje):** Método auxiliar que encapsula la lógica de creación y publicación del evento EventoCompraRealizada.

```java
@Component
public class EventPublisher {

    @Autowired
    private ApplicationEventPublisher publisher;

    // Publica un evento
    public void publicarEvento(String mensaje) {
        // Crear un evento con el mensaje
        publisher.publishEvent(new EventoCompraRealizada(mensaje));
    }
}
```

Figura 28. Codigo de EventPublisher.

Este componente actúa como intermediario para emitir eventos personalizados que serán consumidos por otras capas del sistema (por ejemplo, el controlador SSE que envía los mensajes al cliente web en tiempo real).

EventoCompraRealizada.java

- Este evento es utilizado para notificar cuando se realiza una compra o hay una actualización de stock.

Este evento es capturado por otros componentes que estén escuchando eventos de tipo EventoCompraRealizada, lo que permite la comunicación desacoplada y asincrónica entre capas.

EventoStockRepuesto.java

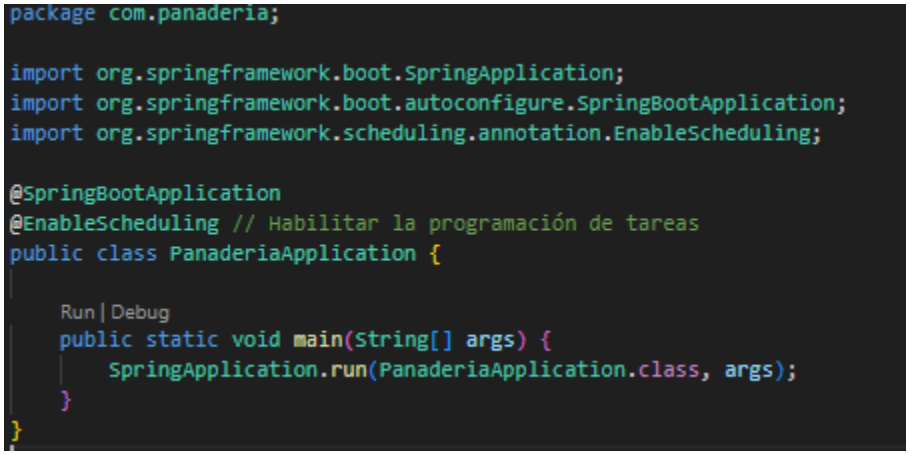
- A diferencia del evento anterior, **no extiende ApplicationEvent**, pero puede ser igualmente publicado usando ApplicationEventPublisher.
- Se utiliza exclusivamente para notificar cuando se ha completado el proceso programado de reabastecimiento del inventario.

Este evento permite enviar notificaciones específicas relacionadas con el horneado automático de productos, separando este caso de uso de las compras manuales realizadas por clientes.

En general, estas clases y componentes forman la base para un sistema **reactivo** dentro de la aplicación, que trabaja en conjunto con tecnologías como **Server-Sent Events (SSE)** para brindar una experiencia de usuario dinámica y actualizada en tiempo real.

8. PanaderiaApplication.java

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.



```java
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 29. Codigo 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.

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 como Progressive Web App (PWA)

Para complementar el backend desarrollado con Spring Boot, se implementó un **frontend como** Progressive Web App **(PWA)** utilizando tecnologías estándar como HTML, CSS y JavaScript. Esta solución no solo permite a los usuarios interactuar con el sistema de panadería de forma intuitiva, sino que también garantiza una experiencia fluida incluso en condiciones de red limitada o sin conexión.

La interfaz fue diseñada con el objetivo de ofrecer una experiencia de usuario optimizada para dispositivos móviles y de escritorio, integrando características como **caché offline, notificaciones en tiempo real, instalación como app y actualizaciones automáticas**, gracias al uso de un Service Worker y un manifest.json.

A continuación, se detallan los archivos y características principales que componen este frontend:

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, se enlazan los estilos, scripts y recursos clave que permiten que la aplicación funcione como una **Progressive Web App (PWA)**, incluyendo el Service Worker y el manifest.json.

Estructura del HTML

La estructura comienza con las etiquetas estándar de HTML5, donde se especifica el idioma (lang="es"), el conjunto de caracteres (UTF-8) y la configuración de visualización adaptable en dispositivos móviles a través de la etiqueta <meta name="viewport">. Además, se incluye el archivo manifest.json, que permite que la aplicación pueda instalarse en la pantalla de inicio del dispositivo, y se define un color de tema para mejorar la apariencia en navegadores móviles.

```
<!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">
    <link rel="manifest" href="manifest.json">
    <meta name="theme-color" content="#f8b400">
</head>
```

Figura 30. Etiquetas básicas de HTML.

Cuerpo del documento

Dentro de la etiqueta <body>, se define la estructura visual de la página, compuesta por un contenedor principal ( ) que incluye:

- Un encabezado con el nombre de la panadería y una breve descripción.
- Una sección central que permite a los usuarios:

- Visualizar los productos disponibles dinámicamente en el elemento , gestionado desde el archivo script.js.
- Ingresar su nombre mediante un campo de texto.
- Confirmar la compra con un botón que ejecuta la función comprarPan().
- Visualizar un mensaje de confirmación o error tras la compra.
- Recibir notificaciones en tiempo real, como eventos de reposición de stock o compras realizadas, las cuales se mostrarán en el área .

```
<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"></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>

            <h3>Notificaciones:</h3>
            <div id="notificaciones" class="notificaciones"></div>
        </div>
    </div>
```

Figura 31. Cuerpo del documento HTML.

Inclusión de scripts

Al final del documento se incluye el archivo script.js, que contiene la lógica de interacción con el backend. Además, se implementa el registro del Service Worker, elemento esencial para que la aplicación funcione en modo offline y se comporte como una app instalada.

```
    <script src="script.js"></script>
    <script>
        // Registro del Service Worker
        if ('serviceWorker' in navigator) {
            navigator.serviceWorker.register('service-worker.js')
                .then(reg => console.log('Service Worker registrado', reg))
                .catch(err => console.error('Error al registrar Service Worker', err));
        }
    </script>
</body>
</html>
```

Figura 32. Uso del script.js.

Esta instrucción permite que la aplicación almacene en caché los archivos estáticos y funcione incluso sin conexión a internet.

En general, este archivo HTML proporciona una interfaz sencilla, moderna y eficiente, donde los usuarios pueden ver productos, realizar compras, y recibir notificaciones en tiempo real, todo desde una experiencia fluida que puede funcionar incluso sin conexión. Gracias al uso de tecnologías como Service Worker, manifest y EventSource, este frontend representa una evolución desde una página web tradicional hacia una PWA completamente funcional.

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 toda la interacción entre el usuario y el sistema, gestionando la visualización de productos, la realización de compras y la recepción de eventos en tiempo real a través de Server-Sent Events (SSE). Este script comunica al frontend con la API REST del backend de la panadería y proporciona una experiencia fluida y dinámica para el usuario.

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 33. 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 34. 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 35. 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 36. 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 37. 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.

Función iniciarSSE()

Esta función permite escuchar eventos en tiempo real del servidor utilizando Server-Sent Events (SSE):

- Se conecta al endpoint /eventos del backend mediante EventSource.
- Cada vez que se recibe un evento (por ejemplo, una actualización de stock), se muestra un mensaje en la sección de notificaciones de la interfaz.
- También se vuelve a llamar a obtenerProductos() para refrescar el stock mostrado.
- Las notificaciones se eliminan automáticamente luego de 10 segundos.
- Si el navegador no soporta SSE, se muestra una advertencia por consola.

```
// Escuchar eventos del servidor para actualizar stock automáticamente
function iniciarSSE() {
    if (!!window.EventSource) {
        const eventSource = new EventSource(`${API_URL}/eventos`);

        eventSource.onmessage = function (event) {
            const mensaje = event.data;

            const notificaciones = document.getElementById("notificaciones");
            const p = document.createElement("p");
            p.textContent = mensaje;
            notificaciones.appendChild(p);

            // Recargar toda la lista de productos para reflejar stock actualizado
            obtenerProductos();

            setTimeout(() => {
                p.remove();
            }, 10000);
        };


        eventSource.onerror = function (error) {
            console.error("Error en SSE:", error);
        };
    } else {
        console.warn("SSE no es compatible con este navegador.");
    }
}
```

Figura 38. Bloque de código para escuchar eventos en tiempo real con SSE.

En general, el archivo script.js cumple un papel central en la experiencia de usuario. Permite:

- Obtener y mostrar productos dinámicamente.
- Validar y enviar compras al backend.
- Recibir notificaciones en tiempo real con SSE.

Esta implementación mejora notablemente la interacción del usuario, ofreciendo una interfaz moderna, eficiente y actualizada, con capacidades propias de una **Progressive Web App (PWA)**.

4. manifest.json

El archivo manifest.json es un componente esencial para convertir una aplicación web en una Progressive Web App (PWA). Este archivo proporciona al navegador información sobre cómo se debe comportar la aplicación cuando se instala en un dispositivo, ya sea móvil o de escritorio.

Este manifiesto define aspectos clave como el nombre de la aplicación, los íconos utilizados, la configuración de colores, y el modo de visualización.

A continuación, se detallan sus atributos:

Atributos principales

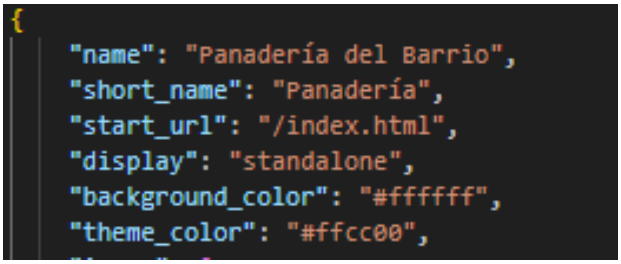


Figura 39. Atributos principales del manifest.json.

- **name**: Nombre completo de la aplicación. Se mostrará en la pantalla de inicio del usuario y en otros contextos donde haya espacio suficiente. → "Panadería del Barrio"
- **short_name**: Versión corta del nombre que se utiliza cuando el espacio es limitado (por ejemplo, debajo del ícono en la pantalla de inicio). → "Panadería"
- **start_url**: URL con la que debe iniciarse la aplicación cuando el usuario la abra desde la pantalla de inicio. → "/index.html"
- **display**: Define el modo en que se muestra la app. "standalone" hace que se vea como una aplicación nativa, ocultando la barra de direcciones y otros elementos del navegador.
- **background_color**: Color de fondo utilizado en la pantalla de carga mientras se inicia la app. → "#ffffff" (blanco)
- **theme_color**: Color temático principal que influye en la apariencia del navegador (barra de herramientas, barra superior en Android, etc.). → "#ffcc00" (un tono amarillo)

Íconos

Figura 40. Iconos del manifest.json.

Se definen los íconos que la aplicación usará cuando sea instalada. Estos deben estar disponibles en la carpeta icons/ y ser imágenes cuadradas en formato PNG.

En general, el archivo manifest.json es clave para que los navegadores reconozcan y traten nuestro sitio como una aplicación instalable. En este caso, el manifiesto está configurado para que la panadería tenga:

- Identidad visual personalizada.
- Inicio directo como app (sin barra del navegador).
- Compatibilidad con múltiples dispositivos.

Y como se pudo ver en el index.html, lo tenemos enlazado para que los navegadores compatibles lo utilicen.

5. service-worker.js

El archivo service-worker.js es una pieza clave en la transformación de la aplicación web de la panadería en una Progressive Web App (PWA). Su principal función es **gestionar el almacenamiento en caché de los recursos estáticos**, permitiendo que la aplicación funcione sin conexión a internet y mejore su rendimiento.

La instalación se realiza mediante el evento "install", donde se abre una caché con un nombre definido (panaderia-cache-v1) y se almacenan en ella todos los archivos esenciales para que la aplicación funcione offline:

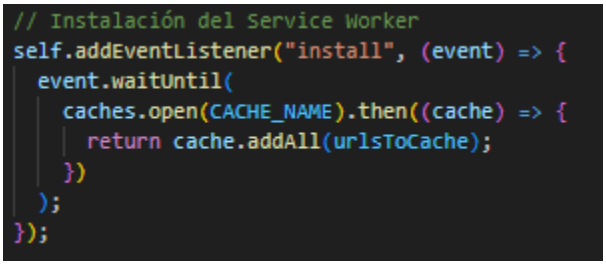


Figura 41. Instalación del service worker.

Los archivos almacenados incluyen:

- La página principal (/index.html)
- Los estilos (/style.css)
- El script de interacción (/script.js)
- El archivo de configuración de PWA (/manifest.json)
- Los íconos necesarios para la instalación en dispositivos (/icons/icon-192.png, /icons/icon-512.png)

Esta caché se usará para servir contenido rápidamente y garantizar disponibilidad sin conexión.

Activación y limpieza de cachés antiguas

Cuando se activa una nueva versión del Service Worker, se ejecuta el evento "activate". En este se eliminan las versiones anteriores de la caché que ya no son necesarias:

```
// Activación del Service Worker - limpieza de caché antigua
self.addEventListener("activate", (event) => {
  const cacheWhitelist = [CACHE_NAME];
  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (!cacheWhitelist.includes(cacheName)) {
            return caches.delete(cacheName); // Elimina cachés viejas
          }
        })
      );
    })
  );
});
```

Figura 42. Activación y limpieza de cache antigua.

Este proceso garantiza que la aplicación siempre use la versión más actualizada de los recursos y evita el uso de archivos obsoletos.

Intercepción de peticiones (evento fetch)

El Service Worker también intercepta todas las peticiones de red que hace la aplicación utilizando el evento "fetch". Esta lógica implementa una estrategia mixta de **"cache-first con actualización en segundo plano"**:

```
// Manejo de peticiones de red - actualiza la caché en segundo plano
self.addEventListener("fetch", (event) => {
  event.respondWith(
    caches.match(event.request).then((cachedResponse) => {
      const fetchRequest = event.request.clone();

      // Si la respuesta está en caché, la devuelve inmediatamente
      if (cachedResponse) {
        // Actualiza la caché con la nueva respuesta sin interrumpir la experiencia del usuario
        fetch(fetchRequest).then((response) => {
          caches.open(CACHE_NAME).then((cache) => {
            cache.put(event.request, response.clone());
          });
        });
        return cachedResponse;
      }

      // Si no está en caché, realiza la solicitud normalmente
      return fetch(fetchRequest);
    })
  );
});
```

Figura 43. Manejo de peticiones.

- Si el recurso solicitado ya se encuentra en caché, se devuelve inmediatamente, asegurando una carga rápida.

- Al mismo tiempo, se realiza una nueva solicitud a la red para obtener una versión más reciente del recurso, que se guarda en la caché de forma silenciosa para futuras visitas.
- Si el recurso no está en la caché, se descarga normalmente de la red.

En general, gracias a este Service Worker, la aplicación:

- Funciona incluso cuando no hay conexión a internet.
- Carga rápidamente usando recursos almacenados localmente.
- Se mantiene actualizada en segundo plano.
- Cumple con los criterios fundamentales de una PWA moderna.

Este archivo, junto al manifest.json y el registro del Service Worker en el HTML, garantiza que la panadería tenga una interfaz web confiable, eficiente y disponible en todo momento, incluso sin conexión.

Con esta implementación, la aplicación web de la panadería está completamente desarrollada como una Progressive Web App funcional. El backend, construido con Spring Boot, se encarga de gestionar el inventario, procesar las compras en tiempo real y emitir notificaciones mediante Server-Sent Events (SSE). Por su parte, el frontend, implementado con HTML, CSS y JavaScript, permite a los usuarios visualizar el stock disponible, realizar pedidos de manera sencilla e inmediata y recibir actualizaciones dinámicas sin necesidad de recargar la página. Además, gracias al uso de tecnologías como Service Worker y manifest.json, la aplicación puede instalarse en dispositivos móviles y funcionar incluso sin conexión a internet. En conjunto, este sistema ofrece una experiencia moderna, rápida y confiable tanto para los clientes como para los administradores de la panadería.

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\PWA_panaderia**

1. Compilación y construcción del proyecto

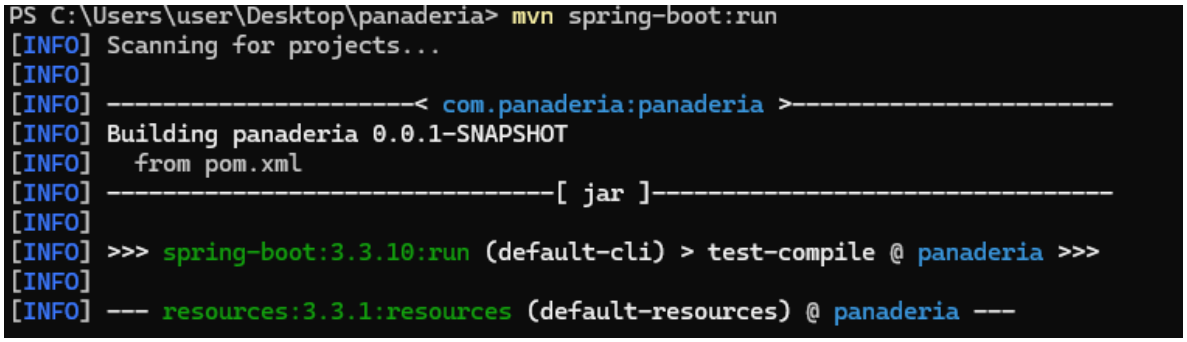
El primer paso es compilar y construir el proyecto con Maven. Para ello, ejecutamos el siguiente comando en la terminal:

```
PS C:\Users\user\Desktop\PWA_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 ---
```

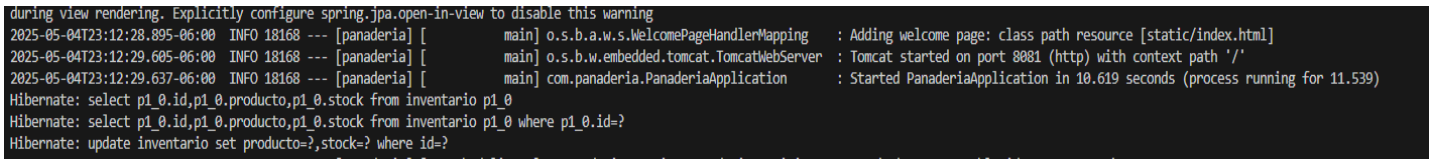
Este comando se encarga de limpiar cualquier compilación previa, descargar las dependencias necesarias y generar el paquete del proyecto. Es un paso fundamental para asegurarnos de que el código está correctamente preparado para ejecutarse.

2. Ejecución del servicio

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



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.



3. Pruebas con Postman o navegador

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

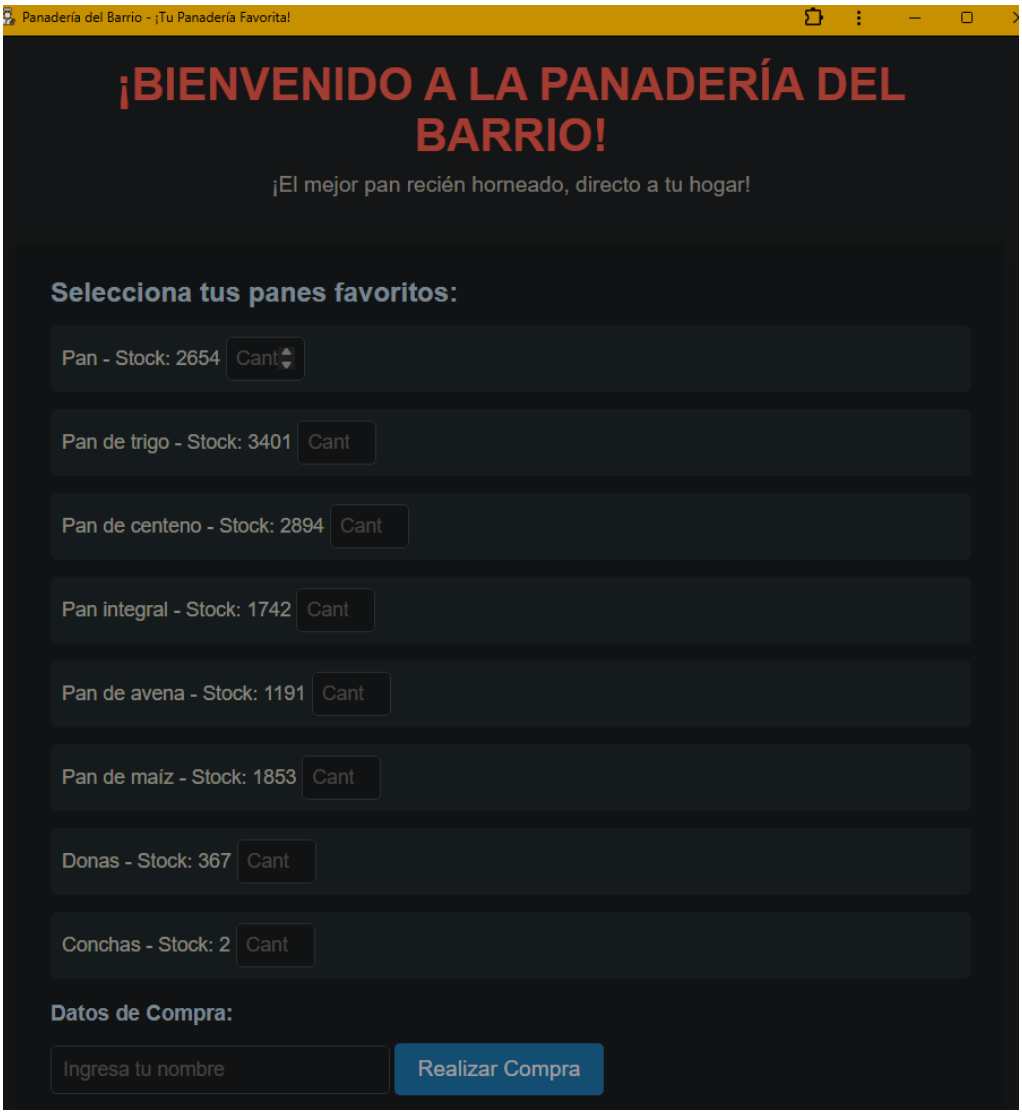
- **Usando Postman:** Enviando solicitudes GET y POST a los endpoints de la API para verificar el funcionamiento del sistema.
- **Usando el frontend:** Para ello, es necesario únicamente acceder desde nuestro navegador a localhost:8081 donde veremos y podremos instalar nuestra PWA.

4. Ejecución del Frontend

Para interactuar con el sistema de manera gráfica, debemos ejecutar el frontend, como se menciono accediento desde localhost:8081.

Una vez cargado el frontend de la panadería, el usuario puede visualizar los productos disponibles, seleccionar las cantidades deseadas, ingresar su nombre y realizar la compra de manera sencilla. Si la transacción se procesa correctamente, el sistema muestra un mensaje de confirmación y actualiza automáticamente el stock gracias a la comunicación en tiempo real mediante SSE. Además, al tratarse de una aplicación web progresiva (PWA), los usuarios pueden instalarla en sus dispositivos móviles o de escritorio, lo que permite acceder a ella como si fuera una app nativa, incluso sin conexión a internet gracias al uso del Service Worker y el almacenamiento en caché. Con esta implementación,

el sistema de la panadería queda completamente operativo, ofreciendo una experiencia moderna, intuitiva y funcional, tanto en línea como en modo offline.



Resultados

El sistema web de panadería progresiva desarrollado con Spring Boot como backend y tecnologías web modernas en el frontend (HTML, CSS, JS) ha demostrado un funcionamiento correcto, eficiente y adaptado a entornos distribuidos. La implementación como Progressive Web App (PWA) permite a los usuarios no solo acceder a las funcionalidades principales desde cualquier navegador, sino también **instalar la aplicación como si fuera nativa**, con soporte para funcionamiento offline, gracias al uso del manifest.json y del service-worker.js.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 44. 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.

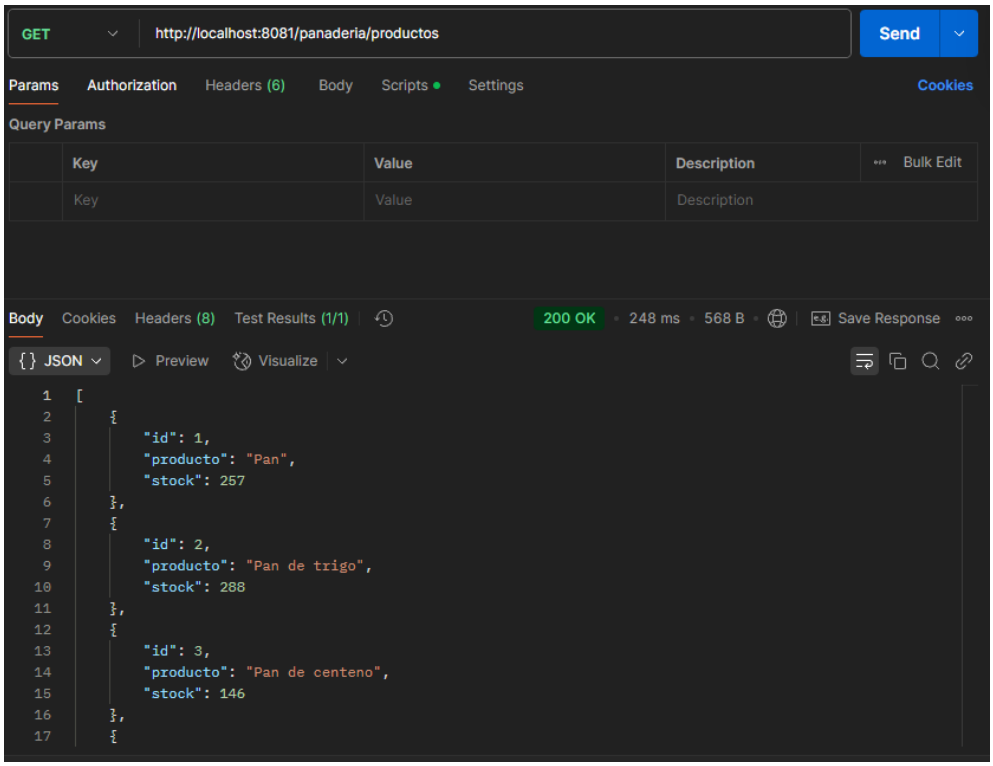


Figura 45. Prueba del método GET, para obtener el listado de productos.

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.

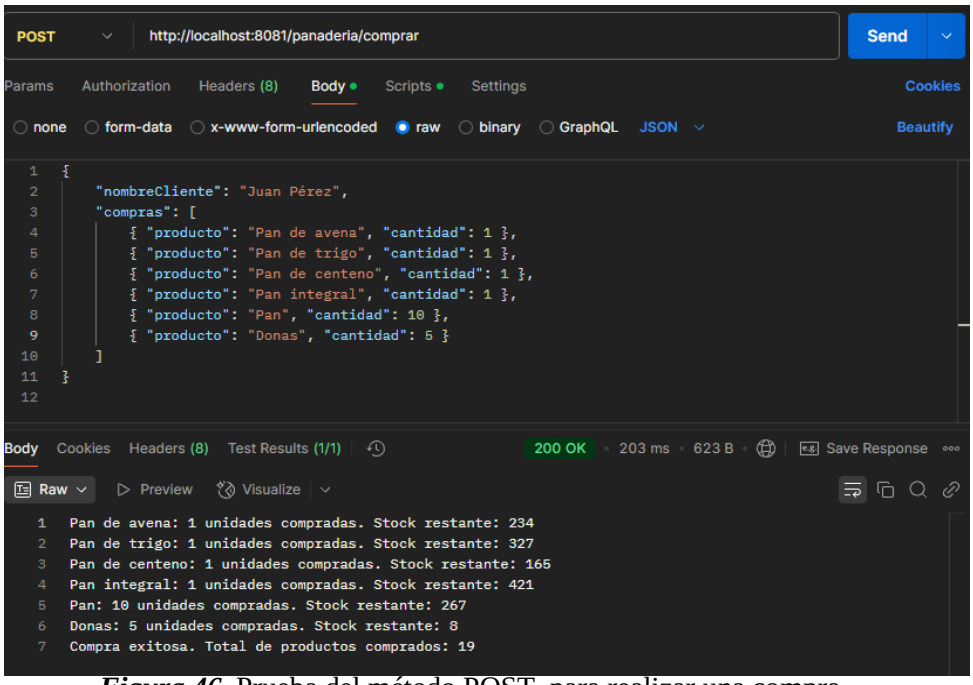


Figura 46. Prueba del método POST, para realizar una compra.

Y en el servidor obtenemos la siguiente respuesta o log.

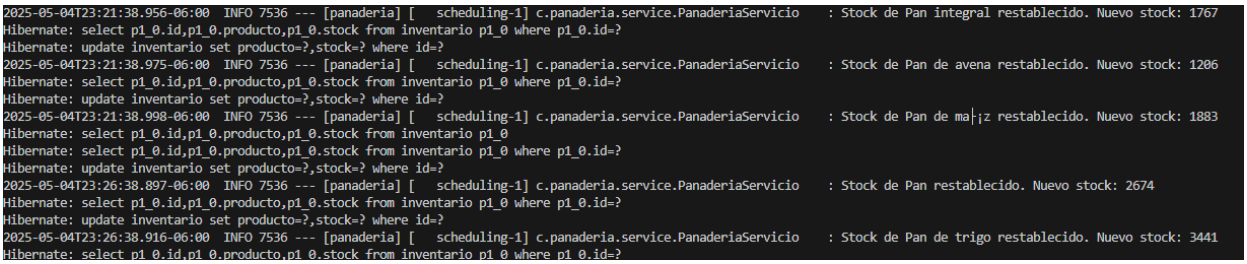


Figura 47. Respuesta del servidor al cliente realizar una compra.

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

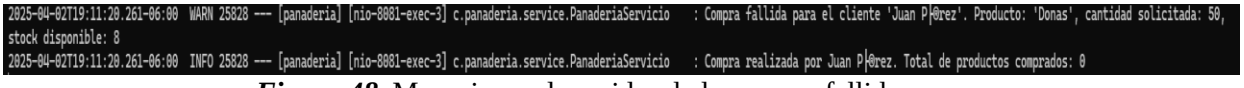


Figura 48. Mensaje en el servidor de la compra fallida.

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├¡z restablecido. Nuevo stock: 622
```

Figura 49. Mensaje en el servidor de la actualización de stock, cada 5 min.

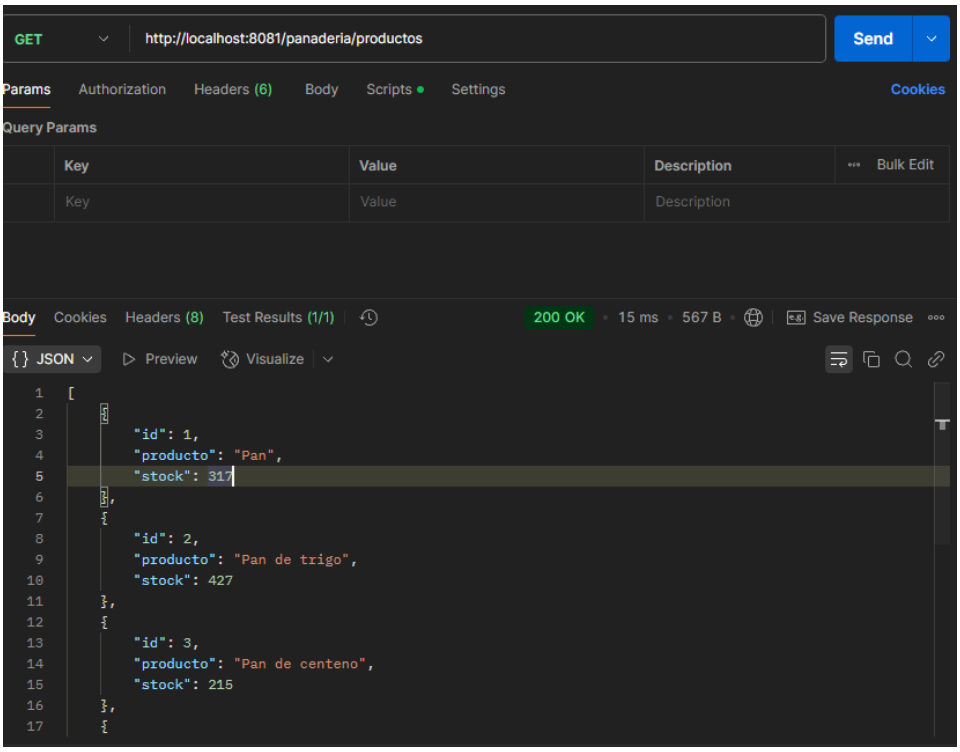


Figura 50. Actualización del stock, tras la simulación del horneo de nuevos panes.

De igual forma, se comprobó que el backend emite correctamente notificaciones en tiempo real mediante la tecnología **Server-Sent Events (SSE)**. Cada vez que se realiza una compra desde Postman (o desde el frontend), es posible visualizar estos eventos accediendo directamente a la URL http://localhost:8081/panaderia/eventos. Esta dirección mantiene una conexión abierta con el servidor, permitiendo recibir mensajes instantáneos sobre acciones relevantes del sistema, como compras realizadas o actualizaciones de stock. Esta funcionalidad fue validada en pruebas simultáneas, donde se mantenía abierta la conexión SSE mientras se ejecutaban peticiones POST desde Postman, observando cómo los eventos eran enviados de forma automática y continua, demostrando así la eficacia del sistema de notificaciones implementado.

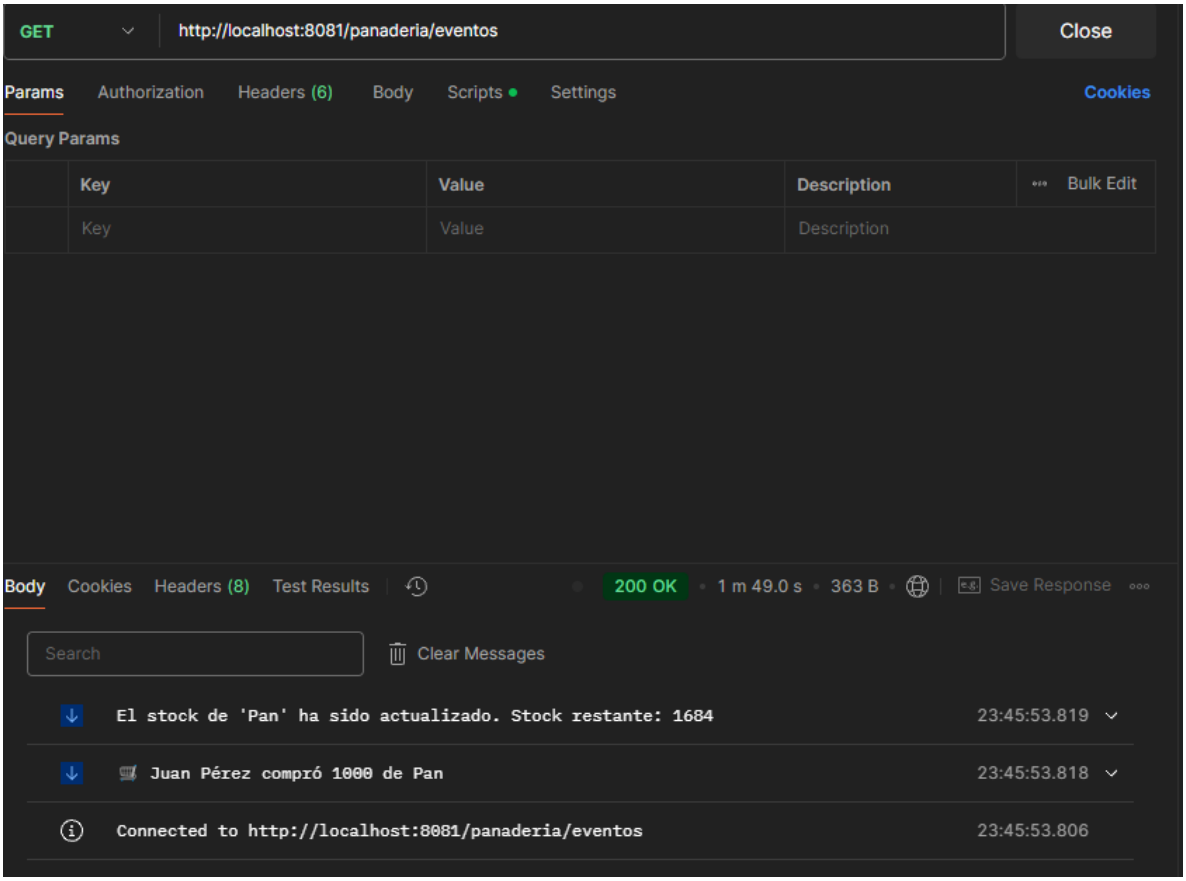


Figura 51. Prueba de las notificaciones desde postman.

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.

Pruebas del frontend (PWA instalada y desde navegador)

Para validar el correcto funcionamiento del frontend como una Progressive Web App (PWA), se realizaron pruebas tanto accediendo desde el navegador como instalando la aplicación en el sistema. Inicialmente, se accedió al servicio a través de la URL http://localhost:8081, lo cual permitió cargar

la interfaz principal del sistema de panadería. Una vez abierto el sitio en el navegador, este detectó automáticamente que la aplicación cumple con las características de una PWA, mostrando una opción para **instalar la aplicación** directamente desde la barra de direcciones o mediante el menú del navegador.

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/, cargando dinámicamente los productos en la interfaz, y permitiendo realizar de igual forma la instalación de la aplicación en el sistema.

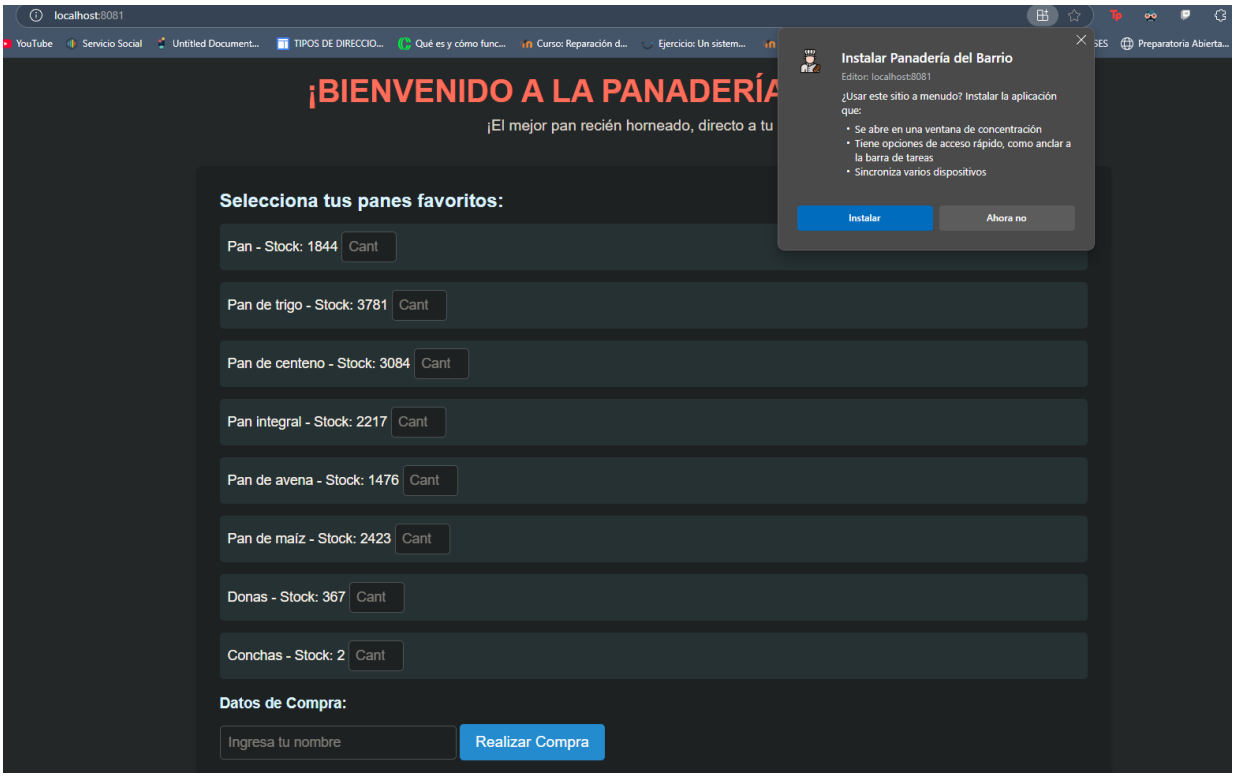


Figura 52. Despliegue de la lista de productos en la página e instalación de la aplicación.

Al proceder con la instalación, la PWA quedó anclada como una aplicación independiente en el sistema operativo, pudiendo ejecutarse desde el escritorio, la barra de tareas o el menú de inicio, sin necesidad de abrir el navegador. Esta instalación ofrece una experiencia más fluida y similar a una aplicación nativa, manteniendo todas las funcionalidades como la visualización de productos, la realización de compras y la recepción de notificaciones en tiempo real.

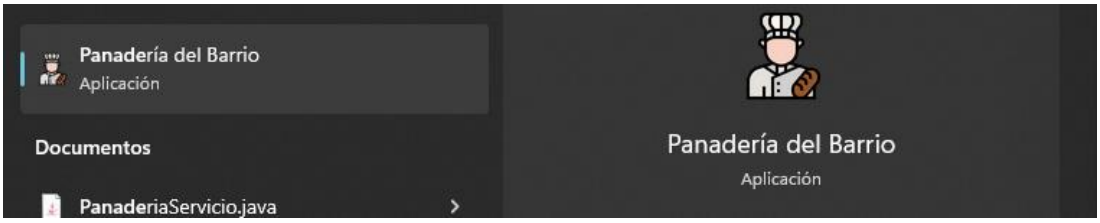


Figura 53. Aplicación instalada en el sistema.

Una vez que abrimos la aplicación instalada, observamos la misma interfaz y funcionalidades que en el sitio web, con la diferencia de que esta se ejecuta de forma independiente al navegador, proporcionando una experiencia más fluida y enfocada. La interfaz muestra correctamente la lista de productos disponibles, el formulario para ingresar el nombre del cliente, la opción de seleccionar cantidades de productos y el botón para confirmar la compra.

Además, al tratarse de una PWA, la aplicación mantiene su funcionamiento incluso sin conexión a internet, gracias a la gestión de caché implementada mediante el Service Worker. Esto permite que los recursos estáticos como el HTML, CSS, imágenes y scripts estén disponibles localmente, garantizando la disponibilidad del servicio en todo momento.

Esta independencia del navegador no solo mejora la experiencia del usuario, sino que también refuerza la percepción de estar utilizando una aplicación nativa, al tiempo que se conservan todas las capacidades de una solución web moderna y dinámica.

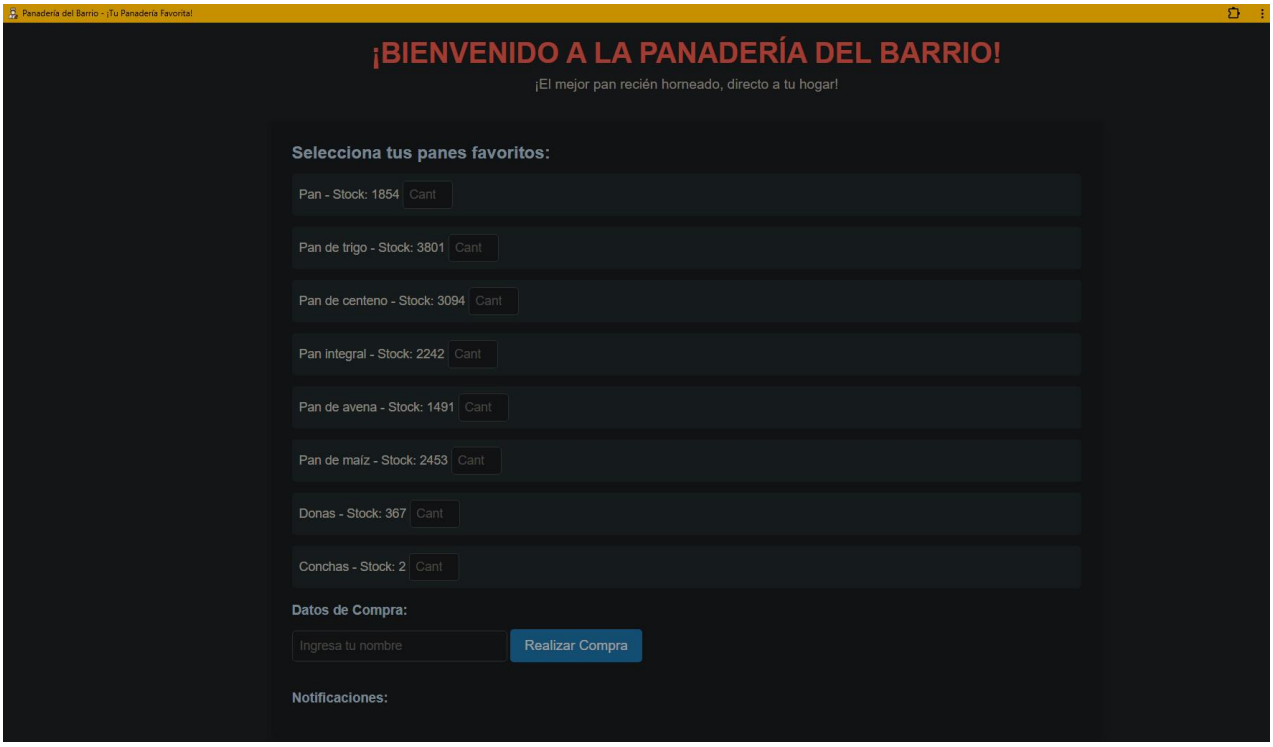


Figura 54. Vista de la aplicación instalada.

Durante las pruebas, se verificó que la aplicación instalada mantiene una correcta comunicación con el backend, carga los recursos necesarios incluso sin conexión a internet (gracias al uso del Service Worker y la caché implementada), y permite operar normalmente desde su entorno aislado.

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.

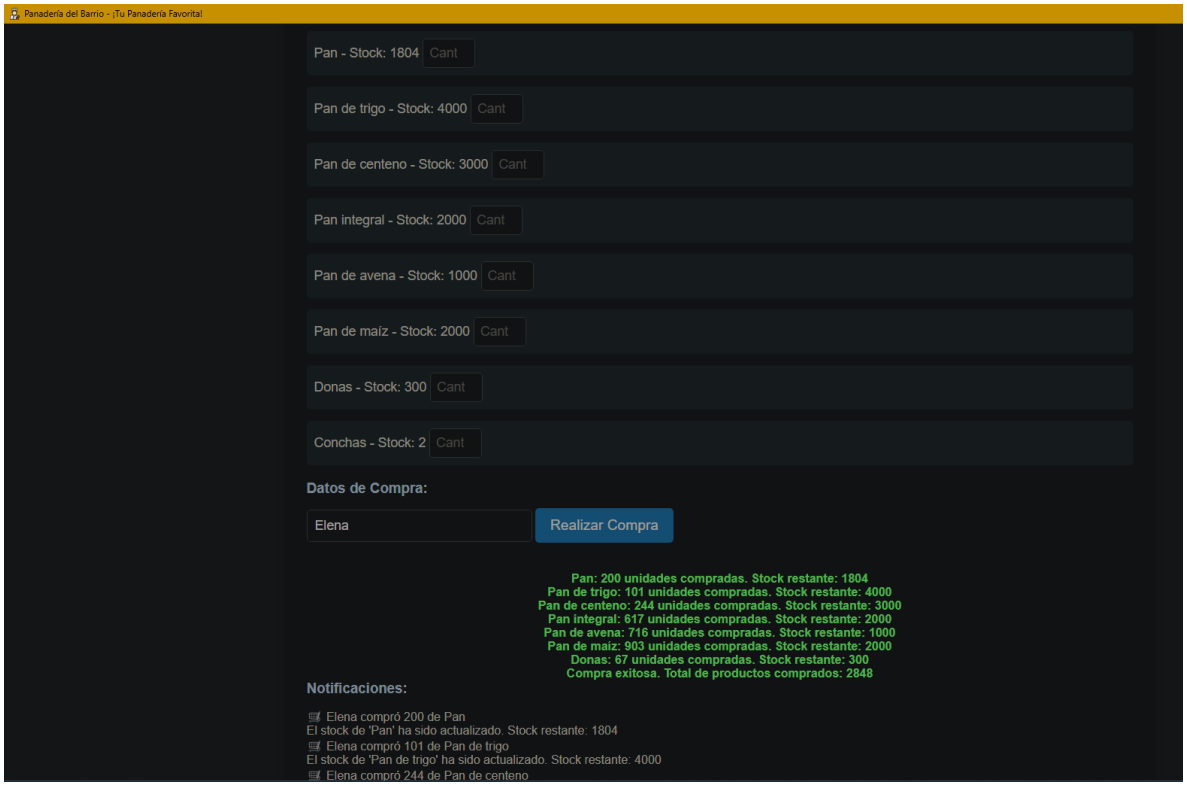


Figura 54. Realización de una compra desde la aplicación.

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

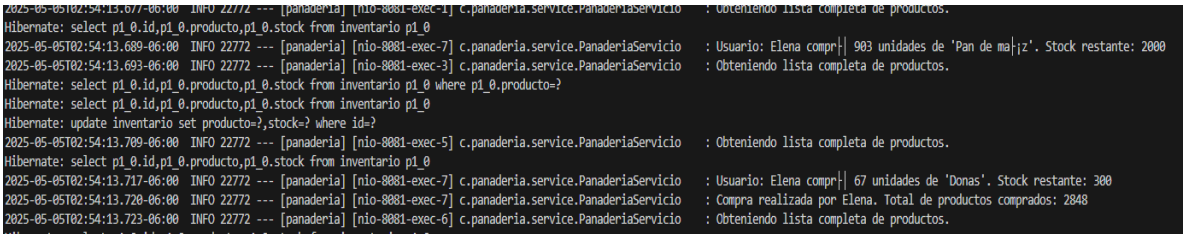


Figura 55. Mensajes del servidor al realizar una compra.

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

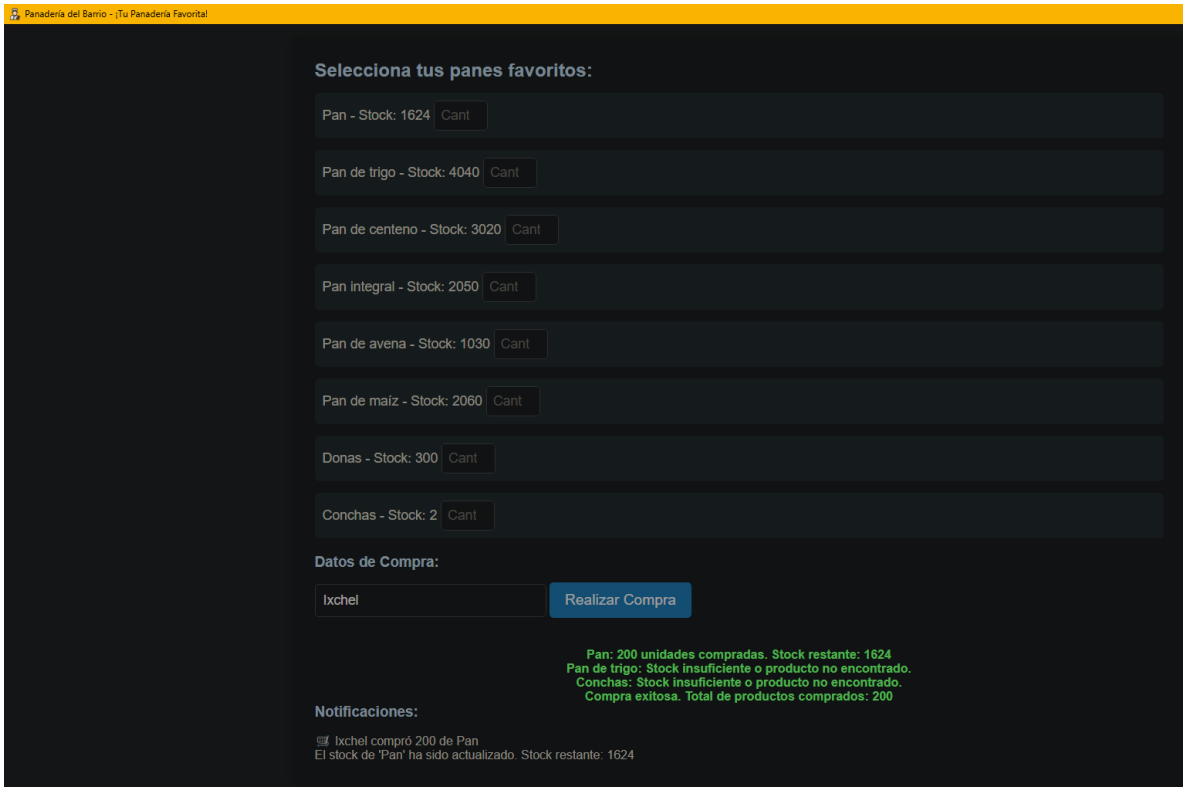


Figura 56. Intento de compra mayor al stock disponible.

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

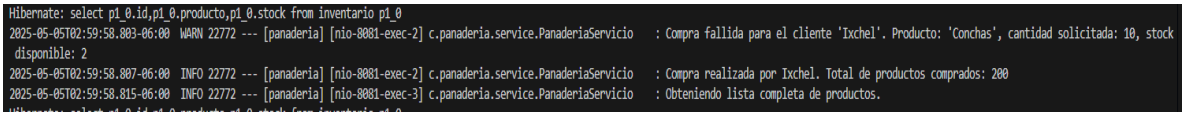


Figura 57. Mensajes en el servidor de la compra fallida.

Tras una compra exitosa, el frontend volvía a solicitar la lista de productos al servidor mediante un nuevo **GET**, reflejando en tiempo real la actualización del stock como se puede observar en las figuras 57 y 58. 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.

En este caso, al hacer una compra y hay otro cliente conectado, el stock se actualiza en tiempo real y a cada cliente se le notifica del cambio en el stock.

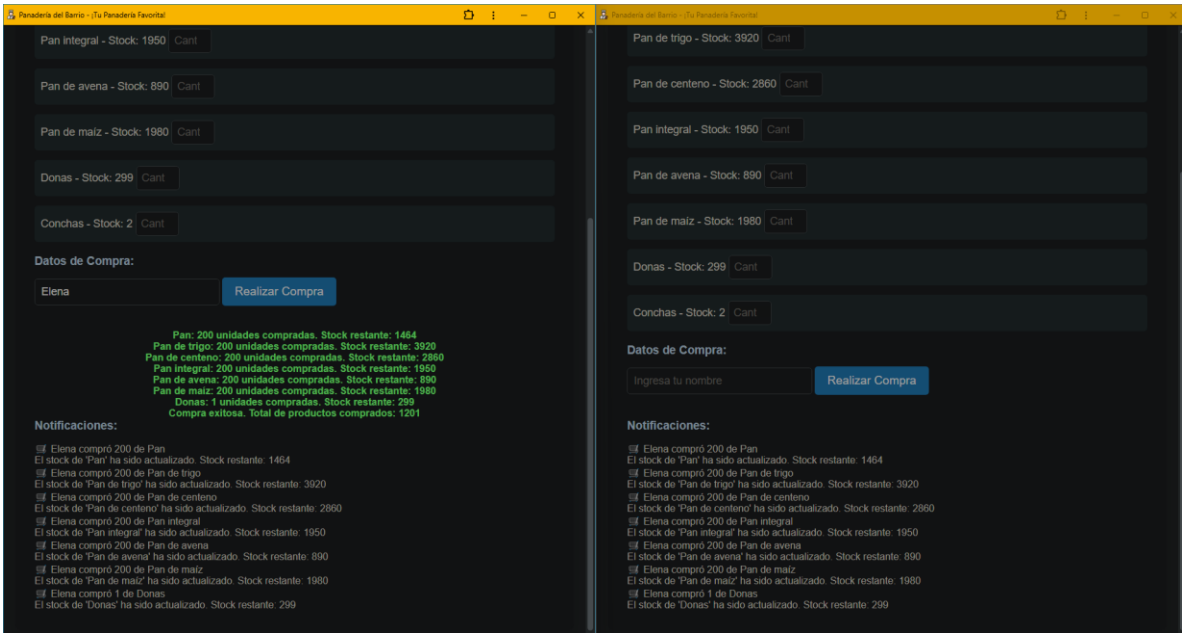


Figura 57. Actualización y notificación de stock cuando un cliente realiza una compra.

De igual forma si realizan compras de manera simultánea, el stock se actualiza correctamente de manera correcta y en tiempo real, además de mostrar en las notificaciones cada evento, como se puede ver en la siguiente figura.

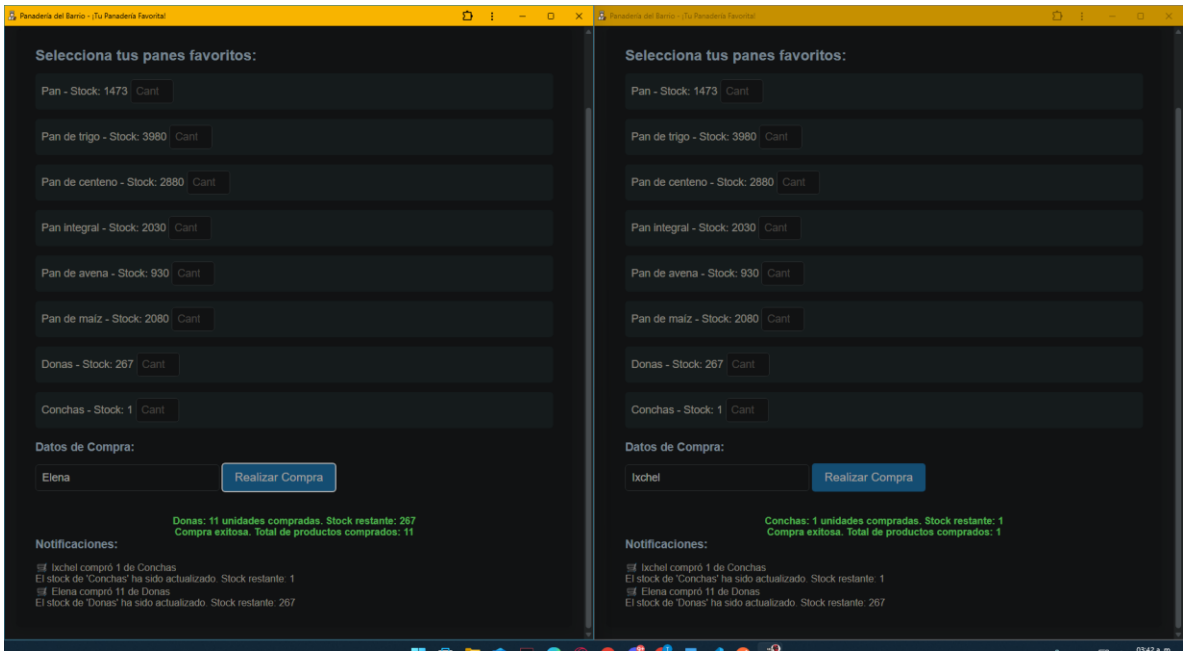


Figura 58. Actualización de stock y notificaciones de cada compra.

De igual forma, en caso de que haya alguna actualización de stock, se le notifica al cliente cada evento, como podemos observar en la siguiente figura:

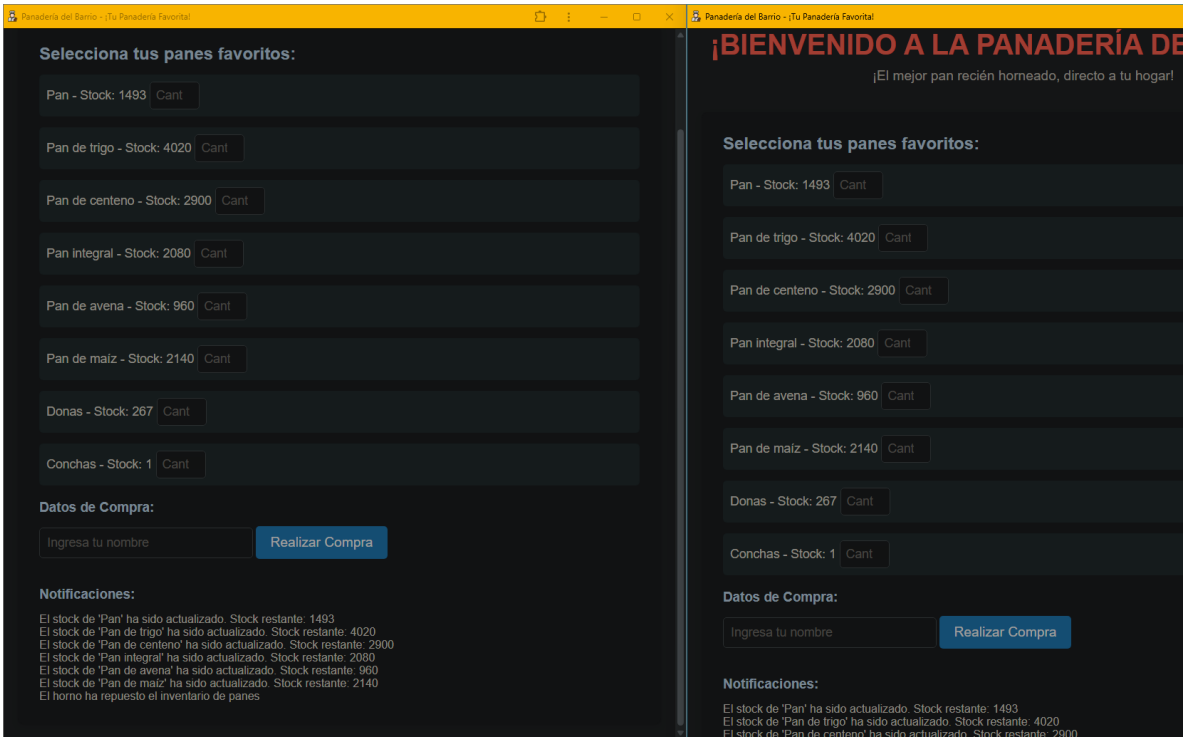


Figura 59. Actualización y notificación de stock.

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 más productos, se le solicita al usuario que debe colocar un nombre.

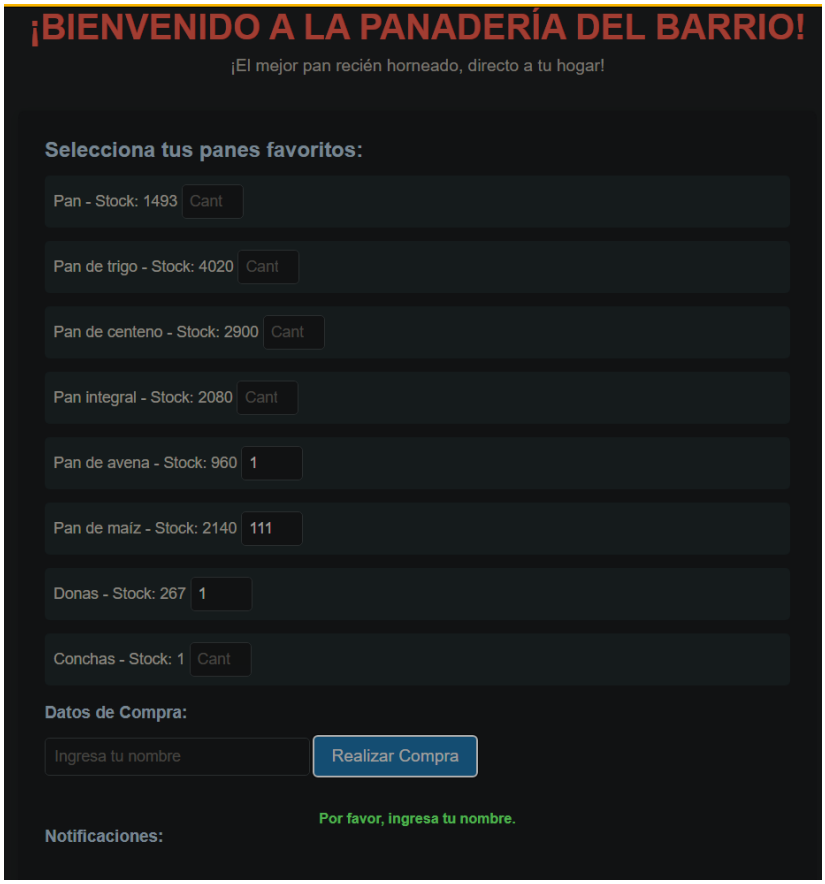


Figura 60. Solicitud al usuario de que seleccione al menos un producto.

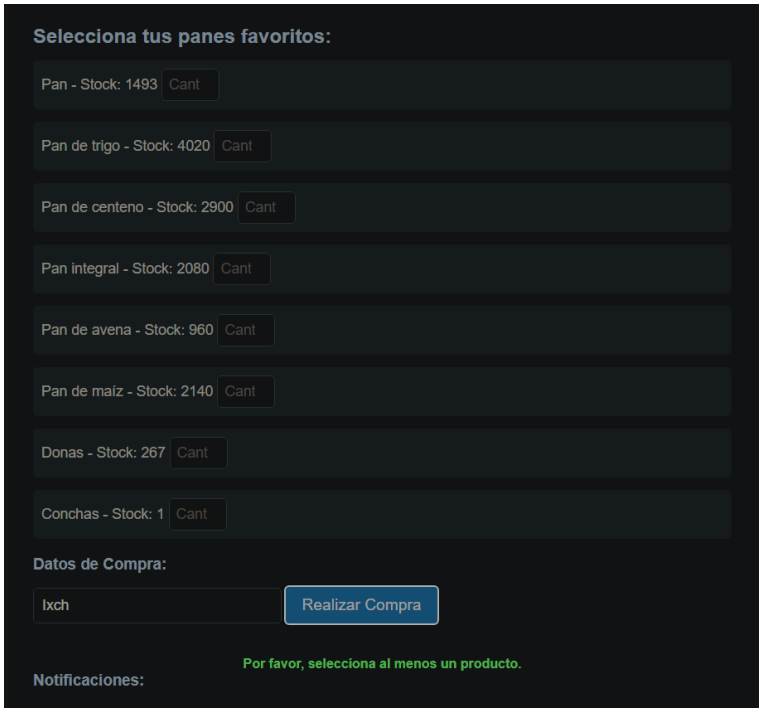


Figura 61. Solicitud al usuario para que ingrese su nombre al realizar una compra.

Gracias al uso de Service Workers, la aplicación mantuvo su funcionalidad incluso sin conexión a internet, permitiendo el acceso a los recursos previamente almacenados en caché. Esto confirmó que la solución cumple con los principios fundamentales de una Progressive Web App: instalabilidad, trabajo sin conexión y comportamiento similar al de una aplicación nativa.

En general, las pruebas realizadas con la PWA confirmaron que el sistema es completamente funcional tanto desde el navegador como desde su versión instalada. La experiencia de usuario es fluida, rápida y consistente, lo que valida el correcto desarrollo de la solución basada en tecnologías web modernas.

Conclusión

Esta práctica me permitió profundizar en el desarrollo de aplicaciones web modernas, integrando un backend robusto en Spring Boot con un frontend adaptable y progresivo mediante tecnologías HTML, CSS y JavaScript. A lo largo del proceso, no solo reforcé mis conocimientos sobre sistemas distribuidos, sino que también exploré conceptos nuevos como las Progressive Web Apps (PWA), logrando implementar una solución instalable, funcional tanto en navegador como como aplicación, y capaz de operar en escenarios reales de uso.

Uno de los aprendizajes más significativos fue la construcción y exposición de una API REST a través de Spring Boot. Comprendí la importancia de estructurar correctamente las rutas, controladores y servicios, así como la necesidad de mantener la coherencia entre la lógica de negocio y el acceso a datos. El uso de una base de datos en memoria facilitó la persistencia temporal del stock, y la incorporación de tareas programadas para simular el horneado de pan me ayudó a entender cómo automatizar procesos repetitivos dentro de un entorno distribuido.

El desarrollo del frontend y su conversión en PWA fue otro aspecto clave. Implementar funcionalidades como la instalabilidad, el almacenamiento en caché y el uso de Service Workers me permitió comprender cómo mejorar la experiencia del usuario, incluso en condiciones de baja o nula conectividad. Aprendí cómo una PWA puede comportarse casi como una aplicación nativa, lo que aporta grandes ventajas en términos de accesibilidad, velocidad y disponibilidad offline. Además, comprendí la importancia del archivo manifest.json y el correcto registro del Service Worker para que la app pudiera instalarse correctamente y mantener sus recursos actualizados.

Cabe destacar que para esta práctica nos apoyamos en lo desarrollado anteriormente en la práctica de servicios web, la cual ya contaba con un backend funcional y una estructura clara para el manejo de productos, compras y stock. Sin embargo, al transformar esa solución en una PWA, nos enfrentamos a nuevos retos, como asegurar que los recursos estáticos se cargaran correctamente, adaptar las rutas para su correcto despliegue en otros dispositivos dentro de la red local, y garantizar que las notificaciones y actualizaciones del stock siguieran funcionando bajo el nuevo enfoque. Aunque al inicio surgieron algunos inconvenientes, como errores de caché, problemas con la instalación o inconsistencias entre versiones, todos fueron solucionados conforme avanzamos, permitiendo que el sistema final se comportara de manera estable y fluida.

Poder observar cómo se reflejan en tiempo real las compras, las notificaciones de eventos mediante SSE, y el stock actualizado tras cada operación, fue una demostración clara del funcionamiento coordinado entre frontend y backend.

En resumen, esta práctica me permitió aplicar de forma integral los conocimientos adquiridos sobre el desarrollo de sistemas web distribuidos. Aprendí no solo a construir y probar un sistema funcional,

sino también a pensar en términos de escalabilidad, usabilidad y experiencia de usuario final. Haber desarrollado una PWA completa desde cero, capaz de gestionar stock, recibir compras y notificar eventos, me brindó una experiencia muy valiosa y actualizada sobre cómo se crean soluciones modernas en el desarrollo de software, además de enseñarme a resolver problemas reales que surgen al migrar una aplicación tradicional a un enfoque más avanzado como el de las Progressive Web Apps.

Bibliografía

- MDN Web Docs, "Progressive Web Apps," 2025. [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps. [Último acceso: 01 Mayo 2025].
- MDN Web Docs, "Caching - Progressive Web Apps," 2025. [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Caching. [Último acceso: 01 Mayo 2025].
- MDN Web Docs, "Making PWAs Installable," 2025. [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Making_PWAs_installable. [Último acceso: 01 Mayo 2025].MDN Web Docs+4MDN Web Docs+4MDN Web Docs+4
- web.dev, "Service Workers," 2025. [En línea]. Disponible en: https://web.dev/learn/pwa/service-workers. [Último acceso: 01 Mayo 2025].
- Microsoft Learn, "Overview of Progressive Web Apps (PWAs)," 2025. [En línea]. Disponible en: https://learn.microsoft.com/en-us/microsoft-edge/progressive-web-apps-chromium/. [Último acceso: 01 Mayo 2025].
- web.dev, "Installation - Learn PWA," 2025. [En línea]. Disponible en: https://web.dev/learn/pwa/installation. [Último acceso: 01 Mayo 2025].
- freeCodeCamp, "What is a PWA? Progressive Web Apps for Beginners," 2025. [En línea]. Disponible en: https://www.freecodecamp.org/news/what-are-progressive-web-apps/. [Último acceso: 01 Mayo 2025].
- Medium, "Best Caching Strategies — Progressive Web App (PWA)," 2025. [En línea]. Disponible en: https://medium.com/animall-engineering/best-caching-strategies-progressive-web-app-pwa-c610d65b2009. [Último acceso: 01 Mayo 2025].
- web.dev, "What does it take to be installable?," 2025. [En línea]. Disponible en: https://web.dev/articles/install-criteria. [Último acceso: 01 Mayo 2025].
- Human Level, "Progressive Web Apps (PWA): qué son y por qué van a mejorar mis visitas," 2025. [En línea]. Disponible en: https://www.humanlevel.com/blog/seo/progressive-web-apps-pwa-que-son-y-por-que-van-a-mejorar-mis-visitas. [Último acceso: 01 Mayo 2025].
- ResearchGate, "Arquitectura de una PWA," 2025. [En línea]. Disponible en: https://www.researchgate.net/figure/Figura-4-Arquitectura-de-una-PWA-Fuente-elaboracion-propia_fig2_353280134. [Último acceso: 01 Mayo 2025].
- MemoCode, YouTube, "Cómo hacer una Progressive Web App (PWA)," 2024. [En línea]. Disponible en: https://www.youtube.com/watch?v=M2X7K2rxm9E&pp=ygUiY29tbyBoYWNlciB1bmEgcHJvZ3Jlc3NpdmUgd2ViIGFwcA%3D%3D. [Último acceso: 01 Mayo 2025].

Anexos PWA

Static/index.html

```
<!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">
    <link rel="manifest" href="manifest.json">
    <meta name="theme-color" content="#f8b400">
</head>
<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"></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>

            <h3>Notificaciones:</h3>
            <div id="notificaciones" class="notificaciones"></div>
        </div>
    </div>

    <script src="script.js"></script>
    <script>
        // Registro del Service Worker
        if ('serviceWorker' in navigator) {
            navigator.serviceWorker.register('service-worker.js')
                .then(reg => console.log('Service Worker registrado', reg))
                .catch(err => console.error('Error al registrar Service
Worker', err));
        }
```

```
    </script>
</body>
</html>
```

Static/script.js

```
const API_URL = "http://192.168.100.12:8081/panaderia";

async function obtenerProductos() {
    try {
        const response = await fetch(`${API_URL}/productos`);
        if (response.ok) {
            const productos = await response.json();
            mostrarProductos(productos);
        } else {
            document.getElementById("mensaje").innerText = "Error al obtener
productos";
        }
    } catch (error) {
        document.getElementById("mensaje").innerText = "No se pudo conectar
al servidor";
    }
}

function mostrarProductos(productos) {
    const productosDiv = document.getElementById("productos");
    productosDiv.innerHTML = "";

    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);
    });
}

async function comprarPan() {
    const nombre = document.getElementById("nombreCliente").value;
```

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

    const compras = [];
    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;
    }

    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),
        });

        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";
    }

    obtenerProductos();
}

// Escuchar eventos del servidor para actualizar stock automáticamente
function iniciarSSE() {
    if (!!window.EventSource) {
        const eventSource = new EventSource(`${API_URL}/eventos`);

        eventSource.onmessage = function (event) {
            const mensaje = event.data;

            const notificaciones =
document.getElementById("notificaciones");
            const p = document.createElement("p");
            p.textContent = mensaje;
            notificaciones.appendChild(p);

            // Recargar toda la lista de productos para reflejar stock
actualizado
            obtenerProductos();

            setTimeout(() => {
                p.remove();
            }, 10000);
        };


        eventSource.onerror = function (error) {
            console.error("Error en SSE:", error);
        };
    } else {
        console.warn("SSE no es compatible con este navegador.");
    }
}

obtenerProductos();
```

```
iniciarSSE();
```

Static/manifest.json

```
{
  "name": "Panadería del Barrio",
  "short_name": "Panadería",
  "start_url": "/",
  "scope": "/",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#ffcc00",
  "icons": [
    {
      "src": "/icons/icon-192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "/icons/icon-512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

Static/service-worker.js

```
const CACHE_NAME = "panaderia-cache-v1";
const urlsToCache = [
  "/",
  "/index.html",
  "/style.css",
  "/script.js",
  "/manifest.json",
  "/icons/icon-192.png",
  "/icons/icon-512.png"
];

// Instalación del Service Worker
self.addEventListener("install", (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME).then((cache) => {
      return cache.addAll(urlsToCache);
    })
  );
```

```
});

// Activación del Service Worker - limpieza de caché antigua
self.addEventListener("activate", (event) => {
  const cacheWhitelist = [CACHE_NAME];
  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (!cacheWhitelist.includes(cacheName)) {
            return caches.delete(cacheName); // Elimina cachés viejas
          }
        })
      );
    })
  );
});

// Manejo de peticiones de red - actualiza la caché en segundo plano
self.addEventListener("fetch", (event) => {
  event.respondWith(
    caches.match(event.request).then((cachedResponse) => {
      const fetchRequest = event.request.clone();

      // Si la respuesta está en caché, la devuelve inmediatamente
      if (cachedResponse) {
        // Actualiza la caché con la nueva respuesta sin interrumpir la
experiencia del usuario
        fetch(fetchRequest).then((response) => {
          caches.open(CACHE_NAME).then((cache) => {
            cache.put(event.request, response.clone());
          });
        });
        return cachedResponse;
      }

      // Si no está en caché, realiza la solicitud normalmente
      return fetch(fetchRequest);
    })
  );
});
```