INSTITUTO POLITÉCTICO NACIONAL

Escuela Superior de Cómputo

Carrera:

Ingeniería en Sistemas Computacionales

Unidad de aprendizaje:

Sistemas Distribuidos

Practica 7:

Progressive Web App

Alumno:

Cardoso Osorio Atl Yosafat

Profesor:

Chadwick Carreto Arellano

Fecha:

06/05/2025

INSTITUTO POLITÉCNICO NACIONAL

ESCOM

Índice de contenido

Índice de figuras

Antecedentes

En la actualidad, el desarrollo web ha evolucionado significativamente, permitiendo crear aplicaciones que ofrecen experiencias similares a las aplicaciones nativas, pero utilizando tecnologías web estándar. Una de las soluciones más innovadoras en este ámbito son las Progressive Web Apps (PWA), que combinan lo mejor del desarrollo web y móvil. Las PWA son aplicaciones accesibles desde un navegador, pero con funcionalidades avanzadas como el trabajo sin conexión, notificaciones push y la posibilidad de ser instaladas en el dispositivo del usuario, gracias al uso de tecnologías como Service Workers, manifest.json y almacenamiento local. 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 son programas que los usuarios utilizan mediante navegadores, accediendo a ellas desde cualquier dispositivo conectado a internet. A diferencia de las aplicaciones de escritorio o móviles, no necesitan ser instaladas en el dispositivo, lo que permite una mayor accesibilidad y facilidad de mantenimiento. En sus inicios, consistían en páginas estáticas construidas únicamente con HTML, sin posibilidades de interacción ni actualización dinámica. Con la incorporación de tecnologías como JavaScript y CSS, estas aplicaciones comenzaron a ofrecer una experiencia más rica, visualmente atractiva e interactiva para el usuario.

Estas aplicaciones pueden cubrir una amplia variedad de propósitos: desde sistemas de gestión empresarial (ERP), aplicaciones educativas y plataformas de comercio electrónico, hasta redes sociales y servicios financieros. Su principal ventaja radica en la ubicuidad y en la facilidad de actualización, ya que los cambios en el software se aplican en el servidor y se reflejan inmediatamente para todos los usuarios.

Un punto de inflexión fue la introducción de AJAX (Asynchronous JavaScript and XML), que permitió actualizar partes específicas de una página sin necesidad de recargar todo el documento. Esto dio paso a la llamada Web 2.0, donde las aplicaciones se volvieron mucho más dinámicas e interactivas, habilitando funcionalidades como chats en tiempo real, sistemas de comentarios en vivo y aplicaciones más reactivas en general.

La posterior evolución hacia HTML5 y la integración de nuevas APIs por parte de los navegadores, como el acceso al almacenamiento local, la geolocalización o las capacidades multimedia, permitió que las aplicaciones web comenzaran a emular funcionalidades tradicionalmente reservadas a aplicaciones de escritorio o móviles. Esta evolución tecnológica sentó las bases para el surgimiento de las Progressive Web Apps (PWA).

A pesar de estos avances, las aplicaciones web tradicionales siguen enfrentando ciertas limitaciones en comparación con las aplicaciones nativas:

Dependencia de conexión a internet: muchas aplicaciones dejan de funcionar parcial o totalmente en entornos offline.

Rendimiento inferior: en general, las apps web no alcanzan el mismo rendimiento que una app nativa, especialmente en tareas intensivas o en dispositivos móviles.

Acceso restringido al hardware: el entorno del navegador impone limitaciones sobre el acceso a funciones como sensores, archivos del sistema, Bluetooth, etc.

Experiencia de usuario más básica: aunque el diseño responsivo ha avanzado, muchas aplicaciones web no logran replicar la fluidez ni la integración de una app móvil instalada.

Estas limitaciones llevaron al desarrollo de las Progressive Web Apps, que buscan cerrar la brecha entre el desarrollo web y el desarrollo móvil nativo.

Rol de las Tecnologías Web en las PWA

Las tecnologías web tradicionales siguen siendo la base para el desarrollo de aplicaciones modernas, incluyendo las PWAs. Estas tecnologías, al combinarse de manera eficiente, permiten crear experiencias de usuario avanzadas, robustas y cada vez más cercanas a las que ofrecen las aplicaciones nativas.

HTML (HyperText Markup Language): es el lenguaje principal que define la estructura de los contenidos en la web. Con la llegada de HTML5, se incorporaron etiquetas semánticas, soporte para audio y video sin plugins, y nuevas APIs para mejorar la funcionalidad del navegador, lo que ha sido esencial para las PWA.

CSS (Cascading Style Sheets): es el lenguaje de diseño que permite definir estilos, colores, fuentes y disposición visual de los elementos. Las mejoras introducidas con CSS3 —como media queries, animaciones y flexbox— han sido clave para lograr interfaces responsivas y adaptables a diferentes dispositivos.

JavaScript: es el lenguaje que permite dar comportamiento dinámico a las aplicaciones. En el contexto de las PWA, JavaScript es fundamental para la implementación de Service Workers, el uso de almacenamiento local (como localStorage, IndexedDB), el manejo de eventos, y la comunicación con servidores mediante Fetch API o WebSockets.

HTTP/HTTPS (HyperText Transfer Protocol): es el protocolo utilizado para la transmisión de datos entre el navegador y el servidor. Para las PWA, es obligatorio el uso de HTTPS, ya que proporciona una capa de seguridad esencial para proteger los datos del usuario y permite el uso de funcionalidades críticas como los Service Workers. La evolución hacia HTTP/2 también ha mejorado la eficiencia en la carga de recursos, permitiendo experiencias más rápidas.

Estas tecnologías son la columna vertebral de cualquier PWA. Su integración adecuada es lo que permite construir aplicaciones que no solo sean funcionales y atractivas, sino también

rápidas, seguras y capaces de operar en condiciones adversas, como en redes lentas o sin conexión.

Progressive Web Apps (PWA)

Las Progressive Web Apps (PWA) son una arquitectura de desarrollo relativamente reciente que busca cerrar la brecha entre las aplicaciones web tradicionales y las aplicaciones móviles nativas. Se definen como aplicaciones construidas con tecnologías web estándar —como HTML, CSS y JavaScript— que, mediante la adopción de ciertas API y buenas prácticas, ofrecen una experiencia de usuario comparable a la de una aplicación nativa, tanto en términos de rendimiento como de funcionalidad.

El término "progresiva" hace referencia a su capacidad de adaptación progresiva: es decir, una PWA debe funcionar para todos los usuarios, independientemente del navegador o dispositivo que estén utilizando, y debe mejorar su funcionalidad a medida que el entorno del usuario lo permita. Esta filosofía de desarrollo se basa en el principio de mejora progresiva (progressive enhancement), que prioriza el acceso básico a la aplicación, incluso en condiciones de baja conectividad o recursos limitados, y añade características avanzadas solo cuando están disponibles.

El concepto de PWA fue promovido principalmente por Google a partir de 2015, y desde entonces ha sido adoptado por múltiples plataformas y navegadores modernos, como Chrome, Firefox, Edge y Safari (aunque con ciertas restricciones en iOS). Las PWAs han ganado popularidad rápidamente, ya que permiten distribuir aplicaciones sin pasar por tiendas oficiales como Google Play o App Store, reduciendo así las barreras de publicación y los costos asociados.

Una de las claves del éxito de las PWAs es que utilizan una serie de tecnologías específicas que les permiten comportarse como aplicaciones nativas. Entre las más importantes se encuentran los Service Workers, el Web App Manifest, y el uso eficiente del almacenamiento en caché. Estas tecnologías, junto con un diseño responsive y buenas prácticas de seguridad (como el uso obligatorio de HTTPS), forman la base técnica sobre la cual se construyen las PWAs.

Desde el punto de vista del usuario, una PWA puede instalarse directamente desde el navegador en la pantalla de inicio, abrirse en modo de pantalla completa sin barra de direcciones, y mantenerse actualizada automáticamente. Además, permite trabajar sin conexión, enviar notificaciones push y cargar con gran rapidez incluso en redes lentas. Todo esto contribuye a una experiencia más fluida, coherente y eficiente.

Desde la perspectiva del desarrollador, construir una PWA no implica aprender tecnologías completamente nuevas, sino aplicar de manera estratégica herramientas y técnicas del desarrollo web moderno. Muchas aplicaciones web existentes pueden convertirse en PWAs

con relativamente pocos cambios, lo que hace que esta arquitectura sea una opción accesible y escalable para proyectos nuevos o ya en producción.

El impacto de las PWAs se ha hecho evidente en casos de uso reales. Empresas como Twitter (con Twitter Lite), Pinterest, Uber y Starbucks han implementado versiones PWA de sus plataformas, obteniendo mejoras significativas en tiempos de carga, retención de usuarios y tasas de conversión, especialmente en mercados emergentes donde las conexiones móviles pueden ser lentas o inestables.

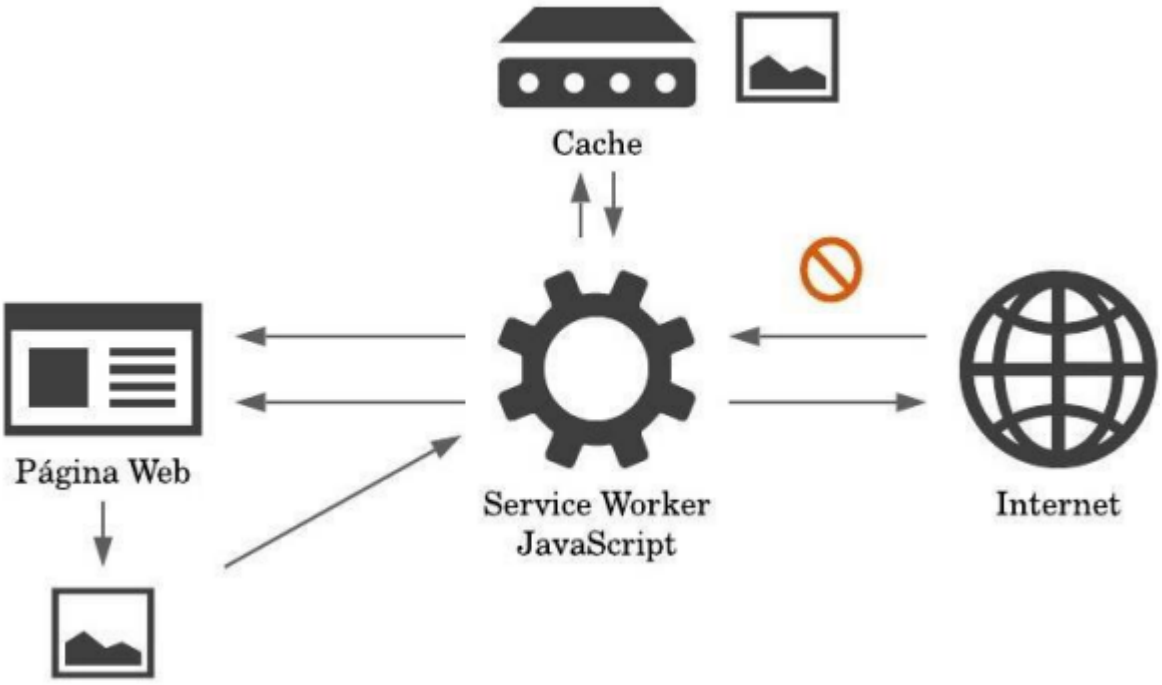


Figura 1. Arquitectura básica de PWA.

Características principales de una PWA

Las Progressive Web Apps se definen por una serie de características esenciales que las distinguen de las aplicaciones web tradicionales:

Progressive (Progresiva): Están diseñadas para funcionar en cualquier navegador moderno, y su funcionalidad mejora progresivamente según las capacidades del dispositivo y del navegador utilizado.

Responsive (Adaptable): Se adaptan a cualquier tamaño de pantalla, ya sea un smartphone, una tableta o un ordenador de escritorio, proporcionando una experiencia de usuario óptima en todos los contextos.

Conectividad independiente: Gracias al uso de Service Workers, una PWA puede funcionar sin conexión o con conexiones inestables, almacenando en caché los recursos clave de la aplicación para garantizar su disponibilidad incluso cuando no hay acceso a internet.

Instalable: Pueden ser añadidas a la pantalla de inicio del dispositivo directamente desde el navegador, sin necesidad de instalar desde una app store. Una vez instaladas, se comportan como una app nativa, con su propio icono, pantalla de inicio y experiencia a pantalla completa.

Actualización automática: Se actualizan en segundo plano mediante el control de los Service Workers, asegurando que el usuario siempre tenga la versión más reciente de la aplicación sin necesidad de realizar actualizaciones manuales.

Seguridad: El uso obligatorio de HTTPS garantiza que las comunicaciones entre el servidor y el cliente sean seguras, protegiendo la integridad de los datos y evitando manipulaciones maliciosas.

Re-engagement (Reenganche del usuario): Las PWAs permiten el envío de notificaciones push, lo que facilita mantener al usuario informado y comprometido, incluso cuando la aplicación no está en uso.

Descubribilidad: Al ser indexadas por los motores de búsqueda, las PWAs pueden ser encontradas e instaladas directamente desde los resultados de búsqueda, lo que amplía su alcance sin depender de tiendas de aplicaciones.

Estas características hacen que las PWAs combinen la accesibilidad de una página web con la funcionalidad de una aplicación móvil, ofreciendo una solución eficaz, ligera y moderna tanto para desarrolladores como para usuarios.

Diferencias entre PWA, aplicaciones nativas e híbridas

Para comprender mejor sus diferencias, a continuación se presenta una tabla comparativa que destaca los aspectos más relevantes entre estas tres opciones de desarrollo.

Criterio	Progressive Web App (PWA)	Aplicación Nativa	Aplicación Híbrida
Tecnologías utilizadas	HTML, CSS, JavaScript, Service Workers, Web APIs	Java/Kotlin (Android), Swift/Objective-C (iOS)	HTML, CSS, JavaScript + frameworks (Ionic, Cordova, etc.)

Instalación	Desde el navegador, sin tienda de apps	Desde tiendas de aplicaciones (App Store, Google Play)	Desde tiendas de aplicaciones
Acceso a hardware del dispositivo	Limitado (según navegador y permisos)	Completo	Parcial (según plugins y plataforma)
Funcionamiento offline	Sí, mediante Service Workers	Sí	Sí (depende de la implementación)
Actualización	Automática, controlada por el navegador	Manual o automática desde la tienda	Manual o automática desde la tienda
Rendimiento	Bueno, pero dependiente del navegador	Alto rendimiento, optimizado para el sistema operativo	Intermedio (menos eficiente que una app nativa)
Distribución	Web (URL) y opcionalmente instalable desde el navegador	Mediante tiendas de aplicaciones	Mediante tiendas de aplicaciones
Desarrollo multiplataforma	Sí, una sola base de código para todos los dispositivos	No, se necesita una versión por plataforma	Sí, una sola base de código
Costo de desarrollo	Bajo a medio	Alto (requiere desarrollo separado por plataforma)	Medio
Revisión por tienda de apps	No requiere	Requiere (proceso de revisión y aprobación)	Requiere
Experiencia de usuario (UX)	Muy buena (puede imitar apps nativas, pero con limitaciones)	Excelente (interfaz y rendimiento completamente integrados)	Buena, pero a veces inconsistente con el sistema operativo

PWA en el Contexto de Sistemas Distribuidos

Un sistema distribuido se caracteriza por estar compuesto por múltiples computadoras independientes que, juntas, operan como un sistema único y cohesionado desde la perspectiva del usuario. Estas computadoras colaboran para llevar a cabo tareas complejas, compartiendo recursos y cooperando entre sí, todo mientras los detalles de su distribución permanecen ocultos. En este marco, una Progressive Web App (PWA) puede considerarse una parte integral de un sistema distribuido debido a varias características técnicas y funcionales que se alinean con las necesidades y principios de este tipo de sistemas.

En primer lugar, la interacción cliente-servidor es uno de los pilares fundamentales de una PWA en un sistema distribuido. Las PWAs actúan como clientes dentro de una arquitectura distribuida, comunicándose con servidores remotos a través de APIs REST o GraphQL, que habitualmente se exponen mediante protocolos HTTP/HTTPS. Esta interacción permite que el cliente (la PWA) envíe solicitudes de datos y reciba respuestas del servidor, facilitando la distribución de la lógica y el almacenamiento de datos de manera centralizada y eficiente.

Además, las PWAs suelen integrar almacenamiento y procesamiento distribuido. Muchas de estas aplicaciones están diseñadas para interactuar con servicios en la nube, como Firebase, AWS, o soluciones personalizadas, que ofrecen servicios como bases de datos, almacenamiento de archivos, autenticación de usuarios y funciones en la nube. Estos servicios están distribuidos geográficamente, lo que asegura que la aplicación pueda escalar de manera eficiente, gestionando grandes cantidades de datos y ofreciendo servicios de alta disponibilidad.

El desacoplamiento de componentes también es clave en la arquitectura moderna de sistemas distribuidos, y las PWAs se alinean con este enfoque. Una PWA puede consumir microservicios distribuidos, cada uno con responsabilidades específicas, desplegados en diferentes entornos. Esta modularidad permite que los componentes del sistema se desarrollen, despliegen y mantengan de manera independiente, lo que aumenta la flexibilidad, la escalabilidad y la facilidad de actualización del sistema en general.

Por otro lado, una de las grandes ventajas de las PWAs en el contexto de sistemas distribuidos es su capacidad para garantizar tolerancia a fallos y replicación. Gracias a técnicas como la caché, el funcionamiento offline y la sincronización en segundo plano, las PWAs mejoran la disponibilidad y resiliencia del sistema, permitiendo que los usuarios sigan interactuando con la aplicación incluso cuando no tienen conexión a Internet o cuando hay desconexiones temporales. Esta capacidad de operar en condiciones de red inestables es especialmente útil en escenarios distribuidos, donde la conectividad entre nodos no siempre es constante.

Finalmente, las actualizaciones independientes son otra característica clave que las PWAs aportan a los sistemas distribuidos. A diferencia de las aplicaciones tradicionales, que requieren la redistribución del cliente cada vez que se realiza una actualización, las PWAs

pueden recibir actualizaciones directamente desde el backend sin necesidad de que el usuario vuelva a descargar la aplicación. Esto asegura que el sistema se mantenga actualizado en todo momento, mejorando la eficiencia operativa y la experiencia del usuario.

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

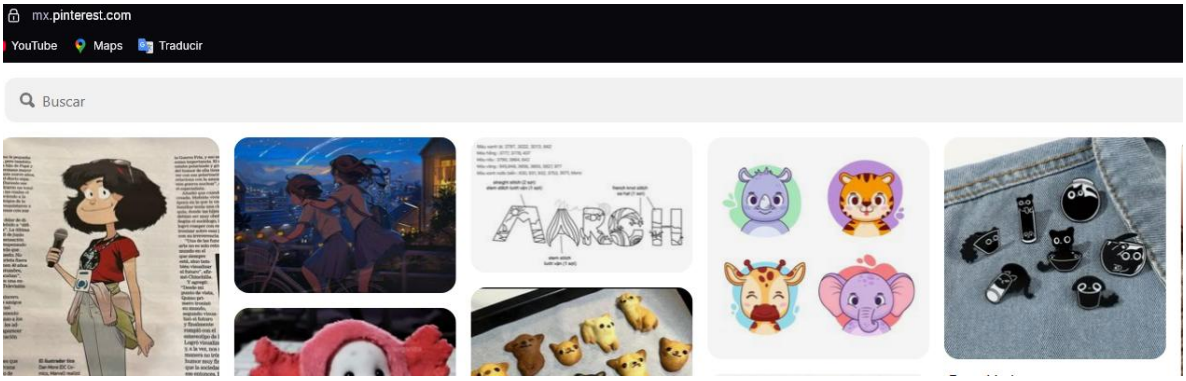


Figura 3. PWA de Pinterest.

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.

Servicios Web RESTful

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

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

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

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

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

- Soporte para HATEOAS (Hypermedia as the Engine of Application State): Este principio sugiere que un cliente debe poder descubrir dinámicamente las acciones disponibles a través de enlaces dentro de las respuestas del servicio web.

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

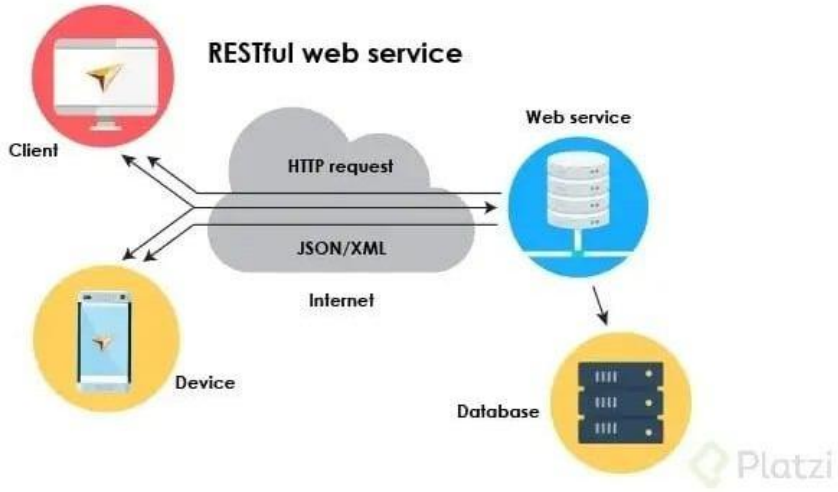


Figura 2. Estructura básica de un servicio web REST

Métodos HTTP Utilizados en una API REST

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

1. GET (Obtener Datos)

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

2. POST (Crear un Nuevo Recurso)

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

3. PUT (Actualizar un Recurso Existente)

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

4. DELETE (Eliminar un Recurso)

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

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

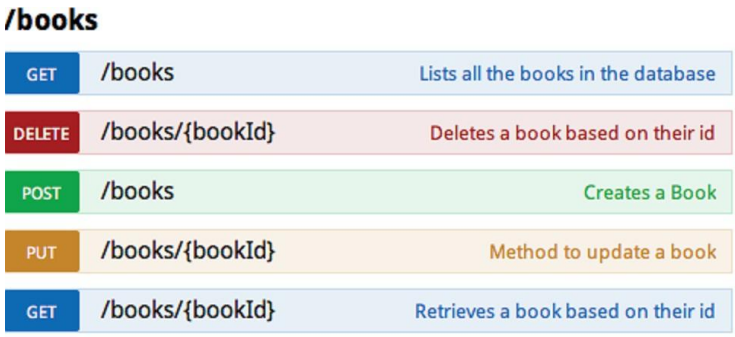


Figura 3. Ejemplo del uso de métodos HTTP

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

Diferencias entre REST y SOAP

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

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

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

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

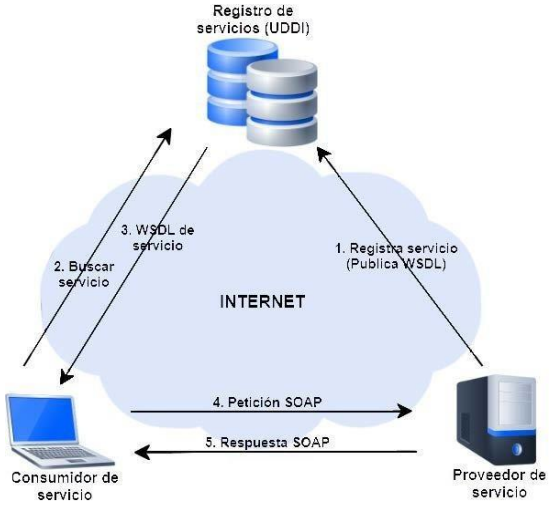


Figura 4. Estructura básica de un servicio web SOAP

Protocolos de comunicación

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

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

HTTP (Hypertext Transfer Protocol): Ampliamente utilizado en la web, HTTP define cómo se comunican los navegadores web y los servidores. Establece un formato para las solicitudes y respuestas, permitiendo la transferencia de recursos como páginas web, imágenes y otros archivos.

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

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

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

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

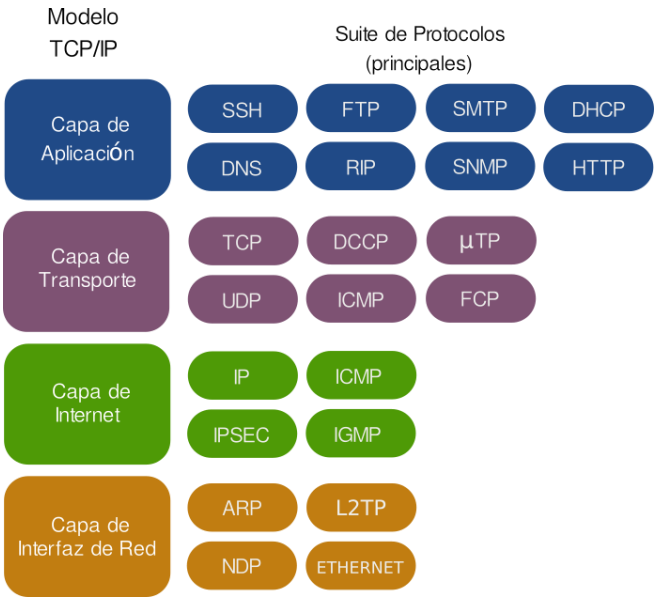


Figura 5. Diagrama de protocolos de comunicación

Planteamiento del problema

En esta práctica, el objetivo principal es desarrollar una Aplicación Web Progresiva (PWA) que facilite la gestión de ventas de boletos para un cine, incorporando tecnologías del lado del cliente como HTML, CSS, JavaScript, y un backend en Node.js con MySQL. La aplicación está diseñada para simular un sistema en el que los usuarios pueden seleccionar películas, realizar compras de boletos, y consultar ventas previas, todo en un entorno interactivo y dinámico.

Una de las problemáticas centrales que enfrenta esta práctica es cómo asegurar la correcta sincronización de datos en un entorno de múltiples usuarios concurrentes. En un sistema real, los boletos disponibles para cada película son limitados, y las compras de los usuarios deben reflejarse de manera precisa y en tiempo real. En este contexto, si dos o más usuarios intentan comprar el mismo boleto de una película simultáneamente, se corre el riesgo de que el sistema registre más ventas de las que el inventario realmente permite, generando frustración en los clientes y posibles inconsistencias en el stock de boletos. Es necesario implementar un control adecuado sobre el manejo del inventario para evitar la venta de boletos agotados, lo que exige la integración de operaciones atómicas y validaciones de stock en tiempo real.

Otro reto importante es la notificación en tiempo real. En un sistema tradicional, los usuarios no recibirían actualizaciones inmediatas sobre el estado del inventario, lo cual es crucial para una experiencia de usuario fluida. Al integrar funcionalidades como notificaciones push y la utilización de un Service Worker para la gestión de eventos en segundo plano, se busca permitir que los usuarios reciban alertas de cambios importantes, como la disponibilidad de boletos o el agotamiento de un filme. Esto proporciona una capa adicional de interactividad y mejora la experiencia de compra en línea.

Un aspecto adicional que plantea desafíos es la necesidad de mantener la aplicación funcional sin conexión. En ambientes móviles, los usuarios suelen experimentar fluctuaciones en la calidad de la conexión a internet, por lo que una de las funcionalidades clave de esta aplicación es su capacidad de operar sin conexión a la red. Para ello, se ha utilizado un Service Worker que permite almacenar en caché los recursos esenciales y garantizar que los usuarios puedan seguir interactuando con la aplicación incluso en condiciones de conectividad limitada.

También el manejo automático de procesos como la actualización periódica del stock de boletos en función de las compras realizadas es un desafío que debe resolverse sin intervención manual. Para garantizar que el inventario refleje siempre el estado más actualizado, la aplicación cuenta con un sistema de actualización en segundo plano, que sincroniza los datos entre el cliente y el servidor a intervalos regulares. El desarrollo de esta aplicación PWA busca abordar estos problemas y ofrecer una experiencia de usuario moderna, eficiente y robusta, adecuada para su uso en cualquier dispositivo y en diversas condiciones de conectividad.

Propuesta de solución

La propuesta de solución para esta práctica se centra en el desarrollo de una Aplicación Web Progresiva (PWA) que permita gestionar la compra de boletos de cine de manera eficiente, asegurando la correcta sincronización del inventario, la modificación en tiempo real y la funcionalidad offline, todo ello implementado con tecnologías modernas y robustas. La solución propuesta aborda los problemas identificados en el planteamiento de forma integral, utilizando una combinación de front-end interactivo y back-end eficiente.

Para resolver el problema de la gestión simultánea de compras por parte de múltiples usuarios, se implementa un control de inventario en el backend mediante una base de datos MySQL. Cada vez que un usuario realiza una compra, se valida el stock de boletos disponible antes de procesar la transacción. Este proceso se lleva a cabo mediante una consulta SQL para verificar la cantidad de boletos disponibles para la película seleccionada y se actualiza en tiempo real en la base de datos.

El servidor usando Node.js gestiona las solicitudes de compra mediante un endpoint POST que valida el stock antes de permitir la transacción. Si el stock es insuficiente, el sistema informará al usuario sobre la imposibilidad de completar la compra. Esta validación de stock asegura que el sistema nunca permita vender más boletos de los disponibles, manteniendo la integridad de los datos.

Por otro lado, para garantizar la funcionalidad de la aplicación en condiciones de conectividad limitada o intermitente, se utiliza el enfoque propio de las PWA, donde los recursos clave como los archivos HTML, CSS y JS se almacenan en la caché del navegador a través del Service Worker. Esto permite a los usuarios seguir interactuando con la aplicación incluso sin conexión a Internet, siempre que hayan cargado previamente los datos esenciales.

Además, el Service Worker maneja la recuperación de datos y las páginas de respaldo en caso de que no se pueda acceder a la red. Por ejemplo, si el usuario intenta acceder a una página cuando está sin conexión, se le redirige a una página de respaldo previamente almacenada en la caché como una página de "offline", brindándole una experiencia de usuario más fluida.

También para mantener el inventario y las ventas actualizados, se propone hacer que la aplicación realiza consultas periódicas al servidor. Cada cierto tiempo, el cliente solicita al servidor los datos más recientes sobre las ventas y el inventario, de manera que el sistema refleje en tiempo real los cambios en el stock de boletos y las ventas realizadas. Esto asegura que los usuarios siempre tengan acceso a la información más precisa al momento de realizar sus compras.

Por último, se propone realizar una interfaz que se centre en la simplicidad y la facilidad de uso. El uso de tecnologías como HTML, CSS y JavaScript en el front-end, junto con la integración del Service Worker, garantiza que la aplicación no solo sea rápida y eficiente, sino también accesible, incluso en condiciones de conectividad inestable.

Materiales y métodos empleados

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

Materiales

1. *Lenguaje de programación:* ***JavaScript***

El lenguaje JavaScript fue seleccionado para desarrollar la lógica del cliente y del servidor, gracias a su versatilidad y amplia adopción en el desarrollo web. En el lado del cliente, JavaScript permite la creación de interfaces interactivas, mientras que, en el servidor, con la ayuda de Node.js, se gestionan las solicitudes y la comunicación con la base de datos.

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

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

3. *Framework Backend:* ***Node.js y Express***

Para el backend, emplee Node.js con el framework Express. Node.js ofrece un entorno asíncrono y de alto rendimiento, adecuado para aplicaciones que requieren alta concurrencia, como en este caso, la compra simultánea de boletos. Express se utilizó para gestionar las rutas y servicios RESTful de manera eficiente.

4. *Base de Datos:* ***MySQL***

La base de datos MySQL fue utilizada para almacenar la información sobre las películas, el inventario de boletos y las ventas realizadas. Esta base de datos relacional garantiza la consistencia de los datos y permite consultas rápidas para verificar el stock y registrar nuevas compras.

5. ***Service Worker para PWA***

Se implementó un Service Worker para gestionar la funcionalidad offline de la aplicación. Este componente intercepta las solicitudes de red y permite almacenar en caché los recursos

esenciales de la aplicación, garantizando que los usuarios puedan seguir interactuando con la aplicación incluso sin conexión a internet.

6. ***Manifesto de la Aplicación***

Se utilizó un archivo manifest.json para definir las propiedades de la aplicación web progresiva, como el nombre, los íconos, la apariencia en la pantalla de inicio y la configuración de pantalla completa. Este archivo es esencial para que la aplicación sea reconocida y utilizada como una PWA.

7. *Frontend:* ***HTML, CSS, JavaScript***

El diseño de la interfaz de usuario se construyó utilizando HTML para la estructura de la página, CSS para la presentación y JavaScript para la lógica interactiva en el lado del cliente. Estos lenguajes permiten crear una experiencia de usuario atractiva y funcional.

8. *Sistema Operativo:* ***Windows***

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

Métodos

Para la implementación del sistema distribuido basado en PWA, se llevaron a cabo diversas metodologías y técnicas que permitieron desarrollar una solución escalable, modular, eficiente y tolerante a fallos. A continuación, se detallan los métodos utilizados:

Arquitectura Cliente-Servidor

La solución se construyó bajo el paradigma de arquitectura distribuida cliente-servidor. En este modelo, el cliente se encarga de la interacción directa con el usuario a través de una interfaz web, mientras que los servidores procesan las solicitudes de manera remota y ejecutan las operaciones correspondientes. En el lado del cliente, la interfaz web permite a los usuarios seleccionar películas, ingresar sus datos y realizar compras de boletos, mientras que el servidor maneja las solicitudes de compra, la validación de stock y el registro de ventas.

Comunicación RESTful

Para la comunicación entre el cliente y el servidor, se utilizó RESTful APIs. El cliente realiza solicitudes GET para consultar la disponibilidad de películas y POST para realizar las compras de boletos. El servidor responde con datos en formato JSON, lo cual facilita la integración con el frontend y asegura una comunicación rápida y sencilla entre las distintas capas del sistema.

Sincronización de Inventario

Un desafío principal en el desarrollo de esta aplicación fue garantizar la correcta sincronización del inventario de boletos cuando múltiples usuarios intentan comprar al mismo tiempo. En el backend, se implementaron métodos para verificar el stock de boletos antes de procesar cada compra, asegurando que las transacciones se manejen de manera atómica. Cuando un usuario compra boletos, el sistema actualiza el inventario de forma inmediata para reflejar la nueva disponibilidad.

Notificaciones en Tiempo Real

Para mantener a los usuarios informados sobre el estado del inventario, se implementó un sistema de notificaciones en tiempo real utilizando el Service Worker. Además de permitir la funcionalidad offline, el Service Worker gestiona la actualización en segundo plano de los datos sobre cambios importantes, como la disponibilidad o agotamiento de boletos.

Funcionamiento Offline y Cacheo

Para asegurar que la aplicación sea funcional incluso en entornos con conectividad limitada, se configuró el Service Worker para almacenar en caché los recursos esenciales de la aplicación. Esto incluye archivos estáticos como HTML, CSS, JavaScript e imágenes, lo que permite que la aplicación continúe funcionando sin conexión a internet. Además, el Service Worker intercepta las solicitudes de red, ofreciendo una página de respaldo en caso de que no se pueda acceder a la red.

Gestión del Estado en el Cliente

Para mantener actualizada la interfaz de usuario, el cliente realiza consultas periódicas al servidor utilizando setInterval. Esta técnica permite que el cliente obtenga información actualizada sobre las ventas y el inventario cada cierto tiempo. Así, los usuarios siempre tienen acceso a los datos más recientes sin necesidad de recargar manualmente la página.

Servidor de Backend y API

El backend se implementó en Node.js con el framework Express. Este servidor gestiona las solicitudes de los usuarios, valida las compras, actualiza el inventario en la base de datos MySQL y responde a las consultas sobre películas y ventas. Además, se utilizaron rutas RESTful para manejar las peticiones de compra de boletos y el registro de ventas.

Validación de Datos

Se implementaron validaciones en el lado del cliente para garantizar que los datos ingresados sean correctos antes de enviarlos al servidor. Esto incluye la verificación de que los campos como el nombre del cliente, la cantidad de boletos y la selección de la película sean completos y válidos, evitando solicitudes erróneas que podrían afectar el funcionamiento de la PWA.

Desarrollo

En esta práctica se desarrolló una Aplicación Web Progresiva (PWA) para gestionar la venta de boletos para un cine, utilizando tecnologías modernas como HTML, CSS, JavaScript, y Node.js para el backend, junto con MySQL para la base de datos. El propósito principal de esta PWA es ofrecer una experiencia de usuario fluida y accesible, permitiendo la compra de boletos en tiempo real, notificaciones de disponibilidad de cartelera y el correcto funcionamiento incluso sin conexión a internet, es decir la posibilidad de poder descargar la PWA.

Instalación de Node.js

Primeramente, para desarrollar el servidor que gestionaría las solicitudes de la aplicación web, se decidió utilizar **Node.js**, un entorno de ejecución para JavaScript del lado del servidor, que es altamente eficiente y adecuado para aplicaciones web escalables. A continuación, describo el proceso seguido para instalar Node.js y configurar el servidor.

El primer paso fue descargar Node.js desde su página oficial https://nodejs.org. En mi caso, seleccioné la versión LTS (Long Term Support), ya que es más estable y recomendada para la mayoría de los proyectos. Después de descargar el instalador adecuado para Windows, lo ejecuté y seguí las instrucciones en pantalla para completar la instalación. El proceso fue sencillo y rápido, y como parte del instalador, también se incluyó npm (Node Package Manager), que es una herramienta esencial para la gestión de paquetes en proyectos de Node.js.

Figura 6. Descarga e instalación de Node.js

Una vez que Node.js estaba instalado, verifiqué que la instalación se hubiera realizado correctamente abriendo la terminal y ejecutando los siguientes comandos, que se muestran en la imagen:

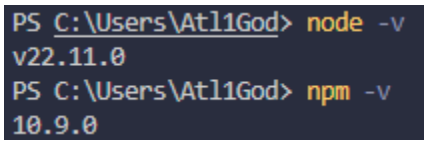


Figura 7. Verificación de las versiones instaladas de Node.js y npm

Estos comandos me devolvieron las versiones de Node.js y npm que había instalado, confirmando que ambos se habían instalado correctamente.

Configuración de la Base de Datos

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

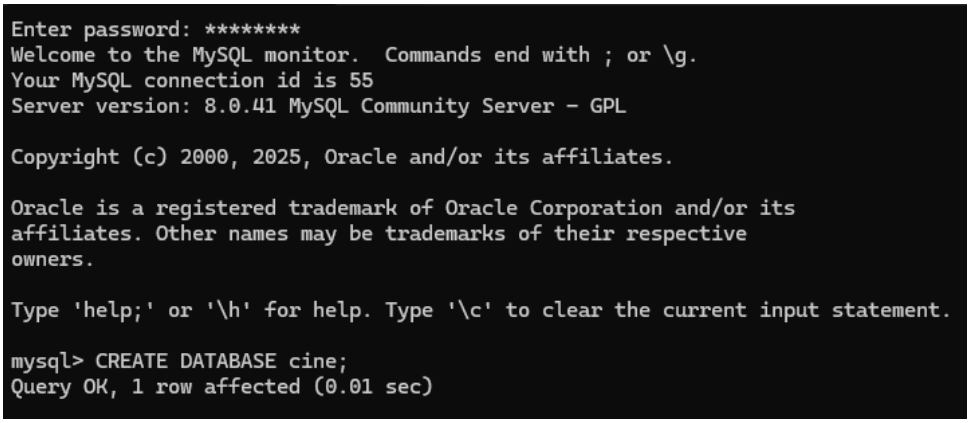


Figura 8. Creación de la base de datos cineboletos

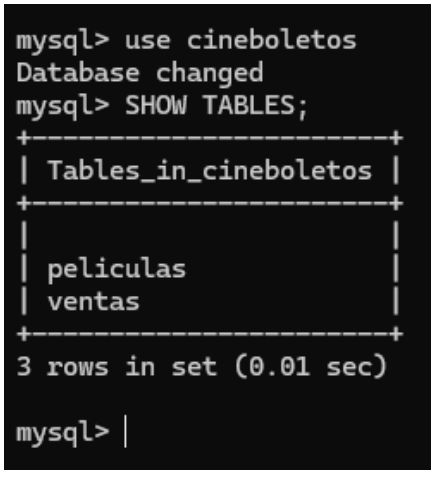


Figura 9. Tablas creadas en la base de datos

Posteriormente se crearon las diversas tablas y se le agregaron los datos correspondientes, esto para realizar pruebas y para verificar el correcto funcionamiento de la PWA en posteriores puntos.

```
mysql> SELECT * FROM peliculas;
+----+-------------+-------+-------------------------------------+
| id | nombre      | stock | descripcion                         |
+----+-------------+-------+-------------------------------------+
|  1 | Película A  |     2 | Una increíble película de acción    |
|  2 | Película B  |     0 | Un drama emocionante                |
|  3 | Película C  |    28 | Comedia para toda la familia        |
+----+-------------+-------+-------------------------------------+
3 rows in set (0.00 sec)

mysql> describe peliculas
    -> ;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| id          | int          | NO   | PRI | NULL    | auto_increment |
| nombre      | varchar(255) | YES  |     | NULL    |                |
| stock       | int          | YES  |     | NULL    |                |
| descripcion | text         | YES  |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

Figura 10. Datos y estructura de la tabla películas

```
mysql> SELECT * FROM ventas;
+----+----------------+----------+-------------+---------------------+
| id | nombre_cliente | cantidad | pelicula    | fecha               |
+----+----------------+----------+-------------+---------------------+
| 26 | Atl            |        5 | Película C  | 2025-05-07 22:15:23 |
| 27 | Atl            |        5 | Película C  | 2025-05-07 22:15:26 |
| 28 | Atl            |        1 | Película A  | 2025-05-07 22:15:30 |
| 29 | Atl            |        1 | Película C  | 2025-05-07 22:15:32 |
| 30 | Atl            |        1 | Película C  | 2025-05-07 22:15:35 |
+----+----------------+----------+-------------+---------------------+
5 rows in set (0.00 sec)

mysql> describe ventas;
+----------------+--------------+------+-----+-------------------+-------------------+
| Field          | Type         | Null | Key | Default           | Extra             |
+----------------+--------------+------+-----+-------------------+-------------------+
| id             | int          | NO   | PRI | NULL              | auto_increment    |
| nombre_cliente | varchar(255) | NO   |     | NULL              |                   |
| cantidad       | int          | NO   |     | NULL              |                   |
| pelicula       | varchar(255) | NO   |     | NULL              |                   |
| fecha          | timestamp    | YES  |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED |
+----------------+--------------+------+-----+-------------------+-------------------+
5 rows in set (0.00 sec)
```

Figura 11. Datos y estructura de la tabla ventas

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

Server para el backend

El siguiente paso fue crear el proyecto donde se desarrollaría el servidor. Utilicé la terminal para crear un nuevo directorio llamado Server y dentro de ese directorio, inicialicé un proyecto de Node.js ejecutando el comando: npm init -y

Este comando generó automáticamente el archivo package.json, que se utiliza para gestionar las dependencias del proyecto. A continuación, instalé las dependencias necesarias para el servidor utilizando el siguiente comando: npm install express mysql2 body-parser cors

Estas dependencias incluyeron:

- **Express:** Un framework para Node.js que facilita la creación de servidores web.
- **mysql2:** Un módulo que permite la conexión y gestión de bases de datos MySQL desde Node.js.
- **body-parser:** Un middleware para analizar los datos JSON enviados en las solicitudes HTTP.
- **cors:** Un middleware que permite gestionar las solicitudes entre diferentes orígenes (muy útil cuando el frontend y backend están en diferentes puertos o dominios).

Con estas dependencias instaladas, procedí a escribir el archivo server.js que configuraría el servidor. Es por ello por lo que la estructura del servidor que realice quedo de la siguiente manera.

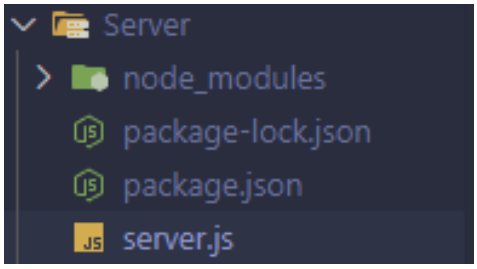


Figura 12. Estructura del proyecto del Server

Server.js

Una vez que realicé la estructura del servidor procedí a realizar el server.js el cual es el corazón del backend de la aplicación de venta de boletos para el cine, y su propósito es gestionar las solicitudes provenientes del cliente, interactuar con la base de datos para obtener y actualizar información, y enviar respuestas adecuadas al frontend. A continuación, se describe el funcionamiento detallado de este archivo.

En primer lugar, utilicé Express para configurar el servidor, lo que facilita la creación y manejo de rutas HTTP en Node.js. Configuré el servidor para que escuchara en el puerto 3000. Este puerto es el que utilizará el cliente para realizar solicitudes al servidor, como la compra de boletos o la consulta de películas disponibles.

```
const express = require('express');
const mysql = require('mysql2');
const bodyParser = require('body-parser');
const cors = require('cors');
const app = express();
const port = 3000;
```

Figura 13. Configuración del servidor

Una de las primeras configuraciones importantes en el archivo fue la integración de CORS (Cross-Origin Resource Sharing). Esta configuración permite que el frontend, que se encuentra en un puerto diferente específicamente en el puerto 5500 que es el puerto que utiliza la extensión de Live Server, pueda realizar solicitudes al backend sin que el navegador las bloquee debido a políticas de seguridad. Esto se logra especificando los orígenes permitidos, los métodos HTTP y los encabezados que pueden ser utilizados por el cliente al interactuar con el servidor.

```
// Configurar CORS
app.use(cors({
  origin: 'http://127.0.0.1:5500',
  methods: ['GET', 'POST'],
  allowedHeaders: ['Content-Type']
}));
```

Figura 14. Configuración CORS

A continuación, utilicé body-parser, un middleware que permite procesar los datos que el cliente envía al servidor en formato JSON. De esta manera, pude acceder a la información enviada por el cliente, como el nombre del cliente, la cantidad de boletos solicitados y la película seleccionada, para luego procesarla adecuadamente.

```
// Middleware
app.use(bodyParser.json());
```

Figura 15. Middleware

La conexión con la base de datos MySQL se realiza utilizando el paquete mysql2, lo que me permitió establecer una conexión al servidor de base de datos en localhost y acceder a la base de datos cineboletos. Esta base de datos contiene las tablas de películas y ventas, que son consultadas o actualizadas según las solicitudes del cliente. Si la conexión a la base de datos es exitosa, el servidor imprime un mensaje en la consola indicando que la conexión fue

establecida correctamente; si ocurre algún error en la conexión, este se captura y muestra en la consola.

```
// Conexión a MySQL
const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'Atl1God$',
  database: 'cineboletos'
});

db.connect((err) => {
  if (err) {
    console.error('Error conectando a la base de datos:', err);
    return;
  }
  console.log('Conectado a la base de datos MySQL');
});
```

Figura 16. Conexión con la base de datos

El servidor maneja varias rutas, cada una con un propósito específico. La ruta /peliculas permite obtener la lista de películas disponibles, que es consultada desde la base de datos y enviada al cliente en formato JSON. De manera similar, la ruta /ventas permite obtener el historial de ventas, proporcionando al cliente información sobre las compras realizadas, como el nombre del cliente, la cantidad de boletos comprados y la película seleccionada.

```
// Ruta para obtener las películas
app.get('/peliculas', (req, res) => {
  db.query('SELECT * FROM peliculas', (err, result) => {
    if (err) {
      return res.status(500).send('Error al obtener las películas');
    }
    res.json(result);
  });
});
```

Figura 17. Ruta para obtener la cartelera

La ruta más importante es /comprar, que gestiona la compra de boletos. Cuando un cliente realiza una compra, el servidor valida que los datos enviados sean correctos, es decir, que se haya proporcionado un nombre, la cantidad de boletos y el nombre de la película. Luego, verifica si hay suficientes boletos disponibles en el inventario para la película seleccionada. Si el stock es suficiente, el servidor actualiza la base de datos para descontar la cantidad de boletos vendidos y registra la venta en la tabla correspondiente. Si el stock es insuficiente o los datos de la solicitud son incorrectos, se devuelve un mensaje de error al cliente.

```
// Ruta para comprar boletos
app.post('/comprar', (req, res) => {
  const { nombre_cliente, cantidad, pelicula } = req.body;

  // Validar los datos recibidos
  if (!nombre_cliente || !cantidad || !pelicula) {
    return res.status(400).json({ error: 'Faltan datos necesarios
  }

  db.query('SELECT stock FROM peliculas WHERE nombre = ?', [pelic
    if (err) {
      return res.status(500).send('Error al verificar los boletos
    }

    const stockDisponible = result[0].stock;
```

Figura 18. Ruta para comprar boletos

Además, implementé una ruta /limpiarVentas que elimina todos los registros de ventas de la base de datos. Esto se hace automáticamente cada minuto mediante un temporizador setInterval, lo que asegura que las ventas no se acumulen indefinidamente y mantenga la base de datos limpia.

```
// Ruta para limpiar las ventas en la base de datos
app.delete('/limpiarVentas', (req, res) => {
  db.query('DELETE FROM ventas', (err, result) => {
    if (err) {
      return res.status(500).send('Error al limpiar las ventas');
    }
    res.status(200).send('Historial de ventas limpio');
  });
});
```

Figura 19. Ruta para limpiar las ventas

Por último, el servidor está configurado para escuchar en el puerto 3000, y cuando está en funcionamiento, espera solicitudes del cliente. Una vez iniciado el servidor, el frontend puede realizar peticiones a las rutas definidas para obtener datos sobre las películas, realizar compras o consultar el historial de ventas.

Frontend para la Progressive Web App (PWA)

Una vez terminado el Server con el backend, procedí a realizar el frontend de la PWA, utilizando tecnologías estándar como HTML, CSS y JavaScript. El objetivo era crear una Progressive Web App (PWA) que ofreciera una experiencia de usuario fluida, tanto en dispositivos móviles como de escritorio. Además, me aseguré de que la aplicación pudiera funcionar incluso sin conexión a Internet, implementando funcionalidades como el caché offline y las actualizaciones automáticas, lo que la hace más accesible y eficiente.

Index.html

Primeramente, el archivo index.html me sirvió como el punto de entrada principal para la interfaz de usuario de la aplicación. Este archivo está estructurado siguiendo los estándares de HTML5 y es responsable de definir la estructura básica de la página. Dentro de la etiqueta <head>, se incluyen las configuraciones esenciales para que la aplicación funcione como una PWA, tales como la inclusión del archivo manifest.json, que permite a la aplicación ser instalada en la pantalla de inicio del dispositivo, y el enlace al archivo style.css para los estilos visuales.

En el cuerpo del documento (<body>), se define la estructura visual de la página, que se organiza en un contenedor principal ( ). Este contenedor incluye un encabezado con el nombre de la aplicación y una breve descripción. La interfaz principal está dividida en varias secciones:

Selección de Película: Los usuarios pueden seleccionar la película de su preferencia desde un desplegable (<select>), que se actualiza dinámicamente con las películas disponibles, obtenidas del backend.

```
<div class="content">
    <h2>Seleccionar Película:</h2>
    <select id="peliculaSeleccionada" class="input-field">
        <option value="">Selecciona una película</option>
    </select>
```

Figura 20. Selección de películas en el HTML

Formulario para Comprar Boletos: Los usuarios pueden ingresar su nombre y la cantidad de boletos que desean comprar, utilizando campos de texto e input numérico. Un botón de compra ejecuta la función comprarBoletos().

```
<h3>Comprar Boletos:</h3>
<input type="text" id="nombreCliente" placeholder="Tu nombre" class="input-fi
<input type="number" id="cantidadBoletos" placeholder="Cantidad" class="input
<button onclick="comprarBoletos()" class="btn">Comprar</button>
```

Figura 21. Formulario de compra de boletos en el HTML

Ventas Realizadas: Se muestra una lista (<ul>) con las ventas previas, que se actualiza constantemente con los datos obtenidos desde el servidor.

```
<h3>Ventas Realizadas:</h3>
<ul id="listaVentas"></ul>
```

Figura 22. Lista de ventas realizadas en el HTML

Style.css

El archivo style.css definí los estilos visuales para la interfaz de usuario, con el objetivo de crear una experiencia agradable y clara. Se utiliza una paleta de colores moderna, con tonos en rojo y azul para resaltar elementos clave, como los botones y el encabezado. Además, se implementan estilos de diseño responsivo para garantizar que la interfaz sea accesible y funcional tanto en dispositivos móviles como en pantallas de escritorio.

El diseño se organiza en un contenedor centrado que permite que el contenido se adapte a diferentes tamaños de pantalla, utilizando márgenes y rellenos adecuados. Los formularios y botones están estilizados para mejorar la experiencia de interacción del usuario, con un enfoque en la facilidad de uso y accesibilidad.

Script.js

Para gestionar la lógica de interacción con el backend, utilicé JavaScript. El archivo script.js contiene las funciones necesarias para conectar el frontend con el servidor. A través de este archivo, la aplicación puede realizar solicitudes al servidor para obtener las películas disponibles, registrar las compras de boletos y actualizar la lista de ventas. Se emplean las funciones fetch() para realizar solicitudes GET y POST al servidor, lo que permite obtener y enviar datos de forma eficiente.

Obtener Películas: La función obtenerPeliculas() hace una solicitud GET al servidor para obtener la lista de películas disponibles. Esta lista se presenta en el desplegable, mostrando el nombre de la película y la cantidad de boletos disponibles para cada una.

```
// Función para obtener las películas
function obtenerPeliculas() {
    fetch('http://localhost:3000/peliculas') // Actualiza la URL c
        .then(response => response.json())
        .then(peliculasData => {
            peliculas.length = 0; // Limpiar el array de películas
            peliculasData.forEach(pelicula => {
                peliculas.push(pelicula);
                const option = document.createElement("option");
                option.value = pelicula.nombre;
                option.textContent = `${pelicula.nombre} - Stock:
                document.getElementById("peliculaSeleccionada").ap
            });
        })
        .catch(error => console.error('Error al obtener las pelícu
}
```

Figura 23. Función para obtener las películas en el Script

Obtener Ventas: La función obtenerVentas() realiza una solicitud GET al servidor para obtener las ventas realizadas, y actualiza dinámicamente la lista de ventas en la interfaz.

```
// Función para obtener las ventas realizadas
function obtenerVentas() {
    fetch('http://localhost:3000/ventas') // Obtener ventas desde e
        .then(response => response.json())
        .then(ventasData => {
            ventas = ventasData; // Guardar las ventas en el array
            actualizarVentas(); // Actualizar la interfaz con las v
        })
        .catch(error => console.error('Error al obtener las ventas'
}
```

Figura 24. Función para obtener las ventas realizadas

Comprar Boletos: Cuando el usuario ingresa su nombre y la cantidad de boletos a comprar, la función comprarBoletos() envía los datos al servidor mediante una solicitud POST. Si la compra es exitosa, el stock de boletos se actualiza en el frontend y se muestra un mensaje de confirmación. Si hay algún error, como un stock insuficiente, el sistema notificará al usuario.

```
// Función para comprar boletos
function comprarBoletos() {
    const nombre = document.getElementById("nombreCliente").value.trim(
    const cantidad = parseInt(document.getElementById("cantidadBoletos"
    const pelicula = document.getElementById("peliculaSeleccionada").va
```

Figura 25. Función para comprar boletos en el Script

Actualizar la Interfaz: Las funciones actualizarPeliculas() y actualizarVentas() se encargan de actualizar los elementos de la interfaz, como el desplegable de películas y la lista de ventas, cada vez que se realiza una compra o cuando se reciben nuevos datos del servidor.

```
// Actualiza las películas en el select
function actualizarPeliculas() {
    const selectPelicula = document.getElementById("p
    selectPelicula.innerHTML = '<option value="">Sele
    peliculas.forEach(pelicula => {
        const option = document.createElement("option
        option.value = pelicula.nombre;
        option.textContent = `${pelicula.nombre} - St
        selectPelicula.appendChild(option);
    });
}

// Actualiza la lista de ventas
function actualizarVentas() {
    const lista = document.getElementById("listaVenta
    lista.innerHTML = ""; // Limpiar la lista de vent
    ventas.forEach(venta => {
        const item = document.createElement("li");
        item.textContent = `${venta.nombre_cliente} c
        lista.appendChild(item);
```

Figura 26. Actualización de datos para la interfaz

Service Worker (sw.js)

El archivo sw.js que implementé en esta práctica tiene la función de manejar el caché de la aplicación y permitir su funcionamiento offline, lo cual es uno de los principales beneficios de desarrollar una Progressive Web App (PWA). Un Service Worker es un script que se ejecuta en segundo plano y que permite interceptar las solicitudes de red realizadas por la aplicación. Esto significa que, incluso cuando el usuario no tiene conexión a Internet, el Service Worker puede responder a las solicitudes utilizando los archivos almacenados en la caché.

El proceso comienza cuando el Service Worker se instala, lo que ocurre mediante el evento install. Durante la instalación, los recursos principales de la aplicación, como los archivos HTML, CSS, JavaScript y las imágenes, se almacenan en la caché. Esto asegura que la aplicación pueda seguir funcionando incluso sin conexión a Internet, ya que estos archivos esenciales ya están disponibles localmente en el dispositivo del usuario.

```
// Instalación del Service Worker
self.addEventListener('install', (e) => {
  console.log('Service Worker instalado');
  e.waitUntil(
    caches.open(CACHE_NAME).then((cache) => {
      return cache.addAll(urlsToCache);
    })
  );
});
```

Figura 27. Instalación del Service Worker

En el evento activate, el Service Worker se encarga de limpiar cualquier caché antigua que pueda estar presente, asegurando que solo se mantengan los archivos más actuales. Esto es importante para evitar que el usuario vea versiones obsoletas de la aplicación.

```
// Activación del Service Worker - Limpieza de caches viejos
self.addEventListener('activate', (e) => {
 const cacheWhitelist = [CACHE_NAME];
 console.log('Service Worker activado');
 e.waitUntil(
   caches.keys().then((cacheNames) => {
     return Promise.all(
       cacheNames.map((cacheName) => {
         if (!cacheWhitelist.includes(cacheName)) {
           console.log('Cache antiguo eliminado:', cacheName
           return caches.delete(cacheName);
         }
```

Figura 28. Actualización del Service Worker

La funcionalidad clave del Service Worker es su capacidad para gestionar las solicitudes de red a través del evento fetch. Cuando el usuario solicita un recurso, el Service Worker primero verifica si ese recurso está en la caché. Si lo encuentra, devuelve el recurso directamente desde allí y, al mismo tiempo, lo actualiza en segundo plano con la versión más reciente obtenida de la red. Si el recurso no está en la caché, el Service Worker lo solicita a la red y, si no hay conexión, muestra una página de respaldo que fue previamente almacenada en caché. Esta estrategia asegura que la aplicación siempre tenga una respuesta, incluso si la red no está disponible.

```
// Manejo de peticiones - Cache y actualización en segu
self.addEventListener('fetch', (event) => {
  // Solo manejar peticiones GET
  if (event.request.method !== 'GET') return;

  event.respondWith(
    caches.match(event.request).then((cachedResponse) =>
      const fetchRequest = event.request.clone();

      // Si hay en caché, devolver y actualizar en segu
      if (cachedResponse) {
        fetch(fetchRequest).then((response) => {
          if (
            response &&
            response.status === 200 &&
            response.type === 'basic'
          ) {
            caches.open(CACHE_NAME).then((cache) => {
              cache.put(event.request, response.clone()
            });
```

Figura 29. Manejo de peticiones, cache y actualización en segundo plano del Service Worker

Manifesto (manifest.json)

El archivo manifest.json que utilicé en esta práctica es fundamental para que la aplicación CineBoleto funcione correctamente como una Progressive Web App (PWA). Este archivo permite definir cómo debe comportarse la aplicación cuando es instalada en el dispositivo del usuario. Específicamente, se encarga de establecer elementos importantes, como el nombre de la aplicación, los iconos, los colores y la página de inicio.

En el manifest.json, definí el nombre de la aplicación como "CineBoleto", que es el nombre que se mostrará cuando el usuario la instale en su dispositivo. Además, incluí un nombre corto "CineBoleto" para que se utilice en la pantalla de inicio. Este archivo también especifica el color de fondo #ffffff y el color del tema #e74c3c, los cuales se aplican al abrir la aplicación y ayudan a mejorar la apariencia visual de la barra de herramientas en dispositivos móviles.

Otro aspecto clave que se define en el manifest.json son los iconos. Se establecen diferentes tamaños de iconos (192x192 px y 512x512 px) que se utilizan al agregar la aplicación a la pantalla de inicio del dispositivo. Estos iconos permiten que la aplicación se vea y funcione de manera similar a una aplicación nativa.

```
"name": "CineBoleto",
"short_name": "CineBoleto",
"start_url": "./index.html",
"display": "standalone",
"background_color": "#ffffff",
"theme_color": "#e74c3c",
"icons": [
    {
        "src": "icons/favicon-192x192.png",
        "sizes": "192x192",
        "type": "image/png"
    },
    {
        "src": "icons/favicon-512x512.png",
        "type": "image/png"
```

Figura 30. manifest.json

Instrucciones para ejecutar el proyecto

Para poder ejecutarlo una vez que tengamos todos nuestros archivos configurados correctamente como se mostró anteriormente tendremos primeramente que iniciar el servido de Node.js es por ello que nos vamos a la ruta del proyecto y ejecutamos el comando *node server.js* lo cual hará que se inicie como se muestra en la siguiente imagen:

Figura 31. Ejecución del Servidor en Node.js

Por último, utilicé Live Server de Visual Studio Code (VS Code) para ejecutar la aplicación CineBoleto en un entorno de desarrollo local. Live Server es una extensión que facilita la previsualización de aplicaciones web en tiempo real, permitiendo ver los cambios al instante sin necesidad de recargar manualmente la página. A continuación, describo cómo ejecuté la aplicación y cómo verifiqué que todo estuviera funcionando correctamente.

Instalación de Live Server en Visual Studio Code

Primero, instalé Live Server en VS Code. Para hacerlo, abrí VS Code, luego fui al panel de extensiones y busqué "Live Server". Seleccioné la extensión creada por Ritwick Dey e hice clic en el botón de Instalar. Una vez instalada, la extensión me permitió abrir cualquier archivo HTML con un servidor local, lo que facilita la visualización de la aplicación sin necesidad de configurar servidores manualmente.

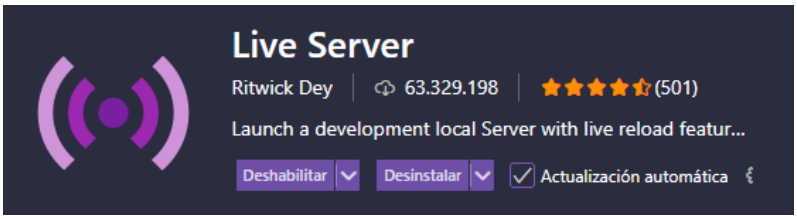


Figura 32. Instalación de Live Server en VS Code

Ejecución con Live Server

Por último, una vez que mi proyecto estaba abierto en en VS Code, pude ejecutar la aplicación rápidamente. Para ello, hice clic derecho sobre el archivo index.html en el explorador de archivos de VS Code y seleccioné la opción "Open with Live Server". Esto lanzó la aplicación en una nueva pestaña del navegador, normalmente en la URL http://127.0.0.1:5500. En ese momento, la aplicación ya estaba funcionando en un servidor local y podía interactuar con ella desde el navegador, como si estuviera en producción.

De esa manera obteniendo como resultado que mi aplicación se visualizara desde el navegador y que se conectara con mi backend y la base de datos. Así como se muestra a continuación:

Figura 33. PWA vista desde el Frontend

Resultados

La aplicación CineBoleto cumplió con los objetivos establecidos en la práctica. Logré implementar una PWA que permite a los usuarios comprar boletos de cine, ver las películas disponibles, y consultar las ventas realizadas en tiempo real. El uso de Service Worker permitió que la aplicación funcione correctamente incluso en condiciones de conectividad limitada o sin conexión, lo que proporciona una experiencia de usuario mejorada.

Pruebas desde el Frontend

Visualización de la Cartelera de Películas

Al abrir la aplicación en el navegador, la cartelera de películas y la disponibilidad de boletos se cargaron correctamente en la interfaz de usuario. Esta funcionalidad fue verificada mediante una solicitud GET realizada desde el archivo script.js, que se comunicó con el backend a través de la URL ***http://localhost:3000/peliculas***. La respuesta fue recibida correctamente y se mostró dinámicamente en el desplegable de películas en la página web, lo que confirmó que el frontend y el backend se comunicaron de manera efectiva.

Esta funcionalidad permitió al usuario ver la lista de películas disponibles con su respectivo stock de boletos en tiempo real. Cada vez que se realiza una compra, el stock de boletos se actualiza tanto en el frontend como en la base de datos, asegurando que la información mostrada siempre esté sincronizada.

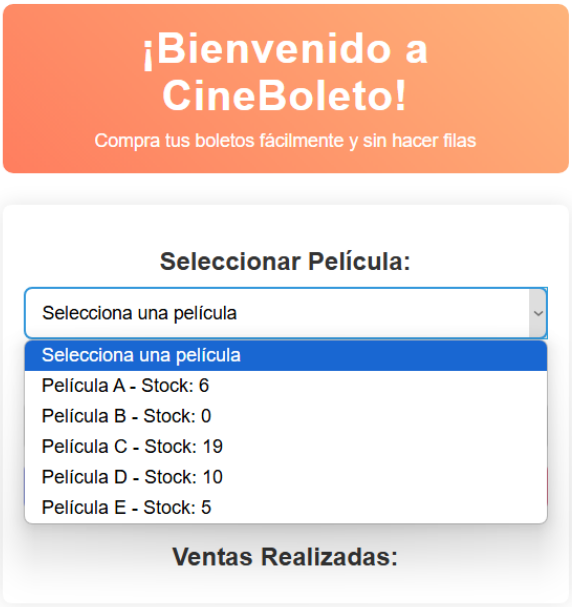


Figura 34. Despliegue de la cartelera de películas en la PWA

Proceso de Compra de Boletos

El proceso de compra fue probado ingresando un nombre de cliente y seleccionando diferentes cantidades de boletos para varias películas. Al presionar el botón "Comprar Boletos", se envió una solicitud POST al backend, a través del endpoint http://localhost:3000/comprar, con los detalles de la compra (nombre del cliente, cantidad de boletos y película seleccionada). El backend procesó la solicitud correctamente, verificando la disponibilidad de boletos y registrando la venta en la base de datos.

La respuesta del backend fue recibida exitosamente y se mostró un mensaje de confirmación al usuario, informando que la compra había sido realizada con éxito. Además, el stock de boletos se actualizó automáticamente, y la interfaz volvió a hacer una solicitud GET para obtener la lista de películas actualizada, reflejando los cambios en tiempo real.

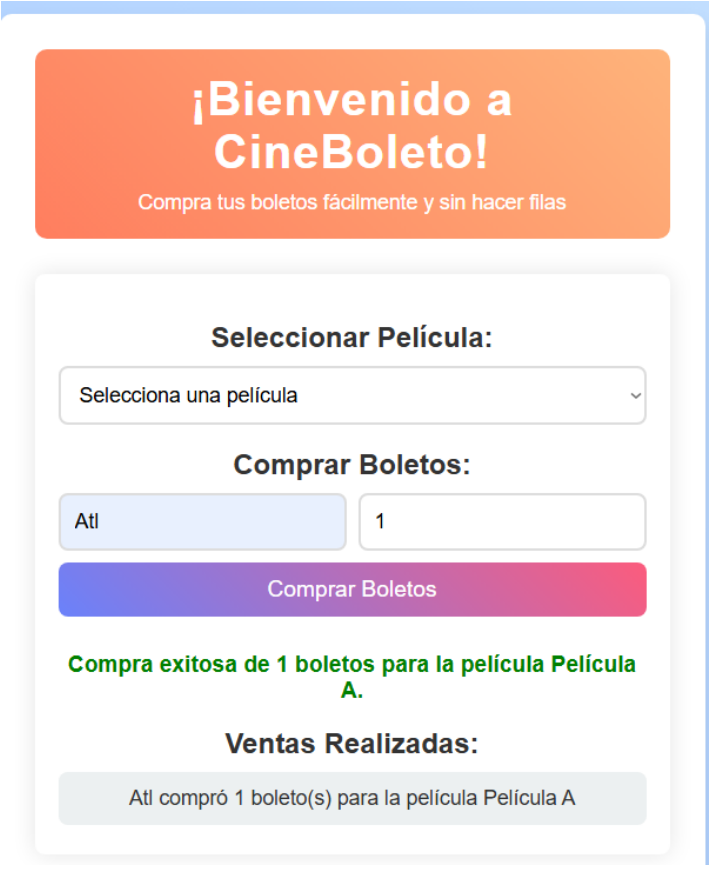


Figura 35. Realización de una compra exitosa desde el sitio web

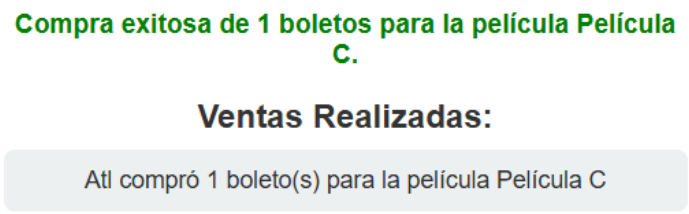
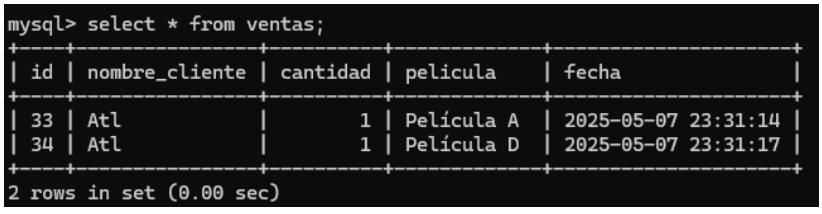


Figura 36. Bitácora en donde se comprueba la conexión exitosa del front



```
mysql> select * from ventas;
+----+----------------+----------+-------------+---------------------+
| id | nombre_cliente | cantidad | pelicula    | fecha               |
+----+----------------+----------+-------------+---------------------+
| 33 | Atl            |        1 | Película A  | 2025-05-07 23:31:14 |
| 34 | Atl            |        1 | Película D  | 2025-05-07 23:31:17 |
+----+----------------+----------+-------------+---------------------+
2 rows in set (0.00 sec)
```

Figura 37 Registro exitoso en terminal

Pruebas de Validación del Sistema

Durante las pruebas de validación, se implementaron restricciones en el frontend para garantizar que los usuarios ingresaran los datos correctamente antes de realizar una compra. Por ejemplo, si un usuario ingresaba su nombre pero no seleccionaba una película, el sistema

solicitaba que eligiera una película para continuar con la compra. Igualmente, si el usuario seleccionaba una película, pero no ingresaba su nombre, se le pedía que completara ese campo antes de proceder con la compra.

Estas validaciones aseguraron que las compras solo se pudieran realizar cuando todos los campos obligatorios estuvieran correctamente completados, evitando errores en el proceso de compra.

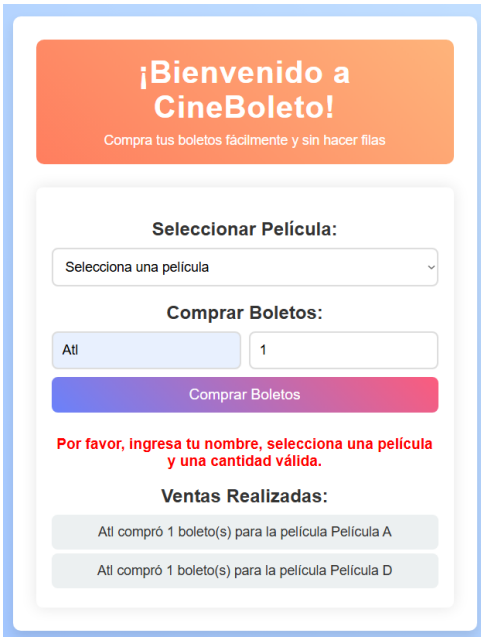


Figura 38. Solicitud al usuario de que ingresa tu nombre, selecciona una película y una cantidad válida.

Pruebas de Funcionalidad Offline

Una de las características clave de esta aplicación es su capacidad para funcionar offline, lo cual fue probado desconectando la red y verificando que la aplicación siguiera operativa. Gracias al Service Worker implementado, la aplicación cargó correctamente desde el caché los recursos esenciales, permitiendo al usuario seguir navegando por las opciones de películas, aunque con algunas funcionalidades limitadas debido a la falta de conexión.

Además, cuando la red estaba disponible, el Service Worker se encargaba de actualizar la caché en segundo plano, asegurando que los recursos de la aplicación estuvieran siempre actualizados. Esta funcionalidad permitió que la aplicación CineBoleto operara de manera eficiente tanto en condiciones de conectividad limitada como sin conexión.

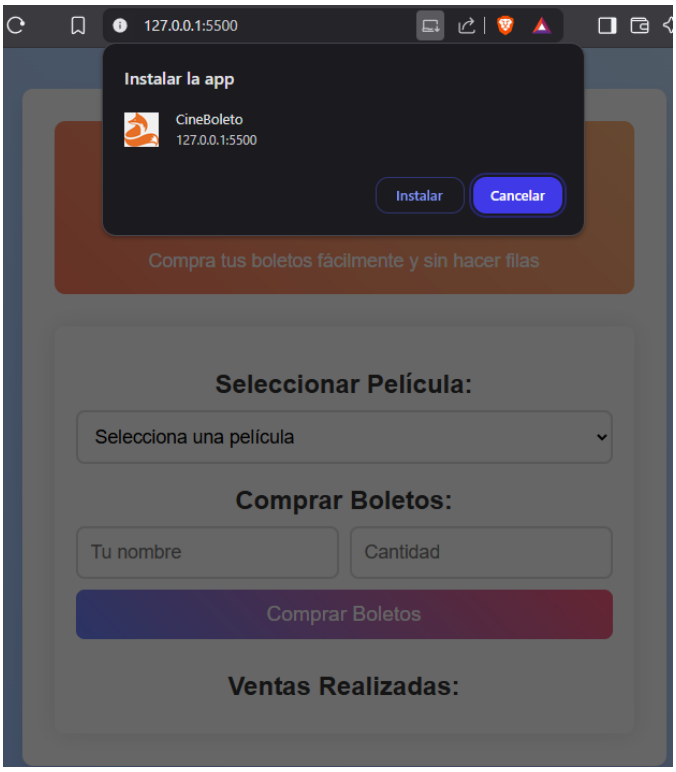


Figura 39.Funcionalidad Offline, instalación de la app

Figura 40. PWA instalada

Pruebas de PWA

La funcionalidad de PWA fue verificada mediante la instalación de la aplicación en la pantalla de inicio de un dispositivo móvil. Al acceder a la aplicación desde el navegador y agregarla a la pantalla de inicio, la aplicación se comportó como una aplicación nativa, sin mostrar los elementos del navegador, como la barra de direcciones. Además, al abrir la aplicación desde la pantalla de inicio, la aplicación se ejecutó en una ventana independiente, lo que mejora la experiencia de usuario al proporcionar una interfaz más limpia y fluida.

El archivo manifest.json fue correctamente reconocido, y el icono de la aplicación se mostró en la pantalla de inicio del dispositivo con los colores y características definidos, lo que permitió una instalación exitosa de la PWA.

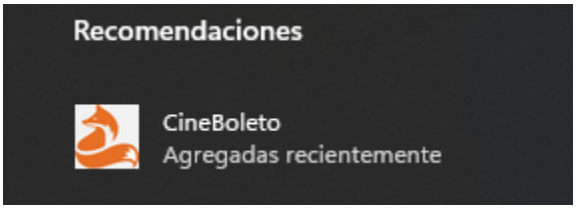


Figura 41. Verificación de la instalación de la PWA

Las pruebas realizadas confirmaron que mi PWA cumple con todas las funcionalidades esperadas para la compra de boletos de cine en línea. El sistema de frontend y backend funcionaron de manera fluida, y la integración del Service Worker permitió que la aplicación operara correctamente en modo offline. Además, la PWA fue instalada con éxito en el dispositivo móvil, lo que mejora la experiencia de usuario al ofrecer una interfaz similar a una aplicación nativa.

Gracias a estas pruebas, pude validar que la solución implementada es efectiva para la gestión distribuida y en tiempo real de la venta de boletos de cine. La combinación de tecnologías como PWA, Service Worker y Node.js garantizó una experiencia de usuario óptima, incluso en condiciones de red inestables o sin conexión. Con estos resultados, la aplicación demuestra ser robusta y escalable, lista para su uso en entornos de producción.

Conclusión

Esta práctica fue un reto significativo y me permitió profundizar en la construcción y desarrollo de Progressive Web Apps (PWA). A través de esta práctica, aprendí a combinar el desarrollo de un backend robusto con tecnologías como Node.js y MySQL, junto con un frontend interactivo y adaptativo utilizando HTML, CSS y JavaScript. El resultado final fue una aplicación funcional llamada CineBoleto, que no solo permite la compra de boletos de cine, sino que también se puede instalar en dispositivos móviles, funcionar offline y gestionar las compras en tiempo real.

Una de las partes más complejas de esta práctica fue la integración de la PWA con el Service Worker y el archivo manifest.json. Desde el inicio, tuve que aprender a gestionar las rutas de red, cómo almacenar en caché los recursos esenciales para permitir la funcionalidad sin conexión, y cómo garantizar que la aplicación pudiera comportarse como una aplicación nativa cuando se instalara en un dispositivo móvil. A pesar de las dificultades iniciales, como los errores de caché y problemas con la actualización de los recursos en segundo plano, logré solucionarlos a medida que avanzaba y entendía mejor cómo funciona el ciclo de vida de un Service Worker.

El proceso de conexión entre el frontend y el backend también fue desafiante. Aunque ya tenía una idea básica de cómo construir un servidor con Node.js, integrar la funcionalidad de compra de boletos y la gestión de ventas con una base de datos MySQL requirió ajustes adicionales. A medida que el servidor respondía correctamente a las solicitudes de GET y POST, pude verificar la actualización en tiempo real de las películas disponibles y las ventas realizadas, lo que demostró la eficiencia del sistema distribuido.

Otro aspecto importante fue la implementación del manifest.json, que permitió que la aplicación fuera instalada en la pantalla de inicio de los dispositivos móviles, proporcionando una experiencia de usuario similar a la de una aplicación nativa. Esta integración de características propias de las PWA contribuyó a mejorar la accesibilidad, la velocidad y la disponibilidad offline de la aplicación.

A pesar de los desafíos, como la adaptación de la solución anterior a una PWA y los problemas iniciales con el manejo de caché y las notificaciones en tiempo real, el resultado final fue muy satisfactorio. Al ver cómo la aplicación respondía correctamente al registrar compras, actualizar el stock y mostrar las ventas realizadas en tiempo real, sentí que había logrado crear una solución completa que no solo funcionaba, sino que también mejoraba la experiencia del usuario.

En lo personal, realizar este proyecto me pareció muy interesante. Fue una excelente oportunidad para aplicar y reforzar mis conocimientos sobre sistemas distribuidos y aprender nuevos conceptos relacionados con el desarrollo de PWA. La experiencia de desarrollar una aplicación que gestiona compras, notificaciones y actualizaciones de stock en tiempo real, mientras funciona offline, me brindó una visión más amplia sobre cómo se pueden crear soluciones modernas y escalables en el desarrollo de software. Sin duda, los conocimientos adquiridos en esta práctica me serán útiles para futuros proyectos y me permitieron resolver problemas reales que surgen al integrar tecnologías avanzadas en el desarrollo web.

Referencias

- 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].
- 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].

Anexos

Server.js

```
const express = require('express');
const mysql = require('mysql2');
const bodyParser = require('body-parser');
const cors = require('cors');
const app = express();
const port = 3000;

// Configurar CORS
app.use(cors({
  origin: 'http://127.0.0.1:5500',
  methods: ['GET', 'POST'],
  allowedHeaders: ['Content-Type']
}));

// Middleware
app.use(bodyParser.json());

// Conexión a MySQL
const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'Atl1God$',
  database: 'cineboletos'
});

db.connect((err) => {
  if (err) {
    console.error('Error conectando a la base de datos:', err);
    return;
  }
  console.log('Conectado a la base de datos MySQL');
});

// Ruta para obtener las películas
app.get('/peliculas', (req, res) => {
  db.query('SELECT * FROM peliculas', (err, result) => {
```

```
    if (err) {
      return res.status(500).send('Error al obtener las
películas');
    }
    res.json(result);
  });
});

// Ruta para obtener las ventas realizadas
app.get('/ventas', (req, res) => {
  db.query('SELECT * FROM ventas', (err, result) => {
    if (err) {
      return res.status(500).send('Error al obtener las ventas');
    }
    res.json(result);
  });
});

// Ruta para limpiar las ventas en la base de datos
app.delete('/limpiarVentas', (req, res) => {
  db.query('DELETE FROM ventas', (err, result) => {
    if (err) {
      return res.status(500).send('Error al limpiar las ventas');
    }
    res.status(200).send('Historial de ventas limpio');
  });
});

// Limpiar las ventas automáticamente cada 1 minuto
setInterval(() => {
  fetch('http://localhost:3000/limpiarVentas', {
    method: 'DELETE',
  })
  .then(response => response.text())
  .then(data => console.log(data))
  .catch(error => console.error('Error al limpiar ventas en la
base de datos:', error));
}, 60000);
```

```
// Ruta para comprar boletos
app.post('/comprar', (req, res) => {
  const { nombre_cliente, cantidad, pelicula } = req.body;

  // Validar los datos recibidos
  if (!nombre_cliente || !cantidad || !pelicula) {
    return res.status(400).json({ error: 'Faltan datos necesarios'
});
  }

  db.query('SELECT stock FROM peliculas WHERE nombre = ?',
[pelicula], (err, result) => {
    if (err) {
      return res.status(500).send('Error al verificar los
boletos');
    }

    const stockDisponible = result[0].stock;

    if (stockDisponible < cantidad) {
      return res.status(400).json({ error: 'No hay suficientes
boletos disponibles' });
    }

    // Actualizar el stock de boletos
    db.query('UPDATE peliculas SET stock = stock - ? WHERE nombre
= ?', [cantidad, pelicula], (err) => {
      if (err) {
        return res.status(500).send('Error al actualizar el
stock');
      }

      // Guardar la venta
      db.query('INSERT INTO ventas (nombre_cliente, cantidad,
pelicula) VALUES (?, ?, ?)', [nombre_cliente, cantidad, pelicula],
(err) => {
        if (err) {
          return res.status(500).send('Error al registrar la
venta');
```

```
        }
        res.status(200).json({ success: true });
      });
    });
  });
});

// Iniciar el servidor
app.listen(port, () => {
  console.log(`Servidor escuchando en http://localhost:${port}`);
});
```

Index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>CineBoleto - PWA</title>
    <link rel="manifest" href="manifest.json">
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <div class="container">
        <header>
            <h1>¡Bienvenido a CineBoleto!</h1>
            <p>Compra tus boletos fácilmente y sin hacer filas</p>
        </header>

        <div class="content">
            <h2>Seleccionar Película:</h2>
            <select id="peliculaSeleccionada" class="input-field">
                <option value="">Selecciona una película</option>
            </select>

            <h3>Comprar Boletos:</h3>
```

```
            <div class="input-container">
                <input type="text" id="nombreCliente"
placeholder="Tu nombre" class="input-field">
                <input type="number" id="cantidadBoletos"
placeholder="Cantidad" class="input-field" min="1">
            </div>
            <button onclick="comprarBoletos()" class="btn">Comprar
Boletos</button>

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

            <h3>Ventas Realizadas:</h3>
            <ul id="listaVentas"></ul>
        </div>
    </div>
    <script src="script.js"></script>
    <script>
        if ('serviceWorker' in navigator) {
          navigator.serviceWorker.register('./sw.js')
            .then((reg) => console.log('Service Worker
registrado', reg))
            .catch((err) => console.error('Error registrando el
Service Worker', err));
        }
    </script>
</body>
</html>
```

Style.css

```
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
body {
    font-family: 'Arial', sans-serif;
    background: linear-gradient(45deg, #a0c4ff, #c6e2ff);
```

```
    color: #333;
    display: flex;
    justify-content: center;
    align-items: center;
    min-height: 100vh;
    text-align: center;
}
.container {
    max-width: 600px;
    width: 100%;
    padding: 30px;
    background: white;
    border-radius: 10px;
    box-shadow: 0 8px 15px rgba(0, 0, 0, 0.1);
}
header {
    margin-bottom: 30px;
    background: linear-gradient(45deg, #FF7E5F, #FEB47B);
    padding: 20px;
    border-radius: 10px;
}
header h1 {
    font-size: 2.5em;
    color: white;
    letter-spacing: 1px;
    margin-bottom: 10px;
}
header p {
    color: #fff;
    font-size: 1.1em;
}
.content {
    padding: 20px;
    background: #fff;
    border-radius: 8px;
    box-shadow: 0px 0px 20px rgba(0, 0, 0, 0.1);
}
h2, h3 {
    color: #333;
```

```
    margin-top: 20px;
    font-size: 1.5em;
    font-weight: 600;
}
.input-field {
    display: block;
    width: 100%;
    padding: 12px;
    margin: 10px 0;
    border: 2px solid #ddd;
    border-radius: 8px;
    font-size: 1.1em;
    transition: border-color 0.3s ease;
}
.input-field:focus {
    border-color: #3498db;
    outline: none;
}
.input-container {
    display: flex;
    justify-content: space-between;
    gap: 10px;
}
.btn {
    background: linear-gradient(45deg, #6A82FB, #FC5C7D);
    color: white;
    border: none;
    padding: 12px;
    width: 100%;
    margin-bottom: 10px;
    border-radius: 8px;
    font-size: 1.2em;
    cursor: pointer;
    transition: background-color 0.3s ease;
}
.btn:hover {
    background: linear-gradient(45deg, #FC5C7D, #6A82FB);
}
#mensaje {
```

```
    margin-top: 20px;
    color: green;
    font-size: 1.2em;
    font-weight: bold;
}
#listaVentas {
    list-style: none;
    padding: 0;
    margin-top: 10px;
}
#listaVentas li {
    background: #ecf0f1;
    margin: 5px 0;
    padding: 12px;
    border-radius: 8px;
    font-size: 1.1em;
    color: #333;
    transition: background-color 0.3s ease;
}
#listaVentas li:hover {
    background: #dfe6e9;
}
```

Script.js

```
let boletosDisponibles = 30;
let ventas = [];
const peliculas = [];

// Función para obtener las películas
function obtenerPeliculas() {
    fetch('http://localhost:3000/peliculas')
        .then(response => response.json())
        .then(peliculasData => {
            peliculas.length = 0; // Limpiar el array de películas
            peliculasData.forEach(pelicula => {
                peliculas.push(pelicula);
                const option = document.createElement("option");
```

```
                option.value = pelicula.nombre;
                option.textContent = `${pelicula.nombre} - Stock:
${pelicula.stock}`;
                document.getElementById("peliculaSeleccionada").ap
pendChild(option);
            });
        })
        .catch(error => console.error('Error al obtener las
películas', error));
}

// Función para obtener las ventas realizadas
function obtenerVentas() {
    fetch('http://localhost:3000/ventas') // Obtener ventas desde
el backend
        .then(response => response.json())
        .then(ventasData => {
            ventas = ventasData; // Guardar las ventas en el array
            actualizarVentas(); // Actualizar la interfaz con las
ventas obtenidas
        })
        .catch(error => console.error('Error al obtener las
ventas', error));
}

// Función para comprar boletos
function comprarBoletos() {
    const nombre =
document.getElementById("nombreCliente").value.trim();
    const cantidad =
parseInt(document.getElementById("cantidadBoletos").value);
    const pelicula =
document.getElementById("peliculaSeleccionada").value;

    // Validar los datos antes de enviar la solicitud
    if (!nombre || isNaN(cantidad) || cantidad <= 0 || !pelicula)
{
        mostrarMensaje("Por favor, ingresa tu nombre, selecciona
una película y una cantidad válida.", true);
```

```
        return;
    }

    // Enviar la compra a la API
    fetch('http://localhost:3000/comprar', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ nombre_cliente: nombre, cantidad:
cantidad, pelicula })
    })
    .then(response => response.json())
    .then(data => {
        // Verifica si la compra fue exitosa
        if (data.success) {
            const peliculaSeleccionada = peliculas.find(p =>
p.nombre === pelicula);
            if (peliculaSeleccionada) {
                peliculaSeleccionada.stock -= cantidad; //
Descontamos los boletos en el frontend
            }
            actualizarPeliculas(); // Volvemos a renderizar las
películas actualizadas en el select
            obtenerVentas(); // Volver a obtener las ventas
actualizadas
            mostrarMensaje(`Compra exitosa de ${cantidad} boletos
para la película ${pelicula}.`);
        } else {
            mostrarMensaje(data.error, true); // Mostrar error si
la compra falla
        }
    })
    .catch(error => {
        mostrarMensaje("Hubo un problema al procesar la compra.",
true);
    });
}

// Actualiza las películas en el select
function actualizarPeliculas() {
```

```
    const selectPelicula =
document.getElementById("peliculaSeleccionada");
    selectPelicula.innerHTML = '<option value="">Selecciona una
película</option>'; // Limpiar las opciones
    peliculas.forEach(pelicula => {
        const option = document.createElement("option");
        option.value = pelicula.nombre;
        option.textContent = `${pelicula.nombre} - Stock:
${pelicula.stock}`;
        selectPelicula.appendChild(option);
    });
}

// Actualiza la lista de ventas
function actualizarVentas() {
    const lista = document.getElementById("listaVentas");
    lista.innerHTML = ""; // Limpiar la lista de ventas antes de
actualizarla
    ventas.forEach(venta => {
        const item = document.createElement("li");
        item.textContent = `${venta.nombre_cliente} compró
${venta.cantidad} boleto(s) para la película ${venta.pelicula}`;
        lista.appendChild(item);
    });
}

// Mostrar mensaje al usuario
function mostrarMensaje(mensaje, error = false) {
    const mensajeEl = document.getElementById("mensaje");
    mensajeEl.textContent = mensaje;
    mensajeEl.style.color = error ? "red" : "green";
}

// Llamar a obtenerPeliculas y obtenerVentas cada 10 segundos para
actualizar el stock y las ventas
setInterval(obtenerVentas, 10000); // Llamar cada 10 segundos para
actualizar las ventas
obtenerPeliculas(); // Llamada inicial para obtener las películas
obtenerVentas(); // Llamada inicial para obtener las ventas
```

```
document.addEventListener("DOMContentLoaded",
actualizarPeliculas);
```

Sw.js

```
const CACHE_NAME = 'cineboletos-cache-v1';
const urlsToCache = [
  './',
  './index.html',
  './style.css',
  './script.js',
  './icons/favicon-192x192.png',
  './icons/favicon-512x512.png',
];

// Instalación del Service Worker
self.addEventListener('install', (e) => {
  console.log('Service Worker instalado');
  e.waitUntil(
    caches.open(CACHE_NAME).then((cache) => {
      return cache.addAll(urlsToCache);
    })
  );
});

// Activación del Service Worker - Limpieza de caches viejos
self.addEventListener('activate', (e) => {
  const cacheWhitelist = [CACHE_NAME];
  console.log('Service Worker activado');
  e.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (!cacheWhitelist.includes(cacheName)) {
            console.log('Cache antiguo eliminado:', cacheName);
            return caches.delete(cacheName);
          }
```

```
        })
      );
    })
  );
});

// Manejo de peticiones - Cache y actualización en segundo plano
self.addEventListener('fetch', (event) => {
  // Solo manejar peticiones GET
  if (event.request.method !== 'GET') return;

  event.respondWith(
    caches.match(event.request).then((cachedResponse) => {
      const fetchRequest = event.request.clone();

      // Si hay en caché, devolver y actualizar en segundo plano
      if (cachedResponse) {
        fetch(fetchRequest).then((response) => {
          if (
            response &&
            response.status === 200 &&
            response.type === 'basic'
          ) {
            caches.open(CACHE_NAME).then((cache) => {
              cache.put(event.request, response.clone());
            });
          }
        }).catch((err) => {
          console.warn('Fallo al actualizar caché:', err);
        });

        return cachedResponse;
      }

      // Si no está en caché, intentar obtenerlo de la red
      return fetch(fetchRequest)
        .then((response) => {
          if (
            response &&
```

```
            response.status === 200 &&
            response.type === 'basic'
          ) {
            const responseToCache = response.clone();
            caches.open(CACHE_NAME).then((cache) => {
              cache.put(event.request, responseToCache);
            });
          }
          return response;
        })
        .catch(() => {
          // Si falla la red, mostrar página de respaldo
          return caches.match('./offline.html');
        });
    })
  );
});
```

manifest.json

```
{
    "name": "CineBoleto",
    "short_name": "CineBoleto",
    "start_url": "./index.html",
    "display": "standalone",
    "background_color": "#ffffff",
    "theme_color": "#e74c3c",
    "icons": [
        {
            "src": "icons/favicon-192x192.png",
            "sizes": "192x192",
            "type": "image/png"
        },
        {
            "src": "icons/favicon-512x512.png",
            "type": "image/png"
        }
    ]
}
```