INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

UNIDAD DE APRENDIZAJE

SISTEMAS DISTRIBUIDOS

PRACTICA No. 8

PROGRESSIVE WEB APPS (PWA)

ALUMNO

GÓMEZ GALVAN DIEGO YAEL

GRUPO

7CM1

PROFESOR

CARRETO ARELLANO CHADWICK

FECHA DE ENTREGA

07 DE MAYO DE 2025

INDICE

ANTESCEDENTES

En los últimos años, el desarrollo de aplicaciones ha experimentado una evolución significativa impulsada por la necesidad de ofrecer experiencias digitales más fluidas, rápidas y accesibles. Esta necesidad se volvió aún más evidente con el aumento del uso de dispositivos móviles y la exigencia de los usuarios por aplicaciones que respondan de manera eficiente, sin importar el tipo de dispositivo o la calidad de la conexión a internet.

Tradicionalmente, las aplicaciones podían clasificarse en dos grandes grupos: aplicaciones web y aplicaciones móviles nativas. Las aplicaciones web se ejecutan en el navegador y son independientes del sistema operativo, lo que permite su acceso inmediato mediante una URL. Sin embargo, estas aplicaciones presentan limitaciones importantes como la imposibilidad de trabajar sin conexión, el acceso restringido al hardware del dispositivo, y una experiencia de usuario menos integrada en comparación con las aplicaciones nativas.

Por otro lado, las aplicaciones móviles nativas ofrecen una experiencia de usuario más inmersiva, con acceso completo a las funcionalidades del dispositivo, pero requieren ser descargadas desde una tienda de aplicaciones, lo que introduce barreras de instalación. Además, implican un mayor esfuerzo de desarrollo, ya que es necesario mantener distintas versiones para cada plataforma, como iOS o Android, lo cual incrementa el costo y la complejidad del mantenimiento.

Frente a este panorama, surge un nuevo enfoque promovido por Google en 2015: las Progressive Web Apps (PWA). Este modelo busca combinar las ventajas de las aplicaciones web y móviles nativas, reduciendo al mínimo sus desventajas. Las PWA son sitios web enriquecidos con tecnologías modernas que les permiten comportarse como aplicaciones móviles, permitiendo la instalación en el dispositivo, funcionar sin conexión, enviar notificaciones push, entre otras funcionalidades, todo sin necesidad de pasar por una tienda de aplicaciones.

Evolución hacia las PWA

La evolución tecnológica de la web ha estado marcada por la incorporación de nuevas APIs y estándares que han ampliado enormemente las capacidades del navegador. Con la introducción de tecnologías como Service Workers, Web App Manifests, IndexedDB, Web Storage, Push API y Cache API, ha sido posible construir experiencias web más robustas, confiables y adaptativas. Estas tecnologías conforman la base técnica de las PWA, habilitando funcionalidades que antes solo eran posibles mediante aplicaciones nativas.

Las Service Workers son scripts que el navegador ejecuta en segundo plano, separados de la página web, que permiten interceptar peticiones de red, almacenar recursos en caché, y responder de forma personalizada a eventos como la pérdida de conexión. Esta capacidad permite que las PWA sigan funcionando incluso cuando el usuario no tiene acceso a internet, brindando una experiencia continua y coherente.

Características principales de una PWA

Para ser considerada una Progressive Web App, una aplicación debe cumplir con las siguientes características clave:

- ✓ **Responsiva:** Debe adaptarse correctamente a diferentes tamaños de pantalla como móvil, tablet, escritorio.
- ✓ **Conectividad independiente:** Debe ser capaz de trabajar sin conexión o con conectividad limitada mediante el uso de caché.
- ✓ **Instalable:** Debe poder añadirse al escritorio o pantalla de inicio como si fuera una aplicación nativa.
- ✓ **Segura:** Debe requerir HTTPS para proteger la integridad y confidencialidad de los datos.
- ✓ **Actualizable:** Mediante el Service Worker, debe asegurar que el contenido esté siempre actualizado.
- ✓ **Descubrible:** Debe poder ser encontrada por motores de búsqueda gracias al Web App Manifest.
- ✓ **Re-enganchable:** Debe permitir notificaciones push para mantener al usuario comprometido.

Ventajas de utilizar PWA

La adopción de PWA presenta múltiples beneficios tanto para desarrolladores como para usuarios:

- ✓ **Instalación rápida y sin fricción:** No es necesario acceder a una tienda ni esperar descargas prolongadas.
- ✓ **Menor uso de almacenamiento:** Ocupan menos espacio en el dispositivo en comparación con las aplicaciones nativas.
- ✓ **Desarrollo multiplataforma:** Reducen el tiempo y esfuerzo de mantenimiento, al tener una sola base de código.
- ✓ **Velocidad y rendimiento mejorado:** Debido al almacenamiento en caché inteligente, las PWA cargan más rápido.
- ✓ **Funcionalidad offline:** Fundamental para usuarios con conexiones inestables o intermitentes.

- ✓ **Compatibilidad creciente:** Es compatible con navegadores modernos como Chrome, Firefox, Edge y Opera ya ofrecen soporte amplio para PWA.

Limitaciones y desventajas

A pesar de sus ventajas, las PWA aún enfrentan ciertas limitaciones:

- ✓ **Compatibilidad parcial con iOS:** Apple ha ido ampliando el soporte, pero aún existen restricciones importantes, como el uso limitado de notificaciones push o actualizaciones en segundo plano.
- ✓ **Acceso limitado al hardware:** Aunque cada vez se expande más el acceso, no iguala todavía el alcance de una app nativa.
- ✓ **No son visibles en todas las tiendas:** Aunque existen iniciativas para permitir la publicación de PWA en tiendas, como Google Play, no todas las plataformas las aceptan.
- ✓ **Experiencia de usuario dependiente del navegador:** Puede haber ligeras diferencias en comportamiento o diseño entre navegadores.

Las PWA representan una alternativa moderna y efectiva para crear aplicaciones que funcionen en múltiples entornos sin necesidad de reescribir el código para diferentes plataformas. Su diseño modular y progresivo facilita la adopción en proyectos de pequeña y mediana escala, así como en proyectos académicos y prototipos. Además, las PWA están alineadas con los principios de accesibilidad, eficiencia y universalidad que rigen el desarrollo web actual.

PLANTEAMIENTO DEL PROBLEMA

En el contexto actual de desarrollo de aplicaciones, la disponibilidad constante y el acceso eficiente a servicios digitales son elementos fundamentales para ofrecer una experiencia de usuario satisfactoria. En muchos entornos, especialmente aquellos con limitaciones de conectividad, los sistemas tradicionales basados únicamente en aplicaciones web presentan desventajas importantes, como la imposibilidad de funcionar sin acceso a internet, tiempos de carga elevados y una experiencia de uso limitada.

Durante el desarrollo inicial del sistema bancario simulado, se optó por una arquitectura de microservicios con una interfaz web desarrollada en HTML, CSS y JavaScript. Aunque esta implementación permitía realizar operaciones básicas como autenticación, consulta de saldo, depósitos, retiros y transferencias, su funcionalidad dependía completamente de la disponibilidad de conexión de red y del servidor web local. Esto implicaba que, en ausencia de internet o con una desconexión del servidor, el sistema quedaba completamente inutilizable para el usuario final, incluso para tareas tan simples como visualizar su interfaz.

Además, en un escenario donde la portabilidad y la accesibilidad son cada vez más valoradas, contar con un sistema que no puede instalarse como una aplicación ni ofrecer funcionalidades offline representa una desventaja frente a las soluciones modernas que sí lo permiten. La ausencia de una experiencia de aplicación nativa también limita la integración con el dispositivo del usuario, reduciendo su usabilidad y percepción de calidad.

Estas limitaciones evidenciaron la necesidad de evolucionar hacia una solución más robusta y versátil que permitiera al sistema mantener su funcionalidad esencial aún en condiciones adversas de red, reducir los tiempos de carga mediante el almacenamiento inteligente en caché, y ofrecer una experiencia de uso más cercana a la de una aplicación móvil.

Por lo tanto, se identificó la necesidad de investigar e implementar un enfoque más moderno y resiliente que permitiera mejorar sustancialmente la disponibilidad, la portabilidad y la experiencia del usuario final.

PROPUESTA DE SOLUCIÓN

Para responder a las limitaciones identificadas en la implementación original del sistema bancario basado únicamente en una arquitectura web tradicional, se propuso una solución centrada en la adopción de tecnologías web modernas que ofrecieran una experiencia de usuario más robusta, confiable y accesible. La estrategia elegida consistió en convertir el sistema existente en una aplicación web progresiva (PWA, por sus siglas en inglés).

El enfoque PWA permite que una aplicación web pueda comportarse como una aplicación nativa: puede instalarse directamente desde el navegador, ejecutarse en modo offline y mantenerse funcional incluso en condiciones de conectividad limitada. Para lograr esto, se integraron componentes clave como el Service Worker (para la gestión del almacenamiento en caché y la interceptación de peticiones), el archivo manifest.json (para definir los metadatos de la app y permitir su instalación), y un mecanismo de servidor local (como http-server en Python) que facilitara las pruebas del comportamiento sin conexión.

La transición a PWA no implicó rediseñar completamente la interfaz ni los microservicios existentes, sino más bien adaptar la arquitectura del frontend para hacerla más resiliente y versátil. Esta solución permitió conservar la modularidad del sistema, aprovechar las funcionalidades ya desarrolladas y extender la accesibilidad del sistema a distintos escenarios de uso, incluyendo aquellos con conectividad limitada o intermitente.

MATERIALES Y METODOS

Para la construcción del sistema bancario distribuido con soporte para Progressive Web App (PWA), se utilizaron diversas herramientas y tecnologías organizadas en dos niveles: el backend basado en microservicios y el frontend adaptable como PWA.

Materiales

- **Sistema Operativo**: Windows 11
  Plataforma principal sobre la cual se desarrollaron y ejecutaron todos los componentes del sistema.
- **Entorno de Desarrollo**: NetBeans 21
  IDE empleado para programar los microservicios Java, organizar el proyecto modularmente y gestionar las dependencias del sistema.
- **Lenguaje de Programación Backend**: Java SE 22
  Utilizado para implementar los microservicios, manipular la lógica de negocio y manejar las peticiones HTTP sin frameworks externos, haciendo uso de HttpServer.
- **Base de Datos**: SQLite
  Sistema ligero de almacenamiento persistente utilizado para mantener la información de cuentas bancarias, movimientos y clientes.
- **Cliente HTTP de pruebas**: Postman
  Herramienta utilizada para enviar peticiones HTTP a los microservicios y verificar el comportamiento correcto de las operaciones (login, saldo, transferencia, etc.).
- **Servidor Web Local**: Python (módulo http.server)
  Utilizado para servir la carpeta del cliente web y permitir la ejecución en un entorno controlado sin necesidad de servidores externos.
- **Frontend**: HTML, CSS, JavaScript
  Tecnologías utilizadas para construir la interfaz de usuario del sistema, permitiendo la interacción directa con los microservicios a través de peticiones fetch.
- **Tecnologías PWA**:
  - **Service Worker**: Encargado del almacenamiento en caché y la interceptación de peticiones para habilitar la funcionalidad offline.
  - **Manifest JSON**: Archivo de configuración que define los metadatos de la aplicación para su instalación como app.
  - **Archivos de íconos**: Diseñados para representar visualmente la app en la pantalla de inicio y navegador.

DESARROLLO DE SOLUCIÓN

El desarrollo de la solución se estructuró en dos grandes bloques: la construcción de los microservicios en Java y la implementación de la interfaz web bajo el enfoque de Progressive Web App (PWA). Cada componente fue diseñado para operar de manera independiente pero colaborativa, formando un sistema distribuido funcional, modular y con capacidad de operar sin conexión.

A continuación, se describen los componentes del sistema y su funcionalidad:

LoginService

Este archivo es el encargado de validar las credenciales de acceso. El LoginHandler.java procesa peticiones POST con número de cuenta y NIP. Si la autenticación es exitosa, se responde con un código 200. Además, se implementó una consulta a la base de datos para recuperar el nombre y sexo del titular, información que posteriormente se almacena en localStorage del navegador. Esto ultimo para brindar un mensaje de bienvenida personalizado.

```
public class LoginService {
    public static void main(String[] args) {
        try {
            HttpServer server = HttpServer.create(new InetSocketAddress(8081), 0);
            server.createContext("/login", new LoginHandler());
            server.setExecutor(null);
            server.start();
            System.out.println("LoginService iniciado en http://localhost:8081/login");
        } catch (Exception e) {
            System.out.println("Error al iniciar el servidor: " + e.getMessage());
        }
    }
}
```

```
public void handle(HttpExchange exchange) throws IOException {
    exchange.getResponseHeaders().set("Access-Control-Allow-Origin", "*");
    exchange.getResponseHeaders().set("Access-Control-Allow-Headers", "Content-Type");
    exchange.getResponseHeaders().set("Access-Control-Allow-Methods", "POST, OPTIONS");
    if ("OPTIONS".equalsIgnoreCase(exchange.getRequestMethod())) {
        exchange.sendResponseHeaders(204, -1);
        return;
    }
    if (!"POST".equalsIgnoreCase(exchange.getRequestMethod())) {
        exchange.sendResponseHeaders(405, -1);
        return;
    }
    InputStreamReader isr = new InputStreamReader(exchange.getRequestBody(), "utf-8");
    BufferedReader br = new BufferedReader(isr);
    StringBuilder jsonBuilder = new StringBuilder();
    String linea;
    while ((linea = br.readLine()) != null) {
        jsonBuilder.append(linea);
    }
    Gson gson = new Gson();
    Login login = gson.fromJson(jsonBuilder.toString(), Login.class);
    boolean valido = BaseDatos.validarCuenta(login.getNumero(), login.getNip());
    String respuestaJson;
    int codigo;
    if (valido) {
        Titular titular = BaseDatos.consultarNombreSexo(login.getNumero());
        if (titular != null) {
            respuestaJson = gson.toJson(new RespuestaLogin(
                "Autenticacion Exitosa",titular.getNombre(),titular.getSexo()));
            codigo = 200;
        } else {
            respuestaJson = gson.toJson(new RespuestaLogin("Error interno"));
            codigo = 500;
        }
    } else {
        respuestaJson = gson.toJson(new RespuestaLogin("Error en autenticación"));

            codigo = 401;
        }
        exchange.sendResponseHeaders(codigo, respuestaJson.getBytes().length);
        OutputStream os = exchange.getResponseBody();
        os.write(respuestaJson.getBytes());
        os.close();
```

Manejador que recibe las solicitudes de autenticación, valida el número de cuenta y NIP, y responde con el nombre y sexo del titular si los datos son correctos.

```java
public class Login {
    private String numero;
    private int nip;

    public Login() {}
    public String getNumero() {
        return numero;
    }
    public void setNumero(String numero) {
        this.numero = numero;
    }
    public int getNip() {
        return nip;
    }
    public void setNip(int nip) {
        this.nip = nip;
    }
}
```

Objeto que representa las credenciales enviadas desde el frontend, compuesto por el número de cuenta y el NIP del usuario.

```java
public class Titular {
    private String nombre;
    private String sexo;

    public Titular(String nombre, String sexo) {
        this.nombre = nombre;
        this.sexo = sexo;
    }
    public String getNombre() {
        return nombre;
    }
    public String getSexo() {
        return sexo;
    }
}
```

Objeto que contiene los datos del titular de la cuenta (nombre y sexo), utilizado para personalizar la experiencia del usuario tras una autenticación exitosa.

AccountService

Contiene el SaldoHandler.java que responde a peticiones GET para consultar el saldo de una cuenta, y a peticiones POST para actualizarlo. Utiliza funciones del archivo BaseDatos.java, que se conecta mediante JDBC a una base de datos SQLite.

```java
public class AccountService {
    public static void main(String[] args) {
        try {
            HttpServer server = HttpServer.create(new InetSocketAddress(8082), 0);
            server.createContext("/saldo", new SaldoHandler());
            server.setExecutor(null);
            server.start();
            System.out.println("AccountService iniciado en http://localhost:8082/saldo");
        } catch (Exception e) {
            System.out.println("Error al iniciar AccountService: " + e.getMessage());
        }
    }
}
```

```java
public void handle(HttpExchange exchange) throws IOException {
    exchange.getResponseHeaders().set("Access-Control-Allow-Origin", "*");
    exchange.getResponseHeaders().set("Access-Control-Allow-Headers", "Content-Type");
    exchange.getResponseHeaders().set("Access-Control-Allow-Methods", "GET, POST, OPTIONS");
    String metodo = exchange.getRequestMethod();
    if ("OPTIONS".equalsIgnoreCase(metodo)) {
        exchange.sendResponseHeaders(204, -1);
        return;
    }
    if ("GET".equalsIgnoreCase(metodo)) {
        manejarConsultaSaldo(exchange);
    } else if ("POST".equalsIgnoreCase(metodo)) {
        manejarActualizacionSaldo(exchange);
    } else {
        exchange.sendResponseHeaders(405, -1);
    }
}
```

Es el manejador de solicitudes HTTP para consultar y actualizar el saldo de una cuenta bancaria. Define los métodos GET y POST para estas operaciones.

```java
private void manejarConsultaSaldo(HttpExchange exchange) throws IOException {
    URI uri = exchange.getRequestURI();
    String query = uri.getQuery();
    String cuenta = null;
    if (query != null && query.startsWith("cuenta=")) {
        cuenta = query.substring("cuenta=".length());
    }
    String respuesta;
    int codigo;
    if (cuenta != null) {
        double saldo = BaseDatos.consultarSaldo(cuenta);
        if (saldo >= 0) {
            respuesta = String.valueOf(saldo);
            codigo = 200;
        } else {
            respuesta = "Cuenta no encontrada";
            codigo = 404;
        }
    } else {
        respuesta = "Parámetro 'cuenta' faltante";
        codigo = 400;
    }
    enviarRespuesta(exchange, codigo, respuesta);
}
```

Método que atiende las solicitudes GET, obtiene el número de cuenta desde la URL y devuelve el saldo actual si la cuenta existe.

```java
private void manejarActualizacionSaldo(HttpExchange exchange) throws IOException {
    InputStreamReader isr = new InputStreamReader(exchange.getRequestBody(), "utf-8");
    BufferedReader br = new BufferedReader(isr);
    StringBuilder json = new StringBuilder();
    String linea;
    while ((linea = br.readLine()) != null) {
        json.append(linea);
    }
    ActualizarSaldoDTO datos = gson.fromJson(json.toString(), ActualizarSaldoDTO.class);
    boolean actualizado = BaseDatos.actualizarSaldo(datos.getCuenta(), datos.getNuevoSaldo());
    String respuesta = actualizado ? "Saldo actualizado correctamente" : "Error: cuenta no encontrada";
    int codigo = actualizado ? 200 : 404;
    enviarRespuesta(exchange, codigo, respuesta);
}
```

Método que atiende solicitudes POST, recibe un JSON con el número de cuenta y el nuevo saldo, y lo actualiza en la base de datos si la cuenta es válida.

```java
public class ActualizarSaldoDTO {
    private String cuenta;
    private double nuevoSaldo;
    public ActualizarSaldoDTO(String cuenta, double nuevoSaldo) {
        this.cuenta = cuenta;
        this.nuevoSaldo = nuevoSaldo;
    }
    public String getCuenta() {
        return cuenta;
    }
    public void setCuenta(String cuenta) {
        this.cuenta = cuenta;
    }
    public double getNuevoSaldo() {
        return nuevoSaldo;
    }
    public void setNuevoSaldo(double nuevoSaldo) {
        this.nuevoSaldo = nuevoSaldo;
    }
}
```

Objeto que representa la estructura de los datos enviados para actualizar el saldo. Contiene dos campos: cuenta (número de cuenta) y nuevoSaldo

TransactionService

Maneja las operaciones de depósito, retiro y transferencia mediante OperacionHandler.java. Internamente consulta el saldo, valida los montos y actualiza el saldo si es correcto. Para garantizar la coherencia en ambientes concurrentes, se conecta a ConcurrencyService para bloquear cuentas durante operaciones sensibles.

```java
public class TransactionService {
    public static void main(String[] args) {
        try {
            HttpServer server = HttpServer.create(new InetSocketAddress(8083), 0);
            server.createContext("/deposito", new OperacionHandler("deposito"));
            server.createContext("/retiro", new OperacionHandler("retiro"));
            server.createContext("/transferencia", new OperacionHandler("transferencia"));
            server.setExecutor(null);
            server.start();
            System.out.println("TransactionService iniciado en http://localhost:8083");
        } catch (Exception e) {
            System.out.println("Error al iniciar TransactionService: " + e.getMessage());
        }
    }
}
```

```java
    public OperacionHandler(String tipoOperacion) {
        this.tipoOperacion = tipoOperacion;
    }

    @Override
    public void handle(HttpExchange exchange) throws IOException {
        exchange.getResponseHeaders().set("Access-Control-Allow-Origin", "*");
        exchange.getResponseHeaders().set("Access-Control-Allow-Headers", "Content-Type");
        exchange.getResponseHeaders().set("Access-Control-Allow-Methods", "POST, OPTIONS");
        if ("OPTIONS".equalsIgnoreCase(exchange.getRequestMethod())) {
            exchange.sendResponseHeaders(204, -1);
            return;
        }
        if (!"POST".equalsIgnoreCase(exchange.getRequestMethod())) {
            exchange.sendResponseHeaders(405, -1);
            return;
        }
        BufferedReader br = new BufferedReader(new InputStreamReader(exchange.getRequestBody()));
        StringBuilder body = new StringBuilder();
        String linea;
        while ((linea = br.readLine()) != null) {
            body.append(linea);
        }
        if (tipoOperacion.equals("transferencia")) {
            TransferenciaDTO dto = gson.fromJson(body.toString(), TransferenciaDTO.class);
            procesarTransferencia(exchange, dto);
        } else {
            OperacionDTO dto = gson.fromJson(body.toString(), OperacionDTO.class);
            if (tipoOperacion.equals("deposito")) {
                procesarDeposito(exchange, dto);
            } else if (tipoOperacion.equals("retiro")) {
                procesarRetiro(exchange, dto);
            }
        }
    }
```

Es el manejador principal del microservicio de transacciones. Recibe solicitudes POST para realizar depósitos, retiros o transferencias, y se comunica con otros microservicios como el de saldo y concurrencia.

```
private void procesarDeposito(HttpExchange exchange, OperacionDTO dto) throws IOException {
    System.out.println(">> Cuenta: " + dto.getCuenta());
    System.out.println(">> Monto recibido: " + dto.getMonto());
    double saldoActual = consultarSaldo(dto.getCuenta());
    if (saldoActual < 0) {
        enviarRespuesta(exchange, 404, "Cuenta no encontrada");
        return;
    }
    double nuevoSaldo = saldoActual + dto.getMonto();
    if (actualizarSaldo(dto.getCuenta(), nuevoSaldo)) {
        BaseDatos.registrarMovimiento(dto.getCuenta(), "Deposito", dto.getMonto());
        enviarRespuesta(exchange, 200, "Depósito exitoso. Nuevo saldo: $" + nuevoSaldo);
    } else {
        enviarRespuesta(exchange, 500, "Error al actualizar saldo");
    }
}
```

Realiza la operación de depósito. Consulta el saldo actual, suma el monto recibido y actualiza el nuevo saldo en el microservicio de saldo. También registra el movimiento.

```
private void procesarRetiro(HttpExchange exchange, OperacionDTO dto) throws IOException {
    if (!solicitarAcceso(dto.getCuenta())) {
        enviarRespuesta(exchange, 423, "La cuenta está en uso. Intente más tarde.");
        return;
    }
    try {
        double saldoActual = consultarSaldo(dto.getCuenta());
        if (saldoActual < 0) {
            enviarRespuesta(exchange, 404, "Cuenta no encontrada");
            return;
        }
        if (dto.getMonto() > saldoActual) {
            enviarRespuesta(exchange, 400, "Saldo insuficiente");
            return;
        }
        double nuevoSaldo = saldoActual - dto.getMonto();
        if (actualizarSaldo(dto.getCuenta(), nuevoSaldo)) {
            BaseDatos.registrarMovimiento(dto.getCuenta(), "Retiro", dto.getMonto());
            enviarRespuesta(exchange, 200, "Retiro exitoso. Nuevo saldo: $" + nuevoSaldo);
        } else {
            enviarRespuesta(exchange, 500, "Error al actualizar saldo");
        }
    } finally {
        liberarAcceso(dto.getCuenta());
    }
}
```

Ejecuta un retiro de cuenta. Solicita acceso exclusivo (sincronización), valida el saldo, descuenta el monto y actualiza el saldo. Finalmente, libera el acceso y registra el movimiento.

```java
private double consultarSaldo(String cuenta) {
    try {
        URL url = new URL("http://localhost:8082/saldo?cuenta=" + cuenta);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        if (conn.getResponseCode() == 200) {
            BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStream()));
            return Double.parseDouble(in.readLine());
        }
    } catch (Exception e) {
        System.out.println("Error al consultar saldo: " + e.getMessage());
    }
    return -1;
}
```

Método auxiliar que envía una solicitud GET al microservicio de saldo para obtener el saldo actual de una cuenta.

```java
private boolean actualizarSaldo(String cuenta, double nuevoSaldo) {
    try {
        URL url = new URL("http://localhost:8082/saldo/actualizar");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-Type", "application/json");
        conn.setDoOutput(true);
        String json = gson.toJson(new ActualizarSaldoDTO(cuenta, nuevoSaldo));
        OutputStream os = conn.getOutputStream();
        os.write(json.getBytes());
        os.flush();
        os.close();
        System.out.println("Actualizando saldo en cuenta: " + cuenta);
        System.out.println("Nuevo saldo en BD: " + nuevoSaldo);
        System.out.println("Código de respuesta: " + conn.getResponseCode());
        return conn.getResponseCode() == 200;
    } catch (Exception e) {
        System.out.println("Error al actualizar saldo: " + e.getMessage());
        return false;
    }
}
```

Método auxiliar que envía una solicitud POST al microservicio de saldo con el nuevo saldo a actualizar en una cuenta específica.

ConcurrencyService

Gestiona el bloqueo y liberación de cuentas para evitar condiciones de carrera. Define rutas /acceso y /liberar, y mantiene en memoria las cuentas ocupadas.

```java
public class ConcurrencyService {
    public static void main(String[] args) {
        try {
            HttpServer server = HttpServer.create(new InetSocketAddress(8084), 0);
            server.createContext("/acceso", new AccesoHandler(true));
            server.createContext("/liberar", new AccesoHandler(false));
            server.setExecutor(null);
            server.start();
            System.out.println("ConcurrencyService iniciado en http://localhost:8084");
        } catch (Exception e) {
            System.out.println("Error al iniciar ConcurrencyService: " + e.getMessage());
        }
    }
}
```

```java
public void handle(HttpExchange exchange) throws IOException {
    if (!"POST".equalsIgnoreCase(exchange.getRequestMethod())) {
        exchange.sendResponseHeaders(405, -1); // Método no permitido
        return;
    }
    BufferedReader reader = new BufferedReader(new InputStreamReader(exchange.getRequestBody()));
    StringBuilder json = new StringBuilder();
    String linea;
    while ((linea = reader.readLine()) != null) {
        json.append(linea);
    }
    CuentaBloqueoDTO dto = gson.fromJson(json.toString(), CuentaBloqueoDTO.class);
    String cuenta = dto.getCuenta();
    String respuesta;
    int codigo;
    if (esSolicitud) {
        boolean concedido = BloqueoManager.obtenerAcceso(cuenta);
        if (concedido) {
            respuesta = "ACCESO_CONCEDIDO";
            codigo = 200;
        } else {
            respuesta = "CUENTA_EN_USO";
            codigo = 423; // Locked
        }
    } else {
        BloqueoManager.liberarAcceso(cuenta);
        respuesta = "ACCESO_LIBERADO";
        codigo = 200;
    }
    byte[] resp = respuesta.getBytes();
    exchange.sendResponseHeaders(codigo, resp.length);
    OutputStream os = exchange.getResponseBody();
    os.write(resp);
    os.close();
}
```

Es el manejador principal del microservicio de concurrencia. Se encarga de controlar el acceso exclusivo a cuentas bancarias durante operaciones críticas como retiros y transferencias, evitando condiciones de carrera y conflictos de escritura concurrente.

```
private boolean solicitarAcceso(String cuenta) {
    System.out.println("LLAMANDO A solicitarAcceso PARA: " + cuenta);
    try {
        for (int i = 0; i < 3; i++) {
            URL url = new URL("http://localhost:8084/acceso");
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-Type", "application/json");
            conn.setDoOutput(true);
            String json = gson.toJson(new OperacionDTO(cuenta, 0));
            try (OutputStream os = conn.getOutputStream()) {
                os.write(json.getBytes());
                os.flush();
            }
            int code = conn.getResponseCode();
            if (code == 200) {
                return true;
            }
            Thread.sleep(500);
        }
    } catch (Exception e) {
        System.out.println("Error al solicitar acceso de concurrencia: " + e.getMessage());
    }
    return false;
}
```

Método que registra en una estructura de datos (generalmente un Set) que una cuenta está en uso. Si la cuenta ya está ocupada, responde con un código que indica bloqueo (423), de lo contrario, permite el acceso.

```
private void liberarAcceso(String cuenta) {
    System.out.println("LLAMANDO A liberarAcceso PARA: " + cuenta);
    try {
        URL url = new URL("http://localhost:8084/liberar");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-Type", "application/json");
        conn.setDoOutput(true);
        String json = gson.toJson(new OperacionDTO(cuenta, 0));
        try (OutputStream os = conn.getOutputStream()) {
            os.write(json.getBytes());
            os.flush();
        }
        System.out.println("Se envió liberación para cuenta: " + cuenta);
        System.out.println("Código respuesta liberar: " + conn.getResponseCode());
    } catch (Exception e) {
        System.out.println("Error al liberar acceso de concurrencia: " + e.getMessage());
    }
}
```

Método que remueve la cuenta del registro de bloqueos, liberando así el acceso para que otras operaciones puedan ejecutarse sobre esa cuenta.

MovementsService

Define MovimientosHandler.java que responde a solicitudes GET para devolver los últimos movimientos de una cuenta. Se conecta a la base de datos mediante BaseDatos.java y entrega una lista en formato JSON.

```
public class MovementsService {
    public static void main(String[] args) {
        try {
            HttpServer server = HttpServer.create(new InetSocketAddress(8085), 0);
            server.createContext("/movimientos", new MovimientosHandler());
            server.setExecutor(null);
            server.start();
            System.out.println("MovementsServices iniciado en http://localhost:8085/movimientos");
        } catch (Exception e) {
            System.out.println("Error al iniciar AccountService: " + e.getMessage());
        }
    }
}
```

```
    public void handle(HttpExchange exchange) throws IOException {
        exchange.getResponseHeaders().set("Access-Control-Allow-Origin", "*");
        exchange.getResponseHeaders().set("Access-Control-Allow-Methods", "GET, OPTIONS");
        String metodo = exchange.getRequestMethod();
        if ("OPTIONS".equalsIgnoreCase(metodo)) {
            exchange.sendResponseHeaders(204, -1);
            return;
        }
        if (!"GET".equalsIgnoreCase(metodo)) {
            exchange.sendResponseHeaders(405, -1);
            return;
        }
        URI uri = exchange.getRequestURI();
        String query = uri.getQuery();
        String cuenta = null;
        if (query != null && query.startsWith("cuenta=")) {
            cuenta = query.substring("cuenta=".length());
        }
        if (cuenta == null || cuenta.isEmpty()) {
            enviarRespuesta(exchange, 400, "Cuenta no especificada");
            return;
        }
        List<Map<String, String>> movimientos = BaseDatos.consultarMovimientos(cuenta);
        String respuestaJson = gson.toJson(movimientos);
        enviarRespuesta(exchange, 200, respuestaJson);
    }
```

Este manejador se encarga de recibir solicitudes HTTP para consultar los movimientos de una cuenta. Opera como un punto de entrada al microservicio de historial de transacciones, utilizando la cuenta como parámetro y devolviendo un listado en formato JSON con los últimos movimientos registrados, incluyendo tipo, monto y fecha.

BaseDatos

Archivo compartido entre servicios que se conecta a Banco.db. Este contiene diversas funciones donde cada una ejecuta consultas SQL usando PreparedStatement para mayor seguridad y eficiencia.

```
public static boolean validarCuenta(String numero, int nip) {
    String sql = "SELECT * FROM cuentas WHERE numero = ? AND nip = ?";
    try (Connection conn = DriverManager.getConnection(BD);
         PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setString(1, numero);
        stmt.setInt(2, nip);
        ResultSet rs = stmt.executeQuery();
        return rs.next();
    } catch (SQLException e) {
        System.out.println("Error en la base de datos: " + e.getMessage());
        return false;
    }
}
```

Verifica si existe una cuenta con el número y NIP proporcionados. Se usa en la autenticación.

```
public static Titular consultarNombreSexo(String cuenta) {
    String sqlID = "SELECT titular_id FROM Cuentas WHERE numero = ?";
    String sqlNombre = "SELECT nombre,sexo FROM Clientes WHERE id = ?";
    try (Connection conn = DriverManager.getConnection(BD);
        PreparedStatement stmt1 = conn.prepareStatement(sqlID)) {
        stmt1.setString(1, cuenta);
        ResultSet rs1 = stmt1.executeQuery();
        if (rs1.next()) {
            int id = rs1.getInt("titular_id");
            try (PreparedStatement stmt2 = conn.prepareStatement(sqlNombre)) {
                stmt2.setInt(1, id);
                ResultSet rs2 = stmt2.executeQuery();
                if (rs2.next()) {
                    String nombre = rs2.getString("nombre");
                    String sexo = rs2.getString("sexo");
                    return new Titular(nombre, sexo);
                }
            }
        }
    } catch (SQLException e) {
        System.out.println("Error en la base de datos (consultarNombreSexo): " + e.getMessage());
    }
    return null;
}
```

Devuelve un objeto Titular con el nombre y sexo del dueño de la cuenta, usado para personalizar la bienvenida en la interfaz.

```java
public static double consultarSaldo(String cuenta) {
    String sql = "SELECT saldo FROM Cuentas WHERE numero = ?";
    try (Connection conn = DriverManager.getConnection(BD);
        PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setString(1, cuenta);
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
            return rs.getDouble("saldo");
        } else {
            return -1; // Cuenta no encontrada
        }
    } catch (SQLException e) {
        System.out.println("Error al consultar saldo: " + e.getMessage());
        return -1;
    }
}
```

Devuelve el saldo actual de una cuenta. Se usa para mostrar el saldo y validar retiros o transferencias.

```java
public static boolean actualizarSaldo(String cuenta, double nuevoSaldo) {
    String sql = "UPDATE Cuentas SET saldo = ? WHERE numero = ?";
    try (Connection conn = DriverManager.getConnection(BD);
        PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setDouble(1, nuevoSaldo);
        stmt.setString(2, cuenta);
        int filasAfectadas = stmt.executeUpdate();
        return filasAfectadas > 0;
    } catch (SQLException e) {
        System.out.println("Error al actualizar saldo: " + e.getMessage());
        return false;
    }
}
```

Actualiza el saldo de una cuenta con el nuevo valor. Se emplea después de un depósito, retiro o transferencia.

```java
public static void registrarMovimiento(String cuenta, String tipo, double monto) {
    String sql = "INSERT INTO Movimientos (cuenta, tipo, monto, fecha) VALUES (?, ?, ?, datetime('now'))";
    try (Connection conn = DriverManager.getConnection(BD);
        PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setString(1, cuenta);
        stmt.setString(2, tipo);
        stmt.setDouble(3, monto);
        stmt.executeUpdate();
    } catch (SQLException e) {
        System.out.println("Error al registrar movimiento: " + e.getMessage());
    }
}
```

Inserta un registro en la tabla de movimientos, indicando el tipo de operación, monto y fecha. Se invoca al completar depósitos, retiros o transferencias.

```java
public static void registrarTransferencia(String origen, String destino, double monto) {
    String sql = "INSERT INTO Transferencias (cuenta_origen, cuenta_destino, monto, fecha) VALUES (?, ?, ?, datetime('now'))";
    try (Connection conn = DriverManager.getConnection(BD);
        PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setString(1, origen);
        stmt.setString(2, destino);
        stmt.setDouble(3, monto);
        stmt.executeUpdate();
    } catch (SQLException e) {
        System.out.println("Error al registrar transferencia: " + e.getMessage());
    }
}
```

Guarda los detalles de una transferencia entre cuentas, incluyendo origen, destino, monto y fecha.

```java
public static List<Map<String, String>> consultarMovimientos(String cuenta) {
    List<Map<String, String>> lista = new ArrayList<>();
    String sql = "SELECT tipo, monto, fecha FROM Movimientos WHERE cuenta = ? ORDER BY fecha DESC LIMIT 10";
    try (Connection conn = DriverManager.getConnection(BD);
        PreparedStatement stmt = conn.prepareStatement(sql)) {
        stmt.setString(1, cuenta);
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Map<String, String> movimiento = new HashMap<>();
            movimiento.put("tipo", rs.getString("tipo"));
            movimiento.put("monto", String.valueOf(rs.getDouble("monto")));
            movimiento.put("fecha", rs.getString("fecha"));
            lista.add(movimiento);
        }
    } catch (SQLException e) {
        System.out.println("Error al consultar movimientos: " + e.getMessage());
    }

    return lista;
}
```

Recupera los últimos 10 movimientos de una cuenta, ordenados por fecha descendente. Devuelve una lista de mapas con la información necesaria para ser transformada en formato JSON por el microservicio de movimientos.

Interfaz Web (ClienteWeb)

El frontend está organizado en carpetas:

- ✓ /html/ contiene index.html (login) y dashboard.html (panel principal).
- ✓ /css/ contiene estilo.css, donde se define el diseño responsivo y minimalista.
- ✓ /js/ incluye login.js, banco.js y pwa.js.

index.html y login.js

Contiene un formulario para capturar el número de cuenta y NIP. Al enviarlo, login.js realiza una petición POST al microservicio de login. Si es exitosa, almacena la información del usuario en localStorage y redirige al dashboard.

```html
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Banco - Iniciar Sesión</title>
  <link rel="manifest" href="manifest.json">
  <meta name="theme-color" content="#0044cc">
  <link rel="stylesheet" href="css/estilo.css">
  <link rel="icon" href="icons/favicon.ico">
</head>
<body>
  <div class="container">
      <h1>Bienvenid@ a tu banca en linea</h1>
      <form id="loginForm">
        <label for="cuenta">Número de cuenta:</label><br>
        <input type="text" id="cuenta" name="cuenta" required><br><br>
        <label for="nip">NIP:</label><br>
        <input type="password" id="nip" name="nip" required><br><br>
        <button type="submit">Iniciar Sesión</button>
      </form>
      <p id="linkInstalar" style="text-align: center; margin-top: 10px; display: none;">
        ¿Quieres instalar la app? <a href="#" onclick="instalarApp()">Haz clic aquí</a>
      </p>
      <p id="mensaje" style="color: red;"></p>
  </div>
  <script src="js/login.js"></script>
  <script src="js/pwa.js"></script>
</body>
</html>
```

```
document.getElementById("loginForm").addEventListener("submit", function (e) {
  e.preventDefault();
  const cuenta = document.getElementById("cuenta").value;
  const nip = parseInt(document.getElementById("nip").value);
  const mensaje = document.getElementById("mensaje");
  fetch("http://localhost:8081/login", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ numero: cuenta, nip: nip })
  })
      .then(response => response.json())
      .then(data => {
      if (data.mensaje && data.mensaje.includes("Exitosa")) {
        // Guardar en localStorage
        localStorage.setItem("cuentaActiva", cuenta);
        localStorage.setItem("nombreActivo", data.nombre);
        localStorage.setItem("sexoActivo", data.sexo);
        window.location.href = "html/dashboard.html";
      } else {
        mensaje.textContent = "Error: " + (data.mensaje || "Respuesta inválida");
      }
    })
    .catch(error => {
      mensaje.textContent = "No se pudo conectar con el servidor. "+error;
      console.error(error);
    });
});
```

dashboard.html y banco.js

El panel principal tiene botones para cada operación. Al hacer clic, se abre un modal (ventana emergente) gestionado por banco.js.

Operaciones como consultar saldo, depositar, retirar, transferir y ver movimientos se realizan mediante peticiones fetch a los respectivos microservicios. En cada caso se construye la interfaz de manera dinámica.

```html
<body>
  <div class="container">
      <h1 id="saludo"></h1>
      <p id="cuentaActiva"></p>
      <button onclick="cerrarSesion()">Cerrar sesión</button>
      <h2>Operaciones</h2>
      <button onclick="abrirModal('saldo')">Consultar Saldo</button>
      <button onclick="abrirModal('deposito')">Depositar</button>
      <button onclick="abrirModal('retiro')">Retirar</button>
      <button onclick="abrirModal('transferencia')">Transferir</button>
      <button onclick="consultarMovimientos()">Movimientos</button>
      <div class="modal" id="modal">
        <div class="modal-content" id="modalContenido">
          <span class="close" onclick="cerrarModal()">&times;</span>
          <div id="contenidoModal"></div>
        </div>
      </div>
      <div class="modal" id="modalMensaje">
        <div class="modal-content">
          <span class="close" onclick="cerrarMensaje()">&times;</span>
          <p id="mensaje"></p>
        </div>
      </div>
  </div>
  <script src="../js/banco.js"></script>
  <script src="../js/pwa.js"></script>
</body>
```

```
function realizarOperacion(tipo) {
  let body = {};
  let url = "";
  if (tipo === "deposito") {
    const monto = parseFloat(document.getElementById("montoDeposito").value);
    if (isNaN(monto) || monto <= 0) return mostrarMensaje("Monto inválido", "red");
    body = { cuenta, monto };
    url = "http://localhost:8083/deposito";
  }
  if (tipo === "retiro") {
    const monto = parseFloat(document.getElementById("montoRetiro").value);
    if (isNaN(monto) || monto <= 0) return mostrarMensaje("Monto inválido", "red");
    body = { cuenta, monto };
    url = "http://localhost:8083/retiro";
  }
  if (tipo === "transferencia") {
    const destino = document.getElementById("cuentaDestino").value;
    const monto = parseFloat(document.getElementById("montoTransferencia").value);
    if (!destino || destino === cuenta) return mostrarMensaje("Cuenta destino inválida", "red");
    if (isNaN(monto) || monto <= 0) return mostrarMensaje("Monto inválido", "red");
    body = {
      cuentaOrigen: cuenta,
      cuentaDestino: destino,
      monto
    };
    url = "http://localhost:8083/transferencia";
  }
  fetch(url, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(body)
  })
    .then(res => res.text())
    .then(mensajeServidor => mostrarMensaje(mensajeServidor, "green"))
    .catch(() => mostrarMensaje("Error en la operación", "red"));
}
```

```
function consultarSaldo() {
  fetch(`http://localhost:8082/saldo?cuenta=${cuenta}`)
    .then(res => res.text())
    .then(saldo => {
      contenidoModal.innerHTML = `
        <h3>Saldo disponible</h3>
        <p style="font-size: 24px; text-align: center;"><strong>$${saldo}</strong></p>
      `;
      modal.style.display = "flex";
    })
    .catch(() => mostrarMensaje("Error al consultar saldo", "red"));
}
```

```
function consultarMovimientos() {
  fetch(`http://localhost:8085/movimientos?cuenta=${cuenta}`)
    .then(res => res.json())
    .then(data => {
      let html = `
        <h3>Últimos movimientos</h3>
        <div style="max-height:300px; overflow-y:auto;">
          <ul style="list-style: none; padding: 0; margin: 0;">
      `;
      if (data.length === 0) {
        html += `<li style="text-align:center; padding:10px;">Sin movimientos registrados.</li>`;
      } else {
        data.forEach(m => {
          const tipo = m.tipo.toLowerCase();
          const monto = parseFloat(m.monto).toFixed(2);
          let color = "green";
          let signo = "+";
          if (tipo === "retiro" || tipo === "transferencia") {
            color = "red";
            signo = "-";
          }
          html += `
            <li style="padding: 10px; border-bottom: 1px solid #ccc;">
              <div style="display: flex; justify-content: space-between;">
                <span><strong>${m.tipo}</strong></span>
                <span style="color:${color};"><strong>${signo}$${monto}</strong></span>
              </div>
              <small style="display: block; color: #555; margin-top: 5px;">${m.fecha}</small>
            </li>
          `;
        });
      }
      html += "</ul></div>";
      contenidoModal.innerHTML = html;
      modal.style.display = "flex";
    })
    .catch(() => mostrarMensaje("Error al obtener movimientos", "red"));
```

Service Worker y Manifest

- service-worker.js intercepta las peticiones del frontend. Cachea los archivos esenciales (HTML, CSS, JS, íconos). Si no hay conexión, responde desde el caché.

```
const CACHE_NAME = 'banco-pwa-cache-v1';
const urlsToCache = [
  '/',
  '/index.html',
  '/html/dashboard.html',
  '/css/estilo.css',
  '/js/login.js',
  '/js/banco.js',
  '/js/pwa.js',
  '/manifest.json',
  '/icons/icon-192.png',
  '/icons/icon-512.png'
];
self.addEventListener('install', (event) => {
  console.log('[Service Worker] Instalando...');
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => cache.addAll(urlsToCache))
  );
});
self.addEventListener('activate', (event) => {
  console.log('[Service Worker] Activando...');
  event.waitUntil(
    caches.keys().then(keys => Promise.all(
      keys.filter(key => key !== CACHE_NAME)
          .map(key => caches.delete(key))
    ))
  );
});
```

- manifest.json define el comportamiento de la app al ser instalada: nombre, íconos, color del tema y orientación.

```json
{
  "name": "Banco Distribuido",
  "short_name": "Banco",
  "start_url": "html/index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#0044cc",
  "description": "Aplicación bancaria distribuida basada en microservicios.",
  "icons": [
    {
      "src": "icons/icon-192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "icons/icon-512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ],
  "scope": "/",
  "orientation": "portrait"
}
```

```javascript
self.addEventListener('fetch', (event) => {
  const request = event.request;
  if (request.url.startsWith(self.location.origin)) {
    event.respondWith(
      caches.match(request)
        .then(response => {
          return response || fetch(request);
        })
        .catch(() => {
          console.warn('[Service Worker] Recurso no encontrado en cache ni red:', request.url);
          return caches.match('/index.html');
        })
    );
  } else {
    console.log('[Service Worker] Petición externa no manejada:', request.url);
  }
});
```

Funcionalidad Offline (PWA)

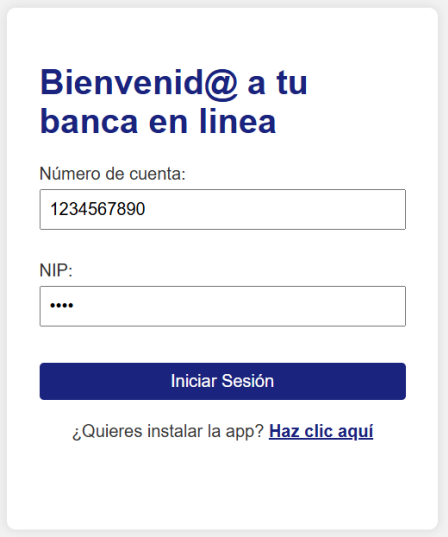
Cuando el servidor web (levantado con python -m http.server) es detenido, la interfaz sigue disponible gracias al caché del Service Worker. Si se intenta realizar una operación como consultar saldo, se produce un error controlado indicando que no hay conexión al backend.

RESULTADOS

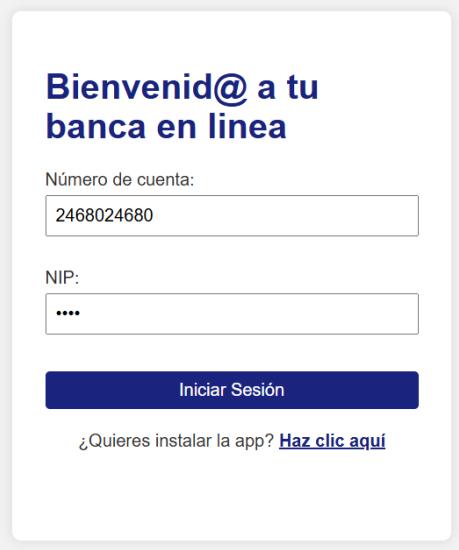
AUTENTICACION

Se verificó el proceso de autenticación utilizando dos cuentas distintas. En ambos casos, el usuario ingresó su número de cuenta y NIP a través de la interfaz web, enviando la solicitud al microservicio correspondiente (LoginService). El sistema respondió de forma adecuada: permitió el acceso solo a usuarios con credenciales válidas y rechazó intentos con datos incorrectos, retornando el mensaje correspondiente.

Autenticación de cliente A

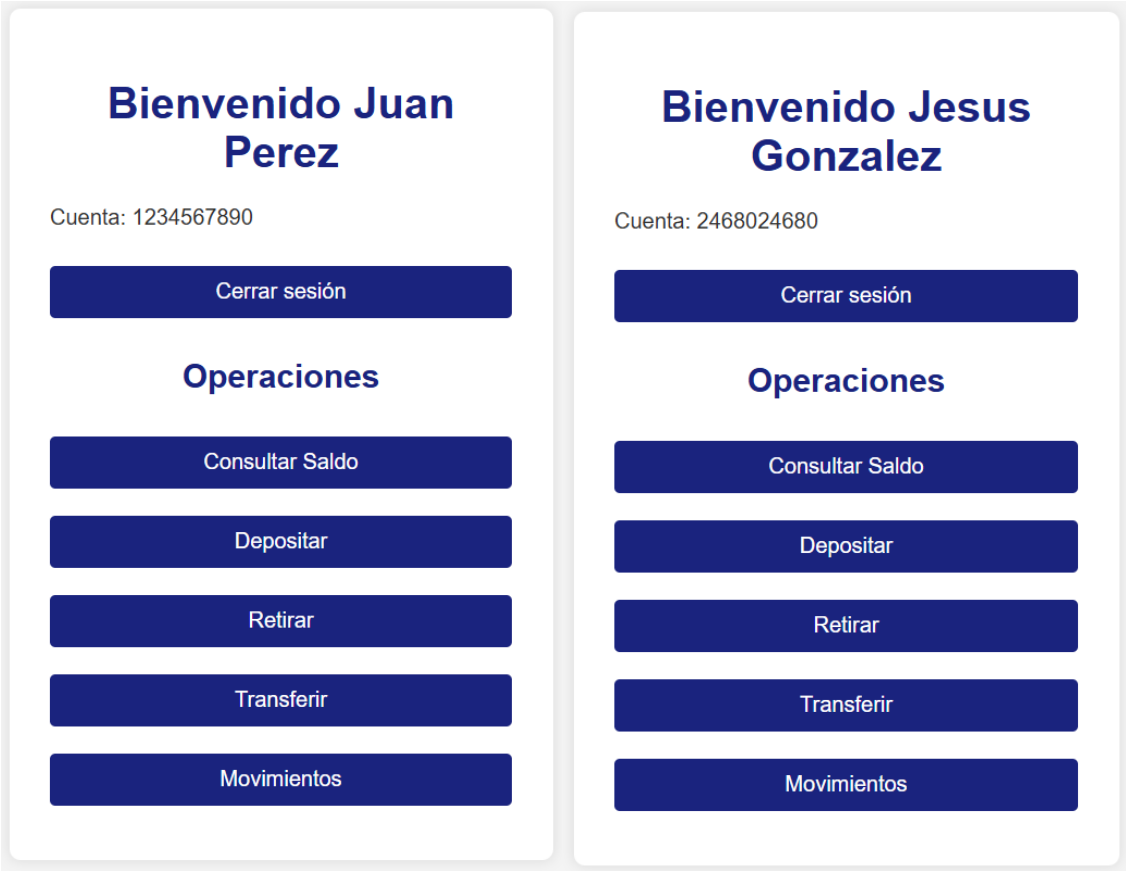


Autenticación de cliente B



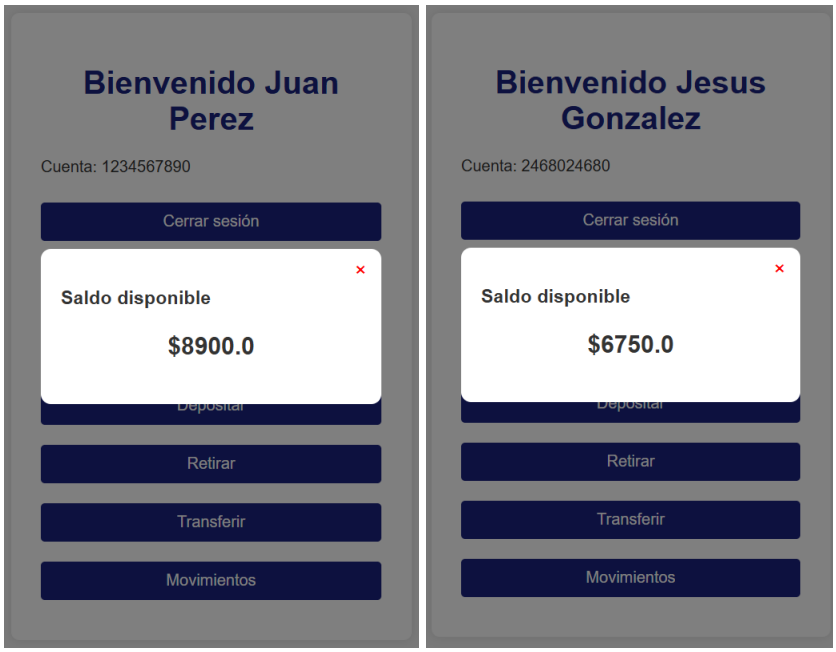
PANTALLA PRINCIPAL

Una vez autenticados, ambos clientes accedieron a la pantalla principal donde obtienen el menú de todas las posible operaciones de pueden realizar, como consulta de saldo, realización de un depósito, realización de un retiro, envió de dinero por medio de transferencia y su historial de últimos movimientos



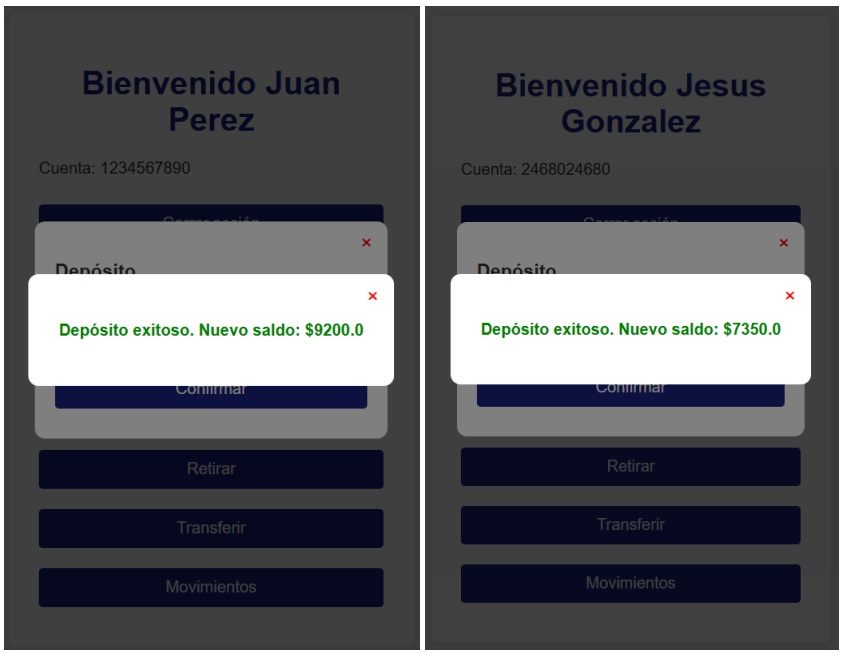
CONSULTA DE SALDO INICIAL

Desde la pantalla principal, ambos clientes accedieron a la funcionalidad de consulta de saldo. Esta acción permitió verificar el estado actual de sus cuentas antes de realizar cualquier operación. El microservicio AccountService respondió con el saldo almacenado en la base de datos, confirmando que la lectura de información era precisa y sincronizada con los datos reales del sistema.



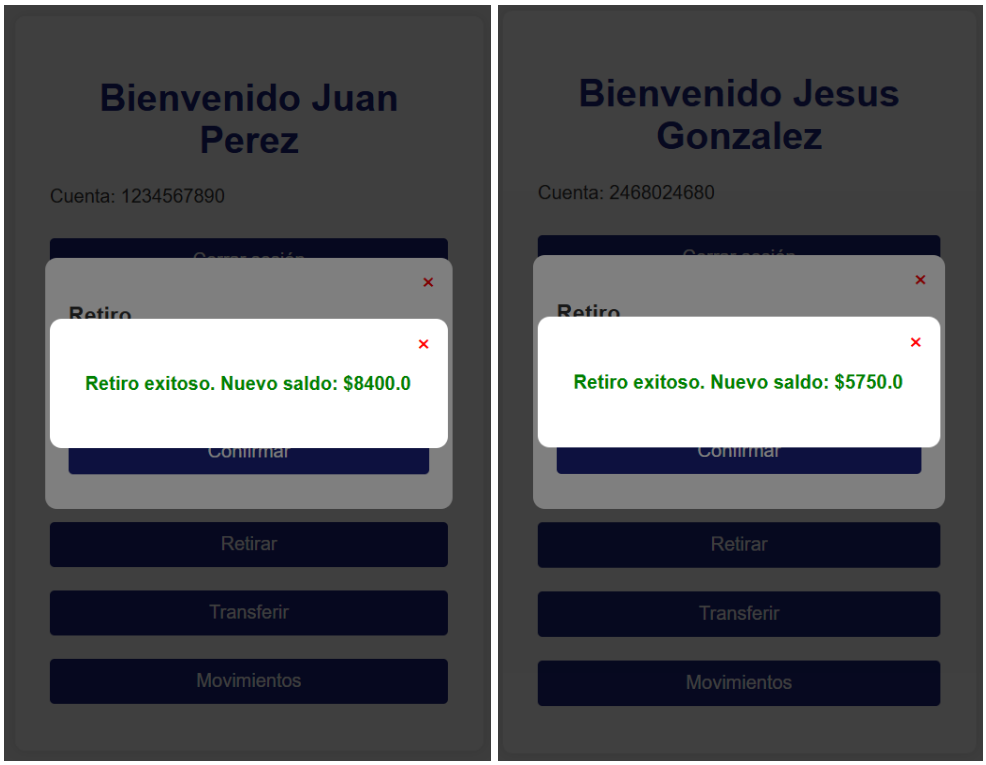
DEPOSITO DE DINERO A LAS CUENTAS

Ambos usuarios realizaron un depósito en sus respectivas cuentas. Se ingresó el monto deseado desde la interfaz web, lo cual generó una solicitud POST hacia el TransactionService. Este servicio consultó el saldo actual, realizó la suma correspondiente y actualizó el nuevo saldo en la base de datos a través de AccountService. Posteriormente, se registró la operación en la tabla de movimientos. La respuesta visual fue inmediata y los nuevos saldos se reflejaron correctamente.



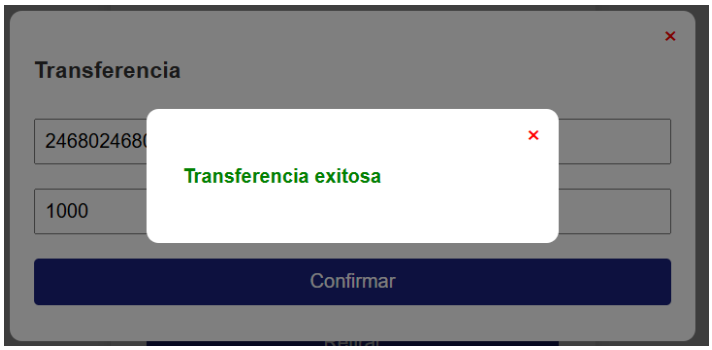
RETIRO DE DINERO DE LAS CUENTAS

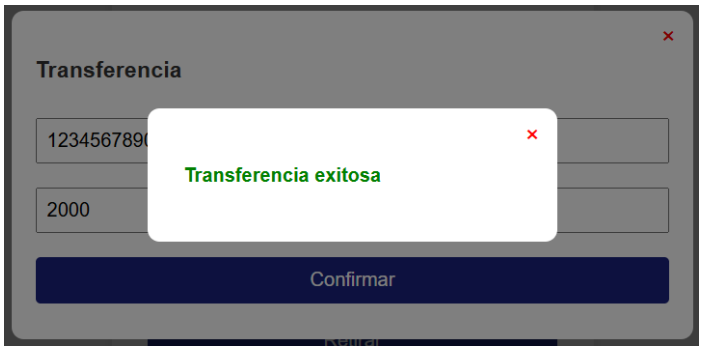
Se probó el retiro de fondos, contemplando dos escenarios: uno con monto válido y otro con monto superior al saldo disponible. En el segundo caso, el sistema respondió adecuadamente con un mensaje de error. Para los retiros válidos, el sistema empleó el microservicio ConcurrencyService para solicitar el acceso exclusivo a la cuenta, evitando conflictos por acceso simultáneo. Una vez concluida la operación, el sistema liberó el recurso y actualizó correctamente el nuevo saldo.



TRANFERENCIA DE DINERO ENTRE CLIENTES

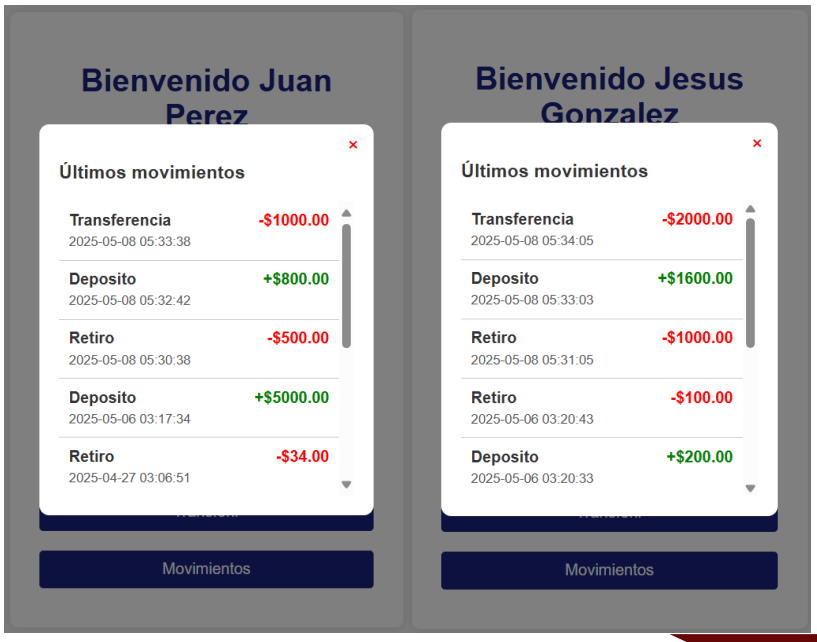
La transferencia de fondos entre los dos clientes también fue ejecutada exitosamente. Se solicitó acceso a ambas cuentas involucradas (origen y destino) mediante el servicio de concurrencia, se verificaron los saldos, se actualizó la base de datos en ambas cuentas y se registró la transferencia en la tabla correspondiente. Durante todo el proceso, se mantuvo el bloqueo temporal de las cuentas para evitar inconsistencias, liberándose al concluir la operación. Las pruebas confirmaron que no hubo errores de concurrencia y los saldos finales coincidían con los movimientos realizados.





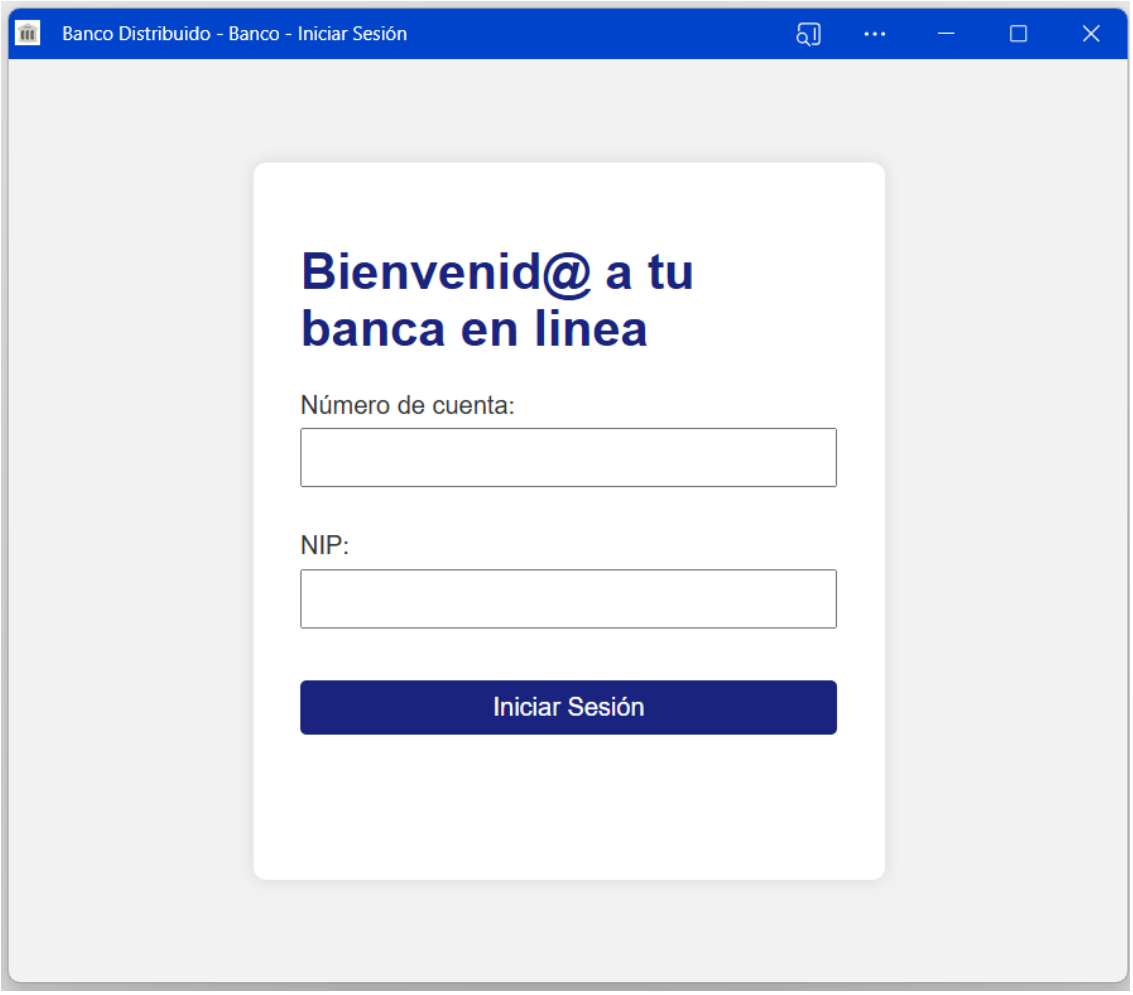
CONSULTA DE MOVIMIENTOS REALIZADOS

Se ejecutó la funcionalidad de consulta de movimientos para los dos clientes, permitiendo verificar el historial individual de transacciones. Al activarse esta opción desde la interfaz, se envía una solicitud al microservicio MovementsService, encargado de acceder a la base de datos y recuperar los diez movimientos más recientes de la cuenta correspondiente.

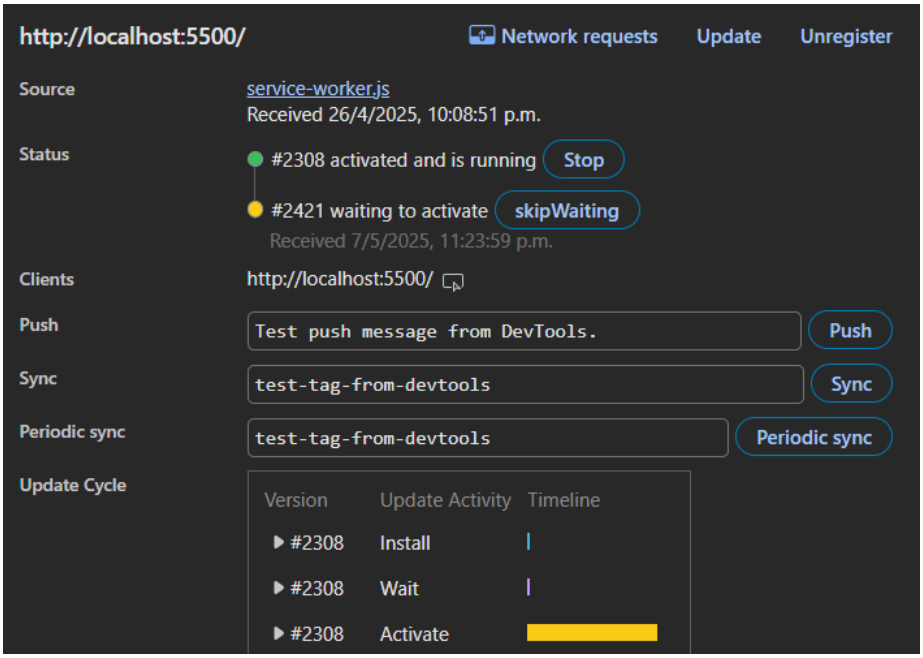


DESCARGA DE APLICACIÓN

Uno de los clientes también pudo descargar el sistema como aplicación, gracias a su implementación como PWA, lo que permite su uso directamente desde el dispositivo sin necesidad de conexión constante. Aquí como ya se descargó, no nos proporciona la opción "¿Quieres instalar app? Haz clic aquí"

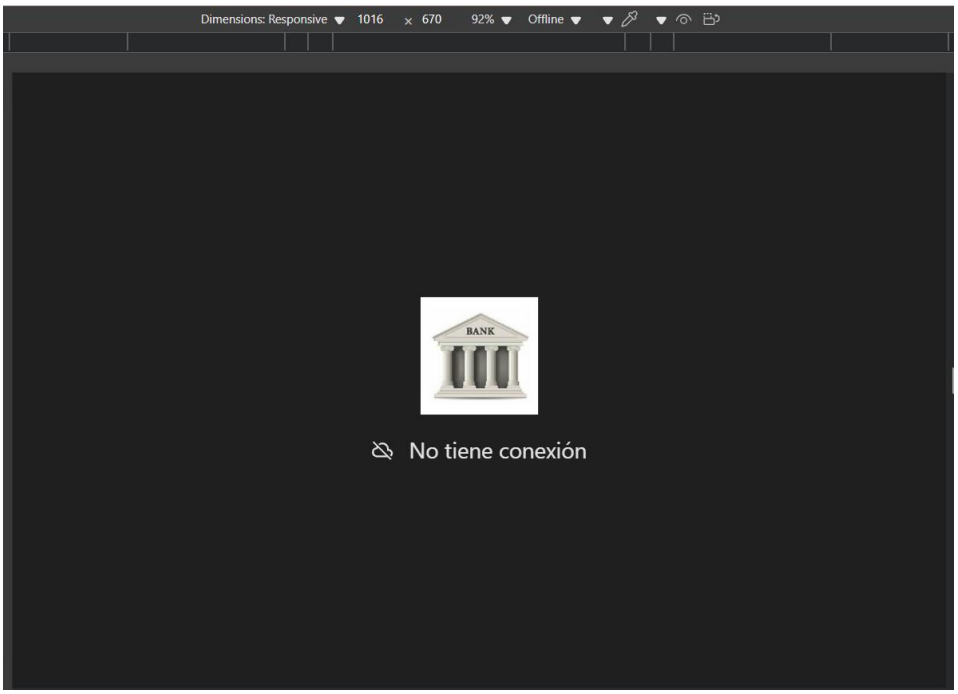


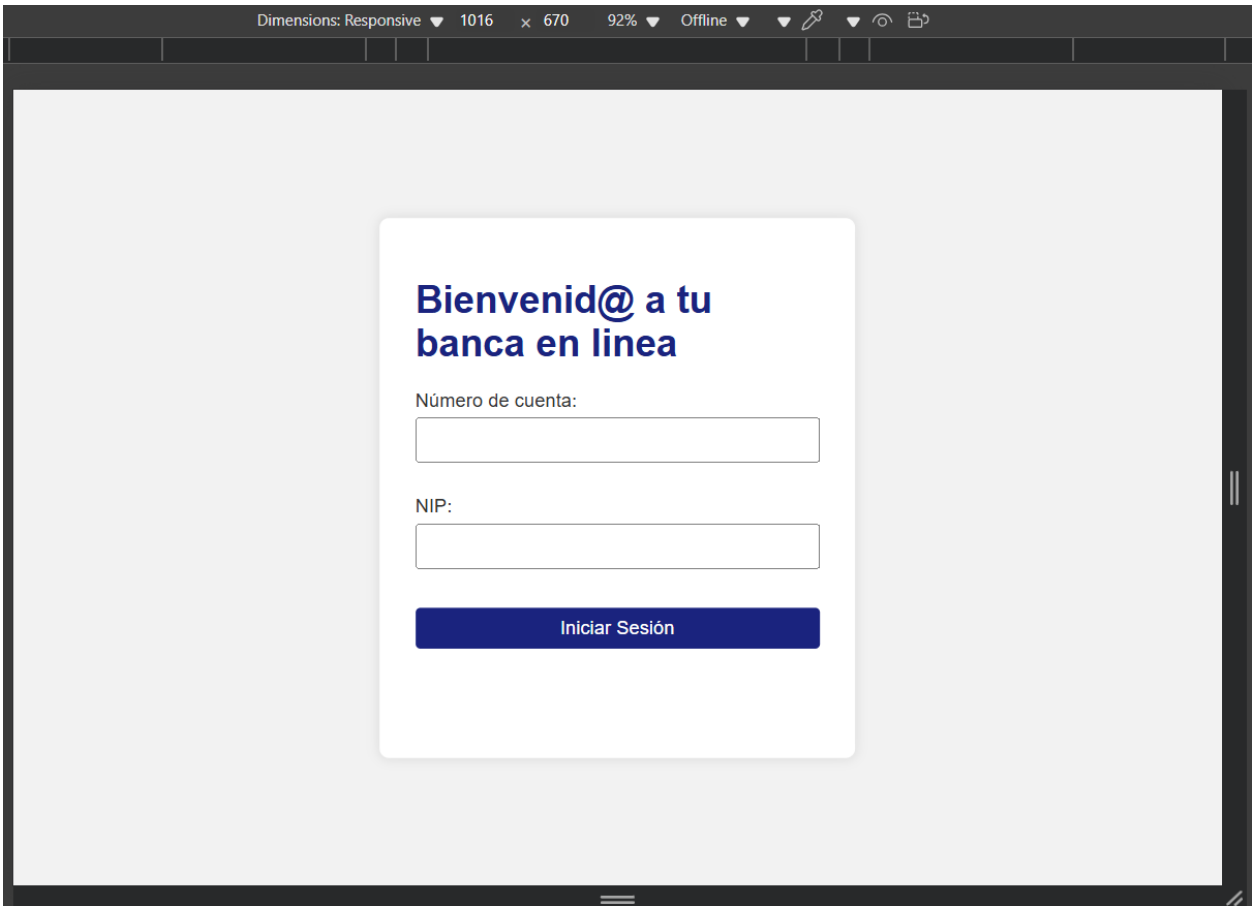
CARGA DE SERVICE WORKER



VERIFICACION DE USO DE MEMORIA CACHE

Primeramente marco que no había conexión, en lo que hacia uso de los recursos almacenados en memoria cache, obteniendo la interfaz aun en modo offline




Dimensions: Responsive 1016 x 670 92% Offline
Bienvenid@ a tu banca en linea
Número de cuenta:
NIP:
Iniciar Sesión

CONCLUSIONES

Durante esta práctica, enfrenté diversos retos que me permitieron aprender y afianzar conocimientos clave en el desarrollo de sistemas distribuidos modernos. Al comenzar con un modelo cliente-servidor tradicional y avanzar hacia una arquitectura basada en microservicios, fui capaz de entender cómo se puede modularizar una aplicación para mejorar su escalabilidad, mantenimiento y reutilización. La separación de responsabilidades en distintos servicios, como autenticación, manejo de saldo, operaciones bancarias y concurrencia, me permitió estructurar un sistema más organizado y resiliente, cuya evolución sería difícil de alcanzar en un enfoque monolítico. Sin embargo, también identifiqué sus limitaciones, especialmente al depender de una conexión constante para funcionar correctamente, lo cual se volvió evidente durante la simulación de fallas de red.

La integración de la interfaz con PWA representó un gran avance frente a las versiones anteriores del sistema. A diferencia del enfoque basado únicamente en servicios web o en objetos distribuidos, que exigen disponibilidad continua del backend, con PWA logré ofrecer una experiencia más fluida y confiable al usuario. Gracias al uso de Service Workers, pude mantener la disponibilidad de la interfaz incluso cuando el servidor estaba inaccesible o el navegador no contaba con conexión a internet, lo que representa un beneficio enorme en términos de usabilidad y accesibilidad.

Me enfrenté a desafíos técnicos importantes, como la gestión del almacenamiento en caché, la actualización de recursos en el tiempo, el registro correcto del Service Worker y su integración con múltiples archivos HTML, CSS y JS que componen el frontend. También tuve que resolver problemas relacionados con la sincronización de datos, como liberar correctamente el acceso a cuentas bloqueadas en transacciones o asegurar la consistencia de las operaciones al usar APIs distribuidas. Cada obstáculo me permitió comprender más a fondo cómo funciona la web moderna, y cómo combinar tecnologías del lado del cliente y del servidor para lograr un ecosistema funcional y eficiente.

Al finalizar, no solo consolidé habilidades en el manejo de peticiones HTTP, concurrencia, consultas a bases de datos y separación de responsabilidades, sino que también entendí por qué una aplicación distribuida debe evolucionar hacia soluciones que prioricen la experiencia del usuario. La implementación de PWA no solo mejora la disponibilidad y velocidad, sino que acerca las aplicaciones web a la experiencia de las apps móviles, sin los inconvenientes de instalación ni dependencia de tiendas. Esta evolución representa un paso natural y necesario en la creación de soluciones accesibles, robustas y modernas. Me llevo de esta práctica una comprensión más integral de los desafíos reales del desarrollo distribuido y la convicción de que la experiencia del usuario debe guiar las decisiones tecnológicas desde las etapas iniciales de diseño.