INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

UNIDAD DE APRENDIZAJE

SISTEMAS DISTRIBUIDOS

PRACTICA No. 7

MICROSERVICIOS

ALUMNO

GÓMEZ GALVAN DIEGO YAEL

GRUPO

7CM1

PROFESOR

CARRETO ARELLANO CHADWICK

FECHA DE ENTREGA

16 DE ABRIL DE 2025

INDICE

ANTESCEDENTES

Los microservicios representan un estilo arquitectónico que propone la construcción de aplicaciones como un conjunto de servicios pequeños, independientes y desplegables de forma autónoma. Cada microservicio se encarga de una funcionalidad específica del negocio y se comunica con otros servicios mediante interfaces bien definidas, generalmente a través del protocolo HTTP y utilizando formatos de datos como JSON o XML.

Esta arquitectura surge como una evolución del enfoque monolítico tradicional, en el cual toda la lógica de la aplicación, incluyendo la interfaz de usuario, la lógica de negocio y el acceso a datos, se encuentra integrada en una única unidad desplegable. A diferencia de este modelo, los microservicios permiten que los componentes del sistema funcionen de manera desacoplada, lo que brinda una mayor flexibilidad en términos de desarrollo, escalabilidad y mantenimiento.

Características de los microservicios

Una de las principales características de los microservicios es la descomposición funcional, donde cada servicio aborda una función única dentro del dominio del negocio. Esta división permite que cada microservicio sea desarrollado, probado, desplegado y mantenido de manera independiente por equipos distintos.

Cada microservicio puede estar desarrollado con diferentes tecnologías, lo que permite elegir las herramientas más adecuadas según las necesidades específicas de cada componente. Asimismo, los microservicios ofrecen la capacidad de escalar de forma granular; es decir, se pueden escalar únicamente aquellos servicios que presentan una alta demanda sin necesidad de replicar la totalidad del sistema.

Otro aspecto importante es la tolerancia a fallos. Al estar desacoplados, el fallo de un microservicio no necesariamente compromete la disponibilidad del resto del sistema. La comunicación entre microservicios se da a través de APIs ligeras, muchas veces utilizando interfaces RESTful, lo que garantiza una interacción sencilla y eficiente. Asimismo, los microservicios promueven la reutilización y la modularidad, facilitando la evolución del sistema y la incorporación de nuevas funcionalidades.

Ventajas de la arquitectura de microservicios

La adopción de microservicios proporciona diversas ventajas. Una de las más relevantes es la posibilidad de desarrollar diferentes partes del sistema en paralelo, lo cual permite una mayor velocidad de entrega y una mejora significativa en la productividad de los equipos. Gracias a su naturaleza modular, los microservicios también permiten realizar despliegues independientes, lo que facilita la implementación de actualizaciones o correcciones sin necesidad de detener todo el sistema.

La escalabilidad específica es otra ventaja destacada. En lugar de escalar toda la aplicación, es posible escalar solamente aquellos servicios que requieren mayor capacidad, lo que optimiza el uso de los recursos. En términos de mantenimiento, los microservicios simplifican la gestión del código, permitiendo aislar errores o actualizar funcionalidades sin afectar el resto del sistema. Además, la arquitectura promueve la resiliencia, dado que los fallos se aíslan y pueden manejarse de forma controlada. La flexibilidad tecnológica que brindan, al permitir el uso de múltiples lenguajes y herramientas, también es muy apreciada en proyectos complejos.

Desventajas de los microservicios

A pesar de sus ventajas, la arquitectura de microservicios también presenta una serie de desafíos. Uno de los más significativos es el aumento en la complejidad operativa. Al tratarse de múltiples servicios desplegados de forma independiente, se requiere una infraestructura más robusta para orquestarlos, monitorearlos y mantenerlos.

Otro reto es la comunicación entre servicios, que puede generar latencias y aumentar los puntos de falla en el sistema, especialmente cuando la red es inestable. La gestión de datos también se vuelve más complicada, ya que los microservicios suelen tener bases de datos separadas, lo que dificulta mantener la consistencia de la información. Las pruebas también se complejizan, ya que además de las pruebas unitarias individuales, es necesario realizar pruebas de integración entre servicios. Además, para implementar correctamente esta arquitectura se requiere personal con mayor conocimiento técnico, así como herramientas especializadas para el despliegue, monitoreo, balanceo de carga y manejo de errores distribuidos.

Comparación con arquitecturas tradicionales

Mientras que la arquitectura monolítica resulta adecuada para aplicaciones pequeñas o con requisitos simples, los microservicios son particularmente beneficiosos en sistemas grandes, con múltiples módulos y necesidades de escalabilidad. A diferencia del enfoque monolítico, donde cualquier cambio implica revisar y volver a desplegar toda la aplicación, los microservicios permiten una evolución gradual y controlada del sistema.

Sin embargo, esta evolución no es trivial. Migrar de una arquitectura monolítica a microservicios implica un rediseño profundo de la lógica de negocio, la estructura de datos y los mecanismos de comunicación interna. Además, requiere adoptar buenas prácticas de desarrollo distribuido, automatización de pruebas y despliegues, así como una cultura organizacional alineada con los principios de modularidad y colaboración continua.

Hoy en día, la arquitectura de microservicios es ampliamente utilizada por organizaciones que necesitan escalar sus sistemas y mantener alta disponibilidad. Empresas como Netflix, Amazon, Spotify, Uber y Google han adoptado este enfoque para permitir el desarrollo rápido, el escalado eficiente y la tolerancia a fallos en sus sistemas. Su aplicación es común en sectores como

comercio electrónico, banca, telecomunicaciones, educación en línea y plataformas de contenido multimedia.

La arquitectura de microservicios también es compatible con otros enfoques modernos como DevOps, integración continua, contenedores y orquestadores como Docker y Kubernetes, lo que refuerza su posición como una de las arquitecturas predominantes en sistemas distribuidos de nueva generación.

PLANTEAMIENTO DEL PROBLEMA

En los últimos años, la arquitectura basada en servicios web ha sido una estrategia ampliamente adoptada para facilitar la comunicación entre módulos de software a través de interfaces bien definidas. Sin embargo, a medida que las aplicaciones se vuelven más complejas, distribuidas y orientadas al crecimiento continuo, el enfoque tradicional de servicios web comienza a mostrar limitaciones significativas.

Una de las principales problemáticas radica en el fuerte acoplamiento entre componentes y la dependencia centralizada en servicios únicos o compartidos. Este diseño provoca dificultades al momento de escalar el sistema, aplicar cambios parciales sin afectar a otros módulos o garantizar una alta disponibilidad ante fallos. En sistemas críticos como los bancarios, donde se requiere una alta confiabilidad, concurrencia de múltiples usuarios, control transaccional y tolerancia a fallos, estas limitaciones se vuelven especialmente problemáticas.

El desarrollo de un sistema bancario que permita operaciones fundamentales como autenticación, consulta de saldo, depósitos, retiros y transferencias, mediante una estructura basada únicamente en servicios web, revela cuellos de botella en términos de mantenimiento, escalabilidad individual por módulo y evolución independiente de funcionalidades. Por ejemplo, un fallo en el servicio de autenticación o de saldo podría comprometer la disponibilidad completa del sistema, impidiendo que los usuarios accedan a funciones que no dependen directamente de dichos servicios.

Adicionalmente, los servicios web tradicionales no ofrecen una solución nativa para el control fino de la concurrencia entre operaciones críticas, ni una separación clara entre responsabilidades que permita desplegar o actualizar módulos de forma autónoma sin interrumpir el sistema completo.

Frente a este contexto, se plantea la necesidad de reevaluar el enfoque arquitectónico y explorar alternativas que brinden mayor flexibilidad, resiliencia y escalabilidad al sistema bancario. Es aquí donde se abre paso la arquitectura de microservicios como una evolución lógica y necesaria para solventar las deficiencias del modelo anterior, permitiendo un desarrollo más modular, dinámico y adaptado a los desafíos actuales del software distribuido.

PROPUESTA DE SOLUCIÓN

Ante las limitaciones que presenta el modelo tradicional de servicios web, se propone como solución el uso de una arquitectura basada en microservicios para el desarrollo del sistema bancario. Esta propuesta busca mejorar la modularidad, escalabilidad y mantenibilidad del sistema mediante la separación de sus funcionalidades principales en componentes independientes.

Cada una de las funciones del sistema, como la autenticación, la gestión de cuentas, el procesamiento de transacciones y el control de concurrencia, será desarrollada como un microservicio autónomo, comunicándose a través de interfaces REST. Esta estructura permitirá una evolución más flexible del sistema, una mejor respuesta ante fallos y una mayor facilidad para aplicar cambios o añadir nuevas funcionalidades en el futuro.

El enfoque de microservicios será la base para una solución robusta, adaptable y alineada con las prácticas actuales en el desarrollo de sistemas distribuidos.

MATERIALES Y METODOS

Para la implementación del simulador, se utilizaron los siguientes recursos:

- ✓ **Sistema operativo:** Windows 11, plataforma sobre la cual se desarrolló y ejecutó el sistema.
- ✓ **IDE de desarrollo:** NetBeans 21, entorno de desarrollo integrado que permitió gestionar los proyectos, compilar el código, configurar Maven y ejecutar cada microservicio de forma independiente.
- ✓ **Lenguaje de programación:** Java SE, versión 22, utilizado para implementar la lógica del sistema y desarrollar los microservicios REST.
- ✓ **Base de datos:** SQLite 3, sistema de gestión de base de datos embebido que almacenó la información de clientes, cuentas, movimientos y transferencias en un archivo persistente (Banco.db).
- ✓ **Servidor embebido:** HttpServer de com.sun.net.httpserver, usado para exponer endpoints HTTP sin necesidad de frameworks externos.
- ✓ **Serialización JSON:** Biblioteca Gson, empleada para transformar objetos Java a JSON y viceversa, facilitando la comunicación entre servicios y con la interfaz web.
- ✓ **Interfaz web:** HTML5, CSS3 y JavaScript, tecnologías utilizadas para diseñar una interfaz sencilla pero funcional que permite al usuario autenticarse y realizar operaciones.
- ✓ **Cliente HTTP para pruebas:** Postman, herramienta que permitió realizar pruebas directas a los endpoints HTTP de los microservicios.
- ✓ **Navegador web:** Google Chrome, utilizado para ejecutar la interfaz gráfica y probar el flujo completo de usuario.
- ✓ **Gestión de concurrencia:** Se diseñó un microservicio adicional responsable de controlar los accesos simultáneos a cuentas, evitando conflictos durante operaciones críticas como retiros y transferencias.

DESARROLLO DE SOLUCIÓN

La solución implementada consistió en el desarrollo de un sistema bancario modular distribuido, organizado bajo una arquitectura de microservicios. Cada microservicio fue implementado de manera independiente, con responsabilidades claramente definidas, y todos ellos se comunican entre sí mediante el protocolo HTTP y mensajes JSON.

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

LoginService

Este archivo implementa el microservicio de autenticación. Su función principal es validar que el número de cuenta y el NIP enviados por el usuario correspondan a un registro válido en la base de datos.

```
public class LoginService {
    public static void main(String[] args) {
        try {
            // Crear el servidor en puerto 8081
            HttpServer server = HttpServer.create(new InetSocketAddress(8081), 0);

            // Asociar la ruta /login con el manejador
            server.createContext("/login", new LoginHandler());

            // Empezar el servidor
            server.setExecutor(null); // Usa el executor por defecto
            server.start();

            System.out.println("Servidor de autenticación 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); // Sin contenido
                    return;
                }
                if (!"POST".equalsIgnoreCase(exchange.getRequestMethod())) {
                    exchange.sendResponseHeaders(405, -1); // Método no permitido
                    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 respuesta = valido ? "Autenticacion Exitosa" : "Error en autenticación";
                int codigo = valido ? 200 : 401;
                exchange.sendResponseHeaders(codigo, respuesta.getBytes().length);
                OutputStream os = exchange.getResponseBody();
                os.write(respuesta.getBytes());
                os.close();
            }
```

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

    public Login() {}

    // Getters y Setters
    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;
    }
}
```

El método handle() recibe una solicitud HTTP de tipo POST con los datos del usuario en formato JSON, los convierte a un objeto Login, y llama a BaseDatos.validarCuenta(...). Si la autenticación es exitosa, responde con código 200; si falla, responde con 401.

SaldoHandler

Este archivo implementa la consulta y actualización del saldo de las cuentas. Se divide en dos tipos de solicitudes: consulta (GET) y actualización (POST).

```java
private void manejarConsultaSaldo(HttpExchange exchange) throws IOException {
    URI uri = exchange.getRequestURI();
    String query = uri.getQuery(); // Ej: "cuenta=1234567890"

    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);
}
```

```
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);
}
```

La clase recibe las solicitudes, interpreta los parámetros o el JSON recibido y utiliza BaseDatos para ejecutar las consultas.

OperacionHandler

Gestiona las operaciones bancarias: depósito, retiro y transferencia. Define una lógica para cada tipo de operación, asegurando primero el acceso exclusivo a la cuenta mediante el servicio de concurrencia.

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

```
    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");
        }
    }
```

```
private void procesarTransferencia(HttpExchange exchange, TransferenciaDTO dto) throws IOException {
    boolean origenOk = false;
    boolean destinoOk = false;
    try {
        if (!solicitarAcceso(dto.getCuentaOrigen())) {
            enviarRespuesta(exchange, 423, "Cuenta origen en uso. Intente más tarde.");
            return;
        }
        origenOk = true;
        if (!solicitarAcceso(dto.getCuentaDestino())) {
            enviarRespuesta(exchange, 423, "Cuenta destino en uso. Intente más tarde.");
            return;
        }
        destinoOk = true;
        double saldoOrigen = consultarSaldo(dto.getCuentaOrigen());
        double saldoDestino = consultarSaldo(dto.getCuentaDestino());
        if (saldoOrigen < 0 || saldoDestino < 0) {
            enviarRespuesta(exchange, 404, "Cuenta origen o destino no encontrada");
            return;
        }
        if (dto.getMonto() <= 0 || dto.getMonto() > saldoOrigen) {
            enviarRespuesta(exchange, 400, "Saldo insuficiente o monto inválido");
            return;
        }
        double nuevoOrigen = saldoOrigen - dto.getMonto();
        double nuevoDestino = saldoDestino + dto.getMonto();
        if (actualizarSaldo(dto.getCuentaOrigen(), nuevoOrigen) &&
            actualizarSaldo(dto.getCuentaDestino(), nuevoDestino)) {

            BaseDatos.registrarMovimiento(dto.getCuentaOrigen(), "transferencia", dto.getMonto());
            BaseDatos.registrarTransferencia(dto.getCuentaOrigen(), dto.getCuentaDestino(), dto.getMonto());

            enviarRespuesta(exchange, 200, "Transferencia exitosa");
        } else {
            enviarRespuesta(exchange, 500, "Error al actualizar los saldos");
        }
    } finally {
        if (destinoOk) liberarAcceso(dto.getCuentaDestino());
        if (origenOk) liberarAcceso(dto.getCuentaOrigen());
    }
}
```

El uso de try/finally garantiza que las cuentas sean liberadas correctamente, incluso en caso de error.

BaseDatos

Contiene todos los métodos de acceso a la base de datos SQLite. Se encarga de validar credenciales, consultar saldos, actualizarlos, y registrar operaciones.

```
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(); // true si encuentra una coincidencia
    } catch (SQLException e) {
        System.out.println("Error en la base de datos: " + e.getMessage());
        return false;
    }
}

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;
    }
}

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;
    }
}
```

Todos los métodos utilizan PreparedStatement para ejecutar consultas SQL seguras sobre la base de datos Banco.db.

ConcurrencyService

Controla el acceso exclusivo a las cuentas bancarias. Este microservicio evita que dos operaciones simultáneas se realicen sobre la misma cuenta, lo que asegura consistencia de datos.

```
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();
}
```

Index

Permite iniciar sesión ingresando el número de cuenta y el NIP. Si las credenciales son válidas, redirige al dashboard.

```html
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Banco - Iniciar Sesión</title>
  <link rel="stylesheet" href="../css/estilo.css">

</head>
<body>
  <h1>Bienvenido al Banco</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="mensaje" style="color: red;"></p>

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

Dashboard

Ofrece las funcionalidades principales (consultar saldo, depositar, retirar, transferir). Cada operación se realiza a través de modales (ventanas emergentes).

```html
  <h1>Bienvenido</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>

  <!-- Ventanas modales -->
  <div class="modal" id="modal">
    <div class="modal-content" id="modalContenido">
      <span class="close" onclick="cerrarModal()">&times;</span>
      <div id="contenidoModal"></div>
    </div>
  </div>

  <!-- Ventana de mensaje -->
  <div class="modal" id="modalMensaje">
    <div class="modal-content">
      <span class="close" onclick="cerrarMensaje()">&times;</span>
      <p id="mensaje"></p>
    </div>
  </div>

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

Banco

Maneja toda la lógica del lado cliente. Se conecta con los microservicios usando fetch().

```
function abrirModal(tipo) {
  let html = "";

  if (tipo === "saldo") {
    consultarSaldo();
    return;
  }

  if (tipo === "deposito") {
    html = `
      <h3>Depósito</h3>
      <input type="number" id="montoDeposito" placeholder="Monto a depositar">
      <button onclick="realizarOperacion('deposito')">Confirmar</button>
    `;
  } else if (tipo === "retiro") {
    html = `
      <h3>Retiro</h3>
      <input type="number" id="montoRetiro" placeholder="Monto a retirar">
      <button onclick="realizarOperacion('retiro')">Confirmar</button>
    `;
  } else if (tipo === "transferencia") {
    html = `
      <h3>Transferencia</h3>
      <input type="text" id="cuentaDestino" placeholder="Cuenta destino">
      <input type="number" id="montoTransferencia" placeholder="Monto a transferir">
      <button onclick="realizarOperacion('transferencia')">Confirmar</button>
    `;
  }

  contenidoModal.innerHTML = html;
  modal.style.display = "flex";
}
```

```
// Cerrar modal
function cerrarModal() {
  modal.style.display = "none";
}

// Mostrar mensaje modal
function mostrarMensaje(texto, color = "green") {
  mensaje.style.color = color;
  mensaje.textContent = texto;
  modalMensaje.style.display = "flex";

  setTimeout(() => {
    cerrarModal();
    cerrarMensaje();
  }, 3000);
}

// Cerrar mensaje modal
function cerrarMensaje() {
  modalMensaje.style.display = "none";
}
```

```
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"));
}
```

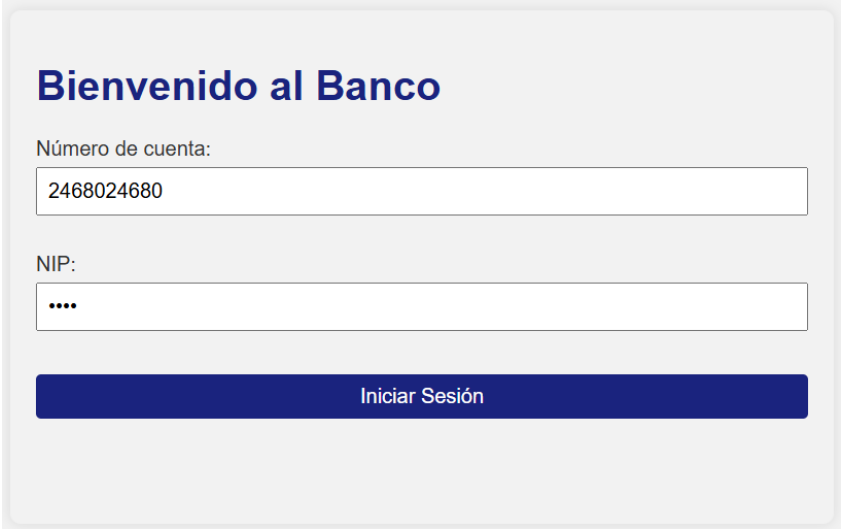
```
function consultarSaldo() {
  fetch(`http://localhost:8082/saldo?cuenta=${cuenta}`)
    .then(res => res.text())
    .then(saldo => {
      contenidoModal.innerHTML = `
        <h3>Saldo disponible</h3>
        <p style="font-size: 24px; text-align: center;"><strong>$${saldo}</strong></p>
      `;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

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.

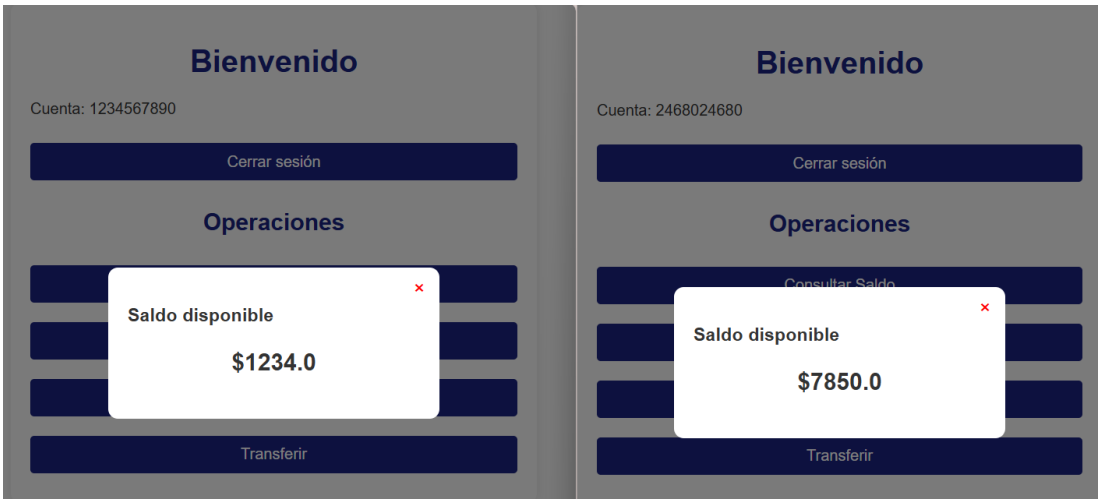
Autenticación de cliente A



Autenticación de cliente B

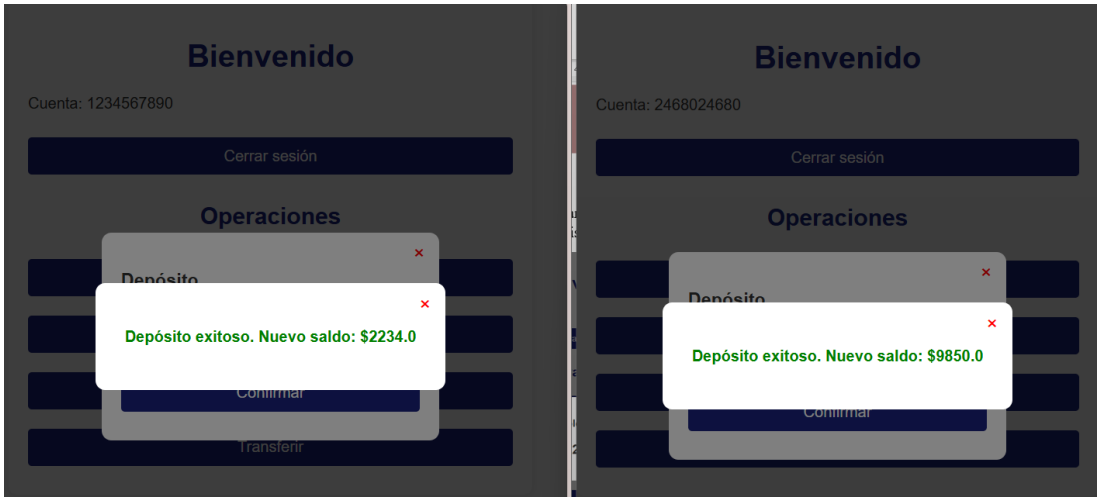
CONSULTA DE SALDO INICIAL

Una vez autenticados, 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.



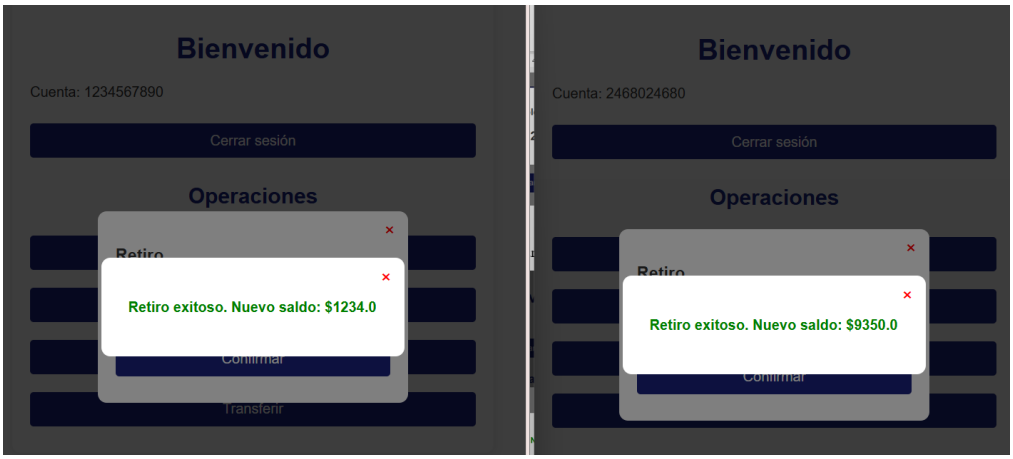
DEPOSITO DE DINERO A LAS CUENTAS

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.



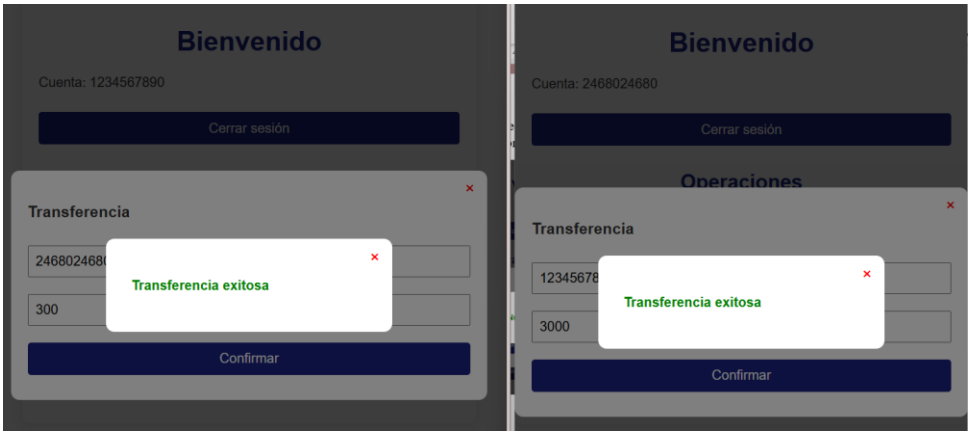
RETIRO DE DINERO DE LAS CUENTAS

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.



CONCLUSIONES

El desarrollo de esta práctica me permitió adquirir una comprensión más profunda sobre la arquitectura de microservicios, particularmente en lo que respecta a la organización modular de aplicaciones, el desacoplamiento de responsabilidades y la implementación de servicios independientes que se comunican entre sí mediante interfaces bien definidas. A lo largo del proceso integré conocimientos clave sobre programación orientada a objetos, diseño RESTful, manejo de bases de datos, control de concurrencia y comunicación entre servicios a través del protocolo HTTP, lo cual fortaleció tanto mi base teórica como mi habilidad práctica en el desarrollo de sistemas distribuidos.

Durante la implementación, enfrenté diversos retos. Uno de los más significativos fue entender y aplicar el modelo de microservicios sin depender de frameworks como Spring Boot. Esto me llevó a construir desde cero la lógica de enrutamiento de peticiones, el manejo de JSON y la configuración de servidores embebidos en cada servicio. Aunque este enfoque incrementó la complejidad inicial, también me permitió conocer en detalle cómo funcionan internamente los distintos componentes del sistema y tener un control total sobre su comportamiento.

Otra problemática importante surgió en el manejo de la concurrencia, especialmente durante operaciones críticas como retiros y transferencias. Al inicio, al no contar con un mecanismo de sincronización centralizado, se presentaron inconsistencias en los saldos por condiciones de carrera. Para solucionarlo, implementé un microservicio de concurrencia que controla el acceso exclusivo a las cuentas, mediante un sistema de bloqueo y liberación. Utilicé estructuras en memoria y bloques try/finally para asegurar que las cuentas se liberaran correctamente incluso en caso de errores.

Durante la creación de la interfaz web también surgieron obstáculos, especialmente en la comunicación con los microservicios. Se presentaron errores relacionados con CORS y diferencias entre los datos enviados desde JavaScript y los que se esperaban en los servicios. Estos problemas los resolví configurando adecuadamente los encabezados HTTP y estandarizando los campos en los objetos JSON, lo que mejoró la interoperabilidad entre cliente y servidor.

En general, esta práctica me permitió consolidar buenas prácticas de diseño modular, separar adecuadamente las capas del sistema (presentación, lógica y datos), y aplicar principios de reusabilidad y mantenimiento del código.