INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

UNIDAD DE APRENDIZAJE

SISTEMAS DISTRIBUIDOS

PRACTICA No. 6

SERVICIOS WEB

ALUMNO

GÓMEZ GALVAN DIEGO YAEL

GRUPO

7CM1

PROFESOR

CARRETO ARELLANO CHADWICK

FECHA DE ENTREGA

02 DE ABRIL DE 2025

INDICE

ANTESCEDENTES

En el desarrollo de aplicaciones distribuidas modernas, la necesidad de establecer una comunicación efectiva entre diferentes sistemas, plataformas y lenguajes de programación ha llevado a la evolución de múltiples paradigmas de interoperabilidad. En este contexto, los servicios web surgieron como una solución clave para lograr dicha interoperabilidad, permitiendo que diversas aplicaciones puedan interactuar a través de una red, principalmente internet, independientemente de su estructura interna.

Un servicio web es una unidad funcional diseñada para ser accedida de forma remota mediante protocolos estándar, principalmente el protocolo HTTP, y utilizando formatos de intercambio de datos como XML o JSON. Estos servicios exponen una serie de funciones o métodos que pueden ser invocados por cualquier cliente autorizado, sin necesidad de conocer la lógica interna del servidor, cumpliendo así con los principios de encapsulamiento y reutilización de software.

Origen y evolución de los servicios web

El concepto de servicios web se consolido a finales de los años noventa y principios del nuevo milenio, en respuesta a las necesidades de conectar aplicaciones empresariales heterogéneas. En sus inicios, los servicios web estaban fuertemente ligados a tecnologías como SOAP (Simple Object Access Protocol), WSDL (Web Services Description Languaje) y UDDI (Universal Description, Discovery and Integration). Estas tecnologías permitían describir, publicar y consumir servicios mediante un enfoque formal y estandarizado.

Sin embargo, el enfoque basado en SOAP, aunque robusto y extensible, también resultó complejo y costoso de implementar, lo que propocio el surmiento de alternativas mas ligeras. Asi nació el modelo de servicios web RESTful (Representational State Transfer), promovido por Roy Fielding, el cual propone un estilo arquitectónico mas sencillo, basado en los principios de la web y el uso de los métodos estándar del protocolo HTTP (GET, POST, PUT, DELETE, entre otros)

REST se consolido rápidamente como el paradigma dominante para la creación de servicios web modernos, debido a su simplicidad, flexibilidad y facilidad de integración con tecnologías emergentes como JavaScript, frameworks frontend (React, Angular, Vue), y entornos móviles.

Características de los servicios web RESTful

Los servicios web RESTful presentan una serie de características fundamentales que los diferencian de otros modelos de integración:

- ✓ Interoperabilidad: Permiten que las aplicaciones escritas en distintos lenguajes de programación puedan comunicarse entre si de forma transparente
- ✓ Escalabilidad: Al estar basados en HTTP, pueden escalar fácilmente en entornos web de alto trafico

- ✓ Orientación a recursos: Cada elemento del sistema se modela como un recurso con una URI única
- ✓ Sin estado: Cada petición HTTP es independiente y no mantiene información del estado del cliente entre solicitudes.
- ✓ Uso de formatos estándares: Como JSON o XML para el intercambio de información, facilitando la serialización de objetos
- ✓ Facilidad de prueba e integración: Herramientas como Postman o Swagger permiten probar y documentar los servicios fácilmente

Ventajas de los servicios web

La implementación de servicios web representa una serie de beneficios relevantes para los desarrolladores, las empresas y los usuarios finales. Entre las ventajas más destacadas se encuentran:

- ✓ Modularidad: Cada servicio puede desarrollarse, desplegarse y mantenerse de forma independiente
- ✓ Reutilización: Los servicios pueden ser consumidos por múltiples aplicaciones sin duplicar código
- ✓ Integración de sistemas: Permite la comunicación entre sistemas antiguos y modernos
- ✓ Independencia de plataforma: Los servicios pueden ser consumidos desde cualquier entorno
- ✓ Adaptabilidad: Se adaptan fácilmente a aplicaciones móviles, web y microservicios

Desventajas y limitaciones

A pesar de sus múltiples beneficios, los servicios web no están exentos de desafíos. Algunas de las principales desventajas incluyen:

- ✓ Complejidad en la seguridad: Especialmente cuando se requiere autenticación, autorización y encriptación de datos
- ✓ Latencia de red: Al depender de conexiones HTTP, las llamadas pueden verse afecyadas por la calidad de la red
- ✓ Falta de control sobre el cliente: El proveedor del servicio no puede asegurar como se utilizará su servicio, lo que puede llevar a errores de implementación por parte del consumidor
- ✓ Dificultad en el manejo de versiones: Cuando múltiples clientes dependen de un servicio, actualizar sus funcionalidades requieren una estrategia cuidadosa

Servicios web en entornos distribuidos

En los entornos distribuidos, donde múltiples procesos o aplicaciones debe interactuar en tiempo real o de manera asíncrona, los servicios web RESTful se convierten en una herramienta esencial. Estos permiten desacoplar los componentes del sistema, facilitar el mantenimiento y

promover la escalabilidad horizontal. Además, su naturaleza sin estado facilita su uso en infraestructuras basadas en contenedores, balanceadores de carga y servicios en la nube

Servicios web y su implementación en Java con Spring Boot

El desarrollo de servicios web en el lenguaje Java ha evolucionado considerablemente con la aparición de framework que simplifican la creación y despliegue de aplicaciones web modernas. Uno de los mas utilizados en la actualidad es Spring Boot, una extensión del ecosistema Spring que permite crear aplicaciones empresariales y servicios RESTful de manera rápida, eficiente y con una configuración mínima.

Spring Boot ofrece una arquitectura clara basada en capas, como controladores, servicios y modelos, lo que facilita la organización del código y promueve buenas practicas de desarrollo. A través del uso de anotaciones como *@RestController*, *@GetMapping*, *@PostMapping*, *@RequestParam*, entre otras, se pueden definir endpoints HTTP de forma concisa y semántica, haciendo que la implementación de servicios sea mas accesible incluso para desarrolladores que recién comienzan en el mundo de las aplicaciones distribuidas.

Además, Spring Boot cuenta con una integración nativa como Maven o Gradle, herramientas que automatizan la gestión de dependencias, la compilación del proyecto y su ejecución. También incluye servidores embebidos como Tomcat, lo que permite ejecutar servicios sin necesidad de desplegarlos manualmente a un servidor externo.

Una característica destacada de este framework es su compatibilidad con herramientas de depuración y prueba con Postman, que permiten simular llamadas HTTP a los servicios implementados y validar su funcionamiento en diferentes escenarios. También es común el uso de herramientas como LiveReload o Spring DevTools, que mejoran la productividad del desarrollador al permitir la recarga automática de los cambios realizados en tiempo real.

Gracias a su diseño modular, escalabilidad y extensibilidad, Spring Boot se ha consolidado como una de las opciones preferidas para el desarrollo de servicios web RESTful, siendo ampliamente adoptado en proyectos académicos, empresariales y de software libre.

PLANTEAMIENTO DEL PROBLEMA

En el desarrollo de sistemas distribuidos, el modelo cliente-servidor ha sido ampliamente utilizado para establecer comunicación entre aplicaciones que requieren compartir recursos a través de una red. Sin embargo, este modelo presenta limitaciones importantes en cuanto a escalabilidad, flexibilidad e integración con plataformas heterogéneas. Como respuesta, surgió el enfoque de objetos distribuidos, el cual permite que los procesos se comuniquen mediante invocaciones remotas, encapsulando la lógica de negocio y promoviendo una arquitectura orientada a objetos.

Aunque los objetos distribuidos, como los implementados en Java RMI, ofrecen una estructura clara y potente, presentan desafíos significativos: fuerte acoplamiento entre componentes, complejidad en su configuración y despliegue, dificultades para escalar o adaptar el sistema a nuevas plataformas, y una limitada interoperabilidad con tecnologías modernas. Estos aspectos se vuelven aún más notorios en sistemas que requieren alta disponibilidad, integración con servicios web o acceso desde clientes móviles.

Frente a estas limitaciones, surge la necesidad de adoptar una arquitectura más flexible, moderna y desacoplada. En este contexto, los servicios web RESTful representan una solución viable, ya que permiten exponer funcionalidades mediante el protocolo HTTP y manejar datos en formatos ligeros como JSON, facilitando la comunicación entre diferentes tipos de aplicaciones. Este modelo promueve la modularidad, la reutilización de componentes y una integración más sencilla con tecnologías actuales.

El problema específico que se aborda es la necesidad de modernizar un sistema distribuido bancario basado en objetos remotos, migrando su arquitectura a un enfoque de servicios web RESTful, lo cual permitirá mejorar su mantenibilidad, escalabilidad y compatibilidad con entornos modernos de desarrollo.

PROPUESTA DE SOLUCIÓN

Con el objetivo de superar las limitaciones presentadas por el modelo de objetos distribuidos, se propone la implementación de un sistema bancario distribuido mediante servicios web RESTful, desarrollados con el framework Spring Boot. Esta solución busca transformar una arquitectura rígida, fuertemente acoplada y dependiente de la infraestructura RMI, en una plataforma más modular, flexible y fácilmente integrable con tecnologías modernas.

El uso de servicios web RESTful permitirá desacoplar la lógica del sistema, facilitando la creación de endpoints accesibles mediante el protocolo HTTP y estructurados en torno a operaciones comunes como autenticación, consulta de saldo, depósito, retiro y transferencia. Estos servicios serán consumidos por clientes mediante herramientas como Postman o integrables a futuro con interfaces gráficas, aplicaciones móviles o sistemas de terceros.

Spring Boot se adopta como plataforma base por su simplicidad en la configuración, su compatibilidad con prácticas modernas de desarrollo y su integración nativa con servidores embebidos como Tomcat. Esto permitirá ejecutar el sistema de forma inmediata, sin requerir configuraciones complejas ni servidores externos, favoreciendo la portabilidad y facilidad de prueba del sistema.

Asimismo, se plantea conservar una estructura de capas, separando los controladores, servicios y modelos, para mantener una arquitectura limpia y coherente. Esto se trabajará con estructuras en memoria (Map) para simular cuentas y operaciones, asegurando la funcionalidad del sistema sin depender de una base de datos.

Con esta propuesta, se busca ofrecer una solución escalable, mantenible y orientada a servicios, que aproveche los beneficios de la web para modernizar el modelo distribuido tradicional, reduciendo la complejidad del desarrollo y mejorando la experiencia del usuario y del desarrollador.

MATERIALES Y METODOS

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

- ✓ **Lenguaje de programación:** Java SE, versión 22, empleado para la construcción de toda la lógica del sistema.
- ✓ **Framework de desarrollo:** Spring Boot 3.4.4, utilizado para la creación y gestión de servicios web RESTful, organización del proyecto y ejecución embebida del servidor.
- ✓ **IDE de desarrollo:** NetBeans 21, entorno de desarrollo integrado que permitió gestionar el proyecto, compilar el código, configurar Maven y ejecutar la aplicación.
- ✓ **Herramienta de pruebas:** Postman, utilizada para simular peticiones HTTP y verificar el funcionamiento de los distintos servicios expuestos.
- ✓ **Gestor de dependencias:** Apache Maven, empleado para administrar las bibliotecas necesarias del proyecto.
- ✓ **Sistema operativo:** Windows 11, plataforma sobre la cual se desarrolló y ejecutó el sistema.
- ✓ **Servidor embebido:** Apache Tomcat (incluido por defecto en Spring Boot), que permite levantar la aplicación como un servicio web sin necesidad de configurar servidores externos.

DESARROLLO DE SOLUCIÓN

La solución planteada se implemento mediante una arquitectura de servicios RESTful utililzando el framework Spring Boot, lo que permitió definir endpoints HTTP de forma estructurada y modular. El sistema simula un entorno bancario básico en el que los usuarios pueden autenticarse y realizar operaciones como consulta de saldo, depósitos, retiros y transferencias, todo a través de peticiones HTTP simples, procesadas por una aplicación Java.

El proyecto fue organizado en tres capas principales*: model*, *service y controller*. Esta separación facilita la mantenibilidad del código y permite que cada componente cumpla con una responsabilidad específica.

Modelo de datos

La clase *Cuenta* representa el modelo de una cuenta bancaria. Contiene tres atributos principales: el número de cuenta, el saldo actual y el NIP (clave de acceso). Esta clase incluye métodos constructores y getters/setters para acceder y modificar los valores, y sirve como estructura base para almacenar información bancaria en memoria.

```
private String numero;
private double saldo;
private int nip;

public Cuenta(String numero, double saldo, int nip) {
    this.numero = numero;
    this.saldo = saldo;
    this.nip = nip;
}
public Cuenta() {
}

public String getNumero() {
    return numero;
}

public void setNumero(String numero) {
    this.numero = numero;
}

public double getSaldo() {
    return saldo;
}

public void setSaldo(double saldo) {
    this.saldo = saldo;
}

public int getNip() {
    return nip;
}
```

```
public void setNip(int nip) {
    this.nip = nip;
}
```

Este modelo fue almacenado en memoria utilizando un *Map*, con el numero de cuenta como clave y el objeto *Cuenta* como valor

Lógica de negocio (Servicios)

La lógica del sistema se implementó en dos servicios: *AutenticacionService y CuentaService*.

El primer servicio se encarga de verificar si un número de cuenta y un NIP coinciden. Internamente, utiliza un *HashMap* para simular una base de datos de cuentas. Si la cuenta existe y el NIP es correcto, se devuelve un mensaje de autenticación exitosa; de lo contrario, se informa del error.

Además, este servicio expone un método auxiliar *obtenerCuenta* que permite a otros servicios acceder a los datos de una cuenta específica, lo que favorece la reutilización de lógica y el acceso controlado.

```
private final Map<String, Cuenta> cuentas = new HashMap<>();

public AutenticacionService() {
    cuentas.put("1234567890", new Cuenta("1234567890", 9500.0, 1234));
    cuentas.put("2468024680", new Cuenta("2468024680", 10000.0, 2468));
    cuentas.put("3692581470", new Cuenta("3692581470", 100000.0, 3692));
    cuentas.put("4826048260", new Cuenta("4826048260", 5000.0, 4862));
}

public String autenticar(String cuenta, int nip) {
    Cuenta c = cuentas.get(cuenta);
    if (c != null && c.getNip() == nip) {
        return "Autenticacion Exitosa";
    }
    return "Error en autenticación";
}

public Cuenta obtenerCuenta(String numero) {
    return cuentas.get(numero);
}

public Map<String, Cuenta> getTodas() {
    return cuentas;
}
```

Por otro lado, *CuentaService* contiene los métodos que implementan la lógica de negocio relacionada con las operaciones bancarias. Las funciones principales son:

- ✓ consultarSaldo(String numero): devuelve el saldo actual de la cuenta si esta existe.
- ✓ depositar(String numero, double monto): aumenta el saldo de la cuenta si el monto es válido.
- ✓ retirar(String numero, double monto): reduce el saldo si hay fondos suficientes.
- ✓ transferir(String origen, String destino, double monto): transfiere fondos entre dos cuentas válidas.

Cada operación incluye validaciones básicas para asegurar la integridad de los datos y evita operaciones inválidas, como depósitos de monto negativo o retiros que exceden el saldo disponible.

```
@Autowired
private AutenticacionService auth;

public String consultarSaldo(String numero) {
    Cuenta c = auth.obtenerCuenta(numero);
    return (c != null) ? "Saldo: $" + c.getSaldo() : "Cuenta no encontrada";
}

public String depositar(String numero, double monto) {
    Cuenta c = auth.obtenerCuenta(numero);
    if (c != null && monto > 0) {
        c.setSaldo(c.getSaldo() + monto);
        return "Depósito exitoso. Nuevo saldo: $" + c.getSaldo();
    }
    return "Error en depósito";
}

public String retirar(String numero, double monto) {
    Cuenta c = auth.obtenerCuenta(numero);
    if (c != null && monto > 0 && c.getSaldo() >= monto) {
        c.setSaldo(c.getSaldo() - monto);
        return "Retiro exitoso. Nuevo saldo: $" + c.getSaldo();
    }
    return "Error en retiro";
}

public String transferir(String origen, String destino, double monto) {
    Cuenta c1 = auth.obtenerCuenta(origen);
    Cuenta c2 = auth.obtenerCuenta(destino);
    if (c1 == null || c2 == null) return "Cuenta(s) no encontrada(s)";
    if (monto <= 0 || monto > c1.getSaldo()) return "Monto inválido o saldo insuficiente";
    c1.setSaldo(c1.getSaldo() - monto);
    c2.setSaldo(c2.getSaldo() + monto);
    return "Transferencia exitosa. Saldo nuevo origen: $" + c1.getSaldo();
}
```

Exposición del servicio (Controladores)

Para permitir el acceso a las funcionalidades mediante peticiones HTTP, se definieron dos controladores REST:

Controlador de autenticación

```java
public class AutenticacionController {

    @Autowired
    private AutenticacionService authService;

    @GetMapping("/{cuenta}/{nip}")
    public String autenticar(@PathVariable String cuenta, @PathVariable int nip) {
        return authService.autenticar(cuenta, nip);
    }
}
```

Este controlador expone el endpoint HTTP */auth/{cuenta}/{nip}* que permite a los usuarios autenticarse en el sistema. Utiliza el servicio de autenticación *AutenticacionService* y responde con un mensaje textual que indica si el acceso fue exitoso o no. La autenticación es el primer paso antes de realizar cualquier otra operación.

Controlador de operaciones bancarias

```java
@RestController
@RequestMapping("/cuenta")
public class CuentaController {

    @Autowired
    private CuentaService cuentaService;

    @GetMapping("/{numero}/saldo")
    public String consultarSaldo(@PathVariable String numero) {
        return cuentaService.consultarSaldo(numero);
    }

    @PostMapping("/{numero}/deposito")
    public String depositar(@PathVariable String numero, @RequestParam double monto) {
        return cuentaService.depositar(numero, monto);
    }

    @PostMapping("/{numero}/retiro")
    public String retirar(@PathVariable String numero, @RequestParam double monto) {
        return cuentaService.retirar(numero, monto);
    }

    @PostMapping("/{origen}/transferencia")
    public String transferir(@PathVariable String origen,
                             @RequestParam String destino,
                             @RequestParam double monto) {
        return cuentaService.transferir(origen, destino, monto);
    }
}
```

Este controlador expone múltiples endpoints para operaciones bancarias. Cada ruta está asociada a una acción específica:

- ✓ GET /cuenta/{numero}/saldo - consulta el saldo de una cuenta.
- ✓ POST /cuenta/{numero}/deposito?monto=... - realiza un depósito.
- ✓ POST /cuenta/{numero}/retiro?monto=...: - efectúa un retiro.
- ✓ POST /cuenta/{numero}/transferencia?destino=...&monto=... - realiza una transferencia entre cuentas.

Las rutas utilizan anotaciones de Spring como *@GetMapping* y *@PostMapping*, y emplean parámetros de URL y query strings para enviar los datos necesarios. Las respuestas son simples mensajes de texto que indican el resultado de la operación.

Pruebas de funcionamiento

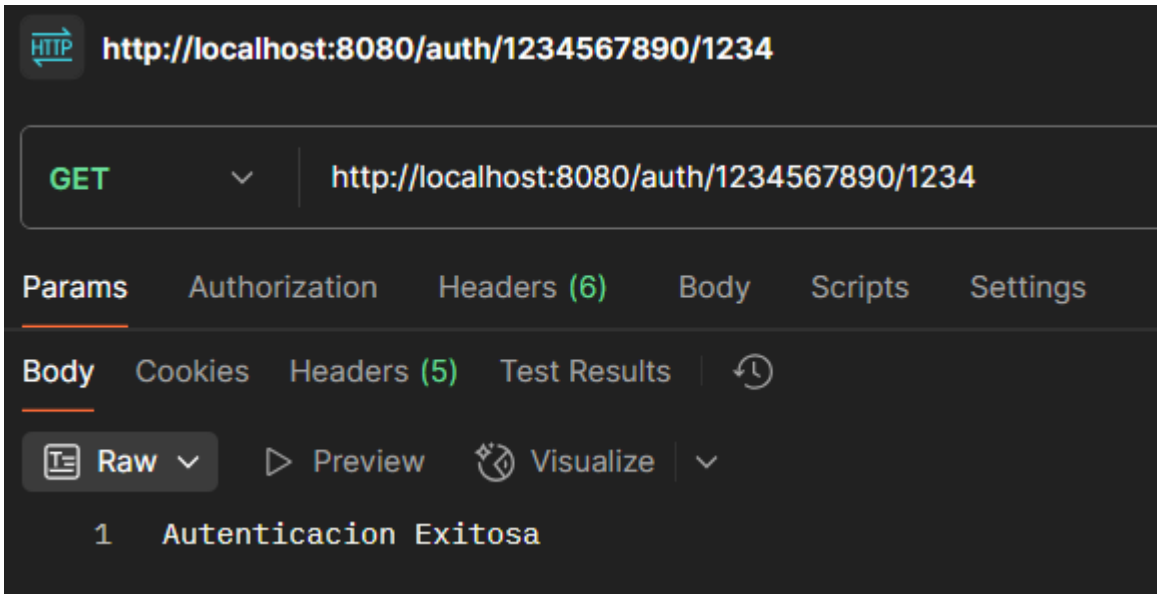
Para probar el correcto funcionamiento del sistema, se utilizó Postman, desde donde se enviaron peticiones HTTP simulando las acciones de los usuarios. Las rutas definidas permiten ejecutar operaciones como:

- ✓ GET /auth/1234567890/1234 → autenticación
- ✓ GET /cuenta/1234567890/saldo → consulta de saldo
- ✓ POST /cuenta/1234567890/deposito?monto=100 → depósito
- ✓ POST /cuenta/1234567890/retiro?monto=50 → retiro
- ✓ POST /cuenta/1234567890/transferencia?destino=2468024680&monto=200 → transferencia

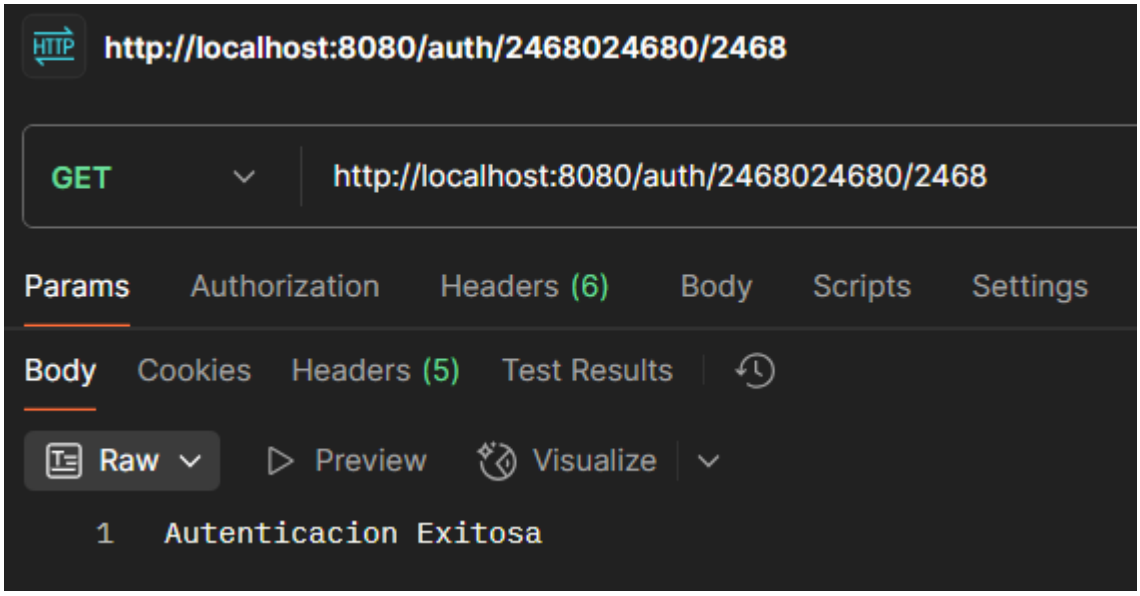
RESULTADOS

AUTENTICACION

Enviando petición de cliente A para su autenticación para acceder al sistema bancario



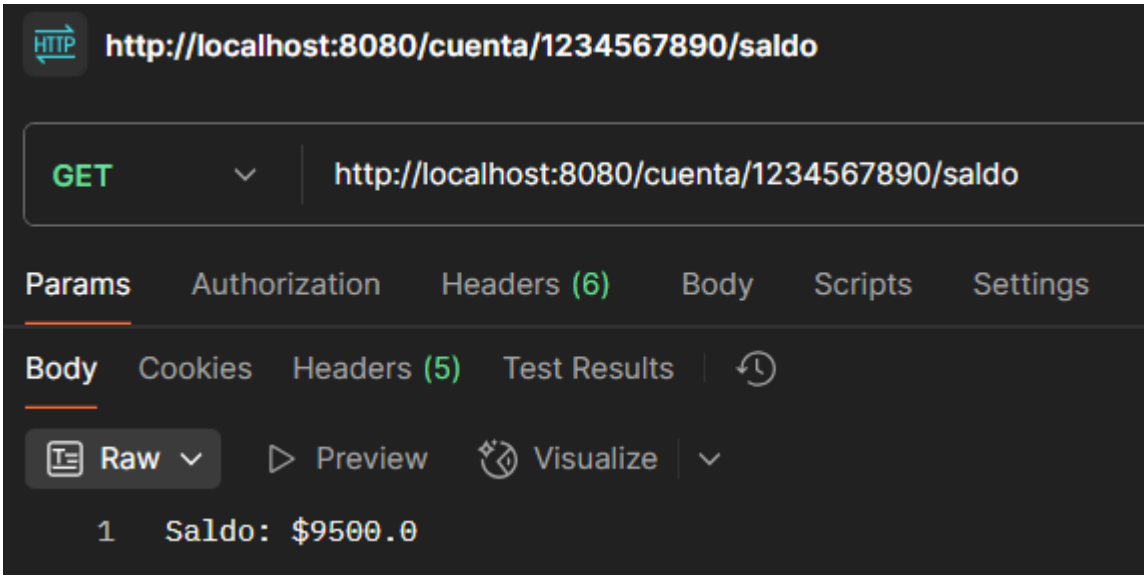
Enviando petición del cliente B para realizar su autenticación para acceder al sistema bancario



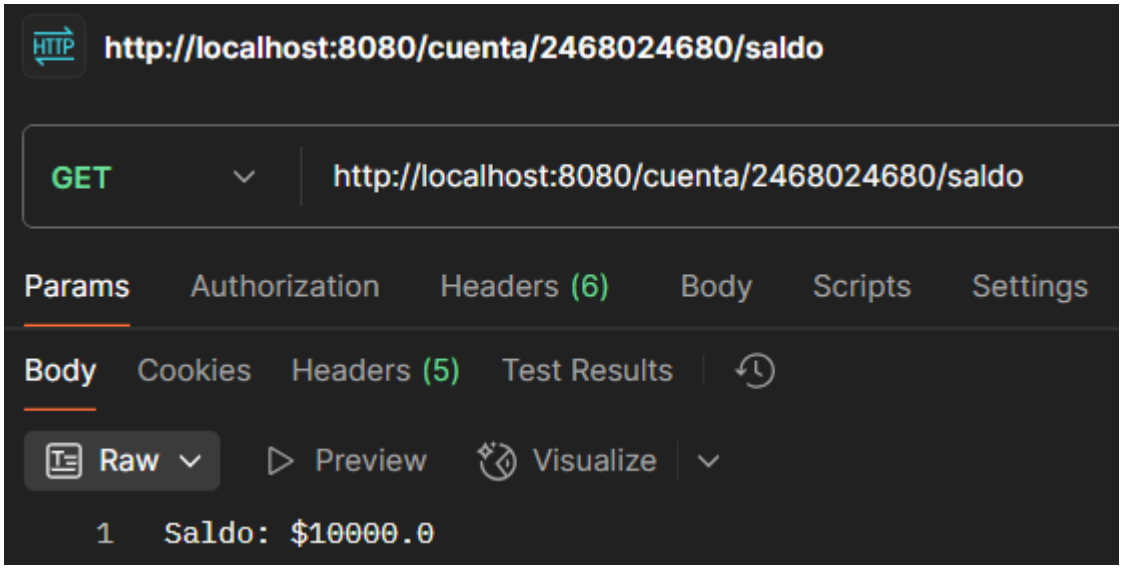
CONSULTA DE SALDO INICIAL

Ambos clientes realizan la consultad de saldo inicial antes de realizar operaciones para verificar cuanto dinero tienen disponible

Cliente A

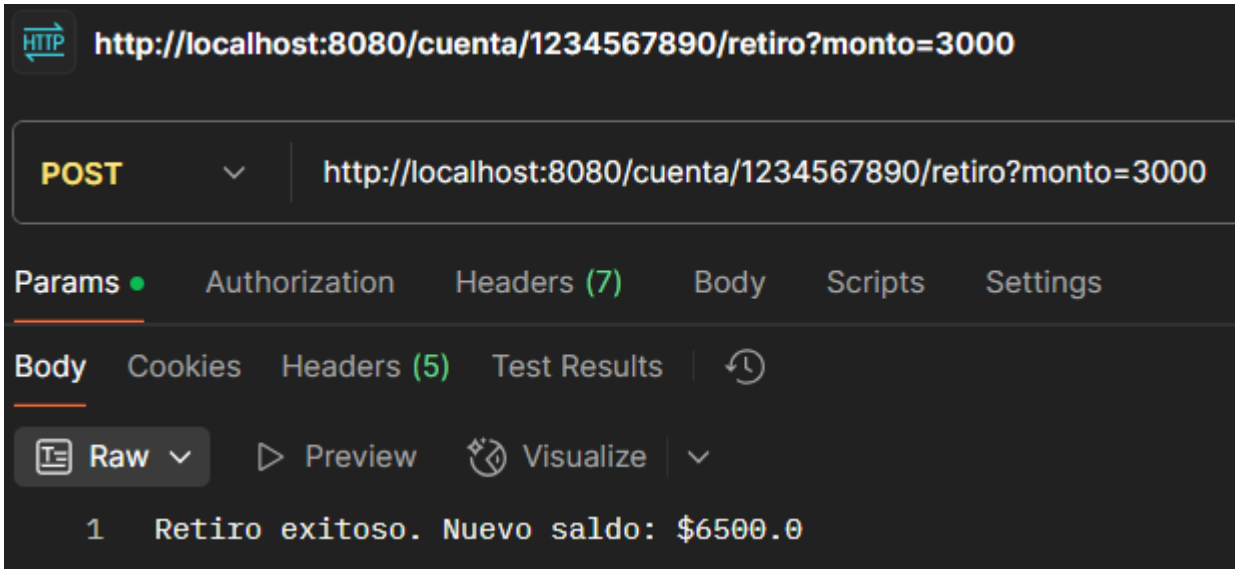


Cliente B

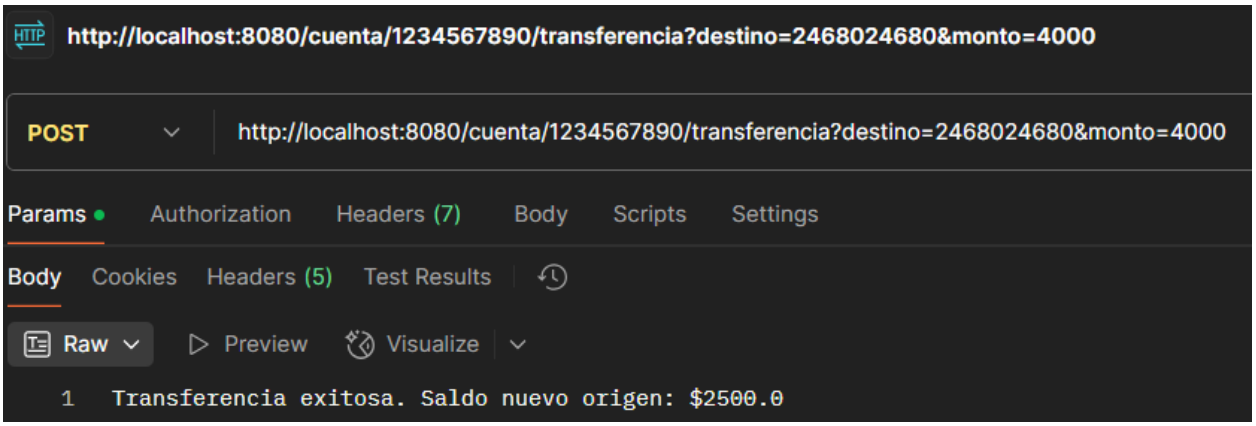


Ahora cliente A realiza un retiro mientras cliente B intenta hacer una trasferencia desde la cuenta de cliente A

Cliente A

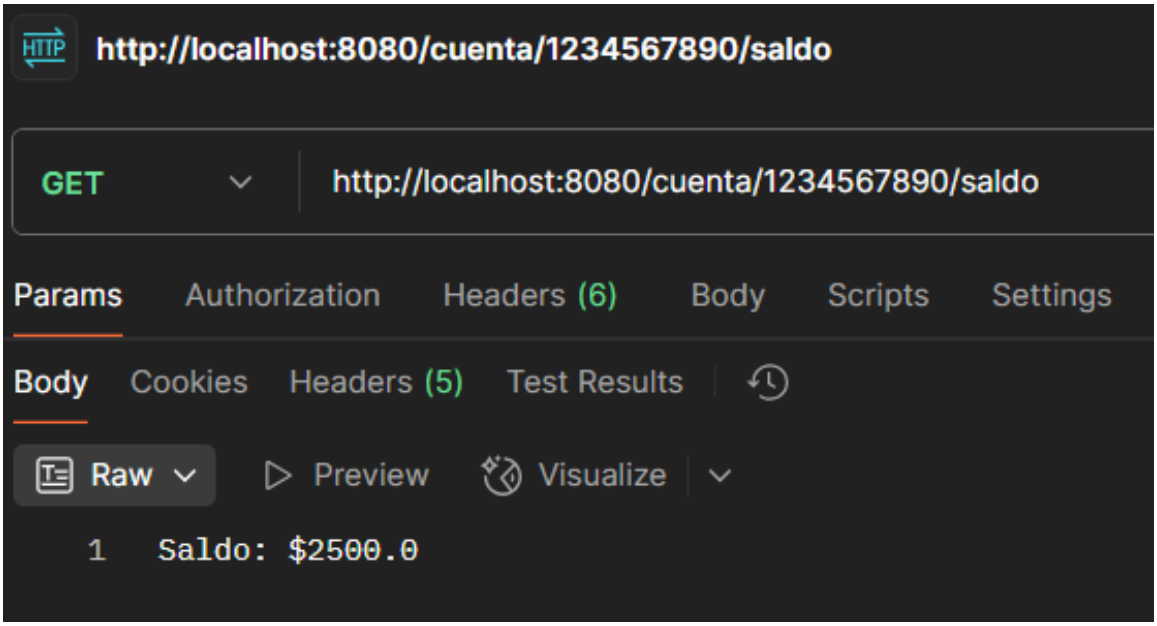


Cliente B

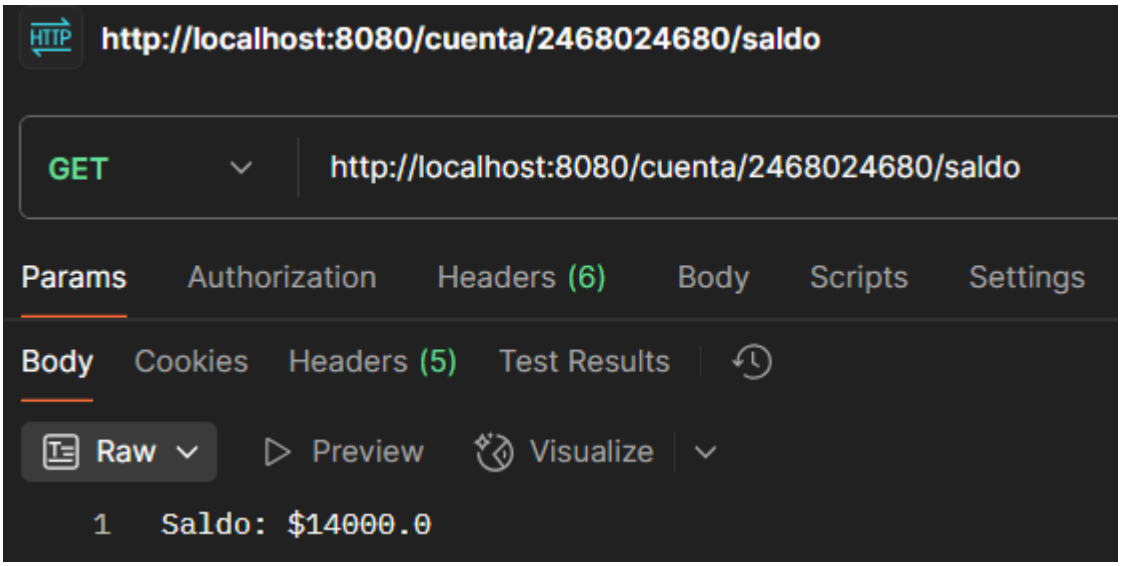


De esta manera generamos un escenario concurrente, donde ambos acceden a la misma cuenta al mismo tiempo

Posteriormente, ambos clientes realizando la consultado final de su saldo, obteniendo lo siguiente:



El cliente A termino con un saldo final menor al inicial derivado de las transacciones realizadas (retiro de 3,000 + Transferencia de 4,000)



El cliente B termino con saldo final mayor derivado del deposito recibido por 3,000 por el cliente A

CONCLUSIONES

A lo largo del desarrollo de esta práctica, me fue posible comprender e implementar un sistema bancario funcional empleando servicios web bajo el enfoque RESTful, utilizando el framework Spring Boot. Este proceso permitió adquirir experiencia práctica en la construcción de servicios escalables, desacoplados y accesibles a través de peticiones HTTP, al mismo tiempo que se consolidó el conocimiento sobre arquitectura de software, controladores, servicios y modelos en el contexto de aplicaciones web modernas.

En comparación con el modelo cliente-servidor tradicional, así como con la implementación previa basada en objetos distribuidos mediante Java RMI, se evidenciaron claras ventajas. Entre ellas destacan la facilidad de acceso desde cualquier plataforma o cliente HTTP, la ausencia de dependencias directas entre cliente y servidor, gracias al uso de JSON como formato de comunicación, y la mayor capacidad de mantenimiento y extensión del sistema. La modularidad natural que ofrece Spring Boot permitió una mejor organización del código y una separación clara de responsabilidades, lo cual facilita la escalabilidad y futuras mejoras.

Además, al abordar temas como la concurrencia y sincronización en entornos distribuidos, se pudo reforzar la importancia de proteger operaciones críticas para garantizar la integridad de los datos. En este caso, la sincronización fue implementada directamente en el servicio, asegurando que múltiples operaciones simultáneas sobre una misma cuenta no generaran inconsistencias, simulando condiciones reales de uso en sistemas bancarios.

En conclusión, esta práctica no solo fortaleció mis habilidades técnicas sobre el desarrollo de servicios web, sino que también me permitió contrastar diferentes paradigmas de comunicación distribuida, resaltando los beneficios del enfoque REST en términos de flexibilidad, interoperabilidad y facilidad de integración con otras tecnologías.