UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Diseño y desarrollo de un sistema de monetización con arquitectura monolítica para clientes

INFORME DE ACTIVIDADES PROFESIONALES

Que para obtener el título de

Ingeniero Geomático

P R E S E N T A

Josué Isaac Estrada Zermeño

ASESORA DE INFORME

Dra. Griselda Berenice Hernández Cruz

Ciudad Universitaria, Cd. Mx., 2025

PROTESTA UNIVERSITARIA DE INTEGRIDAD Y HONESTIDAD ACADÉMICA Y PROFESIONAL
(Titulación con trabajo escrito)

De conformidad con lo dispuesto en los artículos 87, fracción V, del Estatuto General, 68, primer párrafo, del Reglamento General de Estudios Universitarios y 26, fracción I, y 35 del Reglamento General de Exámenes, me comprometo en todo tiempo a honrar a la institución y a cumplir con los principios establecidos en el Código de Ética de la Universidad Nacional Autónoma de México, especialmente con los de integridad y honestidad académica.

De acuerdo con lo anterior, manifiesto que el trabajo escrito titulado DISEÑO Y DESARROLLO DE UN SISTEMA DE MONETIZACION CON ARQUITECTURA MONOLITICA PARA CLIENTES que presenté para obtener el titulo de INGENIERO GEOMÁTICO es original, de mi autoría y lo realicé con el rigor metodológico exigido por mi Entidad Académica, citando las fuentes de ideas, textos, imágenes, gráficos u otro tipo de obras empleadas para su desarrollo.

En consecuencia, acepto que la falta de cumplimiento de las disposiciones reglamentarias y normativas de la Universidad, en particular las ya referidas en el Código de Ética, llevará a la nulidad de los actos de carácter académico administrativo del proceso de titulación.

JOSUE ISAAC ESTRADA ZERMEÑO
Número de cuenta: 313162468

Diseño y desarrollo de un sistema de monetización con arquitectura monolítica para clientes del Grupo Salinas

ÍNDICE

INTRODUCCIÓN

El siguiente trabajo busca implementar la parte informática de la geomática en el ambiente profesional durante el proyecto de desarrollo de un sistema de monetización, para clientes de **Grupo Salinas** como parte de **Alnova**.

En la era de la información, la gestión eficiente de los datos es fundamental para optimizar procesos y sistematizarlos implementando de esta manera las últimas tecnologías. La geomática, una disciplina que integra la recopilación, análisis y representación de datos geoespaciales, ha encontrado un aliado poderoso en las tecnologías emergentes tomando de esta manera más relevancia en el mundo TI, como los microservicios, el **middleware** y los procedimientos almacenados.

El sistema propuesto se basó en un flujo de trabajo donde un procedimiento almacenado inicial consulta la información relevante y la envía a un microservicio. Este microservicio procesa los datos y, a continuación, los transmite a otro procedimiento almacenado para su impacto en la base de datos. El **middleware**, utilizando servicios **Unirest**, orquesta esta comunicación, asegurando que los datos procesados se envíen correctamente al **frontend** para su presentación final. Este enfoque permite una integración y gestión de datos más eficaz, proporcionando una respuesta rápida y precisa a las consultas y operaciones requeridas por el sistema de monetización.

La integración de la geomática en este sistema es crucial, ya que la precisión y la puntualidad en el manejo de datos geoespaciales son esenciales para diversas aplicaciones, desde la cartografía hasta la gestión de recursos naturales. Los procedimientos almacenados y los microservicios facilitan el procesamiento de grandes volúmenes de datos geoespaciales, mientras que el **middleware** garantiza una comunicación fluida y eficiente entre los distintos componentes del sistema.

Este trabajo de titulación no solo presenta una solución técnica avanzada para la monetización de datos, sino que también demuestra cómo la combinación de geomática y tecnologías de la información puede mejorar significativamente la capacidad operativa y la eficiencia en la gestión de datos. A través de esta investigación, se destacan los beneficios de adoptar un enfoque basado en microservicios y **middleware** para resolver desafíos complejos en el ámbito de la geomática.

OBJETIVO GENERAL

Desarrollo de procedimientos almacenados, servicios **REST** y microservicios para la automatización de los sistemas de monetización en formato **json**.

1.SISTEMAS DE MONETIZACIÓN

1.1. Definición

Un sistema de monetización es un conjunto de procesos y herramientas que permiten convertir productos, servicios o datos en ingresos. Este tipo de sistemas se utiliza en diversas industrias, desde aplicaciones móviles hasta plataformas de comercio electrónico, este caso particular es un sistema de monetización para un grupo bancario, cuyas operaciones como la generación de pólizas, recarga de saldo, entre otras cobran una comisión.

1.2. Modelos de bases de datos

Existen varios modelos de bases de datos, entre los cuales podemos encontrar las bases de datos relacionales que son las que organizan la información mediante el uso de filas y columnas, permitiendo relaciones entre diferentes tablas (Fig. 1). Este tipo de base de datos tiene las siguientes ventajas:

1. Coherencia e integridad de los datos: garantiza que los datos sean precisos y consistentes.
2. Facilidad de uso: Utilizan SQL, un lenguaje estándar para gestionar y consultar datos. Ejemplos de manejadores: MySQL, PostgreSQL, Oracle Database.

Por otro lado también encontramos el caso de las bases de datos no relacionales (NoSQL), las cuales no implican el uso de tablas sino que usan varios modelos de datos como documentos, gráficos, clave-valor, y en memoria. Este tipo de base de datos tiene las siguientes ventajas:

1. Flexibilidad: Pueden manejar datos semi estructurados y no estructurados.
2. Escalabilidad: Diseñadas para manejar grandes volúmenes de datos y alta velocidad. Ejemplos: MongoDB, Cassandra, Redis(Amazon Web Services, 2024).

Se decidió utilizar el modelo de bases de datos relacionales en lugar de otros, debido a que la información del negocio a manejar es del tipo **e-commerce** en el ámbito bancario (Fig.1), por lo que se necesitaba cumplir con las siguientes características:

1. Integridad de los datos: Las bases de datos relacionales aseguran que los datos estén completos y sin errores.
2. Estándares bien establecidos: SQL es un lenguaje ampliamente aceptado y utilizado, facilitando la interacción y la formación de personal.
3. Estructura clara: La organización en tablas y relaciones es fácil de entender y manejar para aplicaciones que requieren transacciones complejas y datos interrelacionados(Amazon Web Services, 2024).

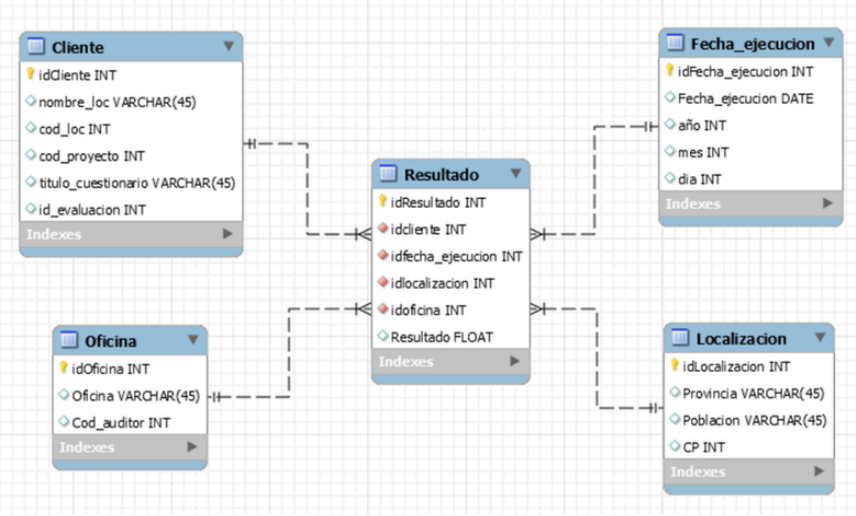

Figura 1. Estructura de una base de datos relacional, donde cada tabla es un objeto cuya propiedad primaria que es el valor único e irrepetible del objeto (llave primaria) está marcada por una llave y cada objeto está relacionado por una línea que los une,

1.3. Arquitecturas

Las arquitecturas de sistemas de monetización pueden ser monolíticas o basadas en microservicios.

Arquitectura monolítica: Todo el sistema se construye como una única unidad de software. Es más simple y fácil de implementar para proyectos pequeños (todo el código se encuentra en un solo archivo).

Arquitectura de microservicios: El sistema se divide en pequeñas partes independientes que se comunican entre sí. Es más escalable y fácil de mantener para proyectos grandes y complejos(Fig. 2).

Para esta aplicación web se decidió utilizar una arquitectura de microservicios ya que como cada microservicio es independiente uno del otro no se saturan de tareas esto porque cada uno hace tareas específicas y si al momento de desplegar la aplicacion alguna parte de la misma sufre afectación o falla, no toda la aplicación fallará sino únicamente esa parte de la aplicación, de manera que los otros servicios funcionarán con normalidad (Atlassian, 2024).

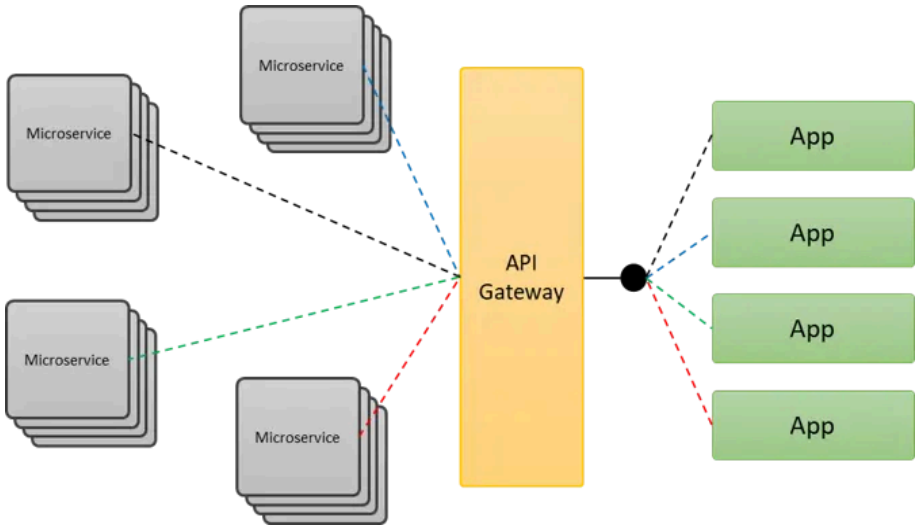

Figura 2. Arquitectura de microservicios: Servicios individuales que pasan como *API Gateway y posteriormente son consumidos.*

1.4. Formato JSON

JavaScript Object Notation (JSON) es un formato de texto ligero cuyo propósito es el de intercambiar datos en aplicaciones web de una manera rápida y eficiente, cuya sintaxis se basa en el lenguaje JavaScript, sin embargo es independiente a cualquier lenguaje de programación. La selección del formato JSON en un proyecto que usa la tecnología **REST** se debe a que permite una fácil representación de los recursos que se manipulan, una comunicación efectiva entre clientes y servidores, además de facilitar operaciones, como lectura, creación, actualización y eliminación de recursos. Su estructura se compone de una clave y su correspondiente valor, como se muestra en el siguiente objeto JSON:

```
{
        “nombre”: “Isaac”,
        “Edad”: 30,
        “País”: “México”
}
```

En este objeto JSON que representa datos básicos de una persona, podemos observar tres claves las cuales son nombre, edad y país, con su correspondiente valor.

2. DISEÑO DE LOS SISTEMAS DE MONETIZACIÓN

2.1. Generalidades de los microservicios .NET

Los microservicios son pequeños servicios independientes que realizan tareas específicas, véase Figura 2. En este proyecto, se usan microservicios desarrollados en C# .NET. De manera que estos servicios, reciben diferentes datos cada uno a través de un procedimiento almacenado, los procesan y luego envían los resultados a otro procedimiento almacenado que impactarán modificando los registros contenidos en la base de datos Oracle (Amazon Web Services, 2024).

2.2. Base de Datos

Se utiliza Oracle como sistema de gestión de bases de datos, donde se almacenan y gestionan los datos. Los procedimientos almacenados son bloques de código SQL que realizan operaciones específicas en la base de datos, facilitando el manejo eficiente de grandes volúmenes de datos, evitando así tener que realizar las tareas manualmente.

2.3. Spring Boot

Spring Boot es un marco de trabajo en Java que facilita la creación de servicios web. En este sistema, los microservicios .NET interactúan con servicios REST desarrollados con Spring Boot, lo que permite una comunicación eficiente y segura entre las distintas partes del sistema ya que dentro de ella se llevarán a cabo procesos de autenticación de accesos, expiración de acceso, encriptado y desencriptado de información, conexión de la aplicación a base de datos así como la vista de la aplicación con la funcionalidad de los microservicios, estos servicios **Rest** fungen como un intermediario que comunica a todas las partes del sistema (Cecilio Álvarez Caules, 2023).

3. ESTRUCTURA Y DESARROLLO DE CÓDIGO

3.1. Microservicios con C#

En la programación, los entornos de desarrollo integrado (IDE) son entornos para la creación y prueba de proyectos. *Visual Studio*, es un IDE con diferentes facilidades entre ellas el manejo del lenguaje **c#**. Se utilizó *Visual Studio 2022*, debido al trabajo con la versión *Net 6.0*.

El primer paso dentro de la metodología es crear un proyecto *API* de *ASP.NET Core*. Para crear un proyecto API web de *ASP.NET* Core es necesario hacer lo siguiente:

1. Abrir *Visual Studio* y seleccionar "Crear un nuevo proyecto".
2. En la ventana de plantillas de proyecto, seleccione *"Aplicación web"* y luego *"API web de ASP.NET Core"*.
3. Asignar un nombre y una ubicación para el proyecto y hacer clic en "Crear".
4. Una vez creado el proyecto, se pueden comenzar a implementar controladores y lógica de negocio en los *endpoints* de la API.
5. Para interactuar con el *stored procedure* de Oracle, se puede utilizar la biblioteca *Oracle.ManagedDataAccess.Client*, que proporciona las clases y métodos necesarios para conectarse y ejecutar *stored procedures* en una base de datos Oracle.

6. Dentro de la clase de microservicio, se puede utilizar una clase para establecer y administrar la conexión con la base de datos Oracle.
7. Utilizar la clase *OracleCommand* para llamar al *stored procedure* de Oracle que envía información y al *stored procedure* que impacta en la base de datos. Configurar los parámetros necesarios y ejecutar los comandos correspondientes.

Con los pasos anteriores se crea un microservicio que recibe información de un *store procedure* de la base de datos, se procesa la información y luego envía esa información a otro *stored procedure* que impacta en la base de datos.

La base de datos "impactada" se administra mediante una clase *OracleConnectionManager*, bajo el siguiente código:

```
using Oracle.ManagedDataAccess.Client;

public class OracleConnectionManager
{
    private string _connectionString;

    public OracleConnectionManager(string connectionString)
    {
        _connectionString = connectionString;
    }

    public OracleConnection GetConnection()
    {
        return new OracleConnection(_connectionString);
    }
}
```

Este código crea una instancia de la clase *OracleConnectionManager* y llama al método *GetConnection*, se obtiene un objeto OracleConnection que se puede utilizar para interactuar con la base de datos Oracle. Cada línea hace lo siguiente:

- **using Oracle.ManagedDataAccess.Client**: Esta línea importa el espacio de nombres *Oracle.ManagedDataAccess.Client*, que contiene las clases necesarias para interactuar con una base de datos Oracle.
- **public class OracleConnectionManager**: Esta línea define una clase llamada *OracleConnectionManager*, que se encargará de administrar la conexión con la base de datos Oracle.

- **private string _connectionString**: Esta línea declara una variable privada llamada *_connectionString* que almacenará la cadena de conexión necesaria para establecer la conexión con la base de datos Oracle.
- **public OracleConnectionManager(string connectionString)**: Este es el constructor de la clase *OracleConnectionManager*. Recibe una cadena de conexión como parámetro y asigna ese valor a la variable *_connectionString*.
- **public OracleConnection GetConnection()**: Este es un método público llamado *GetConnection* que devuelve un objeto *OracleConnection*. Este método se utiliza para obtener una instancia de la clase *OracleConnection*, que representa una conexión con la base de datos Oracle. La conexión se establece utilizando la cadena de conexión almacenada en la variable *_connectionString*.
- **return new OracleConnection(_connectionString)**: Esta línea crea una nueva instancia de la clase *OracleConnection* utilizando la cadena de conexión almacenada en la variable *_connectionString* y la devuelve como resultado.

Una vez establecida la conexión es necesario comenzar a utilizar los *store procedure* de la misma, tanto para recibir como para enviar información y realizar el impacto, esta misma clase por sí sola también puede ser capaz de procesar la información que consulta el *store procedure* haciendo diferentes cálculos o procesos específicos que se deseen. El siguiente código es un ejemplo de cómo realizar esta clase:

```
using Oracle.ManagedDataAccess.Client;

public class Microservice
{
    private OracleConnectionManager _connectionManager;

    public Microservice(OracleConnectionManager connectionManager)
    {
        _connectionManager = connectionManager;
    }

    public void ProcessData()
    {
        // Lógica para recibir información y procesarla
        using (OracleConnection connection = _connectionManager.GetConnection())
        {
            // Lógica para llamar al stored procedure de Oracle que envía información
                                                using (OracleCommand command = new
OracleCommand("nombre_del_stored_procedure_enviar", connection))
            {
                command.CommandType = CommandType.StoredProcedure;
                // Configurar parámetros del stored procedure
                command.Parameters.Add("parametro1", OracleDbType.Varchar2).Value = "valor1";
                command.Parameters.Add("parametro2", OracleDbType.Int32).Value = 123;

                connection.Open();
                command.ExecuteNonQuery();
            }
```

```
        // Lógica para llamar al stored procedure de Oracle que impacta en la base de datos
                                        using (OracleCommand command = new
OracleCommand("nombre_del_stored_procedure_impactar", connection))
        {
            command.CommandType = CommandType.StoredProcedure;
            // Configurar parámetros del stored procedure
            command.Parameters.Add("parametro3", OracleDbType.Varchar2).Value = "valor2";
            command.Parameters.Add("parametro4", OracleDbType.Int32).Value = 456;

            command.ExecuteNonQuery();
        }
    }
  }
}
```

Este código define una clase *Microservice* que se encarga de procesar datos utilizando *stored procedures* de Oracle. Donde cada línea realiza lo siguiente:

- **using Oracle.ManagedDataAccess.Client**: Esta línea importa el espacio de nombres *Oracle.ManagedDataAccess.Client*, que contiene las clases necesarias para interactuar con una base de datos Oracle.
- **public class Microservice**: Esta línea define una clase llamada *Microservice*, que representa el microservicio en sí.
- **private OracleConnectionManager _connectionManager**: Esta línea declara una variable privada llamada *_connectionManager* del tipo *OracleConnectionManager*. Esta variable se utiliza para administrar la conexión con la base de datos Oracle.
- **public Microservice(OracleConnectionManager connectionManager)**: Este es el constructor de la clase *Microservice*. Recibe una instancia de la clase *OracleConnectionManager* como parámetro y asigna esa instancia a la variable *_connectionManager*.
- **public void ProcessData()**: Este es un método público llamado *ProcessData* que no devuelve ningún valor. Este método se encarga de recibir información y procesarla utilizando *stored procedures* de Oracle.
- **using (OracleConnection connection = _connectionManager.GetConnection())**: Esta línea utiliza la instancia de *OracleConnectionManager* para obtener una conexión a la base de datos Oracle mediante el método *GetConnection()*. La conexión se establece dentro de un bloque *using*, lo que garantiza que se libere correctamente al finalizar su uso.
- **using (OracleCommand command = new OracleCommand("nombre_del_stored_procedure_enviar", connection))**:

Esta línea crea una instancia de *OracleCommand* para llamar al *stored procedure* de Oracle que envía información. Se especifica el nombre del *stored procedure* y se pasa la conexión como parámetro.

- **command.CommandType = CommandType.StoredProcedure**: Esta línea establece el tipo de comando como un stored procedure.
- **command.Parameters.Add("parametro1", OracleDbType.Varchar2).Value = "valor1"**: Estas líneas agregan parámetros al comando del *stored procedure*. Se especifica el nombre del parámetro, el tipo de dato (*OracleDbType*) y se asigna un valor.
- **connection.Open()**: Esta línea abre la conexión con la base de datos Oracle.
- **command.ExecuteNonQuery()**: Esta línea ejecuta el comando del stored procedure que envía información.
- **using (OracleCommand command = new OracleCommand("nombre_del_stored_procedure_impactar", connection))**: Esta línea crea otra instancia de *OracleCommand* para llamar al *stored procedure* de Oracle que impacta en la base de datos. Se especifica el nombre del *stored procedure* y se pasa la conexión como parámetro.
- **command.CommandType = CommandType.StoredProcedure**: Esta línea establece el tipo de comando como un *stored procedure*.
- **command.Parameters.Add("parametro3", OracleDbType.Varchar2).Value = "valor2"**: Estas líneas agregan parámetros al comando del *stored procedure*. Se especifica el nombre del parámetro, el tipo de dato (*OracleDbType*) y se asigna un valor.
- **command.ExecuteNonQuery()**: Esta línea ejecuta el comando del *stored procedure* que impacta en la base de datos.

Además de estas clases en una arquitectura de microservicios es necesario considerar el despliegue de los mismos para ser utilizados (consumidos), existen diferentes formas de realizar esto sin embargo, para no perder la escalabilidad, simpleza y aislamiento, minimizando conflictos y errores, se hace el uso de contenedores, en este caso de contenedores *docker*, ya que estos empaquetan el servicio en un contenedor para después desplegarlos. El siguiente código es un ejemplo de un archivo *Docker* que se utiliza para construir una imagen de contenedor para el microservicio:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build-env
WORKDIR /app

# Copiar y restaurar archivos de proyecto
```

```
COPY *.csproj ./
RUN dotnet restore

# Copiar y compilar el código fuente
COPY . ./
RUN dotnet publish -c Release -o out

# Configurar el contenedor y ejecutar el microservicio
FROM mcr.microsoft.com/dotnet/aspnet:6.0
WORKDIR /app
COPY --from=build-env /app/out .
ENTRYPOINT ["dotnet", "NombreDelMicroservicio.dll"]
```

El código establece la imagen base como el *SDK* de *.NET Core 6.0* para construir el entorno de desarrollo, posteriormente, configura el directorio de trabajo en el contenedor como */app*, copia y restaura los archivos de proyecto, luego copia y compila el código fuente, configurando el contenedor para ejecutar el microservicio, copiando la aplicación publicada en el contenedor y definiendo el punto de entrada como *"NombreDelMicroservicio.dll"*.

3.2. Procedimientos almacenados.

En el desarrollo de arquitectura **Rest**, la creación y administración de las bases de datos es indispensable al igual que su automatización, la cual se puede llevar a cabo por medio de disparadores o procedimientos almacenados. La implementación de dichos procedimientos almacenados aligera la carga de trabajo sobre los microservicios, además de otorgar más restricciones garantizando mayor seguridad para el acceso a la información, lo cual se lleva a cabo con administradores de base de datos, para ello existen diferentes administradores, no obstante el presente trabajo se centra en *Oracle Sql Developer*.

La metodología para crear procedimientos en *Oracle Sql Developer*, es la siguiente:

1. Abrir la herramienta de administración *Oracle SQL Developer*.
2. Realizar la conexión a la base de datos en la que se desea crear el procedimiento almacenado.
3. Utilizar la siguiente estructura para crear el procedimiento almacenado:

```
CREATE OR REPLACE PROCEDURE nombre_del_procedimiento
IS
BEGIN
   -- Lógica del procedimiento almacenado
   NULL;
END;
```

4. Reemplazar "nombre_del_procedimiento" con el nombre a dar al procedimiento almacenado.
5. Dentro del bloque "BEGIN" agregar la lógica del procedimiento almacenado, como consultas, actualizaciones, inserciones, etc.
6. Guardar y ejecutar el script para crear el procedimiento almacenado.

Este procedimiento se realiza dos veces uno para crear el procedimiento almacenado que consulta la información enviándola al microservicio y otra para el procedimiento almacenado que recibe la información procesada para impactar en la base de datos, la estructura del **codigo** sera la misma, sin embargo la lógica del proceso varía dependiendo lo que se quiera realizar, las tablas y datos que se desean utilizar.

3.3. Servicios REST (*Middleware*).

Los servicios REST permiten el fácil consumo de API's los cuales no son más que diferentes *endpoints*, esto significa que cada microservicio con su respectivo procedimiento almacenado, es montado en la nube dentro de una arquitectura en línea por un arquitecto acompañado de un diseñador API, los cuales dependiendo de los procesos que realice el microservicio deciden qué *endpoint o endpoints* (url de tipo HTTP) le corresponde y el tipo de petición o peticiones que le corresponde. Algunos tipos de peticiones son:

GET: Consulta de información.

POST: Crear nueva información.

PUT: Actualizar información.

DELETE: Eliminación de información.

Al momento de llevar a cabo el desarrollo de servicios REST existen varios lenguajes de programación de los cuales se selecciona el lenguaje JAVA debido a la fácil implementación de estos servicios mediante el uso de librerías de *spring boot, spring data, spring cloud, maven spring* entre otras. Mientras que la IDE de desarrollo es *Spring tool suite* 4.0 debido a que es una IDE que no está licenciada por lo que sus recursos son gratuitos.

Entonces para crear un servicio REST usando la estructura modelo-vista-controlador (MVC) con Java 17 en *Spring Boot* se siguen los siguiente pasos:

1. Abrir *Spring Tool Suite*
2. Crear un Nuevo Proyecto

- Ve a *File* > *New* > *Spring Starter Project*.

- Completar los campos requeridos como el nombre del proyecto, el grupo y el artefacto. Seleccionar Java 17 como versión de Java.
- Hacer clic en Finish para crear el proyecto.

3. Navegar a la Estructura del Proyecto

- En el panel de *Package Explorer*, expandir el proyecto recién creado.
- Navegar a *src/main/java* donde se encuentra el código fuente.

4. Crear un Nuevo Paquete

- Haz clic derecho en el directorio java y selecciona *New* > *Package*.
- Asignar un nombre al paquete, por ejemplo, com.ejemplo.demo.

5. Crear una Nueva Clase

- Hacer clic derecho en el paquete que se acaba de crear y seleccionar *New* > *Class*.
- En el cuadro de diálogo, ingresar el nombre de la clase, por ejemplo, MiClase.
- Se pueden seleccionar las opciones para generar métodos como *public static void main(String[] args)* si lo deseas.
- Hacer clic en *Finish*.

Tomando en cuenta los pasos anteriores se pueden crear varias clases modelos, controladores y servicios, para todos y cada uno de los *endpoint* o microservicios como ejemplo la clase llamada "*Location*" en la cual van a ir los datos de entrada y salida, cuyo código es el siguiente:

```
package com.ejemplo.model;

public class Location {
    private String name;
    private double latitude;
    private double longitude;

    public Location(String name, double latitude, double longitude) {
        this.name = name;
        this.latitude = latitude;
        this.longitude = longitude;
    }

    // Getters y Setters
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getLatitude() {
        return latitude;
    }

    public void setLatitude(double latitude) {
        this.latitude = latitude;
    }

    public double getLongitude() {
        return longitude;
```

```
    }
    public void setLongitude(double longitude) {
        this.longitude = longitude;
    }
}
```

La clase *Location* es un modelo que representa una ubicación geográfica con un nombre, latitud y longitud. Incluye un constructor para inicializar estos valores y métodos de acceso (*getters*) y modificación (*setters*) para interactuar con las propiedades de la clase. Esto es útil en aplicaciones que manejan datos geográficos, permitiendo encapsular la información de manera organizada y accesible.

Una vez establecido el modelo de datos de entrada y/o salida es necesario crear una clase que contenga la lógica de negocio, generalmente asignado a un paquete *service*, que dentro va a tener una clase *service*, el siguiente código es el correspondiente a la lógica de negocios:

```
package com.ejemplo.service;

import com.ejemplo.model.Location;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.util.List;

@Service
public class GeoService
 {
    private final String oracleServiceUrl = "http://localhost:8080/oracle/locations";
     // URL del microservicio

    public List<Location> getAllLocations() {
        RestTemplate restTemplate = new RestTemplate();
        Location[] locations = restTemplate.getForObject(oracleServiceUrl, Location[].class);
        return List.of(locations);

    }

}
```

Esta clase *GeoService* define un servicio en una aplicación *Spring* que se conecta a un microservicio para obtener datos de ubicación. Utiliza *RestTemplate* para realizar una solicitud HTTP a un *endpoint* específico y devuelve los datos en forma de lista de

objetos *Location*. Este enfoque permite desacoplar la lógica de negocio de la fuente de datos, facilitando la escalabilidad y el mantenimiento del código.

Lo siguiente es establecer el controlador el cual define las rutas de los *endpoint* específicos a los cuales les estamos enviando la solicitud, recordar que cada *endpoint* es un microservicio montado en la nube. El siguiente código corresponde al controlador del servicio *Geoservice*:

```
package com.ejemplo.controller;
import com.ejemplo.model.Location;
import com.ejemplo.service.GeoService;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
public class GeoController {
    private final GeoService geoService;
    public GeoController(GeoService geoService) {
        this.geoService = geoService;
    }

    @GetMapping("/api/locations")
    public List<Location> getLocations() {
        return geoService.getAllLocations();

    }
}
```

Entonces este controlador REST en una aplicación *Spring Boot* que maneja solicitudes para obtener una lista de ubicaciones. Utiliza la inyección de dependencias para acceder a la lógica de negocio a través del servicio *GeoService*, y está diseñado para responder a solicitudes HTTP GET en la ruta */api/locations*. Todo esto para al final enviar un body json el cual será captado por el desarrollador frontend, que conecta este proceso a una interfaz gráfica la cual será la vista de página que ve el usuario (cliente).

4. Conclusiones

Con base en las actividades realizadas en el proyecto desde su definición, desarrollo hasta su publicación se concluye lo siguiente:

- Las arquitecturas a través de microservicios reducen la cantidad de código al dividir cada tarea en un servicio independiente.
- Los sistemas de monetización van a fallar menos al encapsular cada tarea en un servicio ya que si una tarea falla las demás siguen funcionando con normalidad.
- La implementación de store procedures no es estrictamente necesaria para conectarse e impactar a la base de datos sin embargo su implementación brinda más seguridad al proceso y mayor velocidad al dividirse las tareas con los microservicios.
- La definición de un modelo de base de datos correctamente estructurado permite escalar una base de datos a nivel internacional, soportando millones de datos.
- Los servicios **Rest** facilitan el consumo de apis y su exposición al usuario estandarizando por medio de respuestas HTTP la información que le será entregada a la parte frontal del sistema, que es la parte gráfica que ve el usuario.
- Cada acción que realizamos en una plataforma o aplicación corresponde a una petición rest que consume un microservicio ya sea dar de alta un usuario en una aplicación o dar permisos de ubicación (petición Post), consultar nuestro saldo (solicitud Get), actualizar nuestra información de perfil (Put) hasta incluso borrar nuestra información de una aplicación (Delete).

El haber trabajado en un proyecto de monetización en el ámbito bancario y de *ecommerce* en una empresa de nivel internacional realizó un cambio monumental en la automatización de los procesos mediante la programación y sus diferentes arquitecturas. Como se define, se escala y se implementa una base de datos a nivel empresarial, además de poder conocer la demanda del lenguaje estructurado de consultas y los lenguajes de alto nivel en el mercado, la importancia de tener conocimiento sobre diferentes **frameworks**, él como se puede manejar información espacial no solo mediante sistemas de información sino también por servicios. El presente trabajo complementa los conocimientos previos adquiridos durante la experiencia académica. El análisis de la información y resolución de problemas manejado en este estudio, resulta de mayor complejidad desde el punto de vista laboral y resolución de problemas a un nuevo nivel.

5. Referencias

- ‘¿Cuál es la diferencia entre las bases de datos relacionales y las no relacionales?’ © 2023, Amazon Web Services, Inc. o sus filiales. Consultado el 10/06/2024, URL: https://aws.amazon.com/es/compare/the-difference-between-relational-and-non-relational-databases/
- ‘Comparación entre la arquitectura monolítica y la arquitectura de microservicios’, © 2024 Atlassian. Consultado el 10/06/2024, URL: https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith
- ‘¿Qué son los microservicios?’, © 2023, Amazon Web Services, Inc. o sus filiales. Consultado el 10/06/2024, URL: https://aws.amazon.com/es/microservices/#:~:text=Los%20microservicios%20fomentan%20una%20organizaci%C3%B3n,tiempos%20del%20ciclo%20de%20desarrollo
- ‘JSON’ ©1998–2024, mdn web docs. Consultado el 18/09/2024, URL: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/JSON