



## INFORME DE GUÍA PRÁCTICA

### I. PORTADA

Tema:	Sistemas Distribuidos
Unidad de Organización Curricular:	PROFESIONAL
Nivel y Paralelo:	6to – “A”
Alumnos participantes:	Carrasco Paredes Kevin Andres Chimborazo Guaman William Andres Quishpe Lopez Luis Alexander Sailema Gavilanez Ismael Alexander
Asignatura:	Aplicaciones Distribuidas
Docente:	Ing. Jose Ruben, Mg.

### II. INFORME DE GUÍA PRÁCTICA

#### 2.1 Objetivos

##### **General:**

Desarrollar un sistema distribuido que garantice la escalabilidad, disponibilidad y confiabilidad de la información.

##### **Específicos:**

- Investigar y aplicar los conceptos sobre sistemas distribuidos mediante el diseño de una arquitectura basada en microservicios que sea escalable y tolerante a fallos.
- Implementar la arquitectura en microservicios el cual sea escalable y de alta disponibilidad para cada uno de los servidores web y la base de datos usando contenedores Docker y replicación de bases de datos.
- Documentar detalladamente el proceso de desarrollo de la arquitectura distribuida, explicando cada una de las etapas desde la configuración hasta la implementación.

#### 2.2 Modalidad

Presencial

#### 2.3 Tiempo de duración

**Presenciales:** 12

**No presenciales:** 0

#### 2.4 Instrucciones

- El trabajo se desarrollará en grupos de 3 o 4 personas.
- La arquitectura por desarrollar se presentará en clase funcionando.
- Se debe desarrollar un informe en el cual se explique a manera de tutorial el desarrollo de la arquitectura. Es importante que se vaya comentando los pasos que se siguió y la función que cumple cada comando. Se debe justificar el uso de los servicios seleccionados.
- Subir el documento en formato PDF al aula virtual en el espacio habilitado para la tarea.

#### 2.5 Listado de equipos, materiales y recursos

Listado de equipos y materiales generales empleados en la guía práctica:

- Inteligencia artificial, TAC
- Internet
- Bases de datos disponibles en la biblioteca virtual de la Universidad.
- Bibliografía de la asignatura.
- Material disponible en el aula virtual de la asignatura.
- Virtual Box - Linux

TAC (Tecnologías para el Aprendizaje y Conocimiento) empleados en la guía práctica:

- ☒ Plataformas educativas



- ☐ Simuladores y laboratorios virtuales
- ☐ Aplicaciones educativas
- ☒ Recursos audiovisuales
- ☐ Gamificación
- ☒ Inteligencia Artificial
- Otros (Especifique): \_\_\_\_\_

## 2.6 Actividades por desarrollar

- Levantar una aplicación Web (Moodle, Wordpress, Joomla, Prestashop, etc.) aplicando la arquitectura distribuida planteada en el documento adjunto. - Se debe realizar un balanceo de carga en los servidores web y alta disponibilidad en los servidores de base de datos. El servidor Web, base de datos y servicios para realizar el balanceo de carga y la alta disponibilidad se deja a su libertad de elección. - Adicionalmente se debe realizar aplicar replicación en los servidores de bases de datos para mantener la disponibilidad y mantenibilidad de la información.

## 2.7 Resultados obtenidos

La arquitectura de la aplicación consta de un front-end, un API Gateway, múltiples microservicios y bases de datos dockerizadas. La comunicación entre el front-end y el API Gateway se realiza mediante REST/HTTP, mientras que los microservicios se comunican entre sí y con las bases de datos utilizando gRPC para interacciones eficientes y de alto rendimiento.

### Componentes

- **Front-End:** Desarrollado con React, Tailwind CSS y PrimeReact para una interfaz responsive y amigable.
- **API Gateway:** Punto de entrada para el front-end, redirige las solicitudes a los microservicios correspondientes mediante REST/HTTP.
- **Microservicios:**
  - **Microservicio Administración:** Gestiona tareas administrativas.
  - **Microservicio Autenticación:** Administra la autenticación y autorización de usuarios.
  - **Microservicio Consultas Médicas:** Maneja las consultas médicas.
- **Bases de Datos:**
  - **Base de Datos Central:** Almacena datos compartidos.
  - **Base de Datos Local (Cuenca):** Almacena datos específicos de la localidad de Cuenca.
  - **Base de Datos Local (Quito):** Almacena datos específicos de la localidad de Quito. Todas las bases de datos están dockerizadas para facilitar la gestión y el despliegue.

### Tecnologías Utilizadas

- **Front-End:** React, Tailwind CSS, PrimeReact.
- **Back-End:** ASP.NET Core para el desarrollo de los microservicios.
- **Comunicación:**
  - REST/HTTP entre el front-end y el API Gateway.
  - gRPC para la comunicación entre microservicios y bases de datos.
- **Bases de Datos:** Dockerizadas para portabilidad y consistencia.

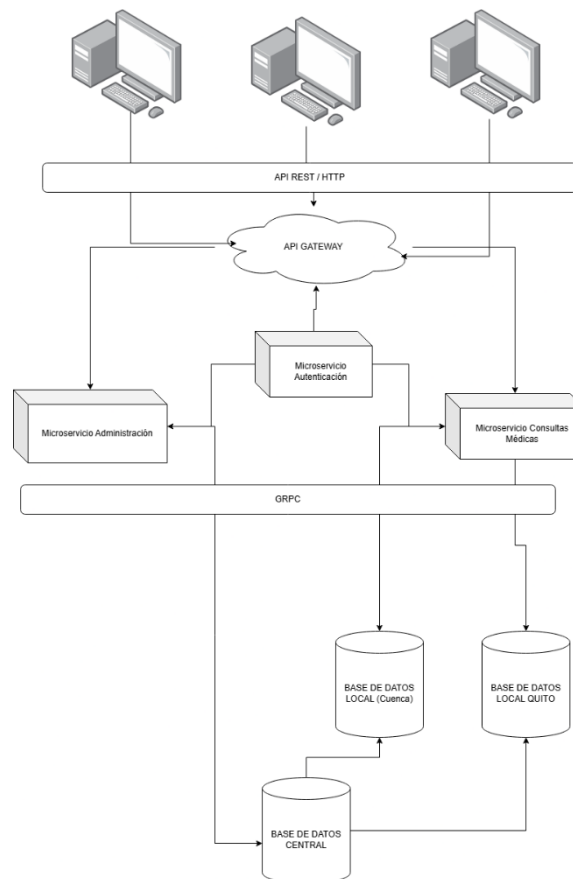
### Implementación

1. **Front-End:** Se desarrolló una interfaz interactiva con React, estilizada con Tailwind CSS y además con componentes de PrimeReact para una experiencia de usuario optimizada.



2. **API Gateway:** Implementado como un punto central de comunicación, recibe las solicitudes del front-end y las enruta a los microservicios correspondientes.
3. **Microservicios:** Cada microservicio se desarrolló con ASP.NET Core, asegurando independencia y escalabilidad. La comunicación entre ellos se optimizó con gRPC.
4. **Bases de Datos:** Se dockerizaron para garantizar un entorno uniforme y facilitar el despliegue en diferentes entornos.

**Esquema de la estructura del proyecto:**



*Ilustración 1 Esquema de la arquitectura del proyecto*

**Docker Compose Administración**

Se configuró un entorno de desarrollo para el microservicio de Administración utilizando Docker Compose, implementando una base de datos MariaDB.



```
1 version: '3.8'
2 ## Docker para subir un servidor de base de datos mariadb de prueba para desarrollar el microservicio
3 services:
4   mariadb:
5     image: mariadb:11
6     container_name: mariadb-dev
7     environment:
8       MARIADB_ROOT_PASSWORD: 200232
9       MARIADB_DATABASE: hospital
10    ports:
11      - "3307:3306"
12    volumes:
13      - mariadb_data:/var/lib/mysql
14
15 volumes:
16   mariadb_data:
17
```

*Ilustración 2 Docker para el api de administración*

Este archivo configura una base de datos MariaDB dockerizada para el microservicio de Administración, creando una base de datos llamada hospital accesible en localhost:3307 con usuario root y contraseña 200232. Los datos persisten mediante un volumen, facilitando el desarrollo y las pruebas.

### **Docker Compose Consultas Médicas**

Se configuró un entorno de desarrollo para el microservicio de Consultas Médicas utilizando Docker Compose con una base de datos MariaDB.

```
1 version: '3.8'
2
3 services:
4   mariadb:
5     image: mariadb:11
6     container_name: mariadb-dev-con
7     environment:
8       MARIADB_ROOT_PASSWORD: 200232
9       MARIADB_DATABASE: consultas
10    ports:
11      - "3306:3306"
12    volumes:
13      - mariadb_data:/var/lib/mysql
14
15
16 volumes:
17   mariadb_data:
18
```

*Ilustración 3 Docker Compose para el api de consultas médicas*

Este archivo configura una base de datos MariaDB dockerizada para el microservicio de Consultas Médicas, creando una base de datos llamada consultas accesibles en localhost:3306 con usuario root y contraseña 200232. Los datos persisten mediante un volumen, facilitando el desarrollo y las pruebas.

### **Dependencias necesarias para el desarrollo del proyecto:**

Dado que este proyecto fue desarrollado con .net y ASP Core se usarón las siguientes dependencias.



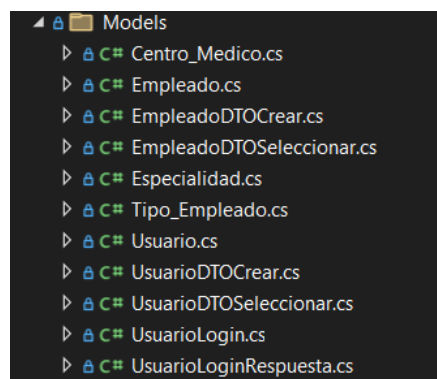
**UNIVERSIDAD TÉCNICA DE AMBATO**  
**FACULTAD DE INGENIERÍA EN SISTEMAS ELECTRÓNICA E INDUSTRIAL**  
**CARRERA DE SOFTWARE**  
**CICLO ACADÉMICO: MARZO – JULIO 2025**



Paquetes de nivel superior (11)		
	<b>Google.Protobuf</b> por Google Inc. C# runtime library for Protocol Buffers - Google's data interchange format.	3.30.2
	<b>Grpc.AspNetCore</b> por The gRPC Authors gRPC meta-package for ASP.NET Core	2.70.0
	<b>Grpc.Net.Client</b> por The gRPC Authors .NET client for gRPC	2.70.0
	<b>Grpc.Tools</b> por The gRPC Authors gRPC and Protocol Buffer compiler for C# projects	2.71.0
	<b>Microsoft.AspNetCore.Authentication.JwtBearer</b> por Microsoft ASP.NET Core middleware that enables an application to receive an OpenID Connect bearer token.	8.0.15 9.0.4
	<b>Microsoft.EntityFrameworkCore</b> por Microsoft Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, Postgre...	8.0.15 9.0.4
	<b>Microsoft.EntityFrameworkCore.Design</b> por Microsoft Shared design-time components for Entity Framework Core tools.	8.0.15 9.0.4
	<b>Microsoft.EntityFrameworkCore.Tools</b> por Microsoft Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.	8.0.15 9.0.4
	<b>Microsoft.VisualStudio.Web.CodeGeneration.Design</b> por Microsoft Code Generation tool for ASP.NET Core. Contains the dotnet-aspnet-codegenerator command used for generating controllers and views.	8.0.7 9.0.0
	<b>Pomelo.EntityFrameworkCore.MySql</b> por Laurents Meyer, Caleb Lloyd, Yuko Zheng Pomelo's MySQL database provider for Entity Framework Core.	8.0.0 8.0.3
	<b>Swashbuckle.AspNetCore</b> por domaindrivendev Swagger tools for documenting APIs built on ASP.NET Core	8.1.1

### Creación de microservicio de administración

Una vez se tienen en funcionamiento los contenedores de las bases de datos se crea el microservicio de administración como proyecto .NET Core, una vez se creó el proyecto se empieza por crear el modelo de cada entidad definida en la base de datos, esto permitirá primero mapear en la base de datos cada entidad y utilizar los mismos en los controladores que se generan en el proyecto.



*Ilustración 4 Modelos del microservicio de administración*

Cada modelo contiene los atributos respectivos de cada entidad de la base de datos.



```
namespace Microservicio_Administracion.Models
{
    public class Empleado
    {
        public int Id { get; set; }

        public int centro_medicoID { get; set; }

        public int tipo_empleadoID { get; set; }

        public string nombre { get; set; }

        public string cedula { get; set; }

        public int especialidadID { get; set; }
        20 referencias
        public string telefono { get; set; }
        20 referencias
        public string email { get; set; }
        15 referencias
        public double salario { get; set; }

        58 referencias
        public Centro_Medico Centro_Medico { get; set; }
        34 referencias
        public Tipo_Empleado Tipo_Empleado { get; set; }
        34 referencias
        public Especialidad Especialidad { get; set; }
    }
}
```

*Ilustración 5 Modelo de empelado del microservicio de administración*

Una vez se definen los modelos, se crea el contexto de los datos el cual servirá para mapear los modelos en la base de datos junto con sus relaciones, de la misma forma este servirá para crear los controladores con el scaffolding que proporciona .NET y los métodos CRUD para la comunicación GRPC a través de los servicios.

```
using Microservicio_Administracion.Models;
using Microsoft.EntityFrameworkCore;

namespace Microservicio_Administracion.Data
{
    23 referencias
    public class AppDbContext : DbContext
    {
        0 referencias
        public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) {
        }

        15 referencias
        public DbSet<Usuario> Usuarios { get; set; }
        20 referencias
        public DbSet<Empleado> Empleados { get; set; }
        16 referencias
        public DbSet<Especialidad> Especialidades { get; set; }
        16 referencias
        public DbSet<Tipo_Empleado> Tipos_Empleados { get; set; }
        16 referencias
        public DbSet<Centro_Medico> Centros_Medicos { get; set; }

        0 referencias
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Usuario>()
                .HasOne(u => u.empleado)
                .WithOne()
                .HasForeignKey<Usuario>(u => u.empleadoId);
            modelBuilder.Entity<Empleado>()
                .HasOne(e => e.Centro_Medico)
                .WithMany()
                .HasForeignKey(e => e.centro_medicoID);

            modelBuilder.Entity<Empleado>()
                .HasOne(e => e.Especialidad)
                .WithMany()
                .HasForeignKey(e => e.especialidadID);

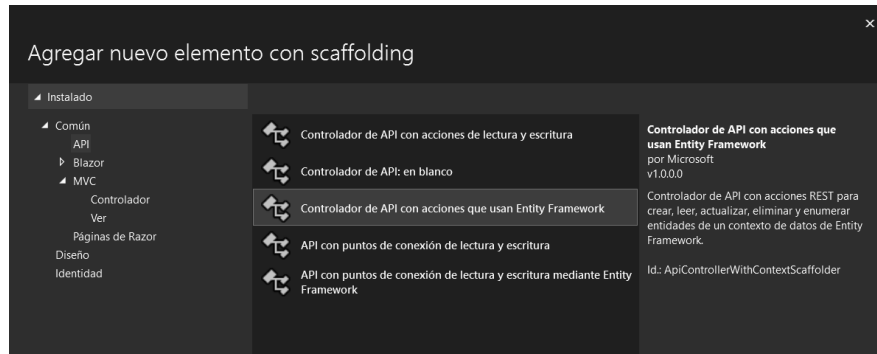
            modelBuilder.Entity<Empleado>()
                .HasOne(e => e.Tipo_Empleado)
                .WithMany()
                .HasForeignKey(e => e.tipo_empleadoID);
        }
    }
}
```

*Ilustración 6 DataContext del microservicio de administración*

Ya se tienen creados los modelos y el contexto de datos se utiliza el scaffolding de .NET para crear los controladores, se tienen que crear a partir de la carpeta de controladores, escogiendo la



opción de Api y con la opción de **Controlador de Api con acciones que usan entity Framework**, se escoge esta opción porque es la que trabaja con el scaffolding. Ya se tienen creados los modelos y el contexto de datos se utiliza el scaffolding de .NET para crear los controladores, se tienen que crear a partir de la carpeta de controladores, escogiendo la opción de Api y con la opción de **Controlador de Api con acciones que usan entity Framework**, se escoge esta opción porque es la que trabaja con el scaffolding.



*Ilustración 7 Uso de scaffolding para la creación del controlador*

Se escoge el modelo en el que se va a basar el controlador en este caso se selecciona el modelo de empleado.



*Ilustración 8 Configuración del modelo para el desarrollo*

Con esto se desarrolla la gestión de los métodos que se tienen en el api rest para la comunicación con la base de datos (Get, Post, Put, Delete)





```
namespace Microservicio_Administracion.Controllers
{
    [Route("api/[controller]")]
    [Authorize(Policy = "TipoEmpleadoPolitica")]
    [ApiController]
    public class EmpleadosController : ControllerBase
    {
        private readonly ApplicationDbContext _context;

        public EmpleadosController(ApplicationDbContext context)
        {
            _context = context;
        }

        // GET: api/Empleados
        [HttpGet]
        public async Task<ActionResult<IEnumerable<EmpleadoDTOSeleccionar>>> GetEmpleados()
        {
            var empleados = await _context.Empleados
                .Include(e => e.Centro_Medico)
                .Include(e => e.Especialidad)
                .Include(e => e.Tipo_Empleado)
                .ToListAsync();

            var empleadosdto = empleados.Select(
                e => new EmpleadoDTOSeleccionar
                {
                    Id = e.Id,
                    cedula = e.cedula,
                    email = e.email,
                    nombre = e.nombre,
                    telefono = e.telefono,
                    Centro_Medico = new Centro_Medico {
                        nombre = e.Centro_Medico.nombre,
                        Id = e.Centro_Medico.Id,
                        ciudad = e.Centro_Medico.ciudad,
                        direccion = e.Centro_Medico.direccion
                    },
                    Especialidad = new Especialidad
                }
            );
        }
    }
}
```

*Ilustración 9 Ejemplo de desarrollo de controlador*

## Comunicación a la base de datos

En appSettings.json se definen los parámetros para la conexión con la base de datos y además los parámetros para la autenticación y autorización con JWT (Json Web Token).

```
schema: https://json.schemastore.org/appsettings.json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Jwt": {
    "Secret": "c089f0752f3e8ba92d7bca8cfd49bbd1c16af58289c7ce64b98867217d871a76",
    "Audience": "user",
    "Issuer": "Microservicio-Autenticacion",
    "TiempoExpira": 3
  }
}
```

*Ilustración 10 Configuración para la base de datos y para el jwt en appSettings*

## Program.cs de administración.

El archivo Program.cs configura el microservicio de Administración desarrollado en ASP.NET Core. Define una API con soporte para gRPC y REST, utilizando MariaDB como base de datos a través de Entity Framework Core, conectándose mediante una cadena de conexión (DefaultConnection). Implementa autenticación y autorización con JWT, validando tokens con parámetros como emisor, audiencia y una clave secreta, y establece una política de autorización que restringe acceso a usuarios con roles "Administrador" o "Doctor". Configura Swagger para documentar la API con soporte para JWT en desarrollo. El servidor Kestrel se ajusta para usar HTTP/2 y escuchar en el puerto 7256 con HTTPS. Al iniciar, aplica migraciones a la base de datos y, si está vacía, inserta datos iniciales como una especialidad ("Sin Especialidad"), un tipo de empleado ("Administrador"), un centro médico ("Central" en Quito), un empleado ("admin") y un usuario ("root" con contraseña "1234"). Finalmente, habilita middleware para Swagger en





desarrollo, redirección HTTPS, autorización y mapeo de servicios gRPC (UsuarioServiceImpl y AdministracionServiceImpl) y controladores REST.

```
Microservicio-Administracion

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<AppDbContext>({
    options => {
        options.UseMySQL(connectionString, ServerVersion.AutoDetect(connectionString));
    }
});

builder.Services.AddGrpc();
//dotnet dev-certs https --trust
builder.WebHost.ConfigureKestrel(options =>
{
    options.ListenAnyIP(7256, listenOptions =>
    {
        listenOptions.UseHttps(); // ? Usa el certificado por defecto o personalizado
        listenOptions.Protocols = HttpProtocols.Http2;
    });
});

//JWT
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer(o =>
{
    o.Events = new JwtBearerEvents
    {
        OnAuthenticationFailed = context =>
        {
            Console.WriteLine($"Error de autenticación: {context.Exception}");
            return Task.CompletedTask;
        },
        OnForbidden = context =>
        {
            Console.WriteLine($"Error de autenticación: {context.Response}");
            return Task.CompletedTask;
        },
        OnTokenValidated = context =>
        {
            Console.WriteLine($"Token validado: {context.SecurityToken}");
        }
    }
});
```

*Ilustración 11 Ejemplo de desarrollo de Program.cs*

**Probar mediante swagger mediante los endpoints**

Centro_Medico			^
GET	/api/Centro_Medico		🔒
POST	/api/Centro_Medico		🔒
GET	/api/Centro_Medico/{id}		🔒
PUT	/api/Centro_Medico/{id}		🔒
DELETE	/api/Centro_Medico/{id}		🔒
Empleados			^
GET	/api/Empleados		🔒
POST	/api/Empleados		🔒
GET	/api/Empleados/{id}		🔒
PUT	/api/Empleados/{id}		🔒
DELETE	/api/Empleados/{id}		🔒
Especialidades			^
GET	/api/Especialidades		🔒
POST	/api/Especialidades		🔒
GET	/api/Especialidades/{id}		🔒
PUT	/api/Especialidades/{id}		🔒

*Ilustración 12 Endpoints del microservicio de administracion*



**UNIVERSIDAD TÉCNICA DE AMBATO**  
**FACULTAD DE INGENIERÍA EN SISTEMAS ELECTRÓNICA E INDUSTRIAL**  
**CARRERA DE SOFTWARE**  
**CICLO ACADÉMICO: MARZO – JULIO 2025**



Tipo_Empleado	
GET	/api/Tipo_Empleado
POST	/api/Tipo_Empleado
GET	/api/Tipo_Empleado/{id}
PUT	/api/Tipo_Empleado/{id}
DELETE	/api/Tipo_Empleado/{id}
Usuarios	
GET	/api/Usuarios
POST	/api/Usuarios
GET	/api/Usuarios/{id}
PUT	/api/Usuarios/{id}
DELETE	/api/Usuarios/{id}

*Ilustración 13 Continuación de endpoints*

Schemas
Centro_Medico >
Empleado >
EmpleadoDOTCrear >
EmpleadoDOTSeleccionar >
Especialidad >
Tipo_Empleado >
Usuario >
UsuarioDOTCrear >
UsuarioDOTSeleccionar >

*Ilustración 14 Esquemas de swagger*

### Implementación de GRPC en el microservicio de administración

Al terminar la creación del APIREST con http, se crea la carpeta Protos la cual contendrá los archivos .proto y los servicios para que utilizará las acciones CRUD en GRPC, al tener lista la carpeta se crea primero el archivo AdministraciónServiceImpl.cs el cual tendrá la lógica de cada método que se utilizará en GRPC, como se muestra en la ilustración.



**UNIVERSIDAD TÉCNICA DE AMBATO**  
**FACULTAD DE INGENIERÍA EN SISTEMAS ELECTRÓNICA E INDUSTRIAL**  
**CARRERA DE SOFTWARE**  
**CICLO ACADÉMICO: MARZO – JULIO 2025**



```
using Microsoft.EntityFrameworkCore;
using Grpc.Core;
using Microservicio_Administracion.Data;
using Microservicio_Administracion.Administracion;
using Microservicio_Administracion.Models;
using Microsoft.AspNetCore.Authorization;

namespace Microservicio_Administracion.Protos
{
    2 referencias
    public class AdministracionServiceImpl : AdministracionService.AdministracionServiceBase
    {
        private readonly AppDbContext _context;

        0 referencias
        public AdministracionServiceImpl(AppDbContext context)
        {
            _context = context;
        }

        [Authorize]
        3 referencias
        public override async Task<EmpleadoLista> GetAllEmpleado(Administracion.RespuestaVacía request, ServerCallContext context)
        {
            if (request == null)
            {
                throw new RpcException(new Status(StatusCode.InvalidArgument, "El empleado id no puede ser null"));
            }

            var e = await _context.Empleados
                .Include(e => e.Centro_Medico)
                .Include(e => e.Especialidad)
                .Include(e => e.Tipo_Empleado)
                .ToListAsync();

            var empleadosLista = new List<Administracion.Empleado>();

            foreach (var u in e)
            {
                var empleadoFila = new Administracion.Empleado
                {
                    Id = u.Id,
                    Cedula = u.cedula,
                    CentroMedicoID = u.centro_medicoID,
                    Email = u.email,
                    EspecialidadID = u.especialidadID,
                    Nombre = u.nombre,
                }
            }
        }
    }
}
```

*Ilustración 15 Servicios GRPC de administración*

Ya se tiene definido cada método GRPC se crea el archivo AdministraciónService.proto el cual brindará acceso a cada uno de los métodos que se definieron anteriormente.

```
service AdministracionService{

    //especialidades
    rpc GetAllEspecialidades(RespuestaVacía) returns (EspecialidadLista);
    rpc GetEspecialidades(EspecialidadGet) returns (Especialidad);
    rpc PostEspecialidad (EspecialidadPost) returns (Especialidad);
    rpc PutEspecialidad (Especialidad) returns (Especialidad);
    rpc DeleteEspecialidad(EspecialidadGet) returns (RespuestaVacía);
    //tipo empleado
    rpc GetAllTipo_Empleado(RespuestaVacía) returns (Tipo_EmpleadoLista);
    rpc GetTipo_Empleado(Tipo_EmpleadoGet) returns (Tipo_Empleado);
    rpc PostTipo_Empleado (Tipo_EmpleadoPost) returns (Tipo_Empleado);
    rpc PutTipo_Empleado (Tipo_Empleado) returns (Tipo_Empleado);
    rpc DeleteTipo_Empleado(Tipo_EmpleadoGet) returns (RespuestaVacía);
    //centros medicos
    rpc GetAllCentro_Medico(RespuestaVacía) returns (Centro_MedicoLista);
    rpc GetCentro_Medico(Centro_MedicoGet) returns (Centro_Medico);
    rpc PostCentro_Medico (Centro_MedicoPost) returns (Centro_Medico);
    rpc PutCentro_Medico (Centro_Medico) returns (Centro_Medico);
    rpc DeleteCentro_Medico(Centro_MedicoGet) returns (RespuestaVacía);
    //empleados
    rpc GetEmpleado (EmpleadoGet) returns (Empleado);
    rpc GetAllEmpleado(RespuestaVacía) returns (EmpleadoLista);
    rpc PostEmpleado (EmpleadoPost) returns (Empleado);
    rpc PutEmpleado (EmpleadoPut) returns (Empleado);
    rpc DeleteEmpleado(EmpleadoGet) returns (RespuestaVacía);
    rpc GetAllEmpleadoByEspecialidad(EspecialidadGet) returns (EmpleadoLista);
    rpc GetAllEmpleadoByCentroMedico(Centro_MedicoGet) returns (EmpleadoLista);
}
```

*Ilustración 16 Archivo .proto del microservicio de Administración*

Al tener ambos archivos correctamente estructurados se modifica el archivo program.cs para que permita la comunicación a través de GRPS.



```
builder.Services.AddGrpc();  
//dotnet dev-certs https --trust  
builder.WebHost.ConfigureKestrel(options =>  
{  
    options.ListenAnyIP(7256, listenOptions =>  
    {  
        listenOptions.UseHttps(); // ? Usa el certificado por defecto o personalizado  
        listenOptions.Protocols = HttpProtocols.Http2;  
    });  
});
```

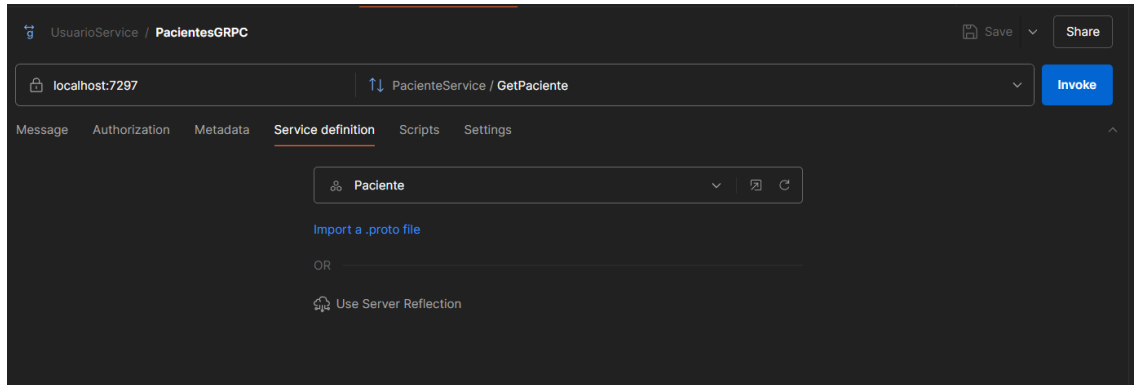
*Ilustración 17 Modificación de program.cs para la comunicación con GRPC*

Finalmente, con la variable app se invocan a los métodos de cada entidad proto en el program.cs

```
app.MapGrpcService<UsuarioServiceImpl>();  
  
app.MapGrpcService<AdministracionServiceImpl>();
```

*Ilustración 18 Invocación de los métodos GRPC en el program.cs*

Se crea una solicitud GRPC en el software postman probando el archivo AdministracionService.proto el cual mostrará los métodos que se programaron en el mismo para enviar las solicitudes.



*Ilustración 19 Prueba de los métodos GRPC en Postman*



## Creación de microservicios Consultas Médicas

### Creación del contenedor para Consultas médicas

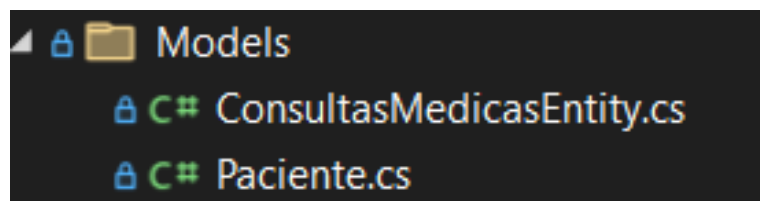
Código de implementación para el contenedor

```
1  version: '3.8'
2
3  services:
4    mariadb:
5      image: mariadb:11
6      container_name: mariadb-dev-con
7      environment:
8        MARIADB_ROOT_PASSWORD: 200232
9        MARIADB_DATABASE: consultas
10     ports:
11       - "3306:3306"
12     volumes:
13       - mariadb_data:/var/lib/mysql
14
15
16  volumes:
17    mariadb_data:
18
```

*Ilustración 20. Modelo del contenedor para Consultas Médicas*

### Creación de la APIRest en ASP.NET core

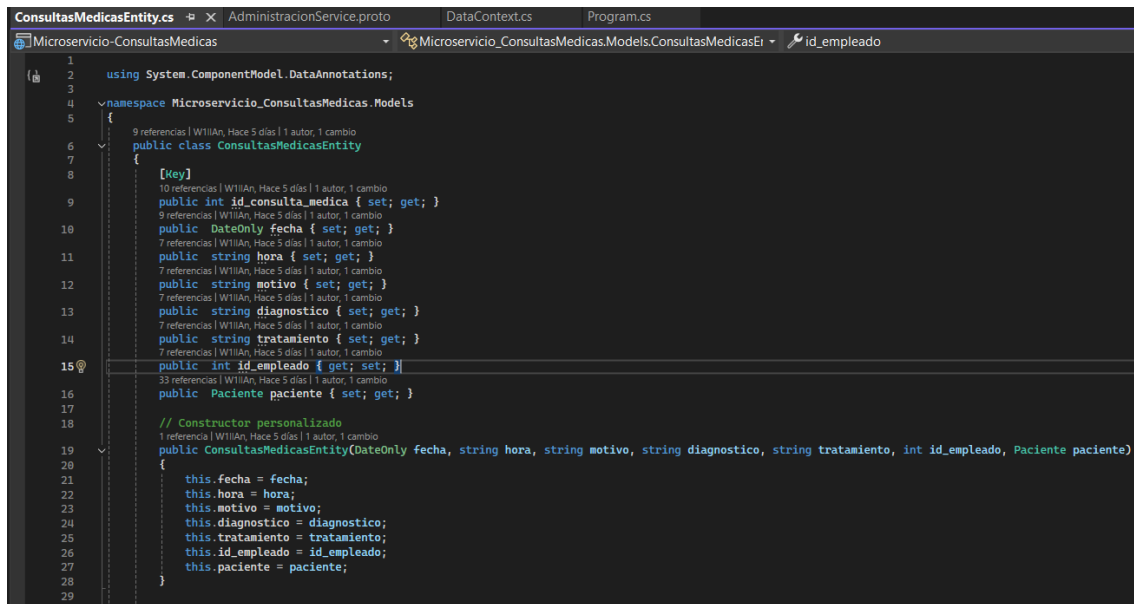
Una vez creado el contenedor para el microservicio de consultar médicas se procede a crear el proyecto para la APIRest en ASP.NET core. Inicialmente se implementa los modelos para consultas médicas esto permitirá mapear la base de datos con cada entidad y se utiliza para la creación de los controladores de cada entidad



*Ilustración 21. Entidades Consultas Médicas*



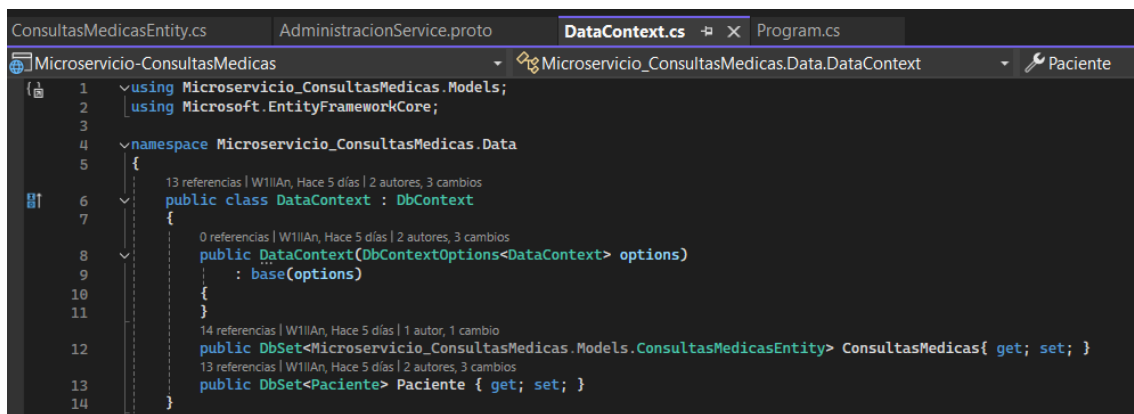
Cada modelo contendrá los atributos necesarios para el manejo de consultas médicas.



```
1 using System.ComponentModel.DataAnnotations;
2
3
4 namespace Microservicio_ConsultasMedicas.Models
5 {
6     9 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
7     public class ConsultasMedicasEntity
8     {
9         [Key]
10         10 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
11         public int id_consulta_medica { get; set; }
12         9 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
13         public DateOnly fecha { set; get; }
14         7 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
15         public string hora { set; get; }
16         7 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
17         public string motivo { set; get; }
18         7 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
19         public string diagnostico { set; get; }
20         7 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
21         public string tratamiento { set; get; }
22         7 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
23         public int id_empleado { get; set; }
24         33 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
25         public Paciente paciente { set; get; }
26
27         // Constructor personalizado
28         1 referencia | W111An, Hace 5 días | 1 autor, 1 cambio
29         public ConsultasMedicasEntity(DateOnly fecha, string hora, string motivo, string diagnostico, string tratamiento, int id_empleado, Paciente paciente)
30         {
31             this.fecha = fecha;
32             this.hora = hora;
33             this.motivo = motivo;
34             this.diagnostico = diagnostico;
35             this.tratamiento = tratamiento;
36             this.id_empleado = id_empleado;
37             this.paciente = paciente;
38         }
39     }
40 }
```

*Ilustración 22. Atributos de la entidad Consultas Médicas.*

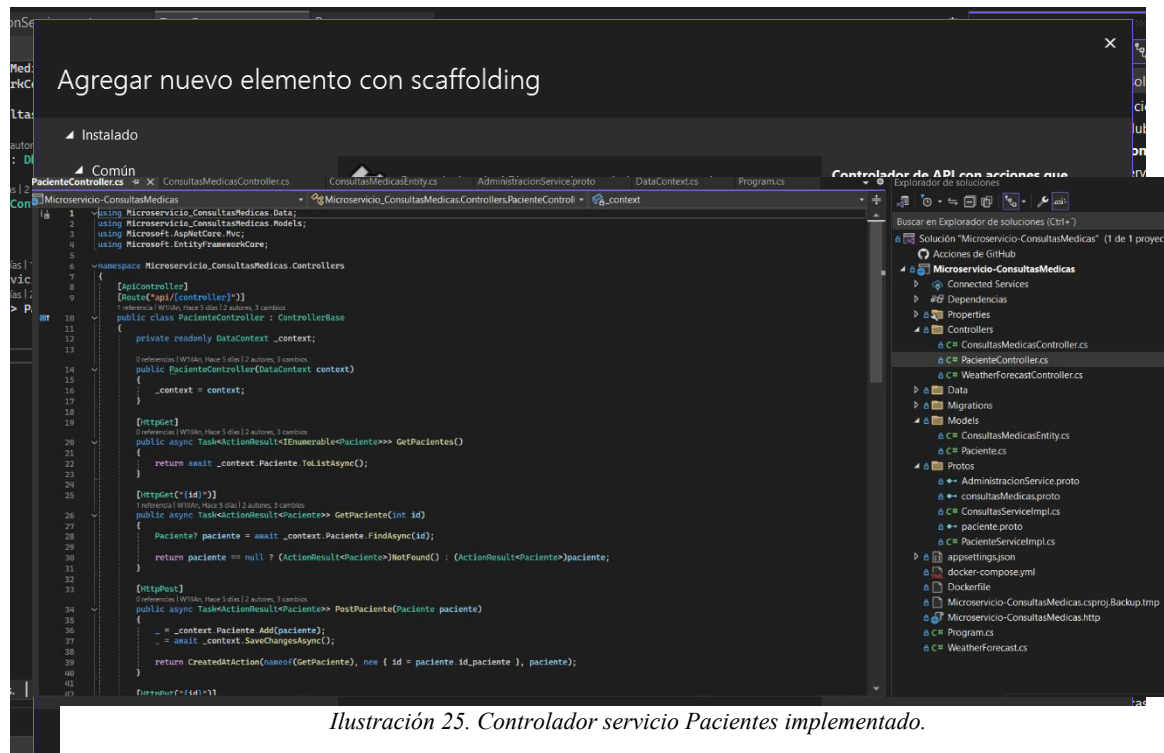
Una vez implementado las entidades necesarias para el manejo del microservicio consultas médicas se crea el contexto de los datos “DataContext”, este método servirá para la creación de los modelos en la base de datos además de crear los controladores y métodos CRUD para el manejo de la comunicación del GRPC a través de los servicios.



```
1 using Microservicio_ConsultasMedicas.Models;
2 using Microsoft.EntityFrameworkCore;
3
4 namespace Microservicio_ConsultasMedicas.Data
5 {
6     13 referencias | W111An, Hace 5 días | 2 autores, 3 cambios
7     public class DataContext : DbContext
8     {
9         0 referencias | W111An, Hace 5 días | 2 autores, 3 cambios
10         public DataContext(DbContextOptions<DataContext> options)
11             : base(options)
12         {
13         }
14         14 referencias | W111An, Hace 5 días | 1 autor, 1 cambio
15         public DbSet<Microservicio_ConsultasMedicas.Models.ConsultasMedicasEntity> ConsultasMedicas { get; set; }
16         13 referencias | W111An, Hace 5 días | 2 autores, 3 cambios
17         public DbSet<Paciente> Paciente { get; set; }
18     }
19 }
```

*Ilustración 23. DataContext microservicio Consultas médicas*

Una vez implementado el DataContext se procede a crear los controladores con ayuda de la función Scaffolding, como se esta trabajando con el modelo MVC este se debe crear en la carpeta controladores eligiendo Api, seguido de Controlador de API con acciones que usan Entity Framework.



*Ilustración 25. Controlador servicio Pacientes implementado.*

*Ilustración 24. Uso de scaffolding para la creación de controlador*

Esta implementación de controlador es esencial para la comunicación de la API con la base de datos, además de manejar los distintos métodos como: GET, PUT, POST, DELET. Este proceso se realiza por cada entidad.

Se configura la comunicación con la base de datos en el archivo appsettings.json, además de la comunicación de con los otros microservicios.

```
WebApplication app = builder.Build();

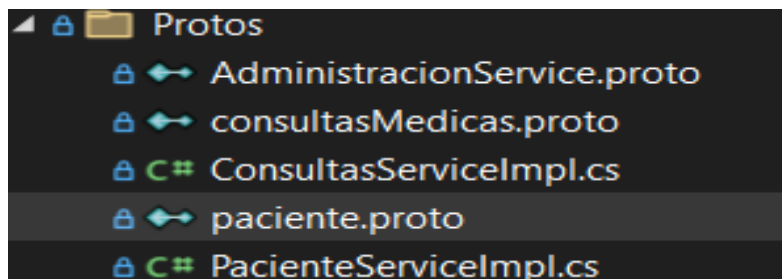
app.MapGrpcService<PacienteServiceImpl>();
app.MapGrpcService<ConsultasServiceImpl>();
```

### Implementación de los métodos GRCP en el microservicio de Consultas médicas





Al terminar la creación de APIRest a continuación se crea la comunicación de GRCP creando la carpeta Protos, el cual servirá para la creación de los métodos CRUD o necesarios en la comunicación de GRCP.



*Ilustración 26. Documentos necesarios para el manejo de comunicación GRCP*

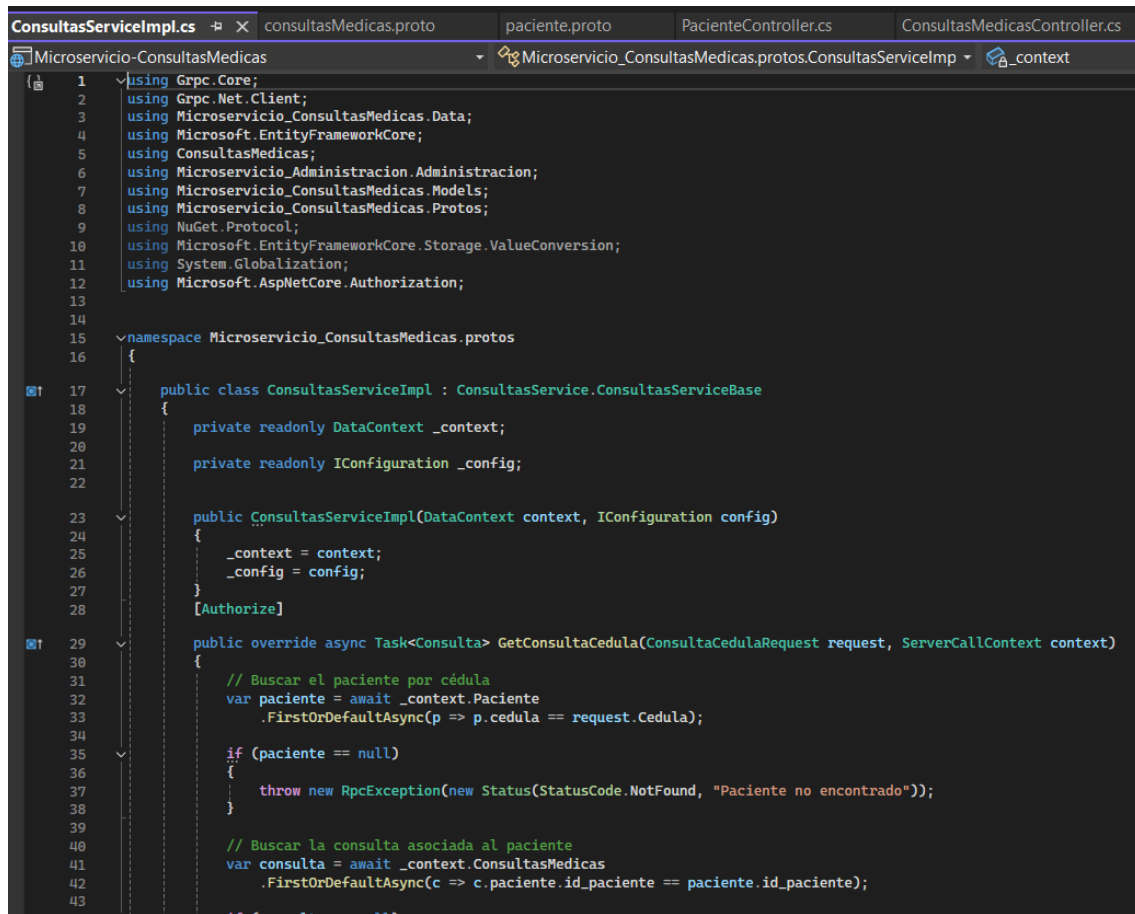
Una vez creada la carpeta se procede a crear los archivos consultasMedicas.proto, pacientes.proto y AdministraciónService.proto.

```
consultasMedicas.proto  paciente.proto  X  PacienteController.cs  C
1  syntax = "proto3";
2
3  option csharp_namespace = "Microservicio_ConsultasMedicas.Protos";
4
5  package paciente;
6
7  message PacienteModel {
8      int32 id_paciente = 1;
9      string nombre = 2;
10     string cedula = 3;
11     string fecha_nacimiento = 4;
12     string telefono = 5;
13     string direccion = 6;
14 }
15
16 message GetPacienteRequest {
17     int32 id_paciente = 1;
18 }
19
20 message GetPacienteResponse {
21     PacienteModel paciente = 1;
22 }
23
24 message GetPacienteListaResponse {
25     repeated PacienteModel pacientes = 1;
26 }
27
28 message CrearPacienteRequest {
29     PacienteModel paciente = 1;
30 }
31
32 message CrearPacienteResponse {
33     PacienteModel paciente = 1;
34 }
35
36 // Nuevos mensajes para Actualizar y Eliminar
37 message ActualizarPacienteRequest {
38     PacienteModel paciente = 1;
39 }
40
41 message ActualizarPacienteResponse {
42     PacienteModel paciente = 1;
43 }
44
45 message EliminarPacienteRequest {
46     int32 id_paciente = 1;
47 }
```

*Ilustración 27. Ejemplo de Proto de Pacientes*



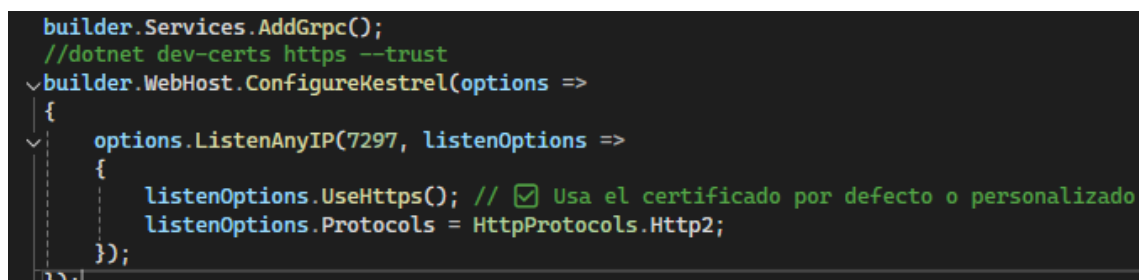
Por siguiente se crean los archivos .cs los cuales tendrán la lógica de los métodos proto.



```
1 using Grpc.Core;
2 using Grpc.Net.Client;
3 using Microservicio_ConsultasMedicas.Data;
4 using Microsoft.EntityFrameworkCore;
5 using ConsultasMedicas;
6 using Microservicio_Administracion.Administracion;
7 using Microservicio_ConsultasMedicas.Models;
8 using Microservicio_ConsultasMedicas.Protos;
9 using NuGet.Protocol;
10 using Microsoft.EntityFrameworkCore.Storage.ValueConversion;
11 using System.Globalization;
12 using Microsoft.AspNetCore.Authorization;
13
14
15 namespace Microservicio_ConsultasMedicas.protos
16 {
17     public class ConsultasServiceImpl : ConsultasService.ConsultasServiceBase
18     {
19         private readonly DataContext _context;
20         private readonly IConfiguration _config;
21
22         public ConsultasServiceImpl(DataContext context, IConfiguration config)
23         {
24             _context = context;
25             _config = config;
26         }
27         [Authorize]
28
29         public override async Task<Consulta> GetConsultaCedula(ConsultaCedulaRequest request, ServerCallContext context)
30         {
31             // Buscar el paciente por cédula
32             var paciente = await _context.Paciente
33                 .FirstOrDefaultAsync(p => p.cedula == request.Cedula);
34
35             if (paciente == null)
36             {
37                 throw new RpcException(new Status(StatusCode.NotFound, "Paciente no encontrado"));
38             }
39
40             // Buscar la consulta asociada al paciente
41             var consulta = await _context.ConsultasMedicas
42                 .FirstOrDefaultAsync(c => c.paciente.id_paciente == paciente.id_paciente);
43             if (consulta == null)
```

Ilustración 28. Implementación del .cs maneja lógica Proto

Al tener los archivos configurados correctamente de los servicios que posee GRPC se procede a configurar el archivo program.cs que ayudará a la comunicación entre microservicio.



```
builder.Services.AddGrpc();
//dotnet dev-certs https --trust
builder.WebHost.ConfigureKestrel(options =>
{
    options.ListenAnyIP(7297, listenOptions =>
    {
        listenOptions.UseHttps(); // Usa el certificado por defecto o personalizado
        listenOptions.Protocols = HttpProtocols.Http2;
    });
});
```

Ilustración 29. Comunicación entre microservicios GRPC

Finalmente, con la variable app. Se invocan los servicios de GRPC implementados.

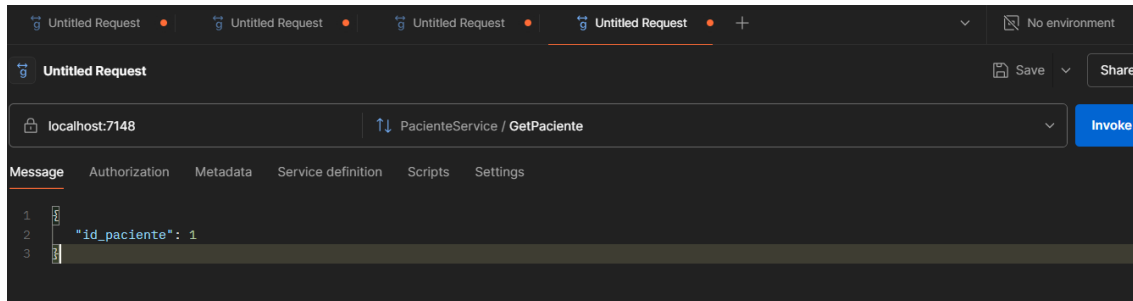
```
WebApplication app = builder.Build();

app.MapGrpcService<PacienteServiceImpl>();
app.MapGrpcService<ConsultasServiceImpl>();
```

Ilustración 30. Invocación de los métodos GRPC.



## Verificación de funcionamiento de microservicio Consultas médicas con GRCP



*Ilustración 31. Microservicios Consultas médicas GRCP*

### JWT

El microservicio de autenticación verifica las credenciales de los usuarios que intentan iniciar sesión. Si son correctas, genera y devuelve un token JWT que permite identificarlos de forma segura en el resto del sistema.

El JWT incluye información esencial del usuario:

- ID del usuario
- Nombre de usuario
- Correo electrónico
- Tipo de empleado (rol)
- Ciudad del centro médico

```
2 referencias
public string Create(Usuario usuario)
{
    string secretKey = configuration["Jwt:Secret"];
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new[] {
            new Claim(JwtRegisteredClaimNames.Sub, usuario.Id.ToString()),
            new Claim(JwtRegisteredClaimNames.UniqueName, usuario.NombreUsuario.ToString()),
            new Claim(JwtRegisteredClaimNames.Email, usuario.Empleado.Email.ToString()),
            new Claim("TipoEmpleado", usuario.Empleado.TipoEmpleado.Tipo.ToString()),
            new Claim("CentroMedico", usuario.Empleado.CentroMedico.Ciudad.ToString())
        }),
        Expires = DateTime.UtcNow.AddMinutes(configuration.GetValue<int>("Jwt:TiempoExpira")),
        SigningCredentials = credentials,
        Audience = configuration["Jwt:Audience"],
        Issuer = configuration["Jwt:Issuer"]
    };
    var handler = new JwtSecurityTokenHandler();
    var token = handler.CreateToken(tokenDescriptor);
    return token;
}
```

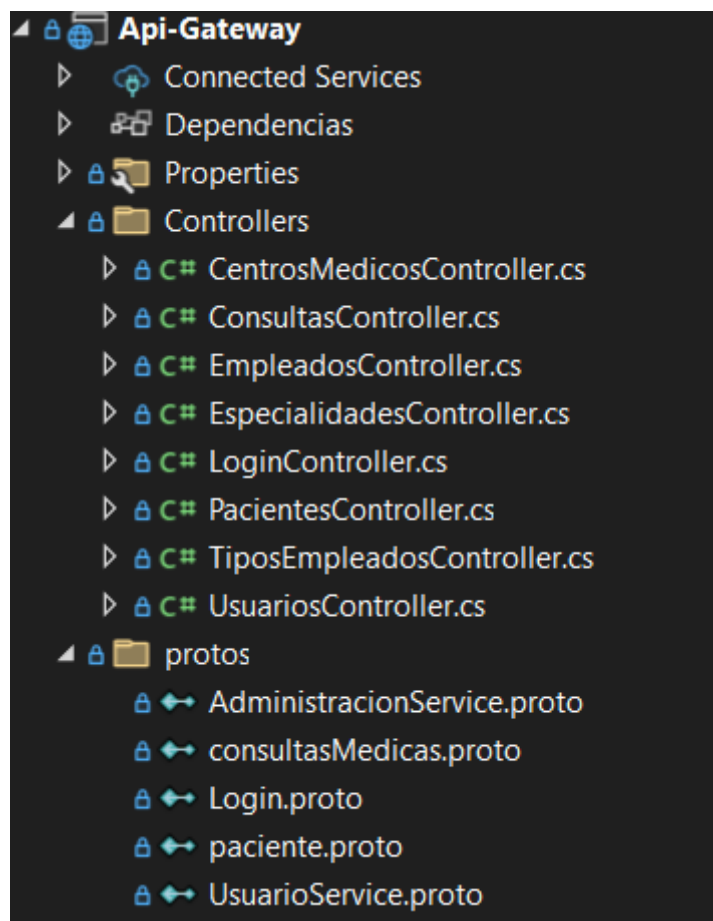
El token se crea usando una clave secreta almacenada en la configuración. Está firmado con un algoritmo seguro y tiene fecha de expiración, emisor e identificador definidos. Se entrega como una cadena de texto. Ese token es el que los clientes usarán en las cabeceras Authorization para hacer solicitudes a otros servicios del sistema.



El API Gateway desarrollado en ASP.NET Core actúa como punto de entrada unificado para los distintos microservicios del sistema. Este componente traduce las solicitudes HTTP RESTful en llamadas gRPC hacia los microservicios especializados, permitiendo una comunicación eficiente y organizada.

### **Integración con Microservicios**

El API Gateway encapsula y expone mediante endpoints REST las funcionalidades definidas en los siguientes servicios gRPC. Cada uno de estos .proto define los métodos gRPC que luego son consumidos internamente por el Gateway, mediante clientes generados con gRPC y configurados dinámicamente desde appsettings.json.



### **Resolución dinámica de servicios**

El Gateway utiliza un sistema de enrutamiento basado en el payload del token JWT:

- Extrae del header Authorization datos como el rol del usuario y el centro médico asociado.
- Si el rol es Administrador, puede redirigir solicitudes hacia múltiples microservicios, recopilando información consolidada desde diferentes sedes (por ejemplo, Cuenca y Guayaquil).
- Si el rol está asociado a una ciudad específica, enruta las solicitudes solo hacia el microservicio correspondiente a dicha sede.

Los endpoints gRPC de los microservicios están parametrizados en appsettings.json, permitiendo flexibilidad en despliegues y escalabilidad futura.



```
"AllowedHosts": "*",  
"grpc": {  
  "centrosMedicos": [ "centroMedico-Cuenca", "centroMedico-Guayaquil"],  
  "administracion": "https://localhost:7256",  
  "autenticacion": "https://localhost:7148",  
  "centroMedico-Cuenca": "https://localhost:7297",  
  "centroMedico-Guayaquil": "https://localhost:7297"  
}
```

## Documentación con Swagger

Para facilitar el consumo desde el frontend y herramientas como Postman o Insomnia, el Gateway utiliza Swagger (Swashbuckle) para documentar y probar de forma interactiva todos los endpoints REST disponibles.

Cada endpoint documentado en Swagger representa una transformación de una operación gRPC a un formato REST estándar, ocultando la complejidad del backend distribuido detrás de una interfaz HTTP simple y centralizada.

**Api-Gateway** 1.0 OAS 3.0  
<https://localhost:7121/swagger/v1/swagger.json>

[Authorize](#)

CentrosMedicos	
GET	/Administracion/CentrosMedicos
POST	/Administracion/CentrosMedicos
GET	/Administracion/CentrosMedicos/{id}
PUT	/Administracion/CentrosMedicos/{id}
DELETE	/Administracion/CentrosMedicos/{id}

Consultas	
GET	/CentroMedico/Consultas
POST	/CentroMedico/Consultas
GET	/CentroMedico/Consultas/Paciente/{cedula}
GET	/CentroMedico/Consultas/Fechas
DELETE	/CentroMedico/Consultas/{id}

## 2.8 Habilidades blandas empleadas en la práctica

- ☐ Liderazgo
- ☐ Trabajo en equipo
- ☐ Comunicación asertiva
- ☐ La empatía
- ☐ Pensamiento crítico
- ☐ Flexibilidad
- ☐ La resolución de conflictos
- ☐ Adaptabilidad
- ☐ Responsabilidad

## 2.9 Conclusiones



Implementación de una arquitectura basada en microservicios (Administración, Autenticación y Consultas Médicas) comunicados mediante gRPC y un API Gateway con REST/HTTP cumple con los requisitos de escalabilidad y tolerancia a fallos. La separación de responsabilidades entre microservicios permite un manejo eficiente de cargas y facilita la recuperación ante fallos individuales.

El uso de contenedores Docker para los microservicios y las bases de datos (MariaDB dockerizadas para cada microservicio) asegura alta disponibilidad y portabilidad. Aunque no se implementó replicación de bases de datos, la configuración actual permite un despliegue rápido y consistente en diferentes entornos, sentando las bases para futuras mejoras en disponibilidad.

Se documentaron todas las etapas del desarrollo, desde la configuración de los microservicios en ASP.NET Core, la integración de JWT para autenticación, hasta la configuración de bases de datos con Docker Compose. Esta documentación detalla el diseño, la implementación y las decisiones técnicas, cumpliendo con el objetivo de proporcionar un registro claro y útil para futuras referencias.

### **2.10 Recomendaciones**

Para mejorar la disponibilidad y tolerancia a fallos, se recomienda configurar la replicación de bases de datos en MariaDB (por ejemplo, un esquema maestro-esclavo). Esto garantizará que los datos estén disponibles incluso si un nodo falla, alineándose con los objetivos de alta disponibilidad.

Aunque Docker Compose es adecuado para desarrollo, en producción se recomienda usar un orquestador como Kubernetes. Esto permitirá gestionar múltiples contenedores, escalar horizontalmente los microservicios según la carga y manejar fallos automáticamente, mejorando la escalabilidad y confiabilidad del sistema.

Se sugiere implementar herramientas de monitoreo (como Prometheus y Grafana) y realizar pruebas de carga para evaluar el rendimiento del sistema bajo alta demanda. Esto ayudará a identificar cuellos de botella y optimizar los microservicios, asegurando que el sistema distribuido cumpla con los requisitos de escalabilidad y disponibilidad en escenarios reales.