DIEGO MADRID – DIEGOMADRID26@GMAIL.COM- ***3107648524***

PRUEBA TÉCNICA

Solución de la prueba

El presente proyecto esta construido mediante una arquitectura a capas:
1. Servicios
2. Negocio
3. Datos
4. Interfaces Comunes y DTO

La arquitectura propuesta incluye una implementación de un repositorio para la entidad utilizando Entity Framework Core y Dapper para interactuar con la base de datos, de forma tal que permite utilizar en paralelo procedimientos almacenados o Entity Framework y posteriormente manipularse con LinQ.

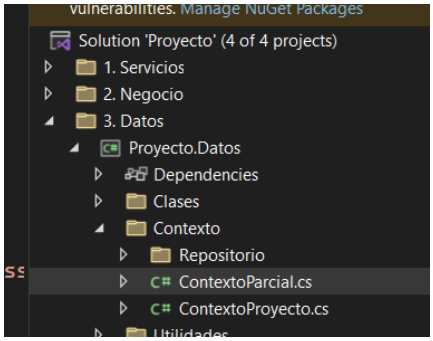
Las tablas paramétricas usan LinQ y la tabla usuario tiene toda su crud y consulta multitabla mediante procedimientos almacenados.

Pasos para la ejecución del proyecto

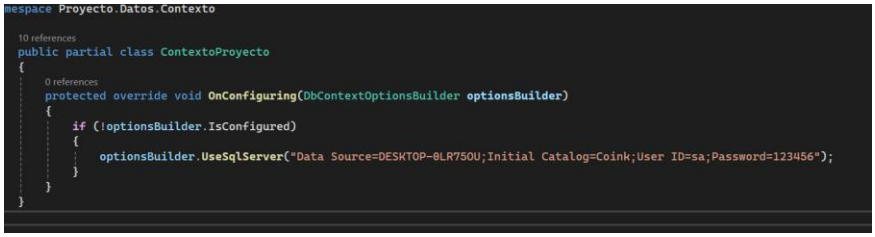
Este desarrollo fue realizado sobre la versión .NET 5.

1. Para poder iniciar correctamente el aplicativo, es necesario ejecutar el script de base de datos sqlServer: **coinkBD**

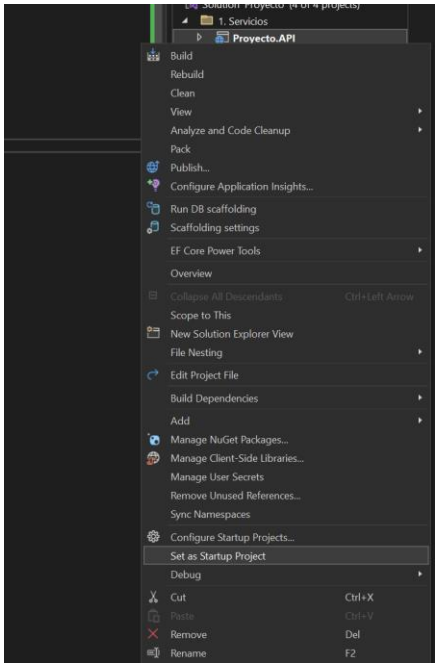
2. Una vez realizado esto, es necesario modificar la cadena de conexión según corresponda al motor de base de datos del equipo, para ello en el proyecto Datos en el contextoParcial.Cs es necesario modificar los valores



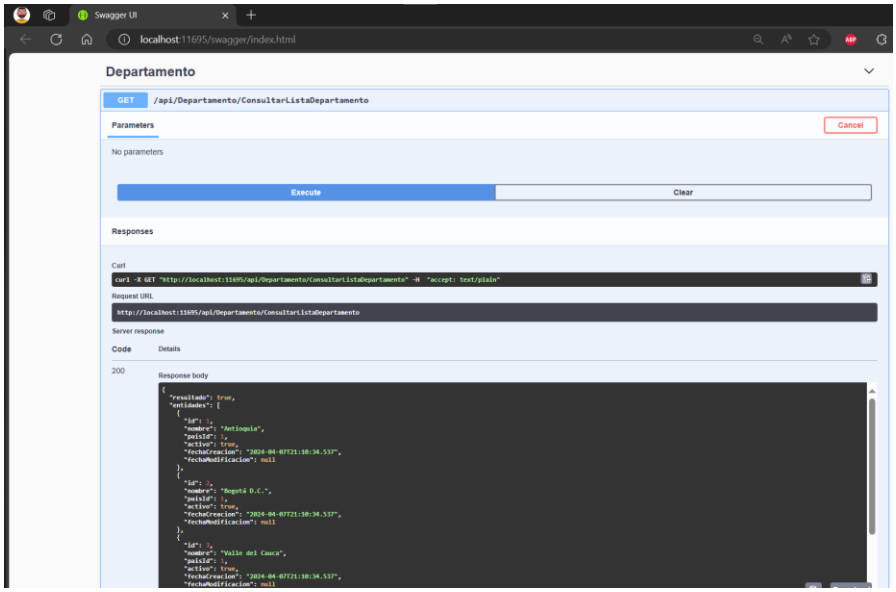
Actualice la cadena de conexión según corresponda a su entorno de base datos.



3. A continuación seleccione el proyecto API y establezca como proyecto de inicio:



4. Una vez realizados estos pasos ya es posible iniciar el proyecto.



Implementación

Estos son algunos patrones, buenas prácticas y principios con las cuales construí el código:

MVC (Modelo-Vista-Controlador):

MVC es un patrón de diseño que separa la aplicación en tres componentes principales: Modelo (representa los datos y la lógica de la aplicación), Vista (presentación de los datos al usuario) y Controlador (gestiona las interacciones del usuario, actualiza el modelo y la vista según sea necesario).

En este código, la estructura de los controladores sigue el patrón MVC, donde cada archivo controller actúa como el controlador que maneja las solicitudes HTTP, el modelo representa la entidad de datos y las vistas se enderezarían en el cliente.

DTO (Data Transfer Object):

DTO es un patrón de diseño que se utiliza para transferir datos entre subsistemas de un software. Ayuda a reducir la cantidad de datos enviados entre clientes y servidores al agrupar los datos relacionados en una sola estructura.Aplicación: Los modelos de la interfaz actúan como DTOs que contienen los datos necesarios para realizar operaciones y hacer un contrato de los parámetros a implementar.

Inyección de Dependencias (DI):

La inyección de dependencias es un patrón de diseño que permite desacoplar las clases dependientes de sus implementaciones concretas, facilitando la prueba unitaria y el mantenimiento. Se utiliza inyección de dependencias a través del constructor en los controller para permitir la sustitución de implementaciones concretas de NegocioAccion.

Uso de Async/Await:

El uso de Async/Await permite realizar operaciones asincrónicas de manera eficiente sin bloquear el hilo principal. Se utilizan métodos asincrónicos en las operaciones CRUD (ConsultarAsync, GuardarAsync, etc.) para evitar bloqueos de hilo en operaciones.

Separación de Responsabilidades:

Este principio establece que cada componente de software debe tener una única responsabilidad y que las distintas responsabilidades deben ser separadas en diferentes componentes. El sigue este principio al dividir las clases en diferentes proyectos y capas según su responsabilidad, como los proyectos de Servicios, Negocio, Datos y las clases separadas para modelos, controladores y utilidades.

Principios Solid:

Los siguientes son algunos ejemplos de la aplicación de los principios solid en el presente proyecto:

1. Principio de Responsabilidad Única

Las clase clases DAL tiene la responsabilidad de interactuar con la base de datos para realizar operaciones CRUD relacionadas con la entidad de usuario. Esta clase se centra en una única responsabilidad: la gestión de la persistencia de datos para la entidad de correspondiente.

2. Principio de Abierto/Cerrado

El diseño del código permite la extensión de funcionalidades mediante la implementación de nuevas clases que heredan de la clase base AccesoComunDAL y sobrescriben sus métodos, como EditarUsuarioAsync, EliminarUsuarioAsync y GuardarUsuarioAsync, sin necesidad de modificar el código existente.

3. Principio de Sustitución de Liskov

Las clases concretas que implementan la interfaz DTO pueden ser sustituidas por sus interfaces en cualquier lugar donde se esperen objetos de esos tipos, sin afectar el comportamiento del programa.

4. Principio de Segregación de la Interfaz

Las interfaces DTO están diseñadas para contener solo las propiedades necesarias para sus respectivos contextos, evitando así la creación de interfaces monolíticas que obliguen a las implementaciones a proporcionar funcionalidades que no necesiten.

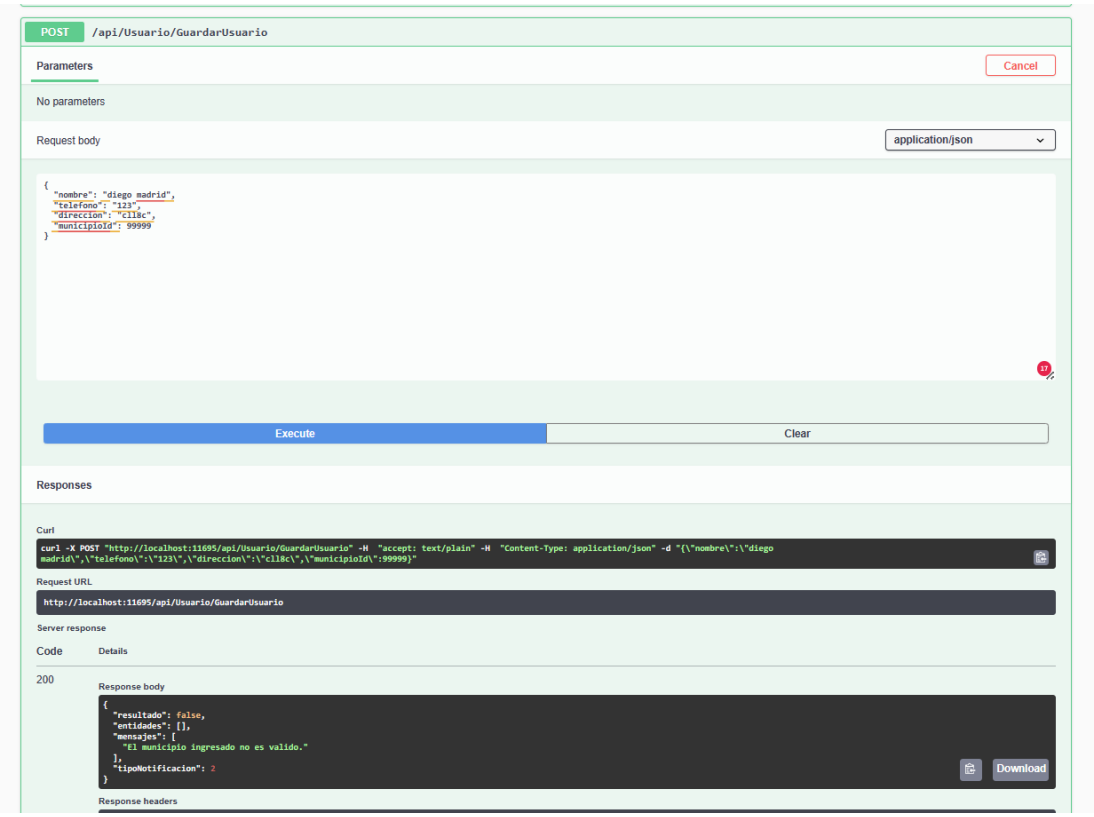
5. Principio de Inversión de Dependencias

La clase UsuarioDAL depende de abstracciones, como las interfaces IUsuarioDTO y IUsuarioConsultaDTO, en lugar de depender de implementaciones concretas. Esto permite que la clase UsuarioDAL sea más flexible y fácilmente intercambiable con otras implementaciones de las interfaces.

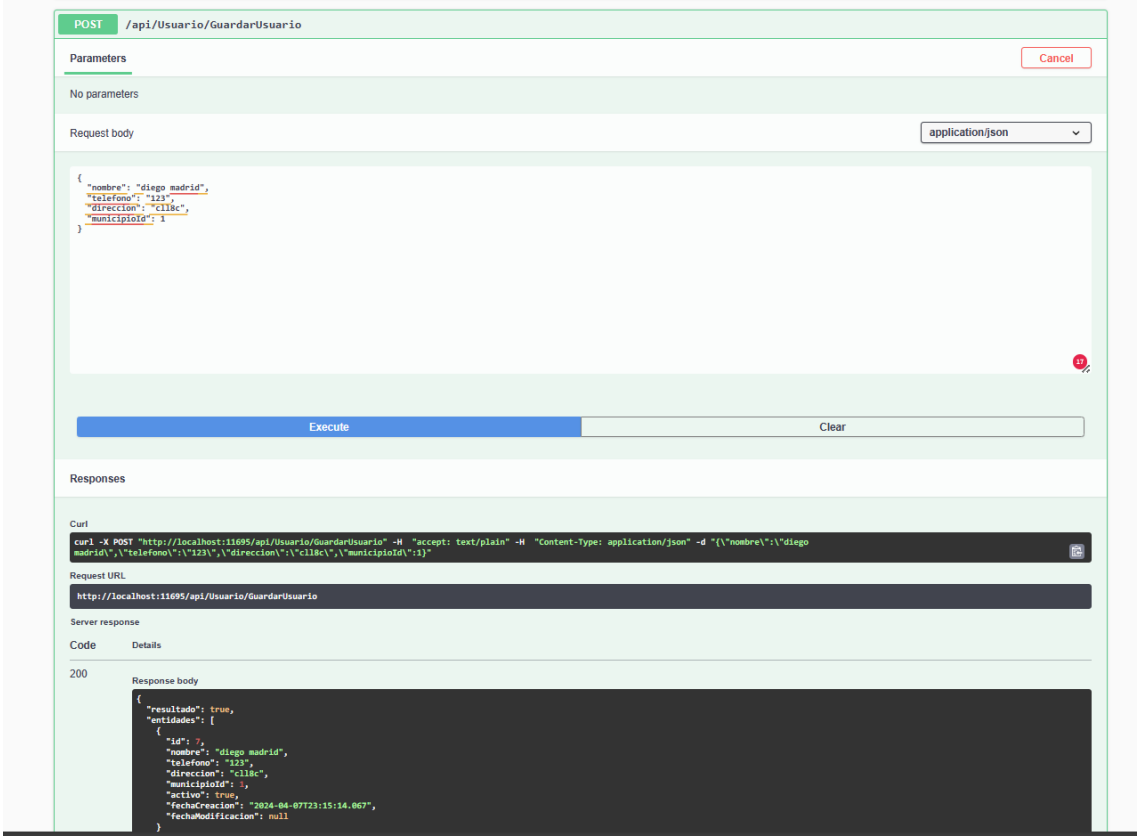
Muestra de la solución

Insertar usuario:

En caso de que se ingrese un municipio no valido se retornara un mensaje y no se guarda la información:

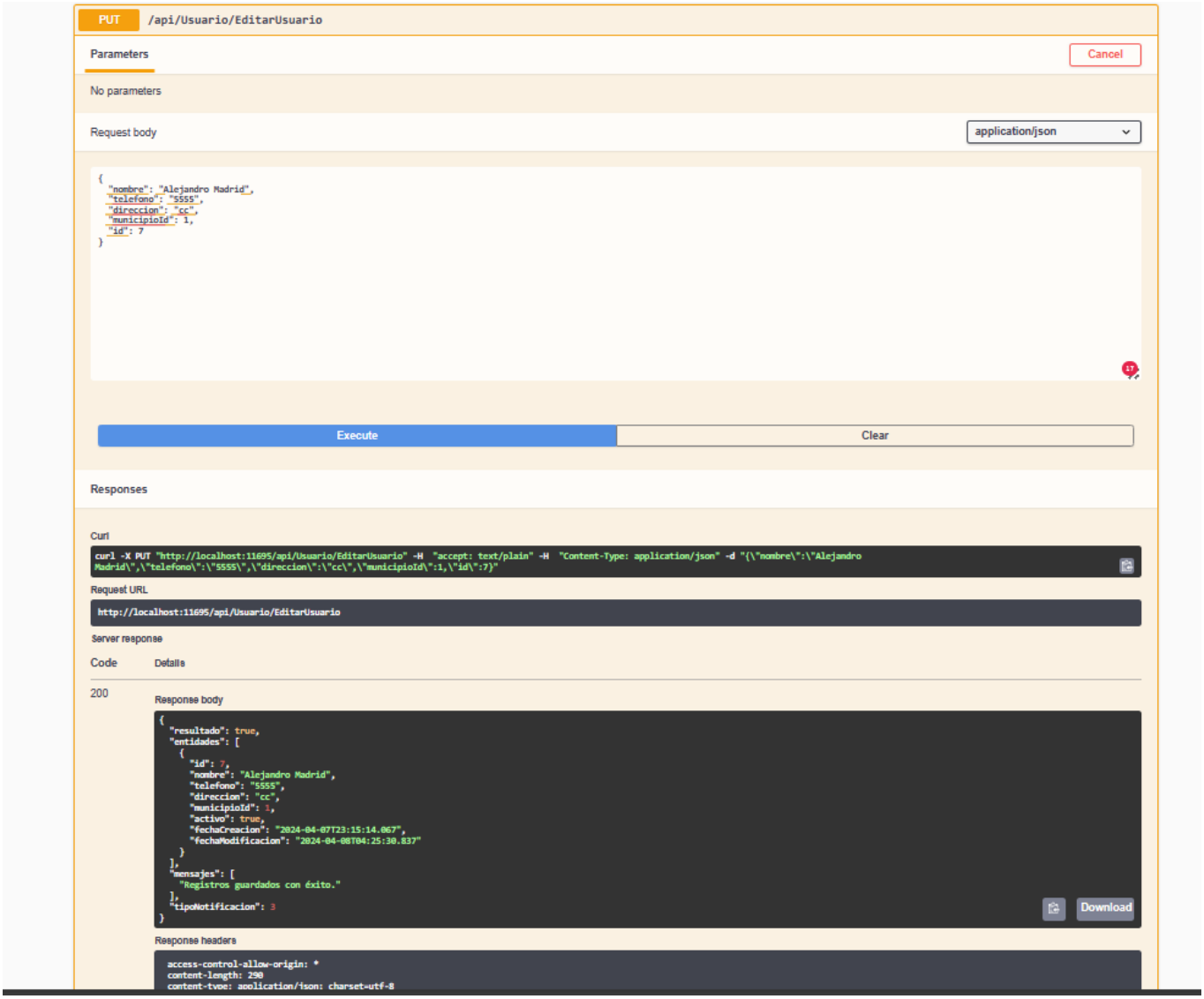


De lo contrario la información se almacena:



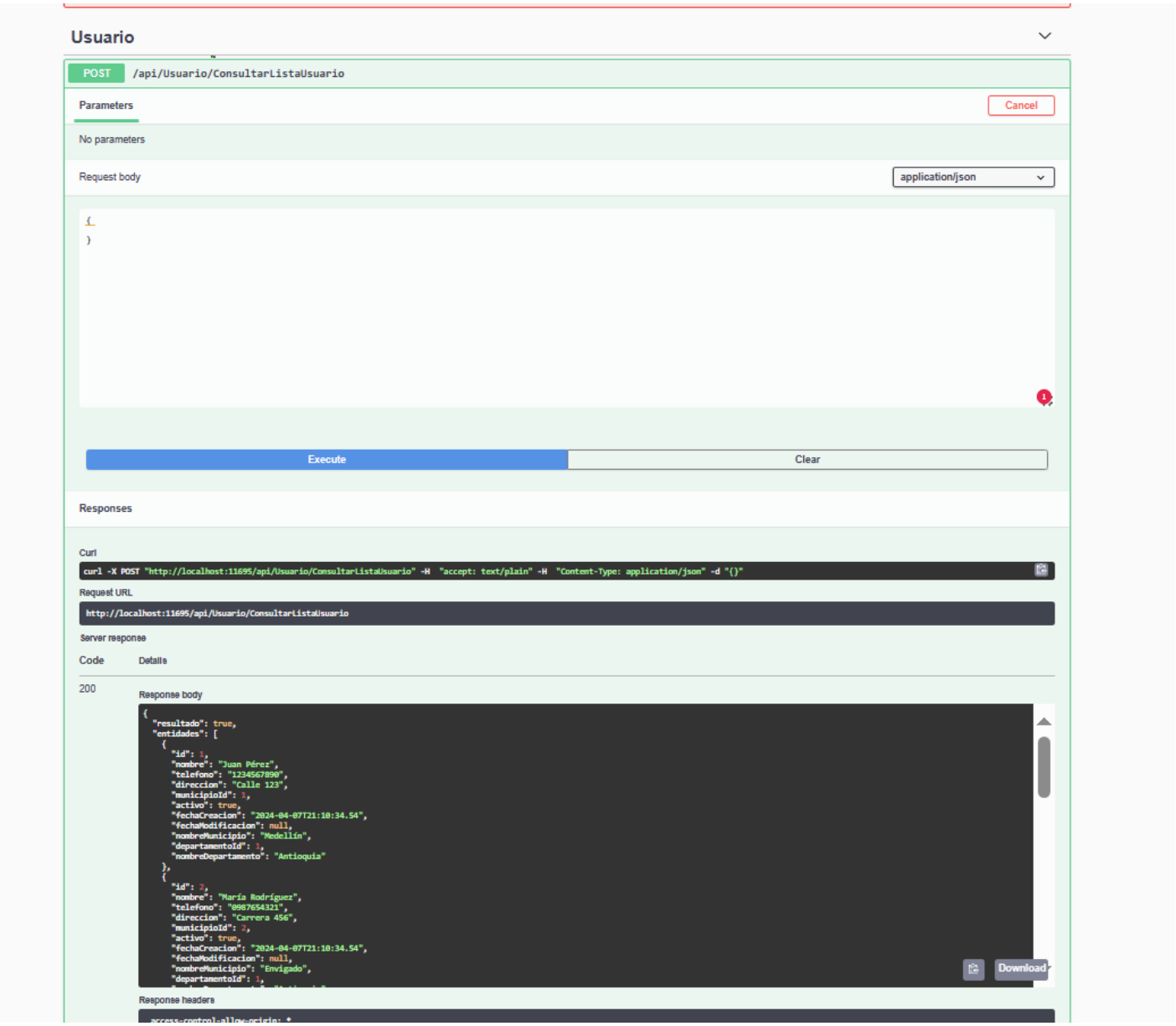
Actualizar usuario

Se actualizan los datos del usuario insertado, las validaciones están dadas por decoradores en las entidades de entrada y uso de expresiones regulares.



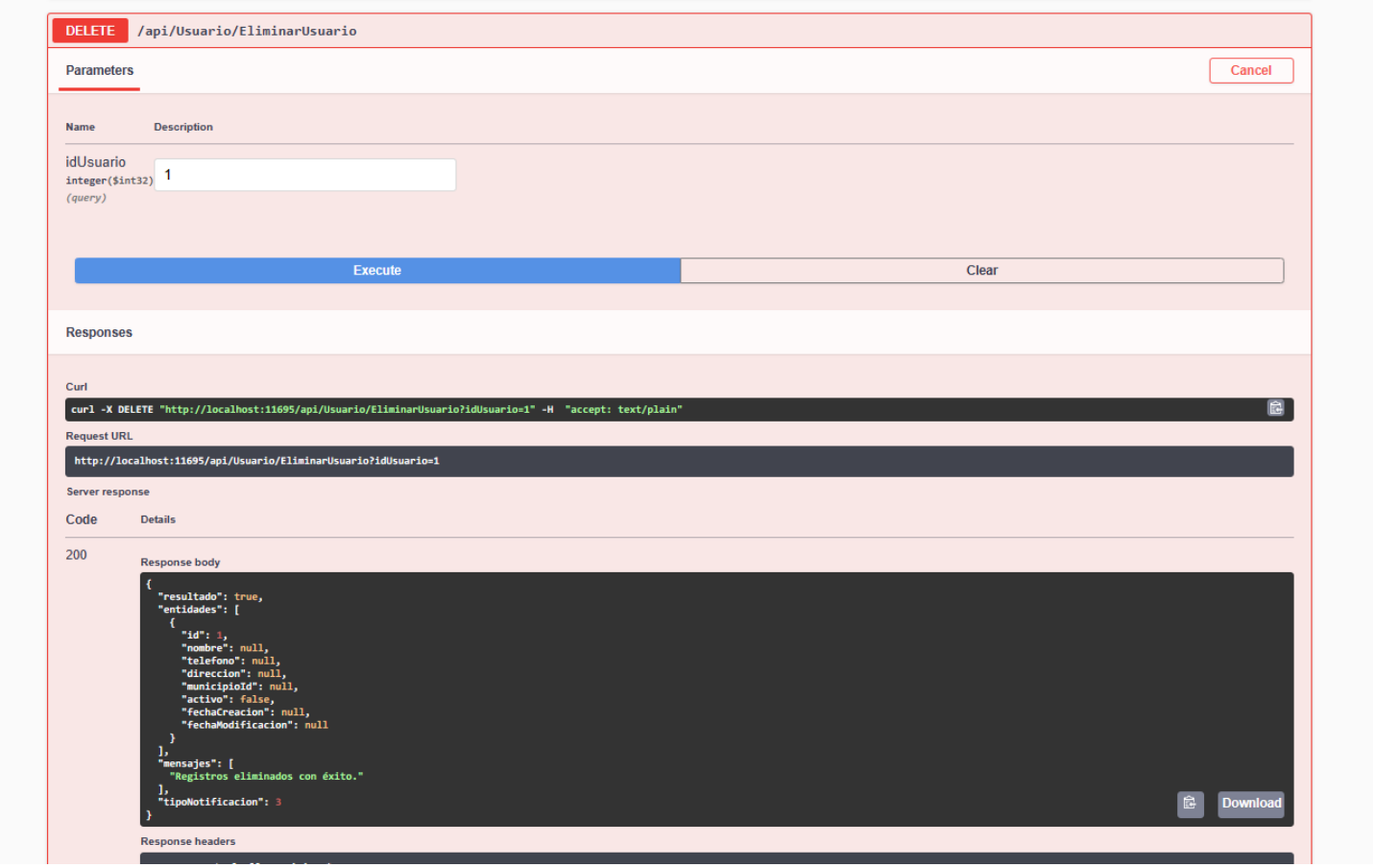
Consultar usuarios

Se realiza consulta de usuarios, en esta se identifica los usuarios con cada departamento y municipio registrados



Eliminar usuario

Se realiza eliminación lógica del registro, se inactiva el registro cambiando el valor del campo activo a false y la consulta solo trae valores activos



Modelo Base de datos: