Nesta terceira parte, daremos continuidade a construção da nossa API REST em .NET Core com Oracle XE e Stored Procedures.

Camada Controller: o ponto de entrada da API

A camada **Controller** em uma aplicação **ASP.NET Core** representa o ponto de entrada das requisições **HTTP**. Ela é responsável por receber, interpretar e encaminhar as chamadas realizadas pelo cliente, como navegadores, aplicativos móveis ou ferramentas de integração, para o serviços apropriados da aplicação. Os **controllers** devem ser simples e diretos, delegando toda a lógica de negócio a camada de serviços (**Application**). Essa separação de responsabilidades mantém o código limpo, testável e aderente aos princípios da arquitetura em camadas. Além disso, cada método do **controller** é geralmente associado a uma rota e um verbo **HTTP**(**GET**, **POST**, **PUT**, **DELETE**), formando os chamados endpoints **RESTful**.

> **Observação**: Embora existam projetos onde os desenvolvedores optam por realizar validações diretamente nas controllers (como verificar se o modelo está bem formado com ModelState.IsValid), é importante tomar cuidado para que essas validações permaneçam simples e estruturais. Regras de negócio, como verificar se um e-mail já está cadastrado ou se o usuário tem permissão para determinada ação, devem sempre ser delegadas à camada de aplicação. Isso evita violar o princípio da responsabilidade única e mantém a arquitetura coesa e sustentável.

Implementando a UserController: expondo os endpoints da API

A classe `UserController` é responsável por expor os endpoints **HTTP** da nossa **API REST**, funcionando como o ponto de entrada das operações relacionadas a usuários. Cada método corresponde a uma ação do *CRUD*, utilizando os *verbos HTTP* apropriados (GET, POST, PUT, DELETE) e delegando as operações à camada de serviço (`IUserService`). Essa estrutura mantém o *controller* enxuto e focado apenas em receber requisições, validar a estrutura dos dados e retornar respostas *HTTP* padronizadas. A lógica de negócio permanece encapsulada no serviço, garantindo separação de responsabilidade e alinhamento com os princípios da arquitetura em camadas.

```
using Microsoft.AspNetCore.Mvc;
```

```
using OracleCrud.Sp.Api.Application.Interfaces;
using OracleCrud.Sp.Api.Domain.Dtos;

namespace OracleCrud.Sp.Api.Controllers;

[ApiController]
[Route("api/[controller]")]
public class UserController : ControllerBase
{
    private readonly IUserService _userService;

    public UserController(IUserService userService)
    {
        _userService = userService;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<UserDto>>> GetAll()
    {
        var users = await _userService.GetAllAsync();
        return Ok(users);
    }

    [HttpPost]
    public async Task<ActionResult<string>> Insert([FromBody] CreateUserDto user)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        var result = await _userService.InsertAsync(user);
        return Ok(result);
    }

    [HttpPut]
    public async Task<ActionResult<string>> Update([FromBody] UserDto user)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        var result = await _userService.UpdateAsync(user);
        return Ok(result);
    }

    [HttpDelete("{id}")]
    public async Task<ActionResult<string>> Delete(string id)
    {
```

```
        var result = await _userService.DeleteAsync(id);
        return Ok(result);
    }
}
```

Program.cs: configurando o pipeline e a injeção de dependências

O arquivo `Program.cs` é o ponto de entrada da aplicação **ASP.NET Core**. Nele configuramos o pipeline de execução da aplicação, os serviços que serão injetados via dependência e os middlewares que vão atuar nas *requisições HTTP*. É nesse local que registramos o `IUserService` e o `IUserRepository` para que o **ASP.NET Core** saiba como instanciar essas dependências sempre que forem solicitadas, por exemplo, dentro da `UserController`. Também configuramos o **Swagger**, que facilita o teste manual da API, e definimos o ambiente da aplicação. Essa organização centralizada torna o ciclo de vida da aplicação mais transparente e controlado, seguindo boas práticas modernas de desenvolvimento com .NET.

Injeção de Dependência: conectando interfaces e implementações

No Program.cs, realizamos o registro das interfaces `IUserService` e `IUserRepository` com suas respectivas implementações `UserService` e `UserRepository` utilizando o método `AddScoped`. Essa abordagem segue o padrão de injeção de dependência nativo do **ASP.NET Core**, permitindo que o framework forneça automaticamente as instâncias, como nos *controllers*. O uso do `Scoped` garante que uma nova instância será criada por requisição HTTP, o que é ideal para serviços que acessam o banco de dados. Essa prática promove desacoplamento, facilidade de testes e clareza na manutenção, sendo um dos pilares da arquitetura limpa adotada no projeto.

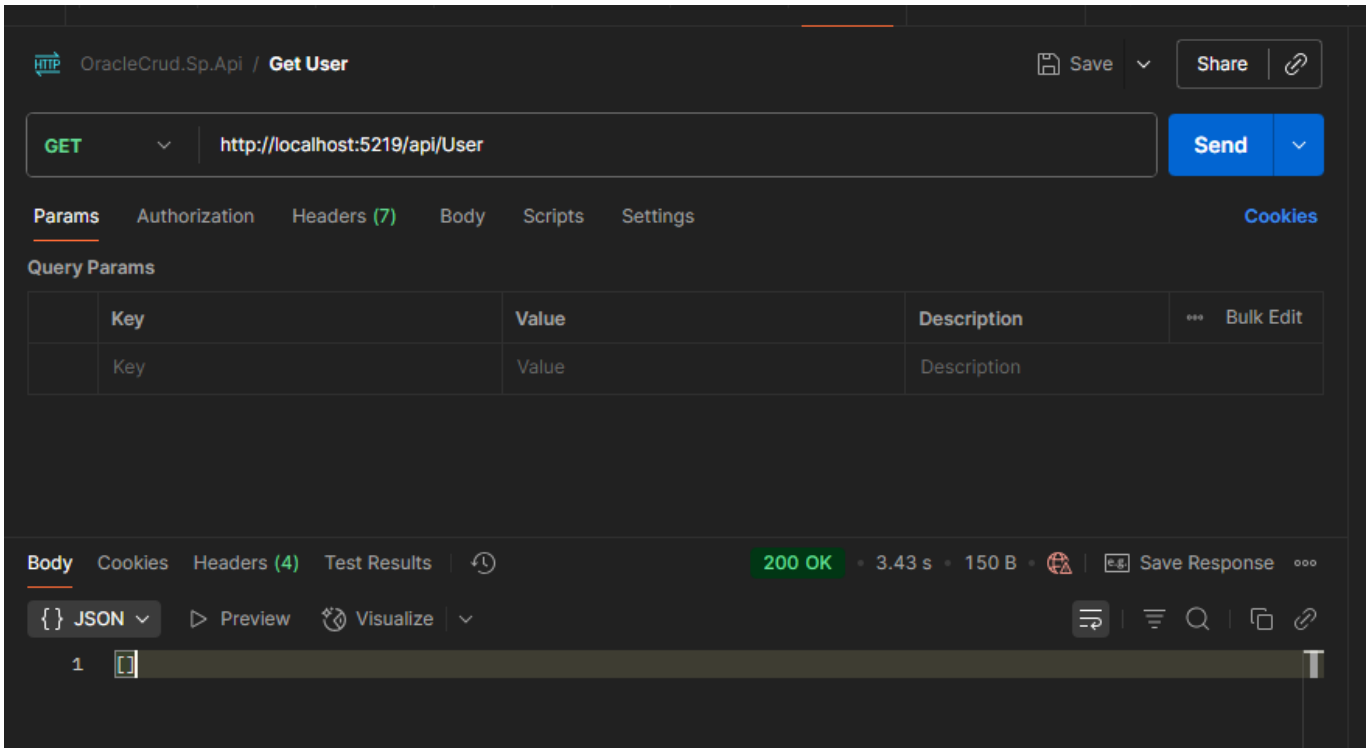
```
builder.Services.AddControllers();

builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<IUserRepository, UserRepository>();
```

Testando o endpoint GET /api/User no Postman

Ao executar o projeto em ambiente de desenvolvimento, o ASP.NET Core gera automaticamente uma URL

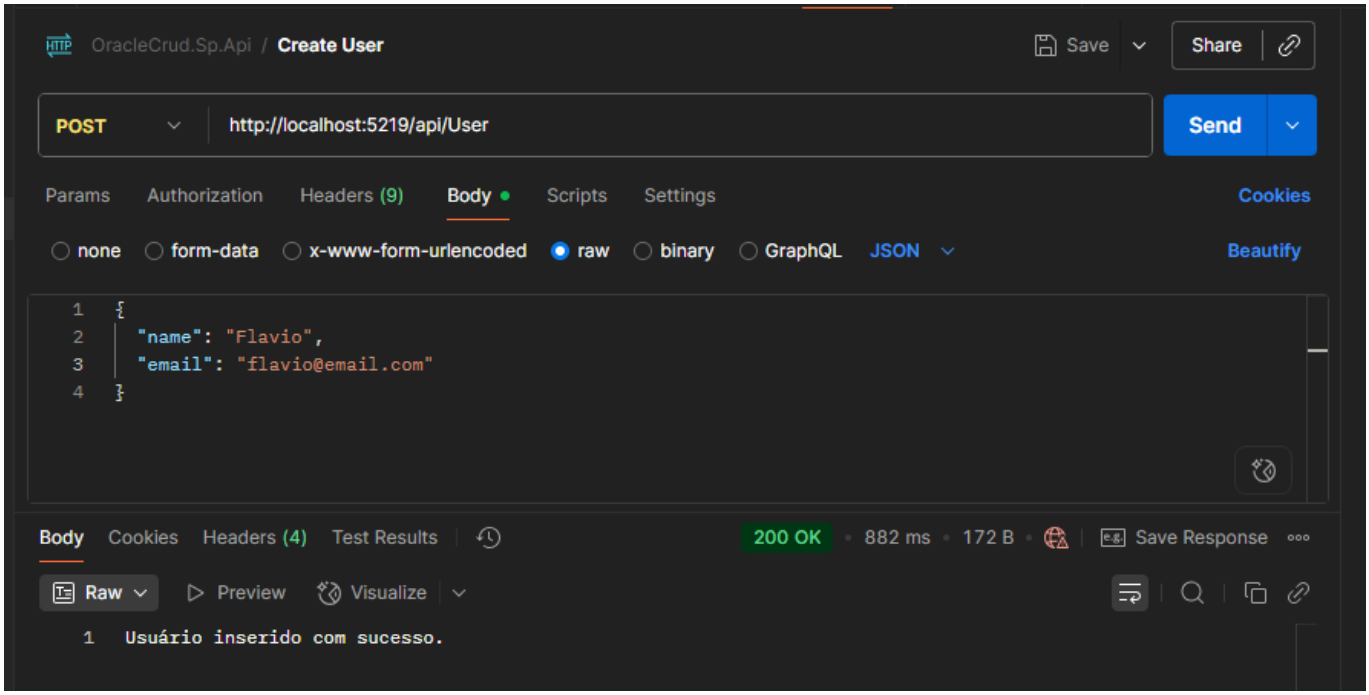
com porta dinâmica, neste caso `http://localhost:5219`, conforme definido no `launchSettings.json`. É importante destacar que, no seu ambiente, essa porta pode ser diferente, como `5000`, ou `7048`, dependendo da configuração ou do que o Visual Studio ou SDK do .NET escolherem no momento da execução. Utilizando o **Postman**, realizamos uma requisição GET para o endpoint `/api/User`, que é responsável por listar todos os usuários cadastrados. A resposta retornou com status **200 OK**, indicando que o endpoint está ativo e funcional. No entanto, o corpo da resposta foi um array vazio (`[]`), o que significa que nenhum usuário está presente no banco de dados no momento, o que é um comportamento esperado em em um ambiente recém iniciado, antes de qualquer inserção



Testando o endpoint POST /api/User no Postman

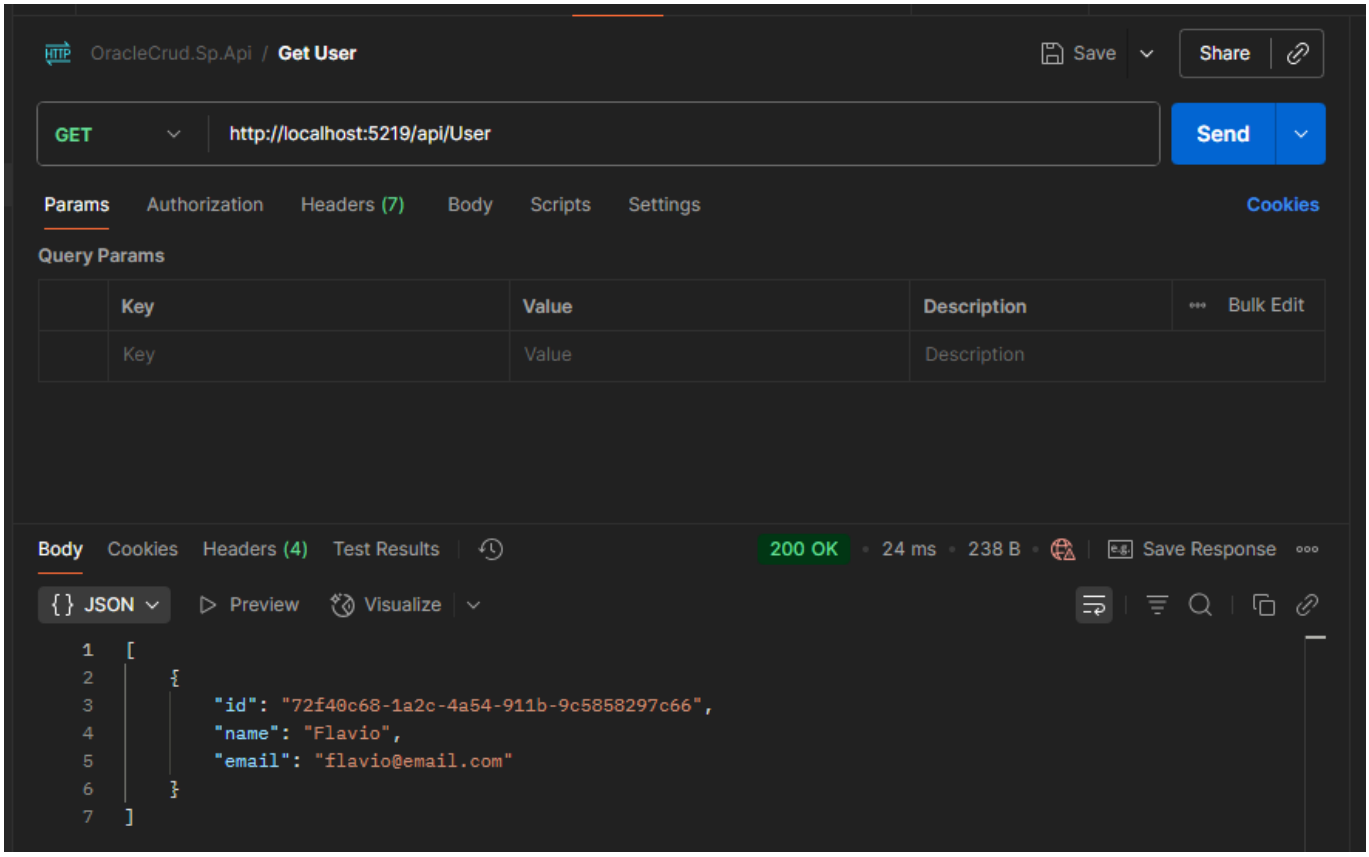
Neste teste, utilizamos o Postman para enviar uma requisição POST ao endpoint `/api/User`, com o objetivo de cadastrar um novo usuário. A requisição foi feita com o corpo no formato `JSON`, contendo os campos `name` e `email`, conforme esperado pelo **DTO** `CreateUserDto`. O endpoint respondeu com status `200 OK` e a mensagem **“Usuário inserido com sucesso.”**, indicando que a operação foi executada corretamente e os dados foram persistidos no banco de dados via stored procedure. Assim como no exemplo anterior, a URL utilizada foi `http://localhost:5219`, porém é importante lembrar que essa porta pode variar de acordo com o ambiente de execução de cada desenvolvedor, conforme definido no `launchSettings.json`. Se no seu projeto a porta for diferente, isso é completamente

normal e não afeta o funcionamento da API.



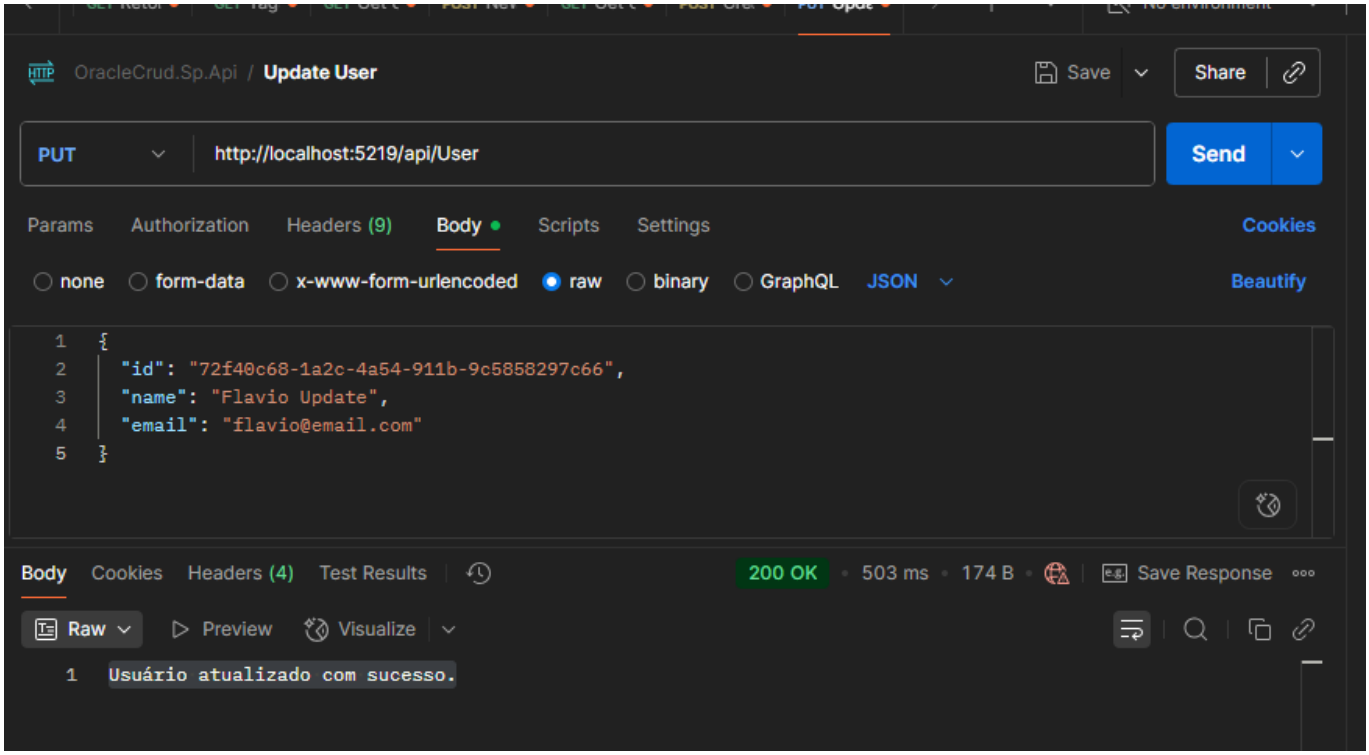
Reexecutando o GET /api/User para consultar o novo registro

Após a inserção bem sucedida de um novo usuário com o endpoint `POST /api/User`, realizamos novamente a requisição GET /api/User para verificar se o dado foi persistido corretamente no banco. Desta vez, o retorno trouxe uma lista contendo o objeto recém cadastrado, incluindo seu `id` gerado automaticamente, além dos campos `name` e `email`. O status `200 OK` confirma que a requisição foi processada com sucesso e que o fluxo de inserção e leitura está funcionando conforme esperado. Esse tipo de verificação é essencial para garantir a integridade dos dados e validar que o repositório está acessando corretamente a base **Oracle** via **stored procedures**.



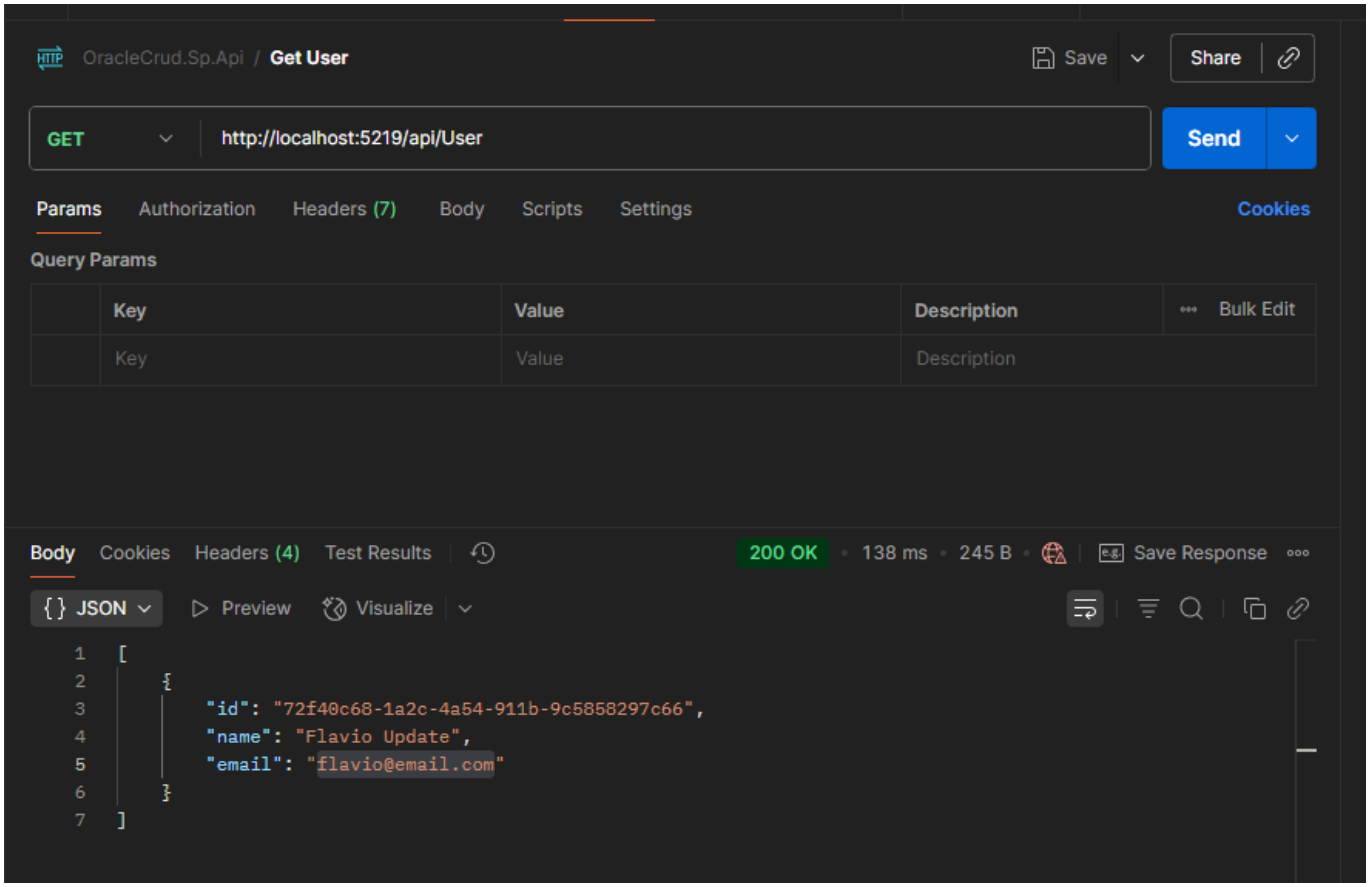
Testando o endpoint PUT /api/User para atualizar um usuário

Neste teste, utilizamos o **Postman** para enviar uma requisição PUT ao endpoint `/api/User`, com o objetivo de atualizar os dados de um usuário já existente. No corpo da requisição, foi informado o `id` do usuário previamente inserido, juntamente com os novos valores para os campos `name` e `email`. A resposta retornou com status `200 OK` e a mensagem **“Usuário atualizado com sucesso.”**, confirmando que a operação foi concluída com êxito. Isso demonstra que a comunicação com a **stored procedure** `sp_update_user` está funcionando corretamente e que a aplicação está atualizando registros no banco de dados conforme esperado. Lembre-se de que, se o id informado não existir, a própria procedure retorna uma mensagem apropriada, garantindo um controle confiável de validação no nível de banco.



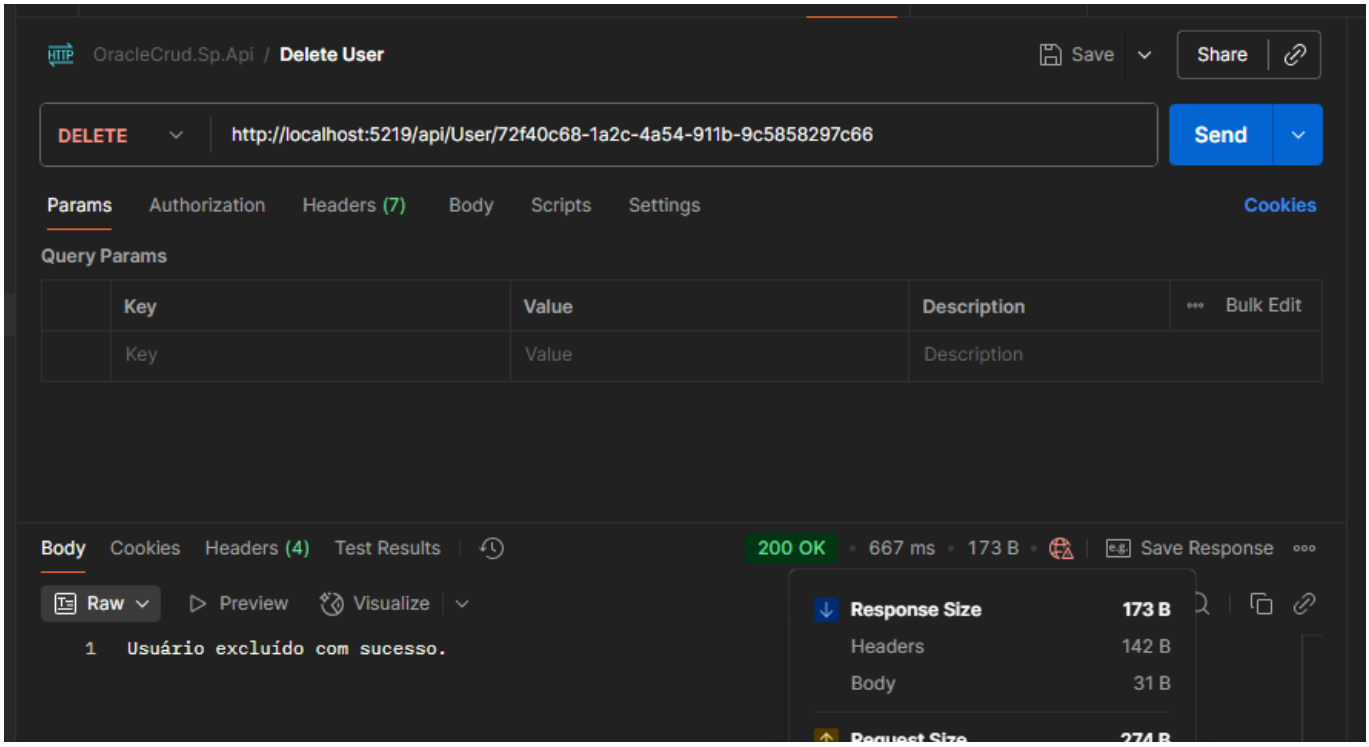
Verificando a atualização com uma nova chamada GET

Após a requisição de atualização com sucesso, reexecutamos o endpoint GET /api/User para verificar se os dados do usuário foram realmente modificados no banco. A resposta confirmou a alteração, retornando o nome atualizado com **“Flavio Update”** e o nomes e-mail previamente cadastrado. Isso demonstra que o processo de persistência com stored procedure está funcionando corretamente, refletindo imediatamente as mudanças realizadas.



Testando o endpoint DELETE /api/User para remover um usuário

Neste teste, utilizamos o **Postman** para enviar uma requisição DELETE ao endpoint `/api/User/{id}`, informando diretamente na URL o id do usuário que havia sido previamente inserido e atualizado. A resposta retornou com status `200 OK` e a mensagem **“Usuário excluído com sucesso.”**, confirmando que a operação foi concluída corretamente. Essa resposta é gerada pela stored procedure `sp_delete_user`, que também valida se o `id` existe no banco e retorna mensagens específicas em caso de erro. Com isso, completamos o ciclo do **CRUD** (**Create**, **Read**, **Update**, **Delete**), validando que todas as operações estão integradas corretamente com o banco Oracle.



Conclusão

Na terceira parte da série **Construindo uma API REST em .NET com Oracle XE e Stored Procedures [Parte 3]**, implementamos a camada de controllers da aplicação e realizamos testes práticos dos principais endpoints do CRUD: GET, POST, PUT e DELETE. Exploramos como os controllers se comunicam com a camada de serviços, como configurar corretamente a injeção de dependência no Program.cs e validamos toda a integração com o **banco de dados Oracle** por meio de chamadas **HTTP** usando o **Postman**. Com isso, nossa API já é capaz de executar todas as operações básicas com segurança e organização. Na **Parte 4**, concluiremos a aplicação com um resumo geral do que construímos até aqui, consolidando o aprendizado e reforçando os principais conceitos abordados ao longo da série.