Aula 35 - Introdução a Desenvolvimento Backend

Docupedia Export

Author:Silio Leonardo (CtP/ETS)
Date:13-Jun-2023 13:53

Table of Contents

- Fundamentos em Arquitetura Web
- Requisições HTTP, REST e JSON
- C# WebAPI
- Rotas e Parâmetros
- Injeção de Dependência e Serviços
- Padrão Repository
- Classe HttpClient

1 Fundamentos em Arquitetura Web

O desenvolvimento Web pode ter muitas abordagens. Uma abordagem antiga porém ainda muito usada são os monólitos. Monólitos são aplicações que não separam seus componentes em mais de um sistema. Isso significa que a aplicação se conectará no banco de dados, fará todo o processamento e, ainda, disponibilizará o front-end de alguma forma. Por exemplo, poderá retornar código CSS/HTML para o cliente com os dados de uma dada página. Aplicações monolíticas rodam sobre uma única máquina e estão sujeitos aos limites de desempenho deste servidor.

Podemos também usar a arquitetura de Microsserviços para melhorar o desempenho. Ela consiste em separar o projeto em vários pequenos programas que podem rodar em diversos computadores ou na nuvem. Esses serviços se comunicam e podem ser acessados por um API Gateway.

Neste curso consideraremos diversas abordagens para resolução dos problemas, muito embora, certamente teremos ao menos 2 serviços: O Angular para frontend e o C# para backend.

Diferentemente de outras oportunidades, não utilizaremos exatamente do MVC para criar aplicações. O MVC geralmente funciona como aplicações como monolíticas que cuidam de todos os processos do software como o ciclo de renderização. Aqui trabalharemos com front e back desacoplados, isto é, duas aplicações independentes que se comunicam. É importante notar que em diferentes projetos podemos ter diferentes abordagens e padrões arquitetônicos.

2 Requisições HTTP, REST e JSON

A comunicação entre processos se da por requisições HTTP. Essas são requisições TCP, ou seja, sem perdas de dados. Como toda requisição HTTP temos componentes em comum, mas isso não é suficiente para caracterizar a forma como vamos trabalhar. Iremos trabalhar sobre o padrão REST. O REST (Representational State Transfer) é um padrão de comunicação que envolve o uso de método, cabeçalho e corpo. Quando uma aplicação segue o REST chamamos ela de RESTful. Abaixo um exemplo de requisição REST:

```
https://myservice.com/endpoint

GET

content-type: application/json

{
    "mydata": 4
}
```

O request tem seu alvo (uma URL). Note que isso caracteriza o REST como uma comunicação unidimensional. Evidentemente, nada impede que 2 sistemas usem o REST e tentem conversar, contudo, a priori, a estruturas não facilitaram uma conversa sequencial e é ineficaz para implementar coisas como jogos online. Apesar disso, ainda sim existe uma, e apenas uma, resposta de toda requisição REST.
Logo depois vem o método. A principio o método só tem sentido semântico. Isso significa ele não afeta em nada o comportamento da Request, porém, os padrões REST apontam métodos válidos e quando usamos cada um deles. Os mais importantes são: GET, usado ao ler dados; POST, usado ao salvar dados; PUT, usado ao alterar os dados; e DELETE, usado ao deletar dados. Muito embora sejam semânticos é importante seguir as recomendações. Além disso, algumas bibliotecas podem bloquear algumas operações inconsistentes com seus métodos.
Ai temos o header. No header traremos várias configurações e definições sobre a requisição em si.
Por último o corpo. Nem todas as requisições devem ter corpo. Por exemplo, em geral requisições GET não possuem (mas seu retorno contém). O corpo são os dados e podem ser estruturados em XML, JSON ou até mesmo HTML. O mais usado será o JSON (JavaScript Object Notation). Basicamente a notação usada para definir objetos no JavaScript como dicionários é extremamente utilizada para enviar dados pela rede em requisições REST.
Ao acessar uma página na web é comum que estejamos realizando um GET e obtendo um HTML da página. Nosso serviço Angular fará isso o tempo todo.

3 C# WebAPI

Para criar uma WEP API Rest em C# que está apta a receber e responder requisições REST. Para isso podemos usar o seguinte comando:

```
dotnet new webapi
```

Ele irá inicializar um projeto bem mais complexo que uma aplicação console. Inclusive alguns arquivos de exemplo que podem ser removidos. São eles o WeatherForecast e Controllers/WeatherForecastController. Na pasta de Controllers ficarão os controladores. Um controlador é uma classe que possui endpoints que, por sua vez, são códigos executáveis que podem receber requisições REST. Após isso faramos um arquivos TestController.cs na pasta de Controllers para criar nosso primeiro endpoint.

```
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace ProjetoWeb.Controllers;
4
5  [ApiController]
6  [Route("test")]
7  public class TestController : ControllerBase
8  {
9      [HttpGet]
10     public string Get()
11     {
12         return "Server is running...";
13     }
14 }
```

Como você pode ver ele será acessado da seguinte forma: um requisição de GET para https://nomedoservidor/test e será apresentando "Server is running..." na tela do navegador, ou seja, na resposta da requisição. A seguir vamos explorar um pouco como configuramos tudo. Por enquanto é bom atentar-se a simplicidade da estrutura: Trata-se de uma função que retorna exatamente o que será recebido do outro lado. Você ainda pode retornar objetos complexos que serão convertidos automaticamente em um JSON para consumir em outros lugares.

4 Rotas e Parâmetros

```
public class CpfService
{
    private int getVerificationDigits(Cpf cpf)
    {
        var str = cpf.Value;

        int sum = 0;
        for (int i = 0; i < 8; i++)
        {
            int digit = str[i] - '0';
            sum += (i + 2) * digit;
        }
        int verifier1 = sum % 11;

        for (int i = 0; i < 8; i++)
        {
            int digit = str[i] - '0';
            sum -= digit;
        }
        sum += 9 * verifier1;
        int verifier2 = sum % 11;

        return 10 * verifier1 + verifier2;
    }

    public void Validate(Cpf cpf)
    {
        if (cpf is null)
            throw new ArgumentNullException("cpf");

        int verifier = getVerificationDigits(cpf);
        bool isValid = verifier == cpf.VerificationDigit;

        cpf.Verified = true;
        cpf.Validated = isValid;
    }
```

```
37
38      public Cpf Generate(int region = -1)
39      {
40          Cpf cpf = new Cpf();
41
42          if (region == -1)
43              region = Random.Shared.Next(10);
44
45          cpf.FiscalRegionDigit = region;
46          cpf.RandomDigits = Random.Shared.Next(100_000_000);
47          cpf.VerificationDigit = getVerificationDigits(cpf);
48
49          cpf.Verified = true;
50          cpf.Validated = true;
51          return cpf;
52      }
53  }
54
55  public class Cpf
56  {
57      public string Value
58      {
59          get => $"{RandomDigits:000.000.00}{FiscalRegionDigit}-{VerificationDigit:00}";
60          set
61          {
62              if (value is null)
63                  throw new InvalidCastException("Value can't be null.");
64
65              value = value
66                  .Replace("-", "")
67                  .Replace(".", "");
68
69              if (value.Length != 11)
70                  throw new InvalidCastException("Invalid number of digits.");
71
72              RandomDigits = int.Parse(value.Substring(0, 8));
73              FiscalRegionDigit = int.Parse(value.Substring(8, 1));
74              VerificationDigit = int.Parse(value.Substring(9, 2));
75          }
```

```
    }

    public string Region
    {
        get => FiscalRegionDigit switch
        {
            1 => "DF, GO, MT, MS e TO",
            2 => "AC, AP, AM, PA, RO e RR",
            3 => "CE, MA e PI",
            4 => "AL, PB, PE e RN",
            5 => "BA e SE",
            6 => "MG",
            7 => "ES e RJ",
            8 => "SP",
            9 => "PR e SC",
            10 => "RS",
            _ => "Região desconhecida"
        };
    }

    public int RandomDigits { get; set; }
    public int FiscalRegionDigit { get; set; }
    public int VerificationDigit { get; set; }
    public bool Verified { get; set; } = false;
    public bool Validated { get; set; } = false;
}
```

Controllers/CpfController.cs

```
using Microsoft.AspNetCore.Mvc;

namespace ProjetoWeb.Controllers;

[ApiController]
[Route("cpf")]
public class CpfController : ControllerBase
{
    [HttpGet("{cpf}")]
```

```
10      public ActionResult<Cpf> Get(string cpf)
11      {
12          Cpf result = new Cpf();
13          try
14          {
15              result.Value = cpf;
16          }
17          catch (Exception ex)
18          {
19              return BadRequest(ex.Message);
20          }
21
22          return result;
23      }
24  }
```

5 Injeção de Dependência e Serviços

Podemos permitir através de injeção de dependência que o .NET gerencie quais serviços, como o CpfService, serão entregues a uma função. Veja abaixo como cadastrar um serviço e como usá-lo:

Program.cs

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddTransient<CpfService>();

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

CpfController.cs

```
    }

    // Usa um serviço usando o atributo FromServices que foi cadastrado no Program.cs
    [HttpGet("generate/{region}")]
    public ActionResult<Cpf> Generate([FromServices]CpfService cpf, int region)
    {
        var result = cpf.Generate(region);

        return result;
    }
}
```

AddTransient criará um objeto sempre que necessário que o objeto seja recriado toda vez que precisar ser utilizado. Ainda existem o AddScoped que cria o objeto que é reutilizado para o mesmo escopo, isso é confuso e facilmente confundível com o AddTransient, mas construiremos um exemplo para podermos diferenciá-los. Ainda existe o AddSingleton que criará um objeto único usado por toda a aplicação.
Além disso, como serviços e controladores são constantemente criados pelo Framework, você pode pedir um tipo no construtor que será injetado com base nas suas configurações e você pode salvar este objeto na classe como se fosse global. Note que a cada requisição recebida os controladores e os serviços não singleton serão recriados com base na demanda.
Outro fator interessante é que podemos criar dependências baseadas em interfaces, isso significa que podemos criar uma estrutura completa baseando-se apenas nas interfaces dos serviços e mudar suas implementações como quisermos. Faremos um exemplo interessante logo a seguir junto de um interessante e novo padrão de projetos.

6 Padrão Repository

Vamos supor que queremos testar uma aplicação criada sem afetar os dados reais que já estão no sistema. Isso pode ser uma tarefa chata ou até mesmo difícil.
Vamos aplicar na conexão com o seguinte banco de dados:

```
create database repoExemplo
go

use repoExemplo
go

create table Mensagem(
    ID int identity primary key,
    Texto varchar(MAX) not null,
    Horario datetime not null
)
go
```

Vamos criar uma conexão com o banco de dados com o Entity framework e vamos definir a seguinte interface para representar um repositório, ou seja uma estrutura de acesso a um conjunto de dados.
Isso nos dará um RepoExemploContext e usaremos ele como um serviço no nosso sistema.
Vamos agora implementar o padrão repository baseado na seguinte interface:

IRepository.cs

```
using System.Linq.Expressions;

public interface IRepository<T>
{
    Task<List<T>> Filter(Expression<Func<T, bool>> exp);
    void Add(T obj);
    void Delete(T obj);
    void Update(T obj);
}
```

Um repositório então, seria um objeto capaz de adicionar, alterar, deletar e ler um objeto de um tipo T. Podemos fazer implementações concretas como o repositório de mensagem abaixo:

MessageRepository.cs

```
1  using backend.Model;
2
3  public class MessageRepository : IRepository<Mensagem>
4  {
5      private RepoExemploContext entity;
6      public MessageRepository(RepoExemploContext service)
7          => this.entity = service;
8
9      public async Task<List<Mensagem>> Filter(Expression<Func<Mensagem, bool>> exp)
10     {
11         return await entity.Mensagems
12             .Where(exp)
13             .ToListAsync();
14     }
15
16     public void Add(Mensagem obj)
17     {
18         entity.Add(obj);
19         entity.SaveChanges();
20     }
21
22     public void Delete(Mensagem obj)
23     {
24         entity.Remove(obj);
25         entity.SaveChanges();
26     }
27
28     public void Update(Mensagem obj)
29     {
30         entity.Update(obj);
31         entity.SaveChanges();
32     }
33 }
```

Note que essa classe pede no construtor um RepoExemploContrext, ou seja, só poderá ser instanciado por injeção de dependência se for configurado um RepoExemploContext na nossa Program.cs. Observe o que será feita neste arquivo abaixo:

Program.cs

```
builder.Services.AddSwaggerGen();

builder.Services.AddScoped<RepoExemploContext>();
builder.Services.AddTransient<IRepository<Mensagem>, MessageRepository>();
builder.Services.AddTransient<CpfService>();

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

Observe que interessante: Ao se pedir um repositório de mensagens você receberá a implementação do MessageRepository. Essa implementação é transiente, ou seja, será instanciado toda que vez que necessário. Já o RepoExemploContext é baseado em escopo, ou seja, para cada requisição será criado uma única instância. Isso significa que se tivermos 2 serviços que utilizam banco de dados sendo usados e ambos precisam usar o contexto do banco de dados o mesmo contexto do banco de dados será utilizado. Isso é ótimo pois impede que duas conexões diferentes sejam usadas em cada repositório (isso pode causar problemas ao fazer joins, já que duas conexões diferentes não podem montar e executar a mesma query).
Podemos fazer outros repositórios que conectam com outros bancos ou tem dados falsos apenas para testes, por exemplo, um repositório fake, observe:

FakeMessageRepository.cs

```
using backend.Model;

public class FakeMessageRepository : IRepository<Mensagem>
{
    private List<Mensagem> fakeData = new List<Mensagem>()
    {
        new Mensagem()
        {
            Id = 1,
            Horario = DateTime.Now.AddDays(-1),
            Texto = "Don, platina o cabelo por favor"
        },
        new Mensagem()
        {
            Id = 2,
            Horario = DateTime.Now.AddDays(-.5),
            Texto = "Se alguém perguntar, você não sabe o que aconteceu com a robodrill"
        },
```

```
        new Mensagem()
        {
            Id = 2,
            Horario = DateTime.Now,
            Texto = "VOCÊ SABE QUEM QUEBROU A ROBODRILL?"
        },
    };

    public void Add(Mensagem obj)
        => fakeData.Add(obj);

    public void Delete(Mensagem obj)
        => fakeData.Remove(obj);

    public void Update(Mensagem obj)
    {
        var old = fakeData
            .FirstOrDefault(m => m.Id == obj.Id);
        if (obj is null)
            return;

        fakeData.Remove(old);
        fakeData.Add(obj);
    }
}
```

E no Program.cs poderemos fazer o seguinte:

Program.cs

```
builder.Services.AddSwaggerGen();

var env = builder.Environment;

builder.Services.AddScoped<RepoExemploContext>();
if (env.IsDevelopment())
    builder.Services.AddTransient<IRepository<Mensagem>, FakeMessageRepository>();
else if (env.IsProduction())
    builder.Services.AddTransient<IRepository<Mensagem>, MessageRepository>();
```

```
10
11  builder.Services.AddTransient<CpfService>();
12
13  var app = builder.Build();
```

Se executar com um "**dotnet run**" executaremos como desenvolvimento usando um repositório fake, se usarmos "**dotnet run --environment Production**" estaremos iniciando em produção e conectando-se a um banco real. Então um controlador pode ter comportamento diferente a depender do environment definido no projeto.

RepositoryController.cs

```
1   using backend.Model;
2   using Microsoft.AspNetCore.Mvc;
3
4   namespace ProjetoWeb.Controllers;
5
6   [ApiController]
7   [Route("message")]
8   public class MesssageController : ControllerBase
9   {
10      [HttpGet]
11      public async Task<ActionResult<IEnumerable<Mensagem>>> GetAll(
12          [FromServices]IRepository<Mensagem> repo
13      )
14      {
15          var result = await repo.Filter(x => true);
16          return result;
17      }
18  }
```

7 Classe HttpClient

Podemos fazer requisições a outros serviços pré-existente para realizar tarefas. Vamos então definir um projeto para buscar dados a respeito de um CEP inserido no sistema. Iremos utilizar o Via Cep para descobrirmos a rua, cidade e estado de um Cep inserido no sistema. Para isso usaremos a classe HttpClient:

```
1  public class CepData
2  {
3      public string Cep { get; set; }
4      public string Logradouro { get; set; }
5      public string Complemento { get; set; }
6      public string Bairro { get; set; }
7      public string Uf { get; set; }
8      public string Ibge { get; set; }
9      public string Gia { get; set; }
10     public string Ddd { get; set; }
11     public string Siafi { get; set; }
12 }
```

ICepService.cs

```
1  public interface ICepService
2  {
3      Task<CepData> Get(string cep);
4  }
```

CepService.cs

```
1  using System.Net;
2
3  public class CepService : ICepService
4  {
5      public CepService(string service)
6          => this.client = new HttpClient()
7          {
8              BaseAddress = new Uri(service)
```

```
        };

    private HttpClient client;

    public async Task<CepData> Get(string cep)
    {
        var response = await client
            .GetAsync($"/{cep}/json");

        if (response.StatusCode != HttpStatusCode.OK)
            return null;

        var obj = await response.Content
            .ReadFromJsonAsync<CepData>();

        return obj;
    }
}
```

Program.cs

```
var env = builder.Environment;
const string url = "https://viacep.com.br/ws/80320330/json/";

builder.Services.AddScoped<RepoExemploContext>();
if (env.IsDevelopment())
    builder.Services.AddTransient<IRepository<Mensagem>, FakeMessageRepository>();
else if (env.IsProduction())
    builder.Services.AddTransient<IRepository<Mensagem>, MessageRepository>();

builder.Services.AddSingleton<ICepService>(p => new CepService(url));

builder.Services.AddTransient<CpfService>();
```

CepController.cs

```
using Microsoft.AspNetCore.Mvc;

namespace ProjetoWeb.Controllers;

[ApiController]
[Route("cep")]
public class CepController : ControllerBase
{
    [HttpGet("{cep}")]
    public async Task<ActionResult<CepData>> Get(
        [FromServices]CepService service, string cep
    )
    {
        var result = await service.Get(cep);

        if (result is null)
            return NotFound();

        return result; // implicit convertion to ActionResult
    }
}
```