



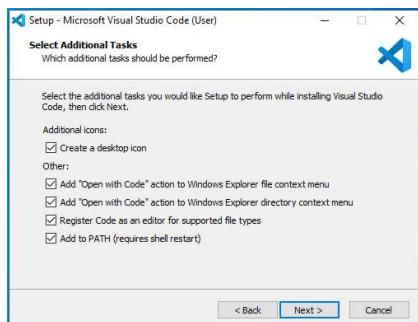
AULA INAUGURAL - DESENVOLVIMENTO DE SISTEMAS - AULA 01

- Apresentação da disciplina
- Competências, habilidades e bases tecnológicas da disciplina
- Formas de Avaliação
- Introdução e desenvolvimento do conteúdo

Download e Instalação do Visual Studio Code

Faça o Download do Visual Studio Code acessando <https://code.visualstudio.com/download>

Instale deixando todas as opções selecionadas



Linhas de Comandos PowerShell

Para criação de projetos utilizaremos a ferramenta de linha de comandos do Windows chamada de powershell



Comandos básicos

Mudar pasta → cd (change directory) - Ex:

```
PS C:\Users\Luiz> cd d:\work\etec\ds2020-1\APIs
PS D:\work\etec\ds2020-1\APIs>
```

Listar arquivos e pastas → ls (list)

```
PS D:\work\etec\ds2020-1\APIs> ls

Directory: D:\work\etec\ds2020-1\APIs

Mode                LastWriteTime       Length Name
----                -              -           -
d----- 26/06/2020     22:43            0 TesteApi
d----- 23/06/2020     16:47            0 testeef
```



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

Criar Pasta → mkdir (make directory)

```
PS D:\work\etec\ds2020-1\APIs> mkdir PastaTeste

Directory: D:\work\etec\ds2020-1\APIs

Mode          LastWriteTime     Length Name
----          -----          ----- 
d-----      30/06/2020    09:30          PastaTeste

PS D:\work\etec\ds2020-1\APIs>
```

- Estes caminhos se referem as pastas locais de exemplo, você poderá criar e listas os diretórios da maneira que achar melhor

Verificando a versão do .NET Core instalado

```
PS C:\Users\Luiz> dotnet --version
5.0.302
```

Se seu computador não exibir a versão ou não reconhecer o comando, instale o .Net Core através do link:
<https://dotnet.microsoft.com/download/dotnet-core>

Version	Status	Latest release	Latest release date	End of support
.NET 6.0	Preview ⓘ	6.0.0-preview.6	July 14, 2021	
.NET 5.0 (recommended)	Current ⓘ	5.0.8	July 13, 2021	
.NET Core 3.1	LTS ⓘ	3.1.17	July 13, 2021	December 03, 2022
.NET Core 2.1	LTS ⓘ	2.1.28	May 11, 2021	August 21, 2021

Vá em computador, clique com o direito do mouse e em propriedades, para verificar se seu Windows é 32 ou 64 bits e faça o download compatível com seu computador

OS	Installers	Binaries
Linux	Package manager instructions	ARM32 ARM64 RHEL 6 x64 x64 x64 Alpine
macOS	x64	Windows 64 bits x64
Windows	x64 x86	Windows 32 bits x64
All	dotnet-install scripts	ARM32 x64 x86



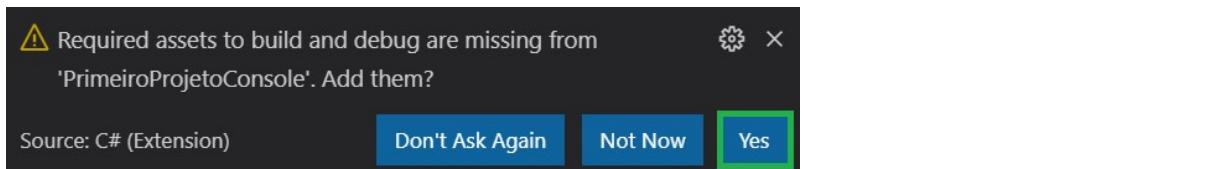
Criação de Projeto no Visual Studio Code

- Crie uma pasta onde costuma guardar os projetos do curso através dos comandos revisados até aqui.
Sugestão: criar uma pasta no endereço `d:/2DS/SEU_NOME/DS` (No laboratório)
- Certifique de que não é um caminho longo e que não existem caracteres especiais nas pastas.

Abra a pasta recém-criada e crie outra chamada **HelloWorld**. Abra o VS Code escolha a opção → File → Open Folder, selecionando a pasta *HelloWorld*. Vá até o menu View → Terminal e realize os comandos a seguir:

- Digite o comando `dotnet new -h`. Ele exibe os tipos de projetos que podem ser criados. Criaremos do tipo console.
- Para criar um projeto utilize a linha de comando a seguir. `dotnet new Console --framework net5.0`
- Digite o comando para abrir o vs code abrindo o projeto criado `code .`

Ao clicar no arquivo *Program.cs* ou qualquer arquivo com a extensão “.cs”, o vs code solicita instalar e extensão C# para habilitar a depuração. Para essa pergunta sempre clique em **Yes**.

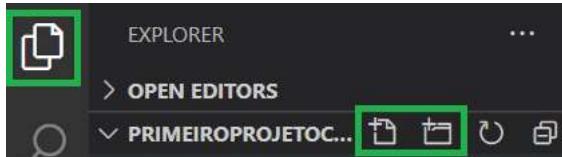


- Para compilar o projeto navegue até o menu View → Terminal e execute `dotnet build`
- O resultado esperado é sempre 0 erros
- Para rodar o projeto execute o comando a seguir no terminal `dotnet run`

Extensões importantes para o VS Code: É possível adicionar extensões ao VS Code. Vá até até o menu View → Extensions e na caixa de busca, digite as extensões listada abaixo e clique no botão install

- C# Extensions – Autor: jchannon
- Material Icon Theme – Autor: Philipp Kief
- C# For Visual Studio Code – Autor: Microsoft (provavelmente já estará adicionada)

Podemos observar os arquivos abertos conforme a imagem abaixo e os ícones em que podemos criar arquivos e pastas, sendo possível criar arquivos e pastas clicando com o botão direito.



- O arquivo `C# Program.cs` é uma classe e é o ponto de partida para a execução do projeto.



Realize a programação abaixo

```
using System; A

namespace HelloWorld B
{
    class Program C
    {
        static void Main(string[] args) D
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

- Execute o comando *dotnet build* para compilar
- Execute o comando *dotnet run* para executar

Observações

- (A) Referência para classes do dotnet ou componentes para programação.
(B) Namespace do arquivo: É um endereço lógico que representa em que hierarquia de pasta o arquivo está contido.
(C) Declaração de classe Program, sendo a assinatura a descrição da classe e tudo que está entre as chaves sendo o corpo desta classe
(D) Método principal da classe. Necessário para inicialização do programa. O comando Console.Write

Altere o programa conforme abaixo:

```
static void Main(string[] args)
{
    Console.WriteLine("Digite seu nome:");
    string nome = Console.ReadLine();

    Console.WriteLine($"Seu nome tem {nome.Length} caracteres.");
    Console.ReadKey();
}
```

Console.WriteLine("mensagem") → Exibe mensagem

Console.ReadLine() → Lê os caracteres digitados para uma variável

Console.ReadKey() → Aguarda uma digitação para encerrar a execução

\$ → Combinado com aspas duplas escreve uma mensagem reservando espaço para uma variável caso chaves sejam inseridas.

- Execute para testar



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

Acrescente a programação a seguir no corpo da classe após o método ReadKey() e execute novamente.

```
Console.WriteLine("Digite a data do seu nascimento: ");
DateTime dtNascimento = DateTime.Parse(Console.ReadLine());

int qtdDiasVividos = DateTime.Now.Subtract(dtNascimento).Days;
Console.WriteLine("Os dias vividos até hoje são: " + qtdDiasVividos);

Console.ReadKey();
```



Aula 01 - Trabalhando com variáveis

Fechе o projeto anterior através do menu File → Close Folder. Retorne ao *explorer* para e crie uma nova pasta chamada **Aula01Variaveis**. Abra a pasta criada no VS Code através do Menu File → Open Folder.

1. Realize a programação abaixo aprendizado de concatenações de variáveis string, datetime e decimal, compilando e executando logo após para testar.

```
namespace Aula02Exemplos
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Digite seu nome: ");
            string nome = Console.ReadLine();

            string frase1 = $"Olá {nome}, hoje é {DateTime.Now}";
            Console.WriteLine(frase1);
            Console.WriteLine(" ");

            Console.WriteLine("Quanto custa um dólar em reais? ");
            decimal valorDolarReais = decimal.Parse(Console.ReadLine());
            string frase2 = string.Format("Hoje é {0:ddd}, o dólar está custando {1:c2} ", DateTime.Now, valorDolarReais);
            Console.WriteLine(frase2);

            Console.WriteLine(" ");
            string cabecalho = string.Format("{0:ddd}, {0:dd} de {0:MMMM} de {0:yyyy} - {0:HH:mm:ss}", DateTime.Now);
            Console.WriteLine(cabecalho.ToUpper());
        }
    }
}
```

2. Podemos criar **Métodos** para programar as operações em um bloco isolado e fazer a chamada desse bloco no método principal. O método se chamará *ConcatenarPalavras*, conforme abaixo

```
static void Main(string[] args)
{
    ConcatenarPalavras();
}

1 reference
public static void ConcatenarPalavras()
{
    Console.WriteLine("Digite seu nome: ");
    string nome = Console.ReadLine();

    string frase1 = $"Olá {nome}, hoje é {DateTime.Now}";
    Console.WriteLine(frase1);
    Console.WriteLine(" ");

    Console.WriteLine("Quanto custa um dólar em reais? ");
    decimal valorDolarReais = decimal.Parse(Console.ReadLine());
    string frase2 = string.Format("Hoje é {0:ddd}, o dólar está custando {1:c2} ", DateTime.Now, valorDolarReais);
    Console.WriteLine(frase2);

    Console.WriteLine(" ");
    string cabecalho = string.Format("{0:ddd}, {0:dd} de {0:MMMM} de {0:yyyy} - {0:HH:mm:ss}", DateTime.Now);
    Console.WriteLine(cabecalho.ToUpper());
}
```



3. Crie mais um método, imediatamente abaixo do fechamento do método anterior. Esse método poderá verificar se a data digitada é um final de semana ou não.

```
public static void VerificarAulaEtec()
{
    Console.WriteLine("Digite a data");
    DateTime data = DateTime.Parse(Console.ReadLine());

    if(data.DayOfWeek == DayOfWeek.Saturday || data.DayOfWeek == DayOfWeek.Sunday)
    {
        Console.WriteLine("Final de semana! Hoje não tem aula! Revisarei exercícios.");
    }
    else
    {
        Console.WriteLine("Dia da semana! Bora pra Etec!");
    }
}
```

4. Faça a chamada do método recém-criado no método principal e realize o teste.

```
static void Main(string[] args)
{
    ConcatenarPalavras();
    VerificarAulaEtec();
}
```

5. Crie o método para calcular média

```
public static void CalcularMedia()
{
    Console.WriteLine("Digite a primeira nota");
    decimal nota1 = decimal.Parse(Console.ReadLine());

    Console.WriteLine("Digite a segunda nota");
    decimal nota2 = decimal.Parse(Console.ReadLine());

    decimal media = (nota1 + nota2) / 2;
    Console.WriteLine($"A média é {media}");

    if(media >= 7)
        Console.WriteLine("Aprovado");
    else if(media < 7 && media >= 4 )
        Console.WriteLine("Recuperação");
    else
        Console.WriteLine("Reprovado");
}
```

6. Como já sabemos que podemos chamar vários métodos no método principal, fazemos a chamada apenas deste método. Até o final da aula conseguiremos criar uma maneira de escolher qual método vamos usar. Execute para testar

```
static void Main(string[] args)
{
    CalcularMedia();
}
```



7. Crie o método para Calcular a Tabuada

```
public static void CalcularTabuada()
{
    Console.WriteLine("Digite a tabuada que deseja calcular");
    int tabuada = int.Parse(Console.ReadLine());
    int contador = 0;

    while(contador <= 10)
    {
        string mensagem = string.Format("{0} X {1} = {2}", tabuada, contador, tabuada * contador);
        Console.WriteLine(mensagem);
        contador++;
    }
}
```

8. Editaremos o método principal para poder escolher qual método vamos querer usar

```
static void Main(string[] args)
{
    Console.WriteLine("Observe o menu abaixo e digite o número referente a opção desejada: ");
    Console.WriteLine("1 - Concatenar Palavras");
    Console.WriteLine("2 - Verificar Dia da Semana");
    Console.WriteLine("3 - Calcular Média");
    Console.WriteLine("4 - Calcular Tabuada");

    int opcaoEscolhida = int.Parse(Console.ReadLine());

    switch (opcaoEscolhida)
    {
        case 1:
            ConcatenarPalavras();
            break;
        case 2:
            VerificarAulaEtec();
            break;
        case 3:
            CalcularMedia();
            break;
        case 4:
            CalcularTabuada();
            break;
        default:
            Console.WriteLine("Opção Inválida");
            break;
    }
}
```



Exercício

Temos dois métodos a serem programados. Utilize o programado durante a aula 02 e siga os detalhes:

- Na entrega deve constar quatro prints no anexo desta atividade: Referente ao código de cada método e a execução de cada método.
- Insira as duas opções de método no Menu do programa seguindo o exemplo dos que já existem.

As tarefas são:

- Criar um método chamado DetalharData que apresente a mensagem para o usuário digitar uma data e armazene em uma variável compatível. Você deverá usar formatação para imprimir na tela qual o dia da Semana (Segunda, Terça...etc) e qual o nome do mês por extenso (Janeiro, Fevereiro etc). Comparar o dia da semana da data, se ela for um domingo, imprimir na tela a hora atual com os minutos (Ex: 14:35)
- Criar um método chamado CalcularDescontoINSS que apresente a mensagem que solicite para o usuário digitar o valor do salário e armazene em uma variável compatível. Você deverá observar a tabela em anexo e apresentar ao usuário o valor de INSS que ele terá que pagar e qual o valor do salário descontado o INSS. Você pode declarar a quantidade de variáveis que quiser para fazer os cálculos.

Novos valores pagos ao INSS em 2022

Valor de contribuição ao INSS
a partir de fevereiro de 2022

2022

Até 1 salário mínimo **7,5%**
(R\$ 1.212)

De R\$ 1.212,01 a **9%**
R\$ 2.427,35

De R\$ 2.427,36 a **12%**
R\$ 3.641,03

De R\$ 3.641,04 a **14%**
R\$ 7.087,22

Fonte: <https://g1.globo.com/economia/noticia/2022/01/20/valores-das-contribuicoes-ao-inss-mudam-a-partir-de-fevereiro-entenda.ghtml>, acessado em 11/02/2022 às 8h



Resoluções

```
public static void DetalharData()
{
    Console.WriteLine("Digite a data desejada");
    DateTime data = DateTime.Parse(Console.ReadLine());

    if (data.DayOfWeek == DayOfWeek.Sunday)
    {
        string frase = string.Format("{0:dddd}, {0:dd} de {0:MMMM} de {0:yyyy} - {0:HH:mm:ss}", data);
        Console.WriteLine(frase);
    }
    else
    {
        string frase = string.Format("{0:dddd}, {0:dd} de {0:MMMM} de {0:yyyy}", data);
        Console.WriteLine(frase);
    }
}
```

```
public static void CalcularDescontoINSS()
{
    Console.WriteLine("Digite o valor do salário");
    double salario = double.Parse(Console.ReadLine());
    double desconto = 0;
    double salarioDescontado = 0;

    if(salario < 1212)
    {
        desconto = salario * 0.075;
        salarioDescontado = salario - desconto;
    }
    else if(salario > 1212.01 && salario < 2427.35)
    {
        desconto = salario * 0.09;
        salarioDescontado = salario - desconto;
    }
    else if(salario > 2427.36 && salario < 3641.03)
    {
        desconto = salario * 0.12;
        salarioDescontado = salario - desconto;
    }
    else
    {
        desconto = salario * 0.14;
        salarioDescontado = salario - desconto;
    }

    string frase = string.Format("O valor do desconto é {0:c2}. O salário descontado o INSS é {1:c2}", desconto, salarioDescontado);
    Console.WriteLine(frase);
}
```



Método Main

```
static void Main(string[] args)
{
    int opcaoEscolhida = 0;

    do
    {
        Console.WriteLine("=====");
        Console.WriteLine("***** Exercícios - Aula 02 *****");
        Console.WriteLine("=====");
        Console.WriteLine("Observe o menu abaixo e digite o número referente a opção desejada: ");
        Console.WriteLine("1 - Concatenar Palavras");
        Console.WriteLine("2 - Verificar Dia da Semana");
        Console.WriteLine("3 - Calcular Média");
        Console.WriteLine("4 - Calcular Tabuada");
        Console.WriteLine("5 - Detalhar Data");
        Console.WriteLine("6 - Calcular Descontos do INSS");
        Console.WriteLine("=====");
        Console.WriteLine("-----Ou tecle qualquer outro número para sair -----");
        Console.WriteLine("=====");

        opcaoEscolhida = int.Parse(Console.ReadLine());

        switch (opcaoEscolhida)
        {
            case 1:
                ConcatenarPalavras();
                break;
            case 2:
                VerificarAulaEtec();
                break;
            case 3:
                CalcularMedia();
                break;
            case 4:
                CalcularTabuada();
                break;
            case 5:
                DetalharData();
                break;
            case 6:
                CalcularDescontoINSS();
                break;
            default:
                Console.WriteLine("Saindo do sistema....");
                break;
        }
    } while (opcaoEscolhida >= 1 && opcaoEscolhida <= 6);

    Console.WriteLine("=====");
    Console.WriteLine("***** Obrigado por utilizar o sistema e volte sempre *****");
    Console.WriteLine("=====");
    Console.WriteLine("");Console.WriteLine("");
}
```



Aula 02 – Classes e Enumerações – Introdução ao Padrão MVC (Model-View-Controller)

Nesta aula faremos a interpretação de diagramas de classe UML para construção de Classes e Enumeradores, programando as características de uma classe (atributos/métodos) e as operações (Métodos). Junto a isso, ao criar as classes, separaremos as mesmas por tipo, buscando uma melhor organização do projeto de acordo com as responsabilidades de cada classe. Esse modelo é conhecido como MVC – Model-View-Controller.

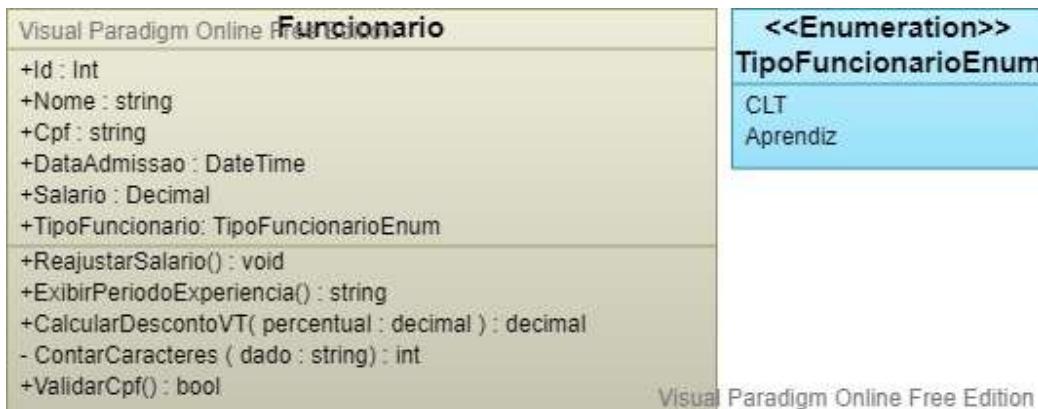
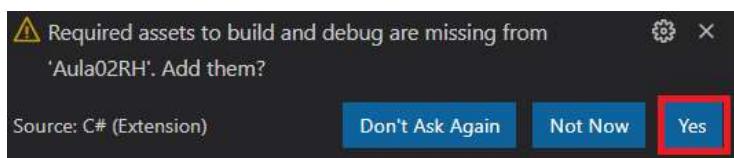


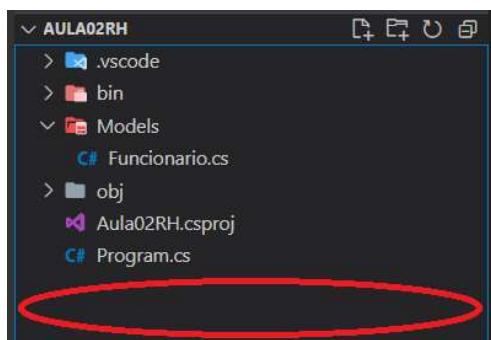
Diagrama de classe do padrão UML

Uma classe é a representação de uma entidade do mundo real que viabilizará a construção de projetos

1. Crie a pasta Aula02RH na pasta em que costuma criar os projetos, abra a mesma no VS Code, ative o terminal e execute o comando para criação de um projeto do tipo Console Application.
2. Clique na classe Program, aguarde até aparecer a notificação para ativação da extensão do C# e clique em “Yes”



3. Clique com o direito do mouse dentro do retângulo de borda azul fina, abaixo da classe Program e escolha a opção New Folder, nomeando a pasta como **Models**.





4. Clique com o direito do mouse na pasta Models e crie uma classe chamada **Funcionario**, realizando a programação das propriedades a seguir.

Dica: Digite “Prop” e pressione TAB para o atalho de criação de propriedades. O Curso automaticamente selecionará o tipo de variável para que você mantenha ou digite outra, após isso clique em TAB mais uma vez para o cursor selecionar o nome da propriedade para que você insira conforme o diagrama.

```
namespace Aula02RH.Models
{
    0 references
    public class Funcionario
    {
        0 references
        public int Id { get; set; }
        0 references
        public string Nome { get; set; }
        0 references
        public string Cpf { get; set; }
        0 references
        public DateTime DataAdmissao { get; set; }
        0 references
        public decimal Salario { get; set; }

    }
}
```

5. Clique com o direito na pasta Models e crie uma pasta chamada **Enums**. Clique com o direito na pasta Enums e crie uma classe chamada **TipoFuncionarioEnum**, fazendo as modificações abaixo

```
namespace Aula02RH.Models.Enums
{
    1 reference
    public enum TipoFuncionarioEnum
    {
        0 references
        CLT=1,
        0 references
        Aprendiz=2
    }
}
```

- O tipo de arquivo foi modificado para ser um enum (enumeração). Usamos esse tipo quando temos dados que não se alteram facilmente e pode ser identificado por um número. Mais a frente isso fará muito sentido na hora de fazer comparações.



6. Volte à classe Funcionario e declare a enumeração criada na forma de propriedade. Perceba que se você digitou o tipo de maneira correta (TipoFuncionarioEnum) ele não vai reconhecer, pois os arquivos estão em pastas diferentes, então será necessário indicar qual o caminho em que a enumeração está. Você pode (1) clicar na lâmpada que aparece ao lado ou (2) usar o atalho CTRL + . (ponto) para fazer a janela de resolução de erros aparecer. Escolha o using da seta e verá que no topo da classe será adicionada uma referência ao endereço da enumeração.

```
public class Funcionario
{
    0 references
    public int Id { get; set; }
    0 references
    public string Nome { get; set; }
    0 references
    public string Cpf { get; set; }
    0 references
    public DateTime DataAdmissao { get; set; }
    0 references
    public decimal Salario { get; set; }
    0 references
    public TipoFuncionarioEnum TipoFuncionario{ get; set; }

    Initialize ctor from properties...
}

using Aula02RH.Models.Enuns; ← Red arrow points here
Enuns.TipoFuncionarioEnum
Gerar tipo 'TipoFuncionarioEnum' -> Gerar class 'TipoFuncionarioEnum' no novo arquivo
```

7. Programe os métodos da classe antes do fechamento do corpo dela, conforme abaixo

```
0 references
public TipoFuncionarioEnum TipoFuncionario{ get; set; }
0 references
public void ReajustarSalario()
{
    Salario = Salario + (Salario * 10 /100);
}
0 references
public string ExibirPeriodoExperiencia()
{
    string periodoExperiencia =
        string.Format("Períodos de Experiência: {0} até {1}", DataAdmissao, DataAdmissao.AddMonths(3));
    return periodoExperiencia;
}

0 references
public decimal CalcularDescontoVT(decimal percentual)
{
    decimal desconto = this.Salario * percentual/100;
    return desconto;
}
```



```
private int ContarCaracteres(string dado)
{
    return dado.Length;
}

0 references
public bool ValidarCpf()
{
    if(ContarCaracteres(Cpf) == 11)
        return true;
    else
        return false;
}
```

- Salve e execute o comando build no terminal para confirmar que não existe erros no código até aqui
- 8. No método principal da classe program iremos fazer a criação de uma cópia da classe em memória, o que chamamos de instância, para pode alimentar ela com dados próprios. Será necessário fazer o using para reconhecer a namespace Models clicando em cima do erro e usando o atalho CTRL + . ou clicando na lâmpada.

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        A Funcionario func = new Funcionario();
    }
}
```

- (A) Declaração de uma variável do tipo **Funcionario** com o nome de **func**, representada pela classe Funcionario.
- (B) Atribuição para func de uma cópia criada em memória, através do operador new, desta maneira, func passa a ser um objeto, do tipo Funcionario como mencionado anteriormente.



9. Agora será possível fazer a alimentação de cada propriedade presente na classe Funcionario através do objeto func e fazer a chamada para os métodos contidos na classe

The screenshot shows a code editor with the following C# code:

```
6  class Program
7  {
8      static void Main(string[] args)
9      {
10         Funcionario func = new Funcionario();
11
12         A func.Id = 10;
13         func.Nome = "Neymar";
14         func.Cpf = "12345678910";
15         func.DataAdmissao = DateTime.Parse("01/01/2000");
16         func.Salario = 10000.00M; C
17         func.TipoFuncionario = Models.Enuns.TipoFuncionarioEnum.CLT;
18
19         string mensagem = func.ExibirPeriodoExperiencia(); B
20         Console.WriteLine("=====");
21         Console.WriteLine(mensagem);
22         Console.WriteLine("=====");
23     }
}
```

Annotations in the code:

- A**: Surrounds the assignment of properties to the object.
- B**: Surrounds the call to the `ExibirPeriodoExperiencia()` method.
- C**: Surrounds the assignment of the `Salario` property.

Below the code editor, there is a terminal window showing the output of the application:

```
PS D:\Work\ETEC\DS2022-1\Concluido\Aula02RH> dotnet run
=====
Períodos de Experiência: 01/01/2000 00:00:00 até 01/04/2000 00:00:00
=====
PS D:\Work\ETEC\DS2022-1\Concluido\Aula02RH>
```

- (A) Inserção de dados nas propriedades do objeto func que é do tipo Funcionario
(B) Chamada do método para exibir o período de experiência.
(C) Perceba que a possilita escolher apenas itens contidos dentro do arquivo dela.

Mas se quisermos fazer com que as propriedades sejam alimentadas com itens que venham da tela, como podemos fazer?



10. Siga a programação abaixo para alimentar as propriedades através da digitação no console

```
Funcionario func = new Funcionario();

Console.WriteLine("Digite o Id do funcionário: ");
func.Id = int.Parse(Console.ReadLine());

Console.WriteLine("Digite o nome do funcionário: ");
func.Nome = Console.ReadLine();

Console.WriteLine("Digite o CPF: ");
func.Cpf = Console.ReadLine();

Console.WriteLine("Digite a data de Admissão: ");
func.DataAdmissao = DateTime.Parse(Console.ReadLine());

Console.WriteLine("Digite o Salário: ");
func.Salario = decimal.Parse(Console.ReadLine());

Console.WriteLine("Escolha o tipo de Funcionário (1 - CLT / 2 - Aprendiz): ");
int opcao = int.Parse(Console.ReadLine());

//Operador Ternário - Interpretação: Se a condição do parenteses for verdadeira,
//escolhe o que está depois da "?", Caso contrário, escolhe o que está de pois dos ":".
func.TipoFuncionario = (opcao == 1) ? TipoFuncionarioEnum.CLT : TipoFuncionarioEnum.Aprendiz;

func.ReajustarSalario();
decimal valorDescontoVT = func.CalcularDescontoVT(6);

Console.WriteLine("=====");
Console.WriteLine($"O salário reajusta do é {func.Salario}.\n");
Console.WriteLine($"O Desconto do VT é {valorDescontoVT}.\n");
Console.WriteLine("=====");
```

- Execute o programa para testar as condições:

Atividades:

- (1) Utilize o programa desenvolvido até aqui para apresentar os valores/informações da execução dos demais métodos. Dica: programar abaixo do operador ternário.
- (2) Crie uma forma do usuário escolher qual método ele quer que tenha a informação exibida em tela, após a digitação dos dados ter sido feita.



Aula 3 – Coleções e lista de tipos

Uma lista é uma coleção de objetos que pode ser desde uma lista de números inteiros, uma lista de strings ou uma lista de objetos da classe Personagem, por exemplo. A sintaxe de criação de uma lista está exemplificada conforme o exemplo abaixo:

É possível realizar diversas operações com lista, como busca, soma, adição de itens, remoção. Aprenderemos a usar aos poucos estas funcionalidades.

1. Crie uma pasta chamada **Aula03Colecoes**, abra o VS Code nesta pasta, exiba o terminal e crie um projeto do tipo console. Utilize o *explorer* para copiar a pasta *Models* do projeto Aula02RH para dentro da pasta do projeto atual e confirme que você conseguirá visualizar ela através do VS Code. Altere o namespace para conter Aula03Colecoes ou invés de Aula02RH
2. Declare uma lista do tipo *Funcionario* de maneira global, ou seja, fora de qualquer método, mas dentro da classe. A palavra *List* necessitará do using de *System.Collections.Generic* e *Funcionario* do using do namespace onde a classe está.

```
using System;
using System.Collections.Generic;
using Aula03Colecoes.Models;
using Aula03Colecoes.Models.Enums;

namespace Aula03Colecoes
{
    0 references
    class Program
    {
        0 references
        static List<Funcionario> lista = new List<Funcionario>();

        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
        }
    }
}
```



3. crie um método que vai inserir elementos na lista e faça a chamada dele no método principal.

```
static void Main(string[] args)
{
    CriarLista();
}

1 reference
public static void CriarLista()
{
    Funcionario f1 = new Funcionario();
    f1.Id = 1;
    f1.Nome = "Neymar";
    f1.Cpf = "12345678910";
    f1.DataAdmissao = DateTime.Parse("01/01/2008");
    f1.Salario = 100.000M;
    f1.TipoFuncionario = TipoFuncionarioEnum.CLT;
    lista.Add(f1);
}
```

- Adicione mais funcionários a esta lista conforme os objetos abaixo. Não esqueça que os objetos devem estar dentro do método

```
Funcionario f2 = new Funcionario();
f2.Id = 2;
f2.Nome = "Cristiano Ronaldo";
f2.Cpf = "01987654321";
f2.DataAdmissao = DateTime.Parse("30/06/2002");
f2.Salario = 150.000M;
f2.TipoFuncionario = TipoFuncionarioEnum.CLT;
lista.Add(f2);

Funcionario f3 = new Funcionario();
f3.Id = 3;
f3.Nome = "Messi";
f3.Cpf = "135792468";
f3.DataAdmissao = DateTime.Parse("01/11/2003");
f3.Salario = 70.000M;
f3.TipoFuncionario = TipoFuncionarioEnum.Aprendiz;
lista.Add(f3);
```



```
Funcionario f4 = new Funcionario();
f4.Id = 4;
f4.Nome = "Mbappe";
f4.Cpf = "246813579";
f4.DataAdmissao = DateTime.Parse("15/09/2005");
f4.Salario = 80.000M;
f4.TipoFuncionario = TipoFuncionarioEnum.Aprendiz;
lista.Add(f4);

Funcionario f5 = new Funcionario();
f5.Id = 5;
f5.Nome = "Lewa";
f5.Cpf = "246813579";
f5.DataAdmissao = DateTime.Parse("20/10/1998");
f5.Salario = 90.000M;
f5.TipoFuncionario = TipoFuncionarioEnum.Aprendiz;
lista.Add(f5);

Funcionario f6 = new Funcionario();
f6.Id = 6;
f6.Nome = "Roger Guedes";
f6.Cpf = "246813579";
f6.DataAdmissao = DateTime.Parse("13/12/1997");
f6.Salario = 300.000M;
f6.TipoFuncionario = TipoFuncionarioEnum.CLT;
lista.Add(f6);
```

4. Crie outro método que será responsável por exibir os dados da lista. Faça a chamada do método no método main e teste o código.

```
public static void ExibirLista()
{
    string dados = "";
    for (int i = 0; i < lista.Count; i++)
    {
        dados += "=====\\n";
        dados += string.Format("Id: {0} \\n", lista[i].Id);
        dados += string.Format("Nome: {0} \\n", lista[i].Nome);
        dados += string.Format("CPF: {0} \\n", lista[i].Cpf);
        dados += string.Format("Admissão: {0:dd/MM/yyyy} \\n", lista[i].DataAdmissao);
        dados += string.Format("Salário: {0:c2} \\n", lista[i].Salario);
        dados += string.Format("Tipo: {0} \\n", lista[i].TipoFuncionario);
        dados += "=====\\n";
    }
    Console.WriteLine(dados);
}
```



-
5. Programa o método para realizar uma busca na lista através de uma propriedade do objeto Funcionario

```
public static void ObterPorId()
{
    lista = lista.FindAll(x => x.Id == 1);
```

6. Crie um método para adicionar um elemento na lista

```
public static void AdicionarItem()
{
    Funcionario fNovo = new Funcionario();
    fNovo.Id = 9;
    fNovo.Nome = "Ronaldo";
    fNovo.Cpf = "11111111110";
    fNovo.DataAdmissao = DateTime.Parse("17/05/1997");
    fNovo.Salario = 300.000M;
    fNovo.TipoFuncionario = TipoFuncionarioEnum.CLT;

    lista.Add(fNovo);

    ExibirLista();
}
```

Outros exemplos de Métodos usando listas

- Ordenando uma lista por critério. Será necessário o using System.Linq na sintaxe `OrderBy`

```
public static void Ordenar()
{
    lista = lista.OrderBy(x => x.Nome).ToList();
    ExibirLista();
}
```

- Contar Itens de uma lista

```
public static void ContarFuncionarios()
{
    int qtd = lista.Count();
    Console.WriteLine($"Existem {qtd} funcionários.");
}
```



- Somando valores da propriedade comum entre objetos de uma lista

```
public static void SomarSalarios()
{
    decimal somatorio = lista.Sum(x=> x.Salario);
    Console.WriteLine(string.Format("A soma dos salários é {0:c2}.", somatorio));
}
```

- Filtrando dados de uma lista de acordo com critérios

```
public static void ExibirAprendizes()
{
    lista = lista.FindAll( x => x.TipoFuncionario == TipoFuncionarioEnum.Aprendiz);
    ExibirLista();
}
```

- Busca por nome aproximado. *ToLower* transforma os caracteres em minúsculo para não termos problemas de distinção. O contrário disso seria o *ToUpper*

```
public static void BuscarPorNomeAproximado()
{
    AdicionarItem();

    lista = lista.FindAll( x => x.Nome.ToLower().Contains("ronaldo"));

    ExibirLista();
}
```

- Filtrando um personagem por algum critério e removendo o mesmo da lista

```
public static void BuscarPorCpfRemover()
{
    Funcionario fBusca = lista.Find( x => x.Cpf == "01987654321");
    lista.Remove(fBusca);
    Console.WriteLine($"Personagem removido: {fBusca.Nome}\nLista Atualizada: \n");

    ExibirLista();
}
```



- Removendo da lista de acordo com filtragem de Ids

```
public static void RemoverIdMenor4()
{
    lista.RemoveAll( x => x.Id < 4);
    ExibirLista();
}
```

- Dada a lista inicial, será feita uma busca através de um Id digitado

```
static void Main(string[] args)
{
    CriarLista();

    Console.WriteLine("Digite o Id do funcionário que você quer buscar");
    int idDigitado = int.Parse(Console.ReadLine());

    ObterPorId(idDigitado);
}

1 reference
public static void ObterPorId(int id)
{
    Funcionario fBusca = lista.Find( x => x.Id == id);

    Console.WriteLine($"Personagem encontrado: {fBusca.Nome}");
}
```

- Buscando os funcionários com salário acima de um valor digitado.

```
static void Main(string[] args)
{
    CriarLista();

    Console.WriteLine("Digite o salário para buscar todos acima deste valor: ");
    decimal salarioDigitado = decimal.Parse(Console.ReadLine());

    ObterPorSalario(salarioDigitado);
}

1 reference
public static void ObterPorSalario(decimal valor)
{
    lista = lista.FindAll( x => x.Salario >= valor);
    ExibirLista();
}
```



Exemplo de método adição de item com validação das propriedades

- Crie o método abaixo. Perceba que o retorno dele é booleano

```
public static bool AdicionarFuncionario(Funcionario fNovo)
{
    if(fNovo.Salario == 0)
    {
        Console.WriteLine("Valor do salário não pode ser 0");
        return false;
    }
    else
    {
        lista.Add(fNovo);
        return true;
    }
}
```

- Faça a chamada para o método conforme abaixo

```
static void Main(string[] args)
{
    CriarLista();
    bool fAdicionado = false;

    while(fAdicionado == false)
    {
        Funcionario f = new Funcionario();

        Console.WriteLine("Digite o nome: ");
        f.Nome = Console.ReadLine();

        Console.WriteLine("Digite o salário: ");
        f.Salario = decimal.Parse(Console.ReadLine());

        Console.WriteLine("Digite a data de admissão: ");
        f.DataAdmissao = DateTime.Parse(Console.ReadLine());

        fAdicionado = AdicionarFuncionario(f);
    }
    ExibirLista();
}
```

Referências para o estudo de listas

<https://www.tutorialsteacher.com/csharp/csharp-list>

<https://www.dotnetperls.com/list>



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

Exercícios

Utilize o projeto da aula para construir os métodos abaixo testando as execuções. Consultar o Teams para realizar entrega dos prints dos métodos em funcionamento na tarefa que será criada.

- a) Método com nome **ObterPorTipo** que selecione a lista de funcionários de acordo com numeração digitada no console.
- b) Método com nome **ObterPorNome** que selecione um funcionário de acordo com o nome digitado e que caso não encontre retorne uma mensagem para o usuário.
- c) Método com nome **ObterFuncionariosRecentes** que remova os personagens com Id menor que 4 e depois faça um filtro na lista para exibi-la em ordem decrescente de **Salário**.
- d) Método com nome **ObterEstatisticas** que exiba a quantidade de personagens da lista e qual o somatório de salário dos funcionários.
- e) Método com nome **ValidarSalarioAdmissao** que não permita que um Funcionário seja adicionado com salário 0 ou data de admissão anterior a data atual. Deve ser exibida uma mensagem ao usuário caso isso aconteça.
- f) Método com nome **ValidarNome** que não permita que um funcionário tenha nome menor que 2 caracteres. Deve ser exibida uma mensagem ao usuário caso isso aconteça.

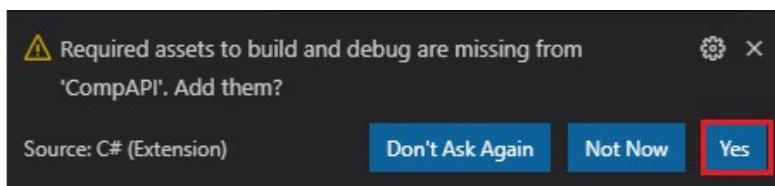


Aula 05 - Projetos WebApi

Crie uma pasta na sua organização de arquivos chamada **RpgApi** e abra ela no VS Code. Depois disso abra uma o terminal e digite o comando para criação de uma API, conforme abaixo

dotnet new WebApi --framework net5.0

Após a criação aparecerá uma mensagem para ativar o modo de depuração para o C#. Escolha sim conforme abaixo



Identificação dos arquivos no Projeto

Classe Startup: Define as configurações gerais que o projeto utilizará e será acionada pela classe Program

Classe Program: Será o ponto de partida ao rodar o projeto, como mencionado acima, nela está apontada a classe Startup.

Arquivo .csproj: Arquivo em que ficará registrado dos os pacotes baixados para utilização no projeto. Framework do banco de dados por exemplo.

Appsettings.json: Arquivo em que pode ser guardado informações de configurações, por exemplo o IP e dados de acesso de um banco de dados por exemplo.

Lauchsettings.json (pasta properties): Arquivo em que estarão informações sobre a execução do projeto, por exemplo qual o endereço que constará no navegador ao rodar a aplicação ou se utilizará o protocolo http ou https por exemplo. Neste arquivo remova o endereço *https* que aparece na propriedade applicationUrl para que o navegador não exiba mensagem de bloqueio ao executar o aplicativo

```
"CompDS": [
    "commandName": "Project",
    "launchBrowser": true,
    "launchUrl": "weatherforecast",
    "applicationUrl": "https://localhost:5001;http://localhost:5000",
    "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
    }
]
```

- Isso é necessário para que ao rodar localmente o projeto, não ocorram problemas por não existir certificado de conexão segura.



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

Abra o terminal através do menu View → Terminal, digite a linha de comando `dotnet run` para rodar o projeto. Abra o navegador, digite o endereço e porta da API e o nome Controller que temos até então:

localhost:5000/WeatherForecast

O Navegador deverá exibir dados aleatórios em Cº e Fº que se trata da avaliação de temperaturas. Execute o comando CTRL + C no Visual Studio Code para interromper a aplicação assim que desejar.

Nas próximas etapas entenderemos melhor o que é uma Controller, mas como uma breve introdução, durante a criação do projeto foi criada uma Controller chamada WeatherForecast automaticamente na pasta correspondente



Testando API com Postman

Como ainda não temos front-end (interface) ainda, precisaremos de uma ferramenta para executar testes no nosso back-end (Programação) e se mostra uma alternativa mais completa para testar todos os recursos que uma API oferece em comparação como o navegador. O Postman pode ser baixado através do endereço abaixo:

<https://www.postman.com/downloads/>

Você pode realizar o login através do gmail e manter o histórico de todos os seus testes dentro da ferramenta.

Em linhas gerais, o Postman é um API Client que podemos utilizar para realizar as requisições na API através dos principais métodos: Get, Post, Put e Delete

Execute a aplicação e realize as seguintes configurações no Postman para poder testar o método Get da API

The screenshot shows the Postman interface with the following details:

- Header Bar:** Shows "Launchpad", "GET http://localhost:5000/WeatherF...", and "No Environment".
- Request Type:** "GET" is selected.
- URL:** "http://localhost:5000/WeatherForecast" is entered in the URL field.
- Buttons:** "Send" (highlighted in blue) and "Cancel" (highlighted in red).
- Tab Bar:** "Params", "Authorization", "Headers (6)" (highlighted in green), "Body", "Pre-request Script", "Tests", and "Settings".
- Body Tab Content:** "Pretty", "Raw", "Preview", "Visualize", and "JSON" dropdown. The JSON content is:

```
1 [ {  
2   "date": "2020-08-05T23:10:46.0918336-03:00",  
3   "temperatureC": -15,  
4   "temperatureF": 6,  
5   "summary": "Chilly"  
6 },  
7   {  
8     "date": "2020-08-06T23:10:46.0931723-03:00",  
9     "temperatureC": 4,  
10    "temperatureF": 39,  
11    "summary": "Balmy"  
12 } ]
```
- Status Bar:** "Status: 200 OK Time: 331 ms Size: 645 B Save" (highlighted in green).

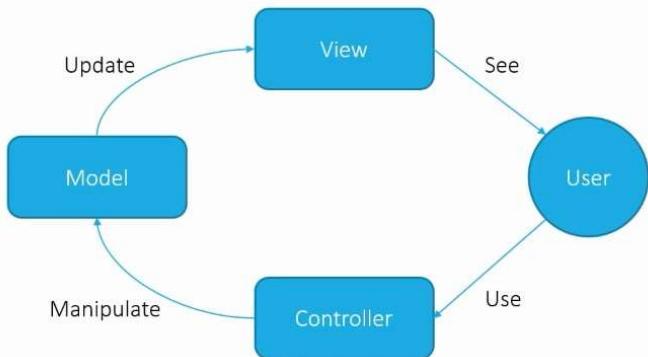
- Em vermelho temos as configurações que devem ser feitas e em verde o resultado da requisição na API.



Padrão MVC em Projetos WebAPI

- *Mapa Mental da Matéria:* <https://mm.tt/1587285354?t=PMFTNcCUTT>

Revisão do Padrão MVC



Divide a lógica de um programa em três elementos interconectados. Podemos exemplificar o padrão MVC conforme acima. O Usuário requisita dados a uma Controlador que carrega o Modelo e expõe os dados numa Visualização.

1. No projeto RpgApi, crie uma pasta chamada **Models** e dentro desta pasta crie uma classe chamada Personagem conforme abaixo. Utilize o atalho (digitando prop + TAB + TAB) para criar as propriedades.

```
namespace RpgApi.Models
{
    public class Personagem
    {
        public int Id { get; set; }
        public string Nome { get; set; }
        public int PontosVida { get; set; }
        public int Forca { get; set; }
        public int Defesa { get; set; }
        public int Inteligencia { get; set; }
    }
}
```



2. Dentro da pasta Models, crie uma outra pasta chamada **Enuns**. Clique com o botão direito na pasta **Enuns** recém-criada e adicione uma classe chamada **ClasseEnum**. Como queremos criar uma enumeração, alteraremos a palavra class por enum na assinatura da classe e acrescentaremos os itens deste enum.

```
namespace RpgApi.Models.Enuns
{
    2 references
    public enum ClasseEnum
    {
        1 reference
        Cavaleiro = 1,
        0 references
        Mago = 2,
        0 references
        Clerigo = 3
    }
}
```

3. Adicione na classe **Personagem** uma propriedade ligada a enumeração recém-criada, conforme abaixo. Será necessário fazer referência ao namespace da enumeração.

```
using RpgApi.Models.Enuns;

namespace RpgApi.Models
{
    0 references
    public class Personagem
    {
        0 references
        public int Id { get; set; }
        0 references
        public string Nome { get; set; }
        0 references
        public int PontosVida { get; set; }
        0 references
        public int Forca { get; set; }
        0 references
        public int Defesa { get; set; }
        0 references
        public int Inteligencia { get; set; }
        0 references
        public ClasseEnum Classe{ get; set; }
    }
}
```



4. Crie uma classe chamada **PersonagensExemploController** na pasta *Controllers* e codifique conforme abaixo. A seguir informaremos a função de cada trecho de código.

```
using Microsoft.AspNetCore.Mvc; D

namespace RpgApi.Controllers
{
    [ApiController] A
    [Route("[Controller]")] B
    0 references
    public class PersonagemExemploController : ControllerBase C
    {
        //Toda codificação será feita aqui
    }
}
```

- a) Atributo *ApiController* usado porque é uma API de requisição *http*
 - b) Atributo *Route* usado para definir rota a ser digitada no navegador para chamar a *controller* e o Método
 - c) Toda classe *Controller* herdará da *ControllerBase*.
 - d) Os atributos *apicontroller*, *Route* e *ControllerBase* necessita do *using* de *Microsoft.AspNetCore.Mvc* que é o Framework utilizado.
5. Programa a criação de uma lista do tipo Personagem dentro da classe controller de maneira global. O *usings* necessários serão *System.Collections.Generic*, *RpgApi.Models* e *RpgApi.Enums*.

```
public class PersonagemExemploController : ControllerBase
{
    //Criação de uma lista já adicionando os objetos dentro.
    0 references
    private static List<Personagem> personagens = new List<Personagem>() {
        new Personagem() { Id = 1, Nome = "Frodo", PontosVida=100, Forca=17, Defesa=23, Inteligencia=33, Classe=ClasseEnum.Cavaleiro },
        new Personagem() { Id = 2, Nome = "Sam", PontosVida=100, Forca=15, Defesa=25, Inteligencia=30, Classe=ClasseEnum.Cavaleiro },
        new Personagem() { Id = 3, Nome = "Galadriel", PontosVida=100, Forca=18, Defesa=21, Inteligencia=35, Classe=ClasseEnum.Clerigo },
        new Personagem() { Id = 4, Nome = "Gandalf", PontosVida=100, Forca=18, Defesa=18, Inteligencia=37, Classe=ClasseEnum.Mago },
        new Personagem() { Id = 5, Nome = "Hobbit", PontosVida=100, Forca=20, Defesa=17, Inteligencia=31, Classe=ClasseEnum.Cavaleiro },
        new Personagem() { Id = 6, Nome = "Celeborn", PontosVida=100, Forca=21, Defesa=13, Inteligencia=34, Classe=ClasseEnum.Clerigo },
        new Personagem() { Id = 7, Nome = "Radagast", PontosVida=100, Forca=25, Defesa=11, Inteligencia=35, Classe=ClasseEnum.Mago }
    };
}
```



6. Crie um método de nome GetFirst listar o primeiro personagem

```
public IActionResult GetFirst()
{
    Personagem p = personagens[0];
    return Ok(p);
}
```

- Método Get que retornará Ok (Status http 200) e os dados contidos no objeto p.

7. Execute a aplicação (*dotnet run*) e faça a chamada no navegador ou postman para o método Get do endereço representado a seguir

The screenshot shows a Postman interface with the following details:

- Method: GET
- URL: http://localhost:5000/PersonagemExemplo
- Body tab selected
- Content type: raw
- JSON selected in the response dropdown
- Response status: Status: 200 OK Time: 21 ms Size: 240 B

Como vimos na aula sobre lista, uma lista é uma coleção de objetos que pode ser desde uma lista de números inteiros, uma lista de strings ou uma lista de objetos da classe Personagem, por exemplo. É possível realizar diversas operações com lista, como busca, soma, adição de itens, remoção. Aprenderemos a usar aos poucos estas funcionalidades.

8. Crie um método Get para que ele possa exibir a lista.

```
public IActionResult Get()
{
    return Ok(personagens);
}
```

- Execute para testar a exibição
- Execute e tente realizar o get no *postman*. Você perceberá que retornará um erro pois existem dois métodos do tipo *HttpGet* e precisaremos diferenciar a rota deles.



9. Para resolver o problema descrito na etapa anterior, determinaremos uma rota para o método **Get** ter uma nomenclatura própria.

```
[HttpGet("GetAll")]
0 references
public IActionResult Get()
{
    return Ok(personagens);
}
```

- Execute novamente chamando um dos métodos como antes e o outro com a rota abaixo

GET Send

10. Crie um método *GetSingle* para que aceite um parâmetro pela rota. Esse parâmetro será usado como critério para fazer uma busca na lista.

```
[HttpGet("{id}")]
0 references
public IActionResult GetSingle(int id)
{
    return Ok(personagens.FirstOrDefault(pe => pe.Id == id));
}
```

- Métodos de operações em lista, como o *FirstOrDefault*, exigem o *using System.Linq*

Faça o teste no *postman*:

GET Send



Métodos do tipo Post

Os métodos *Post* são responsáveis por enviar dados para um servidor via corpo da requisição, logo é possível enviar objetos com suas propriedades totalmente preenchidas para que uma operação seja realizada, por exemplo, o salvamento numa base de dados ou adição em uma lista.

11. Crie um método do tipo *Post* conforme abaixo

```
[HttpPost]
0 references
public IActionResult AddPersonagem(Personagem novoPersonagem)
{
    personagens.Add(novoPersonagem);
    return Ok(personagens);
}
```

- Perceba que o objeto está preenchendo sendo adicionado a lista e ela está sendo retornada para o servidor.

12. Configure o *postman* para o teste do método *Post* e depois clique em *Send*:

The screenshot shows the Postman interface with the following details:

- Method: POST (highlighted with a red box)
- URL: http://localhost:5000/PersonagemExemplo (highlighted with a red box)
- Body tab selected (highlighted with a red box)
- JSON selected under the Body dropdown (highlighted with a red box)
- Request body content:

```
1 {
2     "Id": 8,
3     "nome": "Percival",
4     "pontosVida": 100,
5     "forca": 90,
6     "defesa": 50,
7     "inteligencia": 70,
8     "classe": 1
9 }
```
- Send button highlighted with a red box

- O resultado deverá ser a exibição da lista de personagens, contendo o recém adicionado por você.
- No exemplo acima poderíamos ter adicionado as demais propriedades do objeto personagem. Você pode fazer isso para fins de teste, não esquecendo de separar cada propriedade com vírgula.



Exemplos de Métodos usando listas – Aplicar na Controller PersonagemExemploController

- Ordenando uma lista por critério

```
[HttpGet("GetOrdenado")]
0 references
public IActionResult GetOrdem()
{
    List<Personagem> listaFinal = personagens.OrderBy(p => p.Forca).ToList();
    return Ok(listaFinal);
}
```

- Contar Itens de uma lista

```
[HttpGet("GetContagem")]
0 references
public IActionResult GetQuantidade()
{
    return Ok("Quantidade de personagens: " + personagens.Count);
}
```

- Somando valores da propriedade comum entre objetos de uma lista

```
[HttpGet("GetSomaForca")]
0 references
public IActionResult GetSomaForca()
{
    return Ok(personagens.Sum(p => p.Forca));
}
```

- Filtrando dados de uma lista de acordo com critérios

```
[HttpGet("GetSemCavaleiro")]
0 references
public IActionResult GetSemCavaleiro()
{
    List<Personagem> listaBusca = personagens.FindAll(p => p.Classe != ClasseEnum.Cavaleiro);
    return Ok(listaBusca);
}
```

- Busca por nome aproximado

```
[HttpGet("GetByNomeAproximado/{nome}")]
0 references
public IActionResult GetByNomeAproximado(string nome)
{
    List<Personagem> listaBusca = personagens.FindAll(p => p.Nome.Contains(nome));
    return Ok(listaBusca);
}
```



- Filtrando um personagem por algum critério e removendo o mesmo da lista

```
[HttpGet("GetRemovendoMago")]
0 references
public IActionResult GetRemovendoMagos()
{
    Personagem pRemove = personagens.Find(p => p.Classe == ClasseEnum.Mago);
    personagens.Remove(pRemove);
    return Ok("Personagem removido: " + pRemove.Nome);
}
```

- Filtro pela força

```
[HttpGet("GetByForca/{forca}")]
0 references
public IActionResult Get(int forca)
{
    List<Personagem> listaFinal = personagens.FindAll(p => p.Forca == forca);
    return Ok(listaFinal);
}
```

Exemplo de método Post com validação das propriedades

```
[HttpPost]
0 references
public IActionResult AddPersonagem(Personagem novoPersonagem)
{
    personagens.Add(novoPersonagem);

    if(novoPersonagem.Inteligencia == 0)
        return BadRequest("Inteligência não pode ter o valor igual a 0.");

    return Ok(personagens);
}
```

Informação adicional: Outra forma de executar o projeto, além do comando `dotnet run` é utilizar o comando play.

 (CTRL + Shift + D) → 

Referências para o estudo de listas

<https://www.tutorialsteacher.com/csharp/csharp-list>

<https://www.dotnetperls.com/list>



Aula 06 - Métodos Put e Delete

Método HttpPut

O Método *HttpPut* é utilizado quando queremos fazer uma alteração ou modificar um objeto existente. Para realizar esta alteração teremos que ter um identificador dentro do objeto preenchido para que possamos atualizá-lo. Geralmente utilizamos o Id para isso. Como estamos trabalhando com lista, a nossa lógica será encontrar o objeto através do Id, alterar as propriedades do objeto e depois exibir a lista novamente, conferindo se a atualização foi realizada.

1. Faça a criação do método *HttpPut* conforme a seguir

```
[HttpPost]
0 references
public IActionResult UpdatePersonagem(Personagem p)
{
    Personagem pernsagemAlterado = personagens.Find(pers => pers.Id == p.Id);
    pernsagemAlterado.Nome = p.Nome;
    pernsagemAlterado.PontosVida= p.PontosVida;
    pernsagemAlterado.Forca = p.Forca;
    pernsagemAlterado.Defesa = p.Defesa;
    pernsagemAlterado.Inteligencia = p.Inteligencia;
    pernsagemAlterado.Classe = p.Classe;

    return Ok(personagens);
}
```

2. Realize o teste no *postman* configurando a ferramenta conforme a seguir

The screenshot shows the Postman interface with the following configuration:

- Method: PUT (highlighted with a red box)
- URL: http://localhost:5000/Personagem/ (highlighted with a red box)
- Body tab selected (highlighted with a red box)
- JSON selected as the raw data type (highlighted with a red box)
- Raw data content:

```
1 { "id": 2,
2   "nome": "Galadriel Alterado",
3   "pontosVida": 1000,
4   "forca": 10000,
5   "defesa": 1000,
6   "inteligencia": 1000,
7   "classe": 2
8 }
9 }
```
- Send button highlighted with a red box



Método HttpDelete

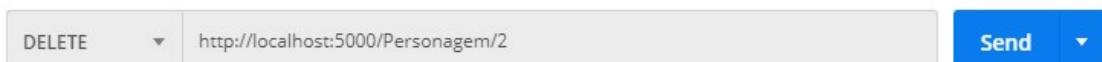
O método *HttpDelete* é utilizado para fazer a remoção de um objeto. A configuração dele necessita que informemos apenas o id no parâmetro da rota.

3. Configure o método *HttpDelete* conforme abaixo

```
[HttpDelete("{id}")]
0 references
public IActionResult Delete(int id)
{
    personagens.RemoveAll(pers => pers.Id == id);

    return Ok(personagens);
}
```

4. Realize o teste com o *postman* passando um id na rota. Verifique se o objeto com o referido id foi removido da lista.





Exercícios

Crie uma controller chamada **PersonagemExercicioController** copie a lista da controller usada em aula, conforme abaixo. E construa os seguintes métodos testando a execução no postman:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using RpgApi.Models;
using RpgApi.Models.Enums;

namespace RpgApi.Controllers
{

    [ApiController]
    [Route("[Controller]")]
    0 references
    public class PersonagemExercicioController : ControllerBase
    {
        //Criação de uma lista já adicionando os objetos dentro.
        0 references
        private static List<Personagem> personagens = new List<Personagem>()
        {
            new Personagem() { Id = 1, Nome = "Frodo", PontosVida=100, Forca=17, Defesa=23, Inteligencia=33, Classe=ClasseEnum.Cavaleiro },
            new Personagem() { Id = 2, Nome = "Sam", PontosVida=100, Forca=15, Defesa=25, Inteligencia=30, Classe=ClasseEnum.Cavaleiro },
            new Personagem() { Id = 3, Nome = "Galadriel", PontosVida=100, Forca=18, Defesa=21, Inteligencia=35, Classe=ClasseEnum.Clerigo },
            new Personagem() { Id = 4, Nome = "Gandalf", PontosVida=100, Forca=18, Defesa=18, Inteligencia=37, Classe=ClasseEnum.Mago },
            new Personagem() { Id = 5, Nome = "Hobbit", PontosVida=100, Forca=20, Defesa=17, Inteligencia=31, Classe=ClasseEnum.Cavaleiro },
            new Personagem() { Id = 6, Nome = "Celeborn", PontosVida=100, Forca=21, Defesa=13, Inteligencia=34, Classe=ClasseEnum.Clerigo },
            new Personagem() { Id = 7, Nome = "Radagast", PontosVida=100, Forca=25, Defesa=11, Inteligencia=35, Classe=ClasseEnum.Mago }
        };

        //Métodos deverão ser construídos a partir daqui.
    }
}
```

- a) Método do tipo *Get* com rota **GetByClasse** que selecione a lista de personagens de acordo com o id da classe informada na requisição.
- b) Método do tipo *Get* com a rota **GetByName** que selecione um personagem de acordo com o nome digitado e que caso não encontre retorne a mensagem *NotFound* informando formalmente na resposta da requisição.
- c) Método do tipo *Post* com rota **PostValidacao** que não permita que um Personagem seja adicionado tendo o valor de defesa menor que 10 ou inteligência maior que 30. Deve retornar um *BadRequest* caso esse dado seja enviado na requisição.
- d) Método do tipo *Post* com rota **PostValidacaoMago** que não permita que um personagem do tipo Mago seja incluído com inteligência menor que 35. Deve retornar um *BadRequest* caso esse dado seja enviado na requisição.
- e) Método do tipo *Get* com rota **GetClerigoMago** que remova os personagens que são cavaleiros, exiba a lista decrescente por *PontosVida*.
- f) Método do tipo *Get* com rota **GetEstatisticas** e exiba a quantidade de personagens da lista e qual o somatório de inteligência dos personagens.



Criação de conta no Somee e configurações

O Somme é uma hospedagem gratuita que conta com gerenciamento de arquivos, banco de dados SQL Server e servidor IIS.

Acesse <https://somee.com> e clique no botão Learn More da opção hospedagem gratuita e logo após clique em Ordem Now.

Free .Net Hosting

- Free ASP.Net web hosting
- 150MB storage, 5GB transfer
- ASP.Net 2.0-4.7, ASP.Net Core 2.2
- 15MB MSSQL 2012/2014/2016
- Free third level domain
- FTP access

\$0.00

Learn More

Free Hosting package

1 x Hosting plan "Freebie"

- Forced advertising
- Storage capacity: 150MB
- Monthly transfer: 5GB/Month
- Web domains: 1
- ASP.Net 6.7/4.6/4.5/4.0/3.5/2.0
- ASP.Net Core 2.2
- AJAX 3.5/1.0
- Silverlight
- MS Access 2007, 2010
- Dedicated web application pool

1 x Email plan "Forwarder"

1 x MS SQL Plan "Novice"

- MS SQL database size: 15MB
- MS SQL log size: 20MB
- Backup storage size: 40MB

\$0.00

Order now

Realize o cadastro com um ID (Grave o ID pois o login futuro será com ele), senha e confirme o cadastro através do código de confirmação recebido no e-mail.



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

Faça o login no somee e procure a opção Managed products → MS SQL → Logins para usuários para o futuro banco de dados

Defina login e senha para o usuário a ser criado (Crie seu usuário e senha do Banco e memorize esses dados em suas anotações). Sahas é um exemplo, dê o nome do seu próprio usuário.

MS SQL database logins

Login name

Login: Crie seu User (3 to 20 characters)

Password: Anote sua senha, não coloque senhas usualmente pessoais

Confirm password:

ADD NEW LOGIN

- Nome de usuário único. Ex: saxyz (super admin xyz)

Procure a opção Manged Products → MS SQL → Databases → Create para criar a base de dados chamada DB-DS-**XY** onde **X** será seu nome e **Y** seu sobrenome.

MS SQL plan: MS SQL Plan "Novice"

Database name (7-30 symbols): DB-DS-LUIZSOUZA (Nome conforme o padrão)

Zone name: somee.com

MS SQL login: Selecione seu login

MS SQL Server version: MS SQL 2019 Express

The process may take up to several minutes. Please be patient.

CREATE EMPTY DATABASE

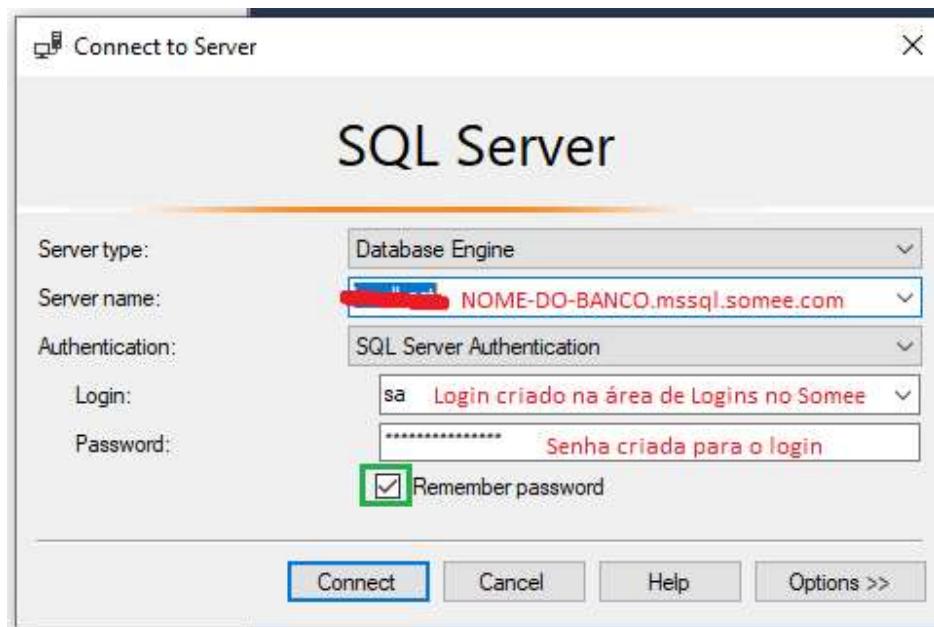


Observe os detalhes da conexão. Usaremos a Connection String futuramente para criar as conexões no VS Code.

The screenshot shows the 'Connection details' pane of the SQL Server Management Studio. On the left, there's a tree view of database objects under 'MS SQL' > 'Databases'. A specific database, 'DB-DS-LUIZSOUZA', is selected and highlighted with a green border. The main pane displays the following connection information:

Connection details	
MS SQL Server version:	MS SQL 2019 Express
MS SQL Server address:	DB-DS-LUIZSOUZA.mssql.somee.com
Login name:	sahas
Login password:	[REDACTED]
Connection string:	workstation id=DB-DS-LUIZSOUZA.mssql.somee.com;packet size=4096;user id=sahas;pwd=[REDACTED];data source=DB-DS-LUIZSOUZA.mssql.somee.com;persist security info=False;initial catalog=DB-DS-LUIZSOUZA

Caso deseje acessar a base via SQL Management Studio, é fornecido o endereço na linha MS SQL Server Address e logo abaixo o login e a senha da base.



Recomendação de extensão do SQL para o VS Code

<https://marketplace.visualstudio.com/items?itemName=ms-mssql.mssql>

Instruções: <https://juniorporfirio.medium.com/realizando-consultas-sql-no-vscode-6a9c2f1cf998>



Aula 07 - Entity Framework Core – ORM para conexão com banco de dados

Frameworks ORM (*Object-Relational Mapping*) representam uma forma de auxiliar o desenvolvedor a conectar uma api/aplicação ao banco de dados sem ter que fazer tudo manualmente e existem diversas ferramentas que empregam esta facilidade.

Esquenta sobre Frameworks ORM indicados

Canal Código Fonte TV – ORM (*A ponte entre a O.O. e o Banco de dados*) // Dicionário do programador:
<https://youtu.be/snOXxJa31GI>

Canal Alura – O que é um ORM: <https://youtu.be/x39vqeBTUmE>

Aprenderemos através do *Entity Framework Core* como realizar a conexão com banco de dados, utilizando o método *Code First Migration*, que cria o banco de dados a partir de classes de classes de Modelo existentes. Isso nos permitirá realizar as principais operações de Banco de dados que costumamos chamar de *CRUD* (Create, Read, Update e Delete)

1. Para configurar o uso do *Entity Framework Core*, abra o terminal e instale o pacote digitando `dotnet add package Microsoft.EntityFrameworkCore.SqlServer -v 5.0.15`. No decorrer do processo observe que será adicionada uma referência ao arquivo `RpgApi.csproj`
2. Agora instalaremos a ferramenta que permite trabalhar com os comandos do *Migration*, digitando no terminal o comando `dotnet tool install --global dotnet-ef -v 5.0.15 (--version 5.0.15)`
3. Para finalizar instale o pacote com o comando `dotnet add package Microsoft.EntityFrameworkCore.Design -v 5.0.15` que permitirá fazer a customização das tabelas do banco de dados via programação C#

Para usar o *Entity Framework Core* precisaremos ter uma classe que realize a interação com o banco de dados representando uma sessão entre ela e o Banco, essa classe será a *DataContext* que herdará os métodos de *DbContext* (contida no framework)

4. Crie uma pasta **Data** e dentro desta pasta crie a classe **DataContext.cs**

```
using Microsoft.EntityFrameworkCore;

namespace RpgApi.Data
{
    0 references
    public class DataContext : DbContext
    {
    }
}
```

- Observe que estamos utilizando a herança à classe *DbContext* que necessita do *using* sinalizado. Use CRTL + . (ponto) para incluir mais rapidamente.



5. Crie um construtor dentro do corpo da classe. Atalho: digite ctor + clique TAB no teclado.

```
public class DataContext : DbContext
{
    0 references
    public DataContext(DbContextOptions<DataContext> options) : base(options)
    {
    }
```

- **Construtor** é como um método padrão que roda quando a esta classe for instanciada em memória, criando um objeto. O construtor **SEMPRE** tem o mesmo nome da classe e o modificador de acesso público.
 - Neste caso como a classe realiza uma herança da classe DbContext. O parâmetro Options que é recebido pelo construtor é passado também para a classe pai ou classe base.
6. Para configurar o mapeamento da classe Personagem no banco de dados utilize a codificação a seguir abaixo do construtor:

```
public DataContext(DbContextOptions<DataContext> options) : base(options)
{
}

}
0 references
1           2
public DbSet<Personagem> Personagens { get; set; }
```

- (1) É a classe da pasta *Models*, será necessário fazer um *using* para adicionar o *namespace*.
 - (2) Representa o nome que será criado para a tabela no banco de dados.
7. Crie o método OnModelCreating que será responsável por alimentar a tabela de Personagens automaticamente quando o banco de dados estiver sendo criado

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Personagem>().HasData
    (
        //Inserir as linhas " new Personagem() { Id = 2,..." da lista de Personagens
    );
}
```

- Ao inserir os objetos da lista será necessário referenciar o using para *RpgApi.Models.Enums*



8. O método ficará com a seguir e ele utiliza a palavra-chave **override** que significa sobrescrever. Sobrescrita é um fundamento da orientação a objeto em que métodos pré-definidos ou existentes podem ser modificados tomando uma nova forma. Como o nome do método é OnModelCreating (durante a criação do modelo), entendemos que a criação acontecerá quando o banco de dados for criado ou atualizado.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Personagem>().HasData
    (
        new Personagem() { Id = 1 }, //Frodo Cavaleiro
        new Personagem() { Id = 2, Nome = "Sam", PontosVida=100, Forca=15, Defesa=25, Inteligencia=18, Velocidade=10, Agilidade=12, Resistencia=15 },
        new Personagem() { Id = 3, Nome = "Galadriel", PontosVida=100, Forca=18, Defesa=21, Inteligencia=20, Velocidade=12, Agilidade=14, Resistencia=17 },
        new Personagem() { Id = 4, Nome = "Gandalf", PontosVida=100, Forca=18, Defesa=18, Inteligencia=22, Velocidade=10, Agilidade=12, Resistencia=15 },
        new Personagem() { Id = 5, Nome = "Hobbit", PontosVida=100, Forca=20, Defesa=17, Inteligencia=18, Velocidade=12, Agilidade=14, Resistencia=16 },
        new Personagem() { Id = 6, Nome = "Celeborn", PontosVida=100, Forca=21, Defesa=13, Inteligencia=20, Velocidade=14, Agilidade=16, Resistencia=18 },
        new Personagem() { Id = 7, Nome = "Radagast", PontosVida=100, Forca=25, Defesa=11, Inteligencia=16, Velocidade=10, Agilidade=12, Resistencia=14 }
    );
    //Área para futuros Inserts no banco
}
```

9. Entre em sua conta do somee, navegue até a parte de banco de dados, copie a *connection string* sinalizada em verde, deixando na sua área de transferência ou em um bloco de notas.

The screenshot shows the 'General' tab of the MS SQL interface. In the left sidebar, under 'Databases', there is a red box around 'DB-DS-LUIZSOUZA'. In the main area, under 'Connection details', there is a red box around the 'Connection string' field, which contains the following value:

```
workstation id=DB-DS-LUIZSOUZA.mssql.somee.com;packet size=4096;user id=sahas;pwd=sasqlh@5;data source=DB-DS-LUIZSOUZA.mssql.somee.com;persist security info=False;initial catalog=DB-DS-LUIZSOUZA
```

10. Agora será necessário configurar o caminho do Banco de Dados: Abra o arquivo appsettings.json e crie a estrutura a seguir, colando a string de conexão entre as aspas indicada na mensagem abaixo.

```
1  {
2
3     "ConnectionStrings": {
4         "ConexaoSomee": "COLOCAR SUA STRING DE CONEXÃO AQUI"
5     },
6
7     "Logging": {
```

11. Devemos ir até a classe Startup.cs e no método ConfigureServices indicar que utilizaremos o contexto do Banco de Dados SQL Server junto com a string de conexão que indicamos

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<DataContext>(x => x.UseSqlServer(Configuration.GetConnectionString("ConexaoSomee")));
    services.AddControllers();
}
```

- Acesse File → Save All para salvar tudo que foi feito () nas classes e arquivos.



Vamos conferir através do comando dotnet ef -h as opções que este comando oferece:

```
Commands:  
database  Commands to manage the database.  
dbcontext  Commands to manage DbContext types.  
migrations  Commands to manage migrations.
```

Como usaremos o *migrations*, use o comando donet net ef migrations -h para ver as opções de comandos.

```
Commands:  
add      Adds a new migration.  
list     Lists available migrations.  
remove   Removes the last migration.  
script   Generates a SQL script from migrations.
```

12. O comando *migrations add* terá como missão transferir toda configuração das classes de modelo para o a migração que será feita. Adicionaremos junto ao comando a palavra *InitialCreate*, pois se trata da primeira vez que estamos fazendo os procedimentos com o *migrations*.

```
dotnet ef migrations add InitialCreate
```

Observe que foi criada uma pasta *Migrations* com alguns arquivos que conterão informações a respeito das tabelas e colunas que o *Entity Framework Core* utilizará para criar efetivamente as tabelas no Banco dados.

```
▼ Migrations  
C 20200906170010_InitialCreate.cs  
C 20200906170010_InitialCreate.Designer.cs  
C DataContextModelSnapshot.cs
```

Observando a classe com sufixo *InitialCreate.cs*, podemos observar os métodos *Up* e *Down*, sendo que o primeiro exibe como será a configuração das tabelas, colunas e constraints detalhadamente, caso a migração seja confirmada, e o segundo desfaz o que for realizado no *Up*, caso seja necessário voltar a operação, o que costumamos chamar de *RollBack* e deletaria a tabela.

13. Execute o comando para confirmar e finalizar a criação da tabela na base de dados

```
dotnet ef database update
```

- Abra o SQL Server, conecte ao servidor e confirme que a tabela e colunas estão criadas.

Nas próximas etapas configuraremos a controller para salvar os dados diretamente na base de dados que criamos nesta aula.



Aula 08 - Operações CRUD com o Entity Framework Core

Agora que temos a tabela devidamente criada no banco de dados, podemos programar a *controller PersonagensController* para que os métodos salvem os dados enviados (via client de API como o postman, o Swagger ou Talend API) nas tabelas criadas na base de dados. Quem viabilizará esta operação é a classe *DataContext* que ao realizar o mapeamento das classes para as tabelas do banco permitirá ações diretamente no banco de dados.

1. Criação da classe **PersonagensController**, dentro da pasta *Controllers*

```
using Microsoft.AspNetCore.Mvc;

namespace RpgApi.Controllers
{
    [ApiController]
    [Route("[Controller]")]
    0 references
    public class PersonagensController : ControllerBase
    {
        //Programação seguinte será aqui

        }//Fim da classe do tipo controller
}
```

2. Dentro da classe *controller* crie um atributo global que será do tipo *DataContext* com o nome de *_context* (1), ele exigirá o using de *RpgApi.Data*. Crie também o construtor para inicializar o atributo *_context*, que receberá os dados via parâmetro chamado de *context* (2)

```
public class PersonagensController : ControllerBase
{
    //Programação de toda a classe ficará aqui abaixo
    1 reference
    1 private readonly DataContext _dataContext;

    0 references
    2 public PersonagensController(DataContext dataContext)
    {
        _dataContext = dataContext;
    }

}//Fim da classe do tipo controller
```



3. Crie o método de nome **GetSingle** seja feita uma busca no contexto do Banco de Dados para retornar um personagem de acordo com o Id. Usings necessários: System.Threading.Tasks;RpgApi.Models; Microsoft.EntityFrameworkCore.Core e System

```
[HttpGet("{id}")]
0 references
public async Task<IActionResult> GetSingle(int id)
{
    try
    {
        Personagem p = await _context.Personagens
            .FirstOrDefaultAsync(pBusca => pBusca.Id == id);

        return Ok(p);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

- Observe que temos um bloco try/catch onde tudo que tem que ser programado ficará dentro do try e se algum erro acontecer o Catch vai capturar a mensagem dele e devolver ao requisitante.
4. Crie um método de rota GetAll chamado Get. Exigirá o using System.Collections.Generic

```
[HttpGet("GetAll")]
0 references
public async Task<IActionResult> Get()
{
    try
    {
        List<Personagem> lista = await _context.Personagens.ToListAsync();
        return Ok(lista);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



5. Criação do método do tipo Post chama de **Add**. O comando `SaveChangesAsync` confirmará a inserção dos dados. Temos também uma validação dos pontos de vida que lança uma exceção. Observe que no retorno do método estamos exibindo Id que o banco de dados atribuirá ao personagem

```
[HttpPost]
0 references
public async Task<IActionResult> Add(Personagem novoPersonagem)
{
    try
    {
        if (novoPersonagem.PontosVida > 100)
        {
            throw new Exception("Pontos de Vida não pode ser maior que 100");
        }
        await _context.Personagens.AddAsync(novoPersonagem);
        await _context.SaveChangesAsync();

        return Ok(novoPersonagem.Id);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

6. Crie um método do tipo Put com o nome de Update. Observe que guardamos a quantidade de linhas afetadas para retornar no resultado da requisição.

```
[HttpPut]
0 references
public async Task<IActionResult> Update(Personagem novoPersonagem)
{
    try
    {
        if (novoPersonagem.PontosVida > 100)
        {
            throw new Exception("Pontos de Vida não pode ser maior que 100");
        }
        _context.Personagens.Update(novoPersonagem);
        int linhasAfetadas = await _context.SaveChangesAsync();

        return Ok(linhasAfetadas);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



7. Crie um método do tipo Delete com o nome **Delete**. Observe que como só temos o id sendo passado, primeiro vamos na base encontrar o objeto que desejamos remover e passamos este objeto para o contexto excluir. Após isso, retornamos da base quantas linhas foram afetadas, que será só uma, pois estamos usando a chave primária.

```
[HttpDelete("{id}")]  
0 references  
public async Task<IActionResult> Delete(int id)  
{  
    try  
    {  
        Personagem pRemover = await _context.Personagens  
            .FirstOrDefaultAsync(p => p.Id == id);  
  
        _context.Personagens.Remove(pRemover);  
        int linhasAfetadas = await _context.SaveChangesAsync();  
  
        return Ok(linhasAfetadas);  
    }  
    catch (Exception ex)  
    {  
        return BadRequest(ex.Message);  
    }  
}
```

- Execute a API e realize os testes com o *Postman*, sempre conferindo as atualizações na tabela criada no *Somee*.



Deploy da API no Somee

Realizar o deploy de uma aplicação ou neste caso, de uma API significa realizar a publicação dos arquivos em um servidor para que fique disponível para uso. O deploy de uma API pode ser feito em um servidor web que tenha o IIS instalado ou outro servidor que não seja windows, desde tenha a plataforma do .NET Core instalada e essa é a principal novidade do framework, pois antes os projetos desenvolvidos no Visual Studio/C# só rodavam em servidores Windows e agora é possível rodar aplicações C# em servidores Linux e outras plataformas que existem.

Esta etapa de deploy será necessária para que os sites, sistemas e aplicativos possam consumir a programação realizada na API. Publicaremos a API na hospedagem gratuita do somee <http://somee.com> que conta com gerenciamento de arquivos, banco de dados SQL Server e servidor IIS, conforme visto anteriormente.

1. Na Api, abra o terminal e realize o comando para gerar os arquivos de publicação

```
\RpgApi> dotnet publish -c release -o ./publish
```

- Será criada uma pasta publish dentro da pasta do projeto, observando a aba de pastas e arquivos do VS Code. Entre nesta pasta (Clique com o direito e Reveal in File Explorer), selecione todos os arquivos/pastas (CTRL + A) e escolha enviar para uma pasta compactada.

2. Entre no somee, vá até Websites e faça a criação de um Website

Create website

Hosting plan: Hosting plan "Freebie"

Although your hosting plan supports global domains you still need to provide initial default domain name which is hosted within our zone. You will be able to add additional domains later in control panel.

Site name (Subdomain): nomeWebsite (Recomendado deixar o mesmo ID do Somee)

Zone name: somee.com

Operating system: Windows Server 2016 (IIS 10.0, ASP, ASP.NET v2.0-4.7, ASP.NET Core 2.2)

ASP.NET version: 4.0, 4.5, 4.6, 4.7

Site title:

Site description:

Info: Your site will have default domain names as 'Sub domain', 'Zone name' and www.'Sub domain', 'Zone name'. If supported by hosting plan, you can use your own domain names, registered with domain name registrar. DNS record will be created instantly, but because of DNS replication delay it may take up to 24 hours for your site to be visible to all users on the Internet. You can try to access your site right after it's created. If it's not working retry it every 30 minutes.

The process may take up to several minutes. Please be patient.

Create website

- Leve em consideração deixar o subdomínio igual o id que usa para entrar no Somee. Logo, o endereço ficará <http://subdominio.somee.com>



3. Na hierarquia de menus, expanda o nome do seu website recém-criado e selecione File Manager, e crie um diretório chamado RpgApi.
4. Clique no diretório RpgApi e siga a sequência para upload, selecionar o arquivo compactado com toda a API e depois fazer o upload descompactando

Site: xyz.somee.com

The screenshot shows a file manager interface with the following elements:

- A left sidebar with five "Choose File" input fields, each labeled "No file chosen". The second field has a red box around it with the number "2".
- To the right of the inputs is a checkbox labeled "Overwrite existing files (not apply for unzipped files)" with a checked mark.
- Below the checkbox is a red box with the number "3" containing the text "Upload and Unzip archives".
- Below the "Upload and Unzip archives" button is another button labeled "Upload files".
- At the bottom of the interface is a toolbar with icons for Root, Up, Cut, Paste, New dir, New page, Delete, Refresh, Reverse, and Upload. The "Upload" icon has a red box around it with the number "1".
- On the right side, there is a status bar showing "KB Current path: RpgApi/" followed by a dropdown arrow.
- At the very bottom is a table header with columns: "File name", "Size", and "Last modified".

5. Procure por “applications” na hierarquia do website, clique na pasta de nome RpgApi e em “Convert folder to application”.

The screenshot shows a "Directory structure" window with the following elements:

- A title "Pick folder: Directory structure".
- A tree view showing a root folder with a subfolder named "RpgApi" which is highlighted with a red box.
- Below the tree view is a button labeled "Convert folder to Application" with a red box around it.

- Agora será possível chamar o endereço da api diretamente do postman.

Ex.: **xyz.somee.com/RpgApi/GetAll** (**xyz** substituído pelo nome do seu website) recuperando assim todos os Personagens.

Faça o teste dos demais métodos confirmando que as informações continuam interagindo com o banco de dados, como buscar um personagem, atualizar um personagem salvo, e remover um personagem.



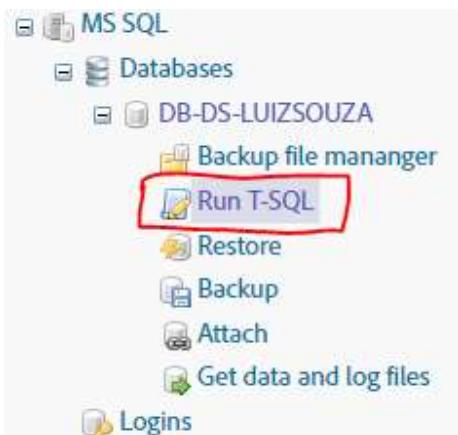
Operações Extras Banco de Dados / Somee

Gerar o script do banco através das migrações para executar direto no somee

VS Code → View → Terminal → Comando → dotnet ef migrations script -o ./script.sql

- Será criado o arquivo abaixo  com os comandos SQL que poderão ser executados no Somee ou no SQL

Rodar script no Somee



- Selecionar a opção RunT-SQL → Open T-SQL Console. Se a execução for de vários comandos juntos, escolher Run AS T-Sql Batch, se for um comando apenas, escolher Run as Query

Deletar o banco de dados para novo comando de criação

- SQL Management Studio → Clicar com o botão direito no database → Delete → escolher as opções para fechar as conexões existentes (Close existente connections)



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

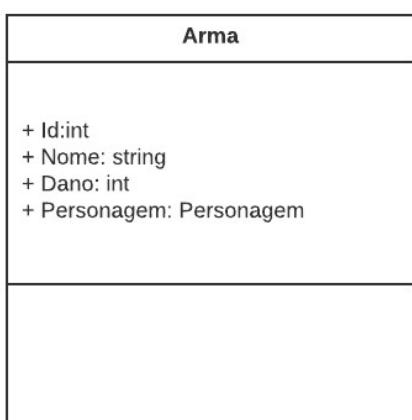
Projeto Avaliativo: Crie um projeto no VS Code utilizando como base as aulas a partir da criação do projeto RpgApi, utilizando as instruções como referência. Escolha um tema para seu projeto

- Crie um diagrama de classes para a classe que você criará. Veja o exemplo do diagrama da classe Personagem de aulas atrás. A sua classe deve ter uma propriedade Id e pelo menos mais cinco propriedades com variação nos tipos. Ex: DateTimes, decimals, strings, ints.
- Crie a pasta Models em seu projeto e dentro desta pasta crie a sua classe e as propriedades que ela terá.
- Abra o terminal e execute os comandos para instalação das referências do EntityFramework. Configure o arquivo appsettings com a string de conexão e a classe Startup para fazer a referência para o nome da string de conexão que será usada.
- Crie a pasta Data e a classe DataContext herdando de DbContext e o construtor inicializando o contexto.
- Adicione no Método OnModelCreating da classe DataContext, dados para adicionar Objetos de sua classe na tabela que será criada através das migrações.
- Faça o Mapeamento da classe para referenciar a futura tabela na classe DataContext.
- Crie uma classe **Controller** dentro da pasta Controllers realizando todas as etapas de definição e configuração da controller e os métodos que utilizam o Entity Framework para manipular o banco de dados.
- Executar o comando no terminal para criação da Migração que deve ser chamada “MigracaoInicial”
- Executar o comando no terminal para a atualização do Banco de dados para execução da migração criada.
- Testar no postman o funcionamento das operações, verificando se vão alterar o Banco de dados.
- Publicação das atualizações da API no somee. Não esqueça de converter a sua pasta para aplicativo.
- Teste no postman do funcionamento da API apontando para o endereço do Somee.
- Envio na atividade do Teams:
 - O print do seu diagrama de classes
 - Os seus endpoint. Ex: <http://xyz.somee.com/PastaProjetoDoSomee/NomeDaController>



Exercício Prático de Recuperação: Mapeamento da classe Arma para o banco de dados e programação na Controller

- Crie uma Classe de acordo com o diagrama a seguir



```
public int Id { get; set; }  
1 reference  
public string Nome { get; set; }  
2 references  
public int Dano { get; set; }
```

- Faça o Mapeamento da classe Arma para referenciar a futura tabela “Armas” na classe DataContext.
- Utilizar a classe **ArmasController** dentro da pasta Controllers realizando todas as etapas de definição e configuração da controller, bem como instância do banco de dados (DataConext) variável global do contexto, construtor inicializando o contexto e os métodos que utilizam o Entity Framework para manipular o banco de dados.
- Adicionar no Método OnModelCreating da classe DataContext, dados para adicionar 3 Armas na tabela.
- Executar o comando no terminal para criação da Migração que deve ser chamada “MigracaoArma”
- Executar o comando no terminal para a atualização do Banco de dados para execução da migração criada.
- Testar no postman o funcionamento das operações do Banco de dados.
- Publicação das atualizações da API no somee.
- Teste no postman do funcionamento da API apontando para o endereço do Somee.
- Envio na atividade do Teams com o endereço dos seus endpoints. Ex:
<http://xyz.somee.com/RpgApi/NomeDaController>



Aula 10 – Relacionamentos: One to Many, One to One e Many to Many

Com a criação da classe **Usuario** criaremos uma tabela no banco de dados chamada **Usuarios** através do mapeamento da classe e das migrações. Esta classe e tabela serão importantes, pois, criará um relacionamento com os dados do personagem do tipo um para muitos, em que um usuário poderá ter diversos personagens atrelados a ele.

No banco de dados usaremos um tipo de dado chamado hash para não expor a senha do usuário e um salt que nada mais é do que caracteres que são concatenados combinados antes, durante ou depois do hash a fim de evitar que a senha seja descoberta com técnicas de quebras de segurança. Mais detalhes poderão ser entendidos com as referências abaixo:

- Hash e Salt de senhas: <https://www.brunobrito.net.br/securanca-salt-hash-senha/>
- Exemplo de criação de hash em C#: <https://www.youtube.com/watch?v=ggPgk4znUEY>

1. Abra o projeto **RpgApi** e crie a classe **Usuario.cs** dentro da pasta **Models**, codificando conforme a seguir.

```
public int Id { get; set; }
2 references
public string Username { get; set; }
1 reference
public byte[] PasswordHash { get; set; }
1 reference
public byte[] PasswordSalt { get; set; }
0 references
public byte[] Foto { get; set; }
0 references
public string Latitude { get; set; }
0 references
public string Longitude { get; set; }
0 references
public DateTime? DataAcesso { get; set; }

[NotMapped]
2 references
public string PasswordString { get; set; }
0 references
public List<Personagem> Personagens { get; set; }
```

- Usings necessários: System, System.ComponentModel.DataAnnotations.Schema e System.Collections.Generic
- Note que além das propriedades normais estamos criando uma lista de personagens. Isso definirá que um Usuário pode possuir vários personagens.



2. Abra a classe Personagem e adicione a codificação sinalizada. Para saber para qual Usuário um objeto do tipo Personagem estará atrelado, faremos a declaração do objeto na classe Personagem conforme abaixo. Vamos aproveitar e criar uma propriedade que futuramente armazenará a foto do Personagem.

```
public ClasseEnum Classe { get; set; }  
0 references  
public byte[] FotoPersonagem { get; set; }  
0 references  
public Usuario Usuario { get; set; }
```

3. Crie uma pasta chamada **Utils** e dentro dela crie a classe **Criptografia** e adicione o método abaixo. Esse método é estático, ou seja, não precisará da classe estanciada para chama-lo futuramente.

```
public static void CriarPasswordHash(string password, out byte[] hash, out byte[] salt)  
{  
    using (var hmac = new System.Security.Cryptography.HMACSHA512())  
    {  
        salt = hmac.Key;  
        hash = hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));  
    }  
}
```

4. Abra a classe DataContext e adicione a referência à classe Usuario recém-criada para o contexto do banco de dados, o que chamamos de mapeamento. Procure a região onde temos outros mapeamentos feitos.

```
public DbSet<Usuario> Usuarios { get; set; }
```

5. Ainda na classe DataContext, posicione o cursor antes do fechamento do método OnModelCreating para preparar um usuário padrão para quando a tabela for alimentada. Exigirá o using para RpgApi.Utils para reconhecer a classe Criptografia.

```
    new Personagem() { Id = 7, Nome = "Radagast", PontosVida=100, Forca=25, Defesa=10 },  
);  
  
    //Início da criação usuário padrão  
    Usuario user = new Usuario();  
    Criptografia.CriarPasswordHash("123456", out byte[] hash, out byte[] salt);  
    user.Id = 1;  
    user.Username = "UsuarioAdmin";  
    user.PasswordString = string.Empty;  
    user.PasswordHash = hash;  
    user.PasswordSalt = salt;  
  
    modelBuilder.Entity<Usuario>().HasData(user);  
    //Fim da criação do usuário padrão.  
  
}//Fim do método OnModelCreating
```



6. Use o terminal para criar a migração para a classe usuário chamada de `MigracaoUsuario` através do comando **`dotnet ef migrations add MigracaoUsuario`**. Observe os arquivos criados na pasta Migration e a atualização da classe `DataContextModelSnapshot.cs`
7. Execute o comando “**`dotnet ef database update`**” para atualizar a base de dados. Caso exista algum bloqueio para a execução do comando no lab, gere o script para esta migração através do comando:

```
A      B      C  
dotnet ef migrations script InitialCreate MigracaoUsuario -o ./scriptMigracaoUsuario.sql
```

- Temos em (A) a migração anterior, em (B) a migração atual e em (C) o arquivo que será gerado.
- Abra o arquivo gerado e execute o script no somee.
- Confira se a tabela e os campos foram criados. Atualize o banco e confirme que a chave estrangeira foi criada na tabela Personagens, bem como um campo de nome `Usuarioid`

Relacionamento One to One

Para representar o aprendizado do relacionamento um para um em nossa matéria, um personagem poderá ter apenas uma arma, e vice-versa na regra de negócio da API.

8. Inclua uma propriedade do tipo Personagem na classe Arma. Pode ser que esta propriedade esteja comentada, apenas acerte o nome dela, conforme abaixo. Crie também a propriedade **`PersonagemId`** que será int.

```
public class Arma
{
    2 references
    public int Id { get; set; }
    0 references
    public string Nome { get; set; }
    0 references
    public int Dano { get; set; }
    0 references
    public Personagem Personagem { get; set; }
    0 references
    public int PersonagemId { get; set; }
}
```

- A propriedade `PersonagemId` existirá para definir a criação de uma chave estrangeira entre Personagem e Arma, tendo o entendimento que um Personagem poderá existir sem ter uma arma, mas que uma arma não poderá ser salva sem que exista um id de Personagem em sua tabela. Esta é a relação de dependência de um relacionamento one to one.



9. Adicione uma propriedade do tipo Arma chamada Arma na classe Personagem. Necessário o using System.Text.Json.Serialization;

```
[JsonIgnore]  
4 references  
public Usuario Usuario { get; set; }  
  
[JsonIgnore]  
0 references  
public Arma Arma { get; set; }
```

- Perceba que estamos colocando a anotação `JsonIgnore` acima da propriedade desta classe Personagem. Isso será necessário para que o EntityFramework não caia em loops de carregamentos nos métodos Get ao carregar um Personagem, já que as classes Arma e Usuário também tem propriedade do tipo Personagem. Até o final desta aula, instalaremos o pacote que evita isso e poderemos remover esta anotação `JsonIgnore`

10. Abra a classe `DataContext` para alterar o método `OnModelCreating`, adicionando dados para o salvamento de uma Arma, relacionando cada arma a um Id de personagem existente.

```
modelBuilder.Entity<Usuario>().HasData(user);  
//Fim da criação do usuário padrão.  
  
modelBuilder.Entity<Arma>().HasData  
(  
    new Arma() { Id = 1, Nome="Arco e Flecha", Dano=35,PersonagemId=1 },  
    new Arma() { Id = 2, Nome="Espada", Dano=33,PersonagemId=2},  
    new Arma() { Id = 3, Nome="Machado", Dano=31,PersonagemId=3 }  
);  
  
}//Fim do método OnModelCreating
```

11. Salve as classes alteradas e crie a migração através do comando “`dotnet ef migrations add MigracaoUmParaUm`”

- Observe no arquivo de criação da migração uma nova coluna para ser criada na tabela armas, bem como a definição da foreign key.
- Já no final do arquivo de design da migração pode ser observado as anotações de `HasOne` e `WithOne` que define a relação um para um além da anotação `OnDelete` que dita o comportamento da remoção de um personagem removido, deletando também a arma dele, conhecido como deleção em cascata.



12. Atualize o banco de dados através do EntityFramework (ef) para que a migração realize as alterações com o comando “dotnet ef database update” ou gere o script conforme aprendemos para rodar manualmente através do comando **“dotnet ef migrations script MigracaoUsuario MigracaoUmParaUm -o ./scriptMigracaoUmParaUm.sql”**

Relacionamento Many to Many

Para fazer uso do relacionamento **muitos para muitos** criaremos uma classe de Habilidades em que poderemos ter diversas delas cadastradas, sendo que um personagem poderá ter várias habilidades e uma mesma habilidade poderá constar em vários personagens. Levando em consideração os aprendizados de banco de dados, isso criará uma terceira tabela na base de dados para juntar o id do personagem e id da habilidade. Vamos aos códigos!!!

13. Crie a classe **Habilidade** dentro da pasta Models. Note que desta vez não teremos o Personagem nem a lista dele sendo declarados ainda, pois como o relacionamento é muitos para muitos, esta vinculação ficará na classe a seguir.

```
namespace RpgApi.Models
{
    0 references
    public class Habilidade
    {
        0 references
        public int Id { get; set; }
        0 references
        public string Nome { get; set; }
        0 references
        public int Dano { get; set; }
    }
}
```

14. Crie outra classe chamada **PersonagemHabilidade** na pasta Models

```
namespace RpgApi.Models
{
    0 references
    public class PersonagemHabilidade
    {
        0 references
        public int PersonagemId { get; set; }
        0 references
        public Personagem Personagem { get; set; }
        0 references
        public int HabilidadeId { get; set; }
        0 references
        public Habilidade Habilidade { get; set; }
    }
}
```



15. Faça o mapeamento das classes recém-criadas na classe DataContext, em preparação para a migração.

```
public DbSet<Habilidade> Habilidades { get; set; }  
0 references  
public DbSet<PersonagemHabilidade> PersonagemHabilidades { get; set; }
```

16. Ainda na classe de mapeamento (DataContext) adicionaremos ao método OnModelCreating, abaixo de onde populamos as armas, uma chave composta para a tabela PersonagemHabilidade (A), e alguns dados de Habilidade para inserir na futura tabela Habilidades (B).

```
modelBuilder.Entity<Arma>().HasData  
(  
    new Arma() { Id = 1, Nome="Arma A", Dano=35, PersonagemId=1  
    new Arma() { Id = 2, Nome="Arma B", Dano=33, PersonagemId=2  
    new Arma() { Id = 3, Nome="Arma C", Dano=31, PersonagemId=3  
);  
  
A modelBuilder.Entity<PersonagemHabilidade>()  
    .HasKey(ph => new { ph.PersonagemId, ph.HabilidadeId});  
  
B modelBuilder.Entity<Habilidade>().HasData  
(  
    new Habilidade() { Id = 1, Nome="Adormecer", Dano=39 },  
    new Habilidade() { Id = 2, Nome="Congelar", Dano=41 },  
    new Habilidade() { Id = 3, Nome="Hipnotizar", Dano=37}  
);  
)
```

17. Adicione uma Propriedade que seja uma lista de PersonagemHabilidades na classe Personagem e na classe Habilidade. Necessitará do using System.Collections.Generic.

```
0 REFERENCES  
public List<PersonagemHabilidade> PersonagemHabilidades {get; set;}
```

- Em algum momento poderemos resgatar todas as habilidades de um personagem ou todos os personagens que tenham uma determinada habilidade. A lista se mostra interessante para este fim.



18. Crie a migração a partir das mudanças realizadas com o comando **dotnet ef migrations add MigracaoMuitosParaMuitos**

- Observe no final do arquivo de design da migração criado, a definição do relacionamento da tabela PersonagemHabilidades com a tabela de habilidade e de personagem.

```
modelBuilder.Entity("RpgApi.Models.PersonagemHabilidade", b =>
{
    b.HasOne("RpgApi.Models.Habilidade", "Habilidade")
        .WithMany("PersonagemHabilidades")
        .HasForeignKey("HabilidadeId")
        .OnDelete(DeleteBehavior.Cascade)
        .IsRequired();

    b.HasOne("RpgApi.Models.Personagem", "Personagem")
        .WithMany("PersonagemHabilidades")
        .HasForeignKey("PersonagemId")
        .OnDelete(DeleteBehavior.Cascade)
        .IsRequired();
});
```

19. Execute a atualização no banco com o comando **dotnet ef database update** ou se precisar gerar script gere através do comando “**dotnet ef migrations script MigracaoUmParaUm MigracaoMuitosParaMuitos -o ./scriptMigracaoMuitosParaMuitos.sql**”

- Confira se as tabelas e relacionamentos foram criados.



Aula 11 – Programação de Controllers - Registro e login de Usuário – Inserção de habilidades

1. Crie a controller **UsuariosController.cs** na pasta Controllers

```
[ApiController]
[Route("[controller]")]
0 references
public class UsuariosController : ControllerBase
```

- Lembre-se que toda *controller* herda características da ControllerBase classe do .NET Core que requer o *using Microsoft.AspNetCore.Mvc*.
- Lembre-se também que usamos a diretiva *ApiController* para sinalizar que é uma controller de API.
- A diretiva *Route* indica a rota pela qual a *controller* será chamada, neste caso está a padrão, logo, o endereço da *controller* será “<http://localhost:5000/Usuarios>”

2. Declare de maneira global a variável que representará a classe do contexto do banco de dados e receba os dados para esta variável no construtor.

```
public class UsuariosController : ControllerBase
{
    1 reference
    private readonly DataContext _context;
    0 references
    public UsuariosController(DataContext context)
    {
        _context = context;
    }
}
```

- Exigirá o *using RpgApi.Data*
- Lembre-se que um construtor é um método padrão que tem o mesmo nome na classe e nele conseguimos inicializar variáveis ou realizar operações que acontecerão assim que a classe for instanciada para virar um objeto por algum componente do sistema que a utilize, como quando testamos através do *Postman*, por exemplo.

3. Crie um método interno que verificará se um usuário existe no banco de dados.

```
public async Task<bool> UsuarioExistente(string username)
{
    if(await _context.Usuarios.AnyAsync(x => x.Username.ToLower() == username.ToLower()))
    {
        return true;
    }
    return false;
}
```

- Exigirá os usings *System.Threading.Tasks* e *Microsoft.EntityFrameworkCore*.



4. Crie um método `HttpPost` com rota chamada **Registrar**. O método verificará se o usuário existe, caso exista, retornará uma mensagem ao usuário, caso contrário irá criptografar a senha gerando o *hash* e o *salt* e depois enviará os dados para o base de dados. Faça o using `RpgApi.Models` e `RpgApi.Utils`.

```
[HttpPost("Registrar")]
0 references
public async Task<IActionResult> RegistrarUsuario(Usuario user)
{
    try
    {
        if (await UsuarioExistente(user.Username))
            throw new System.Exception("Nome de usuário já existe");

        Criptografia.CriarPasswordHash(user.PasswordString, out byte[] hash, out byte[] salt);
        user.PasswordString = string.Empty;
        user.PasswordHash = hash;
        user.PasswordSalt = salt;
        await _context.Usuarios.AddAsync(user);
        await _context.SaveChangesAsync();

        return Ok(user.Id);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

5. Abra a classe Criptografia e adicione um método que receberá a senha como uma *string* simples e vai comparar com que estará na base de dados.

```
public static bool VerificarPasswordHash(string password, byte[] hash, byte[] salt)
{
    using (var hmac = new System.Security.Cryptography.HMACSHA512(salt))
    {
        var computedHash =
hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));
        for (int i = 0; i < computedHash.Length; i++)
        {
            if (computedHash[i] != hash[i])
            {
                return false;
            }
        }
        return true;
    }
}
```



6. Crie um método HttpPost com rota chamada **Autenticar**.

```
[HttpPost("Autenticar")]
0 references
public async Task<IActionResult> AutenticarUsuario(Usuario credenciais)
{
    try
    {
        Usuario usuario = await _context.Usuarios
            .FirstOrDefaultAsync(x => x.Username.ToLower().Equals(credenciais.Username.ToLower()));

        if (usuario == null)
        {
            throw new System.Exception("Usuário não encontrado.");
        }
        else if (!Criptografia
            .VerificarPasswordHash(credenciais.PasswordString, usuario.PasswordHash, usuario.PasswordSalt))
        {
            throw new System.Exception("Senha incorreta.");
        }
        else
        {
            return Ok(usuario.Id);
        }
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

- O método acima consultará o login no banco de dados, e caso este login não exista, retornará mensagem. Caso o login exista, este login e senha serão enviados para a API, sendo criptografados e comparados com os registros do banco de dados. Se a senha for incorreta, retornará mensagem. Caso os dados estejam corretos, será devolvido o Id deste usuário.



7. Realize o teste no *postman* para salvar um usuário e confira na base de dados. Depois tente realizar a operação com os mesmos dados, que conforme validação, não permitirá que o mesmo *username* seja registrado. Ao salvar os dados será retornado o Id do usuário inserido.

POST http://localhost:5000/Usuarios/Registrar Send

Params Auth Headers (9) Body Pre-req. Tests Settings Cookies

raw JSON

1 {
2 "Username": "UsuarioAdmin",
3 "PasswordString": "123456"
4 }

Body 200 OK 3.68 s 149 B Save Response

Pretty Raw Preview Visualize JSON

1 1

- Consulte o banco de dados para visualizar como ficam as colunas do hash e salt da senha. O conteúdo são valores hexadecimais (0 a 9 e 'A' até 'F') que combinados entre si, representam a senha devidamente criptografada.
- Realize o teste no postman para tentar autenticar um usuário crie as três situações possíveis, usuário inexistente, senha incorreta e usuário e senha aceitos para constatar que a programação realizada está funcionando corretamente.

8. Testando o cadastro de Armas:

POST http://localhost:5000/Armas Send

Params Auth Headers (9) Body Pre-req. Tests Settings

raw JSON

1 {
2 "Nome": "Arco e Flecha",
3 "Dano": 70,
4 "PersonagemId": 7
5 }

Body 200 OK 104 ms 149 B Save Response

Pretty Raw Preview Visualize JSON

1 5

- Tente salvar a arma com o mesmo Id de personagem e verá que ocorrerá um erro de chave estrangeira, pois um mesmo personagem só poderá ter uma arma por vez. Outro teste é tentar salvar a arma com o Id de um personagem que não existe.



9. Para evitar o erro no postman, altere o método que adiciona uma arma no contexto do banco de dados editando a classe ArmasController.cs. Será colocada uma validação para que toda arma inserida tenha um Personagem que existe na base de dados. Repita esta validação no Update, salve os arquivos, execute a API e realize o teste no postman.

```
[HttpPost]
0 references
public async Task<IActionResult> Add(Arma novaArma)
{
    try
    {
        if (novaArma.Dano == 0)
            throw new System.Exception("O dano da arma não pode ser 0");

        Personagem p = await _context.Personagens
            .FirstOrDefaultAsync(p => p.Id == novaArma.PersonagemId);

        if(p == null)
            throw new System.Exception("Não existe personagem com o Id informado.");

        await _context.Armas.AddAsync(novaArma);
        await _context.SaveChangesAsync();
        return Ok(novaArma.Id);
    }
}
```

10. Agora crie a classe **PersonagemHabilidadesController.cs** dentro da pasta controller. Utilize o escopo básico de criação de controller abaixo.

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
namespace RpgApi.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class PersonagemHabilidadesController : ControllerBase
    {
        //Codificação geral dentro do corpo da controller.

    }
}
```



11. Programe no corpo da controller a declaração do contexto do banco de dados (A) e crie o construtor para inicializar o atributo do contexto do banco de dados. Utilize o using RpgApi.Data.

```
private readonly DataContext _context; A
0 references
public PersonagemHabilidadesController(DataContext context) B
{
    _context = context;
}
```

12. Faremos a programação do método para salvar na base de dados com comentários logo abaixo. Primeiro crie a estrutura do método e o try/catch. Uma boa prática é inserir a palavra async no final das nomenclaturas dos métodos. Exigirá o using System.Threading.Tasks e RpgApi.Models

```
[HttpPost]
0 references
public async Task<IActionResult> AddPersonagemHabilidadeAsync(PersonagemHabilidade novoPersonagemHabilidade)
{
    try
    {
        //Código aqui
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



13. Realize a programação dentro do bloco try. Exigirá o *using Microsoft.EntityFrameworkCore*

```
try
{
    Personagem personagem = await _context.Personagens
        .Include(p => p.Arma)
        .Include(p => p.PersonagemHabilidades).ThenInclude(ps => ps.Habilidade)
        .FirstOrDefaultAsync(p => p.Id == novoPersonagemHabilidade.PersonagemId);

    A if (personagem == null)
        throw new System.Exception("Personagem não encontrado para o Id Informado.");

    Habilidade habilidade = await _context.Habilidades
        .FirstOrDefaultAsync(h => h.Id == novoPersonagemHabilidade.HabilidadeId);

    B if (habilidade == null)
        throw new System.Exception("Habilidade não encontrada.");

    PersonagemHabilidade ph = new PersonagemHabilidade();
    ph.Personagem = personagem;
    ph.Habilidade = habilidade;
    C await _context.PersonagemHabilidades.AddAsync(ph);
    int linhasAfetadas = await _context.SaveChangesAsync();

    return Ok(linhasAfetadas);
}
```

- (A) Buscamos o personagem no contexto do banco de dados, incluindo as armas que ele tem, e as habilidades através da relação na tabela PersonagemHabilidades, utilizando como parâmetros o id que será enviado por quem requisita o método (o postman nos casos dos testes). O id do usuário do personagem deverá ser igual ao presente na claim do Token, caso contrário retornaremos *bad request*.
- (B) Buscamos a habilidade no contexto do banco de dados através do id indicado por quem requisita o método (o postman nos casos de testes), caso não exista a habilidade retornaremos *bad request*.
- (C) Preenchemos o objeto que enviaremos para o contexto de base de dados com os objetos adquiridos nas etapas A e B.



14. Execute o teste no postman enviando os dados abaixo.

POST <http://localhost:5000/PersonagemHabilidades>

Params Authorization Headers (9) **Body** ● Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```
1 {
2     "PersonagemId": 1,
3     "HabilidadeId": 1
4 }
```

- Caso aconteça erro, pode ser devido a chave composta criada na base de dados que não permite a repetição de uma mesma habilidade para o mesmo personagem. Envie uma habilidade diferente para o personagem, assim, é esperado que ela seja salva.

15. Execute o método para obter o personagem por Id.

GET <http://localhost:5000/Personagem/1>

- Observe no resultado que os dados da arma pertencente ao personagem vêm nulo, bem como a lista de habilidades que ele possui também está nula. Para trazer esses dados preenchidos, vamos modificar o método GetSingle na controller de personagem conforme abaixo. Após isso, tente a busca no postman

```
[HttpGet("{id}")] //Busca pelo id
0 references
public async Task<IActionResult> GetSingle(int id)
{
    try
    {
        Personagem p = await _context.Personagens
            .Include(ar => ar.Arma)//Inclui na propriedade Arma do objeto p
            .Include(ph => ph.PersonagemHabilidades)
                .ThenInclude(h => h.Habilidade)//Inclui na lista de PersonagemHabilidade de p
            .FirstOrDefaultAsync(pBusca => pBusca.Id == id);

        return Ok(p);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



16. É bem provável que tenha acontecido um erro, pois como a classe Personagem e Arma se referenciam entre si, provoca um looping. Faça a instalação do pacote do JSON Converter através da linha de comando **dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson -v 5.0.15**, abra a classe Startup.cs e adicione o trecho de código abaixo. Servirá para evitar loopings nas classes que referenciam umas às outras nos relacionamentos.

```
services.AddControllers().AddNewtonsoftJson(options =>
options.SerializerSettings.ReferenceLoopHandling=Newtonsoft.Json.ReferenceLoopHandling.Ignore
);
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<DataContext>(x => x.UseSqlServer(Configuration.GetConnectionString("ConexaoSomee")));

    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "RpgApi", Version = "v1" });
    });

    services.AddControllers().AddNewtonsoftJson(options =>
        options.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore
    );
}
```



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

Desafios

- (1) Criar um método Put com rota “AlterarSenha” na classe **UsuariosController.cs** que criptografe e altere a senha do usuário no banco e faça com que ele consiga autenticar.
- (2) Criar um método Get para listar todos os Usuarios na classe **UsuariosController.cs**
- (3) Na classe **UsuariosController.cs**, altere o método autenticar para que na linha anterior ao retornar o Id, a propriedade data de acesso seja alimentada com a data/hora atual e salvar alterações no Banco via EF.
- (4) Inserir no método GetSingle da controller de personagem uma forma de exibir o usuário que o personagem pertence. Using de System.Collections.Generic.
- (5) Criar um método na classe **PersonagemHabilidadesController.cs** que retorne uma lista de PersonagemHabilidade de acordo com o id do personagem passado por parâmetro. Using de System.Collections.Generic e System.Linq
- (6) Criar um método na classe **PersonagemHabilidadesController.cs** que retorne uma lista de Habilidades com a rota chamada GetHabilidades.
- (7) Criar um método na controller **PersonagemHabilidadesController.cs** que remova os dados da tabela PersonagemHabilidades. Esse método terá que ser do tipo Post (com rota chamada **DeletePersonagemHabilidade**) pelo fato de ter que receber o objeto como parâmetro, contendo o id do personagem e da habilidade. Use o *FirstOrDefaultAsync* que exige o using System.Linq.

Como deverá ser a entrega

- Você deverá testar as ações localmente e depois fazer o deploy no somee. A entrega deverá ser os Prints do postman contendo o seu endereço da requisição apontando para o somee e o resultado obtido em cada uma das operações citadas acima. Você pode juntá-los todos num documento Word para anexar na atividade.



Resolução dos Desafios

- (1) Método para alteração da senha de um usuário existente.

```
[HttpPost("AlterarSenha")]
0 references
public async Task<IActionResult> AlterarSenhaUsuario(Usuario credenciais)
{
    try
    {
        Usuario usuario = await _context.Usuarios //Busca o usuário no banco através do login
            .FirstOrDefaultAsync(x => x.Username.ToLower().Equals(credenciais.Username.ToLower()));

        if (usuario == null) //Se não achar nenhum usuário pelo login, retorna mensagem.
            throw new System.Exception("Usuário não encontrado.");

        CriarPasswordHash(credenciais.PasswordString, out byte[] hash, out byte[] salt);
        usuario.PasswordHash = hash; //Se o usuário existir, executa a criptografia (linha 122)
        usuario.PasswordSalt = salt; //guardando o hash e o salt nas propriedades do usuário (linhas 123/124)

        _context.Usuarios.Update(usuario);
        int linhasAfetadas = await _context.SaveChangesAsync(); //Confirma a alteração no banco
        return Ok(linhasAfetadas); //Retorna as linhas afetadas (Geralmente sempre 1 linha msm)
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

- (2) Método para listar todos os usuários

```
[HttpGet("GetAll")]
0 references
public async Task<IActionResult> GetUsuarios()
{
    try
    {
        //List exigirá o using System.Collections.Generic
        List<Usuario> lista = await _context.Usuarios.ToListAsync();
        return Ok(lista);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



- (3) Atualização da data de Acesso no método “Autenticar”. Esse método já existe, bastará inserir a operação sinalizada no print. O teste a ser feito é compilar, executar a API e realizar a chamada para a autenticação. Se der Ok, faça um select na tabela de Usuários para verificar se a data de acesso foi gravada.

```
[HttpPost("Autenticar")]
0 references
public async Task<IActionResult> AutenticarUsuario(Usuario credenciais)
{
    try
    {
        Usuario usuario = await _context.Usuarios
            .FirstOrDefaultAsync(x => x.Username.ToLower().Equals(credenciais.Username.ToLower()));

        if (usuario == null)
            throw new System.Exception("Usuário não encontrado.");
        else if (!VerificarPasswordHash(credenciais.PasswordString, usuario.PasswordHash, usuario.PasswordSalt))
            throw new System.Exception("Senha incorreta.");
        else
        {
            usuario.DataAcesso = System.DateTime.Now;
            _context.Usuarios.Update(usuario);
            await _context.SaveChangesAsync(); //Confirma a alteração no banco
            return Ok(usuario.Id);
        }
    }
}
```

- (4) Verifique se sua tabela de usuários tem algum usuário registrado, caso não tenha nenhum, rode seu projeto e através do postman, faça o registro de um usuário (conforme abaixo) e verifique no banco qual foi o Id gerado para este usuário. Se você já tinha um usuário no banco, verifique qual o Id dele.

```
POST http://localhost:5000/Usuarios/Registrar... Send

Params Auth Headers (9) Body ● Pre-req. Tests Settings Cookies

raw ▼ JSON ▼ Beautify

1 {"Username": "UsuarioAdmin", "PasswordString": "123456"}
2 {"Username": "UsuarioAdmin", "PasswordString": "123456"}
3 {"Username": "UsuarioAdmin", "PasswordString": "123456"}
4 {"Username": "UsuarioAdmin", "PasswordString": "123456"}
```

Acesse o seu banco de dados, abra a sua tabela de Personagens para atribuir o Id do usuário que você verificou na etapa anterior para todos os seus personagens já salvos



Programação da exibição do usuário que o personagem pertence no método GetSingle de PersonagensController:

```
[HttpGet("{id}")] //Busca pelo id
0 references
public async Task<IActionResult> GetSingle(int id)
{
    try
    {
        Personagem p = await _context.Personagens
            .Include(ar => ar.Arma)//Inclui na propriedade Arma do objeto p
            .Include(us => us.Usuario)//Inclui na propriedade Usuario do objeto p
            .Include(ph => ph.PersonagemHabilidades)
                .ThenInclude(h => h.Habilidade)//Inclui na lista de PersonagemHabilidade de p
            .FirstOrDefaultAsync(pBusca => pBusca.Id == id);

        return Ok(p);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

Teste através do postman

GET http://localhost:5000/Personagens/1

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	... Bulk Edit
Body	Pretty Raw Preview Visualize JSON	Status: 200 OK Time: 38 ms Size: 958 B	Save Response

1 {
2 "id": 1,
3 "nome": "Frodo",
4 "pontosVida": 100,
5 "forca": 10,
6 "defesa": 10,
7 "inteligencia": 10,
8 "classe": 0,
9 "fotoPersonagem": null,
10 "usuario": {
11 "id": 1,
12 "username": "UsuarioAdmin",
13 "passwordHash": "WB\$vyHXb6jkvW46myHRUuU5E3D0/yVtDqbUkZofb3vd0BMu5LgJ4U2/4RSXHrf+5/Ii0Rb935mZwS6SBVotK5Q==",
14 "passwordSalt": "Awce7bpv5TwFct84k9U/_VJfqXuIDb6UXxR07Da93Hdky0B8VNgRsR538Ibgzfm2KMzF18QKdYDxoalJFh0UFkzEKwp01yagytJyqZ0lesvyEEiyTtpv6cbQ4vL07H/gcg
+80mYydUT0+AJjtRwmrkQCR1Vg8gjP4/ciJtNnQKLlg=",
15 "foto": null,
16 "latitude": null,



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

-
- (5) Busca da listagem de PersonagemHabilidades de um determinado personagem passando o id do personagem por parâmetro.

```
[HttpGet("{personagemId}")]
public async Task<IActionResult> GetHabilidadesPersonagem(int personagemId)
{
    try
    {
        List<PersonagemHabilidade> phLista = new List<PersonagemHabilidade>();
        phLista = await _context.PersonagemHabilidades
            .Include(p => p.Personagem)
            .Include(p => p.Habilidade)
            .Where(p => p.Personagem.Id == personagemId).ToListAsync();
        return Ok(phLista);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

- (6) Busca de todas as habilidades cadastradas na tabela de Habilidades através da controller PersonagemHabilidades. Não faremos uma controller para interagir com as habilidades, a medida que quisermos, faremos a adição diretamente nas tabelas do banco de dados.

```
[HttpGet("GetHabilidades")]
public async Task<IActionResult> GetHabilidades()
{
    try
    {
        List<Habilidade> habilidades = new List<Habilidade>();
        habilidades = await _context.Habilidades.ToListAsync();
        return Ok(habilidades);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

-
- (7) Remoção da habilidade do personagem em **PersonagensController**. Using de System.Collections.Generic e System.Linq. A busca é feita através do Id do personagem e do Id da Habilidade presente no objeto ph.

```
[HttpPost("DeletePersonagemHabilidade")]
public async Task<IActionResult> DeleteAsync(PersonagemHabilidade ph)
{
    try
    {
        PersonagemHabilidade phRemover = await _context.PersonagemHabilidades
            .FirstOrDefaultAsync(phBusca => phBusca.PersonagemId == ph.PersonagemId
            && phBusca.HabilidadeId == ph.HabilidadeId);
        if(phRemover == null)
            throw new System.Exception("Personagem ou Habilidade não encontrados");

        _context.PersonagemHabilidades.Remove(phRemover);
        int linhasAfetadas = await _context.SaveChangesAsync();
        return Ok(linhasAfetadas);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



Aula 12 - Autenticação via Token com JSON Web Tokens (JWT)

Nossas aplicações até o momento tem todos os métodos programados com acesso livre na controller, mas podemos fazer com que os métodos das controllers sejam acessados apenas se houver uma autenticação válida, porém, esse processo acarretaria a necessidade de entrar com as credenciais a cada método solicitado.

Para facilitar os procedimentos, podemos utilizar um token que pode ficar armazenado no navegador de uma aplicação ou de um dispositivo, sendo que este token conterá informações essenciais de identificação de um usuário, mas não informações sensíveis que podem ser roubadas, além disso, o token pode ser criado já com uma data de validade atribuída.

JSON Web Tokens é um recurso muito útil para tudo isso que falamos até então e faremos a aplicação prática na API, mas para que conheça um pouco da teoria do que estamos falando, segue um vídeo para introdução sobre Token e JWT:

Token: <https://youtu.be/LtVb9rhU41c>

JWT: <https://youtu.be/Gyq-yeot8qM>

Artigo: <https://www.brunobrito.net.br/jwt-cookies-oauth-bearer/>

O JWT nada mais é do que uma série de caracteres que contém especificações sobre um usuário, chamamos essas especificações de Claims (Reivindicações), ou seja, através desses dados é possível saber quais os tipos de acesso que determinado usuário poderá ter numa API, por exemplo, ou simplesmente resgar informações armazenadas no token gerado.

1. Abra o projeto RpgApi, vá até o arquivo appsettings.json e insira a codificação que será a chave para gerar o token

```
{  
    "ConfiguracaoToken": {  
        "Chave": "minha chave super secreta"  
    },  
  
    "ConnectionStrings": {  
        "ConexaoSomee": "workstation id=DB-DS"
```

2. Utilize o terminal para instalar os pacotes necessários para programação do método que vai gerar o Token:
→ dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer (-v 5.0.15 se for versão 5)
→ dotnet add package System.IdentityModel.Tokens.Jwt
→ dotnet add package Microsoft.IdentityModel.Tokens

Perceba que ao abrir o arquivo RpgApi.csProj, as referências aos pacotes estarão dentro da tag **ItemGroup** conforme a seguir:

```
<PackageReference Include="Microsoft.IdentityModel.Tokens" Version="6.17.0"/>  
<PackageReference Include="System.IdentityModel.Tokens.Jwt" Version="6.17.0"/>  
<PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="5.0.15"/>
```



-
3. Altere o construtor da classe UsuariosController para permitir acesso as configurações criadas anteriormente. Utilize o using *Microsoft.Extensions.Configuration*

```
private readonly DataContext _context;
1 reference
private readonly IConfiguration _configuration;
0 references
public UsuariosController(DataContext context, IConfiguration configuration)
{
    _context = context;
    _configuration = configuration;
}
```

4. Na classe UsuariosController desenvolva o método que criará o token. Usings: System.Secutiry.Claims, Microsoft.IdentityModel.Tokens, System.Text, System.IdentityModel.Tokens.Jwt,

```
private string CriarToken(Usuario usuario)
{
    List<Claim> claims = new List<Claim>
    {
        new Claim(ClaimTypes.NameIdentifier, usuario.Id.ToString()),
        new Claim(ClaimTypes.Name, usuario.Username)
    };
    SymmetricSecurityKey key = new SymmetricSecurityKey(Encoding.UTF8
        .GetBytes(_configuration.GetSection("ConfiguracaoToken:Chave").Value));

    SigningCredentials creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha5
12Signature);
    SecurityTokenDescriptor tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(claims),
        Expires = DateTime.Now.AddDays(1),
        SigningCredentials = creds
    };
    JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler();
    SecurityToken token = tokenHandler.CreateToken(tokenDescriptor);

    return tokenHandler.WriteToken(token);
}
```



5. Altere o retorno do método de autenticação para que no retorn Ok, usemos o método criar Token

```
public async Task<IActionResult> AutenticarUsuario(Usuario credenciais)
{
    try
    {
        Usuario usuario = await _context.Usuarios
            .FirstOrDefaultAsync(x => x.Username.ToLower().Equals(credenciais.Username.ToLower()));

        if (usuario == null)
            throw new System.Exception("Usuário não encontrado.");
        else if (!VerificarPasswordHash(credenciais.PasswordString, usuario.PasswordHash, usuario.PasswordSalt))
            throw new System.Exception("Senha incorreta.");
        else
        {
            usuario.DataAcesso = System.DateTime.Now;
            _context.Usuarios.Update(usuario);
            await _context.SaveChangesAsync(); //Confirma a alteração no banco

            //return Ok(usuario.Id);
            return Ok(CriarToken(usuario));
        }
    }
    catch (System.Exception ex)
    {return BadRequest(ex.Message);}
}
```

6. Abra o postman e realize o teste de Autenticação de um usuário já salvo na base, confirmando que será exibido o token, caso a autenticação tenha sucesso. Você pode conferir os dados existentes no Token copiando-o e usando no site <https://jwt.io/>
7. Abra a classe Startup.cs e faça a edição do método ConfigureServices como segue abaixo. Esta etapa será importante para que possamos resgatar as informações do Token. Necessário using para Microsoft.AspNetCore.Authentication.JwtBearer; Microsoft.IdentityModel.Tokens e System.Text.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<DataContext>(x => x.UseSqlServer(Configuration.GetConnectionString("ConexaoSomee")));

    services.AddControllers();

    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII
                .GetBytes(Configuration.GetSection("ConfiguracaoToken:Chave").Value)),
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });

    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "RpgApi", Version = "v1" });
    });
}
```



8. Ainda na classe Startup.cs, adicione a linha que implanta a Autenticação como a seguir

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();
        app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "RpgApi v1"));
    }

    app.UseHttpsRedirection();

    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();
}
```

9. Vá até a controller de Usuários e adicione no topo da classe o atributo de Autorização. Com o atributo Authorize iremos limitar quem pode acessar a controller ou algum método dentro da Controller. Iniciaremos realizando os procedimentos de configuração.

```
[Authorize]
[ApiController]
[Route("[controller]")]
0 references
public class UsuariosController : ControllerBase
{
    5 references
    private readonly DataContext _context;
```

- Será necessário o *using Microsoft.AspNetCore.Authorization*.
- Tente realizar o teste do método de autenticação no postman para verificar qual status é retornado.

10. Provavelmente, o teste da etapa anterior deve ter retornado uma mensagem 401 (Unauthorized). Vamos inserir uma permissão para apenas para os métodos “**Autenticar**” e “**Registrar**” conforme a seguir

```
[AllowAnonymous]
[HttpPost("Autenticar")]
0 references
public async Task<IActionResult> AutenticarUsuario(Usuario credenciaisUsuario)
{
    Usuario usuario = await _context.Usuarios.FirstOrDefaultAsync(x =>
        x.Username.ToLower().Equals(credenciaisUsuario.Username.ToLower()));
```

- Esse atributo concede uma exceção para que um usuário anônimo consiga acessar o método contido numa controller restrita.
- Realize novamente o teste e copie o token gerado e guarde em um bloco de notas para usarmos, quando necessário.



11. Adicione o atributo Authorize no topo da controller de Personagem também

```
[Authorize]
[ApiController]
[Route("[controller]")]
0 references
public class PersonagemController : ControllerBase
{
```

- Será necessário o *using Microsoft.AspNetCore.Authorization*

12. Configuraremos o postman para testar o Método GetAll de Personagens passando os dados do Token no cabeçalho da requisição http, conforme as configurações feitas na sinalização em verde. A sinalização em Azul é o resultado obtido e a sinalização em vermelho são as configurações gerais que usamos para métodos get.

The screenshot shows the Postman interface with a GET request to `http://localhost:5000/Personagens/GetAll`. The `Authorization` tab is selected, showing a `Bearer Tok...` type and a token value `eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.e...`. The response status is `200 OK`.

- Perceba que adicionamos a key (chave) “Authorization” e no Type a palavra “Bearer” (indica que é uma autorização de token Jwt), com um espaço e depois o token gerado anteriormente. Dessa maneira, a API reconhecerá que existe permissão para acessar os métodos da controller.

Obtendo os Personagens de acordo com a leitura das Reivindicações de Usuário (Claims)

Para esta etapa, sua tabela de usuários deverá ter mais de um usuário cadastrado, se o cenário não for esse, faça o cadastro de usuários via Postman e não esqueça de usar o token conforme o item anterior.

Sua tabela de personagens também deve ter mais que um personagem, caso o cenário não seja esse, insira via SQL e atribua na coluna `Usuarioid` de todos os personagens, algum Id de usuário existente criando variações. Isso será feito manualmente agora, pois vamos usar o relacionamento criado na aula anterior em que um personagem pertence a um usuário e um usuário pode ter vários personagens, mas nas próximas aulas automatizaremos o processo de inserção de personagem identificando o usuário para o postman.



13. Programe o método GetByIdUser na controller de Personagem.

```
[HttpGet("GetByIdUser")]
0 references
public async Task<IActionResult> GetByIdUserAsync()
{
    try
    {
        1 int id = int.Parse(User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier).Value);

        List<Personagem> lista = await _context.Personagens
        2 .Where(u => u.Usuario.Id == id).ToListAsync();

        return Ok(lista);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

- Necessário using de System.Linq e System.Security.Claims

- (1) Quando criamos o token criamos também Claims, sendo que a identificada como NameIdentifier foi atrelada ao Id do usuário, logo ao recuperar esta claim do Token, conseguimos o Id do usuário.
- (2) O método que vai no Banco de dados para recuperar os personagens tem um filtro em que é passado o Id do usuário, desta forma recuperaremos os personagens atrelados a Id que a Claim continha.

14. Faça a configuração para testar via postman, utilizando o token. Não esqueça que o método criado tem uma rota diferente.

GET Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Type Bearer Token ! Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables [↗](#)

The authorization header will be automatically generated when you send the request. [Learn more about authorization ↗](#)

eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJ...ZQ



Identificação centralizada do Usuário

Ao invés de criar a codificação para identificar o usuário em cada método, criaremos a seguir um método para coletar os dados do usuário através do token que poderá ser usado nos demais métodos da controllers, e caso tenhamos que usar em outras controllers, será desenvolvido da mesma maneira.

15. Abra a classe Startup.cs e adicione a codificação abaixo no método **ConfigureServices** responsável por configurar o acesso a um objeto em memória de maneira única para toda a aplicação. Chamamos isso de Singleton.

```
services.AddSingleton< IHttpContextAccessor, HttpContextAccessor>();
```

- Necessário o using Microsoft.AspNetCore.Http / Microsoft.Extensions.DependencyInjection.Extensions

16. Modifique a controller PersonagemController declarando a variável do tipo IHttpContextAcessor e inicializando-a no construtor

```
private readonly DataContext _context;
1 reference
private readonly IHttpContextAccessor _httpContextAccessor;

0 references
public PersonagemController(DataContext context, IHttpContextAccessor httpContextAccessor)
{
    _context = context;
    _httpContextAccessor = httpContextAccessor;
}
```

- Necessário o using Microsoft.AspNetCore.Http.
- O código abaixo criará a condição para quando esta classe for instanciada em memória, possamos acessar a configuração Singleton.

17. Crie um método que centralizará a ação de obter o Id do usuário. Observe o método está se utilizando da variável criada na etapa anterior para coletar o dado identificador da Claim, o Id.

```
private int ObterUsuarioId()
{
    return int.Parse(_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier));
}
```

Desafio aula 12: Com o relacionamento existente entre as tabelas usuários e personagens, a coluna Usuarioid da base de dados não está sendo preenchida automaticamente ao fazer um post de personagens. Com base nas linhas de programação realizadas nesta aula, verifique uma forma de identificar o id do usuário que está autenticado, carregue um objeto do tipo Usuário através deste id e atribua na propriedade usuário do objeto do tipo Personagem quando for salvar e editar um personagem. Dica: A modificação deve ser feita na controller de Personagem.



Resolução do Desafio da Aula 12

Para que o Id do usuário fique salvo na tabela de personagens, faremos busca do id através das Claims obtidas no Token, depois buscaremos no Banco de Dados, guardando na propriedade Usuário existente no objeto do tipo Personagem.

Salvando o Personagem com o Id do Usuário

```
[HttpPost]  
0 references  
public async Task<IActionResult> Add(Personagem novoPersonagem)  
{  
    try  
    {  
        if (novoPersonagem.PontosVida > 100)  
        {  
            throw new System.Exception("Pontos de vida não pode ser maior que 100");  
        }  
  
        int usuarioId = int.Parse(_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier));  
        novoPersonagem.Usuario = _context.Usuarios.FirstOrDefault(uBusca => uBusca.Id == usuarioId);  
  
        await _context.Personagens.AddAsync(novoPersonagem);  
        await _context.SaveChangesAsync();  
  
        return Ok(novoPersonagem.Id);  
    }  
    catch (System.Exception ex)  
    {  
        return BadRequest(ex.Message);  
    }  
}
```

Atualizando o Personagem com o Id do usuário preenchido. Observe que desta vez o uso await/async

```
[HttpPut]  
0 references  
public async Task<IActionResult> Update(Personagem novoPersonagem)  
{  
    try  
    {  
        if (novoPersonagem.PontosVida > 100)  
        {  
            throw new System.Exception("Pontos de vida não pode ser maior que 100");  
        }  
  
        int usuarioId = int.Parse(_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier));  
        novoPersonagem.Usuario = _context.Usuarios.FirstOrDefault(uBusca => uBusca.Id == usuarioId);  
  
        _context.Personagens.Update(novoPersonagem);  
        int linhaAfetadas = await _context.SaveChangesAsync();  
  
        return Ok(linhaAfetadas);  
    }  
    catch (System.Exception ex)  
    {  
        return BadRequest(ex.Message);  
    }  
}
```



- Para testar via postman é necessário fazer a autenticação do Usuário, copiar o Token e preencher o valor do Token no Header ao cadastrar ou atualizar o novo personagem, conforme fizemos anteriormente.
- Conseguimos realizar esse salvamento e atualização devido o relacionamento um para muitos, presente entre a classe usuário e personagem criado nas aulas anteriores, refletido no banco de dados quando fizemos a migração.

Agora, vamos pensar juntos... durante a aula criamos um método que já traz o id do usuário, logo, podemos reescrever o método de salvamento e de atualização utilizando o método que obtém o id do usuário simplificando e deixando de repetir código, através do método que centraliza a obtenção do Id.

```
[HttpPost]  
0 references  
public async Task<IActionResult> Add(Personagem novoPersonagem)  
{  
    try  
    {  
        if (novoPersonagem.PontosVida > 100)  
        {  
            throw new System.Exception("Pontos de vida não pode ser maior que 100");  
        }  
  
        int usuarioId = int.Parse(_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier));  
        novoPersonagem.Usuario = _context.Usuarios.FirstOrDefault(uBusca => uBusca.Id == usuarioId);  
  
        novoPersonagem.Usuario = _context.Usuarios.FirstOrDefault(uBusca => uBusca.Id == ObterUsuarioId());  
  
        await _context.Personagens.AddAsync(novoPersonagem);  
        await _context.SaveChangesAsync();  
  
        return Ok(novoPersonagem.Id);  
    }  
}
```

```
[HttpPut]  
0 references  
public async Task<IActionResult> Update(Personagem novoPersonagem)  
{  
    try  
    {  
        if (novoPersonagem.PontosVida > 100)  
        {  
            throw new System.Exception("Pontos de vida não pode ser maior que 100");  
        }  
  
        int usuarioId = int.Parse(_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier));  
        novoPersonagem.Usuario = _context.Usuarios.FirstOrDefault(uBusca => uBusca.Id == usuarioId);  
  
        novoPersonagem.Usuario = _context.Usuarios.FirstOrDefault(uBusca => uBusca.Id == ObterUsuarioId());  
  
        _context.Personagens.Update(novoPersonagem);  
        int linhaAfetadas = await _context.SaveChangesAsync();  
  
        return Ok(linhaAfetadas);  
    }  
}
```



Aula 13 – Autenticação baseada em Papéis

Nesta aula criaremos definições de perfis ou papéis para cada usuário, e assim, resgatar qual o tipo de usuário está requisitando operações para apresentar os dados e acessos que ele pode ter. Além disso, com a esta programação da API sendo realizada, será possível programar o projeto MVC para ter aplicar as chamadas da API.

1. Atualize a classe **Usuario** com a propriedade abaixo

```
[Required]  
0 references  
public string Perfil { get; set; }
```

- Será necessário o using System.ComponentModel.DataAnnotations. A anotação Required indica que o campo será not null quando ele existir no banco de dados.
2. Abra a classe DataContext e antes do fechamento do método **OnModelCreating**, faça a codificação que indicará que sempre que um Usuário for salvo sem identificação de Perfil, por padrão a descrição do perfil será a de jogador.

```
modelBuilder.Entity<Habilidade>().HasData  
{  
    new Habilidade(){Id=1, Nome="Adormecer", Dano=39},  
    new Habilidade(){Id=2, Nome="Congelar", Dano=41},  
    new Habilidade(){Id=3, Nome="Hipnotizar", Dano=37}  
};  
  
modelBuilder.Entity<Usuario>().Property(u => u.Perfil).HasDefaultValue("Jogador");
```

3. Execute o comando *dotnet ef migrations add MigracaoPerfil* para criar o arquivo de migração

- Observe que no método Up do arquivo de migração foi determinada que a propriedade nullable está com o valor false e com o defaultvalue como jogador.

```
protected override void Up(MigrationBuilder migrationBuilder)  
{  
    migrationBuilder.AddColumn<string>(  
        name: "Perfil",  
        table: "Usuarios",  
        nullable: false,  
        defaultValue: "Jogador");  
}
```

4. Execute a migração na base de dados: *dotnet ef database update*. Abra o banco de dados para visualizar a nova coluna na tabela usuários e note que o valor para todos os usuários está como Jogador, conforme previsto.



- Volte até a classe Usuario e comente a marcação Required para que ao usar o postman, não seja obrigatório ter que enviar o perfil no corpo da requisição.

Acrecentando mais uma claim para o Token: Até este momento, guardávamos na coleção de claims os Id do usuário e o login dele, faremos a programação para acrescentar uma claim para guardar Perfil.

- Abra a controller de usuário e modifique o método CriarToken

```
private string CriarToken(Usuario usuario)
{
    List<Claim> claims = new List<Claim>
    {
        new Claim(ClaimTypes.NameIdentifier, usuario.Id.ToString()),
        new Claim(ClaimTypes.Name, usuario.Username),
        new Claim(ClaimTypes.Role, usuario.Perfil)
    };
}
```

- Execute a API, faça o teste no postman. Copie o token gerado e use o site <https://jwt.io/> para ver o conteúdo da Claim, devendo constar o Perfil.
- Abra a controller de Personagem e altere a anotação Authorize, que indicará o tipo de usuário que poderá acessar a controller.

```
[Authorize(Roles = "Jogador")]
```

- Altere o usuário que você mais usa para o perfil Admin. Execute a API, autentique e tente buscar todos os personagens

Id	Username	PasswordHash	PasswordSalt	Perfil
1	UsuarioAdmin	0x31AE472BE700C0FB9E6D712F86A...	0x5FAA27F18506D9A16BC4A24EFD61DF74...	Admin
2	UsuarioTeste	0x97242B009D4D98EC3BE3E149339...	0xB01E4D66773B1FCBE8A4117F447802EA...	Jogador
3	UsuarioTeste2	0x3614DB7BE8750BE85EE3685715B...	0x5166F5E277FBC3583D000223C985C111F...	Jogador

- Abra o postman, faça a autenticação e use o Token para obter todos os personagens.

```
GET http://localhost:5000/Personagens/GetAll
```

- Você perceberá que vai dar o erro 403, de proibição. Adicione a palavra Admin nas permissões da controller, execute a API e teste novamente.

```
[Authorize(Roles = "Jogador, Admin")]
```



Filtrando Personagens de acordo com o perfil do usuário

12. Crie o método ObterPerfilUsuario na controller de personagem

```
private string ObterPerfilUsuario()
{
    return _httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.Role);
```

13. Crie o método GetByPerfilAsync para pegar os personagens de acordo com o perfil identificado. Observe que criamos a lista de maneira vazia em memória e consultamos qual o perfil do usuário, se ele for admin retornará todos os personagens, caso contrário vai filtrar os personagens que tem o id de usuário igual ao contido no token.

```
[HttpGet("GetByPerfil")]
0 references
public async Task<IActionResult> GetByPerfilAsync()
{
    try
    {
        List<Personagem> lista = new List<Personagem>();

        if(ObterperfilUsuario() == "Admin")
            lista = await _context.Personagens.ToListAsync();
        else
        {
            lista = await _context.Personagens
                .Where(p => p.Usuario.Id == ObterUsuarioId()).ToListAsync();
        }
        return Ok(lista);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

- Execute a API e tenha dois usuários criados em sua base, um admin e outro jogador, com Personagens que pertençam a estes usuários. Faça o teste logando com cada um destes usuários e use o token para requisitar ao método GetByPerfil para perceber que o resultado das consultas é diferente em cada caso.



Desafio 1 - Aula 13: Validação ao adicionar uma arma – Controller de Armas

- (1) Validação ao salvar uma arma: Alterar o método que adiciona uma arma no contexto do banco de dados. O mesmo pode ser feito no método que atualiza uma Arma. Necessário using de System.Linq.

```
[HttpPost]  
0 references  
public async Task<IActionResult> Add(Arma novaArma)  
{  
    try  
    {  
        if (novaArma.Dano == 0)  
        {  
            throw new System.Exception("O dano da arma não pode ser 0");  
        }  
  
        A Personagem personagem = await _context.Personagens  
            .FirstOrDefaultAsync(p => p.Id == novaArma.PersonagemId);  
  
        B if (personagem == null)  
            throw new System.Exception("Seu usuário não contém personagens com o Id do Personagem informado.");  
  
        C Arma buscaArma = await _context.Armas  
            .FirstOrDefaultAsync(a => a.PersonagemId == novaArma.PersonagemId);  
  
        D if (buscaArma != null)  
            throw new System.Exception("O Personagem selecionado já contém uma arma atribuída a ele.");  
  
        await _context.Armas.AddAsync(novaArma);  
        await _context.SaveChangesAsync();  
  
        return Ok(novaArma.Id);  
    }  
}
```

- (A) Buscamos o personagem com o Id informado no postman
(B) Se não achar ninguém com o Id informado gera bad request
(C) Busca na tabela de Armas uma arma com o Id de personagem informado.
(D) Se achar quer dizer que o personagem não pode ter mais uma arama (relacionamento 1 para 1)



Desafio 2 – Aula 13: Opção de usar Context Acessor para ler dados do Token

Alterar a Controller de Armas para que o construtor possibilite que criemos um método que busque o Id do usuário coletando os dados das claims através do httpContextAcessor

```
private readonly IHttpContextAccessor _httpContextAccessor;
0 references
public ArmasController(DataContext context, IHttpContextAccessor httpContextAccessor)
{
    _context = context;
    _httpContextAccessor = httpContextAccessor;
}
0 references
private int ObterUsuarioId()
{
    return int.Parse(_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier));
}
```

- Necessário usings *Microsoft.AspNetCore.Http* e *System.Security.Claims*;
- Deixe a controller com o atributo [Authorize] para exigir autenticação. Insira o using *Microsoft.AspNetCore.Authorization*;



Aula 14 – Classe Random e Logs - Preparando API para a disputa entre os personagens

1. Acrescente mais três propriedades na classe Personagem e salve o arquivo

```
public int Disputas { get; set; }
public int Vitorias { get; set; }
public int Derrotas { get; set; }
```

2. Crie a classe Disputa na pasta Models, adicionando uma lista de inteiros que armazenará o Id dos personagens envolvidos na disputa, que poderá ser mais que dois e uma lista de strings para armazenar os resultados, além das demais propriedades. Exigirá using System.Collections.Generic, System e System.ComponentModel.DataAnnotations.Schema,

```
public int Id { get; set; }
0 references
public DateTime? DataDisputa { get; set; }
0 references
public int AtacanteId { get; set; }
0 references
public int OponenteId { get; set; }
0 references
public string Narracao { get; set; }

[NotMapped]
0 references
public int HabilidadeId { get; set; }
[NotMapped]
0 references
public List<int> ListaIdPersonagens { get; set; } = new List<int>();
[NotMapped]
0 references
public List<string> Resultados { get; set; } = new List<string>();
```

- Observe que a propriedade Datetime está com uma interrogação. Isso vai determinar que ela poderá ter valor nulo. Isso chama-se nullable types e pode ser aplicado também a variáveis primitivas como int (int?), decimal(decimal?) para que ao invés de armazenar o valor mínimo, possam armazenar nulo.
 - Crie também uma propriedade do tipo string chamada **Email** na classe Usuario.
3. Abra a classe DataContext e faça o mapeamento da classe para o banco

```
public DbSet<Disputa> Disputas{ get; set; }
```
 4. Crie a migração no terminal através do comando “dotnet ef migrations add MigracaoDisputas” e depois execute o comando para atualizar o banco de dados “dotnet ef database update” conferindo se a tabela Personagens foi atualizada e a tabelas Disputas foi criada.



5. Crie uma controller chamada DisputasController.cs.

```
using Microsoft.AspNetCore.Mvc;
namespace RpgApi.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class DisputasController : ControllerBase
    {
        //Construtores e métodos aqui.
    }
}
```

6. Declare a variável do contexto do banco de dados e inicialize ela no construtor. O contexto necessitará do using RpgApi.Data

```
using Microsoft.AspNetCore.Mvc;
using RpgApi.Data;

namespace RpgApi.Controllers
{
    [ApiController]
    [Route("[controller]")]
    0 references
    public class DisputasController : ControllerBase
    {
        1 reference
        private readonly DataContext _context;
        0 references
        public DisputasController(DataContext context)
        {
            _context = context;
        }
    }
}
```

7. Dentro da controller, programe o método que fará um ataque a outro personagem usando uma Arma. Necessário using RpgApi.Models e System.Threading.Tasks. Observe que esse método tem uma rota.

```
[HttpPost("Arma")]
0 references
public async Task<IActionResult> AtaqueComArmaAsync(Disputa d)
{
    try
    {
        //Programação dos próximos passos aqui
        return Ok(d);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



8. Programe dentro do bloco try do método a busca no banco de dados do personagem que está atacando e do seu oponente, através dos dados que foram passados por parâmetro no objeto “d” do tipo Disputa

```
Personagem atacante = await _context.Personagens
    .Include(p => p.Arma)
    .FirstOrDefaultAsync(p => p.Id == d.AtacantId);

Personagem oponente = await _context.Personagens
    .FirstOrDefaultAsync(p => p.Id == d.OponenteId);
```

- Necessário o using Microsoft.EntityFrameworkCore.
9. Tendo identificado os personagens envolvidos na disputa, criaremos a lógica para definir o valor do ataque, o dano da arma e quanto isso custará em pontos de vida para o oponente. Usaremos uma classe randômica para gerar valores aleatórios envolvendo alguma das propriedades

```
A int dano = atacante.Arma.Dano + (new Random().Next(atacante.Forca));

B dano = dano - new Random().Next(oponente.Defesa);

if (dano > 0)
    C oponente.PontosVida = oponente.PontosVida - (int)dano;
if (oponente.PontosVida <= 0)
    D d.Narracao = $"{oponente.Nome} foi derrotado!";
```

- Necessário o using System.
- (A) Somamos o dano da arma do atacante a um valor aleatório da força do atacante no intervalo entre 0 e a força que ele tem no cadastro.
(B) Subtraímos pelo valor da defesa do oponente em um valor aleatório entre 0 e o valor da defesa do oponente.
(C) Se o valor for maior que 0, subtraímos o valor do dano nos pontos de vida do oponente
(D) Se o valor for menor ou igual a 0, guardamos a mensagem que o oponente foi derrotado.

10. Na sequência atualizaremos os dados do oponente, já que ele pode ter pontos de vida subtraídos.

```
_context.Personagens.Update(oponente);
await _context.SaveChangesAsync();
```



11. Insira um resumo do que aconteceu e salve na tabela de disputas antes do retorno do método. Necessário o using System.Text

```
StringBuilder dados = new StringBuilder();
dados.AppendFormat(" Atacante: {0}. ", atacante.Nome);
dados.AppendFormat(" Oponente: {0}. ", oponente.Nome);
dados.AppendFormat(" Pontos de vida do atacante: {0}. ", atacante.PontosVida);
dados.AppendFormat(" Pontos de vida do oponente: {0}. ", oponente.PontosVida);
dados.AppendFormat(" Arma Utilizada: {0}. ", atacante.Arma.Nome);
dados.AppendFormat(" Dano: {0}. ", dano);

d.Narracao += dados.ToString();
d.DataDisputa = DateTime.Now;
_context.Disputas.Add(d);
_context.SaveChanges();

return Ok(d);
```

12. Teste os dados no postman de acordo com os Ids de personagem que você tem na base de dados. O Personagem deve ter uma arma vinculada a ele. Caso seja necessário atualizar os dados dos personagens, abra a tabela no banco de dados e redefina os valores. Futuramente definiremos métodos para realizar estas operações.

POST http://localhost:5000/Disputas/Arma

Params Authorization Headers (8) Body ●

none form-data x-www-form-urlencoded

```
1 {
2   "AtacanteId":3,
3   "OponenteId": 4
4 }
```

Ataque através de Habilidades

13. Crie o método para realizar o ataque através das habilidades com a estrutura a seguir

```
[HttpPost("Habilidade")]
0 references
public async Task<IActionResult> AtaqueComHabilidadeAsync(Disputa d)
{
    try
    {
        //Programação dos próximos passos aqui
        return Ok(d);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



14. Programe dentro do bloco try antes do return Ok.

```
Personagem atacante = await _context.Personagens
    .Include(p => p.PersonagemHabilidades)
    .ThenInclude(ph => ph.Habilidade)
    .FirstOrDefaultAsync(p => p.Id == d.AtacantId);

Personagem oponente = await _context.Personagens
    .FirstOrDefaultAsync(p => p.Id == d.OponenteId);

PersonagemHabilidade ph = await _context.PersonagemHabilidades
    .Include(p => p.Habilidade)
    .FirstOrDefaultAsync(phBusca => phBusca.HabilidadeId == d.HabilidadeId
A && phBusca.PersonagemId == d.AtacantId);
```

(A) Busca da habilidade através do id da habilidade contida no objeto recebido por parâmetro na requisição e também do personagem informado.

```
if (ph == null)
B d.Narracao = $"{atacant.Nome} não possui esta habilidade";
else
{
C int dano = ph.Habilidade.Dano + (new Random().Next(atacant.Inteligencia));
    dano = dano - new Random().Next(oponente.Defesa);

    if (dano > 0)
        oponente.PontosVida = oponente.PontosVida - dano;
    if (oponente.PontosVida <= 0)
        d.Narracao += $"{oponente.Nome} foi derrotado!";

    _context.Personagens.Update(oponente);
    await _context.SaveChangesAsync();

    StringBuilder dados = new StringBuilder();
    dados.AppendFormat(" Atacante: {0}. ", atacante.Nome);
    dados.AppendFormat(" Oponente: {0}. ", oponente.Nome);
    dados.AppendFormat(" Pontos de vida do atacante: {0}. ", atacante.PontosVida);
    dados.AppendFormat(" Pontos de vida do oponente: {0}. ", oponente.PontosVida);
    dados.AppendFormat(" Habilidade Utilizada: {0}. ", ph.Habilidade.Nome);
    dados.AppendFormat(" Dano: {0}. ", dano);

D     d.Narracao += dados.ToString();
    d.DataDisputa = DateTime.Now;
    _context.Disputas.Add(d);
    _context.SaveChangesAsync();
}

return Ok(d);
}
```

(B) Se a habilidade não for encontrada para o atacante, guardamos na narração a mensagem.

(C) Caso contrário realizamos o ataque e atualizamos os dados.

(D) Salvamento dos dados da disputa na tabela correspondente.



15. O teste no postman deve ser realizado conforme a seguir. O personagem que está atacando deve possuir a habilidade informada no Postman

POST <http://localhost:5000/Disputas/Habilidade>

Params Authorization Headers (8) **Body** ● Pr

none form-data x-www-form-urlencoded raw

```
1 {  
2   "AtacanteId":5,  
3   "OponenteId": 2,  
4   "HabilidadeId": 1  
5 }
```

Adicional: Exemplo de Sorteio → Referência para estudo: <https://youtu.be/p4DS4SROf0E>

```
[HttpGet("PersonagemRandom")]
0 references
public async Task<IActionResult> Sorteio()
{
    List<Personagem> personagens =
        | await _context.Personagens.ToListAsync();

    //Sorteio com numero da quantidade de personagens
    int sorteio = new Random().Next(personagens.Count);

    //busca na lista pelo indice sorteado (Não é o ID)
    Personagem p = personagens[sorteio];

    string msg =
        string.Format("Nº Sorteado {0}. Personagem: {1}", sorteio, p.Nome);

    return Ok(msg);
}
```

- Using System.Collections.Generic



Desafio Aula 14: Criando uma disputa coletiva e atualização dos dados de cada disputa

Programar as etapas, testar localmente e publicar no Somee. Você fará o envio de um arquivo word/pdf contendo os prints do postman testando o endereço da sua API publicada no somee com os métodos de ataque com arma, com habilidade e disputa em grupo.

1. Criaremos um método que fará a disputa entre uma lista de Personagens informada pelos ids deles. A sequência do método ficará conforme a programação abaixo. Faça o using **System.Linq**

```
[HttpPost("DisputaEmGrupo")]
0 references
public async Task<IActionResult> DisputaEmGrupoAsync(Disputa d)
{
    try
    {
        //Busca na base dos personagens informados no parametro incluindo Armas e Habilidades
        List<Personagem> personagens = await _context.Personagens
            .Include(p => p.Arma)
            .Include(p => p.PersonagemHabilidades).ThenInclude(ph => ph.Habilidade)
            .Where(p => d.ListaIdPersonagens.Contains(p.Id)).ToListAsync();

        //Contagem de personagens vivos na lista obtida do banco de dados
        int qtdPersonagensVivos = personagens.FindAll(p => p.PontosVida > 0).Count;

        //Enquanto houver mais de um personagem vivo haverá disputa
        while (qtdPersonagensVivos > 1)
        {
            //ATENÇÃO: Todas as etapas a seguir devem ficar aqui dentro do While
        }
        //Código após o fechamento do While. Atualizará os pontos de vida,
        //disputas, vitórias e derrotas de todos personagens ao final das batalhas
        _context.Personagens.UpdateRange(personagens);
        await _context.SaveChangesAsync();

        return Ok(d); //retorna os dados de disputas
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

- ATENÇÃO: Toda codificação das etapas a seguir devem ser realizadas dentro do While



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

-
2. Realize a programação a seguir dentro do while e faça leitura dos comentários para ficar por dentro dos procedimentos.

```
//Enquanto houver mais de um personagem vivo haverá disputa
while (qtdPersonagensVivos > 1)
{
    //Seleciona personagens com pontos vida positivos e depois faz sorteio.
    List<Personagem> atacantes = personagens.Where(p => p.PontosVida > 0).ToList();
    Personagem atacante = atacantes[new Random().Next(atacantes.Count)];
    d.AtacantId = atacante.Id;

    //Seleciona personagens com pontos vida positivos, exceto o atacante escolhido e depois faz sorteio.
    List<Personagem> oponentes = personagens.Where(p => p.Id != atacante.Id && p.PontosVida > 0).ToList();
    Personagem oponente = oponentes[new Random().Next(opONENTES.Count)];
    d.OponenteId = oponente.Id;

    //declara e redefine a cada passagem do while o valor das variáveis que serão usadas.
    int dano = 0;
    string ataqueUsado = string.Empty;
    string resultado = string.Empty;

    //Sorteia entre 0 e 1: 0 é um ataque com arma e 1 é um ataque com habilidades
    bool ataqueUsaArma = (new Random().Next(1) == 0);

    if (ataqueUsaArma && atacante.Arma != null)
    {
        //Programação do ataque com arma
    }
    else if (atacante.PersonagemHabilidades.Count != 0)//Verifica se o personagem tem habilidades
    {
        //Programação do ataque com habilidade
    }
}
```



3. Procure o “if” do ataque com arma, pois programaremos a mesma lógica do método de ataque com arma, guardando o dano e o nome da arma nas variáveis criadas anteriormente

```
if (ataqueUsaArma && atacante.Arma != null)
{
    //Programação do ataque com arma caso o atacante possua arma (o != null) do if

    //Sorteio da Força
    dano = atacante.Arma.Dano + (new Random().Next(atacante.Forca));
    dano = dano - new Random().Next(oponente.Defesa); //Sorteio da defesa.
    ataqueUsado = atacante.Arma.Nome;

    if (dano > 0)
        oponente.PontosVida = oponente.PontosVida - (int)dano;

    //Formata a mensagem
    resultado =
        string.Format("{0} atacou {1} usando {2} com o dano {3}.", atacante.Nome, oponente.Nome, ataqueUsado, dano);
    d.Narracao += resultado; // Concatena o resultado com as narrações existentes.
    d.Resultados.Add(resultado); //Adiciona o resultado atual na lista de resultados.
}
```

4. Para o else, a configuração do ataque com habilidade ficará como a seguir

```
else if (atacante.PersonagemHabilidades.Count != 0)//Verifica se o personagem tem habilidades na lista dele
{
    //Programação do ataque com habilidade

    //Realiza o sorteio entre as habilidade existentes e na linha seguinte a seleciona.
    int sorteioHabilidadeId = new Random().Next(atacante.PersonagemHabilidades.Count);
    Habilidade habilidadeEscolhida = atacante.PersonagemHabilidades[sorteioHabilidadeId].Habilidade;
    ataqueUsado = habilidadeEscolhida.Nome;

    //Sorteio da inteligência somada ao dano
    dano = habilidadeEscolhida.Dano + (new Random().Next(atacante.Inteligencia));
    dano = dano - new Random().Next(oponente.Defesa); //Sorteio da defesa.

    if (dano > 0)
        oponente.PontosVida = oponente.PontosVida - (int)dano;

    resultado =
        string.Format("{0} atacou {1} usando {2} com o dano {3}.", atacante.Nome, oponente.Nome, ataqueUsado, dano);
    d.Narracao += resultado;
    d.Resultados.Add(resultado);
}

//Atenção: Aqui ficará a Programação da verificação do ataque usado e verificação se existe mais de um personagem vivo
}

//Código após o fechamento do While. Atualizará os pontos de vida,
//disputas, vitórias e derrotas de todos personagens ao final das batalhas
_context.Personagens.UpdateRange(personagens);
await _context.SaveChangesAsync();
```

- A próxima codificação deverá ficar abaixo da sinalização do retângulo verde.



5. Após o fechamento do else, faremos a programação a seguir.

```
//Atenção: Aqui ficará a Programação da verificação do ataque usado e verificação se existe mais de um personagem vivo
if (!string.IsNullOrEmpty(ataqueUsado)) A
{
    //Incrementa os dados dos combates
    atacante.Vitorias++;
    oponente.Derrotas++; B
    atacante.Disputas++;
    oponente.Disputas++;

    d.Id = 0;//Zera o Id para poder salvar os dados de disputa sem erro de chave.
    d.DataDisputa = DateTime.Now;
    _context.Disputas.Add(d); C
    await _context.SaveChangesAsync();
}

qtdPersonagensVivos = personagens.FindAll(p => p.PontosVida > 0).Count; D

E
if (qtdPersonagensVivos == 1)//Havendo só um personagem vivo, existe um CAMPEÃO!
{
    string resultadoFinal =
        $"{atacante.Nome.ToUpper()} é CAMPEÃO com {atacante.PontosVida} pontos de vida restantes!";
    F
    d.Narracao += resultadoFinal; //Concatena o resultado final com as demais narrações.
    d.Resultados.Add(resultadoFinal); //Contatena o resultado final com os demais resultados.

    break; //break vai parar o While.
}
} //Fim do While
//Código após o fechamento do While. Atualizará os pontos de vida,
//disputas, vitórias e derrotas de todos personagens ao final das batalhas
```

- (A) Verificação se o ataque usado teve resultado, já que ele não existirá caso o personagem não tenha arma nem habilidades.
(B) Incremento dos dados de disputa do atacante e do oponente.
(C) Preparação para salvar os dados de disputa no banco de dados.
(D) Contagem dos personagens que ainda tem pontuação de vida positiva.
(E) Caso apenas um personagem se enquadre nesta situação, quer dizer que ele será o campeão.
(F) Preparação das mensagens finais e comando break para interromper o While.



6. Para realizar o teste no postman, basta realizar o envio dos Ids dos personagens separados por vírgula e dentro de colchetes, assim a lista chamada ListIdPersonagens da classe Disputa será carregada quando os dados forem deserealizados.

The screenshot shows a Postman request configuration and its corresponding response. The request is a POST to `http://localhost:5000/Disputas/DisputaEmGrupo`. The Body is set to JSON and contains the following payload:

```
1 {  
2     "ListaIdPersonagens": [1,2,3]  
3 }
```

The response status is 200 OK, with a time of 135 ms and a size of 611 B. The response body is:

```
5     "ponenteId": 3,  
6     "narracao": "Frodo atacou Sam usando Arma A com o dano 31.Frodo atacou Galadriel usando Arma A com o dano 30.FRODO é o  
7         CAMPEÃO com 10 pontos de vida restantes!",  
8     "habilidadeId": 0,  
9     "listaIdPersonagens": [  
10         1,  
11         2,  
12         3  
13     ],  
14     "resultados": [  
15         "Frodo atacou Sam usando Arma A com o dano 31.",  
16         "Frodo atacou Galadriel usando Arma A com o dano 30.",  
17         "FRODO é o CAMPEÃO com 10 pontos de vida restantes!"  
18     ]
```

- Para permitir disputas mais duradoras, você pode variar os valores do dano das armas e habilidades e a força, inteligência e defesa dos personagens, além de relacionar mais personagens para a disputa, conforme o body enviado no postman. Tenha Armas e habilidade diversificadas nas tabelas e associadas corretamente a cada Personagem.



Métodos para complementar as operações na API – Disponíveis para cópia – A cada método inserido realize a compilação e verifique se não estará faltando nenhum using.

Copie e adicione na Classe controller **DisputasController** – Apagar Disputas

```
[HttpDelete("ApagarDisputas")]
    public async Task<IActionResult> DeleteAsync()
    {
        try
        {
            List<Disputa> disputas = await _context.Disputas.ToListAsync();

            _context.Disputas.RemoveRange(disputas);
            await _context.SaveChangesAsync();

            return Ok("Disputas apagadas");
        }
        catch (System.Exception ex)
        {
            return BadRequest(ex.Message);
        }
    }
```

Copie e adicione na Classe controller **DisputasController** – Listar Disputas

```
[HttpGet("Listar")]
    public async Task<IActionResult> ListarAsync()
    {
        try
        {
            List<Disputa> disputas =
                await _context.Disputas.ToListAsync();

            return Ok(disputas);
        }
        catch (System.Exception ex)
        {
            return BadRequest(ex.Message);
        }
    }
```



Copie e adicione na Classe controller **PersonagensController** – Restaurar Pontos de Vida

```
[HttpPut("RestaurarPontosVida")]
    public async Task<IActionResult> RestaurarPontosVidaAsync(Personagem p)
    {
        try
        {
            int linhasAfetadas = 0;
            Personagem pEncontrado =
                await _context.Personagens.FirstOrDefaultAsync(pBusca => pBusca.Id == p.Id);
            pEncontrado.PontosVida = 100;

            bool atualizou = await TryUpdateModelAsync<Personagem>(pEncontrado, "p",
                pAtualizar => pAtualizar.PontosVida);
            // EF vai detectar e atualizar apenas as colunas que foram alteradas.
            if (atualizou)
                linhasAfetadas = await _context.SaveChangesAsync();

            return Ok(linhasAfetadas);
        }
        catch (System.Exception ex)
        {
            return BadRequest(ex.Message);
        }
    }
}
```

```
//Método para alteração da foto
[HttpPut("AtualizarFoto")]
public async Task<IActionResult> AtualizarFoto(Personagem p)
{
    try
    {
        Personagem personagem = await _context.Personagens
            .FirstOrDefaultAsync(x => x.Id == p.Id);
        personagem.FotoPersonagem = p.FotoPersonagem;
        var attach = _context.Attach(personagem);
        attach.Property(x => x.Id).IsModified = false;
        attach.Property(x => x.FotoPersonagem).IsModified = true;
        int linhasAfetadas = await _context.SaveChangesAsync();
        return Ok(linhasAfetadas);
    }
    catch (System.Exception ex)
    { return BadRequest(ex.Message);}}
```



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

Copie e adicione na Classe controller **PersonagemController** – Zerar Ranking

```
[HttpPost("ZerarRanking")]
    public async Task<IActionResult> ZerarRankingAsync(Personagem p)
    {
        try
        {
            Personagem pEncontrado =
                await _context.Personagens.FirstOrDefaultAsync(pBusca => pBusca.Id == p.Id);

            pEncontrado.Disputas = 0;
            pEncontrado.Vitorias = 0;
            pEncontrado.Derrotas = 0;
            int linhasAfetadas = 0;

            bool atualizou = await TryUpdateModelAsync<Personagem>(pEncontrado, "p",
                pAtualizar => pAtualizar.Disputas,
                pAtualizar => pAtualizar.Vitorias,
                pAtualizar => pAtualizar.Derrotas);

            // EF vai detectar e atualizar apenas as colunas que foram alteradas.
            if (atualizou)
                linhasAfetadas = await _context.SaveChangesAsync();

            return Ok(linhasAfetadas);
        }
        catch (System.Exception ex)
        {
            return BadRequest(ex.Message);
        }
    }
}
```



Copie e adicione na Classe controller **PersonagemController** – Zerar ranking geral e restaurar vidas geral

```
[HttpPost("ZerarRankingRestaurarVidas")]
    public async Task<IActionResult> ZerarRankingRestaurarVidasAsync()
    {
        try
        {
            List<Personagem> lista =
                await _context.Personagens.ToListAsync();

            foreach (Personagem p in lista)
            {
                await ZerarRankingAsync(p);
                await RestaurarPontosVidaAsync(p);
            }
            return Ok();
        }
        catch (System.Exception ex)
        {
            return BadRequest(ex.Message);
        }
    }
}
```

Copie e adicione na Classe controller UsuariosController.cs os métodos para buscar o usuário por Id e por login, e para atualizar a geolocalização e atualizar o e-mail

```
[HttpGet("{usuarioId}")]
    public async Task<IActionResult> GetUsuario(int usuarioId)
    {
        try
        {
            //List exigirá o using System.Collections.Generic
            Usuario usuario = await _context.Usuarios //Busca o usuário no banco através
do Id
                .FirstOrDefaultAsync(x => x.Id == usuarioId);

            return Ok(usuario);
        }
        catch (System.Exception ex)
        {
            return BadRequest(ex.Message);
        }
    }
}
```



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

```
[HttpGet("GetByLogin/{login}")]
public async Task<IActionResult> GetUsuario(string login)
{
    try
    {
        //List exigirá o using System.Collections.Generic
        Usuario usuario = await _context.Usuarios //Busca o usuário no banco através do login
            .FirstOrDefaultAsync(x => x.Username.ToLower() == login.ToLower());

        return Ok(usuario);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

```
//Método para alteração da geolocalização
[HttpPut("AtualizarLocalizacao")]
public async Task<IActionResult> AtualizarLocalizacao(Usuario u)
{
    try
    {
        Usuario usuario = await _context.Usuarios //Busca o usuário no banco através do Id
            .FirstOrDefaultAsync(x => x.Id == u.Id);

        usuario.Latitude = u.Latitude;
        usuario.Longitude = u.Longitude;

        var attach = _context.Attach(usuario);
        attach.Property(x => x.Id).IsModified = false;
        attach.Property(x => x.Latitude).IsModified = true;
        attach.Property(x => x.Longitude).IsModified = true;

        int linhasAfetadas = await _context.SaveChangesAsync(); //Confirma a alteração no banco
        return Ok(linhasAfetadas); //Retorna as linhas afetadas (Geralmente sempre 1 linha msm)
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

```
//Método para alteração do e-mail
[HttpPut("AtualizarEmail")]
public async Task<IActionResult> AtualizarEmail(Usuario u)
{
    try
    {
        Usuario usuario = await _context.Usuarios //Busca o usuário no banco através do Id
            .FirstOrDefaultAsync(x => x.Id == u.Id);

        usuario.Email = u.Email;

        var attach = _context.Attach(usuario);
        attach.Property(x => x.Id).IsModified = false;
        attach.Property(x => x.Email).IsModified = true;

        int linhasAfetadas = await _context.SaveChangesAsync(); //Confirma a alteração no banco
        return Ok(linhasAfetadas); //Retorna as linhas afetadas (Geralmente sempre 1 linha msm)
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

```
//Método para alteração da foto
[HttpPut("AtualizarFoto")]
public async Task<IActionResult> AtualizarFoto(Usuario u)
{
    try
    {
        Usuario usuario = await _context.Usuarios
            .FirstOrDefaultAsync(x => x.Id == u.Id);

        usuario.Foto = u.Foto;

        var attach = _context.Attach(usuario);
        attach.Property(x => x.Id).IsModified = false;
        attach.Property(x => x.Foto).IsModified = true;

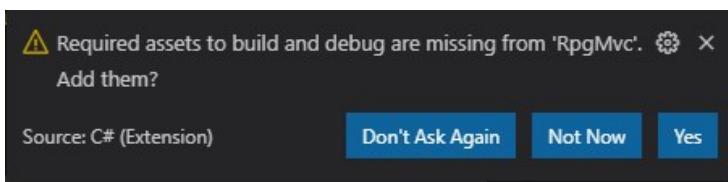
        int linhasAfetadas = await _context.SaveChangesAsync();
        return Ok(linhasAfetadas);
    }
    catch (System.Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```



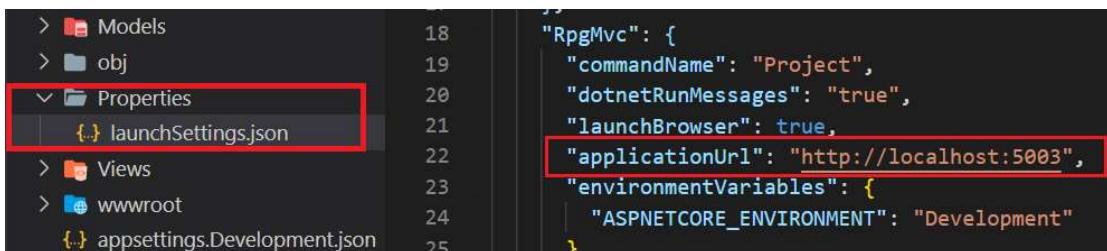
Aula 15 - Registro e login de Usuário no projeto MVC

Na última aula fizemos a publicação da API no servidor Somee e agora será o momento de criar uma aplicação com Front-end para consumir esta API e exibir os dados na página. Para isso iremos utilizar uma classe ViewModel (espécie de Model para Front-end) para que os dados da API preencham ela, uma controller para intermediar a requisição com a API e enviar os dados para a interface e páginas (Views) com extensão ".cshtml", que mescla códigos html e razor. Cada conjunto de operações na controller terá uma View correspondente.

1. Navegue no Windows até a pasta onde costuma salvar seus projetos, crie uma pasta chamada **RpgMvc** e abra a mesma no VS Code.
2. Digite o comando `dotnet new MVC --framework net5.0`
3. Clique em qualquer arquivo .cs (Startup.cs) e caso a mensagem abaixo para ativar a depuração apareça, clique em Yes.



4. Abra o arquivo `launchSettings.json`, dentro da pasta Properties e remova o endereço de execução `https`, mantendo apenas o endereço `http` conforme abaixo.



5. Abra a classe Startup para fazer a configuração para criação de Sessão no Método ConfigureServices

```
services.AddSession(  
    option => {option.IdleTimeout = System.TimeSpan.FromSeconds(3600);}  
)
```

- Utilizaremos Session para poder armazenar o token do usuário. Session é uma variável que fica armazenada no servidor em que aplicação se encontrar e que dura um tempo determinado, expirando logo após o período.
- Note que a configuração aqui realizada é de 60 minutos (3600 segundos)



6. Adicione o uso de Sessão no método Configure da mesma classe. O trecho em destaque tem que ficar exatamente antes da configuração do UseEndpoints

```
app.UseSession();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    endpoints.MapRazorPages();
});
```

7. Crie a classe **UsuarioViewModel** na pasta Models.

```
public class UsuarioViewModel
{
    0 references
    public int Id { get; set; }
    0 references
    public string Username { get; set; }
    0 references
    public string PasswordString { get; set; }
    0 references
    public byte[] Foto { get; set; }
}
```

8. Crie uma controller chamada **UsuariosController** dentro da pasta Controllers. As *controllers* em projetos MVC herdam de *Controller*, sendo necessário o using sinalizado acima. Aclone através do atalho Control + . (ponto)

```
using Microsoft.AspNetCore.Mvc;

namespace RpgMvc.Controllers
{
    0 references
    public class UsuariosController : Controller
    {
        0 references
        public string uriBase = "xyz/Usuarios/";
        //xyz tem que ser substituído pelo endereço da sua API.
    }
}
```

9. Adicionar o pacote de conversão de Json, utilize no terminal o comando:

dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson -v 5.0.15

10. Ainda na *controller*, insira o método *HttpGet* que irá carregar a *view* inicialmente

```
[HttpGet]
0 references
public ActionResult Index()
{
    return View("CadastrarUsuario");
}
```



11. Crie o corpo do método `HttpPost`. Usaremos o `try/catch` ao criar um método para que em caso de erro redirecionaremos para a própria `index` para exibir uma mensagem. exigira os usings: `RpgApi.Models`; `System.Threading.Tasks`.

```
[HttpPost]
0 references
public async Task<ActionResult> RegistrarAsync(UsuarioViewModel u)
{
    try
    {
        //Próximo código aqui
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```

12. Agora criaremos o método `HttpPost` que postará os dados para `API` para registrar o usuário. Usings `HttpClient` → `System.Net.Http`, `JsonConvert` → `Newtonsoft.Json`

```
try
{
    A HttpClient httpClient = new HttpClient();
    string uriComplementar = "Registrar";

    var content = new StringContent(JsonConvert.SerializeObject(u));
    B content.Headers.ContentType = new System.Net.Http.Headers.MediaTypeHeaderValue("application/json");
    HttpResponseMessage response = await httpClient.PostAsync(uriBase + uriComplementar, content);

    C string serialized = await response.Content.ReadAsStringAsync();

    D if (response.StatusCode == System.Net.HttpStatusCode.OK)
    {
        TempData["Mensagem"] =
            string.Format("Usuário {0} Registrado com sucesso! Faça o login para acessar.", u.Username);
        return View("AutenticarUsuario");
    }
    E else
    {
        throw new System.Exception(serialized);
    }
}
```

- (A) Instância do objeto `HttpClient` e criação de string para trecho final da rota do método da API
(B) Serialização do objeto `u`. Definição do conteúdo http como Json e envio dos dados para API, guardando o retorno na variável `response` que armazena todos os dados do retorno da requisição.
(C) Buscando e armazenando dados de de retorno dentro do retorno da requisição
(D) Consultando qual foi o status da requisição, se foi Ok irá guardar uma mensagem temporária e redirecionar para outra a view de login.
(E) Se não for Ok lançará uma exceção com o que retornou de “`serialized`” para o `catch`



13. Crie uma pasta chamada **Usuarios** dentro da pasta **Views** e dentro da pasta **Usuarios** crie o arquivo **CadastrarUsuario.cshtml**, inserindo as tags de design abaixo

```
<!--Namespace da classe de Modelo-->
@model RpgMvc.Models.UsuarioViewModel
<!--Título da View-->
#{@
    ViewBag.Title = "Registrar";
}
<!--Configuração de mensagem temporária-->
@if (@ TempData["MensagemErro"] != null)
{
    <div class="alert alert-danger" role="alert">
        @ TempData["MensagemErro"]
    </div>
}
<h2>Criação de novos Usuários</h2>
<hr />
<!-- Método que será chamado se o usuário fizer o Post -->
@using (Html.BeginForm("Registrar", "Usuarios", FormMethod.Post))
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <div class="form-group">
            <label class="control-label col-md-2">Usuário</label>
            <div class="col-md-6">
                <!--Campo TextBox -->
                @Html.EditorFor(model => model.Username, new { htmlAttributes = new { @class = "form-control" } })
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2">Senha</label>
            <div class="col-md-6">
                <!--TextBox de Senha-->
                @Html.PasswordFor(model => model.PasswordString, new { @class = "form-control" })
            </div>
        </div>
        <input type="submit" value="Registrar" class="btn btn-primary" />
        <!--Exemplo de Link (Texto, View, Controller) -->
        @Html.ActionLink("Retornar", "Index", "Home", null, new { @class = "btn btn-warning" })
    </div>
}
```



14. Crie um arquivo chamado **_LoginPartialRpg.cshtml** dentro da pasta Views/Shared e inclua a tags abaixo

```
<ul class="navbar-nav">
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Usuarios" asp-
action="Index" >Registrar-se</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Usuarios" asp-
action="IndexLogin">Login</a>
    </li>
</ul>
```

- A pasta Shared contém *views* que podem ser utilizadas em todo projeto, e que por padrão a nomenclatura do arquivo se inicia com _ (underline). Perceba que os links estão redirecionando para a *controller* recém-criada.

15. No arquivo _Layout.cshtml, insira a tag partial, com a propriedade name depois do fechamento da ul.

```
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </li>
    </ul>
    <partial name="_LoginPartialRpg" />
</div>
```

- Execute o projeto e perceba as diferenças. O efeito desta mudança é que na tela inicial, ao clicar em registrar-se, será exibida a área de novos usuários criada na etapa 4. Tente fazer as operações para registrar um login já existente no banco de dados e registrar um login inédito, conferindo se os dados de usuário cadastrado estão indo para o banco de dados.
- Você deve ter percebido que quando o usuário foi registrado, a página ficou em branco, isso se deve ao fato de não ter a *view* de login ainda, que é para onde o usuário será direcionado após o cadastro. Está será a próxima etapa.

16. Volte a Controller UsuariosController e adicione o método para abrir a *view* de login.

```
[HttpGet]
0 references
public ActionResult IndexLogin()
{
    return View("AutenticarUsuario");
}
```



17. Crie o corpo do método de autenticação

```
[HttpPost]  
0 references  
public async Task<ActionResult> AutenticarAsync(UsuarioViewModel u)  
{  
    try  
    {  
        //Próximo código aqui  
    }  
    catch (System.Exception ex)  
    {  
        TempData["MensagemErro"] = ex.Message;  
        return IndexLogin();  
    }  
}
```

18. abaixo de para enviar os dados de login via post para a API. Em **SetString (A)**, será necessário o using Microsoft.AspNetCore.Http. Em **(B)** Utilize o using System.Net.Http.Headers

```
try  
{  
    HttpClient httpClient = new HttpClient();  
    string uriComplementar = "Autenticar";  
  
    var content = new StringContent(JsonConvert.SerializeObject(u));  
    content.Headers.ContentType = new MediaTypeHeaderValue("application/json"); B  
    HttpResponseMessage response = await httpClient.PostAsync(uriBase + uriComplementar, content);  
  
    string serialized = await response.Content.ReadAsStringAsync();  
  
    if (response.StatusCode == System.Net.HttpStatusCode.OK)  
    {  
        A HttpContext.Session.SetString("SessionTokenUsuario", serialized);  
        TempData["Mensagem"] = string.Format("Bem-vindo {0}!!!", u.Username);  
        return RedirectToAction("Index", "Personagens");  
    }  
    else  
    {  
        throw new System.Exception(serialized);  
    }  
}
```

- Perceba em (A) que estamos guardando o retorno da requisição numa sessão. Como testado anteriormente no *postman*, o retorno será da chamada é um Token.



19. Crie o arquivo **AutenticarUsuario.cshtml** dentro da pasta Views/Usuarios, inserindo as tags a seguir. Após isso execute o projeto para testar a página de acesso.

```
@model RpgMvc.Models.UsuarioViewModel
@{
    ViewBag.Title = "Autenticar"; }

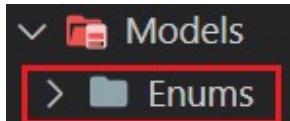
@if (@ TempData["Mensagem"] != null)
{
    <div class="alert alert-success" role="alert">
        @ TempData["Mensagem"]</div>
}
<!--Configuração para exibir mensagem de erro -->
@if (@ TempData["MensagemErro"] != null)
{
    <div class="alert alert-danger" role="alert">
        @ TempData["MensagemErro"]</div>
}
<h2>Área de login do Usuário</h2><hr />
@using (Html.BeginForm("Autenticar", "Usuarios", FormMethod.Post))
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <div class="form-group">
            <label class="control-label col-md-2">Usuário</label>
            <div class="col-md-6">
                @Html.EditorFor(model => model.Username, new { htmlAttributes = new { @class = "form-control" } })
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2">Senha</label>
            <div class="col-md-6">
                @Html.PasswordFor(model => model.PasswordString, new { @class = "form-control" })
            </div>
        </div>
        <input type="submit" value="Acessar" class="btn btn-success" /><br />
        <div class="form-group">
            <!-- Outra forma de incluir um link para uma View-->
            <p> <a href="/Usuarios">Never accessed the system? Click here to register!!!</a> </p>
        </div>
    </div>
}
```

- Desafio da semana: Procurar layout bootstrap gratuitos na internet para testar na view _Layout.cshtml



Aula 16 – Projeto MVC consumindo API – Parte 1: GetAll, Post e Get

1. Crie uma pasta chamada Enums dentro da pasta Models



2. Dentro da pasta models crie uma classe chamada ClasseEnum. Após isso altere a palavra `class` por `public enum`. Dentro do corpo do enum programaremos os itens conforme abaixo

```
public enum ClasseEnum
{
    Cavaleiro = 1,
    Mago = 2,
    Clerigo = 3
}
```

3. Crie a classe **PersonagemViewModel.cs** dentro da pasta Models com as mesmas propriedades da classe Personagens do projeto de API. Comente as propriedades do tipo **Usuario**, **Arma** e **List<PersonagemHabilidade>**, além de adicionar o using para o enumeration. Remova as notações *NotMapped* e *JsonIgnore* que vierem na cópia. Resumindo as propriedade para este momento: Id, Nome, Forca, PontosVida, Defesa, Inteligencia, Classe, FotoPersonagem, Disputas, Vitorias e Derrotas.

- Para o enum será necessário o using de `RpgMvc.Models.Enums`

4. Crie a classe **PersonagensController.cs** dentro da pasta Controllers. Faça a herança de `controller` conforme sinalizado

```
public class PersonagensController : Controller
```

- Requer o using `Microsoft.AspNetCore.Mvc`

5. Crie uma variável global para o endereço base da API

```
public class PersonagensController : Controller
{
    public string uriBase = "xyz/Personagens/";
    //xyz tem que ser substituído pelo nome do seu site da API.
```



6. Crie o método **IndexAsync** conforme abaixo, será comentado após a imagem os *usings* que serão necessários: Task → System.Threading.Tasks, AuthenticationHeaderValue → System.Net.HttpHeaders httpClient → System.Net.Http, List → System.Collections.Generic, PersonagemViewModel → RpgMvc.Models, JsonConvert → Newtonsoft.Json, Session → Microsoft.AspNetCore.Http

```
[HttpGet]
0 references
public async Task<ActionResult> IndexAsync()
{
    try
    {
        1 string uriComplementar = "GetAll";
        2 HttpClient httpClient = new HttpClient();
        3 string token = HttpContext.Session.GetString("SessionTokenUsuario");
        4 httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);

        5 HttpResponseMessage response = await httpClient.GetAsync(uriBase + uriComplementar);
        6 string serialized = await response.Content.ReadAsStringAsync();

        7 if (response.StatusCode == System.Net HttpStatusCode.OK)
        {
            List<PersonagemViewModel> listaPersonagens = await Task.Run(() =>
                JsonConvert.DeserializeObject<List<PersonagemViewModel>>(serialized));

            return View(listaPersonagens);
        }
        8 else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```

- (1) Variável *string* para conter o nome método para concatenar com o endereço base da *api*.
- (2) Variável do tipo *http* que fará toda transição da requisição dos dados na web
- (3) Token recuperado da Sessão para variável *string*
- (4) Carregamento do cabeçalho (header) da requisição com o token
- (5) Variável que vai guardar a resposta da requisição e que guardar várias informações
- (6) Etapa em que o conteúdo da requisição (*json*) vai ser guardado em uma *string* para próxima etapa.
- (7) Se retornar Ok (200) será feita uma desserialização do que era *string* para que vire um objeto. Neste caso está sendo transformado em uma lista de Personagens. A operação é bem-sucedida desde que os campos contidos no *json* tenham identificação parecida com os campos da classe Personagem.
- (8) Se der erro cairá no else, e o que estiver na variável “*serialized*” será uma mensagem de erro que será lançada como exceção.



-
7. Crie uma pasta chamada **Personagens** dentro da pasta **Views** e dentro desta pasta crie um arquivo chamado **Index.cshtml**. Tudo o que tem um @ na frente representa programação razor, ficando explícito que estamos querendo usar algo que está nas classes C# (.cs). O restante é o bom e velho *html* para formar o layout da página.

```
<!--Namespace da classe de Modelo para esta view-->
@model IEnumerable<RpgMvc.Models.PersonagemViewModel>

<!--Inclua os TempData para Sucesso e Erro aqui, conforme exemplo na view de Autenticação-->

{@ViewBag.Title = "Personagens"; }<!--Título da página para o navegador-->
<h2>Relação de Personagens</h2><!--Título da página-->
<p>
    <!--Links apontando para views na mesma pasta-->
    @Html.ActionLink("Criar Novo Personagem", "Create")
</p>
<table class="table">
    <tr><!--Títulos das colunas da tabela-->
        <th>@Html.DisplayNameFor(model => model.Id)</th>
        <th>@Html.DisplayNameFor(model => model.Nome)</th>
        <th>@Html.DisplayNameFor(model => model.Classe)</th>
        <th>@Html.DisplayNameFor(model => model.Disputas)</th>
        <th>@Html.DisplayNameFor(model => model.Vitorias)</th>
        <th>@Html.DisplayNameFor(model => model.Derrotas)</th>
        <th></th>
    </tr>
    <!--Looping para escrever os dados na tabela-->
    @foreach (var item in Model)
    {
        <tr>
            <td>@Html.DisplayFor(modelItem => item.Id)</td>
            <td>@Html.DisplayFor(modelItem => item.Nome)</td>
            <td>@Html.DisplayFor(modelItem => item.Classe)</td>
            <td>@Html.DisplayFor(modelItem => item.Disputas )</td>
            <td>@Html.DisplayFor(modelItem => item.Vitorias)</td>
            <td>@Html.DisplayFor(modelItem => item.Derrotas)</td>
            <td><!--Coluna para Links/botões-->
                @Html.ActionLink("Editar", "Edit", new { id = item.Id } ) |
                @Html.ActionLink("Detalhes", "Details", new { id = item.Id } ) |
                @Html.ActionLink("Deletar", "Delete", new { id = item.Id })
            </td>
        </tr>
    }
</table>
```



8. Insira na `view _Layout.cshtml`, que se encontra na pasta `Views/Shared`, um item de menu para a controller e view que criamos anteriormente. Depois execute o programa e navegue para a página de Personagens para certificar que os dados vão ser carregados.

```
<ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Personagens" asp-action="Index">Personagens</a>
    </li>
```

9. Retorne à `controller` e crie um método `HttpPost` com o nome “**CreateAsync**”. Esse método postará para a `Api` enviando um objeto serializado. `MediaTypeHeaderValue` requer o using `System.Net.Http.Headers`

```
[HttpPost]
0 references
public async Task<ActionResult> CreateAsync(PersonagemViewModel p)
{
    try
    {
        1 HttpClient httpClient = new HttpClient();
        2 string token = HttpContext.Session.GetString("SessionTokenUsuario");
        3 httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);

        4 var content = new StringContent(JsonConvert.SerializeObject(p));
        5 content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
        6 HttpResponseMessage response = await httpClient.PostAsync(uriBase, content);
        7 string serialized = await response.Content.ReadAsStringAsync();

        8 if (response.StatusCode == System.Net HttpStatusCode.OK)
        {
            TempData["Mensagem"] = string.Format("Personagem {0}, Id {1} salvo com sucesso!", p.Nome, serialized);
            return RedirectToAction("Index");
        }
        8 else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Create");
    }
}
```

- Resumo das ações:

- (1) Declaração de Objeto `HttpClient` responsável pelo tráfego de dados na internet.
- (2) Declaração de variável para armazenar o token que está na string e passagem do token recuperado para a propriedade `Authorization` do objeto `httpClient`
- (3) O objeto `p` está sendo *serializado*, ou seja, está sendo transformado numa cadeia de caracteres em série dentro da variável `content`, o famoso formato `json`.



- (4) Está sendo informado no cabeçalho do conteúdo que ele é do tipo *json*.
(5) É declarada uma variável para armazenar o resultado da requisição que está sendo postada com o conteúdo serializado para o endereço base, já que o post o utiliza.
(6) O conteúdo da resposta da requisição é armazenado numa variável no formato de *string*.
(7) Sendo Ok (200) a mensagem será armazenada num TempData para que a View Index possa apresentar o conteúdo.
(8) Caso não seja Ok, lançará uma exceção que será guardada no TempData para a View Index exibir.
10. Retorne à controller e crie um método *HttpGet* com o nome “**Create**”. Esse método será acionado quando o usuário clicar em “Novo Personagem” e não exige nenhum parâmetro, apenas carregará a View.

```
[HttpGet]  
0 references  
public ActionResult Create()  
{  
    return View();  
}
```

11. Crie uma View com o nome **Create.cshtml** na pasta **Personagens** e realize a programação a seguir. Execute e perceba que a *div* sinalizada simboliza a *Label* e uma caixa de texto que são visualizadas na tela. Use o exemplo para criar os outros campos: Pontos de vida, Força, Defesa, Inteligência e classe.

```
@model RpgMvc.Models.PersonagemViewModel  
{@  
    ViewBag.Title = "Novo Personagem";  
}  
<!-- Insira aqui os TempData de Sucesso e Erro para evitar tela da morte -->  
<h2>Criar um Novo Personagem</h2>  
@using (Html.BeginForm())  
{  
    @Html.AntiForgeryToken()  
    <div class="form-horizontal">  
        <hr />  
        <div class="form-group">  
            @Html.LabelFor(model => model.Nome, htmlAttributes: new { @class = "control-label col-md-2" })  
            <div class="col-md-6">  
                @Html.EditorFor(model => model.Nome, new { htmlAttributes = new { @class = "form-control" } })  
            </div>  
        </div>  
        <div class="form-group">  
            <div class="col-md-offset-2 col-md-6">  
                <input type="submit" value="Salvar" class="btn btn-primary" />  
            </div>  
        </div>  
    </div>  
    <div>  
        @Html.ActionLink("Retornar", "Index")  
    </div>
```



12. Volte à controller e crie um método *httpGet* chamado **DetailsAsync**

```
[HttpGet]
0 references
public async Task<ActionResult> DetailsAsync(int? id)
{
    try
    {
        HttpClient httpClient = new HttpClient();
        string token = HttpContext.Session.GetString("SessionTokenUsuario");
        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);
        HttpResponseMessage response = await httpClient.GetAsync(uriBase + id.ToString());
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            PersonagemViewModel p = await Task.Run(() =>
                JsonConvert.DeserializeObject<PersonagemViewModel>(serialized));
            return View(p);
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```

13. Crie uma view chamada **Details.cshtml** dentro da pasta View/Personagens

```
@model RpgMvc.Models.PersonagemViewModel
 @{
    ViewBag.Title = "Detalhes do Personagem";
}
<h2>Detalhes do Personagem</h2>
<div>
    <hr />
    <dl class="dl-horizontal">
        <dt>@Html.DisplayNameFor(model => model.Id)</dt>
        <dd>@Html.DisplayFor(model => model.Id)</dd>

        <dt>@Html.DisplayNameFor(model => model.Nome)</dt>
        <dd>@Html.DisplayFor(model => model.Nome)</dd>
    </dl>
</div>
<p>
    @Html.ActionLink("Editar", "Edit", new { id = Model.Id}) |
    @Html.ActionLink("Retornar", "Index")
</p>
```

- Execute a aplicação e perceba que o trecho sinalizado exibe o título do campo e o conteúdo do campo, no caso “Nome” e o conteúdo do campo Nome, respectivamente. Utilize o modelo para apresentar os demais dados do personagem.



Aula 16 - Projeto MVC consumindo API – Parte 2: Put e Delete

Concluiremos as *views* para que o projeto MVC possa consumir todos os métodos CRUD da API.

1. Crie o método Get com o nome **EditAsync**. Ele carregará a view de edição após pegar da API os dados do personagem.

```
[HttpGet]
0 references
public async Task<ActionResult> EditAsync(int? id)
{
    try
    {
        HttpClient httpClient = new HttpClient();
        string token = HttpContext.Session.GetString("SessionTokenUsuario");

        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);
        HttpResponseMessage response = await httpClient.GetAsync(uriBase + id.ToString());

        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            PersonagemViewModel p = await Task.Run(() =>
                JsonConvert.DeserializeObject<PersonagemViewModel>(serialized));
            return View(p);
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```



-
2. Crie o método Post com o mesmo nome **EditAsync**. Esse método enviará para a api os dados para atualização na base de dados.

```
[HttpPost]
0 references
public async Task<ActionResult> EditAsync(PersonagemViewModel p)
{
    try
    {
        HttpClient httpClient = new HttpClient();
        string token = HttpContext.Session.GetString("SessionTokenUsuario");

        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);
        var content = new StringContent(JsonConvert.SerializeObject(p));
        content.Headers.ContentType = new MediaTypeHeaderValue("application/json");

        HttpResponseMessage response = await httpClient.PutAsync(uriBase, content);
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            TempData["Mensagem"] =
                string.Format("Personagem {0}, classe {1} atualizado com sucesso!", p.Nome, p.Classe);

            return RedirectToAction("Index");
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

-
3. Crie a view chamada **Edit.cshtml** na pasta Views/Personagens com o design abaixo

```
@model RpgMvc.Models.PersonagemViewModel
 @{
    ViewBag.Title = "Editar Personagem";
}
<h2>Editar dados do Personagem</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <hr />
        <div class="form-group">
            @Html.LabelFor(model => model.Id, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-6">
                @Html.EditorFor(model => model.Id, new { htmlAttributes = new { @class = "form-control", @readonly = "readonly" } })
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.Nome, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-6">
                @Html.EditorFor(model => model.Nome, new { htmlAttributes = new { @class = "form-control" } })
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-6">
                <input type="submit" value="Salvar alterações" class="btn btn-primary" />
            </div>
        </div>
    </div>
    <div>
        @Html.ActionLink("Retornar", "Index")
    </div>
}
```

- No exemplo temos apenas os campos de Id e Nome. Utilize o exemplo sinalizado para criar os demais campos presentes na classe Personagem.
- Execute o projeto e teste a edição de um personagem já salvo na base de dados.



-
4. Crie o método HttpGet DeleteAsync para remoção de um personagem

```
[HttpGet]
0 references
public async Task<ActionResult> DeleteAsync(int id)
{
    try
    {
        HttpClient httpClient = new HttpClient();
        string token = HttpContext.Session.GetString("SessionTokenUsuario");
        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);

        HttpResponseMessage response = await httpClient.DeleteAsync(uriBase + id.ToString());
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            TempData["Mensagem"] = string.Format("Personagem Id {0} removido com sucesso!", id);
            return RedirectToAction("Index");
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```

5. Altere o link para remoção presente na view index.cshtml para exibir uma mensagem de confirmação da remoção.

```
<td>
    @Html.ActionLink("Editar", "Edit", new { id = item.Id } ) |
    @Html.ActionLink("Detalhes", "Details", new { id = item.Id }) |
    @Html.ActionLink("Deletar", "Delete", new { id = item.Id })
        , new { onclick = "return confirm('Deseja realmente deletar este personagem ?');"})
</td>
```



-
6. Altere a View Create.cshtml e Edit.cshtml para que ao invés de apresentar um campo `EditorFor`, seja apresentado um campo `DropDownListFor`

```
<div class="form-group">
    @Html.LabelFor(model => model.Classe, htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-6">
        @Html.DropDownListFor(model => model.Classe , Html.GetEnumSelectList(typeof(RpgMvc.Models.Enums.ClasseEnum)),
        "---Selecione---", new { @class = "form-control" })
    </div>
</div>
```

- Execute o projeto e confirme que o `DropDownList` está carregado nas Views Create e Edit através de um enumeration.

Prints para comprovar o funcionamento – Não compacte todos os prints anexe todos!

- (1) Print da tela de listagem dos personagens contendo a mensagem de usuário autenticado.
- (2) Print da tela com a cadastro de personagens preenchido e (2) da listagem dos personagens informando que foi cadastrado com sucesso.
- (3) Print da tela de Detalhes contendo o personagem recém cadastrado no item (2).
- (4) Print da tela de Edição contendo o personagem que foi cadastrado no item (2).
- (5) Print da mensagem que pergunta se deseja excluir o usuário (Escolher um usuário que não seja o cadastrado no print (2)
- (6) Print da tela listagem aparecendo a mensagem de sucesso após excluir um personagem que não seja o que foi cadastrado no print (2)

O que são TempData, ViewData e ViewBag?

TempData, ViewData e ViewBag são recursos que podemos utilizar para transitar dados de uma `controller` para uma `view` ou vice-versa, sendo que utilizaremos para trigar mensagens entre estas camadas. Segue abaixo algumas referências teóricas sobre o tema que aplicaremos no projeto.

- <https://www.eduardopires.net.br/2013/06/asp-net-mvc-viewdata-viewbag-tempdata/>
- http://www.macoratti.net/15/06/mvc_conc1.htm
- <https://pt.stackoverflow.com/questions/273504/quais-as-diferen%C3%A7as-entre-viewbag-viewdata-e-tempdata>



Aula 17 – Projeto MVC Consumindo API – Views para disputa entre os personagens

1. Na view Personagens/Index, crie o primeiro botão de disputas abaixo da tag de fechamento da table.

```
@Html.ActionLink("Clique aqui para um embate com armas!!!", "Index", "Disputas",
    null, new { @class = "btn btn-warning" })
```

- Execute o projeto e verifique se os dados serão apresentados abaixo da tabela.
- 2. Crie uma classe chamada **DisputaViewModel.cs** dentro da pasta Models, com as propriedades a seguir

```
public int Id { get; set; }
0 references
public DateTime? DataDisputa { get; set; }
0 references
public int AtacanteId { get; set; }
0 references
public int OponenteId { get; set; }
0 references
public string Narracao { get; set; }
0 references
public int HabilidadeId { get; set; }
0 references
public List<int> ListaIdPersonagens { get; set; } = new List<int>();
0 references
public List<string> Resultados { get; set; } = new List<string>();
```

- Será necessário o using System.Collections.Generic e System
- 3. Crie a classe chamada **DisputasController.cs** dentro da pasta Controllers, programando a herança à classe Controller e deixando o endereço da controller da sua API definido na variável uriBase

```
public class DisputasController : Controller
{
    0 references
    public string uriBase = "xyz/Disputas/";
    //xyz tem que ser substituído pelo nome do seu site na API.

    //Próximos métodos ficarão aqui
}
```

- Sintaxe Controller exigirá o using Microsoft.AspNetCore.Mvc



4. Ainda na controller criada, programe o método do tipo `HttpGet` que vai gerar carregar a tela de disputas. Adicione os usings necessários nos locais em que o código indicar erros de referência.

```
[HttpGet]
public async Task<ActionResult> IndexAsync()
{
    try
    {
        HttpClient httpClient = new HttpClient();
        A string token = HttpContext.Session.GetString("SessionTokenUsuario");
        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);

        string uriBuscaPersonagens = "http://xyz.somee.com/RpgApi/Personagens/GetAll";
        B HttpResponseMessage response = await httpClient.GetAsync(uriBuscaPersonagens);
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            List<PersonagemViewModel> listaPersonagens = await Task.Run(() =>
                JsonConvert.DeserializeObject<List<PersonagemViewModel>>(serialized));
            C ViewBag.ListaAtacantes = listaPersonagens;
            ViewBag.ListaOponentes = listaPersonagens;
            return View();
        }
        else
            D throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```

- Usings: `System.Collections.Generic`; `System.Net.Http`; `System.Net.Http.Headers`; `System.Threading.Tasks`; `Microsoft.AspNetCore.Http`; `Newtonsoft.Json` e `RpgMvc.Models`;
- (A) Criação da variável `http` e obtenção do token guardado na session
- (B) Definição da rota da API que buscará a lista de personagens na API, retornando uma lista se o método tiver êxito ou uma mensagem caso dê erro. Isso tudo guardando ainda serializado.
- (C) Se o status da requisição for 200 (Ok) então deserializamos para transformar numa lista de personagens, e depois geramos duas `ViewBags` a partir da lista de personagens, uma como os atacantes e outra como os oponentes. `ViewsBags` são maneiras de trafegar dados entre a controller e views e isso é que fará o carregamento do dropdownlist aparecer.
- (D) Mensagem de erro sendo lançada no `else` e capturada no `cacht` sendo guardada através do `TempData`. `TempData` é outra maneira de trafegar dados entre uma controller e uma View.



5. Crie uma pasta chamada **Disputas** dentro da pasta Views e dentro da pasta Disputas, crie o arquivo do chamado **Index.cshtml** com o layout abaixo

```
@model RpgMvc.Models.DisputaViewModel
{@ViewBag.Title = "Index";}

@if (@ TempData["Mensagem"] != null) {
    <div class="alert alert-success" role="alert">
        @ TempData["Mensagem"]</div>
}
@if (@ TempData["MensagemErro"] != null) {
    <div class="alert alert-danger" role="alert">
        @ TempData["MensagemErro"]</div>
}
<h2>Ataque com Arma</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <hr />
        <div class="form-group">
            @Html.DisplayName("Atacante")
            <div class="col-md-6">
                @Html.DropDownListFor(model => model.AtacantId, new SelectList(@ViewBag.ListaAtacantes, "Id", "Nome"),
                    "--- Selecione ---", new { @class = "form-control" })</div>
            </div>
            <div class="form-group">
                @Html.DisplayName("Oponente")
                <div class="col-md-6">
                    @Html.DropDownListFor(model => model.OponenteId, new SelectList(@ViewBag.ListaOponentes, "Id",
                    "Nome"),
                    "--- Selecione ---", new { @class = "form-control" })</div>
                </div>
                <div class="form-group">
                    <div class="col-md-offset-2 col-md-6">
                        <input type="submit" value="Atacar com Arma!!!" class="btn btn-primary" /></div>
                    </div>
                </div>
            }
    <div>@Html.ActionLink("Retornar", "Index", "Personagens")</div>
```

- Execute e verifique se a view será carregada ao clicar no botão de disputa com arma
- Observe como utilizar uma caixa de seleção usando DropDownListFor



6. Volte à controller de disputas e crie o método que realizará a postagem do ataque com armas para a API, acrescente os usings necessários nos locais em que for apresentado erros de referência.

```
[HttpPost]
0 references
public async Task<ActionResult> IndexAsync(DisputaViewModel disputa)
{
    try
    {
        HttpClient httpClient = new HttpClient();
        string uriComplementar = "Arma";

        A var content = new StringContent(JsonConvert.SerializeObject(disputa));
        content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
        HttpResponseMessage response = await httpClient.PostAsync(uriBase + uriComplementar, content);
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            B disputa = await Task.Run(() => JsonConvert.DeserializeObject<DisputaViewModel>(serialized));
            TempData["Mensagem"] = disputa.Narracao;
            return RedirectToAction("Index", "Personagens");
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```

(A) Configuração da rota para a API. Envio da requisição e armazenamento do retorno da requisição.

(B) Se o retorno for código 200 (ok) desserializa para o objeto disputa e armazena no TempData a narração do ataque e retorna para a View Index de Personagens, caso contrário uma lançará um erro.

- Execute o projeto e teste a realização do ataque

Ataque usando Habilidades

7. Crie uma classe chamada **HabilidadeViewModel** dentro da pasta Models com as propriedades Id, Nome e Dano, de forma similar ao que fizemos na API.
8. Crie um outro botão na view Index de personagens para redirecionar para o ataque usando habilidades

```
@Html.ActionLink("Clique aqui para um embate com habilidades!!!", "IndexHabilidades", "Disputas",
    null, new { @class = "btn btn-dark" })
```



9. Crie um método get na controller de disputas que será responsável por carregar a View do ataque com habilidades

```
[HttpGet]  
0 references  
public async Task<ActionResult> IndexHabilidadesAsync()  
{  
    try  
    {  
        //Programação aqui  
    }  
    catch (System.Exception ex)  
    {  
        TempData["MensagemErro"] = ex.Message;  
        return RedirectToAction("Index");  
    }  
}
```

10. Copie a programação de dentro do bloc try do método HttpGet Index (o primeiro feito na aula) e cole do corpo do método criado na etapa anterior, seguindo as orientações abaixo deste print

```
    ViewBag.ListaAtacantes = listaPersonagens;  
    ViewBag.ListaOponentes = listaPersonagens;  
    A - return View(); //Remova esta linha  
}  
else  
    throw new System.Exception(serialized);  
  
string uriBuscaHabilidades = "http://xyz.somee.com/RpgApi/PersonagemHabilidades/GetHabilidades";  
response = await httpClient.GetAsync(uriBuscaHabilidades);  
serialized = await response.Content.ReadAsStringAsync();  
  
if (response.StatusCode == System.Net.HttpStatusCode.OK)  
B {  
    List<HabilidadeViewModel> listaHabilidades = await Task.Run(() =>  
        JsonConvert.DeserializeObject<List<HabilidadeViewModel>>(serialized));  
    ViewBag.ListaHabilidades = listaHabilidades;  
}  
else  
    throw new System.Exception(serialized);  
  
C return View("IndexHabilidades");  
}  
catch (System.Exception ex)  
{  
    TempData["MensagemErro"] = ex.Message;  
    return RedirectToAction("Index");  
}
```

(A) Remova a linha sinalizada.

(B) Codificação que buscará a lista de habilidades e guardará na ViewBag

(C) Com a ViewBag carregada o usuário será direcionado para a View que criaremos a seguir



11. Dentro da pasta Views/Disputas crie o arquivo IndexHabilidades.cshtml. Copie todo o conteúdo do arquivo Views/Disputas/Index.cshtml e cole na view recém-criada.

12. Na view recém-criada (IndexHabilidades.cshtml) altere o título na tag h2 para “Ataque com Habilidade”. Altere também o texto do input.

```
<h2>Ataque com Habilidade</h2>
@using (Html.BeginForm())
{
```

13. Insira a codificação da seleção de ataques abaixo do DropDownList que contém os dados do atacante

```
<div class="form-group">
    @Html.DisplayName("Habilidade")
    <div class="col-md-6">
        @Html.DropDownListFor(model => model.HabilidadeId, new SelectList(@ViewBag.ListaHabilidades,
        "Id", "Nome"), "---Selecione---", new { @class = "form-control" })</div></div>
```

14. Retorne para a controller de disputas e faça a criação do método HttpPost que enviará a disputa para a API.

```
[HttpPost]
0 references
public async Task<ActionResult> IndexHabilidadesAsync(DisputaViewModel disputa)
{
    try
    {
        HttpClient httpClient = new HttpClient();
        string uriComplementar = "Habilidade";
        var content = new StringContent(JsonConvert.SerializeObject(disputa));
        content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
        HttpResponseMessage response = await httpClient.PostAsync(uriBase + uriComplementar, content);
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            disputa = await Task.Run(() =>
                JsonConvert.DeserializeObject<DisputaViewModel>(serialized));
            TempData["Mensagem"] = disputa.Narracao;
            return RedirectToAction("Index", "Personagens");
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```

- Teste selecionando o atacante, a habilidade ele tem e quem será o seu oponente.



-
15. Crie mais um botão na index de listagem de personagens

```
@Html.ActionLink("Clique aqui para um embate em grupo!!!", "DisputaGeral", "Disputas",
    null, new { @class = "btn btn-danger" })
```

16. Na controller de personagens crie o corpo do método que realizará uma disputa geral entre os personagens

```
[HttpGet]
0 références
public async Task<ActionResult> DisputaGeralAsync()
{
    try
    {
        //Próxima codificação aqui

        return RedirectToAction("Index", "Personagens");
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index", "Personagens");
    }
}
```

17. Programe no comentário do bloco try a primeira parte do método, onde carregaremos todos os personagens para uma lista

```
HttpClient httpClient = new HttpClient();

string token = HttpContext.Session.GetString("SessionTokenUsuario");
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);

string uriBuscaPersonagens = "http://xyz.somee.com/RpgApi/Personagens/GetAll";
HttpResponseMessage response = await httpClient.GetAsync(uriBuscaPersonagens);

string serialized = await response.Content.ReadAsStringAsync();

List<PersonagemViewModel> listaPersonagens = await Task.Run(() =>
    JsonConvert.DeserializeObject<List<PersonagemViewModel>>(serialized));
```



-
18. Na linha seguinte ao código anterior faremos a configuração para gerar as disputas

```
list<PersonagemViewModel> listaPersonagens = await Task.Run(() =>
    JsonConvert.DeserializeObject<List<PersonagemViewModel>>(serialized));

string uriDisputa = "http://xyz.somee.com/RpgApi/Disputas/DisputaEmGrupo";
DisputaViewModel disputa = new DisputaViewModel();
disputa.ListaIdPersonagens = new List<int>();
disputa.ListaIdPersonagens.AddRange(listaPersonagens.Select(p => p.Id)); //using System.Linq

var content = new StringContent(JsonConvert.SerializeObject(disputa));
content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
response = await httpClient.PostAsync(uriDisputa, content);

serialized = await response.Content.ReadAsStringAsync();

if (response.StatusCode == System.Net.HttpStatusCode.OK)
{
    disputa = await Task.Run(() => JsonConvert.DeserializeObject<DisputaViewModel>(serialized));
    TempData["Mensagem"] = string.Join("<br/>", disputa.Resultados);
}
else
    throw new System.Exception(serialized);

return RedirectToAction("Index", "Personagens");
}
catch (System.Exception ex)
{
    TempData["Mensagem"] = ex.Message;
}
```

- Necessário Using para System.Linq
- Execute o projeto e confirme que os embates serão realizados. O espero é que seja retornado todos os resultados, atualizando o número de disputas, vitórias e derrotas da lista de personagens.



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

Desafio: Criação de Models, Controllers e Views para PersonagemHabilidades, Habilidades e Disputas

1. Crie uma classe chamada **PersonagemHabilidadesViewModel.cs** conforme abaixo

```
namespace RpgMvc.Models
{
    public class PersonagemHabilidadeViewModel
    {
        public int PersonagemId { get; set; }
        public PersonagemViewModel Personagem { get; set; }
        public int HabilidadeId { get; set; }
        public HabilidadeViewModel Habilidade { get; set; }
    }
}
```

2. Acrescente uma lista de PersonagemHabilidade como propriedade na classe **HabilidadeViewModel** e na classe **PersonagemViewModel**

```
public List<PersonagemHabilidadeViewModel> PersonagemHabilidades {get; set;}
```

- Exigirá o using de System.Collections.Generics

3. Crie a controller PersonagemHabilidadesController dentro da pasta Controllers

```
namespace RpgMvc.Controllers
{
    public class PersonagemHabilidadesController : Controller
    {
        public string uriBase = "http://xyz.somee.com/RpgApi/PersonagemHabilidades/";
        //xyz será substituído pelo nome do seu site na API.
    }
}
```



4. Crie o método para consumir a API que retornará a listagem de habilidades

```
[HttpGet("PersonagemHabilidades/{id}")]
0 references
public async Task<ActionResult> DeleteAsync(int id)
{
    try
    {
        HttpClient httpClient = new HttpClient();
        string token = HttpContext.Session.GetString("SessionTokenUsuario");
        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);

        HttpResponseMessage response = await httpClient.GetAsync(uriBase + id.ToString());
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            List<PersonagemHabilidadeViewModel> lista = await Task.Run(() =>
                JsonConvert.DeserializeObject<List<PersonagemHabilidadeViewModel>>(serialized));

            return View(lista);
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```

- Faça os seguintes usings no topo da controller: System.Threading.Tasks, System.Net.Http, Microsoft.AspNetCore.Http, Microsoft.AspNetCore.Mvc, System.Net.Http.Headers, System.Collections.Generic, RpgMvc.Models, e Newtonsoft.Json;
- Perceba que acabamos de usar o método Get que é chamado com uma rota personalizada. Até então nos outros métodos, as rotas tinha sido padrão.



5. Crie a pasta **PersonagemHabilidades** dentro da pasta Views e dentro da pasta *PersonagemHabilidades*, crie a view **Index.cshtml** que terá o html/razor abaixo

```
@model IEnumerable<RpgMvc.Models.PersonagemHabilidadeViewModel>
{@ ViewBag.Title = "Index"; }
@if (@ TempData["Mensagem"] != null) {
    <div class="alert alert-success" role="alert">
        @ TempData["Mensagem"]</div>
}
@if (@ TempData["MensagemErro"] != null) {
    <div class="alert alert-danger" role="alert">
        @ TempData["MensagemErro"]</div>
}
<h2>Habilidades do Personagem</h2>
<table class="table">
    <tr>
        <th>
            @Html.DisplayName("Personagem")
        </th>
        <th>
            @Html.DisplayName("Habilidade")
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Habilidade.Dano)
        </th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>@Html.DisplayFor(modelItem => item.Personagem.Nome) </td>
            <td>@Html.DisplayFor(modelItem => item.Habilidade.Nome) </td>
            <td>@Html.DisplayFor(modelItem => item.Habilidade.Dano) </td>
            <td>@Html.ActionLink("Deletar", "Delete"
                , new { habilidadeId = item.HabilidadeId, personagemId = item.PersonagemId })
                , new { onclick = "return confirm('Deseja realmente deletar esta habilidade ?');"})
            </td>
        </tr>
    }
</table>
<div>
    @Html.ActionLink("Retornar aos Personagens", "Index", "Personagens")
</div>
```



6. Abra a View Index de Personagens e adicione o link para exibir a palavra “Listar Habilidades”, que carregará o método Index da controller de PersonagemHabilidades, passando o Id do personagem

```
@Html.ActionLink("Deletar", "Delete", new { id = item.Id })
    , new { onclick = "return confirm('Deseja realmente deletar este personagem ?');"}) |  
  
@Html.ActionLink("Listar Habilidades", "Index", "PersonagemHabilidades", new {id = item.Id})
```

7. Crie um método na controller de PersonagemHabilidades para receber os ids da habilidade e do personagem para que a API remova os dados da base.

```
[HttpGet("Delete/{habilidadeId}/{personagemId}")]
0 references
public async Task<ActionResult> DeleteAsync(int habilidadeId, int personagemId)
{
    try
    {
        HttpClient httpClient = new HttpClient();
        A string uriComplementar = "DeletePersonagemHabilidade";
        B string token = HttpContext.Session.GetString("SessionTokenUsuario");
        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);

        PersonagemHabilidadeViewModel ph = new PersonagemHabilidadeViewModel();
        C ph.HabilidadeId = habilidadeId;
        ph.PersonagemId = personagemId;

        var content = new StringContent(JsonConvert.SerializeObject(ph));
        D content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
        HttpResponseMessage response = await httpClient.PostAsync(uriBase + uriComplementar, content);
        string serialized = await response.Content.ReadAsStringAsync();

        if(response.StatusCode == System.Net.HttpStatusCode.OK)
        E    TempData["Mensagem"] = "Habilidade removida com sucesso";
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
    }
    F return RedirectToAction("Index", new {Id = personagemId});
}
```

- (A) Rota da API sendo guardada para concatenar com o endereço base
(B) Recuperação do Token da sessão e armazenamento dela no httpClient
(C) Objeto sendo preenchido pelo que foi passado nos parâmetros do método
(D) Serialização do objeto, envio para a endereço completo da API e armazenamento do retorno em serialized.
(E) If/Else de acordo com o código de retorno
(F) O retorno do método será para a mesma listagem de habilidades do personagem.



8. Crie mais um link na Index de Personagens

```
@Html.ActionLink("Listar Habilidades", "Index", "PersonagemHabilidades", new {id = item.Id}) |  
@Html.ActionLink("Atribuir Habilidade", "Create", "PersonagemHabilidades", new {id = item.Id, nome = item.Nome })
```

- Perceba que estamos coletando o Id e nome do Personagem.

9. Crie o método que vai receber o Id e nome do personagem na controller de PersonagemHabilidades. Este método também consumirá a lista de habilidades na API pra poder apresentar um DropDownList que será carregado através de uma ViewBag.

```
[HttpGet]  
0 references  
public async Task<ActionResult> CreateAsync(int id, string nome)  
{  
    try  
    {  
        string uriComplementar = "GetHabilidades";  
        HttpClient httpClient = new HttpClient();  
        string token = HttpContext.Session.GetString("SessionTokenUsuario");  
        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);  
        HttpResponseMessage response = await httpClient.GetAsync(uriBase + uriComplementar);  
  
        string serialized = await response.Content.ReadAsStringAsync();  
        List<HabilidadeViewModel> habilidades = await Task.Run(() =>  
            JsonConvert.DeserializeObject<List<HabilidadeViewModel>>(serialized));  
        ViewBag.ListaHabilidades = habilidades;  
  
        PersonagemHabilidadeViewModel ph = new PersonagemHabilidadeViewModel();  
        ph.Personagem = new PersonagemViewModel();  
        ph.Habilidade = new HabilidadeViewModel();  
        ph.PersonagemId = id;  
        ph.Personagem.Nome = nome;  
  
        return View(ph);  
    }  
    catch (System.Exception ex)  
    {  
        TempData["MensagemErro"] = ex.Message;  
        return RedirectToAction("Create", new { id, nome });  
    }  
}
```



10. Crie a view Create.cshtml dentro da pasta PersonagemHabilidades com a codificação abaixo:

```
@model RpgMvc.Models.PersonagemHabilidadeViewModel
@{
    ViewBag.Title = "create";
}
<h2>Cadastro de Habilidade para @Model.Personagem.Nome </h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <hr />
        <div class="form-group">
            @Html.LabelFor(model => model.Personagem.Id, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-6">
                @Html.EditorFor(model => model.PersonagemId, new { htmlAttributes = new { @class = "form-control", @readonly = "readonly" } })
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.Personagem, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-6">
                @Html.EditorFor(model => model.Personagem.Nome, new { htmlAttributes = new { @class = "form-control", @readonly = "readonly" } })
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.Habilidade, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-6">
                @Html.DropDownListFor(model => model.HabilidadeId, new SelectList(@ViewBag.ListaHabilidades, "Id", "Nome"),
                    "----Selecione---", new { @class = "form-control" })
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-6">
                <input type="submit" value="Salvar" class="btn btn-primary" />
            </div>
        </div>
    </div>
}
<div>
    @Html.ActionLink("Retornar às habilidades ", "Index", "PersonagemHabilidades", new{id = Model.PersonagemId}) |
    @Html.ActionLink("Retornar aos Personagens", "Index", "Personagens")
</div>
```

- Execute para testar se o cadastro está aparecendo juntamente como DropDownList Carregado.



11. A última etapa para o exercício é construir o método que vai salvar a habilidade para o personagem.

Vamos criar um método Post chamado Create na controller de PersonagemHabilidades.

```
[HttpPost]  
0 references  
public async Task<ActionResult> CreateAsync(PersonagemHabilidadeViewModel ph)  
{  
    try  
    {  
        HttpClient httpClient = new HttpClient();  
        string token = HttpContext.Session.GetString("SessionTokenUsuario");  
        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);  
  
        var content = new StringContent(JsonConvert.SerializeObject(ph));  
        content.Headers.ContentType = new MediaTypeHeaderValue("application/json");  
        HttpResponseMessage response = await httpClient.PostAsync(uriBase, content);  
        string serialized = await response.Content.ReadAsStringAsync();  
  
        if (response.StatusCode == System.Net.HttpStatusCode.OK)  
            TempData["Mensagem"] = "Habilidade cadastrada com sucesso";  
        else  
            throw new System.Exception(serialized);  
    }  
    catch (System.Exception ex)  
    {  
        TempData["MensagemErro"] = ex.Message;  
    }  
    return RedirectToAction("Index", new { id = ph.PersonagemId});  
}
```

- Deixe a tabela Habilidade do Banco de dados com 10 habilidade para que possamos variar bastante na vinculação delas aos personagens.
- Execute e tente realizar o salvamento da habilidade. Perceba que depois de salvar, redirecionaremos o usuário para a controller que lista as habilidades de acordo com o personagem.



12. Crie o arquivo IndexDisputas.cshtml dentro da pasta Views/Disputas e insira a codificação abaixo

```
<!--Namespace da classe de Modelo para esta view-->
@model IEnumerable<RpgMvc.Models.DisputaViewModel>

@if (@ TempData[ "Mensagem" ] != null)
{
    <div class="alert alert-success" role="alert">
        @Html.Raw(@ TempData[ "Mensagem" ])</div>
}
<!--Configuração para exibir mensagem de erro -->
@if (@ TempData[ "MensagemErro" ] != null)
{
    <div class="alert alert-danger" role="alert">
        @ TempData[ "MensagemErro" ]</div>
}
@{ViewBag.Title = "Disputas"; }<!--Título da página para o navegador-->
<h2>Relação de Disputas</h2><!--Título da página-->
<table class="table">
    <tr><!--Títulos das colunas da tabela-->
        <th>@Html.DisplayNameFor(model => model.Id)</th>
        <th>@Html.DisplayNameFor(model => model.DataDisputa)</th>
        <th>@Html.DisplayNameFor(model => model.Narracao)</th>
        <th>@Html.DisplayNameFor(model => model.Resultados)</th>
        <th></th>
    </tr>
    <!--Looping para escrever os dados na tabela-->
    @foreach (var item in Model)
    {
        <tr>
            <td>@Html.DisplayFor(modelItem => item.Id)</td>
            <td>@Html.DisplayFor(modelItem => item.DataDisputa)</td>
            <td>@Html.DisplayFor(modelItem => item.Narracao)</td>
            <td>@Html.DisplayFor(modelItem => item.Resultados)</td>
        </tr>
    }
</table>
@Html.ActionLink("Apagar Disputas" , "ApagarDisputas" , "Disputas" , null
                ,new { @class = "btn btn-danger" , onclick = "return confirm('Deseja
realmente deletar este personagem ?');"} )
```



13. Abra a classe Disputas Controller e adicione o método IndexDisputas

```
[HttpGet]
public async Task<ActionResult> IndexDisputasAsync()
{
    try
    {
        string uriComplementar = "Listar";

        HttpClient httpClient = new HttpClient();
        string token = HttpContext.Session.GetString("SessionTokenUsuario");
        httpClient.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", token);

        HttpResponseMessage response = await httpClient.GetAsync(uriBase +
uriComplementar);
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            List<DisputaViewModel> lista = await Task.Run(() =>
                JsonConvert.DeserializeObject<List<DisputaViewModel>>(serialized));

            return View(lista);
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
        return RedirectToAction("Index");
    }
}
```



14. Ainda na Controller DisputasController adicione o método ApagarDisputas

```
[HttpGet]
    public async Task<ActionResult> ApagarDisputasAsync()
    {
        try
        {
            HttpClient httpClient = new HttpClient();

            string token = HttpContext.Session.GetString("SessionTokenUsuario");
            httpClient.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", token);
            string uriComplementar = "ApagarDisputas";

            HttpResponseMessage response = await httpClient.DeleteAsync(uriBase +
uriComplementar);
            string serialized = await response.Content.ReadAsStringAsync();

            if (response.StatusCode == System.Net.HttpStatusCode.OK)
                TempData["Mensagem"] = "Disputas apagadas com sucesso.";
            else
                throw new System.Exception(serialized);
        }
        catch (System.Exception ex)
        {
            TempData["MensagemErro"] = ex.Message;
        }
        return RedirectToAction("IndexDisputas", "Disputas");
    }
```

15. Crie um link na view _layout (Views/Shared/_layout.cshtml) para abrir a listagem de disputas

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Personagens"
       asp-action="Index">Personagens</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Disputas"
       asp-action="IndexDisputas">Disputas</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Home"
       asp-action="Privacy">Privacy</a>
</li>
```



16. Abra a view Personagens/Index.cshtml e inclua na tabela de personagens os links a seguir para restaurar pontos e zerar o ranking de um personagem individualmente

```
@Html.ActionLink("Listar Habilidades", "Index", "PersonagemHabilidades", new { id = item.Id }) |  
@Html.ActionLink("Atribuir Habilidade", "Create", "PersonagemHabilidades", new { id = item.Id, nome =  
item.Nome }) |  
  
@Html.ActionLink("Restaurar Pontos Vida", "RestaurarPontosVida", new { id = item.Id }  
, new { onclick = "return confirm('Deseja realmente restaurar pontos vida ?');"} ) |  
  
@Html.ActionLink("Zerar Ranking", "ZerarRanking", new { id = item.Id }  
, new { onclick = "return confirm('Deseja realmente zerar ranking ?');"} )  
|</td>
```

17. Inclua um botão (na área de botões) para zerar o ranking e os pontos de vida de todos os personagens

```
@Html.ActionLink("Clique aqui para um embate em grupo!!!", "DisputaGeral", "Disputas",  
null, new { @class = "btn btn-danger" })  
  
@Html.ActionLink("Zerar ranking e restaurar pontos vida", "ZerarRankingRestaurarVidas", "Personagens",  
null, new { @class = "btn btn-secondary", onclick = "return confirm('Deseja realmente realizar esta ação ?');"} )
```

18. Abra a controller de personagens e adicione o método que vai zerar o ranking e restaurar os pontos de todos os personagens

```
[HttpGet]  
public async Task<ActionResult> ZerarRankingRestaurarVidasAsync()  
{  
    try  
    {  
        HttpClient httpClient = new HttpClient();  
        string token = HttpContext.Session.GetString("SessionTokenUsuario");  
        httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", token);  
        string uriComplementar = "ZerarRankingRestaurarVidas";  
        HttpResponseMessage response = await httpClient.PutAsync(uriBase + uriComplementar, null);  
        string serialized = await response.Content.ReadAsStringAsync();  
        if (response.StatusCode == System.Net.HttpStatusCode.OK)  
            TempData["Mensagem"] = "Rankings zerados e vidas dos personagens restauradas com  
sucesso.";  
        else  
            throw new System.Exception(serialized);  
    }  
    catch (System.Exception ex)  
    { TempData["MensagemErro"] = ex.Message; }  
    return RedirectToAction("Index");  
}
```



19. Insira o método que restaurará os pontos de vida de um personagem

```
[HttpGet]
public async Task<ActionResult> RestaurarPontosVidaAsync(int id)
{
    try
    {
        string uriComplementar = "RestaurarPontosVida";
        PersonagemViewModel p = new PersonagemViewModel();
        p.Id = id;

        HttpClient httpClient = new HttpClient();
        string token = HttpContext.Session.GetString("SessionTokenUsuario");
        httpClient.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", token);
        var content = new StringContent(JsonConvert.SerializeObject(p));
        content.Headers.ContentType = new MediaTypeHeaderValue("application/json");

        HttpResponseMessage response = await httpClient.PutAsync(uriBase +
uriComplementar, content);
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            TempData["Mensagem"] = "Pontos de vida do personagem restaurados com
sucesso";
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
    }
    return RedirectToAction("Index");
}
```



20. Insira o método que vai Zerar o ranking de um personagem

```
[HttpGet]
public async Task<ActionResult> ZerarRankingAsync(int id)
{
    try
    {
        string uriComplementar = "ZerarRanking";
        PersonagemViewModel p = new PersonagemViewModel();
        p.Id = id;

        HttpClient httpClient = new HttpClient();
        string token = HttpContext.Session.GetString("SessionTokenUsuario");
        httpClient.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", token);
        var content = new StringContent(JsonConvert.SerializeObject(p));
        content.Headers.ContentType = new MediaTypeHeaderValue("application/json");

        HttpResponseMessage response = await httpClient.PutAsync(uriBase +
uriComplementar, content);
        string serialized = await response.Content.ReadAsStringAsync();

        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            TempData["Mensagem"] = "Ranking do personagem zerado com
sucesso";
        }
        else
            throw new System.Exception(serialized);
    }
    catch (System.Exception ex)
    {
        TempData["MensagemErro"] = ex.Message;
    }
    return RedirectToAction("Index");
}
```

Entregas:

- Print do resultado de uma disputa geral exibindo o vencedor.
- Print da tela após o clique no botão que zera as disputas e restaura os pontos de vida dos personagens.
- Print da tela inicial da controller de disputas que exibe o histórico como data da disputa, a narração e os resultados.



Trabalho de Conclusão do Tópico:

1. Criar a classe **ArmaViewModel.cs** na pasta Models, com as propriedades similares a classe Arma do projeto de API.
2. Criar a controller **ArmasController.cs** na pasta Controllers e todos os métodos necessários para consumir a API publicada no Somee.
3. Criar a pasta **Armas** dentro da pasta Views e dentro dela as views Index, Create, Details e Edit.
 - a. Um detalhe muito importante é que a tabela de armas precisa do ID de um personagem para vincular o personagem a arma. Você pode utilizar o método que lista os personagens, e carregar em um DropDownList como fizemos nas etapas para carregar as habilidades na disputa com habilidades.
4. O projeto deve estar com link na página principal para listar as armas, cadastrar, obtendo os detalhes de uma arma, editando uma arma já cadastrada e deletando, exibindo as mensagens nas operações de salvamento, edição e remoção.

Fazer o print do funcionamento de todas as Views com as devidas mensagens referente as operações, salvar em um arquivo e postar na tarefa criada Teams.



Aula 18 – Framework Micro ORM Dapper - API de compromissos no VS Code

1. Criar uma pasta chamada CompAPI e abrir esta pasta no VS Code. Abra o terminal e execute o comando “dotnet new webapi --framework net5.0”. Crie uma pasta chamada DAL e dentro desta pasta guarde o script de banco da aula. Além disso, execute as linhas de comando no Banco de dados do Somee para deixar as tabelas criadas.



É possível instalar uma extensão para acessar o SQL Server através do VS code e configurar uma conexão para poder rodar o script disponível abaixo

```
CREATE DATABASE DB_COMPROMISSOS
GO
use DB_COMPROMISSOS
GO
CREATE TABLE TB_TIPO_PARTICIPANTE
(
    ID INT IDENTITY(1,1) NOT NULL CONSTRAINT PK_TB_TIPO_PARTICIPANTE PRIMARY KEY,
    TIPO_PARTICIPANTE VARCHAR(50) not null
)
INSERT INTO TB_TIPO_PARTICIPANTE VALUES ('Professor')
INSERT INTO TB_TIPO_PARTICIPANTE VALUES ('Aluno')
GO
CREATE TABLE TB_PARTICIPANTE
(
    ID INT IDENTITY(1,1) NOT NULL CONSTRAINT PK_TB_PARTICIPANTE PRIMARY KEY,
    ID_TIPO_PARTICIPANTE INT NOT NULL CONSTRAINT FK_PART_TB_TIPO_PARTICIPANTE FOREIGN KEY REFERENCES TB_TIPO_PARTICIPANTE(ID),
    TX_NOME VARCHAR (50) NOT NULL,
    TX_CPF VARCHAR(11) NOT NULL,
    TX_EMAIL VARCHAR(50) NOT NULL
)
go
INSERT INTO TB_PARTICIPANTE VALUES (1, 'Professor da Silva', '123', 'psilva@gmail.com');
INSERT INTO TB_PARTICIPANTE VALUES (2, 'Aluno Borges', '321', 'aborges@gmail.com');
```

2. Crie uma pasta chamada Models e dentro dela um arquivo de classe chamado **Participante.cs** e entre as escolhas na lâmpada que aparecerá, selecione “*Change namespace to CompAPI.Participante*” para a classe. Utilize o comando prop + TAB para criar as propriedades a seguir

```
0 references
public int Id { get; set; }
0 references
public int TipoId { get; set; }
0 references
public string Nome { get; set; }
0 references
public string Cpf { get; set; }
0 references
public string Email { get; set; }
```



3. Abra arquivo appSettings.json e configure os dados da string de conexão preenchendo com o endereço do local. Deixe a descrição da string numa linha só. Abaixo você pode copiar o exemplo.

```
tsql          {} appsettings.json X  C# ConnectionFactory.cs
settings.json > ...
{
  "ConnectionStrings": {
    "ConexaoLocal": "SUA STRING DE CONEXÃO"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

Exemplo string de conexão: server=localhost,1433;database=DB_COMPROMISSOS;User ID=sa;Password=123456;TrustServerCertificate=True

4. Abra o terminal e adicione o pacote para uso do Driver de Banco de Dados Sql Server, através do comando a seguir

```
$\CompAPI> dotnet add package Microsoft.Data.SqlClient
```

5. Crie uma classe chamada ConnectionFactory na pasta DAL e programe conforme abaixo. Dê o namespace adequado e adicione using às referências: using Microsoft.Extensions.Configuration, System.Data e Microsoft.Data.SqlClient

```
public class ConnectionFactory
{
  1 reference
  public static string nomeConexao = "ConexaoSomee";
  5 references
  public static IDbConnection GetStringConexao(IConfiguration config)
  {
    return new SqlConnection(config.GetConnectionString(nomeConexao));
  }
}
```

6. Abra a classe launchSettings.Json da pasta Properties e remova o endereço https que aparece na propriedade applicationUrl

```
"CompDS": [
  "commandName": "Project",
  "launchBrowser": true,
  "launchUrl": "weatherforecast",
  "applicationUrl": "https://localhost:5001;http://localhost:5000",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
]
```

- Isso é necessário para que ao rodar localmente o projeto, não ocorram problemas por não existir certificado de conexão segura.



7. Na pasta Controllers crie uma classe chamada **ParticipantesController.cs** que vai herdar de ControllerBase, sendo que esta herança pede o *using Microsoft.AspNetCore.Mvc*.

- Perceba também estamos declarando uma variável do tipo IConfiguration que pedirá o *using Microsoft.Extensions.Configuration* e que será inicializada no construtor para obter as configurações do Banco de dados

```
[ApiController]
[Route("[controller]")]
0 references
public class ParticipantesController : ControllerBase
{
    2 references
    private readonly IConfiguration _config;
    0 references
    public ParticipantesController(IConfiguration config)
    {
        _config = config;
    }
}
```

8. Use o terminal para adicionar o Dapper, que será o Micro ORM que utilizaremos para realizar as operações no Banco de dados: *dotnet add package Dapper*

9. Programe o método para receber obter os participantes conforme abaixo

```
[HttpGet]
0 references
public async Task<IActionResult> GetAllAsync()
{
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))
    {
        conexao.Open();

        StringBuilder sql = new StringBuilder();
        sql.Append("SELECT ID as Id, ID_TIPO_PARTICIPANTE as TipoId, TX_NOME as Nome, ");
        sql.Append("TX_CPF as Cpf , TX_EMAIL as Email ");
        sql.Append("FROM TB_PARTICIPANTE ");

        List<Participante> lista = (await conexao.QueryAsync<Participante>(sql.ToString())).ToList();

        return Ok(lista);
    }
}
```

- Usings necessários: System.Threading.Tasks; System.Collections.Generic; CompAPI.Models; System.Data; CompAPI.DAL; System.Text; Dapper; System.Linq; System.



10. Insira o método que vai obter um participante

```
[HttpGet("{id}")]
0 references
public async Task<ActionResult> GetByIdAsync(int id)
{
    Participante p = null;
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))
    {
        conexao.Open();

        StringBuilder sql = new StringBuilder();
        sql.Append("SELECT ID as Id, ID_TIPO_PARTICIPANTE as TipoId, TX_NOME as Nome, TX_CPF as Cpf , ");
        sql.Append("TX_EMAIL as Email FROM TB_PARTICIPANTE WHERE ID = @Id ");

        p = await conexao.QueryFirstOrDefaultAsync<Participante>(sql.ToString(), new {Id = id});

        if (p != null)
            return Ok(p);
        else
            return NotFound("Participante não encontrado.");
    }
}
```

11. Utilize o terminal para adicionar o pacote do Dapper.Contrib: dotnet add package Dapper.Contrib

12. Faça as modificações conforme abaixo e adicione o using ao Dapper.Contrib.Extensions

```
[Table("TB_PARTICIPANTE")]
4 references
public class Participante
{
    [Key]
    0 references
    public int Id { get; set; }
```

13. Crie o método para inserir. Necessitará do using System.

```
[HttpPost]
0 references
public async Task<ActionResult> InsertAsync(Participante p)
{
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))
    {
        conexao.Open();

        StringBuilder sql = new StringBuilder();
        sql.Append("INSERT INTO TB_PARTICIPANTE (ID_TIPO_PARTICIPANTE, TX_NOME, TX_CPF, TX_EMAIL) ");
        sql.Append("VALUES (@TipoId, @Nome, @Cpf, @Email) ");
        sql.Append("SELECT CAST(SCOPE_IDENTITY() AS INT) ");

        object o = await conexao.ExecuteScalarAsync(sql.ToString(), p);

        if(o != null)
            p.Id = Convert.ToInt32(o);
    }
    return Ok(p);
}
```



14. Crie o método para atualizar

```
[HttpPost]  
0 references  
public async Task<ActionResult> UpdateAsync(Participante p)  
{  
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))  
    {  
        conexao.Open();  
  
        StringBuilder sql = new StringBuilder();  
        sql.Append("UPDATE TB_PARTICIPANTE SET ");  
        sql.Append("ID_TIPO_PARTICIPANTE = @TipoId, TX_NOME = @Nome, TX_CPF = @Cpf, TX_EMAIL = @Email ");  
        sql.Append("WHERE ID = @Id ");  
  
        int linhasAfetadas = await conexao.ExecuteAsync(sql.ToString(), p);  
        return Ok(p);  
    }  
}
```

15. Crie o método para deletar

```
[HttpDelete("{id}")]  
0 references  
public async Task<IActionResult> DeleteAsync(int id)  
{  
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))  
    {  
        conexao.Open();  
  
        StringBuilder sql = new StringBuilder();  
        sql.Append("DELETE FROM TB_PARTICIPANTE ");  
        sql.Append("WHERE ID = @Id ");  
  
        int linhasAfetadas = await conexao.ExecuteAsync(sql.ToString(), new { Id = id});  
        return Ok(linhasAfetadas);  
    }  
}
```

- Execute o projeto para testar com o Postman a API recém-criada.



Postman

O Postman é um API Client que podemos utilizar para realizar as requisições na API através dos principais métodos: Get, post, put e delete. Execute a aplicação e realize as seguintes configurações no Postman para poder testar a API

Get:

GET Send Save

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

1

Body Cookies Headers (4) Test Results Status: 200 OK Time: 5.24 s Size: 420 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 2,
3   "tipoId": 2,
4   "nome": "Aluno Borges",
5   "cpf": "321",
6   "email": "aborges@gmail.com"
```

Pode ser usada a variação da url para buscar apenas um Participante

GET Send Save

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

Post

POST Send Save

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

1 {
2 "Nome": "Carlos",
3 "TipoId": 2,
4 "Cpf": "12345678900",
5 "Email": "carlos@gmail.com"
6 }

Body Cookies Headers (4) Test Results Status: 200 OK Time: 537 ms Size: 149 B Save Response

Pretty Raw Preview Visualize JSON

1 6 → Retorno



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

Put

The screenshot shows a Postman request configuration for a PUT operation. The URL is `http://localhost:5000/Participantes`. The 'Body' tab is selected, showing a JSON payload:

```
1 {
2     "Id": 6,
3     "Nome": "Carlos Sobre nome adicionado",
4     "TipoId": 2,
5     "Cpf": "12345678900",
6     "Email": "carlos@gmail.com"
7 }
```

The response status is 200 OK, time 276 ms, size 149 B.

Delete

The screenshot shows a Postman request configuration for a DELETE operation. The URL is `http://localhost:5000/Participantes/6`. The 'Body' tab is selected, showing no body.

The response status is 200 OK, time 220 ms, size 149 B.

16. Para gerar os arquivos de publicação da API, use o terminal ou o Powershell e estando na pasta da API, realize o comando abaixo. Será criada uma pasta *publish* automaticamente dentro da pasta do projeto com os arquivos de publicação.

```
\CompAPI> dotnet publish -c release -o ./publish
```



Aula 19 – Dapper e Dapper.Contrib - Validação das propriedades de uma classe

1. Adicione o script abaixo ao arquivo DAL e execute no Banco de dados

```
go
CREATE TABLE TB_TIPO_COMPROMISSO
(ID INT IDENTITY(1,1) NOT NULL CONSTRAINT PK_TB_TIPO_COMPROMISSO PRIMARY KEY,
 TX_DESCRICAO VARCHAR (50) NOT NULL)

go
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Aula')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Palestra')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Visita Técnica')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Feira')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Congresso')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('TCC')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Conselho')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Planejamento')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Reunião Pedagógica')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Reunião de País')
INSERT INTO TB_TIPO_COMPROMISSO VALUES ('Festividade')

GO
CREATE TABLE TB_COMPROMISSO
(ID INT IDENTITY(1,1) NOT NULL CONSTRAINT PK_TB_COMPROMISSO PRIMARY KEY,
 ID_TIPO_COMPROMISSO INT NOT NULL CONSTRAINT FK_CCOMP_TB_TIPO_COMPROMISSO FOREIGN KEY REFERENCES
TB_TIPO_COMPROMISSO(ID),
 TX_DESCRICAO VARCHAR(50) not null,
 TX_LOCALIZACAO VARCHAR(500) not null,
 DT_INICIO DATETIME NOT NULL,
 DT_TERMINO DATETIME NOT NULL,
 FL_VISIVEL BIT CONSTRAINT DF_FL_VISIVEL DEFAULT 1)

INSERT INTO TB_COMPROMISSO (ID_TIPO_COMPROMISSO, TX_DESCRICAO, TX_LOCALIZACAO, DT_INICIO, DT_TERMINO,
FL_VISIVEL)
VALUES (1, 'Aula 01/07/2021', 'Aula Remota Teams', '2021-07-01', '2021-07-01', 1)

GO --Não usaremos essas tabelas, mas fica como exemplo de vinculação do compromisso com os participantes
CREATE TABLE TB_COMPROMISSO_PARTICIPANTE
(
    ID_COMPROMISSO INT NOT NULL CONSTRAINT FK_CCP_CCOMPROMISSO FOREIGN KEY REFERENCES
TB_COMPROMISSO(ID),
    ID_PARTICIPANTE INT NOT NULL CONSTRAINT FK_CCP_PARTICIPANTE FOREIGN KEY REFERENCES
TB_PARTICIPANTE(ID))

GO
CREATE TABLE TB_COMPROMISSO_IMAGENS
(ID int IDENTITY(1,1) NOT NULL,
 ID_COMPROMISSO int NOT NULL CONSTRAINT FK_IM_CCOMPROMISSO FOREIGN KEY REFERENCES
TB_COMPROMISSO(ID),
 VB_CONTEUDO_FOTO varbinary(max) NULL,
 TX_CAMINHO_FOTO varchar(max) NULL,
 TX_OBSERVACOES varchar(500) NULL)
```



-
2. Crie a classe TipoCompromisso.cs dentro da pasta Models. Exigirá o using Dapper.Contrib.Extensions

```
[Table("TB_TIPO_COMPROMISSO")]
0 references
public class TipoCompromisso
{
    [Key]
    0 references
    public int Id { get; set; }
    0 references
    public string DescricaoCompromisso { get; set; }
}
```

3. Crie a classe Compromisso.cs na pasta Models. Exigirá o using Dapper.Contrib.Extensions e System

```
[Table("TB_COMPROMISSO")]
0 references
public class Compromisso
{
    0 references
    public int Id { get; set; }
    0 references
    public int TipoCompromissoId { get; set; }
    0 references
    public string Descricao { get; set; }
    0 references
    public string Localizacao { get; set; }
    0 references
    public DateTime DataInicio { get; set; }
    0 references
    public DateTime DataTermino { get; set; }
    0 references
    public bool Visivel { get; set; }
    0 references
    public int ParticipanteId { get; set; }
    0 references
    public List<Participante> Participantes { get; set; }
}
```



4. Observe e realize as anotações a seguir nas propriedades. Exigirá o using System.ComponentModel.DataAnnotations;

```
[Required(ErrorMessage="Escolha o tipo de compromisso")]
0 references
public int TipoCompromissoId { get; set; }

[Required(ErrorMessage="O campo descrição é obrigatório")]
[MaxLength(30, ErrorMessage="O campo descrição deve conter até 30 caracteres")]
[MinLength(10, ErrorMessage="O campo descrição deve conter pelo menos 10 caracteres")]
0 references
public string Descricao { get; set; }

0 references
public string Localizacao { get; set; }

[Required(ErrorMessage="A data de início é obrigatória")]
0 references
public DateTime DataInicio { get; set; }
```

5. Crie a classe TiposCompromissoController e CompromissosController realizando a configuração básica da controller deste projeto. Exigirá os usings *Microsoft.AspNetCore.Mvc* e *Microsoft.Extensions.Configuration*

```
[ApiController]
[Route("[controller]")]
0 references
public class TiposCompromissoController : Controller
{
    3 references
    private readonly IConfiguration _config;
    0 references
    public TiposCompromissoController(IConfiguration config)
    {
        _config = config;
    }
}
```

```
[ApiController]
[Route("[controller]")]
0 references
public class CompromissosController : ControllerBase
{
    6 references
    private readonly IConfiguration _config;
    0 references
    public CompromissosController(IConfiguration config)
    {
        _config = config;
    }
}
```



-
6. Insira os métodos a seguir na classe CompromissosController. Utilize os mesmos usings de ParticipantesController

```
[HttpGet]
public async Task<ActionResult> GetAllAsync()
{
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))
    {
        conexao.Open();

        StringBuilder sql = new StringBuilder();
        sql.Append("SELECT ID as Id, ID_TIPO_COMPROMISSO as TipoCompromissoId, TX_DESCRICAO as Descricao, TX_
LOCALIZACAO as Localizacao, ");
        sql.Append("DT_INICIO as DataInicio, DT_TERMINO as DataTermino, FL_VISIVEL as Visivel ");
        sql.Append("FROM TB_COMPROMISSO ");

        List<Compromisso> lista = (await conexao.QueryAsync<Compromisso>(sql.ToString())).ToList();

        return Ok(lista);
    }
}

[HttpGet("{id}")]
public async Task<ActionResult> GetByIdAsync(int id)
{
    Compromisso c = null;
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))
    {
        conexao.Open();

        StringBuilder sql = new StringBuilder();
        sql.Append("SELECT ID as Id, ID_TIPO_COMPROMISSO as TipoCompromissoId, TX_DESCRICAO as Descricao, TX_
LOCALIZACAO as Localizacao, ");
        sql.Append("DT_INICIO as DataInicio, DT_TERMINO as DataTermino, FL_VISIVEL as Visivel ");
        sql.Append("FROM TB_COMPROMISSO WHERE ID = @Id ");

        c = await conexao.QueryFirstOrDefaultAsync<Compromisso>(sql.ToString(), new { Id = id});

        if (c != null)
            return Ok(c);
        else
            return NotFound("Compromisso não encontrado.");
    }
}

[HttpPost]
public async Task<ActionResult> InsertAsync(Compromisso c)
```



DESENVOLVIMENTO DE SISTEMAS – 2022/1

Luiz Fernando Souza / Quitéria Danno

```
{  
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))  
    {  
        conexao.Open();  
  
        StringBuilder sql = new StringBuilder();  
        sql.Append("INSERT INTO TB_COMPROMISSO (ID_TIPO_COMPROMISSO, TX_DESCRICAO, TX_LOCALIZACAO, DT_INICIO,  
DT_TERMINO, FL_VISIVEL) ");  
        sql.Append("VALUES (@TipoCompromissoId, @Descricao, @Localizacao, @DataInicio, @DataTermino, @Visivel  
) ");  
        sql.Append("SELECT CAST(SCOPE_IDENTITY() AS INT) ");  
  
        object o = await conexao.ExecuteScalarAsync(sql.ToString(), c);  
  
        if(o != null)  
            c.Id = Convert.ToInt32(o);  
    }  
    return Ok(c);  
}  
[HttpPut]  
public async Task<ActionResult> UpdateAsync(Compromisso c)  
{  
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))  
    {  
        conexao.Open();  
  
        StringBuilder sql = new StringBuilder();  
        sql.Append("UPDATE TB_COMPROMISSO SET ");  
        sql.Append("ID_TIPO_COMPROMISSO = @TipoCompromissoId, ");  
        sql.Append("TX_DESCRICAO = @Descricao, ");  
        sql.Append("TX_LOCALIZACAO = @Localizacao, ");  
        sql.Append("DT_INICIO = @DataInicio, ");  
        sql.Append("DT_TERMINO = @DataTermino, ");  
        sql.Append("FL_VISIVEL = @Visivel ");  
        sql.Append("WHERE ID = @Id ");  
  
        int linhasAfetadas = await conexao.ExecuteNonQuery(sql.ToString(), c);  
        return Ok(c);  
    }  
}  
[HttpDelete("{id}")]  
public async Task<ActionResult> DeleteAsync(int id)  
{
```



```
using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))
{
    conexao.Open();

    StringBuilder sql = new StringBuilder();
    sql.Append("DELETE FROM TB_COMPROMISSO ");
    sql.Append("WHERE ID = @Id ");

    int linhasAfetadas = await conexao.ExecuteAsync(sql.ToString(), new { Id = id });
    return Ok(linhasAfetadas);
}
```

7. E os métodos a seguir na classe TiposCompromissoController

```
[HttpGet]
public async Task<ActionResult> GetAllAsync()
{
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))
    {
        conexao.Open();

        StringBuilder sql = new StringBuilder();

        sql.Append("SELECT ID as Id, TX_DESCRICA0 as DescricaoCompromisso ");

        sql.Append("FROM TB_TIPO_COMPROMISSO ");

        List<TipoCompromisso> lista = (await conexao.QueryAsync<TipoCompromisso>(sql.ToString())).ToList();
        return Ok(lista);
    }
}
```



```
[HttpGet("{id}")]
public async Task<ActionResult> GetByIdAsync(int id)
{
    TipoCompromisso p = null;
    using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))
    {
        conexao.Open();
        StringBuilder sql = new StringBuilder();
        sql.Append("SELECT ID as Id, TX_DESCRICA0 as DescricaoCompromisso ");
        sql.Append("FROM TB_TIPO_COMPROMISSO WHERE ID = @Id ");

        p = await conexao.QueryFirstOrDefaultAsync<TipoCompromisso>(sql.ToString(), new {Id = id});

        if (p != null)
            return Ok(p);
        else
            return NotFound("Tipo de Compromisso não encontrado.");
    }
}
```

8. Faça a alteração no método de Insert da classe CompromissosController para que seja feita validação antes de enviar os dados ao banco.

```
public async Task<ActionResult> InsertAsync(Compromisso c)
{
    if (ModelState.IsValid)
    {
        using ( IDbConnection conexao = ConnectionFactory.GetStringConexao(_config))
        {
            conexao.Open();

            StringBuilder sql = new StringBuilder();
            sql.Append("INSERT INTO TB_COMPROMISSO (ID_TIPO_COMPROMISSO, TX_DESCRICA0, @Localizacao, @DataI");
            sql.Append("VALUES (@TipoCompromissoId, @Descricao, @Localizacao, @DataI");
            sql.Append("SELECT CAST(SCOPE_IDENTITY() AS INT) ");

            object o = await conexao.ExecuteScalarAsync(sql.ToString(), c);

            if (o != null)
                c.Id = Convert.ToInt32(o);
        }
        return Ok(c);
    }
    else
        return BadRequest(ModelState);
}
```

- Execute a API e realize o teste no postman deixando de enviar a descrição ou a data, por exemplo.
- Veja outros tipos de validação: http://www.macoratti.net/13/12/c_vdda.htm ou em <https://docs.microsoft.com/pt-br/aspnet/core/mvc/models/validation?view=aspnetcore-5.0>



Aula 20 - Projeto ASP NET Core MVC utilizando Entity Framework – Scaffold e Code Generator

1. Abra o PowerShell e navegue (comando cd) até a pasta em que deseja criar o projeto. Crie uma pasta chamada CompMVC e navegue para dentro da mesma

```
>-1> md CompMVC
>-1> cd CompMVC
>-1\CompMVC>
```

2. Crie o projeto através da linha de comando a seguir

```
dotnet new mvc --auth individual
```

- Execute o comando “code .” para abrir o projeto no VS Code.
 - Quando perguntado para executar o C# (Canto inferior direito), clique em Yes.
3. Execute o script disponibilizado para a aula. Nesta aula, criaremos uma classe no VS Code através de uma tabela já existente no Banco, para isso teremos que instalar alguns pacotes via linha de comando utilizando o terminal (Menu View → Terminal).

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet tool install -g dotnet-aspnet-codegenerator
```

- Observe que o arquivo CompMVC.csproj adicionou as referências aos pacotes.
4. Execute o comando abaixo no terminal para que criemos uma classe a partir de uma tabela do banco de dados, não se esquecendo de inserir sua string de conexão do somee dentro das aspas duplas:

```
CompMVC> dotnet ef dbcontext scaffold "Sua String de Conexão do Somee"
Microsoft.EntityFrameworkCore.SqlServer -o Models -f -c DataContext
```

- dotnet ef dbcontext → Comando
 - Microsoft.EntityFrameworkCore.SqlServer → Provider do banco de dados
 - -o Models → Local em que as classes serão criadas
 - -f → Sobrescreve um código anteriormente gerado
 - -c DataContext → nome do contexto utilizado na aplicação
-
- Após execução do comando, perceba que temos classes criadas na pasta Models. Desconsidere as



5. Arraste a classe **DataContext** da pasta Models para a pasta **Data**. Altere o namespace da classe DataContext para adequá-lo a nova pasta e adicione o using para CompMVC.Models
6. Altere o método ConfigureServices da classe Startup.cs para utilização do Banco SQL Server definindo uma chave chamada ConexaoSomee, conforme a seguir

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlite(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddDbContext<DataContext>(x =>
        x.UseSqlServer(Configuration.GetConnectionString("ConexaoSomee")));
}
```

7. No arquivo AppSettings.json devemos configurar o nome da chave inserida anteriormente e a string de conexão do somee

```
".ConnectionStrings": {
    "DefaultConnection": "DataSource=app.db;Cache=Shared"
    , "ConexaoSomee": "Sua string de Conexão do Somee aqui"}
```

8. Criaremos a Controller e as Views para a classe TbParticipante através da linha de comando a seguir

```
dotnet aspnet-codegenerator controller -name CompromissosController --model
CompMVC.Models.TbCompromisso --dataContext CompMVC.Data.DataContext --
relativeFolderPath Controllers --useDefaultLayout --referenceScriptLibraries --readWriteActions
```

- dotnet aspnet-codegenerator → comando
- controller -name CompromissosController (Plural da classe de modelo + palavra “Controller”) → O nome da controller a ser criada
- --model CompMVC.Models.TbCompromisso → namespace da classe de modelo + nome da classe
- --dataContext CompMVC.Data.DataContext → nome da classe de contexto do Banco de dados
- --relativeFolderPath Controllers → Nome da pasta em que a controller se encontra
- Perceba que foi criada uma pasta Views e outra Controller. A primeira contém os layouts de tela criados automaticamente (técnica chamada de scaffolding) e a segunda pasta contém as classes Controller que farão a intermediação entre as classes de modelos, abastecidas com os dados e as Views que fará exibição das informações.



-
9. No arquivo Properties/launchsettings.json configure o uso apenas do endereço http conforme fizemos nas primeiras aulas com VS Code.

```
launchSettings.json > ...
"CompMVC": {
    "commandName": "Project",
    "launchBrowser": true,
    "applicationUrl": "http://localhost:5000",
    "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
    }
}
```

10. Configure também a classe Startup.cs para que ao iniciar, seja executada a Controller que criamos

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Compromissos}/{action=Index}/{id?}");
    endpoints.MapRazorPages();
});
```

Atividades

- Realizar através do codegenerator a criação da Controller e das Views de Participantes.

```
dotnet aspnet-codegenerator controller -name ParticipantesController --model
CompMVC.Models.TbParticipante --dataContext CompMVC.Data.DataContext --relativeFolderPath
Controllers --useDefaultLayout --referenceScriptLibraries
```

- Ajustar a descrição dos campos que foram criados na View.
- Criar link para as Controllers para que apareçam na página inicial.

Referências:

https://dev.to/_patrickgod/net-core-3-1-web-api-entity-framework-jumpstart-part-1-4jla

<https://www.youtube.com/watch?v=but7iqjopKM>