ETEC PROF. HORÁCIO AUGUSTO DA SILVEIRA

PROGRAMAÇÃO DE APLICATIVOS MOBILE II – 2023/1

Luiz Fernando Souza / Elaine Marion

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:xxxx/**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.

```
private 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.

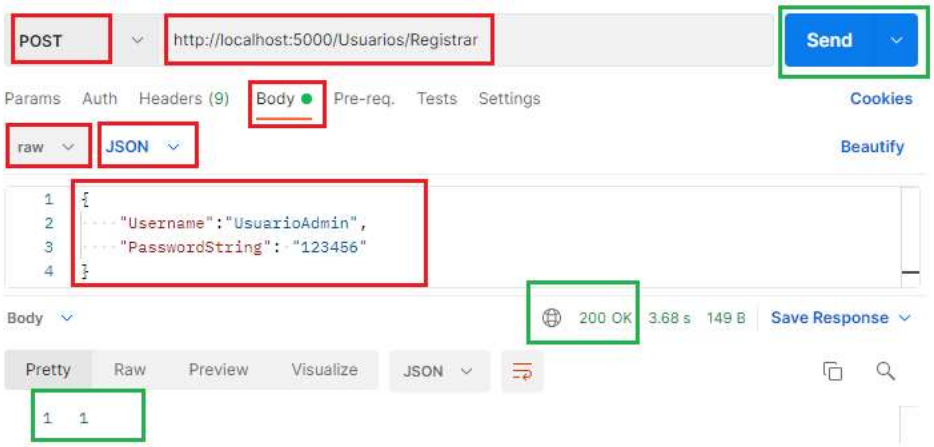
```
public static bool VerificarPasswordHash(string password, byte[] hash, byte[] salt)
        {
            using (var hmac = new System.Security.Cryptography.HMACSHA512(salt))
            {
                var computedHash =
hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));
        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

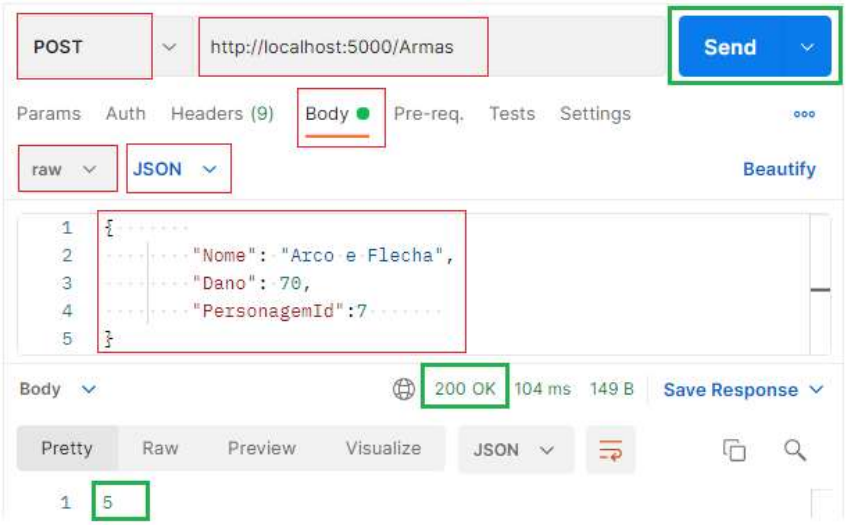
7. Realize o teste no *postman* para salvar um usuário e confira na base de dados. Depois tente realizar a operação com os mesmos dados, que conforme validação, não permitirá que o mesmo *userrname* seja registrado. Ao salvar os dados será retornado o Id do usuário inserido.



- 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. O body enviado será o mesmo do método registrar. Altere apenas a rota.

8. Insira a classe ArmasController.cs, caso você não tenha e teste o cadastro de Armas:



- 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.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*

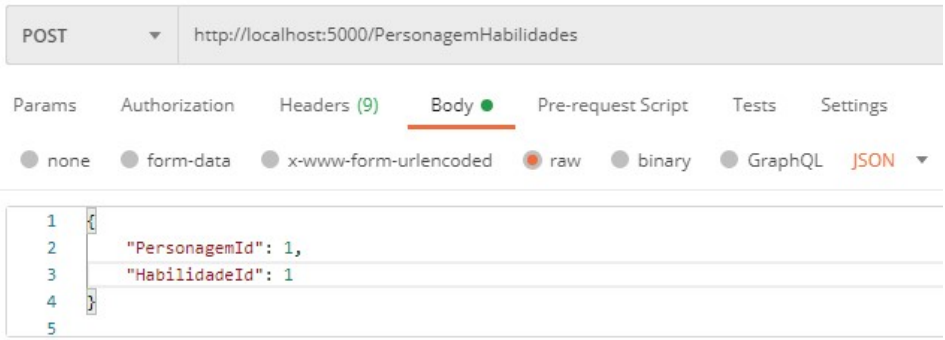
```
try
{
    Personagem personagem = await _context.Personagens
        .Include(p => p.Arma)
        .Include(p => p.PersonagemHabilidades).ThenInclude(ps => ps.Habilidade)
        .FirstOrDefaultAsync(p => p.Id == novoPersonagemHabilidade.PersonagemId);

    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);

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

    PersonagemHabilidade ph = new PersonagemHabilidade();
    ph.Personagem = personagem;
    ph.Habilidade = habilidade;
    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.*

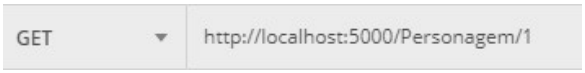
(C) Preenchemos o objeto que enviaremos para o contexto de base de dados com os objetos adquiridos nas etapas A e B.

14. Execute o teste no postman enviando os dados abaixo. Observe a porta que sua API roda.



- 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.



- 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,*** abra a classe Program.cs e adicione o trecho de código abaixo na área sinalizada no print mais abaixo. Servirá para evitar loopings nas classes que referenciam umas às outras nos relacionamentos.

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

Program.cs

```
using RpgApi.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<DataContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("ConexaoFreeAspHosting"));
});

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

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

ETEC PROF. HORÁCIO AUGUSTO DA SILVEIRA

PROGRAMAÇÃO DE APLICATIVOS MOBILE II – 2023/1

Luiz Fernando Souza / Elaine Marion

Atividade

- (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 “return Ok, a propriedade data de acesso do objeto “*usuario*” seja alimentada com a data/hora atual e salve as 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 e a entrega deverá ser os prints da tela do postman obtidos em cada uma das operações citadas acima. Você deve inserir os resultados em um documento PDF para anexar na entrega da atividade.