



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 de 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/seguranca-salt-hash-senha/>
 - Exemplo de criação de hash em C#: <https://www.youtube.com/watch?v=ggPgk4znUEY>
1. Abra o projeto **RpgApi** e abra classe **Usuario.cs** dentro da pasta Models, acrescentando as propriedades sinalizadas abaixo:

```
0 references  
public DateTime? DataAcesso { get; set; }  
  
[NotMapped]  
0 references  
public string PasswordString { get; set; } //using System.ComponentModel.DataAnnotations.Schema;  
0 references  
public List<Personagem> Personagens {get; set;}
```

- 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 ClassEnum 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. 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 Arma() { Id = 6, Nome = "Foice", Dano = 33 },  
new Arma() { Id = 7, Nome = "Cajado", Dano = 32 }  
);  
//Início da criação do 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;  
user.Perfil = "Admin";  
user.Email = "seuEmail@gmail.com";  
user.Latitude = -23.5200241;  
user.Longitude = -46.596498;  
modelBuilder.Entity<Usuario>().HasData(user);  
//Fim da criação do usuário padrão.  
//Define que se o Perfil não for informado, o valor padrão será jogador  
modelBuilder.Entity<Usuario>().Property(u => u.Perfil).HasDefaultValue("Jogador");  
//Fim do método OnModelCreating
```



5. 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 Migrations e a atualização da classe DataContextModelSnapshot.cs
6. Execute o comando “**dotnet ef database update**” para atualizar a base de dados e confira se o banco foi criado localmente. Caso exista algum bloqueio para a execução do comando no lab, gere o script para esta migração através do comando:

```
dotnet ef migrations script A B C -o ./script03_TabelaUsuarios.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 no somee. Atualize o banco e confirme que a chave estrangeira foi criada na tabela Personagens, bem como um campo de nome Usuariold

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.

7. 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.



8. 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 loppings 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
9. 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<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 },
    new Arma() { Id = 4, Nome = "Punho", Dano = 30, PersonagemId = 4 },
    new Arma() { Id = 5, Nome = "Chicote", Dano = 34, PersonagemId = 5 },
    new Arma() { Id = 6, Nome = "Foices", Dano = 33, PersonagemId = 6 },
    new Arma() { Id = 7, Nome = "Cajado", Dano = 32, PersonagemId = 7 }
);
```

10. 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.



11. 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 ./script04_TabelaArmas_Atualizacao.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!!!

12. 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; }
    }
}
```

13. 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; }
    }
}
```




14. 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; }
```

15. Ainda na classe de mapeamento (DataContext) adicionaremos ao método OnModelCreating, abaixo de onde populamos as armas, uma chave composta para a tabela PersonagemHabilidade (A), alguns dados de Habilidade para inserir na futura tabela Habilidades (B), além de Informações para relacionar o Personagem à Habilidade (C).

```
new Arma() { Id = 7, Nome = "Cajado", Dano = 32, PersonagemId = 7 }  
);  
  
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}  
);  
  
C modelBuilder.Entity<PersonagemHabilidade>().HasData  
(  
    new PersonagemHabilidade() { PersonagemId = 1, HabilidadeId =1 },  
    new PersonagemHabilidade() { PersonagemId = 1, HabilidadeId =2 },  
    new PersonagemHabilidade() { PersonagemId = 2, HabilidadeId =2 },  
    new PersonagemHabilidade() { PersonagemId = 3, HabilidadeId =2 },  
    new PersonagemHabilidade() { PersonagemId = 3, HabilidadeId =3 },  
    new PersonagemHabilidade() { PersonagemId = 4, HabilidadeId =3 },  
    new PersonagemHabilidade() { PersonagemId = 5, HabilidadeId =1 },  
    new PersonagemHabilidade() { PersonagemId = 6, HabilidadeId =2 },  
    new PersonagemHabilidade() { PersonagemId = 7, HabilidadeId =3 }  
);
```

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

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



17. Adicione a mesma propriedade na classe Personagem (A) e complemente com demais propriedades que vamos trabalhar mais à frente (B).

```
[JsonIgnore]
0 references
public Arma Arma { get; set; }
0 references
A public List<PersonagemHabilidade> PersonagemHabilidades { get; set; }
0 references
public int Disputas { get; set; }
0 references
B public int Victorias { get; set; }
0 references
public int Derrotas { 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**. Faça também a geração do script através do comando **"dotnet ef migrations script MigracaoUmParaUm MigracaoMuitosParaMuitos -o ./script05_TabelaHabilidades_TabelaPersonagemHabilidades.sql"**

- Confira se as tabelas e relacionamentos foram criados.