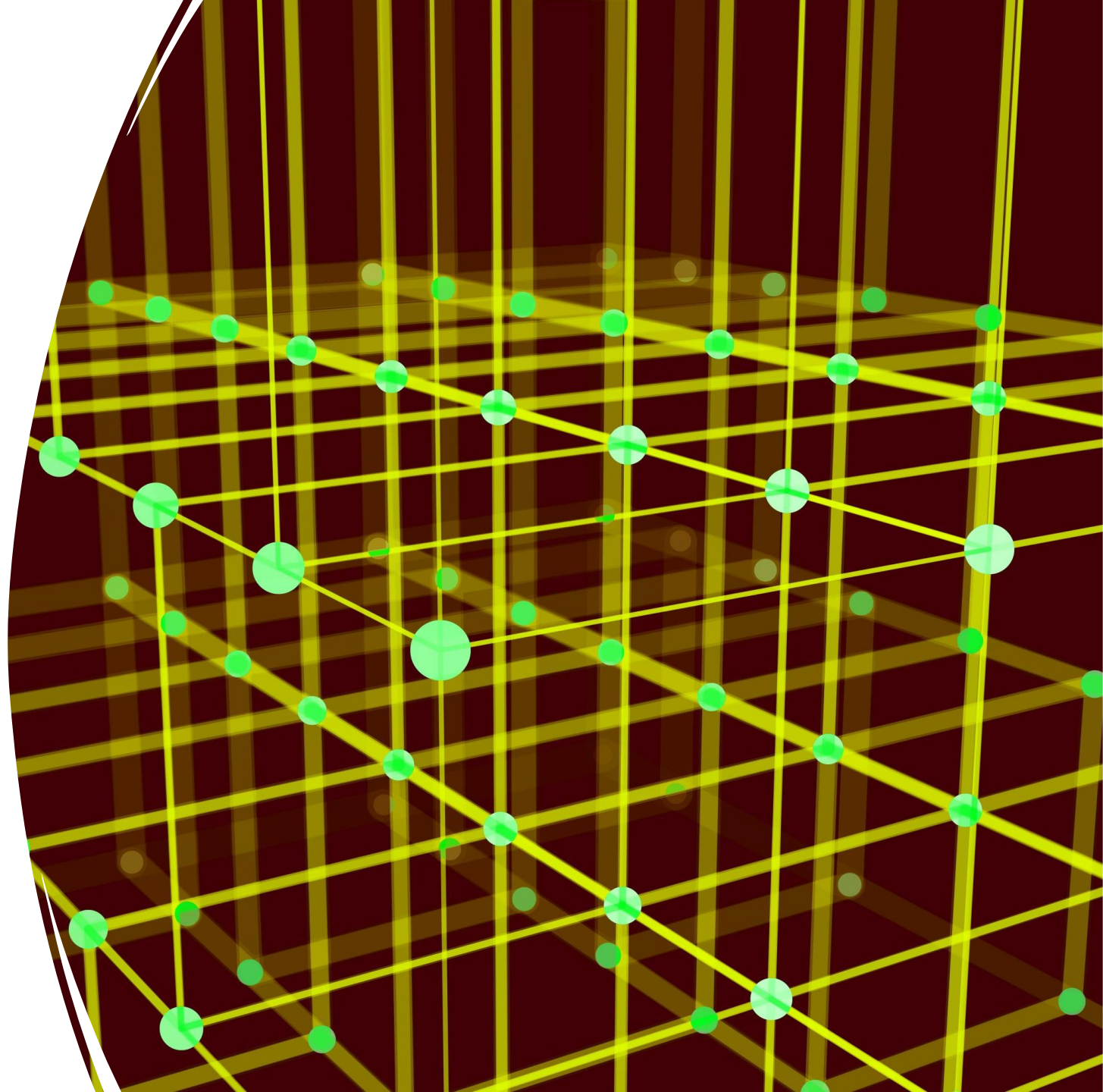


Entity Framework

Fabrício Tonetto Londero

Ricardo Frohlich da Silva



An abstract geometric pattern on the left side of the slide, featuring various colored squares (blue, purple, orange, green, pink, black, grey) and thin grey lines forming a network-like structure.

ORM : Object Relational Mapper

- ORM é uma técnica de mapeamento objeto relacional que permite fazer uma relação dos objetos com os dados que os mesmos representam.
- Surgiu por alguns programadores não se sentirem a vontade com código SQL e pela **produtividade** proporcionada

Exemplos

- Hibernate -> Java
- Nhibernate -> .Net
- Eloquent -> laravel(PHP)
- Entity -> .net
- JPA -> Java
- DjangoORM -> Python
- Sequelize -> Node
- Exposed -> Kotlin
- Dapper -> .Net

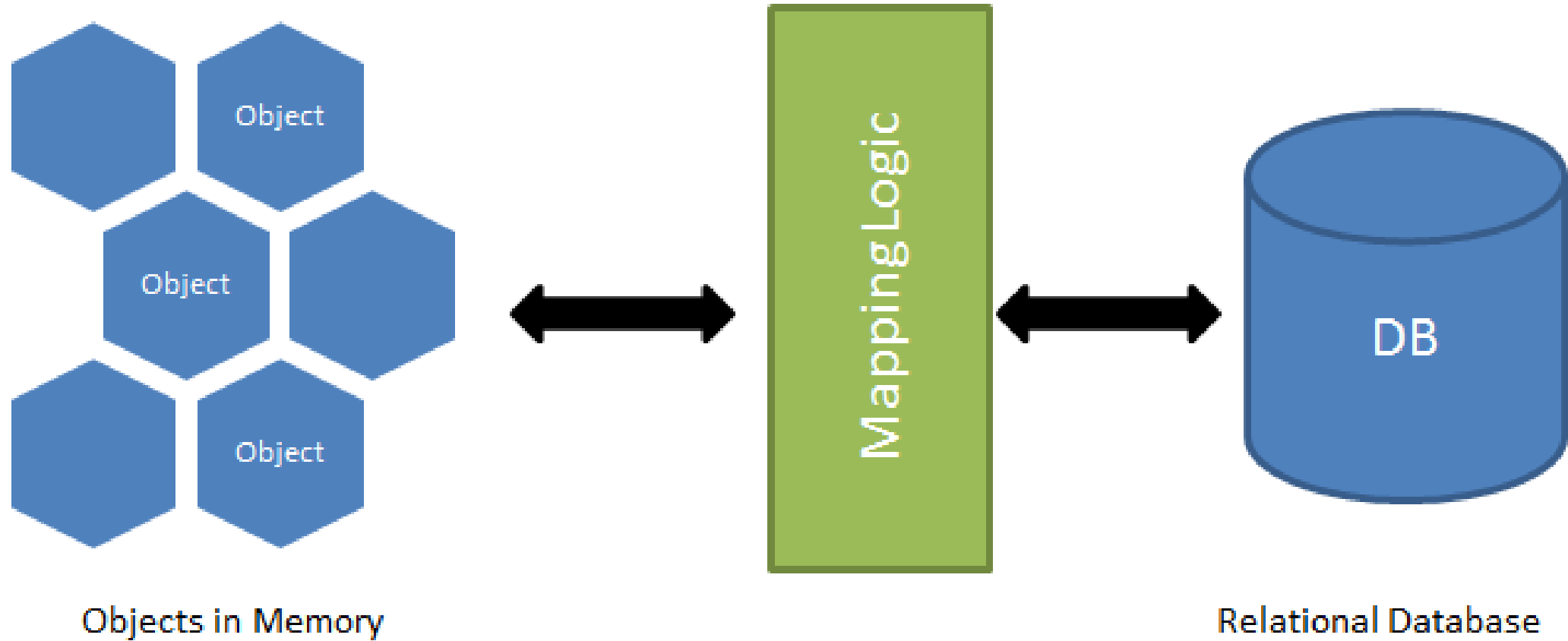
ORM



A ideia é o programador focar “No que fazer” e não no “Como fazer”!

Deve ser feito o mapeamento das classes para o banco de dados e cada ORM tem suas particularidades para gerar o SQL referente ao CRUD (inserção, alteração, remoção e consulta) do objeto que corresponde a uma tabela no banco de dados.

O/R Mapping



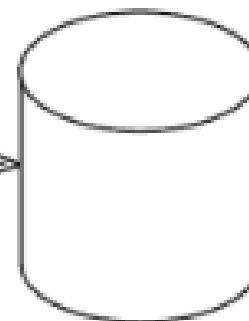
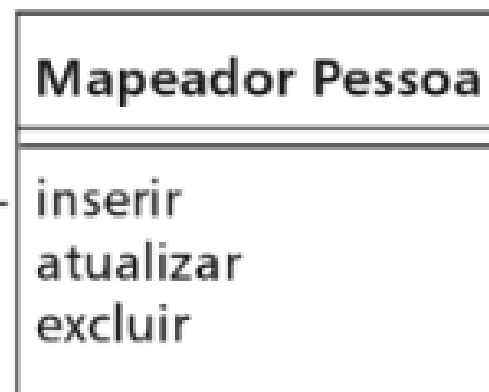
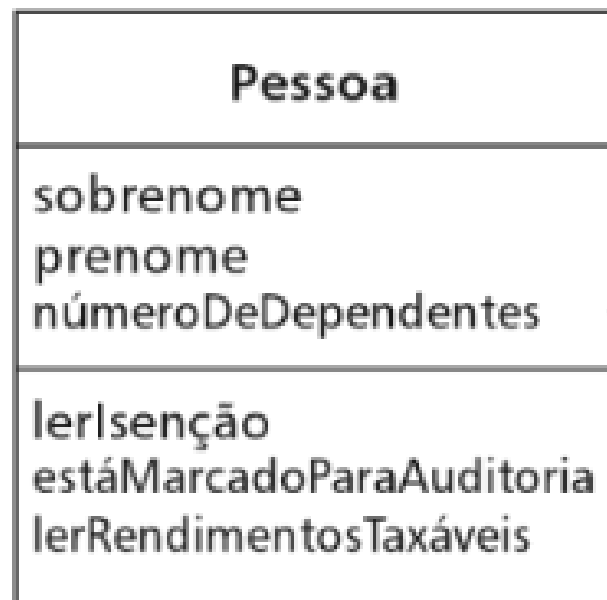
PRODUTO OBJETO

ID: 12
NOME: BICICLETA
PREÇO: R\$800
DESCRIÇÃO: ENGRENAGEM FIXA,
AZUL, RÁPIDA

ORM

TABELA: PRODUTO

ID	NOME	PREÇO	DESCRIÇÃO
12	BICICLETA	R\$800	ENGRENAGEM FIXA, AZUL, RÁPIDA
13	CAPACETE	R\$20,99	PRETO, AJUSTÁVEL
14	UNIFORME	R\$35	PEQUENO (FEMININO), VERDE E BRANCO



Entity Framework



Entity Framework

- Surgiu a partir do Nhibernate, a versão .Net do Hibernate criado para o Java
- Inicialmente, funcionava apenas para o SqlServer da Microsoft
- Melhorias comparado aos anteriores e em constante evolução

Abordagens



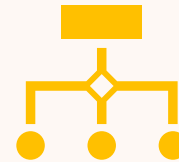
Code First

O programador cria as classes em um primeiro momento, e a partir delas, o Entity cria o banco de dados e as tabelas criadas



Database First

Com o banco de dados e suas tabelas prontas, o Entity faz a leitura e gera o código (classes) no projeto



Model First

Deve ser criado um diagrama (EDM) e a partir dele, o Entity gera o código (classes) e o banco de dados



Migrations

- Uma Migration é um modo de manter a base de dados e o projeto sincronizado, de maneira a preservar os dados armazenados.

Criando um projeto .Net

O Entity Framework possibilita criar projeto de qualquer tipo, desde web, WinForms, ClassLibrary, ConsoleApp e entre outros.

Após criar um projeto de preferencia (interessante começar com ConsoleApp), devemos adicionar as referencias necessárias para o EF funcionar

Para as referencias, utilizamos o NuGet para gerenciar esses pacotes

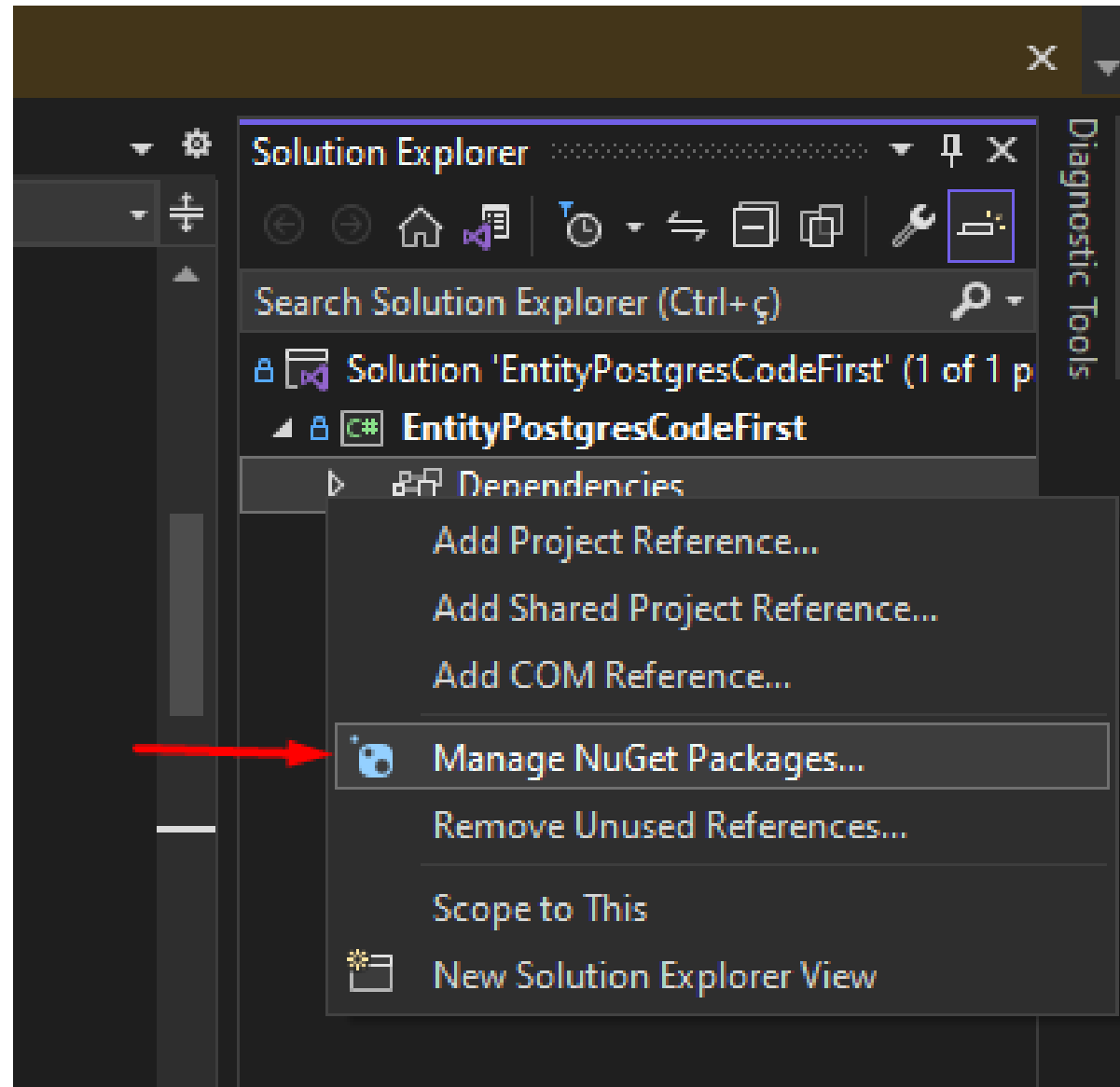
NuGet

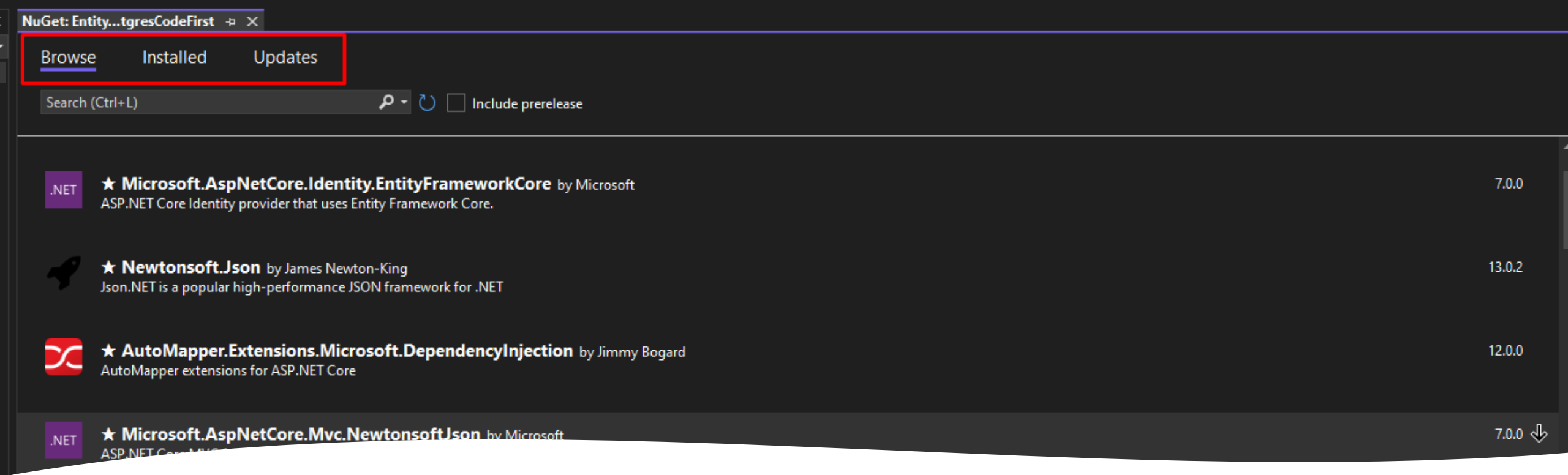
- É o gerenciador de pacotes da plataforma .Net
- Permite que desenvolvedores utilizem, criem e publiquem pacotes
- Os pacotes podem ser bibliotecas inteiras, que apresentem funcionalidades já implementadas e prontas para serem utilizadas
 - Desde ferramentas oficiais para conexão com base de dados, frameworks dos mais variados e ferramentas desenvolvidas de forma independente



Projeto

Com um projeto criado, clicamos com o direito do mouse e em Gerenciador de Pacotes NuGet










Dependências

- Com isso, existem 3 abas, a Browser, para buscar novos pacotes, Installed que lista os pacotes instalados e Updates que lista aqueles pacotes instalados que existem versões mais recentes

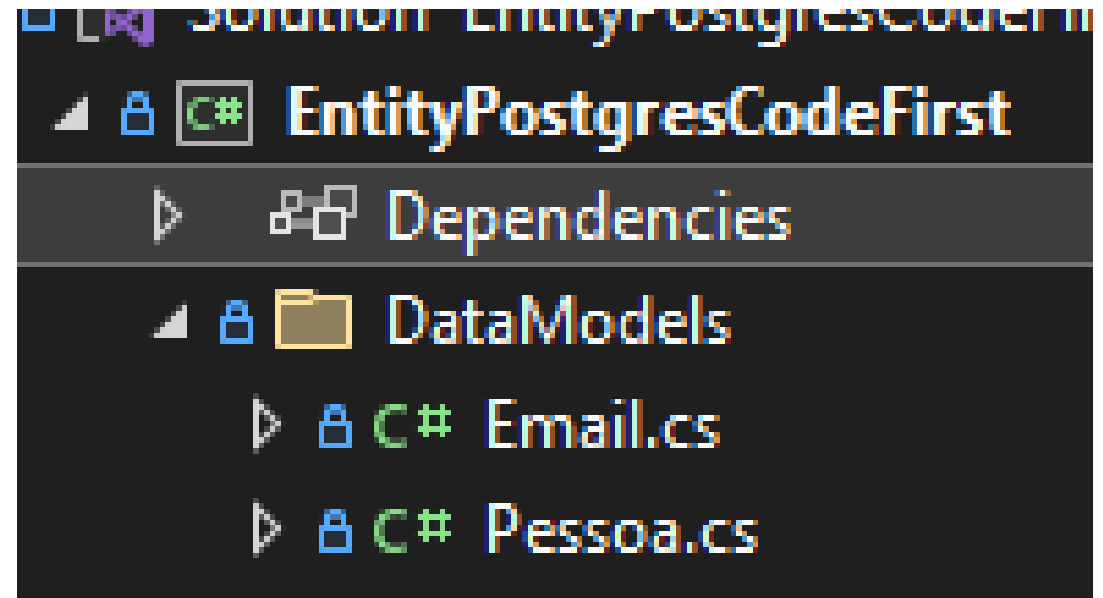
Dependências

Para esse primeiro projeto ConsoleApp, vamos utilizar os seguintes pacotes

	Microsoft.EntityFrameworkCore by Microsoft	7.0.0
✓	Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases through a provider plugin API.	
	Microsoft.EntityFrameworkCore.Proxies by Microsoft	7.0.0
✓	Lazy loading proxies for Entity Framework Core.	
	Microsoft.EntityFrameworkCore.Tools by Microsoft	7.0.0
✓	Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.	
	Npgsql.EntityFrameworkCore.PostgreSQL by Shay Rojansky, Austin Drenski, Yoh Deadfall	7.0.0
✓	PostgreSQL/Npgsql provider for Entity Framework Core.	
	System.Configuration.ConfigurationManager by Microsoft	7.0.0
✓	Provides types that support using configuration files.	

Projeto

- Vamos começar com a abordagem CodeFirst
- Para isso, vamos criar uma pasta chamada DataModels (para melhor organização dos códigos) e dentro dela, vamos criar a classe Email e a classe Pessoa



Classe Email

```
public class Email
{
    public int id { get; set; }

    public string email { get; set; }

    public virtual Pessoa pessoa { get; set; }
}
```

Classe Pessoa

11 references

```
public class Pessoa
```

```
{
```

0 references

```
public int id { get; set; }
```

5 references

```
public string nome { get; set; }
```

6 references

```
public virtual ICollection<Email> Emails { get; set; }
```

```
}
```

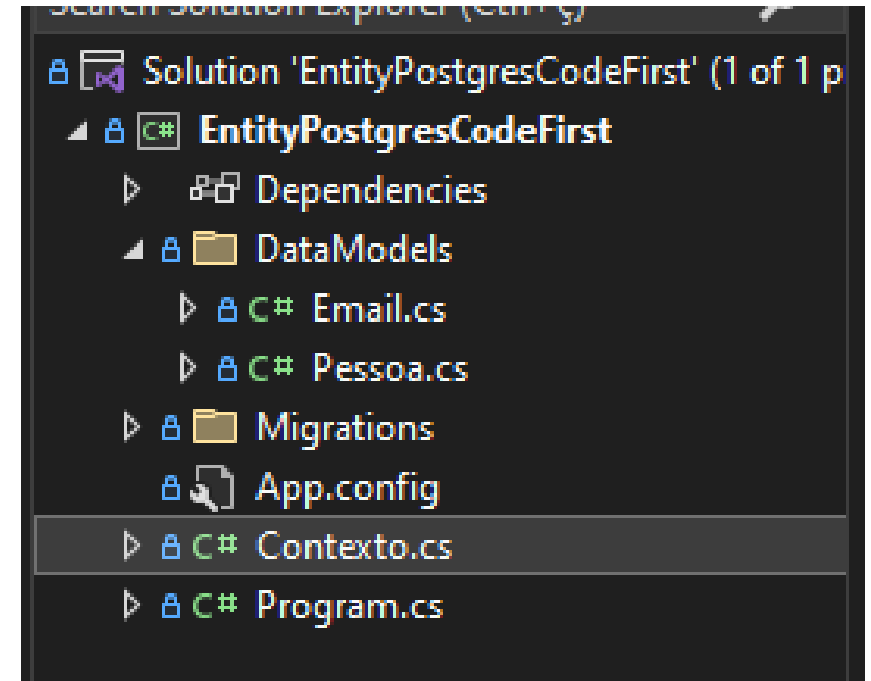
AppConfig

- Criamos também, um arquivo de configuração do App
- Nesse arquivo, vamos armazenar a string de conexão
 - Essa string contém os dados necessários para o nosso projeto saber conectar na nossa base de dados, pois armazena o endereço, nome do banco, usuário, senha e provider de conexão
 - Damos um nome para a conexão (tag) e configuramos ela conforme o exemplo
- Cada banco de dados possui uma sintaxe diferente para a string de conexão

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="EntityPostgresql"
      connectionString="server=localhost;Port=5432;user id=pos
      providerName="Npgsql" />
  </connectionStrings>
</configuration>
```

Contexto

- Toda conexão com o EF se dá por uma classe de contexto, então, na raiz do projeto, vamos criar uma classe chamada Contexto
- Essa classe herda da classe DbContext, existente dentro das bibliotecas do EF



```
5 references
public class Contexto : DbContext
{
}
```

Contexto

- Dentro da classe, devemos criar um DbSet para cada classe do nosso projeto que é um Mapper do banco de dados, pois nem toda classe necessariamente vai ser um espelho do banco de dados
- Também devemos deixar explícito o construtor

```
8 references
public DbSet<Pessoa> Pessoas { get; set; }
0 references
public DbSet<Email> Emails { get; set; }

1 reference
public Contexto()
{
    :
    :
}
```

Contexto

- Para configurar e criar o nosso contexto de conexão, fazemos um override do método OnConfiguring
- Aqui, configuramos a conexão do EF com a base de dados, com informações resgatadas do nosso AppConfig
- UseLazyLoadingProxies permite que uma classe que possui outra classe como atributo traga esses objeto preenchido automaticamente

O references

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    ConnectionStringSettings settings = ConfigurationManager.ConnectionStrings["EntityPostgresql"];
    string retorno = "";

    if (settings != null)
        retorno = settings.ConnectionString;

    optionsBuilder.UseNpgsql(retorno);

    optionsBuilder.UseLazyLoadingProxies();
}
```

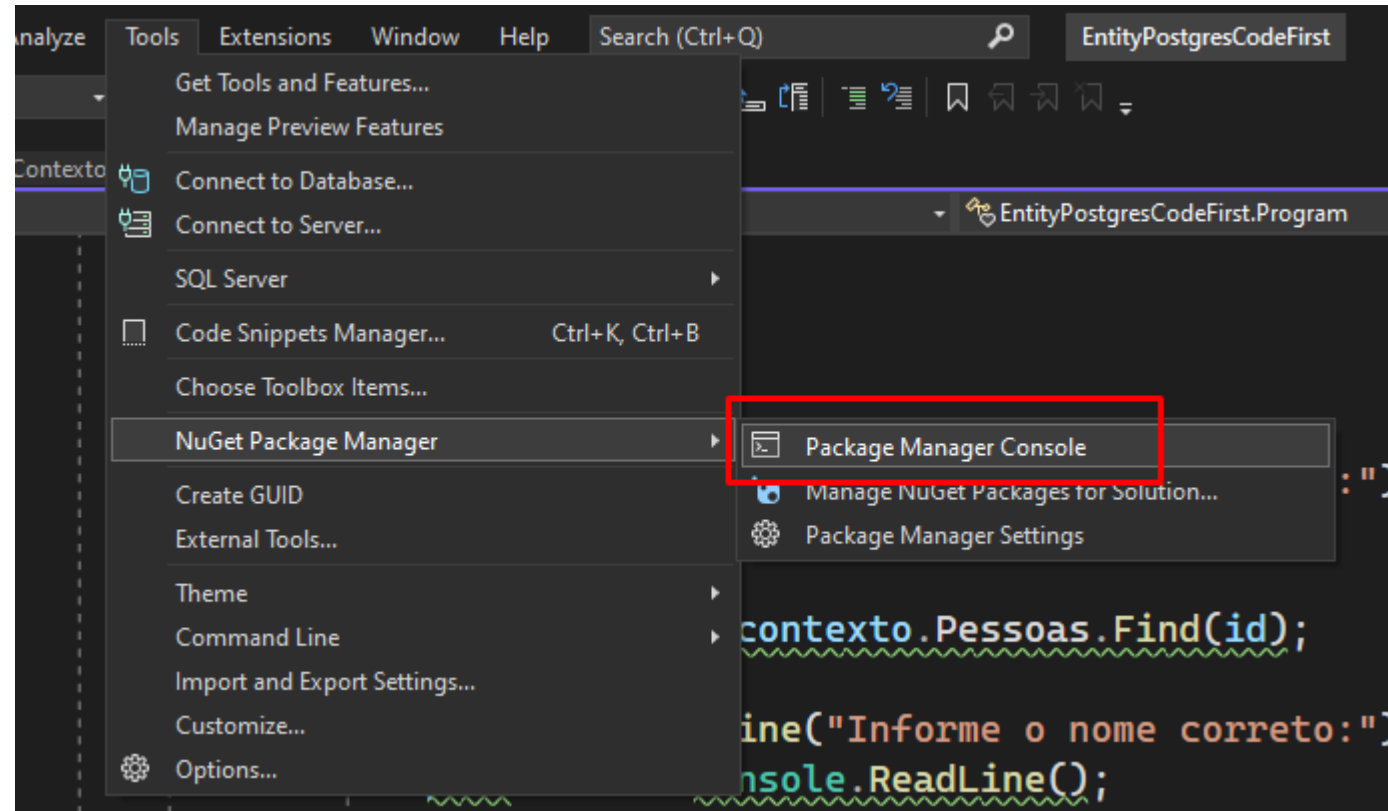
Contexto

- Por fim, fazemos o override do método `OnModelCreating`
- Aqui, informamos alguns detalhes sobre como as entidades vão se comportar
- Aqui, informamos que a entidade `Email` possui uma pessoa vinculada a ele e que essa pessoa pode possuir vários emails, além disso, deixamos o *delete cascade* ativado.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Email>()
        .HasOne(e => e.pessoa)
        .WithMany(e => e.Emails)
        .onDelete(DeleteBehavior.ClientCascade);
}
```

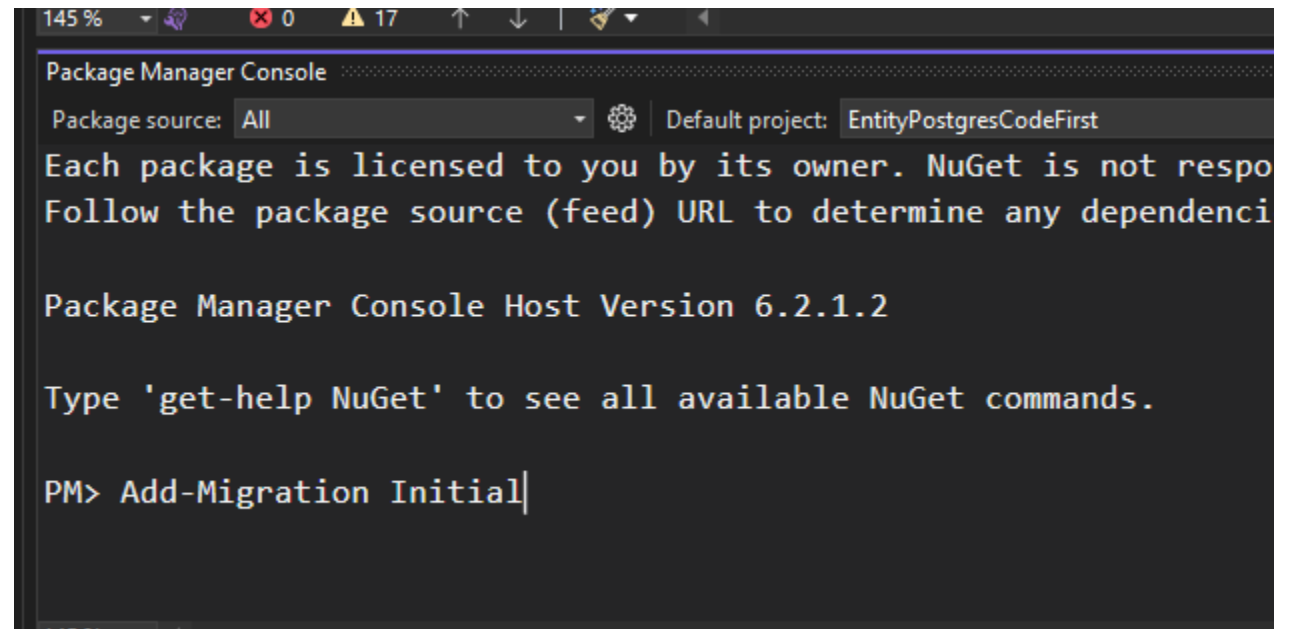
Migration

- Via linha de comandos, devemos sincronizar o nosso projeto com o banco de dados
- A linha de comandos deve ser executada no prompt de comandos do NuGet



Migration

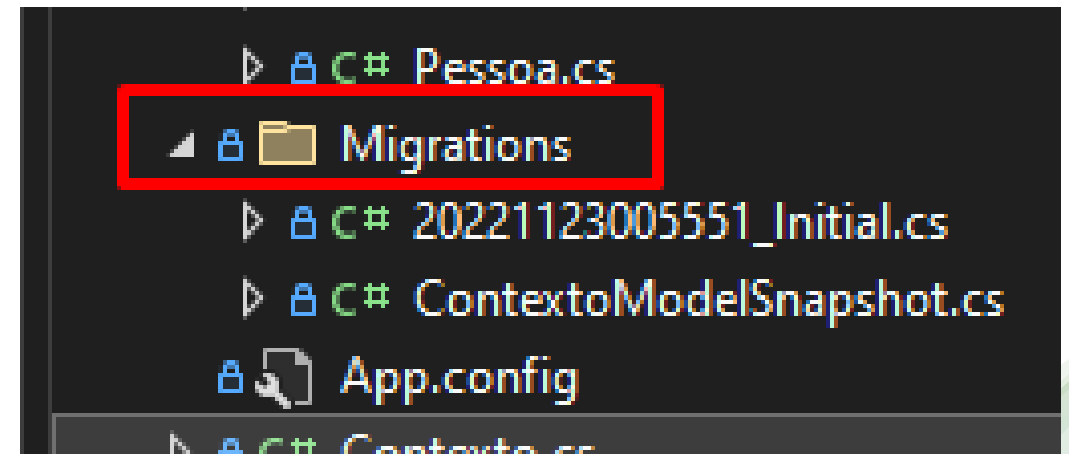
- Digitamos o comando Add-Migration seguido de um nome, conforme o exemplo



```
145 %  
Package Manager Console  
Package source: All Default project: EntityPostgresCodeFirst  
Each package is licensed to you by its owner. NuGet is not respo  
Follow the package source (feed) URL to determine any dependenci  
  
Package Manager Console Host Version 6.2.1.2  
  
Type 'get-help NuGet' to see all available NuGet commands.  
  
PM> Add-Migration Initial|
```

Migration

- Esse comando, cria uma pasta com os dados e comandos necessários para a criação do nosso banco de dados, também contem o arquivo responsável pelo versionamento da migration, para futuras alterações na base de dados.



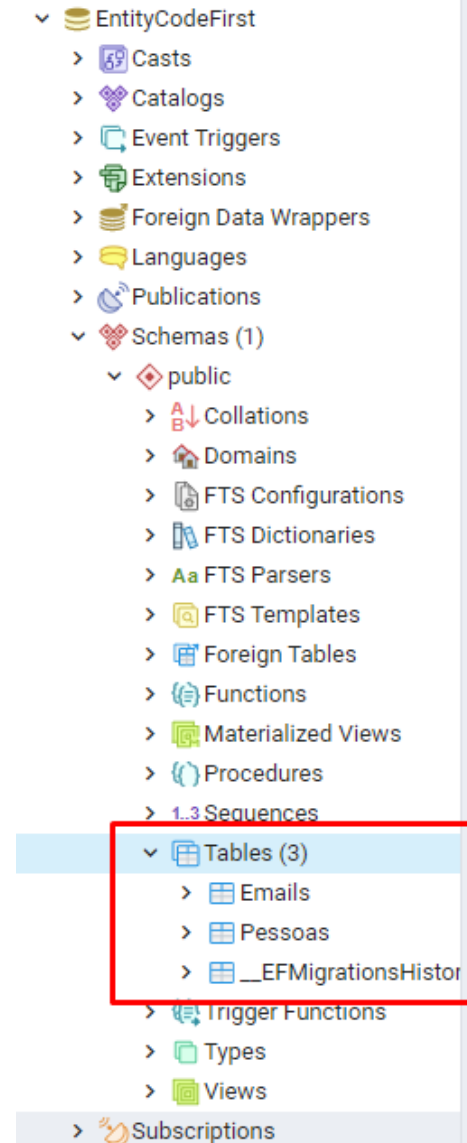
Migration

- Com a migration finalizada, devemos executar o comando Update-Database, que agora, de fato, vai sincronizar o banco de dados com o projeto (migration)

```
PM> Update-Database
```

Migration

- No PgAdmin, podemos ver que a base de dados com as nossas entidades foram criadas
- Além delas, uma tabela para o controle e versionamento da Migration



Program.cs

- Tudo pronto e configurado! Agora vamos começar a fazer o nosso CRUD
- Vamos montar um menu e capturar a escolha do usuário
- Criamos uma instância da classe Contexto

```
Console.WriteLine("Digite:\n" +  
    "1 para criar uma pessoa\n" +  
    "2 para alterar o nome da pessoa\n" +  
    "3 para inserir um email\n" +  
    "4 para excluir uma pessoa\n" +  
    "5 para consultar tudo\n" +  
    "6 para consultar pelo ID\n");  
  
int op = int.Parse(Console.ReadLine());  
  
Contexto contexto = new Contexto();
```

Main

Para cada um dos Cases do nosso Switch (para cada possibilidade), vamos por um try catch

Para todas as 6 possibilidades

```
switch (op)
{
    case 1:
        try
        {
```

```
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        break;
    case 2:
        try
        {
```

Case 1

- No Try o código ao lado, referente a inserção de uma nova pessoa, começa instanciando uma nova Pessoa, e preenchendo os seus atributos, inclusive criando um novo Email e vinculando a List<Email> da classe Pessoa
- Com isso, utilizamos o Contexto para, nas entidades pessoas, adicionarmos o objeto **p**, que contem todos os dados informados.
- Para finalizar, chamamos o método SaveChanges do contexto, que analisa todas as mudanças ocorridas no contexto e de fato, salva essas alterações na base de dados.

```
Console.WriteLine("Inserir o nome da pessoa:");
Pessoa p = new Pessoa();
p.nome = Console.ReadLine();

Console.WriteLine("Informe um email:");
string emailTemp = Console.ReadLine();

p.Emails = new List<Email>()
{
    new Email()
    {
        email = emailTemp
    }
};

contexto.Pessoas.Add(p);
contexto.SaveChanges();

Console.WriteLine("Pessoa inserida com sucesso!");
```

Case 2

- Para alteração de uma pessoa no banco, fazemos a busca (no caso, pelo ID) e utilizamos o método Find (uma das possibilidades de busca) para retornar a pessoa que contem o ID informado
- Com isso, alteramos os atributos o objeto retornado
- Informamos ao contexto as mudanças ocorridas (update) e chamamos o método SaveChanges para finalizar.

```
Console.WriteLine("Informe o ID da pessoa:");  
int id = int.Parse(Console.ReadLine());  
  
Pessoa pAlt = contexto.Pessoas.Find(id);  
  
Console.WriteLine("Informe o nome correto:");  
pAlt.nome = Console.ReadLine();  
  
contexto.Pessoas.Update(pAlt);  
contexto.SaveChanges();
```


Case 3

- Agora, vamos adicionar um novo email a uma pessoa já cadastrada
- Novamente, consultamos a pessoa para então adicionar o novo email na sua lista
- Repetimos o Update e o SaveChanges da Contexto

```
Console.WriteLine("Informe o ID da pessoa:");
int id = int.Parse(Console.ReadLine());
Pessoa p = contexto.Pessoas.Find(id);

Console.WriteLine("Informe o novo email:");
string emailTemp = Console.ReadLine();

p.Emails = new List<Email>()
{
    new Email()
    {
        email = emailTemp
    }
};

contexto.Pessoas.Update(p);
contexto.SaveChanges();
Console.WriteLine("Inserido com sucesso!");
```

Case 4

- Para remoção, consultamos a pessoa desejada, mostramos os dados retornados e solicitamos a confirmação da exclusão
- Com isso, chamamos o método Remove das Pessoas contidas no Contexto passando o objeto que desejamos remover do banco por parâmetro
- SaveChanges para finalizar

```
Console.WriteLine("Informar o ID da pessoa");
int id = int.Parse(Console.ReadLine());
Pessoa p = contexto.Pessoas.Find(id);

Console.WriteLine("Confirmar a exclusão de " + p.nome);
Console.WriteLine("E dos seus emails:");

foreach (Email item in p.Emails)
{
    Console.WriteLine("\t" + item.email);
}

Console.WriteLine("1 para SIM e outra tecla para NÃO");
if (int.Parse(Console.ReadLine()) == 1)
{
    contexto.Pessoas.Remove(p);
    contexto.SaveChanges();
    Console.WriteLine(p.nome + " excluída com sucesso!");
}
else
{
    return;
}
```

Case 5

- Nesse exemplo, vamos consultar todas as pessoas cadastradas, e mostrar também os emails vinculados a elas
- Começamos criando uma Lista de Pessoa que recebe o retorno de uma consulta dentro do nosso Contexto
 - Essa consulta faz uso da linguagem Linq e sua sintaxe, para retornar toda e qualquer pessoa e incluir os emails a ela vinculado, o retorno então é convertido para uma Lista de Pessoa
- Percorremos as pessoas retornadas e para cada uma delas, percorremos os emails para mostrar as informações para o usuário

```
List<Pessoa> lista = (from Pessoa p in contexto.Pessoas
                     select p).Include(pes => pes.Emails).ToList<Pessoa>();

foreach (Pessoa item in lista)
{
    Console.WriteLine(item.nome);

    foreach (Email itemE in item.Emails)
    {
        Console.WriteLine("\t" + itemE.email);
    }
    Console.WriteLine();
}
```

Case 6

- Aqui, vamos consultar apenas a pessoa solicitada
- Poderíamos usar o Find, mas apresentamos um outro modo de consulta, também utilizando Linq, que faz o filtro pelo ID e retorna apenas um objeto do tipo Pessoa
- Também mostra os emails vinculados a pessoa pesquisada

```
Console.WriteLine("Informe o id da pessoa");
int idPessoa = int.Parse(Console.ReadLine());
Pessoa pessoa = contexto.Pessoas
    .Include(p => p.Emails)
    .FirstOrDefault(x => x.Id == idPessoa);
Console.WriteLine(pessoa.Nome);

if (pessoa.Emails != null)
{
    foreach (Email item in pessoa.Emails)
    {
        Console.WriteLine("    " + item.Email1);
    }
}
```

Database First

- Com a base de dados criada, devemos criar um projeto e adicionar as dependências necessárias
 - As mesmas do projeto CodeFirst
- Feito isso, abrimos o prompt de comandos do NuGet e digitamos:
 - **Scaffold-DbContext “STRING DE CONEXÃO COMPLETA;”**
- Após isso, estando tudo Ok, pode ser solicitado para informar o provider (mecanismo de conexão), informe:
 - **Npgsql.EntityFrameworkCore.PostgreSQL**
- Com isso, o banco vai ser lido pelo Entity e as classes(models e contexto) serão criadas e estarão prontas para o uso

Exercício 1

- Crie um projeto para cadastrar e vincular Alunos e Cursos
- O vínculo entre alunos e cursos é a Matrícula (n x n)
- Faça utilizando CodeFirst e depois com Database First (criando o banco na mão)