**Entity Framework** é um ORM (Object-Relational Mapper) da Microsoft que permite aos desenvolvedores interagir com bancos de dados usando objetos .NET, o que significa que você pode trabalhar com um banco de dados usando a mesma linguagem que você usa para escrever suas aplicações.

Vamos ver o papel de cada pasta:

**1. Models**

Esta pasta contém classes que representam as tabelas do seu banco de dados. Por exemplo, se você tem uma tabela **Coupons**, você terá uma classe **Coupon** correspondente em sua aplicação. Cada instância dessa classe representa uma linha na tabela do banco de dados.

**2. Data**

Aqui, você geralmente encontrará a classe **DbContext**, que é uma parte central do EF. Esta classe gerencia as entidades durante o tempo de execução, o que inclui recuperar e salvar dados no banco de dados. Em projetos .NET, esta classe é frequentemente encontrada no arquivo **AppDbContext.cs**. A classe **DbContext** também define o conjunto de entidades (representadas pelas classes **DbSet**) que o EF conhece e pode trabalhar.

**3. Migrations**

Migrations são uma forma de gerenciar mudanças incrementais e controle de versão do seu modelo de banco de dados. Quando você faz alterações em suas classes de modelo (por exemplo, adiciona uma nova propriedade a uma classe), você precisa de uma forma de atualizar o banco de dados para refletir essas mudanças. As migrations permitem que você faça isso de uma maneira controlada e versátil. A pasta **Migrations** contém classes que o EF usa para atualizar o banco de dados. Cada arquivo de migration tem dois métodos principais:

* **Up()**: Define as alterações a serem aplicadas ao banco de dados (como adicionar uma nova tabela ou coluna).
* **Down()**: Define como desfazer as alterações feitas pelo método **Up()**.

Por exemplo, se você adicionar uma nova coluna a uma tabela, o método **Up()** conterá o código para adicionar essa coluna, e o método **Down()** conterá o código para removê-la, caso você decida voltar atrás na migration.

Exemplo de Migration:

Se você criar uma migration chamada **AddCouponToDb**, o EF gerará um código que descreve como criar a tabela **Coupons**. Se você executar essa migration, o método **Up()** será chamado e a tabela será criada. Se você decidir que não precisa dessa tabela e quiser revertê-la, poderá executar um comando para reverter a migration, e o EF chamará o método **Down()** para excluir a tabela.

Seed Data:

Além disso, você mencionou a classe **SeedCouponTables** em sua migration. Isso é usado para preencher as tabelas com dados iniciais, o que é útil para configurar um banco de dados com um estado inicial conhecido que pode incluir dados de teste ou configuração necessária para a aplicação rodar.

Em resumo, o Entity Framework é uma poderosa ferramenta que abstrai a interação com o banco de dados, permitindo que os desenvolvedores trabalhem em um nível mais alto de abstração com objetos e classes, ao invés de escrever SQL manualmente. As migrations são uma parte essencial deste sistema, gerenciando as alterações na estrutura do banco de dados de maneira controlada.

**DbContext**

Entity Framework permite interagir com banco de dados usando objetos .NET(trabalha com banco de dados usando C#).

DbContext é um componente do EntityFramework, ele gerencia perações de CRUD, conexão com o banco de dados e mapaeamento entre variáveis e tabela do banco de dados, sendo que esse faz essas operações “por baixo dos panos”.

No projeto deve existir uma classe que irá herdar O DbContext e suas funcionalidades, assim fazendo parte do EF, como aqui por exemplo:

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)

{

}

public DbSet<Coupon> Coupons { get; set; }

}

Para definir quais objetos serão mapeados para a tabelas no banco de dados é utilizado o DbSet, Sendo assim, ao criar um DbSet você aponta para o nome da classe que será mapeada essa tabela, no caso Coupon que estaria dentro da model( por isso é usado using Mango.Services.CouponAPI.Models; no inicio).  
  
 public DbSet<Coupon> Coupons { get; set; }

Assim, o EF irá utilizar a classe Coupon.cs para a tabela Coupons do banco de dados, e cada variável dentro de Coupon.cs se tornará uma coluna:

public class Coupon

{

[Key]

public int CouponId { get; set; }

[Required]

public string CouponCode { get; set; }

[Required]

public double DiscountAmount { get; set; }

public int MinAmount { get; set; }

}

caso não exista essa tabela, ela seria criada. Após isso, é possível fazer operações CRUD com essa tabela Coupons através do AppDbContext.

**Controller**

Então, para fazer estas operações dentro da controller, é necessário chamar:  
   
using Mango.Services.CouponAPI.Data;

Sendo a pasta Data a que possui a classe AppDbContext.cs.

É necessário injetar a instancia do AppDbContext para acessar e manipular o banco de dados:  
  
 public class CouponAPIController : ControllerBase

{

private readonly AppDbContext \_db;

public CouponAPIController(AppDbContext db)

{

\_db = db;

}

// Métodos da API que utilizam \_db para operações de banco de dados

}  
  
 Aqui, o AppDbContext é injetado e é armazenado em uma variável chamada \_db, está variável é usada para interagir com o banco de dados através das API’s.

**Obs:**

O caminho de conexão ao banco de dados é feita em appsettings.json, sendo a string de conexão a “configuração” desta conexão.

Para configurar a injeção de dependência do AppDbContext na aplicação .NET é necessário em Program.cs:  
  
builder.Services.AddDbContext<AppDbContext>(options =>

{

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));

});  
  
Assim, adicionando AppDbContext ao sistema de injeção de dependência junto ao SQL Server, você permite que que a aplicação possa ser usada para a conexão com o banco de dados(no caso controllers).

**DTO**

**DTOs são usados para transferir dados de forma segura e eficiente.**

**Eles não incluem todos os dados de uma entidade, apenas os necessários para a operação específica.**

**Eles ajudam a proteger as entidades do banco de dados, não expondo informações sensíveis ou desnecessárias.**

**Utilizam menos recursos, pois simplificam os dados transferidos.**

**Recebimento de Dados Externos (Entrada):**

1. **Dados Chegam como DTOs: Quando a API recebe dados de uma fonte externa (como um cliente de frontend), esses dados geralmente chegam no formato de um DTO.**
2. **Transformação para Entidades do Modelo: Antes de salvar esses dados no banco de dados, eles são transformados de DTOs para as entidades do modelo(variáveis normais). Este processo envolve mapear os dados do DTO para a estrutura das entidades que representam suas tabelas e colunas no banco de dados, que no caso o mapeamento é feito pelo MappingConfig**
3. **Salvamento no Banco de Dados: Após a transformação, os dados agora na forma de entidades do modelo são salvos no banco de dados.**

**Envio de Dados para Fora (Saída):**

1. **Recuperação de Dados do Banco de Dados: Quando a API precisa enviar dados para fora, ela primeiro recupera os dados necessários do banco de dados, onde estão armazenados como entidades do modelo(variáveis normais).**
2. **Transformação para DTOs: Esses dados são então transformados em DTOs. Este passo é crucial para incluir apenas os dados necessários e relevantes para o destinatário, além de excluir informações sensíveis ou irrelevantes.**
3. **Envio dos DTOs: Finalmente, os DTOs são enviados para o destinatário externo, como uma resposta de uma solicitação API.**

**Essa abordagem com DTOs oferece várias vantagens: segurança de dados, redução de carga de dados transmitidos, e uma clara separação entre a lógica de negócios interna e a interface de comunicação externa. Isso torna as APIs mais seguras, eficientes e fáceis de manter.**

[HttpPost]

public ResponseDto Post([FromBody] CouponDto couponDto)

{

try

{

Coupon obj = \_mapper.Map<Coupon>(couponDto);

\_db.Coupons.Add(obj);

\_db.SaveChanges();

\_response.Result = \_mapper.Map<CouponDto>(obj);

}

catch (Exception ex)

{

\_response.IsSucess = false;

\_response.Message = ex.Message;

}

return \_response;

}

API Post chamada ResponseDto irá receber os dados externos.

O FromBody irá desserializar o json recebido externo para um objeto da classe CouponDto chamado de couponDto.

**Post([FromBody] CouponDto couponDto)**

entrará aqui: um Objeto da classe Coupon chamado de obj receberá as variáveis transformadas da classe DTO para o modelo de sua classe(<Coupon>) devido ao Mappingconfig.cs, como está em:

**Coupon obj = \_mapper.Map<Coupon>(couponDto);**

O entity framework com base no DbContext irá salvar na tabela Coupons como informado em \_db.Coupons os novos itens que estão na variável obj(objeto de Coupon.cs)

**\_db.Coupons.Add(obj);**

E para persistir(salvar) no BD, será em:

**\_db.SaveChanges();**

Lembrando que é possível fazer isso pois foi criado um DbSet para Coupon(tabela Coupons).