Modularização de Projetos .NET: Migrações do Entity Framework Core, Publicação e `dotnet run`

1. Migrações do Entity Framework Core em Projetos Modularizados

Quando você modulariza seu projeto .NET, separando o `DbContext` e as entidades em projetos de biblioteca de classes distintos (como `API.Control.Infrastructure` para o `DbContext` e `API.Control.Core` para as entidades), o processo de migração do Entity Framework Core (EF Core) requer uma abordagem ligeiramente diferente, mas ainda é bastante direto. O EF Core precisa saber onde encontrar o `DbContext` e onde as migrações devem ser geradas e aplicadas.

1.1. Onde o EF Core Procura o `DbContext` e as Migrações

Por padrão, os comandos da CLI do EF Core (como `dotnet ef migrations add` e `dotnet ef database update`) procuram o `DbContext` no projeto de inicialização (startup project) da sua solução. No seu caso, o projeto de inicialização é o `API.Control` (ou `API.Control.Web` / `API.Control.Api` após a refatoração), que é o projeto que contém o `Program.cs` e onde a aplicação é executada.

No entanto, o `DbContext` em si (o `AppDbContext.cs`) e as migrações geradas residirão no projeto `API.Control.Infrastructure`. Isso significa que você precisa instruir o EF Core a usar o projeto `API.Control.Infrastructure` como o projeto de migrações (migrations project) e o projeto `API.Control` como o projeto de inicialização.

1.2. Comandos da CLI do EF Core para Projetos Modularizados

Para adicionar uma nova migração ou atualizar o banco de dados em um projeto modularizado, você usará os seguintes comandos da CLI do .NET, especificando o

projeto de inicialização (`--startup-project`) e o projeto de migrações (`--project`):

Adicionar uma Nova Migração:

Para adicionar uma nova migração, você deve estar no diretório raiz da sua solução (onde o arquivo `.sln` está localizado) ou em qualquer subdiretório. O comando seria:

```
dotnet ef migrations add [NomeDaMigracao] --startup-project API.Control --
project API.Control.Infrastructure
```

- `[NomeDaMigracao]` : Um nome descritivo para a sua migração (e.g., `AddUsersTable` , `UpdateProductSchema`).
- `--startup-project API.Control` : Indica ao EF Core que o projeto `API.Control` é o projeto de inicialização, que contém as configurações da aplicação e a injeção de dependência do `DbContext` .
- `--project API.Control.Infrastructure` : Indica ao EF Core que o projeto `API.Control.Infrastructure` é onde o `DbContext` está localizado e onde os arquivos de migração devem ser gerados.

Ao executar este comando, o EF Core:

1. Identificará o `DbContext` no projeto `API.Control.Infrastructure` .
2. Analisará as mudanças no seu modelo de dados (entidades) em relação ao estado atual do banco de dados (conforme registrado nas migrações existentes).
3. Gerará os arquivos de migração (uma classe C# com os métodos `Up` e `Down`) dentro da pasta `Migrations` do projeto `API.Control.Infrastructure` .

Atualizar o Banco de Dados:

Para aplicar as migrações pendentes ao seu banco de dados, o comando é similar:

```
dotnet ef database update --startup-project API.Control --project
API.Control.Infrastructure
```

- `--startup-project API.Control` : Novamente, especifica o projeto de inicialização.
- `--project API.Control.Infrastructure` : Indica onde as migrações a serem aplicadas estão localizadas.

Este comando fará com que o EF Core:

1. Leia as migrações existentes no projeto `API.Control.Infrastructure`.
2. Compare-as com o estado atual do banco de dados.
3. Execute as migrações pendentes para atualizar o esquema do banco de dados.

1.3. Considerações Importantes

- **Referências de Projeto:** Certifique-se de que o projeto de inicialização (`API.Control`) tenha uma referência ao projeto de infraestrutura (`API.Control.Infrastructure`). Isso é fundamental para que o EF Core possa encontrar o `DbContext` e as migrações.
- **Strings de Conexão:** A string de conexão do banco de dados é geralmente configurada no projeto de inicialização (e.g., `appsettings.json` no `API.Control`). O EF Core usará essa string de conexão ao executar os comandos de migração.
- **Contexto de Design-Time:** O EF Core usa um

contexto de design-time para criar instâncias do seu `DbContext` durante a execução dos comandos. Se você tiver lógica complexa no construtor do seu `DbContext` ou na configuração dos serviços, pode ser necessário implementar uma `IDesignTimeDbContextFactory<TContext>` no seu projeto de infraestrutura para auxiliar o EF Core a instanciar o `DbContext` corretamente em tempo de design.

Em resumo, a modularização não impede as migrações do EF Core; apenas exige que você seja explícito sobre qual projeto contém o `DbContext` e as migrações, e qual é o projeto de inicialização que fornece o contexto de execução.

2. Publicação de Projetos Modularizados

Publicar um projeto .NET modularizado é, na maioria dos casos, tão simples quanto publicar um projeto monolítico. O comando `dotnet publish` é inteligente o suficiente para identificar as dependências entre os projetos e incluir todos os assemblies necessários na saída da publicação.

2.1. O Processo de Publicação

Quando você executa o comando `dotnet publish` no seu projeto principal (o projeto `API.Control` ou `API.Control.Web` / `API.Control.Api`):

```
dotnet publish API.Control -c Release -o ./publish
```

- `API.Control`: É o nome do seu projeto principal (o arquivo `.csproj` da sua API).
- `-c Release`: Especifica a configuração de build (Release é a mais comum para publicação).
- `-o ./publish`: Define o diretório de saída onde os arquivos publicados serão colocados.

O SDK do .NET fará o seguinte:

1. **Compilar Todos os Projetos:** Ele compilará todos os projetos na sua solução, respeitando as dependências entre eles. Isso significa que `API.Control.Core` será compilado primeiro, seguido por `API.Control.Infrastructure`, `API.Control.Services` e, finalmente, `API.Control`.
2. **Coletar Dependências:** O processo de publicação coletará todos os assemblies (arquivos `.dll`) gerados por cada um dos seus projetos de biblioteca de classes (`API.Control.Core.dll`, `API.Control.Infrastructure.dll`, `API.Control.Services.dll`, etc.) e os colocará no diretório de saída.
3. **Incluir Dependências Externas:** Todas as dependências NuGet que seus projetos utilizam também serão incluídas no diretório de publicação.
4. **Gerar Executável:** Um executável (e.g., `API.Control.exe` no Windows) será gerado, que é o ponto de entrada da sua aplicação.

O resultado final no diretório `./publish` será um conjunto completo de arquivos que podem ser implantados em um servidor. Você não precisa publicar cada projeto de biblioteca de classes individualmente, pois o projeto principal já

incluirá todas as dependências necessárias.

2.2. Publicação para Ambientes Específicos

Você pode especificar o runtime de destino para a publicação, o que é útil para criar executáveis auto-contidos ou para plataformas específicas:

```
dotnet publish API.Control -c Release -r win-x64 --self-contained true -o
./publish-win-x64
```

- `-r win-x64` : Publica para o runtime Windows de 64 bits.
- `--self-contained true` : Cria um executável auto-contido que inclui o runtime do .NET, eliminando a necessidade de ter o .NET instalado no servidor de destino.

Em resumo, a publicação de projetos modularizados é gerenciada de forma eficiente pelo `dotnet publish`, que se encarrega de coletar e empacotar todas as partes da sua aplicação em um único diretório de implantação.

3. Esclarecendo o Uso do `dotnet run` em Projetos Modularizados

A pergunta "tenho que rodar dotnet run em cada projeto separado?" é muito comum ao se deparar com projetos modularizados. A resposta curta é: **não, você não precisa rodar `dotnet run` em cada projeto separado.**

3.1. Como o `dotnet run` Funciona

O comando `dotnet run` é projetado para executar o projeto de inicialização da sua aplicação. No contexto de uma API ASP.NET Core, o projeto de inicialização é aquele que contém o arquivo `Program.cs` e que define o ponto de entrada da aplicação (onde o `WebApplication.CreateBuilder` e `app.Run()` são chamados). No seu caso, este é o projeto `API.Control` (ou `API.Control.Web` / `API.Control.Api`).

Quando você executa `dotnet run` a partir do diretório do seu projeto principal (ou especificando-o):

```
dotnet run --project API.Control
```

O SDK do .NET fará o seguinte:

1. **Restauração de Pacotes:** Ele primeiro garantirá que todos os pacotes NuGet e referências de projeto estejam restaurados.
2. **Compilação:** Em seguida, ele compilará o projeto especificado (`API.Control`) e todas as suas dependências (ou seja, `API.Control.Core` , `API.Control.Infrastructure` , `API.Control.Services`).
3. **Execução:** Finalmente, ele executará o assembly de saída do projeto principal. Durante a execução, o runtime do .NET carregará automaticamente os assemblies das bibliotecas de classes referenciadas, pois eles estarão no mesmo diretório de saída ou em subdiretórios conhecidos.

3.2. Por que Não Rodar em Cada Projeto Separado?

- **Projetos de Biblioteca de Classes Não São Executáveis:** Projetos de biblioteca de classes (`.csproj` com `Sdk=

"Microsoft.NET.Sdk" em vez de "Microsoft.NET.Sdk.Web" ou "Microsoft.NET.Sdk.Worker") não são aplicações executáveis por si só. Eles contêm código que é consumido por outros projetos (geralmente um projeto executável, como sua API). * **Dependências Resolvidas Automaticamente:** O sistema de build do .NET e o runtime são projetados para resolver automaticamente as dependências entre projetos. Quando seu projeto principal é executado, ele sabe onde encontrar os assemblies dos projetos de biblioteca de classes que ele referencia.

3.3. Cenários Onde Você Pode Interagir com Projetos de Biblioteca

Embora você não execute `dotnet run` em projetos de biblioteca, você pode interagir com eles de outras maneiras:

- **Testes Unitários:** Você pode ter projetos de teste separados (e.g., `API.Control.Services.Tests`) que dependem do seu projeto `API.Control.Services` e que você executa usando `dotnet test` .
- **Ferramentas da CLI:** Como visto nas migrações do EF Core, você pode usar o `--project` para direcionar comandos da CLI para um projeto de biblioteca específico, mas isso não é o mesmo que

executar o projeto.

Em resumo, para desenvolver e testar sua API modularizada, você continuará usando `dotnet run` no seu projeto principal (`API.Control`). As dependências serão resolvidas automaticamente, e você não precisará executar comandos separados para cada biblioteca de classes.

Conclusão

A modularização de projetos .NET, embora adicione uma camada inicial de complexidade na organização, oferece benefícios substanciais em termos de manutenibilidade, escalabilidade e reusabilidade do código. As migrações do Entity Framework Core e o processo de publicação são bem suportados nesse modelo, exigindo apenas uma compreensão clara de como os comandos da CLI do .NET interagem com os diferentes projetos da sua solução.

Ao seguir as práticas recomendadas para a estrutura de projetos, gerenciamento de dependências e uso correto das ferramentas da CLI, você poderá construir aplicações robustas e organizadas, capazes de evoluir e se adaptar às necessidades futuras. Lembre-se que o `dotnet run` é para o projeto executável, e as bibliotecas de classes são compiladas e incluídas como dependências, não executadas diretamente.