Guia Detalhado de Modularização para o Projeto API.Control

1. Introdução à Modularização

A modularização de um projeto de software é uma prática essencial para garantir a manutenibilidade, escalabilidade e reusabilidade do código, especialmente em aplicações de médio a grande porte. Ao dividir um projeto monolítico em módulos menores e coesos, cada um com sua responsabilidade bem definida, é possível gerenciar a complexidade, facilitar o desenvolvimento paralelo por equipes diferentes e isolar falhas, tornando o sistema mais robusto.

No contexto do seu projeto API.Control, que visa o gerenciamento de endpoints Windows, a modularização pode trazer benefícios significativos. Atualmente, o projeto é um único assembly (`API.Control.csproj`), o que pode dificultar a manutenção e a evolução à medida que novas funcionalidades são adicionadas. Ao separar as camadas em projetos de biblioteca de classes distintos, você criará uma arquitetura mais limpa e flexível.

Este guia detalhará o processo de modularização do seu projeto API.Control, propondo uma estrutura de módulos e fornecendo um passo a passo para a refatoração. O objetivo é transformar o projeto em uma solução mais organizada, onde cada componente tem um propósito claro e as dependências são bem controladas.

2. Estrutura Modular Proposta

Para modularizar o projeto API.Control, propomos a criação de múltiplos projetos de biblioteca de classes (.NET Standard ou .NET Core) dentro da mesma solução. Cada projeto terá uma responsabilidade específica, promovendo a separação de preocupações e facilitando a manutenção. A estrutura proposta é a seguinte:

2.1. API.Control.Core

Este projeto será a base da sua aplicação, contendo as definições de entidades, interfaces e objetos de valor que são compartilhados por todas as outras camadas. Ele deve ser o

módulo com o menor número de dependências externas, garantindo que as definições de domínio sejam limpas e reutilizáveis.

Conteúdo:

- **Entidades (Entities):** Classes que representam os objetos de negócio do seu domínio (e.g., `Device`, `Application`, `DeviceModel`). Estas classes devem ser POCOs (Plain Old CLR Objects) e não devem conter lógica de negócio complexa ou dependências de infraestrutura.
- **Interfaces de Serviço (Interfaces):** Definições das interfaces para os serviços de negócio (e.g., `IDeviceService`, `IApplicationService`). As interfaces definem os contratos que os serviços devem implementar, permitindo a injeção de dependência e o desacoplamento.
- **DTOs (Data Transfer Objects):** Classes utilizadas para transferir dados entre as camadas da aplicação, especialmente entre a API e os serviços. Elas representam a forma como os dados são expostos ou recebidos pelos endpoints (e.g., `DeviceReadDTO`, `DeviceCreateDTO`).
- **Objetos de Valor (Value Objects):** Classes que representam conceitos de domínio que não possuem identidade própria e são definidos por seus atributos (e.g., `MacAddress`, `ComputerName`).
- **Enumerações (Enums):** Quaisquer enumerações utilizadas no domínio da aplicação.
- **Validações Comuns (Common Validations):** Regras de validação que são aplicáveis a entidades ou DTOs e que podem ser reutilizadas em diferentes contextos (e.g., validadores FluentValidation para objetos de valor).

Dependências: Nenhuma ou apenas bibliotecas de propósito geral (e.g., `System.ComponentModel.DataAnnotations`).

2.2. API.Control.Infrastructure

Este projeto será responsável por toda a lógica de persistência de dados e outras preocupações de infraestrutura. Ele implementará as interfaces definidas no

`API.Control.Core` e será o único módulo com dependência direta do Entity Framework Core e do banco de dados.

Conteúdo:

- **Contexto de Banco de Dados (DbContext):** A classe `AppDbContext` e suas configurações de mapeamento para o banco de dados (`OnModelCreating`).
- **Migrações (Migrations):** As classes de migração do Entity Framework Core que gerenciam o esquema do banco de dados.
- **Repositórios (Repositories):** Implementações concretas de padrões de repositório, se você optar por utilizá-los, para abstrair as operações de acesso a dados. No entanto, com o Entity Framework Core, muitas vezes os serviços podem interagir diretamente com o `DbContext` .
- **Configurações de Persistência:** Qualquer configuração específica relacionada ao provedor de banco de dados (e.g., SQLite, SQL Server).
- **Implementações de Serviços de Infraestrutura:** Serviços que interagem com sistemas externos, como serviços de e-mail, sistemas de arquivos, etc.

Dependências: `API.Control.Core` , `Microsoft.EntityFrameworkCore` e seus provedores (e.g., `Microsoft.EntityFrameworkCore.Sqlite`).

2.3. API.Control.Services

Este projeto conterá as implementações concretas das interfaces de serviço definidas no `API.Control.Core` . Ele encapsulará a lógica de negócio principal da aplicação, orquestrando as operações e interagindo com a camada de infraestrutura.

Conteúdo:

- **Implementações de Serviço (Service Implementations):** Classes como `DeviceService` , `ApplicationService` , etc., que contêm a lógica de negócio para cada domínio. Elas devem depender das interfaces de repositório (se usadas) ou diretamente do `AppDbContext` (via `API.Control.Infrastructure`).

- **Mapeamentos (Mappings):** Perfis do AutoMapper que definem como os DTOs são mapeados para as entidades e vice-versa. Embora possam estar no `Core` se forem muito genéricos, é comum mantê-los próximos às implementações de serviço que os utilizam.

Dependências: `API.Control.Core`, `API.Control.Infrastructure`, `AutoMapper`.

2.4. API.Control.Web (ou API.Control.Api)

Este será o projeto principal da sua aplicação ASP.NET Core, responsável por expor os endpoints da API, configurar a injeção de dependência, o middleware e o pipeline de requisições. Ele atuará como a camada de apresentação da sua API.

Conteúdo:

- **Endpoints (Endpoints):** As classes que definem os endpoints da API (e.g., `DeviceEndpoints`, `ApplicationEndpoints`).
- **`Program.cs`:** O arquivo principal de configuração da aplicação, onde os serviços são registrados, o pipeline de requisições é configurado e o host da aplicação é construído.
- **`appsettings.json`:** Arquivos de configuração da aplicação.
- **Configuração de Autenticação e Autorização:** Toda a lógica de configuração do JWT Bearer e das políticas de autorização.
- **Configuração do Swagger/OpenAPI:** A configuração para a geração da documentação da API.

Dependências: `API.Control.Core`, `API.Control.Services`, `Microsoft.AspNetCore.App` (ou pacotes específicos do ASP.NET Core), `Swashbuckle.AspNetCore`.

2.5. API.Control.Domain (Opcional, para Domínios Ricos)

Para projetos que adotam uma abordagem de Domain-Driven Design (DDD) mais rigorosa, um projeto `API.Control.Domain` pode ser criado para conter a lógica de negócio mais complexa, agregados, entidades com comportamento e especificações. Neste caso,

`API.Control.Core` conteria apenas as interfaces e DTOs mais básicos, enquanto `API.Control.Domain` conteria as entidades com comportamento e as regras de negócio.

Conteúdo:

- **Entidades com Comportamento:** Entidades que encapsulam lógica de negócio complexa e garantem a consistência do domínio.
- **Agregados (Aggregates):** Agrupamentos de entidades que são tratados como uma única unidade transacional.
- **Serviços de Domínio (Domain Services):** Lógica de negócio que não pertence a uma única entidade ou agregado.
- **Especificações (Specifications):** Padrões para encapsular regras de negócio e critérios de consulta.

Dependências: Nenhuma ou apenas bibliotecas de propósito geral.

Observação: Para o seu projeto atual, a estrutura com `Core`, `Infrastructure`, `Services` e `Web` já seria um grande avanço. A criação de um `Domain` separado é mais indicada para projetos com um domínio de negócio muito complexo e que se beneficiariam de uma modelagem mais rica.

3. Passo a Passo para a Refatoração

Agora que temos uma estrutura modular proposta, vamos detalhar o passo a passo para refatorar o projeto `API.Control` existente em múltiplos projetos. Este processo envolve a criação de novos projetos, a movimentação de arquivos, o ajuste de referências e a atualização de namespaces.

3.1. Preparação

Antes de iniciar a refatoração, é crucial garantir que você tenha um backup do seu projeto atual e que esteja utilizando um sistema de controle de versão (como Git). Isso permitirá que você reverta as alterações caso algo dê errado e facilitará a colaboração.

1. **Crie um Branch:** No seu sistema de controle de versão, crie um novo branch para esta refatoração (e.g., `feature/modularizacao`).
2. **Verifique o Estado Atual:** Certifique-se de que seu projeto atual está compilando e todos os testes (se houver) estão passando.

3.2. Criação dos Novos Projetos

Vamos começar criando os novos projetos de biblioteca de classes dentro da sua solução existente. Você pode fazer isso usando o Visual Studio ou a CLI do .NET.

Usando a CLI do .NET:

Abra o terminal na raiz da sua solução (`API.Control.sln`) e execute os seguintes comandos:

Bash

```
dotnet new classlib -n API.Control.Core -o API.Control.Core
dotnet new classlib -n API.Control.Infrastructure -o
API.Control.Infrastructure
dotnet new classlib -n API.Control.Services -o API.Control.Services

# Adicione os novos projetos à sua solução
dotnet sln add API.Control.Core/API.Control.Core.csproj
dotnet sln add API.Control.Infrastructure/API.Control.Infrastructure.csproj
dotnet sln add API.Control.Services/API.Control.Services.csproj
```

Após a execução desses comandos, você terá três novos diretórios e projetos na sua solução.

3.3. Movimentação de Arquivos para API.Control.Core

Este é o primeiro e mais importante passo. Mova os arquivos que pertencem à camada de Core para o novo projeto `API.Control.Core` .

1. **Entidades:** Mova todos os arquivos da pasta `Entities` (e suas subpastas, como `Auxiliary`) do projeto `API.Control` para a pasta `API.Control.Core/Entities` .
2. **Interfaces de Serviço:** Mova todos os arquivos da pasta `Services/Interfaces` do projeto `API.Control` para a pasta `API.Control.Core/Services/Interfaces` .

3. **DTOs:** Mova todos os arquivos da pasta `DTOs` do projeto `API.Control` para a pasta `API.Control.Core/DTOs`.
4. **Value Objects:** Mova todos os arquivos da pasta `ValueObjects` do projeto `API.Control` para a pasta `API.Control.Core/ValueObjects`.
5. **Validações Comuns:** Mova os validadores que são genéricos e não dependem de infraestrutura (e.g., `MacAddress_Validator.cs`) da pasta `Validators` para `API.Control.Core/Validators`.

Importante: Após mover os arquivos, você precisará ajustar os namespaces em cada arquivo para refletir a nova estrutura. Por exemplo, um arquivo que estava em `API.Control.Entities` agora deve ser `API.Control.Core.Entities`.

3.4. Movimentação de Arquivos para API.Control.Infrastructure

Agora, mova os arquivos relacionados à infraestrutura para o projeto `API.Control.Infrastructure`.

1. **Contexto de Banco de Dados:** Mova o arquivo `AppDbContext.cs` da pasta `Data` do projeto `API.Control` para a pasta `API.Control.Infrastructure/Data`.
2. **Migrações:** Mova a pasta `Migrations` completa do projeto `API.Control` para `API.Control.Infrastructure/Migrations`.
3. **DbInitializer:** Mova o arquivo `DbInitializer.cs` da pasta `Helpers` para `API.Control.Infrastructure/Helpers`.

Ajuste de Namespaces: Lembre-se de atualizar os namespaces nos arquivos movidos (e.g., `API.Control.Data` para `API.Control.Infrastructure.Data`).

3.5. Movimentação de Arquivos para API.Control.Services

Em seguida, mova as implementações dos serviços de negócio para o projeto `API.Control.Services`.

1. **Implementações de Serviço:** Mova todos os arquivos da pasta `Services/Implementations` do projeto `API.Control` para a pasta

`API.Control.Services/Implementations`.

2. **Mapeamentos (AutoMapper):** Mova todos os arquivos da pasta `Mappings` do projeto `API.Control` para a pasta `API.Control.Services/Mappings`.

Ajuste de Namespaces: Atualize os namespaces (e.g., `API.Control.Services.Implementations` para `API.Control.Services.Implementations`).

3.6. Ajuste de Referências entre Projetos

Este é um passo crítico. Você precisará adicionar referências entre os projetos para que eles possam se comunicar corretamente.

Usando a CLI do .NET:

Na raiz da sua solução, execute os seguintes comandos:

```bash
dotnet add API.Control.Infrastructure/API.Control.Infrastructure.csproj
reference API.Control.Core/API.Control.Core.csproj
dotnet add API.Control.Services/API.Control.Services.csproj reference
API.Control.Core/API.Control.Core.csproj
dotnet add API.Control.Services/API.Control.Services.csproj reference
API.Control.Infrastructure/API.Control.Infrastructure.csproj
dotnet add API.Control/API.Control.csproj reference
API.Control.Core/API.Control.Core.csproj
dotnet add API.Control/API.Control.csproj reference
API.Control.Services/API.Control.Services.csproj
```

Explicação das Dependências:

- `API.Control.Infrastructure` depende de `API.Control.Core` (para entidades e interfaces).
- `API.Control.Services` depende de `API.Control.Core` (para interfaces e DTOs) e `API.Control.Infrastructure` (para o `DbContext` ou repositórios).
- O projeto principal `API.Control` (que se tornará `API.Control.Web` ou `API.Control.Api`) dependerá de `API.Control.Core` (para DTOs e interfaces) e `API.Control.Services` (para as implementações dos serviços).

3.7. Atualização do Projeto Principal (API.Control.Web/Api)

O projeto `API.Control` original agora será o seu projeto de apresentação (API). Você precisará remover os arquivos que foram movidos e ajustar as referências e as configurações.

1. **Remova Arquivos Duplicados:** Exclua as pastas `Entities`, `Services/Interfaces`, `DTOs`, `ValueObjects`, `Data`, `Migrations`, `Services/Implementations`, `Mappings` do projeto `API.Control`.
2. **Atualize `Program.cs`:**
   - Remova os `using`s antigos que apontam para os namespaces que foram movidos.
   - Adicione os novos `using`s para os namespaces dos projetos `Core`, `Infrastructure` e `Services`.
   - Ajuste as injeções de dependência para usar as interfaces e implementações dos novos projetos.
   - Certifique-se de que o `AddDbContext` e o `AddAutoMapper` estão configurados para usar os assemblies corretos (e.g., `typeof(AppDbContext).Assembly` para o projeto `Infrastructure`, e os perfis do AutoMapper do projeto `Services`).
3. **Endpoints:** Os arquivos na pasta `Endpoints` permanecerão no projeto principal, mas seus `using`s precisarão ser atualizados para referenciar os DTOs e interfaces de serviço do `API.Control.Core`.

3.8. Ajuste de Namespaces e `using`s

Este é um processo iterativo. Após mover os arquivos e ajustar as referências, você provavelmente terá muitos erros de compilação relacionados a namespaces. Você precisará ir em cada arquivo e:

- **Atualizar o Namespace:** Altere o namespace declarado no arquivo para refletir a nova localização do arquivo (e.g., de `API.Control.Entities` para `API.Control.Core.Entities`).

- **Ajustar `using`s:** Adicione ou remova as declarações `using` para que o arquivo possa acessar as classes e interfaces dos outros projetos.

3.9. Instalação de Pacotes NuGet

Certifique-se de que cada novo projeto tenha os pacotes NuGet necessários. Por exemplo:

- `API.Control.Core`: Pode não precisar de muitos pacotes, talvez apenas `System.ComponentModel.DataAnnotations`.
- `API.Control.Infrastructure`: Precisará de `Microsoft.EntityFrameworkCore`, `Microsoft.EntityFrameworkCore.Sqlite` (ou outro provedor de banco de dados).
- `API.Control.Services`: Precisará de `AutoMapper` e `AutoMapper.Extensions.Microsoft.DependencyInjection`.
- `API.Control`: Precisará de todos os pacotes relacionados ao ASP.NET Core, Swagger, FluentValidation, etc.

3.10. Compilação e Teste

Após todas as movimentações e ajustes, tente compilar a solução. Resolva quaisquer erros de compilação que surgirem. Uma vez que a solução compile, execute a aplicação e teste todos os endpoints para garantir que tudo está funcionando como esperado. Se você tiver testes automatizados, execute-os para verificar regressões.

3.11. Considerações Adicionais

- **GlobalUsings.cs:** Se você estiver usando `GlobalUsings.cs`, você pode precisar criar um para cada novo projeto ou ajustar o existente para incluir os namespaces mais comuns de cada camada.
- **Testes:** A pasta `Tests` pode ser movida para a raiz da solução e conter projetos de teste separados para cada módulo (e.g., `API.Control.Services.Tests`, `API.Control.Infrastructure.Tests`).
- **Refatoração Contínua:** A modularização é um processo contínuo. À medida que seu projeto evolui, você pode identificar novas oportunidades para refatorar e otimizar a

estrutura dos módulos.

Ao seguir este guia, você transformará seu projeto API.Control em uma solução mais modular e organizada, o que facilitará o desenvolvimento, a manutenção e a escalabilidade a longo prazo.