Clean Architecture - Projetando com .NET

Rodrigo Machado

1 Clear Archtecture

A Clean Architecture, proposta por Robert C. Martin (o famoso Uncle Bob), é uma abordagem para organizar projetos de software com foco em separação de responsabilidades, baixo acoplamento e alta coesão. Com ela, conseguimos deixar o código mais testável, escalável e fácil de manter. Ademais, com a aplicação da mesma temos a certeza da aplicação do SOLID em nossos projetos.

Neste artigo, quero mostrar como aplico a Clean Architecture nos meus projetos .NET, com uma estrutura que uso no dia a dia e que já está bem madura. A ideia é passar por cada camada do projeto, explicar o propósito e mostrar como tudo se conecta na prática.

O projeto que vou usar como exemplo foi finalizado dois meses antes da escrita deste artigo, se chama Cestas De Maria. Ele foi desenvolvido para o gerenciamento de doação de cestas básicas, que é realizado por um centro espírita aqui da minha cidade e onde meu amigo trabalha. Estarei expondo aqui o código (não completo) do backend desse projeto. Tanto ambientes de produção e testes não estão disponíveis para testes, afinal o foco aqui não é o projeto, mas sim a forma de organização do mesmo.

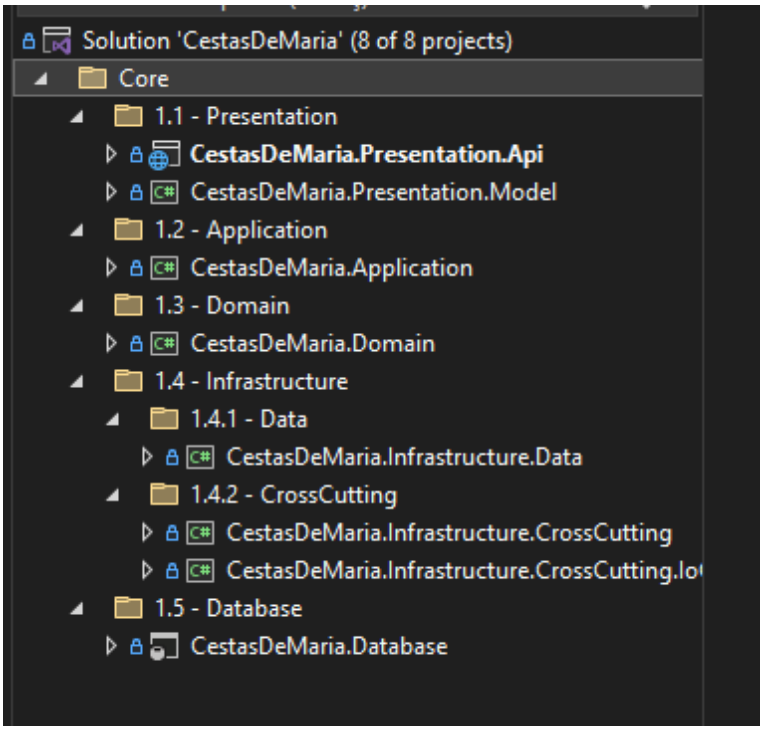
Essa é a organização do meu projeto, nomeiei a solução como **CestasDeMaria**, as pastas internas seguem o padrão a seguir:

- 1.1 - Presentation: Nessa pasta da solução criei o projeto CestasDeMaria.Presentation.Api e o projeto CestasDeMaria.Presentation.Model;
- 1.2 - Application: Nessa pasta contém o projeto CestasDeMaria.Application;
- 1.3 - Domain: Nessa pasta está o projeto CestasDeMaria.Domain;
- 1.4 - Infrastructure: Nessa pasta criei outras duas pastas de solução:
  - 1.4.1 - Data: Nessa pasta está o projeto CestasDeMaria.Infrastructure.Data
  - 1.4.2 - CrossCutting: Nessa pastá contém o projeto CestasDeMaria.Infrastructure.CrossCutting e o projeto CestasDeMaria.Infrastructure.CrossCutting.IoC;
- 1.5 - Database: Aqui está o projeto de banco de dados CestasDeMaria.Database.

É importante reforçar como os projetos foram criados: todos eles são projetos do tipo biblioteca (class library), com exceção do '.Api', que é uma ASP.NET Core API, e do

‘.Database‘, que é um Database Project. Os tipos de projetos são muito importantes para o entendimento de como a solução criada será utilizada.

A imagem a seguir apresenta a estruturação do projeto:



A seguir, explicarei cada uma das camadas e como os projetos são estruturados, a fim de demonstrar uma forma simples e robusta de desenvolver utilizando Clean Architecture.

2 Camada Presentation

A camada **Presentation** é a responsável por fazer a ponte entre o mundo externo e a aplicação. Nela ficam os pontos de entrada da aplicação, ou seja, os endpoints que expõem os serviços da API, além da definição de como os dados chegam e saem do sistema.

Ela é a primeira camada acessada quando uma requisição é feita, e sua responsabilidade é receber a requisição HTTP, tratar e validar os dados recebidos, chamar os serviços da camada *Application*, e retornar a resposta adequada ao cliente. Importante lembrar que essa camada **não deve conter regras de negócio**, apenas orquestração e adaptação de dados.

Essa camada foi dividida em dois projetos:

- **CestasDeMaria.Presentation.Api**: responsável pelos endpoints, middlewares e tratamento geral da API.
- **CestasDeMaria.Presentation.Model**: responsável pelos modelos de entrada e saída da API, separando a lógica de formatação dos dados da estrutura da API.

Abaixo explico cada pasta presente nesses dois projetos:

CestasDeMaria.Presentation.Api

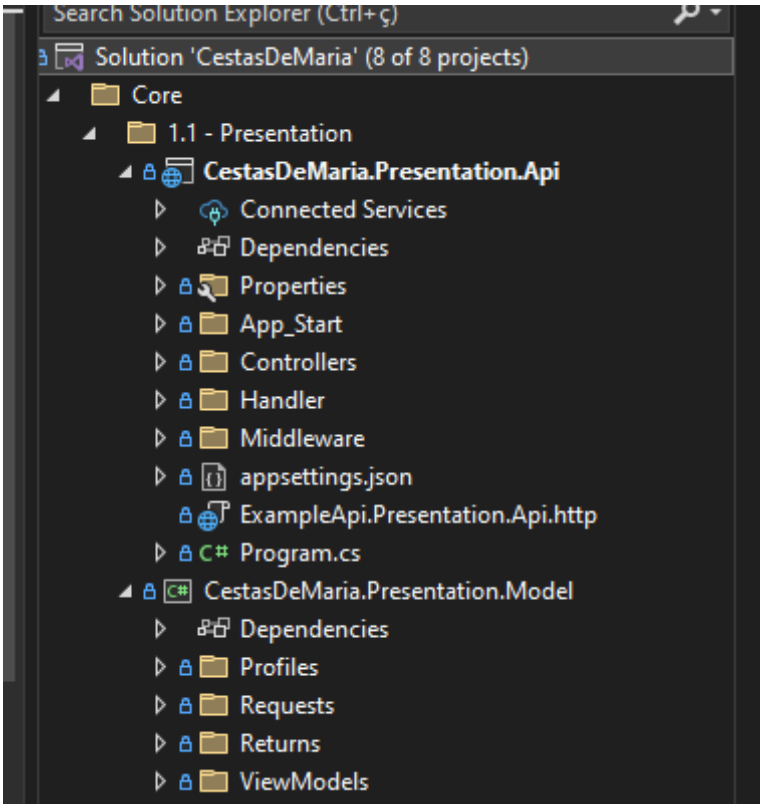
- **App_Start**: contém configurações iniciais da aplicação, como CORS, Swagger, autenticação JWT, e outros serviços que são configurados no startup.
- **Controllers**: onde estão os *Controllers* da API, cada um mapeando endpoints específicos para um domínio da aplicação (ex: usuários, produtos, autenticação, etc). Cada método do controller recebe os dados, chama a camada *Application* e retorna o resultado.
- **Handler**: pasta onde organizo os *Exception Handlers*, responsáveis por capturar erros lançados ao longo do pipeline da requisição e retornar respostas amigáveis e padronizadas.
- **Middleware**: contém middlewares personalizados, como log de requisição/resposta, autenticação por token, controle de exceções ou qualquer outro comportamento transversal que precise atuar no pipeline HTTP.
- **Program.cs e appsettings.json**: arquivos padrão de configuração e inicialização da API. O `Program.cs` define a construção do host e o `appsettings.json` centraliza as configurações externas como string de conexão, chaves secretas, entre outros.
- **ExampleApi.Presentation.Api.http**: arquivo usado para realizar testes de endpoints via VS Code, Postman ou diretamente pelo Visual Studio.

CestasDeMaria.Presentation.Model

- **Profiles**: aqui estão os mapeamentos do AutoMapper entre ViewModels e DTOs. Isso garante que as transformações de dados fiquem centralizadas e reutilizáveis.
- **Requests**: modelos usados para representar os dados de entrada enviados pelo cliente (ex: ao criar ou editar um registro).
- **Returns**: modelos usados para representar a resposta enviada pela API. Dessa forma, conseguimos ter um padrão de retorno para o front-end ou outros consumidores.

- **ViewModels**: modelos que representam a junção de dados para exibição específica, geralmente utilizados em listagens ou visualizações mais completas, diferentes dos retornos simples.

Separar os modelos de entrada e saída em um projeto distinto (`.Model`) traz maior organização e facilita o reuso em testes, outros front-ends ou até mesmo microserviços que consumam a mesma estrutura de dados. O resultado da estruturação está na imagem a seguir:



3 Camada Application

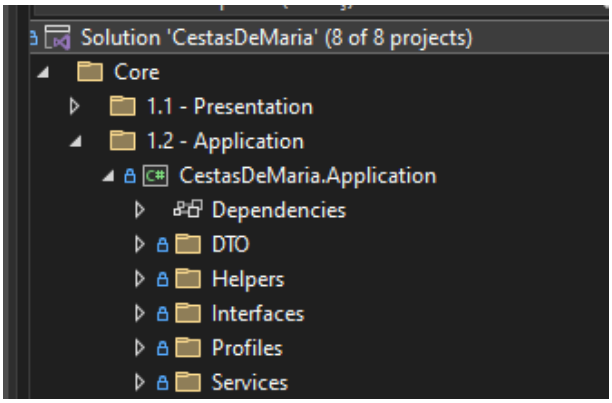
A camada **Application** é onde fica a orquestração da lógica de negócio da aplicação. É aqui que os casos de uso são implementados e as regras mais específicas são tratadas, sempre com foco em manter o baixo acoplamento e respeitar o princípio da inversão de dependência.

Essa camada atua como intermediária entre a camada *Presentation* e a camada *Domain*. Ela define contratos (interfaces) para os serviços e repositórios e implementa os fluxos de execução do sistema, sempre se baseando nas regras definidas no domínio.

No projeto `CestasDeMaria.Application`, a estrutura de pastas está organizada da seguinte forma:

- **DTO**: Contém os objetos de transferência de dados utilizados internamente entre serviços e camadas. Esses objetos geralmente não são expostos externamente, diferentemente dos 'Requests' e 'Returns' da camada Presentation. São utilizados para facilitar a comunicação entre camadas e aplicar transformações nos dados quando necessário.
- **Helpers**: Contém classes utilitárias que auxiliam nas operações da camada, como formatadores, validadores, manipuladores de strings, datas ou qualquer outro tipo de funcionalidade reutilizável que não seja regra de negócio.
- **Interfaces**: Aqui são definidas as interfaces utilizadas pela aplicação. Isso inclui contratos de repositórios, serviços externos, notificações, autenticação, e outros. Essas interfaces são implementadas na camada de Infrastructure, respeitando o princípio da inversão de dependência (D de SOLID).
- **Profiles**: Contém os perfis do AutoMapper utilizados para mapear automaticamente os dados entre entidades e DTOs. Essa separação facilita a manutenção dos mapeamentos da aplicação.
- **Services**: Onde os casos de uso são realmente implementados. Cada serviço da aplicação é responsável por realizar uma ação específica, como cadastrar um usuário, gerar um relatório, ou realizar uma validação. Esses serviços orquestram chamadas para as interfaces, manipulam os dados recebidos e retornam os resultados esperados, mantendo a lógica de negócio fora dos controllers.

Essa camada é extremamente importante para manter a organização do sistema e separar a lógica de orquestração da lógica de domínio. Ao centralizar os fluxos aqui, conseguimos deixar os controllers leves e o domínio limpo, além de facilitar muito a criação de testes unitários para os casos de uso. A seguir está como o projeto está estruturado na solução:



4 Camada Domain

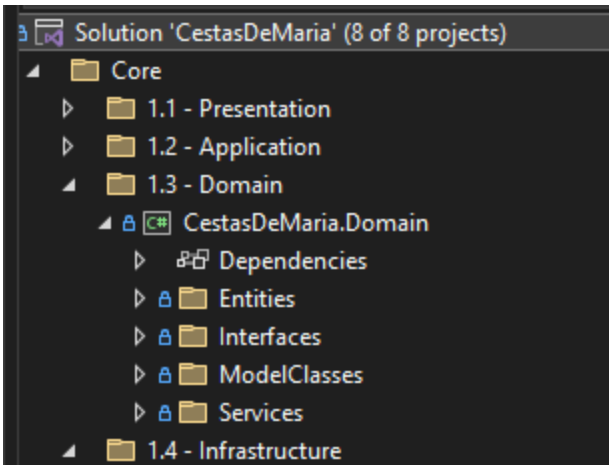
A camada **Domain** é o núcleo da aplicação. É aqui que estão definidas as regras de negócio puras, as entidades do sistema e os contratos fundamentais. Essa camada representa o que é mais importante no sistema, independente da tecnologia utilizada — seja banco de dados, APIs externas ou a forma de apresentação.

A ideia central é que a camada de domínio seja completamente independente. Ela **não depende de nenhuma outra camada**, e qualquer projeto que utilize essa estrutura deve respeitar essa separação. Isso garante que as regras de negócio possam ser reutilizadas, testadas e mantidas com facilidade.

No projeto `CestasDeMaria.Domain`, temos a seguinte estrutura:

- **Entities**: contém as entidades centrais do sistema. Cada classe representa um conceito importante do negócio (como Usuário, Produto, Reserva, etc). Essas classes devem conter apenas as regras de negócio, propriedades e comportamentos diretamente relacionados à entidade.
- **Interfaces**: define os contratos de repositórios e serviços que serão implementados posteriormente na camada Infrastructure. Isso garante que a camada de domínio possa ser testada e reutilizada sem conhecer detalhes técnicos como banco de dados ou serviços externos.
- **ModelClasses**: armazena modelos auxiliares utilizados pelas entidades, como objetos de valor (value objects), enums, ou qualquer estrutura de suporte ao domínio que não seja diretamente uma entidade.
- **Services**: contém os serviços de domínio, responsáveis por regras que envolvem múltiplas entidades ou operações mais complexas dentro do próprio domínio. Diferente dos serviços da camada Application, aqui o foco é executar lógica de negócio pura, sem dependências externas.

Manter a camada de domínio isolada garante que a aplicação esteja alinhada com os princípios da **Clean Architecture** e do **DDD (Domain-Driven Design)**. É como se essa camada definisse o "coração" do que o sistema realmente faz, sem se preocupar com como ou por onde isso será executado. Segue como está configurado na solução:



5 Camada Infrastructure

A camada **Infrastructure** é responsável por implementar tudo aquilo que foi definido por interfaces na camada *Domain*. Isso inclui acesso a banco de dados, envio de e-mails, autenticação, log, entre outros. Essa camada lida diretamente com a tecnologia, e por isso pode ser alterada com o tempo sem impactar o domínio do sistema.

É importante destacar que a dependência aqui é sempre de fora para dentro. A camada Infrastructure conhece Application e Domain, mas nunca o contrário.

No projeto `CestasDeMaria`, a pasta Infrastructure foi dividida em três partes:

CestasDeMaria.Infrastructure.Data

Responsável por tudo relacionado ao acesso a dados. É aqui que implementamos os repositórios, contextos e classes auxiliares de persistência.

- **Context**: contém o `DbContext` (no caso do Entity Framework) e as configurações gerais de mapeamento das entidades com o banco de dados.
- **Helpers**: contém métodos auxiliares relacionados a banco de dados, como extensões, conversores ou tratamentos específicos.
- **Repository**: onde ficam as implementações concretas das interfaces de repositório definidas na camada *Domain*. Essa separação permite que os testes possam ser feitos com mocks, desacoplando completamente a lógica de persistência.

CestasDeMaria.Infrastructure.CrossCutting

Essa pasta armazena funcionalidades que atravessam o sistema todo, como envio de e-mails, templates de mensagens, enums globais e adaptadores genéricos. É o famoso *"cross-cutting concerns"*.

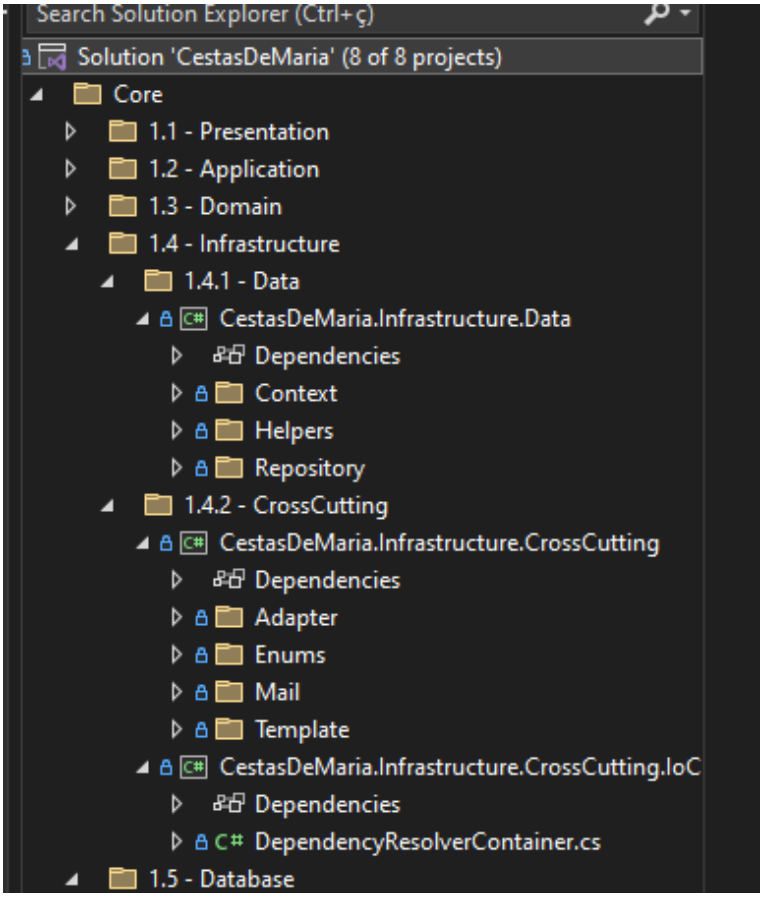
- **Adapter**: contém adaptadores que fazem a ponte entre implementações internas e bibliotecas externas. Um exemplo clássico seria o envio de e-mails via SMTP, SendGrid, etc.
- **Enums**: enums compartilhados por todo o projeto.
- **Mail**: implementações para envio de e-mails, configuração de remetente, modelo de conteúdo, etc.
- **Template**: armazena templates de e-mail, mensagens ou qualquer conteúdo pré-formatado usado de forma dinâmica.

CestasDeMaria.Infrastructure.CrossCutting.IoC

Responsável por registrar as dependências no container de Inversão de Controle (IoC), utilizando os padrões da Microsoft ou bibliotecas externas como Autofac.

- **DependencyResolverContainer.cs**: classe onde todas as interfaces e serviços são registrados no container. É aqui que conectamos a interface com a implementação (por exemplo, `IUserRepository` com `UserRepository`).

Essa separação dentro da Infrastructure garante organização e flexibilidade. Se amanhã trocarmos o banco de dados ou a forma de enviar e-mails, não precisaremos tocar em nenhuma outra camada — basta alterar os detalhes aqui. Segue a imagem de como a configuração foi feita no projeto:



6 Camada Database

A camada **Database** é responsável por organizar tudo que envolve o banco de dados do sistema de forma separada da lógica de negócio. Aqui concentro os scripts de criação e alteração de tabelas, além de dados de exemplo e estruturação base para deploy do banco.

Mesmo utilizando um ORM como o Entity Framework, gosto de manter essa camada para registrar de forma clara como o banco foi modelado, possibilitando até a utilização em ambientes com migração controlada via CI/CD.

No projeto `CestasDeMaria.Database`, tenho as seguintes pastas:

- **Scripts**: aqui ficam scripts SQL que envolvem inicialização do banco, inserção de dados default, procedimentos de manutenção, entre outros.
- **Tables**: contém os scripts de criação de tabelas do banco de dados. Mesmo que o

sistema utilize Migrations, essa estrutura ajuda a documentar o banco, permite criação manual ou rápida visualização da modelagem.

Essa separação em um projeto próprio facilita o versionamento do banco e permite integração com pipelines de CI/CD. Em muitos projetos, essa camada é usada por DBAs ou por outras equipes responsáveis por validar alterações na estrutura do banco antes de aplicar em produção.

Além disso, manter os scripts aqui permite que o mesmo projeto backend seja facilmente adaptado a diferentes ambientes ou bancos, sem depender exclusivamente do ORM para manter a estrutura do banco de dados.

7 Conclusão

A utilização da **Clean Architecture** nos meus projetos tem sido uma prática constante e extremamente benéfica. Ela me permite manter uma organização clara, com separação de responsabilidades, facilitando a manutenção, a testabilidade e a escalabilidade da aplicação.

A estrutura apresentada no projeto `CestasDeMaria` é o padrão que costumo seguir em todas as aplicações que desenvolvo em .NET. Cada camada tem um papel bem definido, desde a entrada da requisição até a persistência no banco de dados. Isso garante que qualquer desenvolvedor que venha a trabalhar no projeto consiga entender rapidamente a arquitetura e aplicar melhorias sem medo de quebrar o sistema.

Outro ponto que considero essencial é a utilização de **Docker** com `docker-compose`. Sempre que inicio um projeto, já deixo preparado um ambiente completo para desenvolvimento local, sem a necessidade de configurar bancos de dados ou serviços externos manualmente. Isso garante que qualquer pessoa possa clonar o repositório e executar o sistema com um simples comando, sem depender de um ambiente de testes ou de infraestrutura adicional.

Essa abordagem de isolamento e organização me permite criar sistemas robustos, reutilizáveis e prontos para crescer com segurança.

Espero que esse artigo tenha ajudado a entender como estruturo meus projetos com base nos princípios da Clean Architecture. Fique à vontade para adaptar essa estrutura conforme sua realidade e suas necessidades. O mais importante é sempre manter a clareza e o foco na separação de responsabilidades.