Escola	Ada Tech
Professor	Fábio Fonseca
Estudante	Carolina Armentano Ana Carolina Armentano e Silva
Data de Entrega	16/01/2024

Resumo: Camadas DDD, Aggregates e Injeção de Dependência do .NET

No universo do desenvolvimento de software, falamos sobre Camadas DDD, Agregados e Injeção de Dependência. Esses conceitos são importantes para organizar e construir sistemas de maneira eficiente.

O Domain-Driven Design (DDD) foca em entender o negócio, usando uma linguagem que todos possam entender. Aggregates, por sua vez, são como grupos de coisas relacionadas que ajudam a manter as coisas organizadas. Já a Injeção de Dependência é uma prática que torna o código mais flexível, separando as partes do sistema de forma inteligente.

Ao conectar essas ideias, podemos construir sistemas de software melhores e de mais fácil manutenção.

Microsserviços orientados a DDD

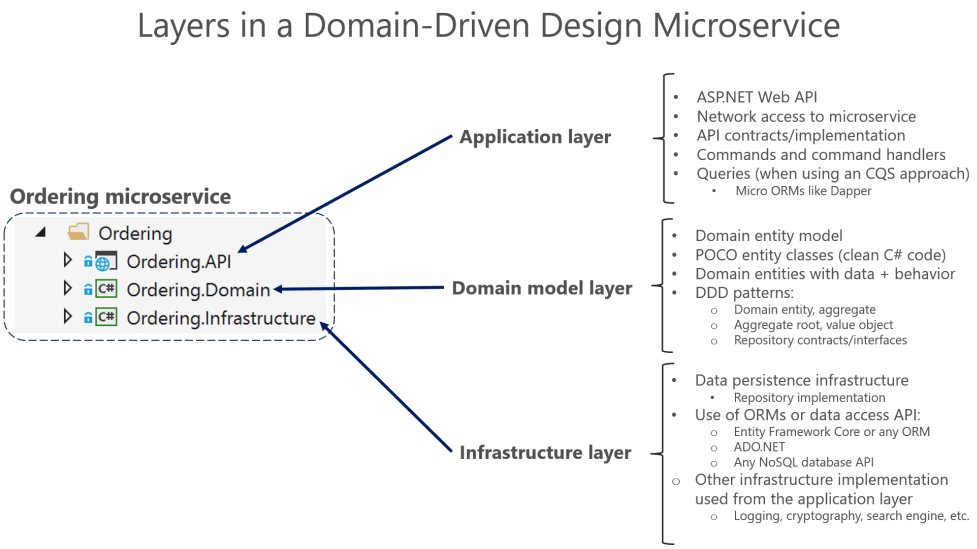
O Design Orientado a Domínio (DDD) emerge como uma abordagem essencial no desenvolvimento de software, propondo uma modelagem alinhada à realidade dos negócios.

Aplicação do DDD

A aplicação prática do DDD manifesta-se na criação de microsserviços, onde Contextos Limitados delimitam áreas independentes de problemas no domínio do negócio. Essa abordagem busca balancear a criação de microsserviços pequenos com a necessidade de evitar comunicações excessivas entre eles.

Organização em Camadas

A organização em camadas é um componente fundamental na gestão da complexidade de aplicativos empresariais. Cada camada, como a de Modelo de Domínio, Aplicativo e Infraestrutura, desempenha um papel específico na estruturação e funcionalidade do sistema.



Camada de Modelo de Domínio

É responsável por representar conceitos de negócios, contendo entidades de domínio e regras de negócio. A independência da infraestrutura é crucial nesta camada, garantindo que as entidades de domínio permaneçam focadas nos conceitos do negócio e não sejam contaminadas por detalhes de persistência.

Camada de Aplicativo

Coordenando tarefas e interações, esta camada contém a lógica de aplicativo que depende das entidades de domínio. Sua responsabilidade é manter-se fina, não contendo regras de negócio, mas sim coordenando as operações entre as entidades de domínio.

Camada de Infraestrutura

Responsável pela persistência de dados, esta camada lida com a implementação concreta dos repositórios e mecanismos de armazenamento. Ela deve ser separada da camada de modelo de domínio, seguindo os princípios de Ignorância de Persistência e Ignorância de Infraestrutura.

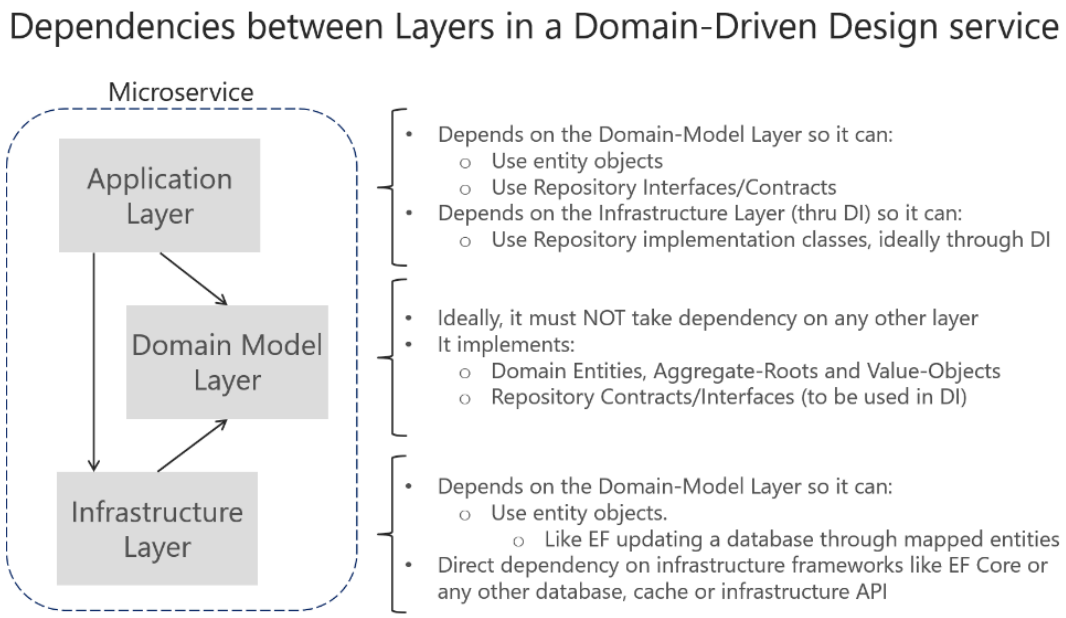
Delimitação de Microsserviços e Dependências entre Camadas

A delimitação adequada de microsserviços é um desafio, e o equilíbrio entre coesão interna e comunicação eficiente é crucial.

Para tanto, é essencial controlar as dependências entre as camadas. Uma abordagem onde as camadas são implementadas como bibliotecas independentes pode facilitar o controle dessas dependências. A camada de modelo de domínio deve ser

independente, não dependendo diretamente da infraestrutura. O design de camadas deve ser independente e aplicável a cada microsserviço.

A dependência entre as camadas deve ser gerenciada cuidadosamente, buscando um design independente e aplicável a cada microsserviço. A Camada de Aplicativo coordena tarefas e interações, enquanto a Camada de Infraestrutura lida com a persistência de dados, mantendo-se separada do Modelo de Domínio para preservar a independência e a expressividade deste último:



Conclusão

Em resumo, a aplicação eficaz do DDD em microsserviços envolve a criação de modelos de domínio expressivos, a delimitação adequada de microsserviços e a organização eficiente em camadas, garantindo a independência entre o modelo de domínio e a infraestrutura. Esses princípios contribuem para a construção de sistemas flexíveis, escaláveis e alinhados com os objetivos de negócios.

Aggregates

Em DDD, um Aggregate é um padrão que ajuda a organizar e estruturar as entidades de domínio em um modelo. Um Aggregate é um grupo de entidades e objetos de valor que são tratados como uma única unidade coesa. Essa unidade é tratada como uma transação atômica, o que significa que todas as operações dentro do Aggregate são consistentes e duráveis, ou seja, ou todas são executadas com sucesso ou nenhuma é.

O Aggregate é liderado por uma entidade específica conhecida como a **raiz do Aggregate (Root Entity)**. A raiz do Aggregate é a única entrada pela qual outras partes do sistema podem acessar o Aggregate. Todas as operações no Aggregate devem ser

iniciadas pela raiz, e a raiz é responsável por garantir a consistência e a integridade do Aggregate como um todo.

O uso de Aggregates ajuda a evitar a complexidade desnecessária e melhora a escalabilidade em sistemas distribuídos, já que as operações dentro de um Aggregate podem ser tratadas localmente sem a necessidade de comunicação constante com outras partes do sistema.

Implementar um modelo de domínio de microsserviço com o .NET

Modelo de domínio de microsserviço

A organização de pastas em uma biblioteca .NET Personalizada seguindo o modelo DDD contém agregações representadas por grupos de entidades e objetos de valor. Contratos de repositório (interfaces) são incluídos na camada de modelo de domínio, representando os requisitos de infraestrutura do modelo.

Estruturação de Agregações

Uma agregação é um grupo de objetos de domínio consistentemente transacionais. A consistência transacional implica que uma agregação está garantidamente consistente e atualizada após uma ação de negócios.

Entidades de Domínio como Classes POCO

As entidades de domínio são implementadas como classes POCO (Plain Old CLR Objects), sem dependências diretas do EF Core ou de qualquer outro ORM. A raiz de agregação é marcada com uma interface vazia (IAggregateRoot) para indicar seu papel como raiz de uma agregação.

Encapsulamento de Dados nas Entidades de Domínio

Propriedades de navegação de coleções são expostas como somente leitura, e métodos explícitos são fornecidos para manipular essas coleções. Métodos da classe raiz agregada controlam as operações de atualização das entidades filhas, mantendo a consistência e aplicando regras de negócio.

Mapeamento de Propriedades com Apenas Acessadores Get para Campos no EF Core 1.1 ou Posterior

No EF Core 1.1 ou posterior, é possível mapear propriedades definidas apenas com acessadores get para campos na tabela do banco de dados. Isso permite modelar entidades DDD de forma mais expressiva, mantendo o controle sobre a consistência dos dados.

Mapeamento de Campos sem Propriedades no EF Core 1.1 ou Posterior

É possível mapear colunas para campos diretamente, sem a necessidade de propriedades correspondentes. Isso é útil para campos internos que não precisam ser acessados externamente.

Aggregate Root na modelagem de domínios ricos

O DDD (Domain-Driven Design) surgiu com a intenção de melhorar a modelagem das aplicações, auxiliando na construção de softwares mais aderentes ao mundo real e às operações de negócio. Um padrão muito utilizado no DDD é o padrão agreggate, que auxilia no agrupamento de funcionalidades de um objeto de domínio partindo sempre de uma raiz de agregação (aggregate root).

Padrão de Agregação

O padrão de agregação no DDD envolve agrupar funcionalidades de objetos de domínio em torno de uma raiz de agregação. A raiz de agregação é responsável por gerenciar objetos de domínio filhos e garantir que os dados sejam salvos em conjunto.

Entidades e Raiz de Agregação

No exemplo, são apresentadas duas entidades: FarmArea (Área da Fazenda) e FarmAreaCoordinate (Coordenada da Área da Fazenda). A entidade FarmArea é a raiz de agregação, sendo responsável por manipular e validar as entidades filhas (FarmAreaCoordinates).

Validação e Consistência

Cada entidade deve ser responsável por sua própria validação, garantindo dados consistentes e evitando a persistência de dados inválidos. As regras de negócio são contidas dentro dos objetos de domínio.

Testabilidade

Essa abordagem facilita a criação de testes, permitindo validar as regras de negócio sem a necessidade de mocks de repositório. A interface de marcação (IAggregateRoot) é utilizada para garantir que o repositório receba apenas a raiz de agregação.

AssertionConcern

É mencionado o uso do conceito de AssertionConcern do Vernon para auxiliar na validação. Esse conceito se refere a um conceito associado ao autor Vaughn Vernon e ao seu trabalho no campo do Domain-Driven Design (DDD).

O termo AssertionConcern em DDD se relaciona à ideia de encapsular a lógica de validação e assertivas dentro do próprio domínio. Em outras palavras, as entidades e objetos de valor devem ter a capacidade de validar seu próprio estado e impor as regras de consistência do domínio.

A principal motivação por trás disso é garantir que as regras de negócio e as invariantes do domínio estejam centralizadas nas próprias entidades e objetos de valor. Isso

evita que estados inválidos sejam representados no sistema, uma vez que cada entidade é responsável por garantir sua própria consistência.

Persistência

O repositório recebe apenas a raiz de agregação para persistência, não permitindo o acesso direto às entidades filhas.

Esses conceitos destacam a importância da estruturação do modelo de domínio, validação dentro das entidades, e a utilização do padrão de agregação para criar sistemas mais aderentes ao negócio e facilmente testáveis.

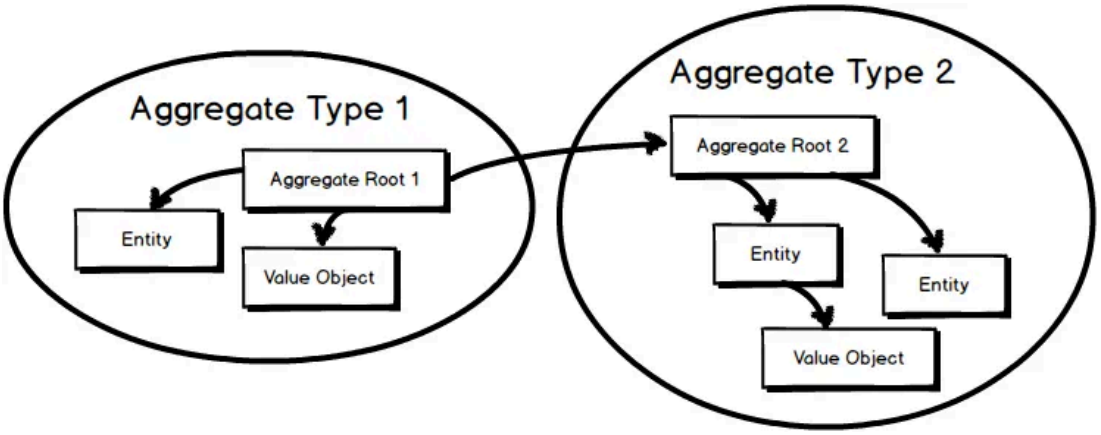
Explorando o poder dos Aggregates em Domain-Driven Design e Clean Architecture

---

No desenvolvimento de software, os conceitos de Domain-Driven Design (DDD) e Clean Architecture oferecem orientações valiosas para a construção de sistemas robustos e de fácil manutenção.

Aggregates

Aggregates são elementos-chave em ambas as metodologias, representando conjuntos de objetos de domínio tratados como uma unidade única. Eles encapsulam entidades e objetos de valor relacionados, garantindo consistência e definindo limites transacionais.



Raiz de Agregação

A raiz de agregação é a entidade principal dentro de um aggregate, sendo o ponto de entrada para acessar e modificar o estado interno do aggregate. Ela protege a integridade do conjunto e aplica regras de negócio.

Benefícios dos Aggregates

- **Consistência e Integridade:** Garantem que as entidades dentro deles estejam sempre em um estado consistente.
- **Limites Transacionais:** Possibilitam atomicidade e integridade transacional, mantendo a consistência dos dados.
- **Escalabilidade e Desempenho:** Otimizam o acesso e a modificação de dados, reduzindo conflitos e melhorando o desempenho.
- **Testes Simplificados:** Facilitam testes unitários, permitindo testar comportamentos de domínio de forma isolada.

Diretrizes para Projetar Aggregates

- **Identificar Limites:** Encontre áreas no domínio onde entidades naturalmente se agrupam e defina raízes claras de agregação.
- **Projeto Coerente:** Garanta que cada aggregate foque em um único conceito de negócio coeso, evitando complexidade.
- **Definir Regras de Consistência:** Estabeleça regras claras de consistência e invariantes dentro de cada aggregate.
- **Interfaces Bem-Definidas:** Crie interfaces para comunicação entre aggregates, seguindo o Princípio da Inversão de Dependência.
- **Armazenamento e Recuperação de Dados:** Considere os mecanismos de persistência ao projetar aggregates.

Exemplo Prático

Um exemplo prático envolve um aggregate chamado BlogPost, que encapsula entidades como Author e Comment, e objetos de valor como PostId e Content. O BlogPost demonstra como aggregates organizam dados relacionados, aplicam regras de negócio e fornecem uma interface coesa.

Conclusão

Aggregates são ferramentas poderosas para construir aplicações escaláveis e testáveis. Ao encapsular entidades e objetos de valor relacionados, eles promovem consistência, limites transacionais e facilitam testes.

Aplicados eficientemente, os aggregates melhoram a modularidade, flexibilidade e facilitam a manutenção dos sistemas, alinhando-os mais de perto aos requisitos do domínio. Compreender o poder dos aggregates e aplicá-los com discernimento pode aprimorar significativamente a qualidade e a longevidade das aplicações de software.

Injeção de Dependência do .NET

O .NET dá suporte ao padrão de design de software de DI (injeção de dependência). A injeção de dependência no .NET é uma parte interna da estrutura, juntamente com a configuração, o registro em log e o padrão de opções.

Injeção de Dependência (DI)

A DI é uma técnica para alcançar a Inversão de Controle (IoC) entre classes e suas dependências.

Dependências

Dependências são objetos dos quais outros objetos dependem. Podem vir a existir problemas com dependências embutidas no código, como dificuldade de substituição e configuração em projetos grandes.

Solução com Injeção de Dependência

A utilização de interfaces ou classes base podem abstrair a implementação da dependência. As dependências são registradas em um contêiner de serviço usando o IServiceCollection, no construtor da classe onde é usada.

Exemplo Prático

No texto, há o exemplo da interface IMessageWriter e a implementação do MessageWriter. É feito o registro do serviço no contêiner usando AddSingleton e ocorre o uso da DI no construtor da classe Worker.

Atualizações Dinâmicas

Com a injeção de dependência, pode se alterar facilmente a implementação da dependência sem modificar a classe que a utiliza.

Várias Regras de Descoberta de Construtor

Quando um tipo tem vários construtores, o DI usa o construtor com mais parâmetros resolvíveis por DI. No exemplo fornecido, há três construtores. O primeiro construtor é sem parâmetros e não requer nenhum serviço do provedor de serviços. Suponha que o registro em log e as opções tenham sido adicionados ao contêiner di e sejam serviços resolvíveis por DI.

Quando o contêiner tentar resolver o tipo ExampleService, ele gerará uma exceção, pois os dois construtores são ambíguos. É possível evitar a ambiguidade definindo um construtor que aceita ambos os tipos resolvíveis por DI.

Tempos de Vida do Serviço

- **Transitório:** Criado sempre que solicitado.
- **Com Escopo:** Criado uma vez por solicitação de cliente (conexão).
- **Singleton:** Criado na primeira solicitação e reutilizado.

Comportamento da Injeção de Construtor

Os construtores precisam ser públicos para serem resolvidos por DI. A injeção de construtor exige um construtor aplicável.

Validação de Escopo

Verificações devem ser realizadas para garantir que os serviços com escopo não são resolvidos do provedor raiz e não são injetados em singletons.

Serviços com Chave

Há suporte a registros e pesquisas de serviços com base em uma chave, assim como a possibilidade de registrar e resolver serviços diferentes com a mesma interface usando uma chave.

Referências

Camadas DDD:
https://learn.microsoft.com/pt-br/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice

GitHub DDD:
https://github.com/VaughnVernon/IDDD_Samples_NET/tree/master

Aggregate:
https://learn.microsoft.com/pt-br/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/net-core-microservice-domain-model

Aggregate:
https://renatogontijo.medium.com/aggregate-root-na-modelagem-de-dom%C3%ADnios-ricos-7317238e6d97

Aggregate:
https://medium.com/@edin.sahbaz/exploring-the-power-of-aggregates-in-domain-driven-design-and-clean-architecture-6408d6128d3b

Injeção de dependência:
https://learn.microsoft.com/pt-br/dotnet/core/extensions/dependency-injection