Dan faz código

Arquitetura Limpa - Uma Introdução

04 de maio de 2020 | 12 minutos de leitura

Há muito tempo uso a arquitetura clássica "N-Tier" (UI Layer -> Logic Layer -> Data Layer) na maioria dos aplicativos que construo. Eu confio muito em interfaces e aprendi há muito tempo que IoC (Inversão de Controle) é seu amigo. Essa arquitetura me permitiu criar aplicativos testáveis e pouco acoplados, e tem me servido bem até agora. No entanto, como muitos engenheiros de software profissionais, estou sempre procurando como posso melhorar minha arquitetura ao projetar aplicativos.

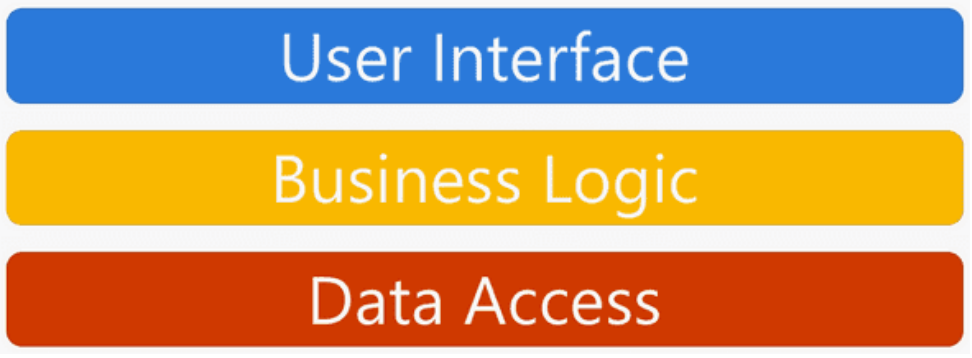
Recentemente, me deparei com Clean Architecture a partir de uma apresentação de Jason Taylor em uma conferência Goto, e fiquei fascinado com essa arquitetura/padrão. Ele validou algumas das coisas que eu já estava fazendo, mas melhorou em outras áreas que sempre me pareceram um pouco desajeitadas (como integração com serviços de terceiros, e onde diabos vai a validação?).

NOTA: Embora essa arquitetura seja independente de linguagem e estrutura, mencionarei alguns termos do .NET Framework para ajudar a ilustrar os conceitos.

O que é arquitetura N-Tier?

Primeiro, vamos examinar a arquitetura clássica N-Tier. Isso existe há mais de 20 anos e ainda é comum na indústria hoje.

N-Tier geralmente tem 3 camadas:



Fonte: Microsoft Docs

Cada camada só pode se comunicar com a próxima camada abaixo (ou seja, a interface do usuário não pode se comunicar diretamente com os dados). Na superfície, essa limitação pode parecer uma boa ideia, mas na implementação, significa um conjunto diferente de modelos para cada camada, o que resulta em muito código de mapeamento. Esse problema é amplificado quanto mais camadas você adiciona.

Camada de interface do usuário

- Controladores
- VerModelos
- DTOs
- Mapeadores
- Visualizações
- Ativos estáticos

Normalmente, isso seria um projeto MVC ou API da Web.

Camada de lógica de negócios

- Lógica de Negócios/Aplicativos (geralmente implementado como Serviços)
- Integração com serviços de terceiros
- Chamadas diretamente para a *camada de dados*

É aqui que está a essência de um aplicativo N-Tier.

Camada de dados

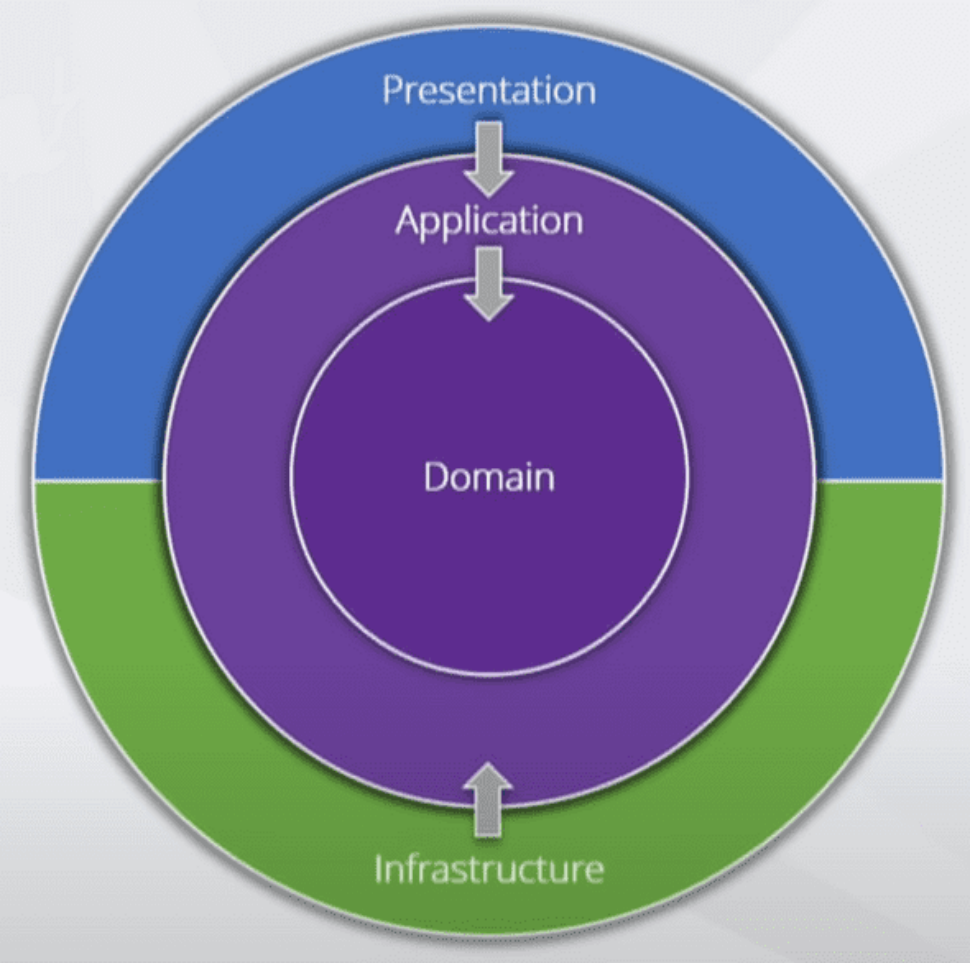
- Modelos de dados
- Consultas
- Acesso ao banco de dados
- Migrações

Apresentando a Arquitetura Limpa

Esta arquitetura teve muitos nomes ao longo dos anos. Onion Architecture , Hexagonal Archecture , Screaming Architecture e outros. Essa abordagem não é nova, mas também não é tão comum quanto talvez devesse ser.

Então, comparado ao N-Tier, o que é Arquitetura Limpa e como ela é diferente?

Vamos começar com uma imagem:



Fonte: Clean Architecture, Jason Taylor - Conferência Goto

A primeira coisa a notar aqui é a direção das dependências. Todas as dependências fluem para dentro. As camadas externas podem se comunicar com QUALQUER camada interna (compare isso com N-Tier, onde cada camada só pode se comunicar com a camada abaixo). Isso segue o Princípio de Inversão de Dependência , o que significa que as dependências são injetadas em vez de serem criadas explicitamente. Outro nome para isso é o *Princípio de Hollywood: não ligue para nós, nós ligaremos para você* . 😊

Aplicativo e *Domínio* são considerados o 'núcleo' do aplicativo. *A aplicação* depende de *Domain* , mas *Domain* não depende de nada.

Quando o *aplicativo* precisa de funcionalidade da *infraestrutura* (por exemplo, acesso ao banco de dados), o *aplicativo* definirá suas próprias interfaces que a

*infraestrutura*irá implementar. Essa dissociação é enorme e é um dos principais

benefícios dessa abordagem. Não só permite testes unitários mais fáceis, mas também significa que ignora a persistência. Ao consultar dados, o armazenamento de dados subjacente pode ser um banco de dados, serviço da Web ou até mesmo arquivo simples. O aplicativo não se importa e não precisa saber. Com os detalhes de implementação fora do núcleo, isso nos permite focar na lógica de negócios e evita a poluição com detalhes menos importantes. Ele também oferece flexibilidade, pois hoje os dados podem vir de uma fonte de dados, mas no futuro podem precisar vir de uma fonte de dados diferente. Devido ao baixo acoplamento, apenas a camada de infraestrutura precisará ser alterada para dar suporte a isso.

Jeffrey Palermo definiu os quatro inquilinos da arquitetura limpa:

- O aplicativo é construído em torno de um modelo de objeto independente
- Camadas internas definem interfaces. Camadas externas implementam interfaces
- A direção do acoplamento é para o centro
- Todo o código principal do aplicativo pode ser compilado e executado separadamente da infraestrutura

Tudo isso se esforça para tornar mais fácil para os desenvolvedores fazer as coisas certas e difícil para eles fazerem as coisas erradas. Estamos nos esforçando para "forçar os desenvolvedores no poço do sucesso".

Camada de Domínio

- Entidades
- Objetos de valor
- Agregados (se estiver fazendo DDD)
- Enumerações

A *Camada de Domínio* é o coração de seu aplicativo e responsável por seus modelos principais. Os modelos devem ser ignorantes quanto à persistência e encapsular a lógica sempre que possível. Queremos evitar acabar com Modelos Anêmicos (ou seja, modelos que são apenas coleções de propriedades).

A *Camada de Domínio* pode ser incluída na *Camada de Aplicação* , mas se você estiver usando uma estrutura de entidade do tipo ORM, a *Camada de Infraestrutura* precisará fazer referência aos modelos de domínio; nesse caso, é melhor dividir em uma camada separada.

As anotações de dados devem ser deixadas de fora dos modelos de domínio. Isso deve ser adicionado na *camada de infraestrutura* usando a sintaxe fluente. Se você está acostumado a usar as anotações de dados para sua validação, recomendo usar Fluent Validation na *camada de aplicativo* , que fornece mais capacidade de eventos do que anotações.

Camada de aplicação

- Interfaces de aplicativos
- Ver Modelos / DTOs
- Mapeadores
- Exceções de aplicativos
- Validação
- Lógica
- Comandos / Consultas (se estiver fazendo CQRS)

Este é o uso do *Aplicativo* do seu *Domínio* para implementar os casos de uso para o seu negócio. Isso fornece o mapeamento de seus modelos de domínio para um ou mais modelos de exibição ou DTOs.

A validação também entra nessa camada. Pode-se argumentar que a Validação entra no domínio, mas existe o risco de que os erros levantados possam fazer referência a campos não presentes no DTO / View Model, o que causaria confusão. IMO é melhor ter validação potencialmente duplicada, do que validar um objeto que não foi passado para o comando/consulta.

Minha preferência é usar comandos e consultas CQRS para lidar com todas as solicitações de aplicativos. O MediatR pode ser usado para facilitar isso e adicionar comportamentos adicionais, como registro, armazenamento em cache, validação automática e monitoramento de desempenho a cada solicitação. Se você optar por não usar o CQRS, poderá trocá-lo pelo uso de serviços.

A *Camada de Aplicação* APENAS faz referência à *Camada de Domínio* . Ele não sabe nada de bancos de dados, serviços web, etc. No entanto, ele define interfaces (por exemplo, IContactRepository, IContactWebService, IMessageBus), que são implementadas pela *Camada de Infraestrutura* .

Camada de Infraestrutura

- Base de dados
- serviços web
- arquivos
- Barramento de mensagens
- Exploração madeireira
- Configuração

A *Camada de Infraestrutura* implementará interfaces da *Camada de Aplicação* para fornecer funcionalidade para acessar sistemas externos. Eles serão conectados pelo contêiner IoC, geralmente na *camada de apresentação* .

A *Camada de Apresentação* normalmente terá uma referência à *Camada de Infraestrutura* , mas apenas para registrar as dependências com o container IoC. Isso pode ser evitado com contêineres IoC como Autofac com o uso de Registros e varredura de montagem.

Camada de apresentação

- Controladores MVC
- Controladores de API da Web
- Swagger / NSwag
- Autenticação / Autorização

A *camada de apresentação* é o ponto de entrada para o sistema do ponto de vista do usuário. Suas principais preocupações são o roteamento de solicitações para a *camada de aplicativo* e o registro de todas as dependências no contêiner IoC. Autofac é meu container favorito, mas use o que te deixa feliz. Se você estiver usando ASP.NET, as ações nos controladores devem ser muito finas e, na maioria das vezes, simplesmente passarão a solicitação ou comando para MediatR .

Camada de teste

A *camada de teste* é outro ponto de entrada para o sistema. Principalmente, isso deve visar a *camada de aplicativo* , que é o núcleo do aplicativo. Como toda infraestrutura é abstraída por interfaces, zombar dessas dependências se torna trivial.

Se você estiver interessado em aprender mais sobre testes, recomendo o Clean Testing .

Variações

As camadas descritas até aqui, compõem a abordagem básica da Arquitetura Limpa. Você pode precisar de mais camadas dependendo da sua aplicação.

Se você não estiver usando um ORM, poderá combinar as Camadas de *Domínio* e de *Aplicação* para simplificar.

A mais externa direita também pode ter mais segmentos. Por exemplo, você pode querer dividir a infraestrutura em outros projetos (por exemplo, Persistência).

Essa abordagem funciona bem com o Domain-Driven Design, mas funciona igualmente bem sem ele.

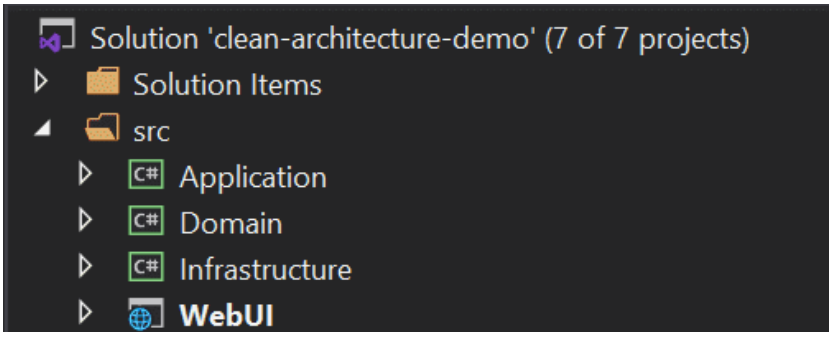
CQRS é a abordagem recomendada para o ponto de entrada na *camada de aplicação* . No entanto, você também pode usar serviços típicos se não se sentir confortável com isso.

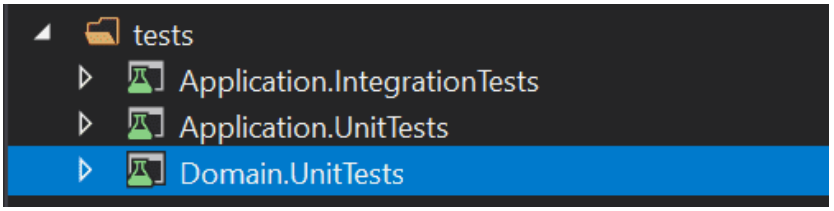
Também vi implementações em que o núcleo do aplicativo é dividido em 4 camadas internas. IMO isso é um exagero para a maioria dos projetos. Queremos equilibrar clareza com simplicidade.

Modelos iniciais

Isso tudo soa muito bem, certo? Mas como posso começar?

Felizmente para todos nós, Jason Taylor criou um modelo de solução .NET Core, que contém um aplicativo totalmente funcional com um Frontend Angular 9 e testes de unidade e integração associados. Este é um ótimo ponto de partida para ver tudo em ação.





Você pode encontrar mais sobre isso aqui

Resumo

Clean Architecture não é de forma alguma nova e não é nada inovador. No entanto, com alguns ajustes na arquitetura N-Tier típica, o resultado é uma solução completamente testável e mais sustentável que pode se adaptar às mudanças mais rapidamente. Devido ao baixo acoplamento entre as camadas externa e interna, as modificações podem ser facilitadas, o que pode ser a diferença entre uma aplicação com duração de 2 anos e 10 anos.

Ainda estou resolvendo os problemas em minhas próprias implementações, mas realmente vejo as vantagens com essa abordagem e estou animado para ver os resultados ao longo do tempo.

Dê-lhe uma tentativa e deixe-me saber como você vai.

Recursos

- Arquitetura Limpa - Jason Taylor
- Regras para uma arquitetura mais limpa
- Microsoft Docs - Arquitetura limpa
- A Arquitetura Limpa - Robert C. Martin
- Descascando a arquitetura da cebola - Tony Sneed
- Onion Architecture - Jeffrey Palermo
- MediatR

Daniel Mackay vive e trabalha na Sunshine Coast, na Austrália, e está empolgado com todas as coisas da web. Desenvolvedor em tempo integral. Pai em tempo integral. Surfista de meio período.
LinkedIn | GitHub | Estouro de pilha | Twitter