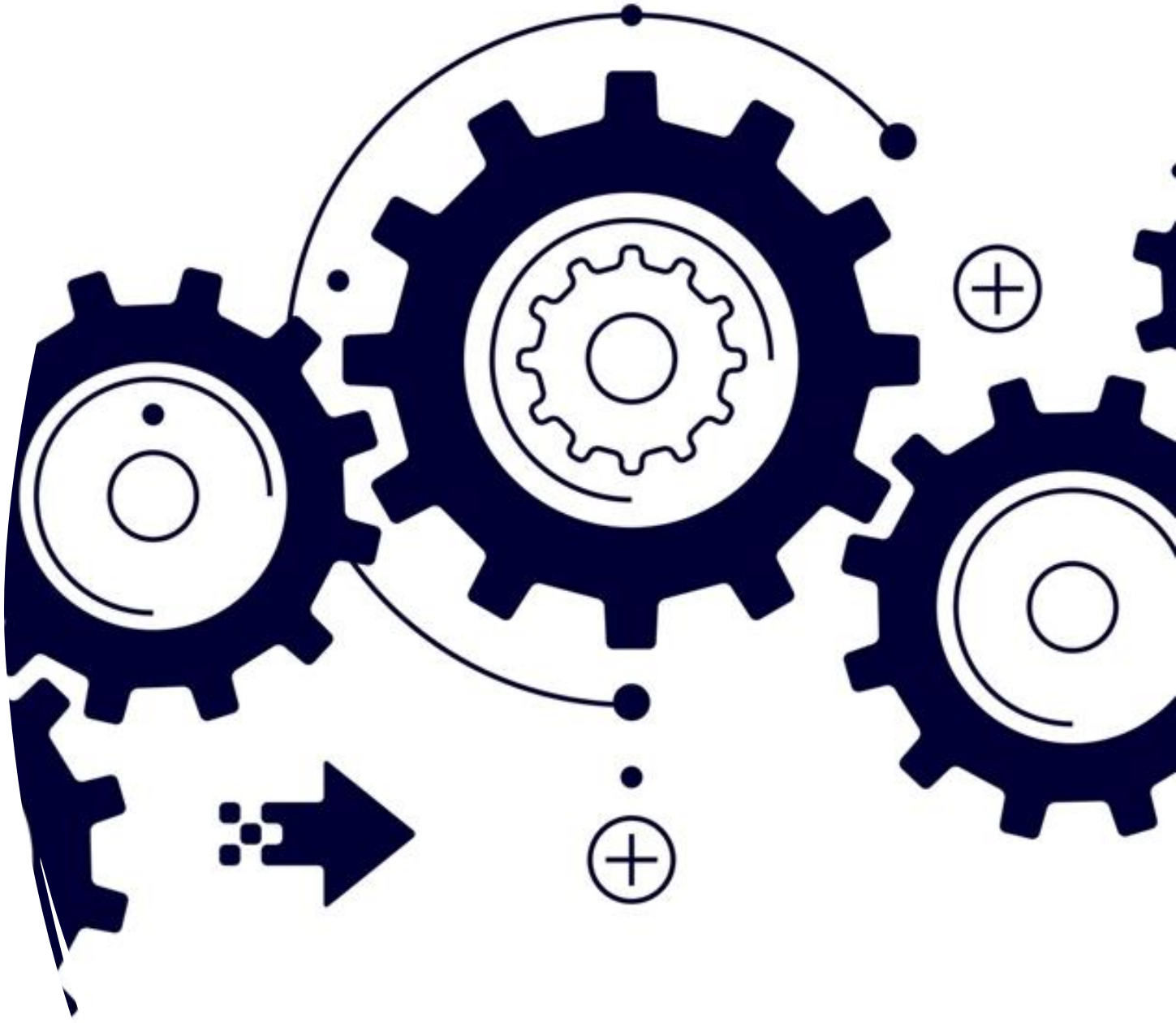
Padrões e Práticas Avançadas

FIAP



Sobre

Carlos Fernando Sylverio

- Técnico em Eletrônica
- Bacharel em Sistemas de Informação
- MBA em Gestão de Projetos
- Certificado Scrum Master
- 20 anos atuando com desenvolvimento
- Gestão de desenvolvimento desde 2014
- Aŕea de interesse: Arquitetura de Software e Orientação a Objetos

https://www.linkedin.com/in/csylverio/

Padrões e Práticas Avançadas

Objetivo

Refatorar projeto MyFinance (monolito) aplicando padrões arquiteturais e boas práticas de desenvolvimento em .NET C#.

O que veremos nesta live:

- Criar uma arquitetura de 3 camadas no projeto
- Aplicação de injeção de dependência e inversão de controle (IoC)
- Criação de testes unitários

Refatorando projeto MyFinance

Estrutura inicial (branch *main*)

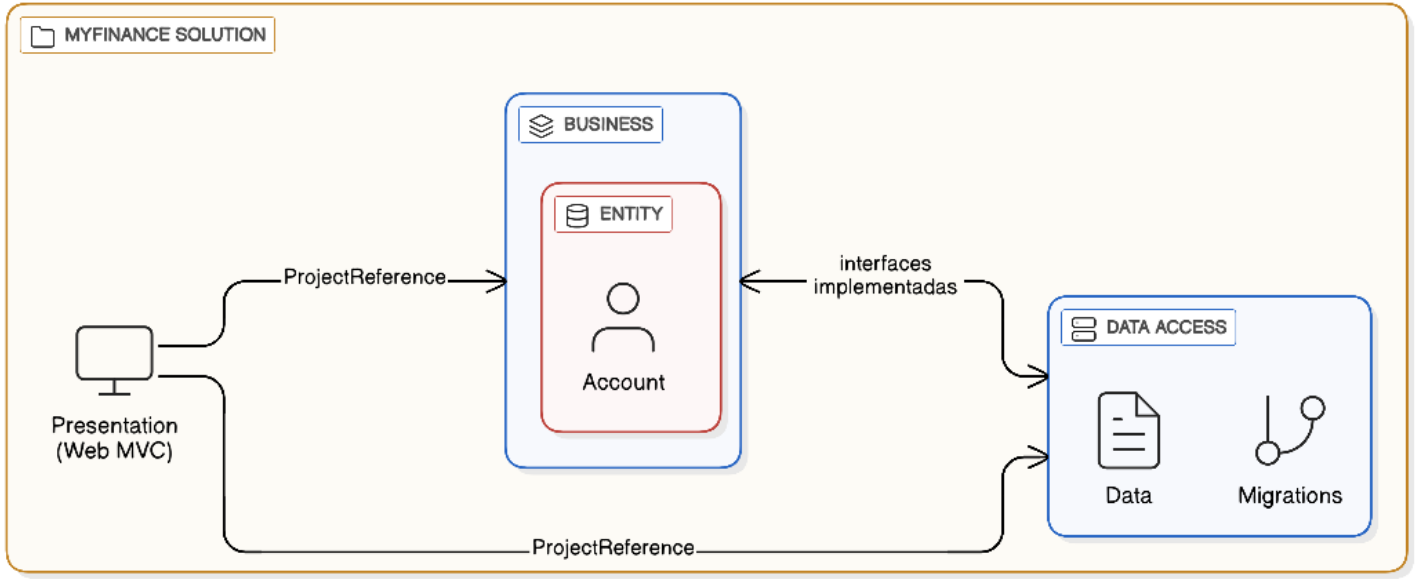
Repo: https://github.com/csylverio/tds-3-tier

Na branch `main` temos:

- **Único projeto Web MVC**
- Solution com apenas o projeto **MyFinance**.
- **Models de domínio + Models de apresentação** misturados.
- **DbContext,** diretório **`Data`, migrations do EF Core** dentro do próprio projeto MVC.
- Controllers acessando **diretamente o contexto de dados** (ou serviços muito acoplados à infraestrutura).

Arquitetura de 3 Camadas

- Organiza uma aplicação em três camadas lógicas distintas
- Objetivo é separar a interface de usuário, as regras de negócio e o acesso a dados, reduzindo acoplamento e aumentando a modularidade



Referencias: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

Arquitetura de 3 Camadas

Camada de Apresentação (Presentation Layer)

- Responsável pela **interface com o usuário** ou pela **exposição da API**, é a camada que **recebe requisições, valida entradas básicas** e encaminha para a camada de negócio

Camada de Negócio (Business Layer ou Domain Layer)

- É o **núcleo da aplicação**, onde residem as **regras de negócio**, lógica de processamento, validações e regras de domínio.
- Essa camada **não deve depender da camada de apresentação**, mas pode depender da camada de dados **apenas por meio de contratos ou abstrações**.

Camada de Acesso a Dados (Data Access Layer)

- Responsável pela comunicação com **fontes externas de dados**, geralmente bancos relacionais ou NoSQL. Fornece às camadas superiores **métodos de persistência** e **recuperação de dados**, escondendo detalhes de infraestrutura.

Refatorando projeto MyFinance

Estrutura refatorada para 3 camadas (branch feature/3-tiers)

Repo: https://github.com/csylverio/tds-3-tier/tree/feature/3-tiers

Na branch `feature/3-tiers` temos uma **solution de 3 projetos**, seguindo a arquitetura de 3 camadas:

- **MyFinance**
  - Continua sendo a **camada de apresentação (Web MVC)**.
  - Depende das outras camadas via `ProjectReference`.
- **MyFinance.Business**
  - Projeto Class Library que representa a **camada de negócio**.
  - Adicionamos um "pacote Entity" com `Account`, tratando a entidade como representação do domínio, seguindo conceito de DDD.
- **MyFinance.DataAccess**
  - Projeto Class Library para **acesso a dados / infraestrutura**.
  - Movemos o diretório **`Data`** e **`Migrations`** para cá (DbContext, configuração de EF Core, migrations etc.).

Injeção de Dependência (DI)

Implementação Tradicional (acoplamento forte)

- A própria classe cria diretamente suas dependências
- Uso da palavra reservada new

Porque usar DI:

- Interfaces como **contratos** para permitir a troca de implementações sem alterar o código consumidor
- O Controller/Serviço depende **da abstração**, não da implementação.
- Permite usar padrões como **Mock** (unit test), **Decorator** (desing pattern), **Proxy** (desing pattern), etc.
- Torna o sistema altamente **flexível** e **substituível em runtime**.
- Base do DI: você injeta interfaces, não classes concretas.

Inversão de Controle (IoC)

Em **.NET C**ore, o contêiner de DI nativo usa IServiceCollection para registrar serviços.

```
builder.Services.AddScoped<IAccountService, AccountService>();
builder.Services.AddScoped<IAccountRepository, AccountRepository>();
```

O contêiner IoC faz:

- Identificação das dependências do Controller via reflection.
- Criação automática de instâncias.
- Injeção no construtor.
- Respeito ao ciclo de vida escolhido.
- Descarte quando necessário.

Inversão de Controle (IoC)

O **IServiceCollection** é o coração do mecanismo de Inversão de Controle (IoC) do ASP.NET Core. O contêiner IoC faz:

- Identificação das dependências do Controller via reflection.
- Criação automática de instâncias.
- Injeção no construtor.
- Respeito ao ciclo de vida escolhido.
- Descarte quando necessário.

Inversão de Controle (IoC)

Ciclos de vida dos serviços: Transient, Scoped, Singleton

- **Singleton**
  - Instância única para toda a aplicação
  - Criada no startup
  - Mesma instância para todos os requests
- **Scoped**
  - Uma instância por *request HTTP*
  - Compartilhada dentro do pipeline daquela requisição
- **Transient**
  - Nova instância a cada resolução
  - O contêiner garante:
  - *Nunca* injetar um serviço de ciclo maior em um menor (ex.: Singleton → Scoped)

Inversão de Controle (IoC)

Comparativo entre tipos de injeção de dependência

Item	Tradicional	Com DI
Criação de objetos	A própria classe cria	IoC cria e fornece
Acoplamento	Alto	Baixo
Testabilidade	Difícil	Excelente
Flexibilidade	Baixa	Alta
Uso de interfaces	Opcional	Essencial

Refatorando projeto MyFinance

Estrutura com teste unitário (branch *unit-tests*)

Repo: https://github.com/csylverio/tds-3-tier/tree/feature/unit-tests

Na branch `unit-tests` temos:

- Projeto Class Library, somente para testes
- Adicionar pacotes de teste **xunit** e **Moq**
- Adicionar referências aos projetos que serão testados (Business/DataAccess).
- **Testes unitários puros** (isolando dependências com Moq)
- **Testes de integração** de serviços **com EF** usando **InMemory**

Testes Unitários e Framework xUnit

XUnit x NUnit

- O xUnit é mais moderno, eficiente em paralelismo, e preferido em novos projetos por sua simplicidade e código mais moderno.
- O NUnit é mais antigo, é frequentemente visto em projetos legados e é valorizado por documentação mais estável em algumas áreas, como configuração de testes

Aspecto	xUnit	NUnit
Ferramenta escolhida pelo time do .NET	Sim	Não
Suporte à DI nativa	Excelente	Limitado
Compatibilidade com templates oficiais	Default	Precisa instalar pacotes extra
Execução via dotnet test	Natural	Ótima, mas exige mais configurações para cenários avançados

Padrões e Práticas Avançadas

Dúvidas