GUIA DEFINITIVO DE ARQUITETURA BACK-END UTILIZANDO .NET CORE

BECOMING A BACK END EXPERT

KENERRY SERAIN

Isenção de Responsabilidade

Todas as informações contidas neste guia são provenientes de minhas experiencias pessoais com programação ao longo de vários anos de estudos. Embora eu tenha me esforçado ao máximo para garantir a precisão e a mais alta qualidade dessas informações e acredite que todas as técnicas e métodos aqui ensinados sejam altamente efetivos para qualquer estudante de programação desde que seguidos conforme instruídos, nenhum dos métodos ou informações foi cientificamente testado ou comprovado, e eu não me responsabilizo por erros ou omissões. Sua situação e/ou condição particular pode não se adequar perfeitamente aos métodos e técnicas ensinados neste guia. Assim, você deverá utilizar e ajustas as informações deste guia de acordo com sua situação e necessidades.

Todos os nomes de marcas, produtos e serviços mencionados neste guia são propriedades de seus respectivos donos e são usados somente como referência. Além disso, em nenhum momento neste guia há a intenção de difamar, desrespeitar, insultar, humilhar ou menosprezar você leitor ou qualquer outra pessoa, cargo ou instituição. Caso qualquer escrito seja interpretado dessa maneira, eu gostaria de deixar claro que não houve intenção nenhuma de minha parte em fazer isso. Caso você acredite que alguma parte deste guia seja de alguma forma desrespeitosa ou indevida e deva ser removida ou alterada, pode entrar em contato

diretamente comigo através do e-mail kenerry.software.engineer@gmail.com.

Direitos Autorais

Sobre o Autor

Meu nome é Kenerry Serain, atualmente sou arquiteto back-end na Accenture, uma das maiores consultorias do mundo. Tenho 23 anos. Desde cedo sou apaixonado por programação, ainda por volta dos 17 anos, entrei no curso de Ciência da Computação, UNIP, onde se formaram na data prevista, apenas um outro guerreiro e eu. Por ser apaixonado por aquilo que faço, sou extremamente autodidata, tenho facilidade para ler e aplicar livros de programação, alguns dos meus preferidos: Use a Cabeça C#, Clean Code, Domain Driven Design, Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure, Patterns of Enterprise Application Architecture, Refactoring, entre muitos outros. Acumulando tanto conhecimento de boas práticas de Arquitetura, C#, .NET, .NET CORE, optei por partilhar deste conhecimento, escrevendo artigos, no meu portal no medium, palestrando em eventos, desde eventos internos de empresas até mesmo palestras na Microsoft. Como todo mundo, meu primeiro programa foi o hello-world, mas, com muita dedicação e empenho, hoje escrevo extensões para o visual studio, pacotes NuGet, imagens docker, padrões de arquitetura e etc. Realmente acredito que qualquer pessoa, pode aprender a programar, mas isto deve ser feito do jeito certo, programar é uma arte, e para te auxiliar, estou escrevendo este livro com alguns princípios arquiteturais que aprendi e/ou criei, para que você comece a todo vapor!

Sumário

Introdução

Seja muito bem-vindo e muito obrigado por ter realizado o download do **Guia Definitivo de Arquitetura Back-End Utilizando .Net Core**, realmente é uma honra ter você aqui comigo. Na sua frente encontra-se o melhor guia de design e padrões de arquitetura escrito em língua portuguesa. Neste e-book vou te ensinar conceitos, técnicas e abordagens que vão mudar o jeito como você programa e pensa em soluções. *Desenvolver softwares exige muito mais do que saber programar* e, se é mais do que só programar, imagine do que só copiar! Na verdade, não há problema nenhum em copiar o bug fix do stack overflow, o problema somente passa a existir na hora que você copia o código, sem se quer entender como ou porque aquela solução funcionou! Neste caso, do que adianta estabilizar o software se você não aprender nada com isso? Não adianta nada, *o que manda é o como*, é ter informação. Quem tem informação, domina a situação.

Entre os mais diferentes cenários dentro do universo da programação, sempre existe o melhor jeito de executar uma determinada tarefa, seja ela, corrigir um bug ou projetar um sistema, sempre existe a melhor maneira de se fazer e, é sobre esse tipo de raciocínio que vamos discorrer aqui. Projetar um sistema, por exemplo, é muito mais do que reunir uma série de tecnologias porque elas são legais ou porque você e seu time são programadores early adopters. Lembre-se sempre que a

tecnologia não é um fim, mas sim um meio para um fim.

Me incomodo ao ver, o desenvolvimento de software, sendo tratado, como algo trivial, de maneira até que, alguns, inclusive ousam dizer: “Programar é que nem receita de bolo, é só seguir o passo a passo” e, é aí que me pergunto porque a Palmirinha não é arquiteta de software na Google.

Além de questionar raciocínios prontos e, debater com você nos próximos capítulos temas verdadeiramente ortodoxos ao programador inteligente, vou lhe falar pragmaticamente sobre uma coleção de boas práticas na hora do código que, farão de você um Asp .Net Core Master e, para você absorver todo o conteúdo deste livro de maneira eficaz, acompanhando as explicações conceituais de forma linear e empírica, desenvolvi uma aplicação sample para você. *TeamManagement*, uma API para realizar a gestão de times de futebol. Um software relativamente simples escrito em Asp .Net Core 2, demonstrando implementações e boas práticas de: Inversion of Control (IoC), Native Dependency Injection (DI), Domain Driven Design (DDD), Generics, SOLID, Don’t Repeat Yourself (DRY), You Ain’t Gonna Need It (YAGNI), Persistence Ignorance, Docker, para hospedar aplicação e o banco de dados, em containeres, Docker Compose, para orquestrar os containeres em âmbito de desenvolvimento e testes, Swagger, para documentação

da API, Unit Tests (XUnit), Integration Tests (XUnit) utilizando Entity Framework Core, Http Conventions, AutoMapper e muito outros conceitos e técnicas de figuras como Martin Fowler, Eric Evans, Jimmy Bogard, Uncle Bob, entre outros!

Sobre este guia

Este livro é um guia para o desenvolvimento de aplicações baseadas em Asp .Net Core. A intenção deste modelo é propor novas soluções e ideias mediante a problemas conhecidos e desconhecidos. Aqui, serão apresentados fundamentos de desenvolvimento de software e, diversas abordagens de implementação utilizando .Net Core. As demonstrações aqui apresentadas, terão como foco, a construção de um bom design de aplicação, em tempo de desenvolvimento, abolindo a priori preocupações com questões de infraestrutura. Independente se seu ambiente de produção é cloud ou on-premise, as decisões de infraestrutura devem ser tomadas depois, quando você já tiver uma aplicação pronta para o ambiente de produção em mãos.

Domain Driven Design
Fundamentos

Domain Driven Design ou Desenvolvimento Orientado a Domínio, é uma abordagem de desenvolvimento de software criada por Eric Evans, onde, as premissas são:

- Foco na resolução de um determinado problema complexo partindo do core business. Nesta abordagem Evans propõe a modelagem do sistema de dentro para fora, do core business para o business out, isto significa que, por exemplo, se você vai desenvolver um software, para gerenciamento de uma pequena lanchonete, você não deve inicia-lo pelo desenvolvimento do serviço de envio de comprovantes por e-mail, pois isto não é o core do sistema, você poderia começar, pelo registro de ingredientes, lanches, preços, e todas as dependências ao redor do propósito principal do software.
- Centralização da regra de negócio do seu código no domínio da aplicação. Evans compartilha a visão de que o domínio é como se fosse o coração do programa, e então o incumbe de resguardar os processos e regras responsáveis pelo funcionamento do corpo (sistema) como um todo. Seguindo este design, muitos pequenos grandes problemas são mitigados, tais como: Regra de negócio segmentada em diversos trechos do código, manutenibilidade baixa,

legibilidade complexa, alto acoplamento, baixa coesão, entre outros.

- Ter uma linguagem comum entre os experts do domínio e o time de desenvolvimento, a linguagem ubíqua. Na minha humilde opinião, um dos pontos principais para a modelagem de um domínio de sucesso é o preceito linguagem ubíqua. Quando você vai modelar um domínio complexo, é necessário ter muita clareza do que está se fazendo, principalmente quando se trata de um passo em direção ao desconhecido. Por exemplo, imagine que você vai fazer um software para o governo, um software de cálculo de imposto embutido, porém, só te passam a solução e não te passam o problema, você tem a matemática, mas não os significados, basicamente seu conhecimento do negócio é zero, você não faz ideia da definição das siglas que eles utilizam nas reuniões, muito provavelmente você vai se confundir e não conseguirá fazer uma entrega de qualidade, pois falta embasamento conceitual, em outras palavras, sem linguagem ubíqua, quer dizer, sem entender do negócio e sem entender do negócio não existe projeto.

Fundamentos Implementados

Desenvolvimento Orientado a Domínio (DDD), transcende a importância de somente um bom código, como o próprio nome já diz, a proposta, visa centrar a solução para o usuário final e não para o programador. Dito isto, em seu livro, *Domain Driven Design: Tackling Complexity in the Heart of Software,* Evans propõe um possível desenho arquitetural para você iniciar sua aplicação. A divisão proposta é o seguinte:

Domain: Separe um dos projetos da sua aplicação, para ser o projeto de domínio, o projeto que representa coração do seu software. Neste projeto, deverão estar centralizados em serviços ou repositórios específicos todas as regras de negócio e ou processos existentes referente ao core business. Lembre-se, que é possível que um determinado negócio tenha múltiplos domínios, por exemplo, imagine um e-commerce, entre os muitos possíveis domínios, poderíamos ter no mínimo dois, um domínio de produtos e um de pedidos, neste caso, Evans propõe que você crie um domínio core ou shared, denotando o conceito de camada transversal ou cross cutting layer, que nada mais é, do que uma camada que atravessa outras camadas, que para este caso em especifico, se trata de um domínio compartilhado com outros domínios. É importante relembrar que como nesta

camada é alojado o core business, o conceito de Persistence Ignorance, cai como uma luva, ou seja, o jeito como o registro é salvo no banco de dados deve ter impacto zero nesta camada. Veja exemplo, abaixo:

```
3 referências | Kenerry Pierre, há 13 dias | 1 autor, 2 alterações
public abstract class DomainServiceBase<TEntity> : IDomainServiceBase<TEntity> where TEntity : Entity
{
    private readonly IRepositoryBase<TEntity> _entityRepository;
    2 referências | Kenerry Pierre, há 16 dias | 1 autor, 1 alteração | 0 exceções
    protected DomainServiceBase(IRepositoryBase<TEntity> entityRepository)
    {
        _entityRepository = entityRepository;
    }

    5 referências | Kenerry Pierre, há 16 dias | 1 autor, 1 alteração | 0 exceções
    public virtual async Task<TEntity> GetByIdAsync(int entityId)
    {
        return await _entityRepository.GetByIdAsync(entityId);
    }

    4 referências | Kenerry Pierre, há 16 dias | 1 autor, 1 alteração | 0 exceções
    public virtual async Task<TEntity> AddAsync(TEntity entity)
    {
        await _entityRepository.AddAsync(entity);
        await _entityRepository.SaveChangesAsync();
        return entity;
    }

    4 referências | Kenerry Pierre, há 13 dias | 1 autor, 2 alterações | 0 exceções
    public virtual async Task<TEntity> UpdateAsync(int entityId, TEntity entity)
    {
        await _entityRepository.UpdateAsync(entityId, entity);
        await _entityRepository.SaveChangesAsync();
        return entity;
    }
    4 referências | Kenerry Pierre, há 13 dias | 1 autor, 2 alterações | 0 exceções
    public virtual async Task DeleteAsync(TEntity entity)
    {
        await _entityRepository.RemoveAsync(entity);
        await _entityRepository.SaveChangesAsync();
    }
}
```

Figura 1-Domain Service Base Design

Perceba que a implementação acima, utiliza do recurso de Generics, para que todo o código trivial (CRUD), não necessite ser

repetido uma vez para cada entidade, desfrutando do padrão Don’t Repeat Yourself (DRY). Você pode se perguntar, beleza, mas como vou escrever regras de negócio genéricas? É aí que está, você não vai! A regra de negócio de cada entidade não deve estar em uma classe genérica, mas sim na classe especifica do business. Por isso, cada método da classe *DomainServiceBase* está marcada com a palavra reservada, *virtual*, o que, significa que na classe filha específica, você pode sobrescrever, o método que quiser, empregando a palavra reservada, *override*, e, adicionando a regra de negócio necessária.

Application: A camada de aplicação é a responsável por orquestrar as chamadas aos serviços de domínio ou as chamadas a camada de aplicação de outros subsistemas. Neste ponto, Evans propõe a utilização do conceito de camada magra (Thin Layer) e reforça que nenhuma regra de negócio deverá existir neste nível. A ideia é que esta camada, não saiba qual o estado atual das validações de regra de negócio, mas sim o estado atual do progresso da execução das tarefas.

Dica Extra: A camada de aplicação de serviços, não pode expor as entidades de domínio. Em outras palavras, isto quer dizer que, o acesso a modificação de estado de uma entidade, não deve acontecer fora da domain layer. Para isto, é muito comum a utilização de padrões de mapeamento, por exemplo,

AutoMapper, onde muito se utiliza, o conhecido, ViewModel. Porém, para sair um pouco da caixinha, no código abaixo, apresento uma implementação diferente a utilização corriqueira de ViewModel. O Request, Response Pattern, que se traduz em: Cada entidade, deve conter, um objeto de request e um de response, ou seja, para entidade, Player, por exemplo, no método de Insert, o parâmetro a ser recebido, não deve ser um objeto Player, mas sim um objeto PlayerRequest, que então será mapeado para o objeto Player e, será enviado para o PlayerDomainService, onde será trabalhada toda regra de negócio da entidade Player, e em caso de sucesso retornará um objeto Player, contendo o objeto inserido no banco de dados, que será mapeado em um objeto PlayerResponse e, devolvido a camadas superiores. Veja exemplo, abaixo:

```
/// <summary>
/// Registra um jogador em um determinado time
/// </summary>
/// <param name="playerRequest"></param>
/// <returns></returns>
[HttpPost]
[ProducesResponseType(201)]
[ProducesResponseType(500)]
1 referência | 0/1 passando | Kenerry Pierre, há 17 dias | 1 autor, 1 alteração | 0 solicitações | 0 exceções
public async Task<IActionResult> RegisterPlayerAsync([FromBody]PlayerRequest playerRequest)
{
    var playerResponse = await _playerApplicationService.AddAsync(playerRequest);
    return CreatedAtRoute(nameof(GetPlayerByIdAsync), playerResponse.Id, playerResponse);
}
```

Figura 2-Web Api Player Register

```
5 referências | Kenerry Pierre, há 17 dias | 1 autor, 1 alteração | 0 exceções
public virtual async Task<TResponse> AddAsync(TRequest entity)
{
    var entityObject = _mapper.Map<TEntity>(entity);
    return _mapper.Map<TResponse>(await _domainServiceBase.AddAsync(entityObject));
}
```

Figura 3-Application Service Base Design

Infrastructure: Esta camada pode ter diversas utilidades. Pode-se a utilizar para disparo de e-mails, acessos externos a outros sistemas, repositórios de banco de dados, log, segurança e muitos outros. Vale neste ponto, reforçar o conceito de camada transversal, log e segurança, por exemplo, são alguns recursos, que podem estar presentes em N camadas. No que se refere ao repositório de banco de dados, pode-se ter implementações de controle de transações com unit of work também, ficaria nesta camada. Segue exemplo de repositório, utilizando Generics e Entity Framework Core:

```
3 referências | Kenerry Pierre, há 12 dias | 1 autor, 2 alterações
public abstract class RepositoryBase<TEntity> : IRepositoryBase<TEntity> where TEntity : Entity
{
    protected DbSet<TEntity> DbSet;
    protected TeamManagementContext DbContext;
    2 referências | Kenerry Pierre, há 16 dias | 1 autor, 1 alteração | 0 exceções
    protected RepositoryBase(TeamManagementContext dbContext)
    {
        DbContext = dbContext;
        DbSet = DbContext.Set<TEntity>();
    }

    11 referências | Kenerry Pierre, há 16 dias | 1 autor, 1 alteração | 0 exceções
    public virtual async Task<TEntity> GetByIdAsync(int entityId)
    {
        return await DbSet.FindAsync(entityId).ConfigureAwait(false);
    }

    8 referências | Kenerry Pierre, há 16 dias | 1 autor, 1 alteração | 0 exceções
    public virtual async Task AddAsync(TEntity entity)
    {
        await DbSet.AddAsync(entity).ConfigureAwait(false);
    }

    4 referências | Kenerry Pierre, há 12 dias | 1 autor, 1 alteração | 0 exceções
    public virtual async Task UpdateAsync(int entityId, TEntity newEntity)
    {
        var trackedEntity=  await DbSet.SingleOrDefaultAsync(register => register.Id == entityId);
        DbContext.Entry(trackedEntity).CurrentValues.SetValues((TEntity)newEntity.WithId(entityId));
    }

    4 referências | Kenerry Pierre, há 12 dias | 1 autor, 1 alteração | 0 exceções
    public virtual Task RemoveAsync(TEntity entity)
    {
        DbSet.Remove(entity);
        return Task.CompletedTask;
    }

    14 referências | Kenerry Pierre, há 16 dias | 1 autor, 1 alteração | 0 exceções
    public virtual async Task SaveChangesAsync()
    {
        await DbContext.SaveChangesAsync();
    }
}
```

Figura 4-Repository Base Design

É importante ressaltar, que no repositório de banco de dados, segundo Eric Evans, não deve ser retornado nada, senão uma entidade de domínio.

Traduzindo em arquitetura de solução todas as caixinhas conceituais explicadas acima, a modelagem arquitetural da aplicação TeamManagement ficou da seguinte maneira:

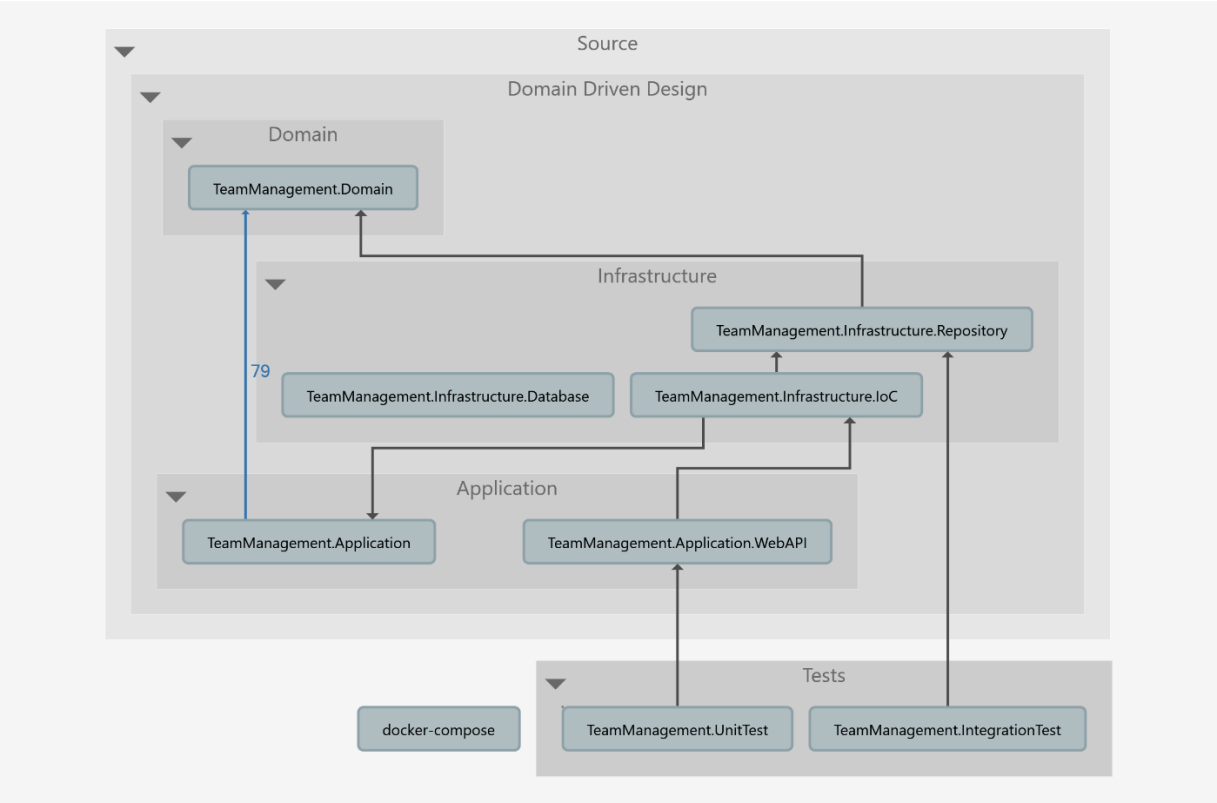

Figura 5-Team Management Architecture Design

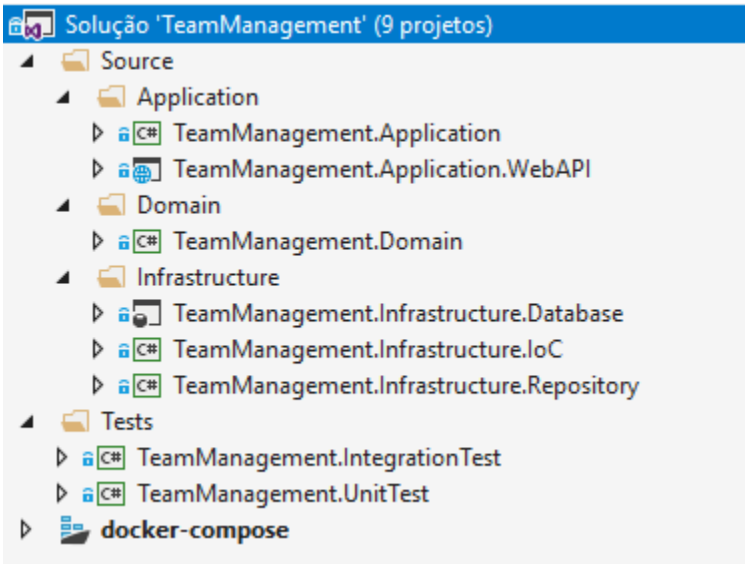

Figura 6-Team Management Solution Design

Para finalizar, é necessário, esclarecer que, o objetivo deste capítulo não foi explicar o conceito de Domain Driven Design em sua totalidade, para isto, existe o próprio livro, a ideia, é conseguir prover uma visão simplificada, de forma empírica, principalmente, para quem está dando os primeiros passos, e ou ainda está confuso com este Pattern. Quando você já estiver dominando todos os conceitos aqui apresentados, busque aprender sobre: Value Objects, Aggregate Root, Bounded Context e outros mais. Temas estes, que poderão ser abordados em uma versão dois deste livro.

Solution Infrastructure

Docker

Docker é um projeto de código aberto, para automatização de deploys portáteis utilizando containeres. Utilizando Docker é possível atrelar todas as dependências da aplicação em uma “caixinha”, tornando o deploy, antes complexo, extremamente, simplificado e autossuficiente (sem dependências externas). Lembrando que, na hora de hospedar a aplicação, você pode escolher, colocar os seus containeres na sua nuvem favorita ou on premises, ambos funcionarão a partir do mesmo setup. Neste momento, não se preocupe muito com a terminologia Docker,

coisas como: Registros, Imagens, Containeres, Repositório, Build, Cluster, Tag, Dockerfile, Docker Hub, Orquestrador e, alguns outros. Foque apenas em entender a essência do Docker para que você não sofra uma overdose de informação e então não absorva nada. Logo à frente veremos os containeres funcionando na prática com Docker Compose.

Docker Compose

Docker Compose é uma ferramenta utilizada para orquestração de aplicações multi-containeres. A ideia é que, com um único comando, todos os containeres da aplicação já estejam trabalhando. No caso da aplicação TeamManagement, temos dois containeres, um para hospedar a aplicação e outro para hospedar o banco de dados.

```yaml
version: '3.6'

networks:
  teammanagement-network:
    driver: bridge

services:
  teammanagement.application.webapi:
    image: teammanagement.application.webapi
    build:
      context: .
      dockerfile: Source/DDD/Application/TeamManagement.Application.WebAPI/Dockerfile
    networks:
      - teammanagement-network
    depends_on:
      - sql.database

  sql.database:
    image: microsoft/mssql-server-linux:2017-latest
```

Figura 7-Team Management Docker Compose

Veja a facilidade que um arquivo Docker Compose, nos permite descomplicar a orquestração de containeres, basta especificar a imagem (Dockerfile), qual ele utilizará para criação do container e Voila! Os containeres estão criados. No exemplo acima, o container da WebAPI, utiliza um Dockerfile local, enquanto para a base de dados utilizamos direto um container do Docker Hub disponibilizado pela Microsoft. O fluxo é seguinte, basta colocar o nome da imagem, se você não tiver esta imagem local irá procurar a imagem no Docker Hub. Outro detalhe importante a se notar, é que um container pode depender de outros containeres, por exemplo, de nada adianta subir o container da WebAPI do TeamManagement, se nenhuma operação de CRUD poderá ser concluída com sucesso, já que todas acessam o banco de dados, então, você pode, atrelar o container de banco de dados, como dependência, ao container de WebAPI, desta maneira, antes de subir o container de WebAPI o Docker Compose subirá todas as dependências primeiro.

Dica Expert: No momento da escrita deste e-book, existem muitos debates, sobre hospedar banco de dados de produção em container. Muito se discute sobre o overhead de I/O de rede, porém a Docker, já resolveu isto de algumas maneiras. Uma delas é, você colocar os containeres em uma rede compartilhada, isto acarretará com que, um container de aplicação, consiga acessar o container de banco de dados sem especificar a porta externa do

container, neste caso, o Docker Gateway (causador do overhead) será ignorado e o overhead será quase insignificante.

Testes

Fundamentos

Se você já tem algum tipo de experiencia no mercado de trabalho da programação, provavelmente, você já trabalhou em algum sistema que o código era cheio de gambiarras, altamente acoplado e sem documentação nenhuma. Um sistema tão mal escrito, que é um milagre estar em produção. Na verdade, pode ser, que eu tenha acabado de descrever seu projeto atual. Quando se vai desenvolver uma demanda qualquer neste tipo de programa, é muito comum que erros “idiotas”, não sejam captados em tempo de desenvolvimento, muito pelo contrário, somente em homologação ou produção.

Em um projeto com testes, mesmo que a arquitetura não seja das melhores, quando você desenvolver um determinado requerimento, alterando algum ponto crítico ou acoplado da aplicação, você consegue captar o erro ainda em tempo de desenvolvimento, basta rodar os testes de novo e então corrigir

sua aplicação antes mesmo de gerar deploy. Agora, se você estiver desenvolvendo uma nova feature, é ainda mais fácil realizar a escrita do teste, afinal você conhece exatamente o que está desenvolvendo, certo? Deste modo, você também facilita a vida de quem for dar manutenção no seu código depois.

Diferente do que alguns dizem, desenvolvedor tem sim a obrigação de testar. Testar código significa dar mais qualidade e garantias ao sistema. Existem alguns tipos de erros, que não precisam e não deveriam chegar na área de qualidade. Não é que testar "perde mais" tempo, esta comparação está errada, não se pode comparar o tempo do projeto com qualidade versus o tempo do projeto sem qualidade. A comparação correta seria, quantidade de bugs em homologação e produção em uma esteira com qualidade e quantidade de bugs em homologação e produção em uma esteira sem qualidade.

Testes Unitários

Existem diversos tipos de testes automatizados: Testes Unitários, Testes Integrados, Testes de Interface, entre outros, porém, de acordo com Martin Fowler, o tipo de teste que traz mais benefício para uma aplicação é o teste unitário. O Teste unitário consiste em testar uma unidade da sua aplicação de forma isolada, de

modo que, entre as prioridades e princípios deste tipo de teste, chegar ao banco de dados não é uma premissa, por isso utilizamos mockups, no teste unitário, o importante é garantir o funcionamento de cada regra de negócio. Veja o exemplo abaixo:

```
1 referência | Kenerry Pierre, há 19 dias | 1 autor, 1 alteração
public class PlayerControllerShould
{
    private readonly PlayerController _playerController;
    private readonly Mock<IPlayerApplicationService> _playerApplicationServiceMock;
    0 referências | Kenerry Pierre, há 19 dias | 1 autor, 1 alteração | 0 exceções
    public PlayerControllerShould()
    {
        _playerApplicationServiceMock = new Mock<IPlayerApplicationService>();
        _playerController = new PlayerController(_playerApplicationServiceMock.Object);
    }

    [Theory]
    [InlineData(0)]
    [InlineData(1)]
    | 0 referências | Kenerry Pierre, há 19 dias | 1 autor, 1 alteração | 0 exceções
    public async Task Return_NotFound_When_NotFind_Player(int playerId)
    {
        _playerApplicationServiceMock.Setup(playerApp => playerApp.GetByIdAsync(playerId))
            .ReturnsAsync(It.IsAny<PlayerResponse>());

        var returned = (NotFoundResult)await _playerController.GetPlayerByIdAsync(playerId);
        Assert.Equal(returned.StatusCode, (int)HttpStatusCode.NotFound);
    }
```

Figura 8-Player Unit Testing

Note alguns padrões. O nome na classe de teste tem sufixo “Should”, Player Controller “Deve”. O nome do método é “Return_NotFound_When_NotFind_Player”, ou seja, estou dizendo que a classe Player Controller deve retornar “Not Found”, quando não encontrar o jogador. Perceba a ênfase na clareza e na legibilidade do código. Não precisa ser sênior para escrever um método bem escrito, muito menos para entende-lo. Este é o tipo de legibilidade que você deve dar ao seu código.

Dica extra: Fique sempre atento para não cair na cilada de

adaptar seu teste para passar no código, mas sim seu código para passar no teste.

Observação: Se por acaso você não souber o que é Moq: Moq é o mock framework mais popular da comunidade .Net. Até o momento da escrita deste livro, possui 33 milhões de downloads do NuGet. Como o nome já descreve, é utilizado para realizar o mockup de métodos. Comumente utilizado em testes unitários.

Testes de Integração

Conforme discorremos nos tópicos acima, o teste de integração, também deve ser um dos testes que compõe a garantia de qualidade do código. Diferentemente do teste unitário, onde a premissa é testar uma única unidade de código, neste teste o conceito é justamente ao contrário, testar múltiplas unidades de código, ao mesmo tempo. O teste integrado, visa garantir não só o funcionamento da regra de negócio, mas o fluxo completo, após unir as “caixinhas”, para tanto, para que se aproxime da realidade, o máximo possível, o ideal é que nos testes integrados o acesse o banco de dados esteja disponível, veja abaixo o exemplo, utilizando Entity Framework Core:

```
[Theory]
[InlineData("Barcelona")]
0 referências | Kenerry Pierre, há 17 dias | 1 autor, 1 alteração | 0 exceções
public async Task Return_Team_After_Insert(string teamName)
{
    var teamEntity = new Team(teamName);
    await _teamRepository.AddAsync(teamEntity);
    await _teamRepository.SaveChangesAsync();
    Assert.NotEqual(0, teamEntity.Id);

    var databaseTeamEntity = await _teamRepository.GetByIdAsync(teamEntity.Id);
    Assert.Equal(databaseTeamEntity.Id, teamEntity.Id);
    Assert.Equal(databaseTeamEntity.Name, teamEntity.Name);
}
```

Figura 9-Team Unit Testing

Final

Espero que você tenha gostado deste livro. Espero que, de alguma forma, ele tenha te ajudado. Se você aprendeu uma coisa com o que passei para você, uma única coisa se quer, com certeza isto já valeu a pena para mim. Explore a arquitetura, veja alguns tópicos implementados não abordados aqui, explicitamente. Entenda como as coisas estão funcionando, questione; esta é a mensagem principal. Um ponto importante: não deixe de contribuir e compartilhar com outras pessoas o que você aprendeu! Até uma próxima!