Teste ASP.NET Core XUnit

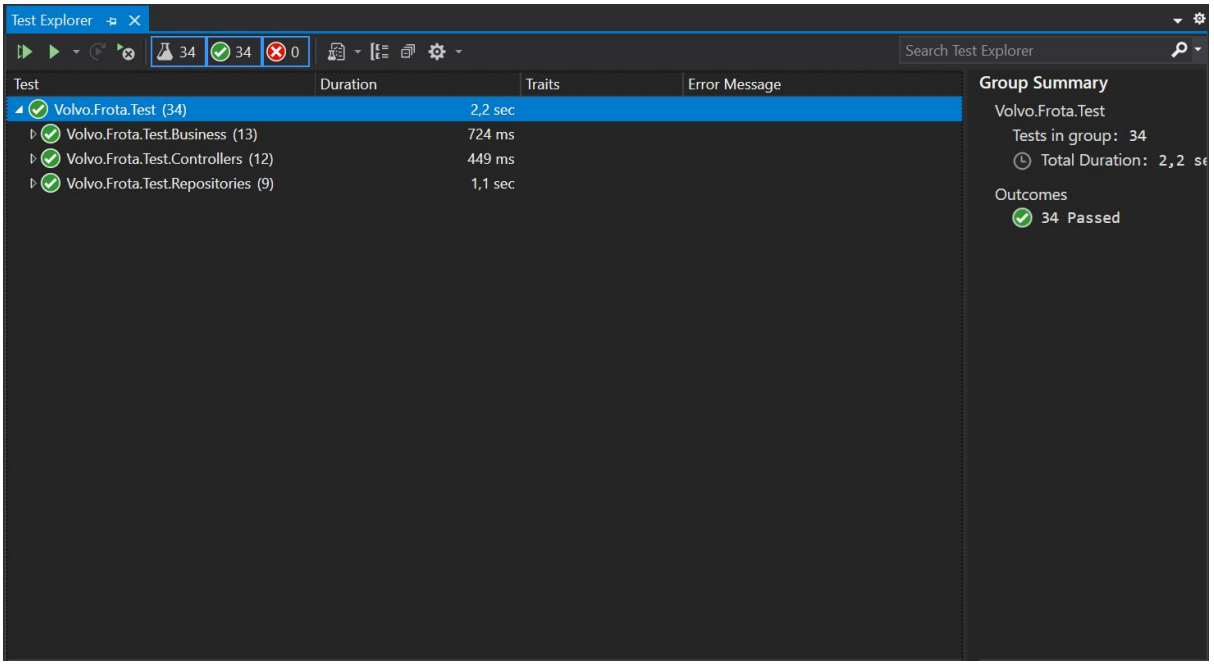
Wagner de Almeida Santos

São Paulo, 24 de Dezembro de 2020

Sumário

Introdução

A aplicação conta com um projeto de teste utilizando XUnit. Dentro deste contexto, foram realizados testes unitários em repositórios, classes de Negócios e Controllers. Cada camada de teste está separada, para visualizar a separação basta acessar em View e selecionar opção Test Explorer, onde é possível visualizar todos os testes conforme imagem abaixo.



Execução

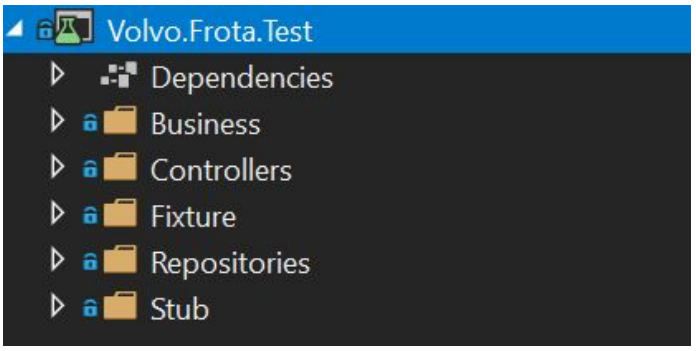
A execução dos testes é feita através do ícone de execução ou pelo comando (Ctrl + R, T). Durante o processo de desenvolvimento foram utilizadas algumas abordagens como Stub e Fixture para facilitar a realização dos testes. Além disso, testes feitos no repositório e controllers são realizados com banco em memória, um recurso que podemos utilizar com o EntityFramework.

Abstraindo as configurações em uma startup própria para teste, foram colocadas as configurações para o projeto teste, como as configurações para utilizar o banco em memória.

```
1 reference
public IConfiguration Configuration { get; }
0 references
public StartupTest(IConfiguration configuration)
{
    Configuration = configuration;
}

0 references
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddDbContext<VolvoContext>(options =>
    {
        options.UseInMemoryDatabase("integrationTests");
    });

    services.RegisterDependencyInjection();
}
```

Arquitetura dos Testes

A parte de organização do código está de acordo com as camadas que a aplicação tem, então cada parte está separada de acordo com a sua responsabilidade, além de outras configurações para facilitar o teste. Abaixo temos a demonstração de como está a arquitetura.



Para facilitar o reuso e evitar a duplicação de código, foi utilizado o conceito de Stub, onde são criados objetos para cada cenário, seja ele feito na camada de negócio ou na Controller. Com isso temos centralizado em um único ponto as configurações necessárias para criar os objetos, além de facilitar para manutenções futuras tendo em vista que as mudanças precisam ser feitas em um único ponto do código.

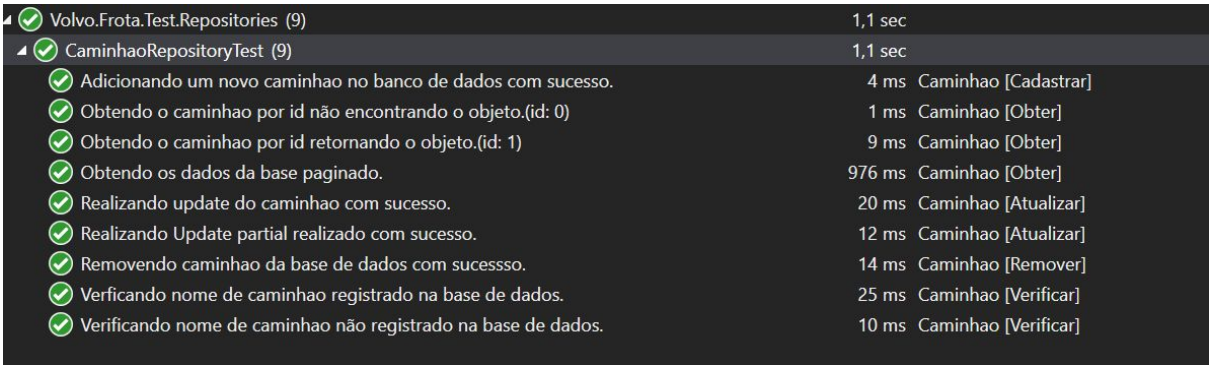
Repositório

Os testes realizados no repositório são feitos realizando o InMemoryDatabase, com esse recurso é possível testar o comportamento de escrita e leitura na base de dados. Como o repositório não faz acesso direto ao contexto do EntityFramework Core, uma classe Entities foi utilizada para acessar o contexto e disponibilizar os métodos para o repositório.

```
2 references
private VolvoContext _context { get; set; }
17 references | 8/8 passing
public Entities Entities { get; set; }

0 references
public RepositoryTest()
{
    this._context = CreateContext();
    this.Entities = new Entities(this._context);
}

1 reference
private VolvoContext CreateContext()
{
    var options = new DbContextOptionsBuilder<VolvoContext>()
        .UseInMemoryDatabase(Guid.NewGuid().ToString())
        .Options;

    return new VolvoContext(options);
}
```

Na imagem acima, está a configuração da classe RepositoryTest que irá disponibilizar todo o setup necessário para realizar os testes no repositório.

Para executar somente os teste do repositório, acesse Test Explorer e selecione a opção Volvo.Frota.Test.Repositories com o botão direito do mouse e clique em run, com isso todos os testes do repositório serão executados e ao final da execução retornar se o teste foi realizado com sucesso ou se teve falha.

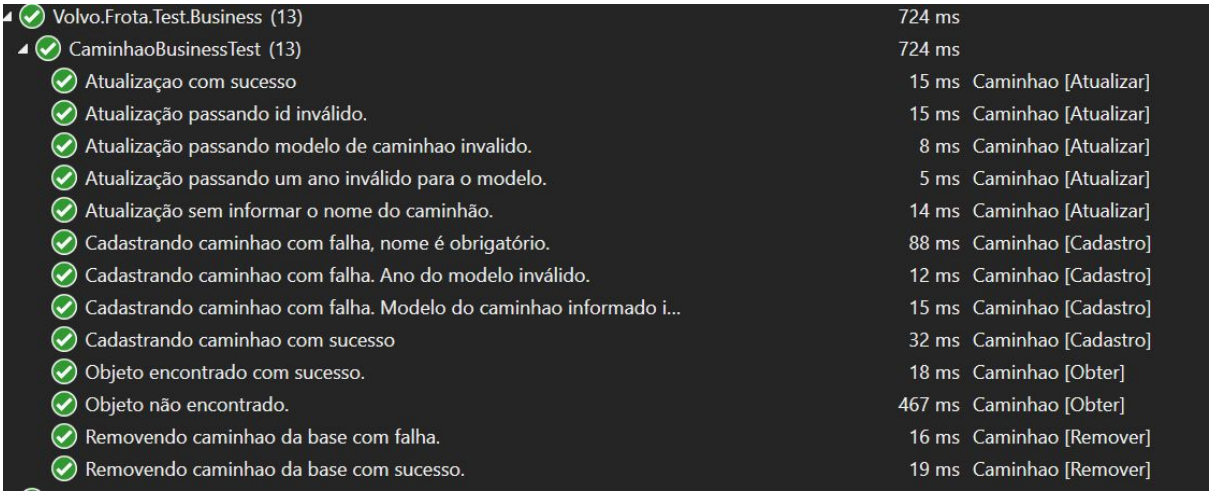


Business

Os testes realizados na camada de negócio têm algumas particularidades a mais, nesta camada foi utilizado o Faker, onde podemos simular alguns dados, como CPF, CNPJ, etc. Além do Faker, também utilizamos o conceito de Mock, para realizar o mockup dos objetos e comparar com o resultado retornar usando o serviço da camada de negócio. Para abstrair configurações como do AutoMapper, foi criado uma classe base, onde todas as configurações para camada de negócio ficassem centralizadas em um único ponto e que permita reutilizar para futuros testes.

```
1 reference
public class BusinessTest
{
    private IMapper _mapper;

    1 reference
    public IMapper Mapper
    {
        get
        {
            if(_mapper == null)
            {
                _mapper = new Mapper(AutoMapperUtils.GetConfigurationMappings());
            }

            return _mapper;
        }
    }
}
```

Para executar somente os teste na camada de negócio, acesse Test Explorer e selecione a opção Volvo.Frota.Test.Business com o botão direito do mouse e clique em run, com isso todos os testes de negócio serão executados e ao final da execução retornar se o teste foi realizado com sucesso ou se teve falha.



Controller

Os testes realizados na controller usam o conceito de ``fixture", onde é realizado a abstração das configurações em uma classe específica para isso. Além de abstrair o uso da fixture é feito a através da injeção de dependência, utilizando os recursos que o XUnit disponibiliza para realizar essa abordagem.

```
public class IntegrationFixture<TStartup> : IDisposable where TStartup : class
{
    public HttpClient HttpClient;
    private readonly ApiAppFactory<TStartup> _factory;

    0 references
    public IntegrationFixture()
    {
        var clientOptions = new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = true,
            BaseAddress = new Uri("http://localhost:62837")
        };

        _factory = new ApiAppFactory<TStartup>();
        HttpClient = _factory.CreateClient(clientOptions);
    }

    0 references
    public void Dispose()
    {
        HttpClient.Dispose();
        _factory.Dispose();
    }
}
```

A imagem acima ilustra a configuração da Fixture base, onde será utilizado uma configuração focada nos testes, como o banco em memória. Além de uma classe Factory, onde podemos indicar o ambiente que será utilizado para esse cenário.

```
[CollectionDefinition(nameof(CaminhaoFixtureCollection))]
2 references
public class CaminhaoFixtureCollection : ICollectionFixture<CaminhaoTestFixture> { }
4 references
public class CaminhaoTestFixture : IntegrationFixture<StartupTest>
{
    0 references
    public CaminhaoTestFixture()
    {}
}
```

Abstraindo as configurações na IntegrationFixture, podemos criar as configurações de outras fixture específicas e herdar o comportamento que se repete para as demais.
Abaixo segue um exemplo utilizando a fixture específica para o cenário, onde o mesmo é utilizado usando via injeção de dependência.

```
[Collection(nameof(CaminhaoFixtureCollection))]
1 reference
public class CaminhaoControllerTest
{
    private readonly CaminhaoTestFixture _caminhaoTestFixture;
    0 references
    public CaminhaoControllerTest(CaminhaoTestFixture caminhaoTestFixture)
    {
        _caminhaoTestFixture = caminhaoTestFixture;
    }

    [Fact(DisplayName = "Cadastro realizado com sucesso.")]
    [Trait("Caminhao", "cadastrar")]
    0 references
    public async Task Post_CadastrarCaminhao_CadastroRealizadoComSucessoDeveRetornarCreated()
    {
        //ARRANGE
        var stringContent = ObterDado(NewCaminhaoDtoStub.CompletoModeloCaminhaoFH());

        //ACT
        var response = await _caminhaoTestFixture.HttpClient.PostAsync("api/caminhoes", stringContent);

        //ASSERT
        Assert.Equal(HttpStatusCode.Created, response.StatusCode);
    }
```

Para executar somente os teste da Controller, acesse Test Explorer e selecione a opção Volvo.Frota.Test.Controllers com o botão direito do mouse e clique em run, com isso todos os testes da Controller serão executados e ao final da execução retornar se o teste foi realizado com sucesso ou se teve falha.