Testes de integração em .NET com NUnit

No mundo acelerado do desenvolvimento de software, a qualidade e a eficiência são duas metas que todos almejam alcançar. Uma das ferramentas mais eficazes para atingir esses objetivos é a implementação de testes de integração dentro do ciclo de vida do desenvolvimento de software.

Os testes de integração permitem que os desenvolvedores verifiquem se diferentes módulos de um sistema se comunicam de forma correta. Eles ajudam a identificar e corrigir bugs, melhorar a qualidade do código e garantir que o software funcione como esperado quando entregue ao usuário final.

Neste artigo, iremos explorar a implementação de testes de integração, utilizando a ferramenta N-Unit, para validar a interação entre uma API desenvolvida em .Net Core e um banco de dados SQL Server. O foco principal será demonstrar como os

testes de integração podem assegurar a funcionalidade correta da API em conjunto com o banco de dados.

Para ilustrar isso de maneira prática, apresentaremos um projeto no qual foi desenvolvido um sistema CRUD (Create, Read, Update, Delete) para a criação de veículos. Este sistema servirá como nosso caso de estudo, permitindo-nos examinar em detalhes como os testes de integração podem ser efetivamente aplicados em um cenário do mundo real.

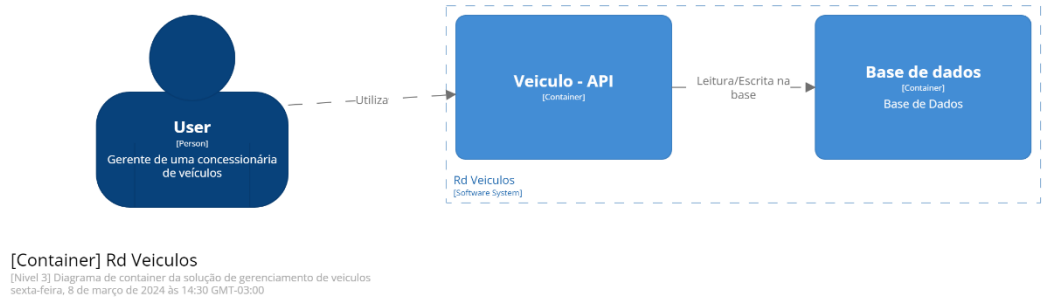
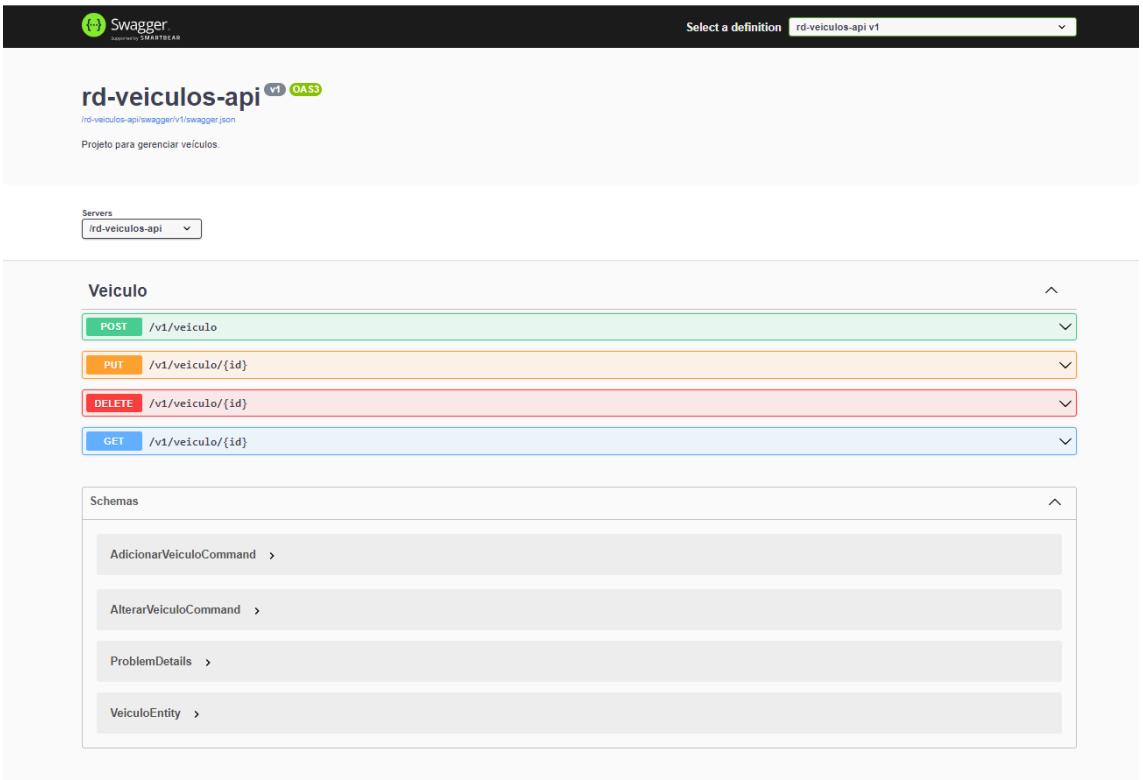


Diagrama de container da aplicação

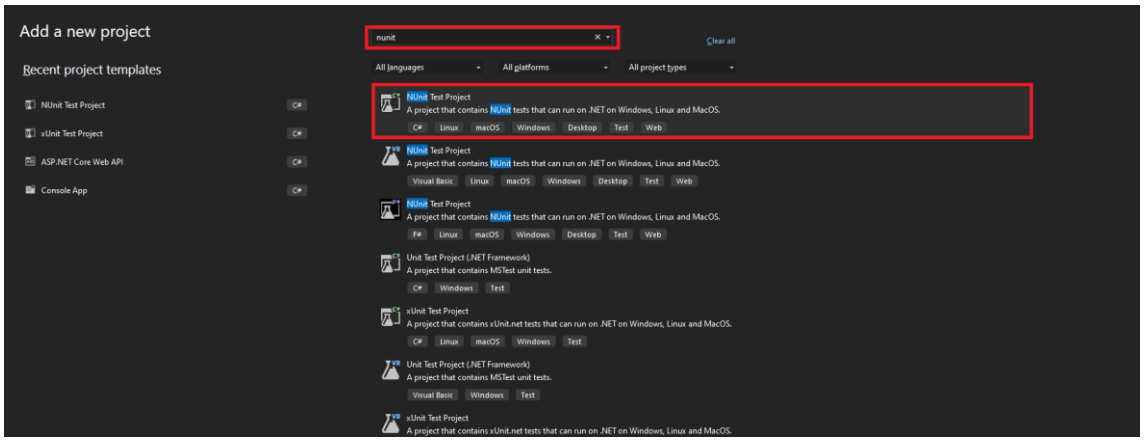


Swagger da API com as rotas disponíveis

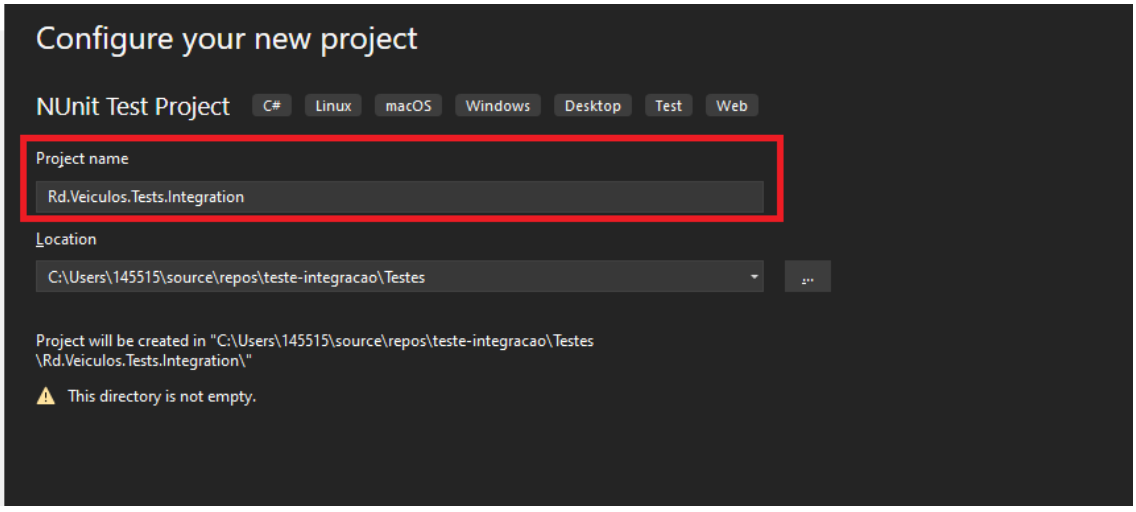
Ao longo do artigo, discutiremos as melhores práticas, desafios comuns e soluções eficazes no contexto dos testes de integração. Nosso objetivo é fornecer aos leitores uma compreensão clara e aplicável de como implementar testes de integração de maneira eficaz em seus próprios projetos de desenvolvimento de software.

O primeiro passo é estabelecer a criação de um novo projeto NUnit. É recomendável que este projeto seja alocado dentro de uma nova solução. Isso facilitará sua inclusão em pipelines automatizadas, simplificando assim o processo de integração contínua.

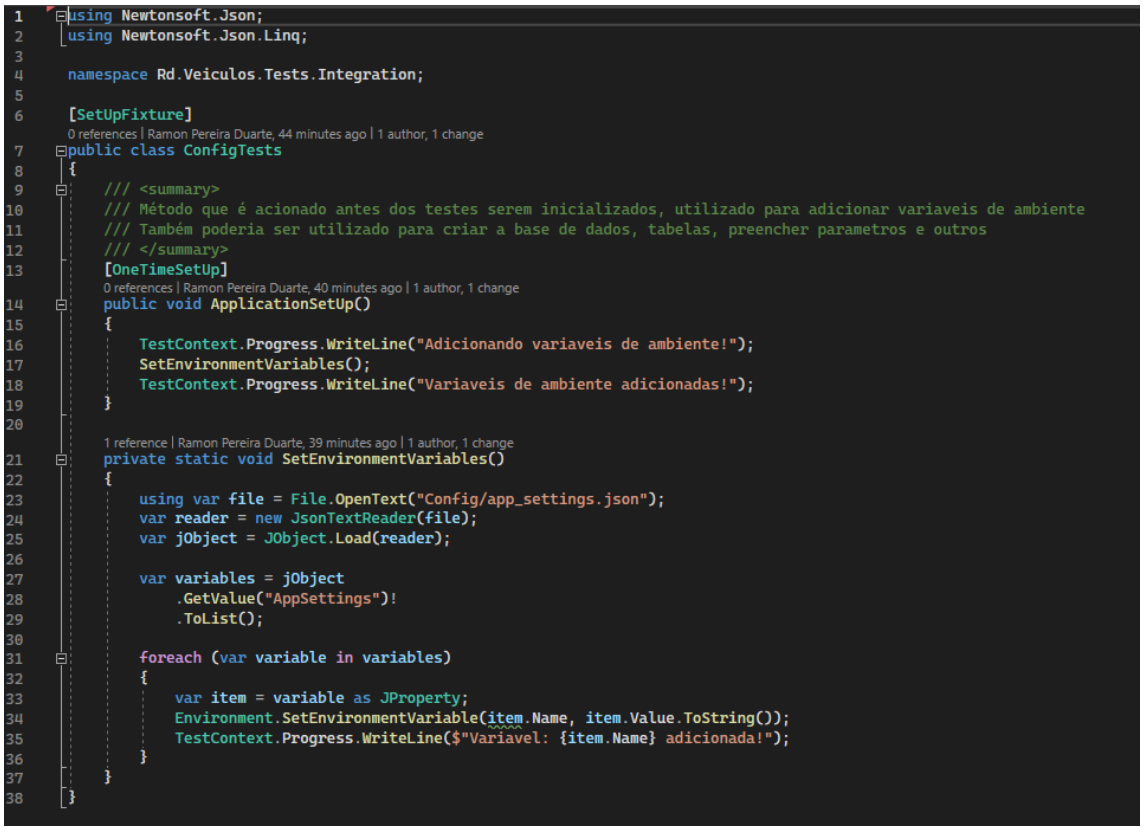
1 – Crie um projeto



2 – Defina o nome do seu novo projeto, neste caso foi definido como *“Rd.Veiculos.Tests.Integration”*



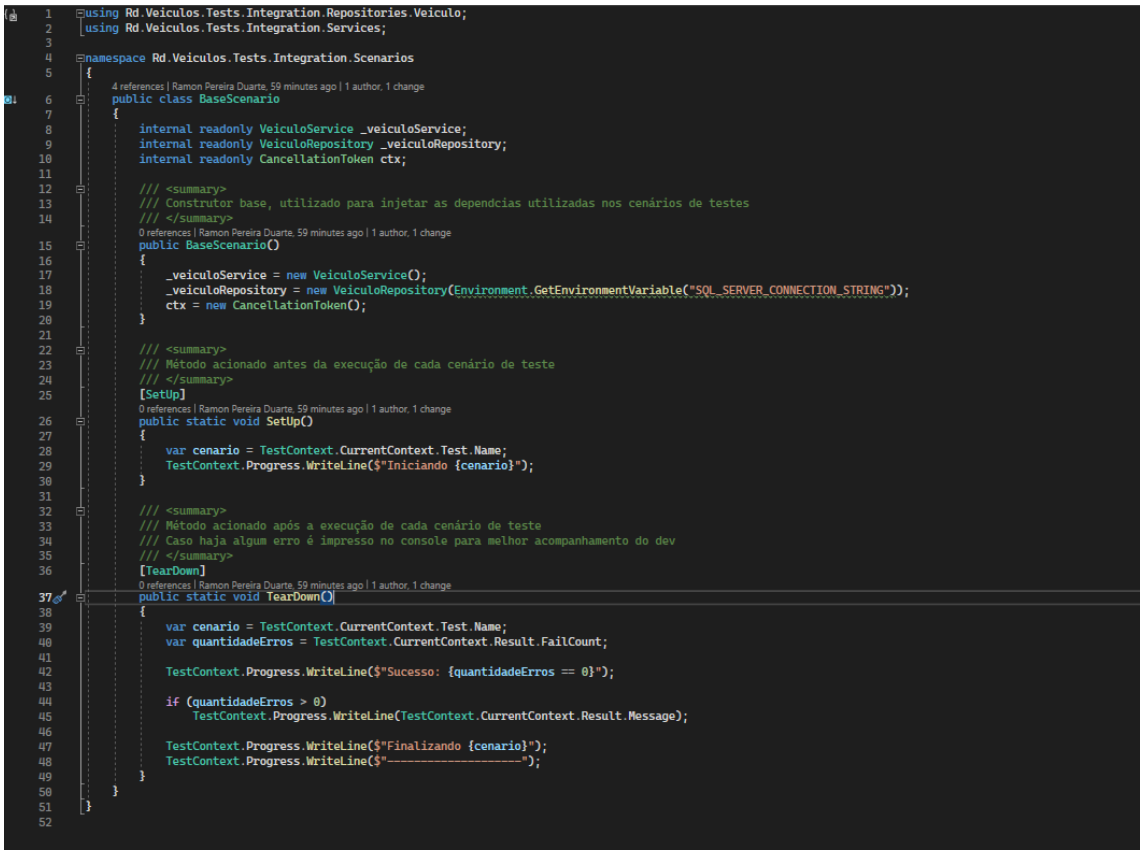
Com o projeto vazio, vamos criar um arquivo de configuração de NUnit para configurar todas as dependências dos nossos cenários de teste.



```
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

namespace Rd.Veiculos.Tests.Integration;

[SetUpFixture]
0 references | Ramon Pereira Duarte, 44 minutes ago | 1 author, 1 change
public class ConfigTests
{
    /// <summary>
    /// Método que é acionado antes dos testes serem inicializados, utilizado para adicionar variaveis de ambiente
    /// Também poderia ser utilizado para criar a base de dados, tabelas, preencher parametros e outros
    /// </summary>
    [OneTimeSetUp]
    0 references | Ramon Pereira Duarte, 40 minutes ago | 1 author, 1 change
    public void ApplicationSetUp()
    {
        TestContext.Progress.WriteLine("Adicionando variaveis de ambiente!");
        SetEnvironmentVariables();
        TestContext.Progress.WriteLine("Variaveis de ambiente adicionadas!");
    }

    1 reference | Ramon Pereira Duarte, 39 minutes ago | 1 author, 1 change
    private static void SetEnvironmentVariables()
    {
        using var file = File.OpenText("Config/app_settings.json");
        var reader = new JsonTextReader(file);
        var jObject = JObject.Load(reader);

        var variables = jObject
            .GetValue("AppSettings")!
            .ToList();

        foreach (var variable in variables)
        {
            var item = variable as JProperty;
            Environment.SetEnvironmentVariable(item.Name, item.Value.ToString());
            TestContext.Progress.WriteLine($"Variavel: {item.Name} adicionada!");
        }
    }
}
```

O método OneTimeSetup do NUnit é invocado antes da execução dos cenários de testes. Este método é particularmente útil para a configuração inicial necessária para os testes, como a adição de variáveis de ambiente, a inicialização da base de dados, a criação de tabelas, entre outros preparativos essenciais.

Após a criação da configuração do projeto, vamos criar um arquivo de cenário base para melhor acompanhamento da execução de nossos cenários de testes, e injeção das dependências que utilizaremos em nossos cenários de teste.

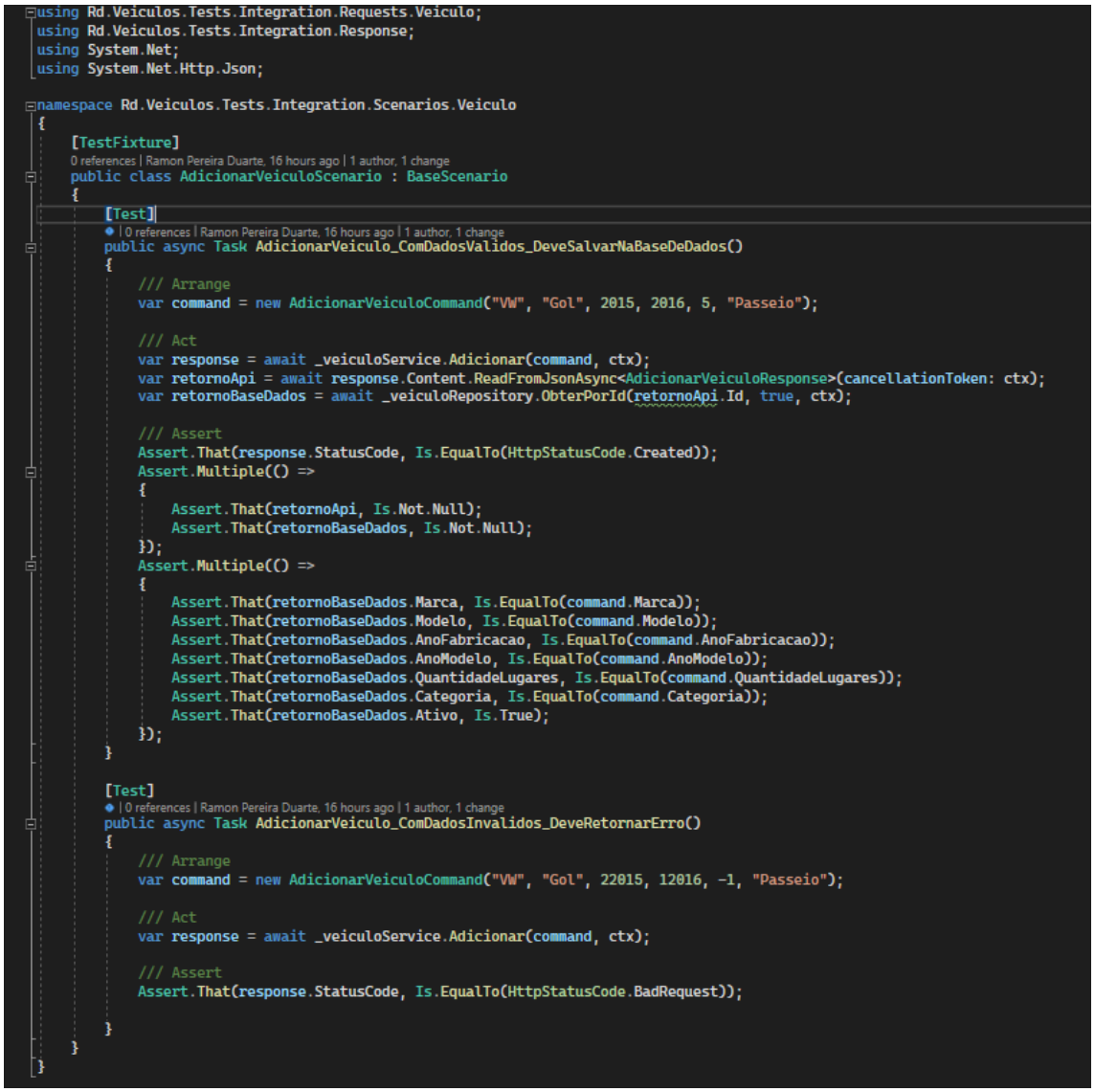


```
using Rd.Veiculos.Tests.Integration.Repositories.Veiculo;
using Rd.Veiculos.Tests.Integration.Services;

namespace Rd.Veiculos.Tests.Integration.Scenarios
{
    4 references | Ramon Pereira Duarte, 59 minutes ago | 1 author, 1 change
    public class BaseScenario
    {
        internal readonly VeiculoService _veiculoService;
        internal readonly VeiculoRepository _veiculoRepository;
        internal readonly CancellationToken ctx;

        /// <summary>
        /// Construtor base, utilizado para injetar as dependcias utilizadas nos cenários de testes
        /// </summary>
        0 references | Ramon Pereira Duarte, 59 minutes ago | 1 author, 1 change
        public BaseScenario()
        {
            _veiculoService = new VeiculoService();
            _veiculoRepository = new VeiculoRepository(Environment.GetEnvironmentVariable("SQL_SERVER_CONNECTION_STRING"));
            ctx = new CancellationToken();
        }

        /// <summary>
        /// Método acionado antes da execução de cada cenário de teste
        /// </summary>
        [SetUp]
        0 references | Ramon Pereira Duarte, 59 minutes ago | 1 author, 1 change
        public static void SetUp()
        {
            var cenario = TestContext.CurrentContext.Test.Name;
            TestContext.Progress.WriteLine($"Iniciando {cenario}");
        }

        /// <summary>
        /// Método acionado após a execução de cada cenário de teste
        /// Caso haja algum erro é impresso no console para melhor acompanhamento do dev
        /// </summary>
        [TearDown]
        0 references | Ramon Pereira Duarte, 59 minutes ago | 1 author, 1 change
        public static void TearDown()
        {
            var cenario = TestContext.CurrentContext.Test.Name;
            var quantidadeErros = TestContext.CurrentContext.Result.FailCount;

            TestContext.Progress.WriteLine($"Sucesso: {quantidadeErros == 0}");

            if (quantidadeErros > 0)
                TestContext.Progress.WriteLine(TestContext.CurrentContext.Result.Message);

            TestContext.Progress.WriteLine($"Finalizando {cenario}");
            TestContext.Progress.WriteLine($"---------------------");
        }
    }
}
```

Cenário de Adição de veículo

Após finalizarmos todas as configurações do nosso projeto de testes, é hora de nos concentrarmos na criação dos cenários de teste. Nesse contexto, validaremos tanto o componente da nossa API quanto o componente do banco de dados. Vamos começar criando um cenário específico: a adição de novos veículos à base de dados.

Para esse cenário, iremos realizar a validação de dois casos de teste, uma para sucesso no processamento da requisição e outro um cenário de falha. Esses testes são essenciais para garantir a robustez e a confiabilidade do nosso sistema. Vamos em frente e criar esses cenários! 🚗🔍



```
using Rd.Veiculos.Tests.Integration.Requests.Veiculo;
using Rd.Veiculos.Tests.Integration.Response;
using System.Net;
using System.Net.Http.Json;

namespace Rd.Veiculos.Tests.Integration.Scenarios.Veiculo
{
    [TestFixture]
    0 references | Ramon Pereira Duarte, 16 hours ago | 1 author, 1 change
    public class AdicionarVeiculoScenario : BaseScenario
    {
        [Test]
        ◆ | 0 references | Ramon Pereira Duarte, 16 hours ago | 1 author, 1 change
        public async Task AdicionarVeiculo_ComDadosValidos_DeveSalvarNaBaseDeDados()
        {
            /// Arrange
            var command = new AdicionarVeiculoCommand("VW", "Gol", 2015, 2016, 5, "Passeio");

            /// Act
            var response = await _veiculoService.Adicionar(command, ctx);
            var retornoApi = await response.Content.ReadFromJsonAsync<AdicionarVeiculoResponse>(cancellationToken: ctx);
            var retornoBaseDados = await _veiculoRepository.ObterPorId(retornoApi.Id, true, ctx);

            /// Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.Created));
            Assert.Multiple(() =>
            {
                Assert.That(retornoApi, Is.Not.Null);
                Assert.That(retornoBaseDados, Is.Not.Null);
            });
            Assert.Multiple(() =>
            {
                Assert.That(retornoBaseDados.Marca, Is.EqualTo(command.Marca));
                Assert.That(retornoBaseDados.Modelo, Is.EqualTo(command.Modelo));
                Assert.That(retornoBaseDados.AnoFabricacao, Is.EqualTo(command.AnoFabricacao));
                Assert.That(retornoBaseDados.AnoModelo, Is.EqualTo(command.AnoModelo));
                Assert.That(retornoBaseDados.QuantidadeLugares, Is.EqualTo(command.QuantidadeLugares));
                Assert.That(retornoBaseDados.Categoria, Is.EqualTo(command.Categoria));
                Assert.That(retornoBaseDados.Ativo, Is.True);
            });
        }

        [Test]
        ◆ | 0 references | Ramon Pereira Duarte, 16 hours ago | 1 author, 1 change
        public async Task AdicionarVeiculo_ComDadosInvalidos_DeveRetornarErro()
        {
            /// Arrange
            var command = new AdicionarVeiculoCommand("VW", "Gol", 22015, 12016, -1, "Passeio");

            /// Act
            var response = await _veiculoService.Adicionar(command, ctx);

            /// Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.BadRequest));

        }
    }
}
```

Começaremos criando a classe AdicionarVeiculoScenario, que será responsável por testar o cenário de adição de um novo veículo. Essa classe herdará da classe base que criamos no passo anterior.

A classe AdicionarVeiculoScenario será marcada com o atributo [TestFixture], que é usado no NUnit para identificar classes que contêm testes. Além disso, essa classe pode conter métodos de configuração ou limpeza, caso necessário.

Teste de Sucesso

No cenário de teste de sucesso, nosso objetivo é criar um comando para adicionar um novo veículo. Após chamar a nossa API com esse comando, faremos uma consulta à nossa base de dados para validar se os dados informados no comando

foram corretamente persistidos. Além disso, verificaremos o status code da requisição feita à API. Esperamos que o status code seja Created, conforme o padrão REST para a criação de um novo recurso.

Teste de Falha

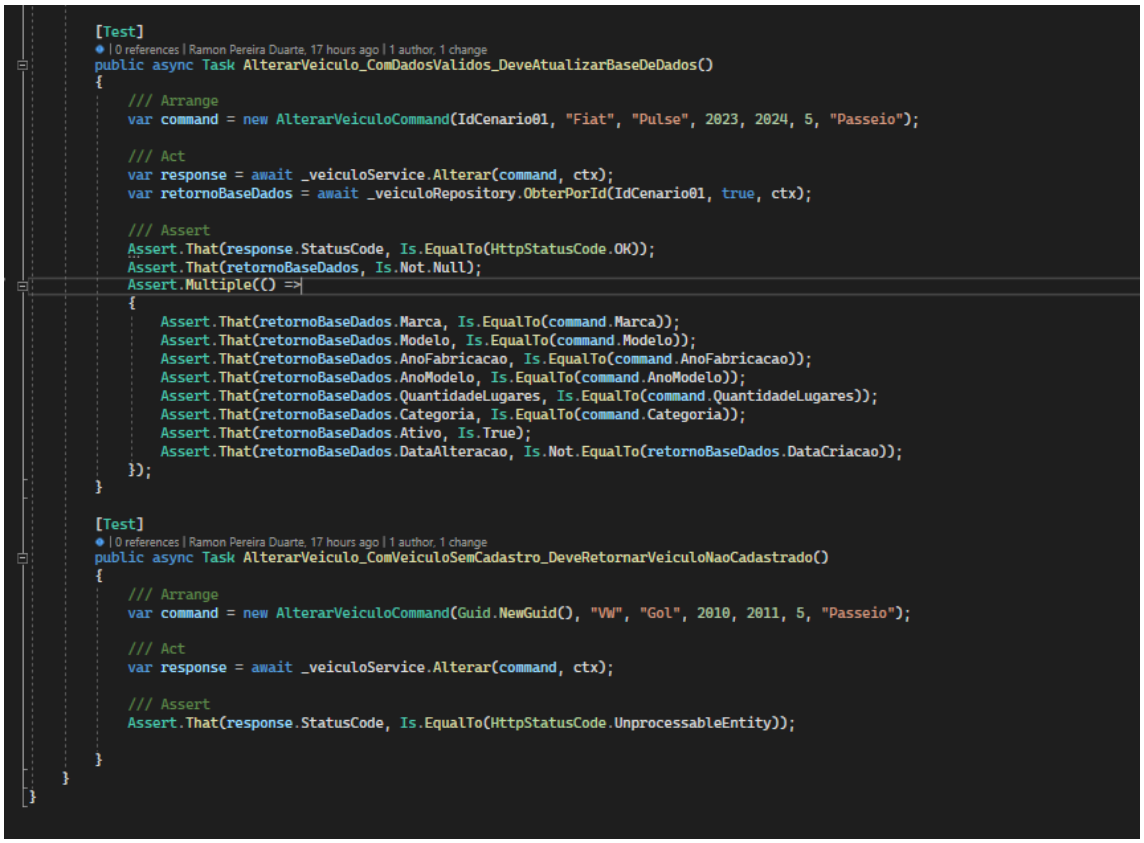
No cenário de teste de falha, iremos criar um comando com dados inválidos e esperamos que nossa API não processe esses dados. O resultado esperado é um erro de má requisição (BadRequest). Esse teste nos ajudará a garantir que nossa API está tratando adequadamente entradas inválidas.

Cenário de Alteração de um veículo

Vamos agora criar cenários para a alteração de veículos. Esse é um cenário mais complexo, pois antes de realizar a alteração de um veículo, precisamos cadastrá-lo em nossa base de dados. Para situações como essa, o NUnit oferece dois métodos que executam ações antes e/ou depois da inicialização dos testes, mas apenas uma vez: OneTimeSetup e OneTimeTearDown. O método OneTimeSetup é invocado uma única vez antes do início dos testes, enquanto o OneTimeTearDown é utilizado após a conclusão dos testes. No nosso caso, utilizaremos o OneTimeSetup para criar um veículo antes de executar o teste de alteração.

```
using System.Net.Http.Json;

namespace Rd.Veiculos.Tests.Integration.Scenarios.Veiculo
{

    [TestFixture]
    0 references | Ramon Pereira Duarte, 17 hours ago | 1 author, 1 change
    public class AlterarVeiculoScenario : BaseScenario
    {
        3 references | 0/1 passing | Ramon Pereira Duarte, 17 hours ago | 1 author, 1 change
        public Guid IdCenario01 { get; set; }

        /// <summary>
        /// Método acionado antes da inicializacao dos cenários de testes
        /// Utilizado para preparar os dados para a execução dos cenários de testes sem depêndencias com outros cenários
        /// É realizado a adicao de um veiculo, para posteriormente a alteração do mesmo
        /// </summary>
        /// <returns></returns>
        [OneTimeSetUp]
        0 references | Ramon Pereira Duarte, 17 hours ago | 1 author, 1 change
        public async Task PrepararDadosDoCenario()
        {
            TestContext.Progress.WriteLine($"Adicionando dependencias do cenarios de alteracao");
            var command = new AdicionarVeiculoCommand("Alterar", "Cenario01", 2024, 2024, 5, "Passeio");
            var response = await _veiculoService.Adicionar(command, ctx);
            var retornoApi = await response.Content.ReadFromJsonAsync<AdicionarVeiculoResponse>(cancellationToken: ctx);
            IdCenario01 = retornoApi.Id;
            TestContext.Progress.WriteLine($"Finalizando dependencias dos cenarios de alteracao");
            TestContext.Progress.WriteLine($"--------------------");
        }
```

Após configurarmos as dependências dos testes, prosseguiremos com a criação dos cenários de sucesso e falha, seguindo o mesmo padrão do cenário anterior. No cenário de falha, simularemos a alteração de um veículo que **não está cadastrado em nossa base de dados** e verificaremos como o sistema se comporta.



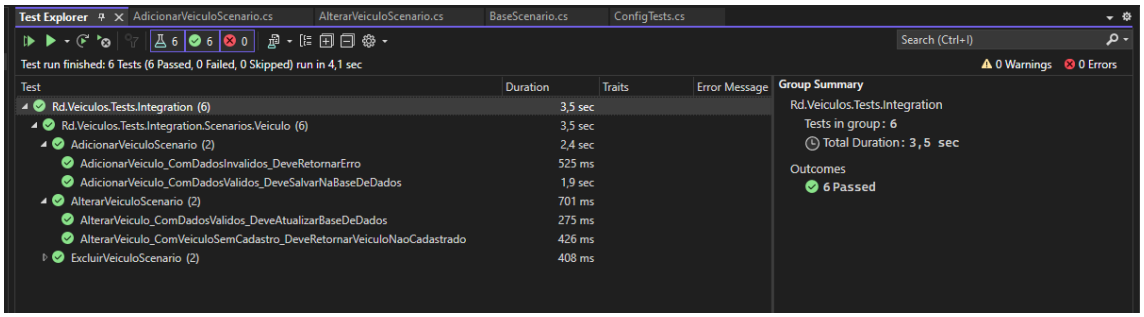
```
        [Test]
        ● | 0 references | Ramon Pereira Duarte, 17 hours ago | 1 author, 1 change
        public async Task AlterarVeiculo_ComDadosValidos_DeveAtualizarBaseDeDados()
        {
            /// Arrange
            var command = new AlterarVeiculoCommand(IdCenario01, "Fiat", "Pulse", 2023, 2024, 5, "Passeio");

            /// Act
            var response = await _veiculoService.Alterar(command, ctx);
            var retornoBaseDados = await _veiculoRepository.ObterPorId(IdCenario01, true, ctx);

            /// Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.OK));
            Assert.That(retornoBaseDados, Is.Not.Null);
            Assert.Multiple(() =>
            {
                Assert.That(retornoBaseDados.Marca, Is.EqualTo(command.Marca));
                Assert.That(retornoBaseDados.Modelo, Is.EqualTo(command.Modelo));
                Assert.That(retornoBaseDados.AnoFabricacao, Is.EqualTo(command.AnoFabricacao));
                Assert.That(retornoBaseDados.AnoModelo, Is.EqualTo(command.AnoModelo));
                Assert.That(retornoBaseDados.QuantidadeLugares, Is.EqualTo(command.QuantidadeLugares));
                Assert.That(retornoBaseDados.Categoria, Is.EqualTo(command.Categoria));
                Assert.That(retornoBaseDados.Ativo, Is.True);
                Assert.That(retornoBaseDados.DataAlteracao, Is.Not.EqualTo(retornoBaseDados.DataCriacao));
            });
        }

        [Test]
        ● | 0 references | Ramon Pereira Duarte, 17 hours ago | 1 author, 1 change
        public async Task AlterarVeiculo_ComVeiculoSemCadastro_DeveRetornarVeiculoNaoCadastrado()
        {
            /// Arrange
            var command = new AlterarVeiculoCommand(Guid.NewGuid(), "VW", "Gol", 2010, 2011, 5, "Passeio");

            /// Act
            var response = await _veiculoService.Alterar(command, ctx);

            /// Assert
            Assert.That(response.StatusCode, Is.EqualTo(HttpStatusCode.UnprocessableEntity));

        }
    }
}
```

O cenário de exclusão de veículos também está presente dentro do repositório que iremos deixar disponível mais abaixo, ele é bastante semelhante ao cenário de alteração.

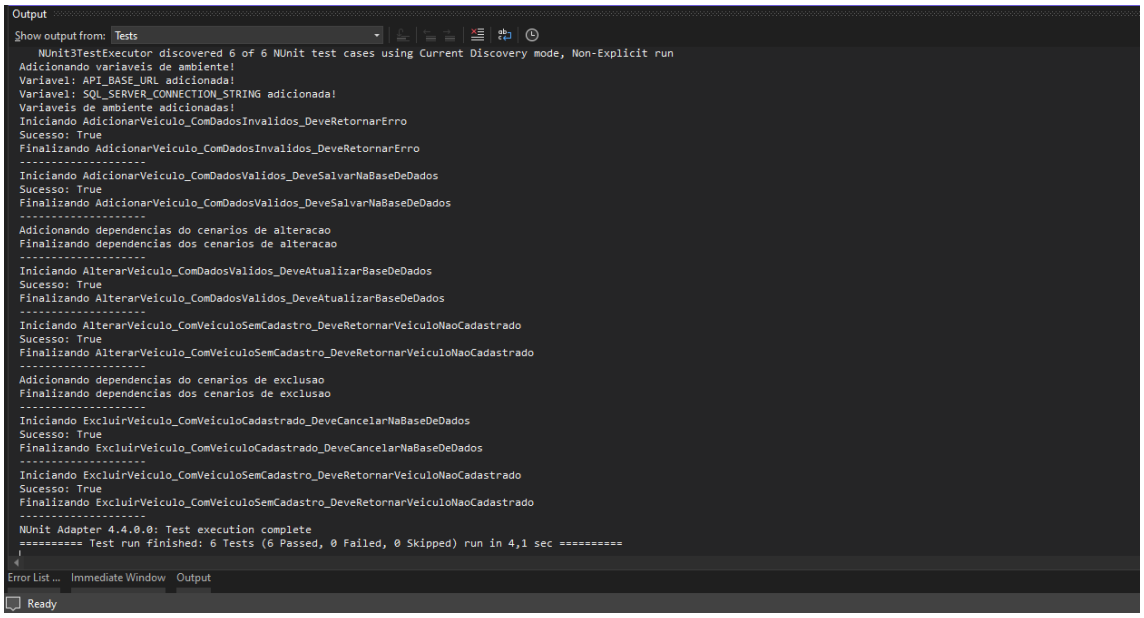
Validação dos testes

Após a preparação dos cenários de testes, é hora de validá-los. Para essa etapa, utilizaremos a ferramenta Test Explorer, disponível dentro do Visual Studio. Nela, acionaremos a execução de todos os cenários de testes. O resultado esperado é que todos os cenários sejam marcados como bem-sucedidos após a execução.

Lembre-se de deixar a API em execução em uma outra janela do seu Visual Studio. Na imagem abaixo, podemos observar que todos os cenários foram executados com sucesso, validando a execução da nossa API e a comunicação com o banco de dados.



Na janela de saída dos logs de execução dos testes, é possível acompanhar todos os outputs gerados nos nossos cenários de teste.



Conclusão

Em resumo, neste artigo exploramos a criação de testes de integração em .NET usando a ferramenta NUnit. Focamos na validação de uma API e de um banco de dados, abordando a configuração do NUnit e a criação dos testes de integração. Espero que o conteúdo tenha sido útil para os leitores interessados em aprimorar suas habilidades de teste e garantir a qualidade de suas aplicações.

Para acessar todos os arquivos da API e os testes de integração, confira o repositório no GitHub: https://github.com/ramonpduart/Rd.Veiculos/ .

Se tiver alguma dúvida ou feedback, não hesite em compartilhar! 🚀

Referências:

NUnit.org

Teste de unidade em C# com NUnit e .NET Core - .NET | Microsoft Learn