PEDRO HENRIQUE COSTA DA SILVA MOREIRA

DESENVOLVIMENTO DE WEB SERVICES UTILIZANDO O FRAMEWORK .NET CORE

2020

PEDRO HENRIQUE COSTA DA SILVA MOREIRA

DESENVOLVIMENTO DE WEB SERVICES UTILIZANDO O FRAMEWORK .NET CORE

Relatório de estágio supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. Neumar Costa Malheiros
Orientador

2020

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Moreira, Pedro Henrique Costa da Silva

Desenvolvimento de Web services utilizando o framework .NET Core / Pedro Henrique Costa da Silva Moreira, Neumar Costa Malheiros. – Lavras : UFLA, 2020.

39 p. : il.

Relatório de Estágio(graduação)–Universidade Federal de Lavras, 2020.

Orientador: Prof. Neumar Costa Malheiros.

Bibliografia.

1. TCC. 2. .NET Core. 3. Web Service. 4. Back-end. I. Malheiros, Neumar Costa II. Título.

CDD-808.066

Dedico à minha família, em especial a minha mãe e minha tia Beth.

AGRADECIMENTOS

Agradeço primeiramente à todos os meus familiares que de alguma forma me ajudaram na minha trajetória de vida. Agradeço à minha namorada, por toda ajuda e influência positiva durante a minha graduação. Agradeço aos meus amigos e colegas de trabalho, que muito me ensinaram, ajudaram, divertiram e me aguentaram durante todo esse tempo. Agradeço à UFLA pelo apoio e por todas as experiências maravilhosas que a universidade me proporcionou. Agradeço à Ioasys, pela oportunidade de colaborar nessa empresa que tenho tanto orgulho de integrar. Agradeço ao Prof. Neumar Malheiros que me orientou e colaborou com a elaboração deste trabalho e como sempre realizou bem mais do que o esperado.

O destino é inexorável.

(Uhtred Uhtredson)

RESUMO

Este projeto teve como objetivo a manutenção e o desenvolvimento de novas funcionalidades em duas APIs para serem utilizadas juntamente com uma aplicação web, implementada com o framework React JS. Estas APIs são componentes de um sistema para gestão de vagas e oportunidades, tanto para pessoas externas à empresa, quanto para pessoas que já trabalham na empresa cliente. Estas vagas seriam criadas pelos recrutadores da empresa, e após isso, seriam disponibilizadas para que pessoas se candidatassem. Os candidatos teriam prazos para envio de documentos, provas, etapas de consulta médica e entre outros (dependendo exclusivamente dos requisitos de cada vaga). O projeto foi implementado utilizando-se o framework .Net Core para criação das APIs, aplicando conceitos de orientação a objetos, estruturas de dados e arquiteturas escaláveis.

Palavras-chave: WEB API. REST. DotNet.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Este relatório têm por finalidade apresentar o desenvolvimento pessoal e profissional e descrever as atividades desempenhadas durante estágio realizado na empresa Innovation Oasys Desenvolvimento de Sistemas Ltda (ioasys). Este estágio permitiu a aplicação de conhecimentos adquiridos nas disciplinas oferecidas pela UFLA no curso de Ciência da Computação.

A principal atuação durante o estágio foi no desenvolvimento de APIs utilizando a plataforma .Net Core, bem como testes unitários para o validação dos códigos criados.

1.1 A ioasys

Fundada em 2012 na cidade de Belo Horizonte - MG, e atualmente com aproximadamente 200 funcionários, a ioasys é uma empresa especializada na busca de soluções digitais inovadores, que implementa aplicações sob demanda para seus clientes, sendo a inovação, design e tecnologia apenas meios que utilizam para trazer o melhor do universo digital para os clientes. Sobre o colaboradores, são formados por um time diverso e múltiplo. O principal recurso da empresa é a produção de aplicações sobre demanda.

A ioasys já realizou trabalhos junto a algumas das maiores empresas do mercado nacional e internacional, tais como: Banco Inter, Burger King, BASF, VR Benefícios, LATAM Airlines, Pfizer, Localiza, Fleury, DMCard, Fundação Dom Cabral e Suvinil.

Este estágio foi realizado no escritório da empresa, localizado na cidade de Lavras, que conta com aproximadamente 20 colaboradores.

1.2 Sobre o projeto

O projeto objeto deste trabalho consiste no desenvolvimento de um sistema para processos seletivos externos e internos. Ademais, foram implementados dois serviços: um para os administradores do sistema e o outro para os candidatos. Os dois serviços foram implementados utilizando-se a linguem de programação C#, e o framework .Net Core.

O serviço administrador é responsável pela criação das vagas que seriam disponibilizadas para os candidatos, assim como o controle e *feedback* para os participantes dos processos seletivos. Para o acesso ao serviço, é necessário uma conta administrativa cadastrada no mesmo, sendo ela criada a partir de outras contas administrativas já cadastradas previamente. Todas as informações sobre as vagas são preenchidas pelos seus responsáveis. Algumas das informações fornecidas acerca das vagas são: nome, salário, horário de trabalho, benefícios, requisitos e etc.

No serviço aos candidatos, é possível visualizar as vagas criadas e disponibilizadas pelos administradores, porém, para a candidatura à vaga, é necessário que o usuário crie uma conta no sistema. Esta conta visa armazenar os dados básicos do possível candidato, criando um acesso único e seguro. Algumas informações presentes no perfil do usuário são: nome, data de nascimento, formação, competências e outros dados básicos. Após o cadastro, o usuário pode se candidatar para uma das vagas disponibilizadas e participar das suas etapas, sendo possível ser reprovado, aprovado ou colocado como banco de talentos. As etapas pelas quais o candidato deve passar variam muito de acordo com a vaga criada, mas algumas possíveis são: entrevista, exame médico, questionário e treinamento.

1.3 Estrutura do documento

No Capítulo 2, é apresentado o conhecimento necessário para compreender os conceitos e tecnologias utilizadas neste projeto. São abordadas, principalmente, as tecnologias relacionadas ao .NET e .Net Core, bem como conceitos básicos sobre *web services*, REST e HTTP.

No Capítulo 3, são descritas as atividades gerais e algumas das tecnologias utilizadas durante o estágio e que são amplamente utilizadas pela comunidade .Net. Alguns tópicos discutidos são: testes unitários, validações de entidades e outros.

Por fim, no Capítulo 4, são apresentadas as conclusões finais, nas quais são realizadas considerações do estágio, local de trabalho e como a experiência deste período contribuiu para a formação do autor.

2 REFERENCIAL TEÓRICO

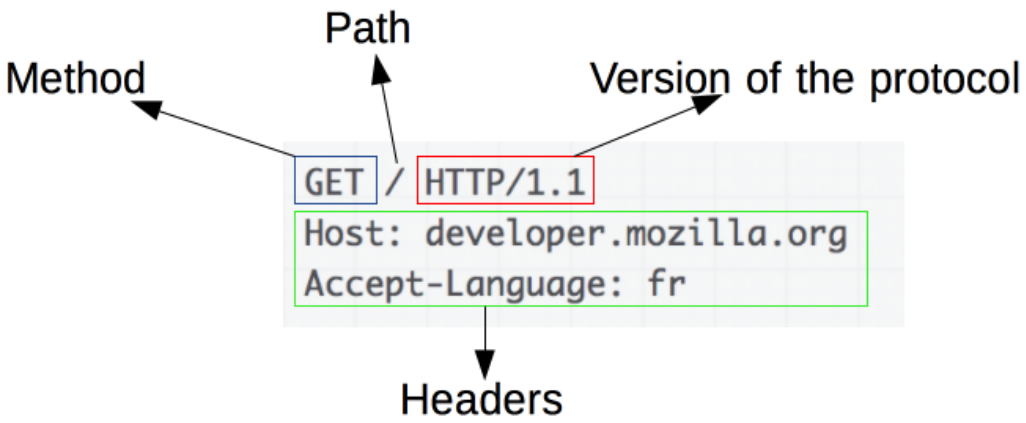
Neste capítulo, são apresentados os principais conceitos e tecnologias relacionados com as atividades desenvolvidas durante o estágio.

2.1 O Protocolo HTTP

O Hypertext Transfer Protocol (HTTP) é um protocolo que permite a obtenção de recursos, tais como documentos HTML, de um servidor. Este protocolo é a base de qualquer troca de dados na Web. É um protocolo cliente-servidor, no qual as requisições são iniciadas pelo cliente, geralmente um navegador da Web, e respondidas pelo servidor. Um documento completo é reconstruído a partir dos diferentes sub-documentos obtidos, por exemplo, texto, descrição do layout, imagens, vídeos, scripts e etc (MOZILLA,).

Uma requisição HTTP é composta dos seguintes elementos: método HTTP, o caminho do recurso a ser buscado, a versão do protocolo HTTP, cabeçalhos e opcionalmente um corpo de dados. Esta estrutura pode ser vista na Figura 2.1.

Figura 2.1 – Exemplo de uma requisição HTTP.



Fonte: https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview

2.1.1 Verbos HTTP

O protocolo de comunicação HTTP define métodos de requisição, que são responsáveis por indicar qual ação deve ser executada. Comumente nomeados de verbos HTTP, eles são: GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE e PATCH. O verbo GET, normalmente, é utilizado para solicitar informações do um recurso. O PUT, usualmente, é definido para substituir todos os dados do recurso por novos dados e o DELETE remove um recurso especificado (FIELDING; RESCHKE, 2014).

2.1.2 Cabeçalhos HTTP

Cabeçalhos HTTP são ferramentas que permitem aos clientes e servidores a passagem de informações padronizadas juntamente com a resposta ou solicitação HTTP.

O formato de um cabeçalho é o seu nome desejado (desconsidera-se diferenças entre letras maiúsculas e minúsculas) seguido do dois pontos (":") e o seu valor. É importante ressaltar que existem os cabeçalhos padrões, porém caso seja necessário, é possível utilizar um cabeçalho customizado que normalmente seu nome inicia-se com "X-". Um exemplo de cabeçalho pode ser o que normalmente contêm as credenciais para autenticar um usuário em um servidor, de nome "Authorization", uma exemplificação do uso real seria: "Authorization:eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9"(NOTTINGHAM; MOGUL, 2005).

2.1.3 Códigos de Resposta HTTP

Os códigos de status de respostas das requisições HTTP servem para indicar erros, sucessos e outras informações das resposta HTTP. Estes códigos numéricos podem ser organizados em 5 grupos:

- Respostas de informação (100 até 199);
- Respostas de sucesso (200 até 299);
- Redirecionamentos (300 até 399);

- Erros do cliente (400 até 499);
- Erros do servidor (500 até 599).

Conforme apresentado em (FIELDING; RESCHKE, 2014), exemplos de códigos de status HTTP são:

- 200 OK: a requisição foi um sucesso;
- 400 Bad Request: A requisição não pode ser entendida pelo servidor devido a erros na sintaxe;
- 500 Internal Server Error: o servidor encontrou um erro e não pode completar a requisição.

2.2 Web Service

Durante o desenvolvimento das tecnologias, dos sistemas e da internet, houve a necessidade do desacoplamento entre aplicações, sendo elas implementadas na mesma linguagem ou não. Este desacoplamento foi necessário porque os sistemas se tornavam cada vez mais complexos e com mais comunicações entre seus componentes ou com outros sistemas, exigindo, assim, que as aplicações fossem mais independentes.

Devido a estes fatos surgiram os *web services*, que funcionam independentemente de sistema operacional, plataforma, servidor ou linguagem, transmitindo informações em formatos padronizados. Estes serviços podem ser facilmente modificados e aprimorados sem a necessidade de criação de um novo serviço do zero.

Com o desenvolvimento dos *web services*, modelos como o SOAP (Simple Object Access Protocol) e o REST (Representational State Transfer), que normalmente utilizam o protocolo de comunicação HTTP, começaram a aparecer como uma possível solução para comunicação entre componentes e integração entre componentes e integração de aplicações. Os formatos de troca de informação usados são normalmente o XML ou o JSON (CHRISTENSEN, 2001).

Entre os benefícios dos *web services* está a integração de informação e sistemas, já que é necessário apenas o uso, de uma forma simples, do XML/JSON e HTTP para funcionarem. Outro beneficio, é a reutilização, ja que um *web service* pode ser utilizado e reutilizado por varias aplicações ao mesmo tempo, sem nenhuma saber da existência das outras (OPENSOFT, 2016).

2.3 O Estilo Arquitetural REST

O REST foi introduzido em 2000 por Roy Fielding em sua tese de doutorado, sendo não um padrão ou uma tecnologia, mas sim um conjunto de restrições e regras, como: não possuir monitoração de estado, ter um relacionamento cliente/servidor e conter uma interface uniforme.

Apesar de estar relacionado com o HTTP, ele não é restrito só ao mesmo, e se tornou conhecida por ser uma opção mais leve e simples que o SOAP para implementação de serviços web. A cada dia, o REST têm ganhado mais força e espaço entre a comunidade como modelo para comunicação entre componentes de software por meio de tecnologias web. Um serviço que segue as regras estabelecidas pelo REST é denominado RESTful.

Ao contrario do SOAP, que é baseado em estado, o REST é baseado em três pilares: *Resources, URI, Representação* (MACORATTI,).

- **Recursos:** é o serviço em si, e pode interagir com qualquer coisa, desde sensores a bancos de dados;
- **Representação:** Ao receber uma requisição HTTP, o serviço não envia, como resposta, o recurso em si, mas sim uma representação textual do recurso. Normalmente, são a representação dos recursos é feita utilizando-se HTML, XML ou JSON;
- **Mensagens :** São as requisições e repostas HTTP, que utilizam além de outros recursos, os verbos HTTP, para identificar a operação desejada sobre o recurso.

2.3.1 REST e os verbos HTTP

Como foi dito anteriormente, o REST é bem relacionado com o protocolo HTTP. Por esse motivo, utiliza os verbos HTTP para especificar a operação sobre um recurso, ou seja, quando criar, atualizar, recuperar e excluir dados de um recurso no banco de dados. Em geral, os métodos utilizados normalmente são: POST, PUT, GET, e DELETE. Para ilustrar o uso desses verbos, pode-se imaginar uma API que administra os dados dos professores do Departamento de Ciência da Computação.

O verbo GET é utilizado para recuperar dados requeridos do banco de dados. Utilizando a API hipotética, caso uma solicitação fosse enviada para essa API com o verbo GET e a URI `/professores`, a lista contendo todos os professores seria retornada para o cliente. Caso outra solicitação fosse enviada com a URI `/professores/1`, o professor com o identificador "1" seria retornado para o cliente.

O verbo POST é utilizado para a criação de novas instâncias. Caso uma requisição HTTP com o verbo POST, URI `/professores/` e com as informações necessárias para a criação de um novo professor no corpo da mensagem, fosse enviada para a API, um novo professor seria criado. Após a criação desse professor, uma resposta HTTP poderia ser enviada para o cliente informando o sucesso da operação.

O verbo PUT, é utilizado para a atualização de uma instância. Funciona de forma parecida com o método POST. A principal diferença é a necessidade da requisição HTTP informar qual instância específica deve ser alterada. Isso pode ser feito utilizando-se a URI ou o corpo da requisição.

O verbo DELETE, é utilizado para remover uma instância. Caso uma requisição HTTP com o verbo DELETE e URI/professores/1 fosse enviada para o servidor, o professor com identificador 1 seria removido do banco.

2.4 .Net Core

De acordo com a Microsoft, o .NET Core é uma plataforma de desenvolvimento de software livre, de uso geral. Com essa plataforma, pode-se criar aplicativos .NET Core para Windows, macOS e Linux, para processadores x64, e para x86, ARM32 e ARM64, usando várias linguagens de programação (MICROSOFT, e).

As linguagens que podem ser utilizadas para escrever aplicativos e bibliotecas para o .NET Core são o C#, Visual Basic e F# , e as IDE normalmente utilizadas pelos desenvolvedores são o Visual Studio e o Visual Studio Code.

2.4.1 .Net Core vs .Net Framework

De acordo com a Microsoft, há duas implementações de .NET com suporte para a criação de aplicativos do lado do servidor: .NET Framework e .NET Core. Ambas compartilham muitos componentes e é possível, em alguns casos, compartilhar código entre as duas, pois além de utilizarem as mesmas linguagens, uma grande quantidade das bibliotecas externas, são construídas em cima do *.Net Standart*. O *.Net Standart* é uma especificação formal das APIs de todos os produtos .Net, por isso os código construídos em cima dessa especificação, são portáveis entre os produtos .Net (MICROSOFT, a).

No entanto, há diferenças fundamentais entre as duas. Uma escolha entre elas tem que considerar os resultados esperados e os requisitos específicos de cada projeto (MICROSOFT, c).

É indicado o uso do .NET Core para o aplicativo para servidores se:

- Houver necessidades de desenvolvimento ou uso em outro sistema operacional que não seja o Windows;
- For uma aplicação com a abordagem arquitetônica microsserviços;
- Utilização de contêineres do Docker;
- Necessitar de alto desempenho e sistemas escalonáveis.

Use o .NET Framework para o aplicativo para servidores se:

- O aplicativo já usa o .NET Framework atualmente (a recomendação é estender em vez de migrar);
- o aplicativo usa bibliotecas .NET de terceiros ou pacotes NuGet não disponíveis para o .NET Core;
- o aplicativo usa tecnologias .NET que não estão disponíveis para o .NET Core;
- Seu aplicativo usa uma plataforma que não oferece suporte ao .NET Core. Windows, macOS e Linux oferecem suporte ao .NET Core.

2.4.2 Entity framework Core

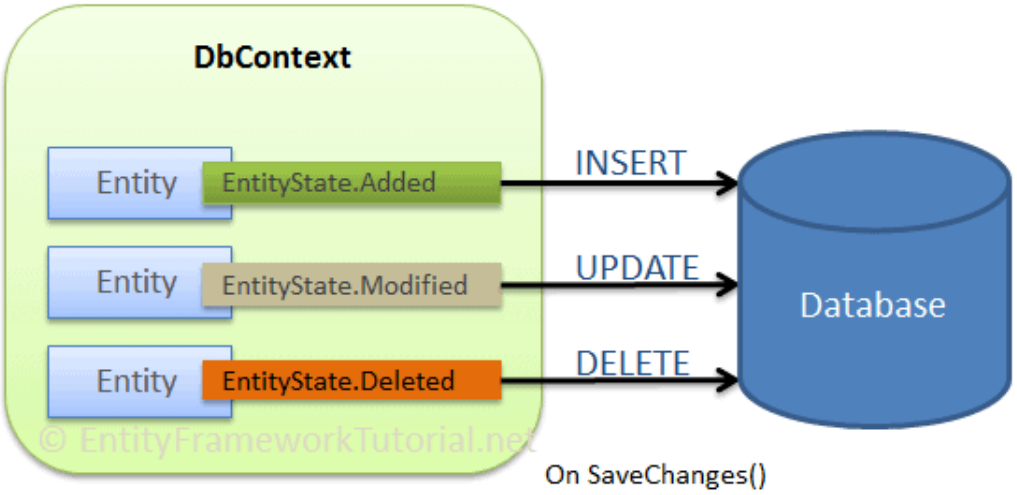
Segundo a própria Microsoft, Entity Framework Core é uma versão leve, extensível, de software livre e multiplataforma da popular tecnologia de acesso a dados do Entity Framework.

Esta tecnologia pode servir como um ORM (Mapeador de Objeto Relacional), permitindo que os desenvolvedores de .NET trabalhem com um banco de dados usando objetos do .NET e eliminando a necessidade de grande parte do código de acesso aos dados que eles geralmente precisam escrever, além de ser compatível com vários mecanismos de banco de dados, como o SQL Lite, o SQL Server e outros provedores. Portanto, utilizando o Entity, os desenvolvedores trabalham com os dados em forma de objetos e propriedades, aumentando o nível de abstração (MICROSOFT, b).

O acesso aos dados é feito utilizando um modelo, que é composto de entidades e um objeto para contexto que representa a sessão com o provedor de banco de dados escolhido. O *Entity Framework* não é apenas uma solução para mapeamento objeto relacional, ele foi criado para possibilitar a alteração e o acesso a dados representados por entidades. Isso é realizado por meio da utilização de informações de entidades e mapeamentos, com a conversão de uma consulta sobre objetos para uma consulta específica à fonte de dados (MICROSOFT, d).

Na Figura 2.2, é possível observar o mapeamento das entidades para o banco de dados e a relação dos estados nos quais a entidade se encontra em memória para a operação que será realizada no banco de dados.

Figura 2.2 – Mapeamento de entidades para o banco.



Fonte: https://www.entityframeworktutorial.net/efcore/saving-data-in-connected-scenario-in-ef-core.aspx

2.5 Testes de Unidade

Os sistemas, ao serem criados, naturalmente contêm entradas, processamento e saídas de dados. Para ganhar mais confiança sobre as funcionalidades implementadas, testes manuais ou automatizados podem ser implementados.

Dentre os vários tipos de testes de códigos e sistemas, o teste de unidade pode ser escolhido para garantir mais confiabilidade em funcionalidades específicas. O teste de unidade é chamado deste modo porque divide as funcionalidades do software em comportamentos discretos que podem ser testados como unidades individuais.

Várias ferramentas podem ser utilizadas para desenvolver testes de unidade em projetos .Net, dentre elas podem ser citadas o xUnit e o NUnit.

2.5.1 xUnit

De acordo com o site oficial[1] do xUnit.net (ou xUnit, como é mais conhecido), ele é uma ferramenta de teste open source, focada na comunidade de .Net Framework. Escrita pelo inventor original do NUnit v2, xUnit é a tecnologia mais recente para testes de unidade, com suporte ao C#, F#, VB.NET e outras linguagens .NET (XUNIT.NET, a).

Nas classes que devem ser testadas, é incluído o atributo `[Fact]` nos métodos de teste para indica-lo como um método a ser testado. Além dos métodos com o atributo `[Fact]`, podem ser criados outros métodos auxiliares, que não vão ser testados em si, mas serão usados por métodos de teste. Um exemplo do uso desse atributo pode ser visto na Figura 2.3.

Figura 2.3 – Exemplo do uso do atributo `[Fact]`.

```
1 //Atributo que indica que esse método é um teste de unidade.
2 [Fact]
3 public void TesteExemplo()
4 {
5     //código de teste
6 }
7
```

Além do atributo `[Fact]`, tambem pode ser utilizado o atributo `[Theory]`, sendo a diferença entre eles apenas que, com o atributo `[Fact]`, não é possível a passagem de parâmetro para o método, e, com o atributo `[Theory]`, é possível passagem de parâmetros para o método (XUNIT.NET, b).

Na Figura 2.4 é possível observar a criação de uma classe que irá testar o método `Add`, o método de nome `PassingTest` 'passará' no teste. Por outro lado, o teste `FailingTest` não irá passar, necessitando assim de uma correção no teste ou no método `Add`, sendo que neste caso o método de teste foi implementado de modo incorreto segundo as regras matemáticas de somatório.

[1] https://xunit.net/

Figura 2.4 – Exemplo de uso do atributo `[Fact]` numa método de adição.

```
using Xunit;

namespace MyFirstUnitTests
{
    public class Class1
    {
        [Fact]
        public void PassingTest()
        {
            Assert.Equal(4, Add(2, 2));
        }

        [Fact]
        public void FailingTest()
        {
            Assert.Equal(5, Add(2, 2));
        }

        int Add(int x, int y)
        {
            return x + y;
        }
    }
}
```

Fonte: https://xunit.net/docs/getting-started/netcore/cmdline

Na Figura 2.5, é possível visualizar um método de teste nomeado de `MyFirstTheory`, que está testando a função `IsOdd`. Neste trecho de código, é utilizado o atributo `[Theory]` e abaixo dele os atributos `[InlineData()]`, nos quais são informados os parâmetros que serão passados pro método de teste. Devido a existência destes três atributos, este método será chamado três vezes. Em cada chamada o valor da variável `value` será atualizado, sendo que nos dois primeiros, o teste passará, e no terceiro irá falhar.

Ao executar os códigos das figuras 2.4 e 2.5 com a ferramenta *Visual Studio* e utilizando a aba *Text Explorer*, o resultado seria o ilustrado na Figura 2.6. A aba de testes disponibiliza varias informações sobre os testes executados, e caso necessitar verificar o local da falha, o motivo, o tempo gasto no teste ou até mesmo pilha de execução, isso é facilmente acessado.

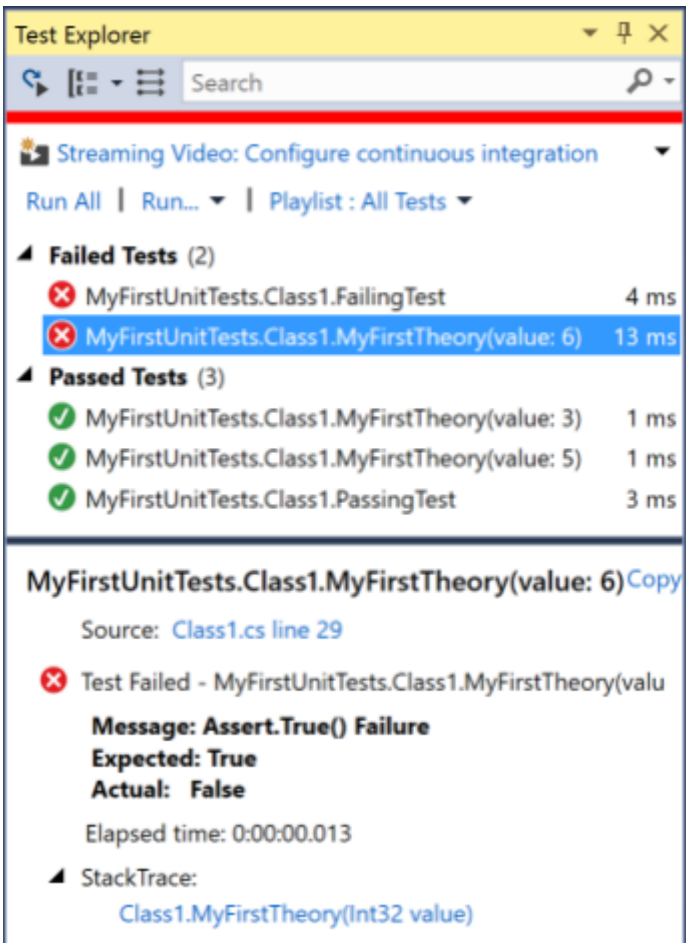
Figura 2.5 – Exemplo de uso do atributo `[Theory]`.

```
[Theory]
[InlineData(3)]
[InlineData(5)]
[InlineData(6)]
public void MyFirstTheory(int value)
{
    Assert.True(IsOdd(value));
}

bool IsOdd(int value)
{
    return value % 2 == 1;
}
```

Fonte: https://xunit.net/docs/getting-started/netcore/cmdline

Figura 2.6 – Resultado do teste de unidade.



Fonte: https://xunit.net/docs/getting-started/netcore/cmdline

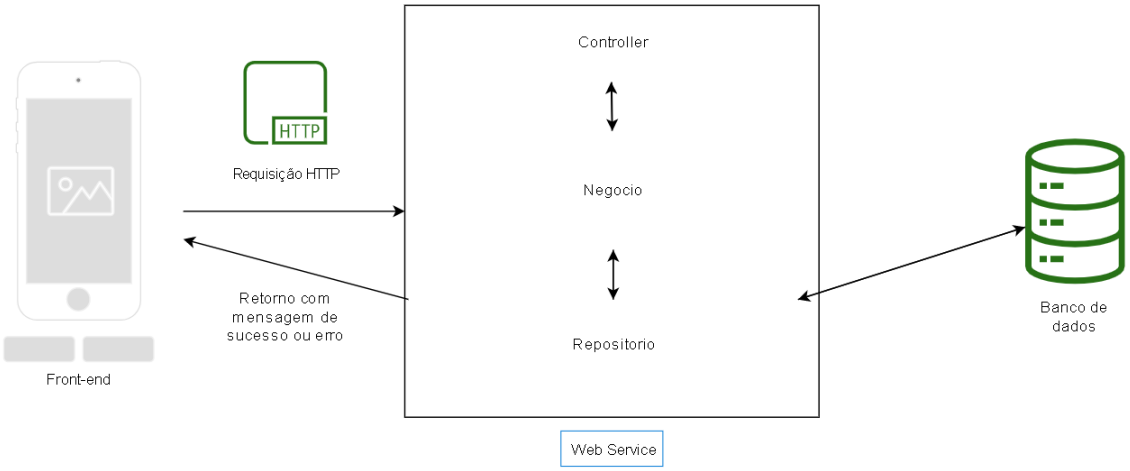
3 ATIVIDADES REALIZADAS

Para o desenvolvimento do projeto, usou-se a metodologia agil Scrum, contemplando todos os ritos requeridos.

A equipe do projeto foi constituída por 7 pessoas. Designou-se duas pessoas responsáveis para criação de interfaces de usuário, um Scrum Master, no qual teve a função de garantir que a equipe utilizasse de forma correta a metodologia Scrum. Ademais,um Product owner que representava o cliente e suas regras de negocio, e dois desenvolvedores .Net responsáveis pela criação do web service e da modelagem do banco de dados, sendo o autor um deles. Por fim, um design para a criação visual das telas.

A função do autor no projeto foi o desenvolvimento do web service e a modelagem do banco de dados. Na Figura 3.1, observa-se uma representação da estrutura básica desenvolvida.

Figura 3.1 – Representação do funcionamento do sistema implementado.



3.1 Conformidade com o Estilo Arquitetural REST

Neste trabalho, todas as tarefas de desenvolvimento ou manutenção de módulos foram feitas utilizando alguns dos padrões do estilo de arquitetura REST. Esta abordagem foi escolhida devido as integrações e facilidades ao usar esse estilo em conjunto com o .Net Core. Dentre essas facilidades, podem ser citadas: a parte de roteamento, retorno de representações em JSON com status HTTP, utilização de verbos HTTP e outras mais. Na Figura 3.2 é exemplificado uma classe `controller`, que implementa um *endpoint* para a listagem de idiomas.

Figura 3.2 – Exemplo de uma classe *controller* com um *endpoint* de listagem de idiomas.

```
[Route("api/[controller]")]
public class IdiomasController : ControllerBase
{
    private IdiomaNegocio _idiomaNegocio = new IdiomaNegocio();

    [HttpGet]
    public IActionResult RecuperarIdiomas()
    {
        try
        {
            var idiomas = _idiomaNegocio.RecuperarIdiomas();
            return Ok(idiomas);
        }
        catch (Exception exceçao)
        {
            return StatusCode(500, exceçao);
        }
    }
}

```

Para construir uma rota disponível para requisições, incluiu-se o no pipeline de requisições uma configuração nativa do AspNetCore de roteamento. Esta configuração pode ser inserido na classe de inicialização do projeto (por convenção nomeada de *Startup*), adicionando-se o serviço de MVC ao projeto e posteriormente incluindo-o no pipeline de requisições HTTP. Além disso, é necessário incluir um atributo *Route*, com a rota

desejada, na classe ou no método a ser roteado. Na linha 1 da Figura 3.2 é indicada a rota padrão dos *endpoints* dessa `Controller`, utilizando-se o atributo *Route* e a rota fornecida.

A definição do verbo HTTP a ser utilizado é feita incluindo-se atributos antes do *endpoint*, como, `[HttpGet]`, `[HttpPut]`, `[HttpPost]`, entre outros. Estes atributos, são classes que herdam da classe `Attribute` localizada no namespace `System`. Na Linha 6 da Figura 3.2 pode ser observada a utilização desses atributos, nesse caso o HttpGet, para indicar que esse *endpoint* apenas poderá ser acessado caso a requisição utilize o verbo GET em seu cabeçalho.

O envio de representações dos recursos é, frequentemente, feito retornando-se objetos que implementam a interface `IActionResult`. Os métodos para esse retorno estão contidos na classe `ControllerBase` que, de modo similar a interface `IActionResult`, faz parte do `namespace Microsoft.AspNetCore.Mvc`. Este retorno, depende do verbo HTTP utilizado, do procedimento que será feito, e da ocorrência ou não uma exceção. Este procedimento pode ser observado na Figura 3.2, na linha 12, no qual o método `Ok(idiomas)` é chamado, indicando que a resposta da requisição terá o status HTTP 200 e a lista de idiomas recuperada no seu corpo. Na linha 16 da Figura 3.2, pode-se observar o retorno, caso ocorra qualquer tipo de exceção no código, retornando um status HTTP 500 e a mensagem da exceção no corpo da mensagem.

A representação de dados é realizada, majoritariamente, utilizando-se o formato compacto de troca de dados JSON. A definição para se utilizar o JSON é feita de forma automática ao incluir o serviço MVC no projeto, como pode ser observado na Linha 5 da Figura 3.3, podendo ser modificada caso necessidade do projeto.

Figura 3.3 – Adicionando serviço MVC no projeto.

```
1    public class Startup
2    {
3        public void ConfigureServices(IServiceCollection
   services)
4        {
5            services.AddMvc();
6        }
7    }
8
```

3.2 Consulta e Persistência de Dados

Este projeto envolve diretamente a manipulação/gerenciamento de base de dados, já que depende de cadastros, atualizações, consultas, exclusão e outros procedimentos. Estes dados armazenados são todos gerados por usuários de gerência, candidatos ou o próprio serviço, no caso de datas, horas e regras de negócio. Para a consulta e persistência desses dados, foi utilizado o *Framework Entity Framework Core*, localizado no `namespace Microsoft.EntityFrameworkCore`, como ORM juntamente com o *SQL Server* como sistema gerenciador de banco de dados. Apesar da escolha pelo SQL Server, a troca para outro sistema gerenciador pode ser facilmente feita atualizando apenas algumas configurações.

Como um ORM, o *Entity Framework Core* realiza mapeamentos objeto relacional, utilizando entidades e objetos de contexto. Estas entidades ficam na camada *Repositorio*, juntamente com os contextos. Para que seja possível utilizar o *framework*, é necessário uma classe que herda da classe `DbContext`. Essa, tem diversos métodos de adição, consulta, deleção, alteração e entre outros métodos necessários para a utilização de um banco de dados.

Para a execução de qualquer requisição no banco de dados, é preciso uma instância de `DbContextOptions` que transporta informações de configuração, como o provedor de banco de dados a ser usado. Neste caso, o SQL Server, como pode ser visto na Linha 3 da Figura 3.4, assim como a *string* de conexão com o banco.

Figura 3.4 – Exemplo do método de configuração da classe que herda de *DbContext*.

```
1 protected override void OnConfiguring(DbContextOptionsBuilder
    optionsBuilder)
2 {
3     optionsBuilder.UseSqlServer("string de conexão"));
4 }
5
```

As classes de repositório foram implementadas de forma que, sempre que fosse necessário a inserção de novas classes, o padrão já estabelecido seria seguido, para maior controle dos métodos básicos de acesso ao banco. Esse controle foi feito utilizando a implementação de interfaces, generalização e herança. A classe `RepositorioBase` e a interface `IRepositorioBase` já estavam prontas antes do trabalho deste estágio. Portanto, ao longo do projeto, foi necessário, diversas vezes, criar novas classes e interfaces que utilizavam os códigos de repositório base.

Como exemplificação de implementação, a Figura 3.5 ilustra o códigos necessários para o acesso a uma tabela no banco de dados chamada Idioma. Na Linha 2, é informado o nome da tabela correspondente no banco de dados. Na Linha 3, uma classe é criada de nome Idioma, que contem uma propriedade do tipo *string* e nomenclatura "Nome". Esta classe será a classe mapeada para o banco de dados pelo *Entity Framework* e se tornará uma tabela. Na Linha 8, é criada uma interface para a classe "Idioma", que apenas implementa a interface base, que contém os métodos básicos de busca, inserção, atualização, etc. Na Linha 12, é criada a classe de acesso ao banco de dados da tabela "Idioma", na qual é implementada a interface criada acima (`IIdiomaRepositorio`), e é herdada a classe base de repositório, que contém as implementações dos métodos requeridos pela interface base dos repositórios (`IRepositorioBase`). Desse modo, tem-se uma classe utilizável, com todos os métodos básicos necessários, e caso seja requerido, pode-se criar outros métodos, apenas com adição do cabeçalho do método na interface de repositório de idioma, e implementação na classe repositório de idioma.

Figura 3.5 – Exemplo das estruturas necessárias para o mapeamentos objeto relacional.

```
1
2  [Table("Idioma")]
3  public class Idioma
4  {
5      public string Nome { get; set; }
6  }
7
8  public interface IIdiomaRepositorio : IRepositorioBase<Idioma>
9  {
10 }
11
12 public class IdiomaRepositorio : RepositorioBase<Idioma>,
       IIdiomaRepositorio
13 {
14 }
15
```

A utilização de um ORM foi de extrema importância para o desenvolvimento do projeto no tempo delimitado, pois com a utilização dessa técnica e das tecnologias envolvidas, a produtividade aumentou consideravelmente, uma vez que não era necessário se preocupar com as consultas SQL em si, mas apenas em utilizar o *EntityFramework*, o .Net Core e a linguagem de programação C#. Além da produtividade, a manutenibilidade dos códigos, foi incrementada, visto que as classes e interfaces estavam bem divididas e organizadas.

Um ponto negativo da utilização de um ORM é que, normalmente, a performance que o ORM terá para realizar as consultas ao banco é pior comparado com a utilização do SQL. Isso ocorre, principalmente, pela dificuldade de mapear de forma totalmente correta, e mesmo quando feito, em consultas mais complexas o ORM pode gerar operações desnecessárias ou ineficientes. Para o projeto em questão, esse *gap* de performance não foi um impeditivo, apesar da necessidade de performance, a perda por parte do ORM não foi perceptível ou impactante.

3.3 Testes unitários

Teste de unidade são de extrema importância para uma sistema, tanto para confiabilidade quanto para a manutenibilidade. Os testes de unidade utilizados neste projeto foram feitos utilizando a ferramenta de teste *xUnit*. Estes testes, em geral, permitiam a verificação de casos de sucesso e de falha, para que, a partir das correções, garantir que todas as funcionalidades estavam sendo feitas corretamente.

Toda os componentes da aplicação evoluíram juntamente com os testes, sendo que cada nova funcionalidade inserida, era acompanhada de seus testes unitários. Desse modo, caso houvesse alguma modificação ou incremento, os códigos de testes estavam verificando se essa alteração de alguma modo causou comportamento do código que não era previsto ou esperado. Este método de desenvolvimento trouxe grande confiança, pois garantia a funcionalidade dos procedimentos, fazendo com que a equipe não perdesse tempo testando o código e todas suas possibilidades exaustivamente após qualquer tipo de modificação ou incremento.

Entretanto, deve-se ressaltar que testes de unidade não possui apenas vantagens, pois é possível criar testes que não simulavam situações reais e que não iriam acusar falha ou até mesmo testes que apenas "caminham"pelo caminho de sucesso. Além disso, o tempo gasto no desenvolvimento dos teste não foi baixo, tendo uma quantidade significativa de tempo de desenvolvimento direcionado em testes e validações. Este tempo utilizado é convertido à longo prazo, em menos problemas, confiança, fácil identificação de bugs, e desnecessário testes manuais.

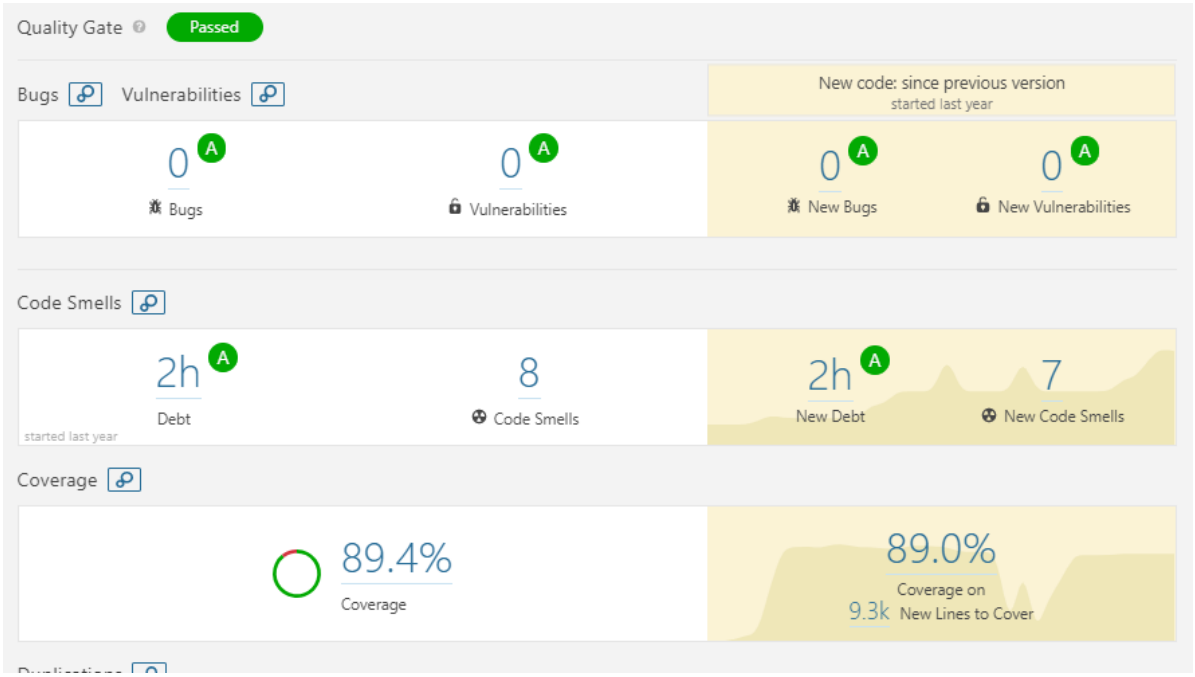
Os testes foram implementados depois das funcionalidades, utilizando-se a linguagem de programação C#. Portanto, desenvolveu-se os códigos para o sistema, e após finaliza-los, desenvolvia-se os testes para esses códigos. Toda a criação dos testes foi realizado pela mesma equipe que desenvolvia as funcionalidades do sistema, e a cada mudança era necessário a verificação, para averiguação da situação dos testes. Os testes eram executados utilizando a aba *Test Explorer* da IDE Visual Studio.

3.3.1 Sonarqube

Grande parte dos projetos realizado na empresa, são submetidos ao *Sonarqub*, uma plataforma com código aberto, que tem como objetivo realizar inspeções de qualidade de código, venerabilidades, bugs e verificação da porcentagem de código coberto por testes unitários.

Na Figura 3.6, é apresentado um exemplo de resultado de verificação do sistema implementado nesse trabalho. Nesse painel de resultados, pode-se observar: o número de bugs e venerabilidades identificados é 0, o numero de *code smells* (códigos possivelmente mal feitos ou com complexidade alta) é 8, o que é consideravelmente baixo. Sobre a cobertura de testes, o mínimo exigido pela empresa, é de 40%, porem conseguimos mais que dobrar esse número, conseguindo quase 90% de cobertura. Estes valores mostram de forma simplista, que o projeto teve planejamento, procura por qualidade e boas práticas. Esta alta porcentagem de cobertura de código trouxe várias vantagens ao ter mais segurança, e facilidade para a realização de incrementos ou manutenções nos códigos.

Figura 3.6 – Painel do projeto no Sonarqube.



Fonte: autor

3.3.2 Mock

Na programação orientada a objetos, Mocks são objetos que simulam e imitam o comportamento de objetos reais. Normalmente, são objetos programados para que dependendo do método e dos parâmetros que forem chamados, uma resposta estática é retornada.

Neste trabalho, foi utilizada a biblioteca Moq[1], que tem, como grande vantagem, a integração total com expressões lambda e o Linq. O Linq é uma linguagem de consulta integrada ao .NET, localizado no `namespace System.Linq`. Outro motivo da utilização do Moq é a sua popularidade entre os desenvolvedores .Net, tendo assim, uma documentação extensa, com muitos questionamentos e dúvidas comuns entre desenvolvedores.

Na Figura 3.7, pode-se observar uma função que retorna um Mock da interface `IIdomaRepositorio`. Este mock simula os métodos escritos nessa interface e seus retornos. Na Linha 3, é criado um objeto do tipo Mock, utilizando a interface `IIdiomaRepositorio`. Após a criação do objeto, nas linhas 4 e 5, são definidos os dados que esse objeto irá retornar, caso os métodos `RecuperarEntidade()` e `RecuperarEntiadeId(1)` sejam chamados.

Com este método criado, sempre que algum código que deve ser testado utilizasse o repositório de idiomas, ao chamar a função `Repositorio()`, uma mock da interface `IIdiomasRepositorio` estaria pronto, com todos os métodos necessários para o teste. Na Figura 3.8 é ilustrado o uso do mock criado.

Ao longo do trabalho foram criados vários mocks para a sua utilização nos testes de unidade. Em geral, criou-se uma classe e dentro dela um método para retornar o mock de cada classe de repositório. Sendo assim, cada classe de repositório tinha sua respectiva classe para criação do mock. Dessa forma, ao incrementar o sistema com novas classes e funcionamentos, além do método de teste do código a ser testado, era necessário a criação das classes de mock de cada novo repositório. Como dito anteriormente, a criação de toda essa estrutura e novas classes demanda um tempo considerável de desenvolvimento.

[1] <https://github.com/moq/moq4>

Figura 3.7 – Exemplo de método para retorno do Mock de uma interface.

```
1 public IIdiomaRepositorio Repositorio()
2 {
3     var repositorioMock = new Mock<IIdiomaRepositorio>();
4     repositorioMock.Setup(c => c.RecuperarEntidades()).Returns(
    EntidadesFake());
5     repositorioMock.Setup(c => c.RecuperarEntidadeId(1)).Returns
    (EntidadeFake());
6
7     return repositorioMock.Object;
8 }
9
```

3.3.3 Assert

A classe *Assert*, localizada na ferramenta de testes *xUnit*, disponibiliza vários métodos de teste, como por exemplo:

- `IsType`, que verifica se um retorno é do mesmo tipo indicado;
- `True`, que verifica se a expressão passada é verdadeira.

Na Figura 3.8, é possível observar um exemplo de teste de unidade utilizando a ferramenta de teste *xUnit*. Na linha 1, é adicionado o atributo `[Fact]`, que indica que este método é um teste de unidade. Após indicar que o método é um teste de unidade, na linha 4, é criado o mock de uma classe controladora, neste caso, a classe controladora de Idiomas. Na linha 5, é definido o retorno quando o método RecuperarIdiomas() é chamado pela classe "mocada". Na linha 9, é utilizado o método de teste `IsType` para verificar se o tipo de retorno é do tipo esperado (`OkObjectResult`). Caso não seja, o teste irá retornar erro e uma refatoração será necessária.

Cada teste de unidade deste projeto testa apenas uma parcela de código, porém nada impede do teste verificar mais de uma condição, ou melhor dizendo, nada impede o teste de ter mais de um *assert*.

Figura 3.8 – Exemplo de teste para validação de retorno em uma controller.

```
[Fact]
public void RecuperarIdiomas()
{
    var controller = new Mock<IIdiomaController>;
    controller = controller.Setup(c => c.RecuperarIdiomas()).
   Returns(Idiomas());

    var resultado = controller.RecuperarIdiomas();

    Assert.IsType<OkObjectResult>(resultado);
}

```

3.4 Fluent Validation

Uma parte considerável dos endpoints criados nesse projeto, recebiam dados no corpo das requisições HTTP, por exemplo, nomes, identificadores, dados pessoais e etc. Algumas dessas informações podem não ser enviadas, ser enviadas de forma errônea, ou formatadas de um modo não esperado. Para evitar que estas situações ocorram e os dados sejam armazenados de forma incorreta, antes de realizar a persistência dos dados, é utilizado a biblioteca de validação *Fluent Validation*. Esta biblioteca, de forma simples, testa os dados da entidade em questão e caso todos os testes passem, o código continuará normalmente. Caso um ou mais erros de validação aconteçam, retorna-se uma exceção para o requerente, com os erros encontrados.

A utilização dessa biblioteca aumentou consideravelmente a produtividade da equipe, devido a dois motivos principais. O primeiro, é a não necessidade da implementação de métodos de validação simples como, por exemplo, a verificação de campos vazios e nulos, campos maiores ou menores do que desejado, e-mails inválidos ou outras validações básicas presentes na documentação. O segundo motivo é a facilidade do retorno dos erros encontrados para o requerente.

Durante este projeto, foram implementados ou modificados vários processos de validação. Na Figura 3.9, pode-se observar uma classe que valida a entidade Idioma. Na

linha 1, é criada essa classe, que herda da classe `AbstractValidator`, com a entidade Idioma sendo passada como o objeto a ser validado. As regras de validação são definidas dentro do construtor da classe, como pode ser visto nas linha 4 e 5. Na linha 4, é estabelecido que a propriedade Nome não deve ser vazia ou nula, e, caso ela seja, a validação retorna a mensagem "Nome vazio ou nulo". Na linha 5, é criada outra regra, para a propriedade Nivel, que é do tipo do enum e tem um conjunto de constantes nomeadas do tipo numérico integral subjacente. Caso por algum motivo seja enviado um nível que não pertence ao enumerador, é retornado o erro "Nível informado inválido".

Figura 3.9 – Classe de validação da entidade Idioma.

```
1 public class IdiomaValidacao : AbstractValidator<Idioma>{
2     public IdiomaValidacao()
3     {
4         RuleFor(x => x.Nome).NotEmpty().WithMessage("Nome vazio
    ou  nulo");
5         RuleFor(x => x.Nivel).IsInEnum().WithMessage("Nível
    informado inválido");
6     }
7 }
8
```

Na Figura 3.10, o método `CriarIdioma` tem como objetivo testar os dados recebidos e, caso sejam válidos, persisti-los. Na linha 2, o objeto JSON, recebido no corpo da requisição HTTP, é convertido em uma entidade Idioma. Na linha 3, é criada uma instância da classe de validação e, logo após, é chamado o método `Validate`, que é implementado na classe `AbstractValidator`. Esse método `Validate` basicamente aplica as regras às propriedades da entidade passada como parâmetro. O resultado dessa chamada é um objeto do tipo `ValidationResult`, que contem duas propriedades: `IsValid` e `Erros`. Na linha 5, pode ser visto a utilização a propriedade `IsValid`, que é do tipo *bool*, e tem o valor de verdadeiro, caso não tenha ocorrido nenhum erro de validação, ou false, caso tenha. Se a entidade for válida, ela é persistida, como é visto na linha 6. Caso a entidade não esteja correta, é chamada, na linha 8, uma exceção com os erros encontrados, que são retornados para o requerente.

Figura 3.10 – Uso da classe de validação.

```
1 public void CriarIdioma(IdiomaJson idiomaJson)
2     var entidade = ConverterJsonEmEntidade(idiomaJson);
3     var validacao = new IdiomaValidacao().Validate(entidade);
4
5     if (validacao.IsValid)
6         _repositorio.SalvarEntidade(entidade);
7     else
8         throw new Exception(validacao.Errors);
9
```

3.5 Módulo do Sistema de Recrutamento

Foi implementado um módulo do sistema gestão de recrutamento, que atualiza as redes sociais do usuário. O perfil de usuário necessitava de vários campos, como, nome, CPF, redes sociais, idiomas falados e etc. Essas informações eram enviadas e armazenadas pela API de diversas formas e utilizando diversos endpoints.

Para demostrar uma tarefa de implementação de um endpoint completo para atualizar as redes sociais de um usuário, é necessário saber que é utilizado o padrão de projeto de injeção de dependência com interfaces. Esse padrão visa diminuir o nível de acoplamento de código da aplicação e deixar a responsabilidade da criação das classes para o sistema, definindo os tempos de vida de cada classe e interface.

Primeiramente, é necessário a criação de um método na controller, definindo o verbo HTTP e a rota requerida. Pode ser visto na Figura 3.11, que é utilizado o verbo HTTP PUT, ja que a ação executada é a de atualização. Alem disso, é chamado o método EditarRedesSociais, que está localizado na classe de serviço, sendo essa classe instanciada via injeção de dependência. Caso ocorra um erro durante o processo de atualização, uma exceção é disparada para o tratamento do erro na linha 7.

Após a criação do método na classe controladora, é implementado o método na classe de serviço que contem as regras de negócio, a chamada para o repositório e o retorno de sucesso ou erro. Na Figura 3.12, é possível observar a atualização dos dados de redes sociais do usuário. Na linha 2, é buscado no banco de dados a entidade do usuário

Figura 3.11 – Método da classe controladora UsuarioController.

```
[HttpPut]
public IActionResult EditarRedesSociais(RedesSociais
   redesSociais){
    try{
        _service.EditarRedesSociais(redesSociais);
    }
    catch (Exception e){
        throw new TratarException(e);
    }

```

e armazenada na variável `usuario`. Após isso, são atualizados os conteúdos da entidade, sobrescrevendo os valores antigos com os novos recebidos como argumentos na chamada do método. Depois de atualizar os dados, é persistida no banco a entidade modificada. Os métodos utilizados para a recuperação e atualização dos da usuário são métodos já implementados na classe `RepositorioBase`.

Figura 3.12 – Método da classe de negocio UsuarioNegocio.

```
public void EditarRedesSociais(RedesSociais redesSociais){
   var usuario = _repositorio.RecuperarEntidadeAtivaPorId(
   IdUsuario);
   usuario.RedesSociais.Facebook = redesSociais.Facebook
   usuario.RedesSociais.linkedin = redesSociais.linkedin
   usuario.RedesSociais.Instagram = redesSociais.Instagram

    _repositorio.AtualizarEntidade(usuario);

```

4 CONCLUSÃO

O objetivo deste trabalho de estágio foi o desenvolvimento de um *web service* para criação e administração de vagas de emprego, utilizando o *framework* .Net Core. Este trabalho mostrou várias das principais bibliotecas e *frameworks* oferecidos ao desenvolvedor pela plataforma .NET, grande parte deles criados exclusivamente pela comunidade. Além disso, foram discutidas as atividades gerais realizadas durante o estágio.

O estágio proporcionou varias experiencias e conhecimentos interessantes como por exemplo:

- Quando utilizar .Net Core e quando não utiliza-la;
- Trabalho em equipe entre desenvolvedores *back-end* e *front-end*;
- Utilização de banco de dados relacionais, que relembraram e melhoraram as habilidades utilizando a linguagem SQL e seus sistemas gerenciadores;
- Utilizações de *design patterns* e melhores práticas de programação utilizando C#.
- Noções de trabalho em equipe utilizando o método ágil *Scrum*.

Sobre os conhecimentos adquiridos durante a graduação, e utilizados durante o estágio, pode-se citar as noções básicas no desenvolvimento de algoritmos e estruturas de dados, aprendidas nas matérias de Introdução aos Algoritmos, Estrutura de Dados e Paradigmas de Linguagens de Programação. Outro conhecimento importante utilizado foi o adquirido durante as aulas de Introdução a Sistemas de Banco de Dados e Sistemas Gerenciadores de Banco de Dados que foram de extrema importância para a criação e administração do banco de dados relacional respectivo ao projeto. É importante ressaltar as aulas de Práticas de Programação Orientada a Objetos que apresentaram várias noções de programação utilizando a orientação a objetos, assim como boas práticas e *design patterns*. Os conhecimentos adquiridos na disciplina de Sistemas Distribuídos foram veementemente utilizados para a implementação do trabalho descrito neste documento, principalmente nos conceitos sobre comunicação entre cliente e servidor e *WebSockets*.

A estrutura disponibilizada pela empresa onde foi realizado o estágio ultrapassou expectativas e auxiliou a efetividade e eficiência no desenvolvimento das atividades realizadas durante o estágio, pois tem um modelo focado tanto na produtividade quanto na satisfação dos colaboradores. Além da estrutura física, a empresa disponibilizou cursos nas principais plataformas, para o aprendizado e melhora na qualidade dos profissionais envolvidos.

Os profissionais que integram a empresa são capacitados e sempre presentes quando dúvidas ou questionamentos apareciam, criando-se sempre discussões saudáveis sobre os assuntos tratados. Este ambiente é resultado tanto dos colaboradores quanto da cultura que a empresa estabelece, sempre visando a troca de conhecimentos e o trabalho em equipe.

REFERÊNCIAS

CHRISTENSEN, F. C. G. M. S. W. E. **Web Services Description Language (WSDL) 1.1**. W3, 2001. Disponível em: <https://www.w3.org/TR/2001/NOTE-wsdl-20010315>.

FIELDING, R. T.; RESCHKE, J. **Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content**. RFC Editor, 2014. RFC 7231. (Request for Comments, 7231). Disponível em: <https://rfc-editor.org/rfc/rfc7231.txt>.

MACORATTI, J. C. **Compreendendo o estilo REST**. [S.l.]. Disponível em: <http://www.macoratti.net/16/05/net_rest1.htm>. Acesso em: 23 julho 2020.

MICROSOFT. **Entity Framework Core**. [S.l.]. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/standard/net-standard>. Acesso em: 23 agosto 2020.

MICROSOFT. **Entity Framework Core**. [S.l.]. Disponível em: <https://docs.microsoft.com/pt-br/ef/core>. Acesso em: 23 julho 2020.

MICROSOFT. **.NET Core versus .NET Framework para aplicativos de servidor**. [S.l.]. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/standard/choosing-core-framework-server>. Acesso em: 21 julho 2020.

MICROSOFT. **Visão geral de Entity Framework**. [S.l.]. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/framework/data/adonet/ef/overview>. Acesso em: 23 julho 2020.

MICROSOFT. **Visão geral do .NET Core**. [S.l.]. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/core/introduction>. Acesso em: 21 julho 2020.

MOZILLA. **HTTP**. [S.l.]. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP>. Acesso em: 22 julho 2020.

NOTTINGHAM, M.; MOGUL, J. **HTTP Header Field Registrations**. RFC Editor, 2005. RFC 4229. (Request for Comments, 4229). Disponível em: <https://rfc-editor.org/rfc/rfc4229.txt>.

OPENSOFT. **Web service: o que é, como funciona, para que serve?** [S.l.], 2016. Disponível em: <https://www.opensoft.pt/web-service>.

XUNIT.NET. **About xUnit.net**. [S.l.]. Disponível em: <https://xunit.net/>. Acesso em: 23 julho 2020.

XUNIT.NET. **Comparing xUnit.net to other frameworks**. [S.l.]. Disponível em: <https://xunit.net/docs/comparisons>. Acesso em: 12 setembro 2020.