UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

EDUARDO OSIELSKI SCHONHOFEN

Desenvolvimento de um Sistema de Catalogação de Recursos para o Museu do Mar

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Leandro Krug Wives
Coorientador: Ms. Francisco Dutra dos Santos Jr

Porto Alegre
2024

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patricia Pranke
Pró-Reitora de Graduação: Prof. Cíntia Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Marcelo Walter
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço à todos os familiares, amigos, professores, colegas de trabalho e demais conhecidos que fizeram parte da minha jornada profissional e acadêmica durante os últimos 8 anos.

RESUMO

O Museu do Mar, um museu marítimo com diversos artigos coletados por pescadores, está sendo edificado no balneário de Quintão no município de Palmares do Sul. Entretanto, é necessário um software para permitir catalogar e exibir esses artigos virtualmente, através de imagens e diversas descrições. Este TCC tem como objetivo desenvolver uma aplicação web, permitindo o registro desses itens, sua exibição virtual para o público, e também disponibilizar uma API para que futuros projetos possam consumir os dados desses itens, como, por exemplo, um Museu Virtual em 3D. Neste projeto, o front-end da aplicação foi desenvolvido utilizando React, e o backend com .NET 8 e SQL Server. Além disso, cada framework utilizou um diverso conjunto de bibliotecas para permitir uma melhor qualidade de código e aplicação de diversos padrões de projeto. Este documento descreve as tecnologias utilizadas no desenvolvimento da aplicação, seus componentes, arquitetura e elementos. Também apresenta um guia de uso e uma análise com usuários.

Palavras-chave: Ciência da Computação. UFRGS. Aplicação Web. Backend. Frontend. .NET. React.

Development of a Resource Cataloging System for the Sea Museum

ABSTRACT

The "Museu do Mar", a maritime museum with various fishing articles from fisherman, is being built on the coast of Quintão in the municipality of Palmares do Sul. However, software is needed to allow cataloging and displaying these articles virtually, using images and several descriptions; this monography aims to develop a web application enabling the recording of these items' virtual display to the public and also providing an API so that future projects can consume the data from these items, such as, for example, a Virtual 3D Museum. In this project, the application's frontend was developed using React, and the backend was developed with Dot NET 8 and SQL Server. Additionally, each framework utilizes a diverse set of libraries to enable better code quality and application of various design patterns. This document describes the technologies used to develop the application, its components, architecture, and elements. It also presents a user guide and an analysis of users.

Keywords: Computer Science, UFRGS, Web Application, Backend, Frontend, .NET, React.

LISTA DE FIGURAS

Figura 3.1 Screenshot da visualização do tipo de retorno de DataGalleryState.getTree() 17
Figura 3.2 Exemplo de formulário que segue o padrão Antd 17

Figura 4.1 Modelagem do Banco de Dados 23
Figura 4.2 Diagrama de Pacotes da Aplicação.. 24

Figura 5.1 Diagrama Simplificado do Sistema 42

Figura 6.1 Tela de Login do sistema 43
Figura 6.2 Galeria 44
Figura 6.3 Novo Registro 44
Figura 6.4 Editar Registro 44
Figura 6.5 Deletar Registro 45
Figura 6.6 Página Inicial 46

Figura 7.1 Eu acho que gostaria de usar esse sistema com frequência. 48
Figura 7.2 Eu acho o sistema desnecessariamente complexo. 48
Figura 7.3 Eu achei o sistema fácil de usar. 49
Figura 7.4 Eu acho que precisaria de ajuda de uma pessoa com conhecimentos técnicos para usar o sistema. 49
Figura 7.5 Eu acho que as várias funções do sistema estão muito bem integradas. 50
Figura 7.6 Eu acho que o sistema apresenta muita inconsistência. 50
Figura 7.7 Eu imagino que as pessoas aprenderão como usar esse sistema rapidamente. 51
Figura 7.8 Eu achei o sistema atrapalhado de usar. 51
Figura 7.9 Eu me senti confiante ao usar o sistema. 52
Figura 7.10 Eu precisei aprender várias coisas novas antes de conseguir usar o sistema. 52

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
JSON	JavaScript Object Notation
SQL	Structured Query Language
IoT	Internet of Things
SUS	System Usability Scale
OData	Open Data Protocol
REST	Representational State Transfer
EF	Entity Framework
LINQ	Language Integrated Query
CLI	Command Line Interface
JWT	JSON Web Token
CSS	Cascading Style Sheets
IDE	Integrated Development Environment
Antd	Ant Design
CRUD	Create,Read,Update,Delete
HTTP	Hypertext Transfer Protocol
IIS	Internet Information Services

SUMÁRIO

1 INTRODUÇÃO **10**
2 TRABALHOS RELACIONADOS **11**
2.1 Museu do Mar Virtual **11**
2.2 BORBRS®2 .... **11**
2.3 Tu em Torres **11**
2.4 CatalogIt **11**
3 CONCEITOS E TECNOLOGIAS **13**
3.1 .NET **13**
3.1.1 OData 13
3.1.2 Entity Framework Core 8 14
3.1.2.1 Entity 14
3.1.2.2 Configuration 14
3.1.2.3 Migrations 14
3.1.3 LINQ 15
3.1.4 JWT Token 15
3.1.5 AutoMapper 15
3.1.6 IIS 15
3.1.7 Bcrypt 16
3.2 React **16**
3.2.1 Typescript 16
3.2.2 Ant Design 17
3.2.3 Axios 18
3.2.4 React Tree Store 18
3.2.5 Blob 18
3.2.6 Local Storage 18
3.3 Microsoft SQL Server **18**
3.4 Ngrok **19**
4 MODELAGEM E PROJETO **20**
4.1 Requisitos **20**
4.1.1 Levantamento de Requisitos 20
4.1.2 Organização das Entregas 21
4.1.3 Modelagem de Dados 21
4.2 Arquitetura do Sistema **23**
4.2.1 Frontend 23
4.2.2 Backend 25
5 IMPLEMENTAÇÃO **26**
5.1 Frontend **26**
5.1.1 Service 26
5.1.2 Entities 29
5.1.3 Stores 30
5.1.4 Pages 31
5.1.5 Router 33
5.1.6 Helpers 33
5.2 Backend **34**
5.2.1 Domain 34
5.2.2 Repository 35
5.2.3 Services 35
5.2.4 Controllers 37

5.2.5 Settings...37
5.2.6 OData...37
5.2.7 Migrations...38
5.2.8 Mappings...38
5.2.9 Helpers...39
5.3 Patterns...39
5.3.1 Repository Pattern...40
5.3.2 Unit Of Work...40
5.3.3 Dependency Injection...41
5.3.4 Deploy...42
6 MANUAL DO USUÁRIO...43
6.1 Login...43
6.2 Galeria...43
6.3 Cadastro e Atualização...43
6.4 Página Inicial...45
7 AVALIAÇÃO E VALIDAÇÃO...47
8 CONCLUSÕES...53
REFERÊNCIAS...55
9 APÊNDICES...58
9.1 Apêndice A...58
9.2 Apêndice B...59

1 INTRODUÇÃO

O Museu do Mar, sendo construído fisicamente em Balneário Quintão, no município de Palmares do Sul, abrigará diversos itens coletados por pescadores locais. Esse museu possuirá uma versão virtual, com um ambiente 3D, mas primeiramente será necessário catalogar os dados desses itens coletados em um sistema virtual. Existem diversos sistemas de catálogo de museu já desenvolvidos, cada um com sua especificidade, alguns para obras de arte, de outros mais genéricos que permitem o cadastro de qualquer item, como o CatalogIt, que possui uma interface web e mobile, e permite a colaboração de diversos usuários, entretanto, possui um custo elevado entre $12.50 a $37.50 dólares, o que se torna inviável para ser utilizado por um órgão público.

A proposta deste trabalho é de desenvolver um sistema web que permita catalogar e visualizar esses itens coletados. Para isso, o usuário irá utilizar a aplicação para registrar esses itens individualmente, com as imagens e dados necessários.

Além disso, a aplicação disponibilizará uma API para futuramente ser possível o desenvolvimento de uma aplicação 3D para consumir os dados dessa API. O sistema web consiste de um frontend desenvolvido em React, utilizando a linguagem Typescript, de um backend desenvolvido em .NET 8 e C# 12, e de um banco de dados desenvolvido em Microsoft SQL Server. Além disso, foi aplicada uma pesquisa de System Usability Scale para alguns usuários avaliarem a usabilidade do sistema desenvolvido.

O restante do texto está estruturado da seguinte forma: No próximo capítulo são descritos os conceitos relacionados e as tecnologias utilizadas no desenvolvimento da aplicação, dividida em Backend, com suas bibliotecas específicas .NET e em Frontend React, também com suas tecnologias específicas. Depois temos um capítulo focado em descrever como surgiu o conceito inicial do projeto, e como foram definidas as telas que o usuário irá utilizar e a modelagem dos dados, junto com a arquitetura do sistema. No capítulo 4, temos detalhes específicos da implementação das principais partes do projeto, junto com os padrões adotados. No capítulo 5 é mostrado o resultado da avaliação e Validação da pesquisa SUS com os usuários participantes. E por último temos um Manual do Usuário com todas as telas do sistema.

2 TRABALHOS RELACIONADOS

Existem alguns trabalhos com temática similar ao apresentado, sendo alguns desenvolvidos em monografias e outros são softwares comerciais, os principais são:

2.1 Museu do Mar Virtual

O trabalho de graduação "Proposta de um sistema para o registro de informações de um Pescador para apoio ao Museu do Mar Virtual"também implementa para o Museu do Mar um software para solucionar as mesmas necessidades, entretanto, esta implementação é feita em Java, Spring *Framework*,PostgreSQL,React e Docker. Algumas diferenças são os dados de entrada e o boletim de condições do mar. (SILVA, 2021)

2.2 BORBRS®2

O BorbRS®2 é um sistema que disponibilizada dados e realiza análises sobre padrões de borboletas no estado do Rio Grande do Sul. Entretanto, possui uma interface complexa, onde diversos outros trabalhos tentam melhorar a sua eficiência e usabilidade, como por exemplo o trabalho de graduação "Implementação de um módulo de consultas geográficas para o BorbRS®2". (SANTOS, 2011)

2.3 Tu em Torres

O trabalho de graduação "“Tu em Torres” : auxiliando a aprendizagem sobre a fauna e flora de Torres para alunos do ensino fundamental", onde é desenvolvido um aplicativo educacional com conceitos de gamificação par alunos do ensino fundamental, mas que também possui interfaces para registros taxonômicos. (CECHIN, 2019)

2.4 CatalogIt

O sistema CatalogIt é um software comercial desenvolvido nos Estados Unidos, Califórnia, especificamente para catalogar itens de museu, possui uma interface mobile

amigável, permitindo customizar os tipos e propriedades dos itens do acervo. Uma desvantagem desse sistema é o elevado custo da assinatura mensal, podendo ultrapassar 30 dólares mensais. (CATALOGIT, 2024)

3 CONCEITOS E TECNOLOGIAS

Para um melhor entendimento sobre este trabalho, alguns conceitos e tecnologias precisam ser conhecidos. Este capítulo traz uma explicação sobre estes assuntos.

A aplicação foi desenvolvida com um *frontend* em React, *backend* em .NET e banco de dados Microsoft SQL Server, cada um com sua própria linguagem e bibliotecas. Todas essas tecnologias foram escolhidas de maneira a facilitar a curva de aprendizagem e exercitar e consolidar o conhecimento já adquirido.

3.1 .NET

O *backend* foi desenvolvido utilizando .NET Core 8.0, sendo esta a nova versão do .NET, um *framework* popular desenvolvido pela Microsoft, sendo possível utilizá-lo em diversas aplicações como: Desenvolvimento Web, *Mobile*, *Cloud*, Microsserviços, *Machine Learning*, IoT e Jogos. A versão mais recente do .NET 8 utiliza a nova versão da linguagem de programação C# 12. (MICROSOFT, 2024g; MICROSOFT, 2024h)

A seguir serão descritas as principais bibliotecas e extensões utilizadas junto ao *framework*.

3.1.1 OData

OData é um protocolo REST que define um conjunto de boas práticas para construir e consumir dados de uma API RESTful. Para .NET, foi utilizado o pacote Microsoft.AspNetCore.OData 8.24.

A principal vantagem de utilizar OData em uma API é permitir que o consumidor especifique o que deseja obter em um *endpoint*, por exemplo, é possível escolher as propriedades de uma classe, quais propriedades externas adicionar, filtrar por propriedades, adicionar paginação, contar o número de elementos, tudo em uma única requisição feita para a API. (ODATA, 2024; MICROSOFT ODATA, 2024)

Segue um exemplo de requisição OData feita pelo *frontend* para a API:

```
1 https://localhost:44375/publicMuseumRecords\$top=10\&\$skip=0\&\$count=
    true\&\$expand=Images
```

No exemplo, $publicMuseumRecords$ é uma das entidades, e com a sintaxe OData

podemos paginar o resultado a cada 10 elementos, requisitar o número total de dados e adicionar a entidade externa de $Images$.

3.1.2 Entity Framework Core 8

O Entity Framework é um mapeador de relação de objetos que permite criar uma camada de acesso a dados limpa, portátil e de alto nível com o .NET (C#) em diferentes bancos de dados, incluindo o Banco de Dados SQL Server, SQLite, MySQL, PostgreSQL e Azure Cosmos DB. Ele dá suporte a consultas LINQ, controle de alterações, atualizações e migrações de esquema. (MICROSOFT, 2024b; MICROSOFT ASPNET, 2024b)

Os principais conceitos de Entity Framework são:

3.1.2.1 Entity

É uma classe definida em C#, que será mapeada para uma tabela no banco de dados, cada propriedade da classe será uma coluna. Além disso, é possível adicionar outras classes como propriedade, que serão mapeadas para uma *foreign key* ou para uma tabela *many-to-many*, dependendo da relação.

3.1.2.2 Configuration

É uma classe definida em C#, que será utilizada para configurar uma Entity previamente definida. Nessa classe é possível configurar o Schema do banco de dados, o nome da tabela, e propriedades específicas de cada propriedade da classe, como o número de caracteres de uma string, valor default, obrigatoriedade, e o tipo de relação entre uma tabela e outra.

3.1.2.3 Migrations

Após definir as Entities e as Configurations, é possível rodar um comando no .NET CLI que gera uma Migration, que é um conjunto de comandos que atualiza o banco de dados incrementalmente, ou seja, é recomendado rodar uma migration a cada entity e configuration criada. Após gerar a migration, é possível aplicar ela ao banco de dados.

3.1.3 LINQ

LINQ (*Language Integrated-Query*, ou consulta integrada à linguagem) é um conjunto de tecnologias que permitem abstrair a camada de gestão de dados, realizando consultas ao banco de dados utilizando a própria sintaxe do C#. (MICROSOFT, 2024d; MICROSOFT, 2026)

Um exemplo de consulta LINQ que busca no banco de dados, filtra por Id e retorna o primeiro elemento seria o seguinte:

```
1 var record = await repository.GetQueryable().Where(record => record.Id
     == updatedRecord.Id).FirstOrDefaultAsync();
```

3.1.4 JWT Token

JWT Web Tokens representa o padrão RFC 7519, que permite uma comunicação segura entre o backend e o frontend. Na nossa aplicação, foi utilizado para Autenticar o acesso ao sistema para usuários logados. (Internet Engineering Task Force (IETF), 2015)

Para .NET existem o pacotes $System.IdentityModel.JWT7.2.0$ e $Microsoft.AspNetCore.Authentication.JwtBearer$ que são utilizados para implementar essa funcionalidade. (MICROSOFT ASPNET, 2024a; AZUREAD, 2024)

3.1.5 AutoMapper

É uma biblioteca que permite facilmente mapear um objeto para outro objeto, podendo mapear de maneira explícita, ou implícita, bastando os nomes e tipos das propriedades serem iguais, compatível com EF e OData. (AUTOMAPPER, 2024a; AUTOMAPPER, 2024b)

3.1.6 IIS

É servidor *web* padrão da Microsoft para executar aplicações .NET, gerenciando requisições HTML e de arquivos. (MICROSOFT, 2024c)

3.1.7 Bcrypt

É o método de criptografia utilizado para gerar a *hash* das senhas, gerando um *hash* para a senha e uma *hash* denominada *salt*, aumentando a complexidade. (PROVOS; MAZIèRES, 1999; CHRISMCKEE, 2024)

3.2 React

React é uma biblioteca Javascript para o desenvolvimento de interfaces de usuário de aplicações *web*. Com essa biblioteca, podemos estruturar as páginas em diferentes componentes e simplificar o uso de estados em cada página. (SOURCE, 2024; FACEBOOK JOSH STORY, 2024)

É possível manipular as páginas facilmente misturando Javascript com HTML. Além disso, é possível simplificar e organizar o gerenciamento do estado da aplicação passando parâmetros para cada componente React.

Um componente React pode ser uma página inteira ou um pequeno trecho de uma página, portanto, cada componente pode ter seu próprio HTML, Javascript e CSS.

3.2.1 Typescript

Typescript é uma extensão da linguagem Javascript, com a adição de que permite a tipagem das variáveis e métodos, permitindo maior facilidade ao programar com a IDE e maior segurança de que os dados estão corretos, pois ele resultará em erro de compilação caso haja algum problema com algum tipo de parâmetro. (MICROSOFT, 2024j; MICROSOFT, 2024i)

Um exemplo de como tipar uma função escrita em Typescript, com os parâmetros para $number$ e o retorno para $Promise < void >$ seria:

```
async function  GetDataForGallery(top:number,skip:number):Promise<void>
{
   ...
}
```

A Figura 3.1 é um exemplo de como os tipos são visualizados na IDE Visual Studio Code.

Figura 3.1: Screenshot da visualização do tipo de retorno de DataGalleryState.getTree()

Fonte: do autor.

3.2.2 Ant Design

Ant Design (antd) é um *template* para componentes com designs padronizados. Temos essa implementação do Antd feita para React, Angular e outros *frameworks*. Utilizar esta biblioteca permite maior facilidade ao inserir componentes prontos, como modais, headers, carrousels, formulários, entrada de dados e tabelas. (GROUP, 2024b; GROUP, 2024a)

Alguns exemplos de empresas que usam Ant Design são: Alibaba,Tencent e Baidu. Um exemplo de Formulário utilizando Antd é apresentado na Figura 3.2.

Figura 3.2: Exemplo de formulário que segue o padrão Antd

Fonte: https://ant.design/components/form

3.2.3 Axios

Axios é um cliente HTTP para Node, que utilizan o conceito de Promises, podendo rodar no navegador e no Node.Js com o mesmo código, no nosso caso ele é utilizado no navegador, utilizando XMLHttpRequests. Com Axios temos uma maior facilidade para configurar métodos REST, inserir *middlewares* de autenticação e mudar o comportamente de acordo com a resposta do servidor. (SARJEANT, 2024; ZABRISKIE; CONTRIBUTORS, 2024)

3.2.4 React Tree Store

React Tree Store é um pequeno pacote Typescript que permite controlar o estado dos componentes de maneira compartilhada de maneira simples (SCHöLLER, 2021).

3.2.5 Blob

Um objeto imutável que representa um arquivo, pode ser lido como texto ou dados binários. (FOUNDATION, 2024a)

3.2.6 Local Storage

Local Storage é o local onde ficam salvos dados dentro do navegador do usuário, utilizamos a Local Storage para salvar o token de autenticação do usuário (FOUNDATION, 2024b)

3.3 Microsoft SQL Server

Foi utilizado o banco de dados Microsoft SQL Server pelos seguintes motivos:

- **Curva de Aprendizado**: Como eu já havia trabalhado com Microsoft SQL Server, decidi utilizá-lo para evitar qualquer problema.
- **Compatibilidade com .NET**: Apesar da biblioteca .NET EF permitir qualquer banco de dados, o padrão é Microsoft SQL Server, então é possível poupar algumas

linhas de código de configuração com essa escolha.

- **Compatibilidade com Linux**: Nas últimas versões, o Microsoft SQL Server possui total compatibilidade com as versões atuais de Linux.

(MICROSOFT, 2024f)

3.4 Ngrok

Ngrok é um software que permite disponibilizar o computador ou aplicações específicas na internet, de maneira simples e rápida. Foi utilizado para disponibilizar o *frontend* e o *backend* para ser acessado por usuários externos para testes. (NGROK, 2024)

4 MODELAGEM E PROJETO

A implementação do sistema foi realizada utilizando princípios como boas práticas de programação, Clean Coding, Design Patterns, Clean Architecture e tecnologias que sejam utilizadas no mercado de trabalho. Neste capítulo será descrito o processo de levantamento de requisitos, e de desenvolvimento da arquitetura e do código do projeto. (MARTIN, 2008)

4.1 Requisitos

4.1.1 Levantamento de Requisitos

Primeiramente, foi realizada uma reunião para identificar quais as necessidades do projeto, os participantes da reunião foram o desenvolvedor, professor orientador, e participantes do projeto, junto com a bióloga responsável pelo registros dos itens. Foram definidos 3 tipos de usuários:

- **Administrador** Usuário administrador do sistema, será utilizado pelo curador do museu para gerenciar os itens do museu.
- **Pescador** O pescador é o usuário que utilizará o boletim de condições do mar, onde ele irá registrar a condição do mar visualizados no momento atual e registrar algumas condições como: lixo encontrado na praia, presença de algas no mar, avistamento de algum animal, etc.
- **Público** São usuários que não estarão logados no sistema, vão acessar a aplicação para estudar e pesquisar os itens do museu.

No final da reunião foram definidas que as seguintes funcionalidades eram necessárias:

- **Cadastro de Itens** Será necessário uma tela tela para o administrador do museu registrar os itens do museu. Uma tela similar será utilizada para atualizar itens já adicionados.
- **Deletar Item** O administrator precisará poder deletar itens previamente incluídos.
- **Visualizar Itens do Museu** O administrador deverá poder visualizar todos os itens cadastrados e poder filtrar por suas propriedades como nome popular e taxonomia.

- **Visualização de Itens pelo público** Qualquer pessoa deve poder acessar o sistema e visualizar os itens cadastrados, podendo filtrar pelas propriedades, para fins educacionais e de pesquisa.
- **Registro de Boletim do Mar** Tela acessada pelo pescador, para registrar informações relacionadas ao mar observadas por ele naquele momento.

Além da implementação das funcionalidades na tecnologia escolhida, será necessário publicar a aplicação para ser utilizada na internet.

4.1.2 Organização das Entregas

Por questões de cronograma, não foi possível organizar mais reuniões com a bióloga, então foi definida a seguinte prioridade para entrega das funcionalidades:

- **Criação Inicial do Projeto** Configurar todas as tecnologias utilizadas na implementação
- **Visualização e Cadastro de Itens** Primeiro seria implementada a tela de galeria, para o administrator poder visualizar os itens e cadastrar novos itens, primeiramente sem as imagens.
- **Atualizar,Deletar Itens e Adicionar e Deletar Imagens** Foram implementadas todas as funcionalidades para salvar e deletar as imagens no disco, junto com a atualização dos itens.
- **Autenticação e Visualizar Itens Públicos**: O administrador deverá poder logar no sistema para visualizar a página de galeria e adicionar itens. Criação da tela para visualizar itens públicos por usuários não autenticados.
- **Paginação e Filtros**: Ambas as telas de visualização devem permitir filtrar e visualizar apenas 25 registros por vez.
- **Registro de Boletim do Mar**: Criação da Tela do Boleti do Mar para ser acessada pelo usuário pescador.

4.1.3 Modelagem de Dados

Após a reunião inicial, a bióloga responsável enviou um documento detalhando as informações necessárias para registrar as propriedades de um item do museu e as propri-

edades do boletim do mar, juntamente com exemplos de artigos que já foram cadastrados manualmente em um documento Word. No documento de requisitos constava as seguintes informações sobre as propriedades para cadastro de uma espécie:

- **Foto** É possível armazenar múltiplas fotos do artigo.
- **Nome popular** Único campo obrigatório para salvar.
- **Espécie**
- **Gênero**
- **Família**
- **Ordem**
- **Classe**
- **Link para trabalhos**
- **Referências**
- **Campo para descrição de características**

Juntamente com as seguintes informações para cadastro do boletim de condição do mar:

- **Temperatura**
- **Direção do vento**
- **Maré**

Após a análise dos documentos, que resultou na seguinte modelagem, os detalhes específicos e chaves estrangeiras dos tipos de dados foram gerados automaticamente através das migrations do EF.

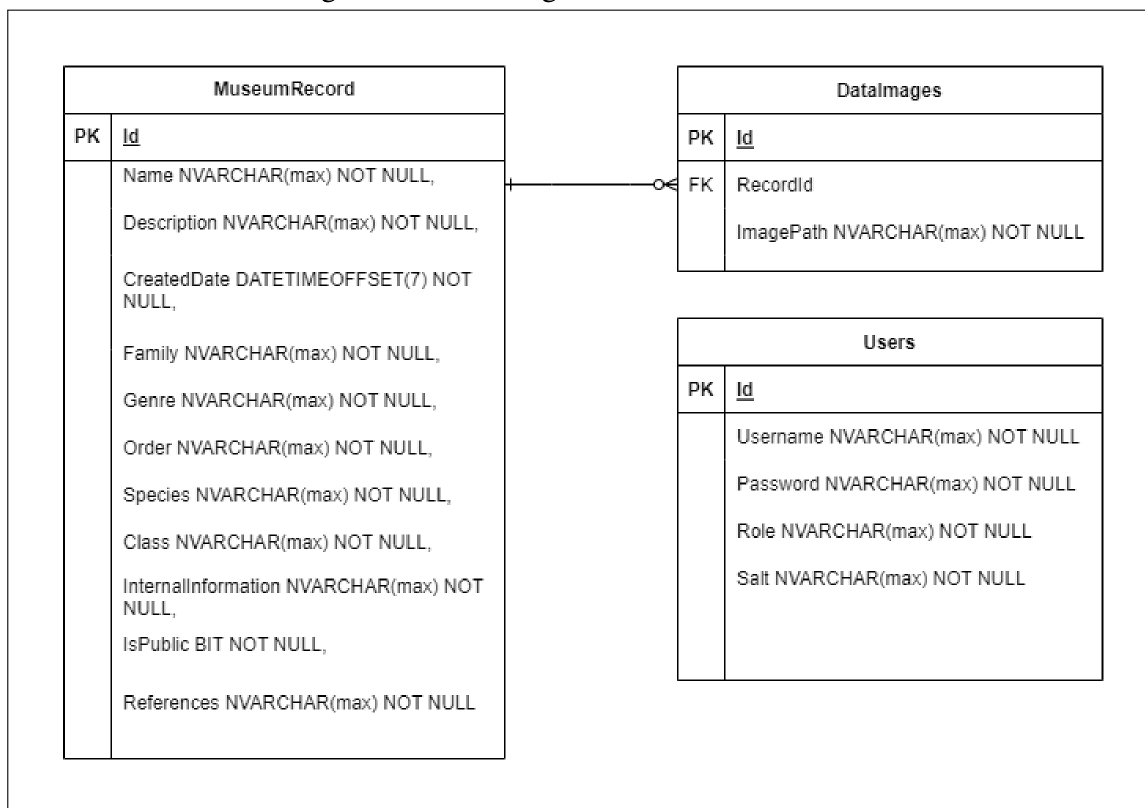
Um MuseumRecord(Registro de Museu) irá possuir diversos dados relacionado a taxonomia do animal, além de informações administrativas e internas para controlar a visibilidade do registro. O campo para definir se um registro é público, e o campo para informações internas foram definidos no final da implementação, como uma maneira de permitir que o administrador mantenha informações privadas sem precisar deletar os dados.

Um dado do tipo DataImages necessita do RecordId para identificarmos a qual registro de museu esta imagem pertence, além do ImagePath, que descreve o nome do arquivo salvo no servidor.

Um user possui seu nome de usuário e senha, além do campo Role, para futuras implementações onde possa ser necessário, e o campo Salt que armazena uma *hash* única da senha.

A Figura 4.1 demonstra a modelagem do banco de dados.

Figura 4.1: Modelagem do Banco de Dados



Fonte: do autor.

4.2 Arquitetura do Sistema

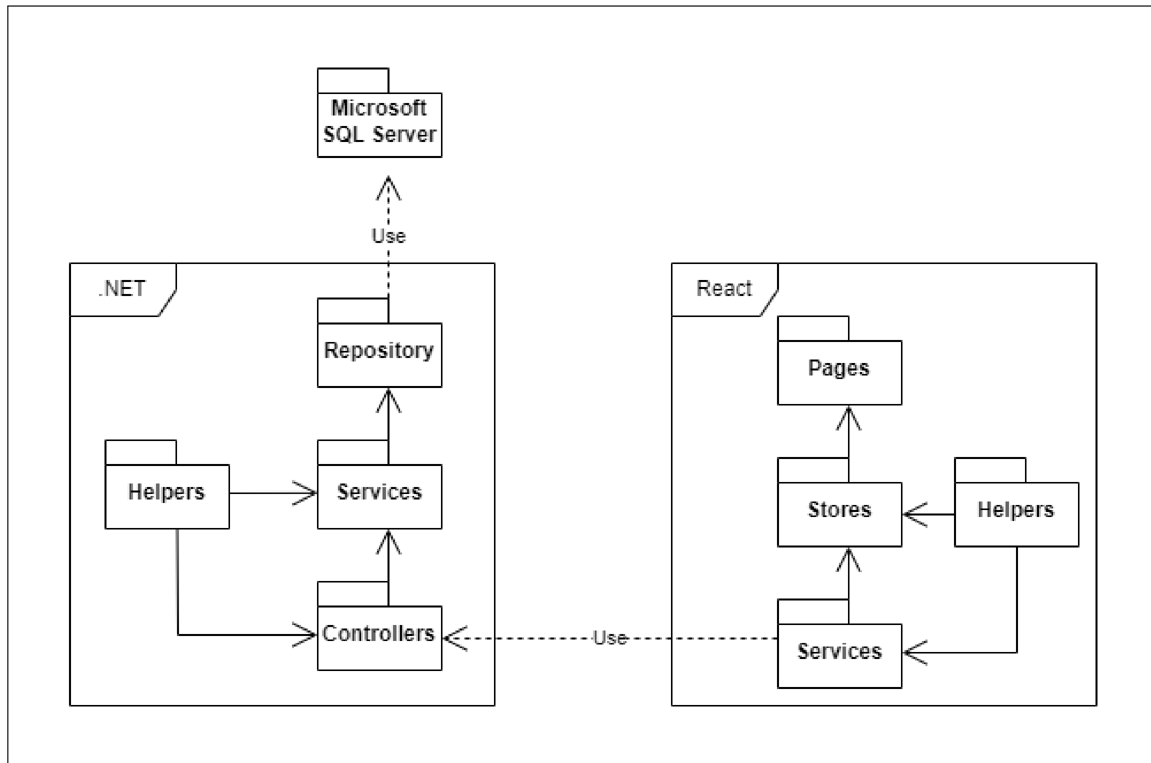
O sistema foi separado em diversos pacotes para facilitar a sua manutenção, onde um pacote só se comunica com a camada interior. A Figura 4.2 demonstra o diagram de pacotes, mostrando cada camada da aplicação e como o *frontend* se comunica com o *backend*.

4.2.1 Frontend

O *frontend* foi implementado usando React para facilitar a curva de aprendizado, pois já possuo conhecimentos da tecnologia e de suas bibliotecas. O *frontend* foi dividido nos seguintes pacotes:

- **Services**: É a camada que se comunica com a API, utilizando a biblioteca Axios efetuamos as chamadas REST para o servidor. Nesta camada temos o arquivo Base-

Figura 4.2: Diagrama de Pacotes da Aplicação



Fonte: do autor.

Service, que implementa os métodos REST básicos, além de um método específico para OData e outro para download e envio de imagens.

Outros arquivos nesta camada são MuseumRecordService e RecordImagesService, que efetuam as operações CRUD e de upload e download de imagem utilizando o BaseService.

- **Entities**: Esta camada não está no diagrama por não fazer parte do fluxo de dados, mas ela armazena os tipos de dados que são retornados da API ou que são utilizados internamente na aplicação.

  Dois arquivos desta camada são: MuseumRecord e RecordImagem, que armazenam as propriedades das entidades de maneira similar ao que é retornado pela API.
- **Stores**: Esta camada efetua a ligação entre as páginas e a Service, onde obtém os dados da Service e é capaz de atualizar os dados que são observados pela camada Pages. Além de repassar os dados da Pages para a Service salvá-los.
- **Pages**: Camada que armazena todas as páginas e seus componentes, utiliza a Store para obter e salvar os dados necessários.
- **Helpers**: Possui métodos utilitários para manipular imagem e exibir nas páginas e também gerenciar os cookies na LocalStorage.

Além disso, temos o arquivo Router, que captura a rota do navegador e encaminha para o componente que representa determinada página.

4.2.2 Backend

O *backend* foi implementado usando .NET para diminuir a curva de aprendizado, apesar de já possuir conhecimento na tecnologia, há muitas noções que foram necessárias aprender, como autenticação e configurar o projeto do zero, definindo a injeção de dependências e a conexão com o banco de dados. O backend foi dividido nos seguintes pacotes:

- **Migrations**: Esta camada não está no diagrama pois não faz parte do fluxo da API, ela armazena todas as alterações aplicadas as entidades do banco de dados, de maneira que é possível desfazer *migrations* executadas no banco de dados incrementalmente.
  Outros arquivos nesta camada são MuseumRecordService e RecordImagesService, que efetuam as operações CRUD e de upload e download de imagem utilizando o BaseService.
- **Domain**: Esta camada representa todas as Entidades, que são mapeadas para o banco de dados através de suas propriedades e de suas classes de Configuração.
  Dois arquivos desta camada são: MuseumRecord e RecordImagem, que armazenam as propriedades das entidades de maneira similar ao que é retornado pela API.
- **Repository**: Esta camada realiza operações simples e se comunica com o banco de dados, realizando operações CRUD para cada entidade.
- **Services**: Esta camada realiza operações e cálculos mais complexos, utilizada para ligar a Controller à Repository. Idealmente deve ser uma Service para cada Entidade para não quebrar o princípio de responsabilidade única .
- **Controllers**: Camada que disponibiliza as rotas da API, sendo uma controller por entidade, implementando métodos HTTP de GET,POST,PUT e DELETE que são mapeados para métodos equivalentes na Service. (BARRY, 2024)

5 IMPLEMENTAÇÃO

Neste capítulo, veremos detalhes específicos de implementação das partes mais importantes do projeto.

5.1 Frontend

A aplicação React está organizada com a estrutura específicada no capítulo 9.1.

5.1.1 Service

A service é a camada onde são efetuadas as requisições para a API. Como precisamos nos comunicar com o servidor, onde alguns endpoints foram implementados em OData, é recomendável utilizar alguma biblioteca de OData para o frontend montar as requisições. Foi utililizada a biblioteca odata-query-builder que permite montar *queries* OData facilmente. (MAHAN, 2024)

Exemplo de método que utiliza a biblioteca odata-query-builder, é possível notar a utilização do método count(), para especificar que queremos o número total de elementos da tabela que correspondem a *query*, top() e skip(), que específicam o tamanho e qual página da busca queremos, e o expand(), que determina que queremos obter além dos MuseumRecords, também as Images desse registro. AO enviar a solicitação para o *backend* no método getOdata, concatenamos a query na rota da API como um *query parameter*. No final retornamos esse *count* através da propriedade "@odata.count", gerado automaticamente pela biblitoeca OData no backendm junto com o valor da *response*:

```
1 async function  GetRecords(top:number,skip:number):Promise<{data:
    MuseumRecord[],count:number}> {
2    const query = new QueryBuilder()
3    .count()
4    .top(top)
5    .skip(skip)
6    .expand("Images")
7    .toQuery()
8    const response = await BaseService.getOdata(route+query);
9    return {data:response.data.value as MuseumRecord[],count:response.
     data["@odata.count"]};
```

```
}
```

- **BaseService**: O arquivo BaseService.tsx implementa os 4 métodos CRUD e mais um método OData e um middleware que adiciona o Token de Autenticação ao Header da Requisição, além de configurar um comportamente específico para erros. Neste trecho, podemos ver na linha 1 que a URL para a API está salva como variável de ambiente, como é criado e configurado a requisição através dessa URL,juntamente com o middleware interceptor na linha 5 que salva o *header* de Autenticação.

```
const endpoint = process.env.REACT_APP_BACKEND_API

const API = axios.create({baseURL:endpoint})

API.interceptors.request.use(config => {
    const authToken = LocalStorageHelper.getAuthToken()
    if(authToken)
        config.headers.setAuthorization(`bearer ${authToken}`)

    return config
}))
```

Neste trecho, podemos ver como é criado os métodos GET, POST, PUT, DELETE, GetOData e Download para imagens. É possível notar nos parâmetros que não temos a URL da API, pois ela já foi gerada anteriormente. Recebemos somente o nome da entidade, definida na variável modelName, e a rota, definida na variável route.

```
function get<Type,Params>(modelName:string,route = '',query?:
    Params):Promise<AxiosResponse<{value:Type}>>
{
    return API.get(`${modelName}${route}`)
}

function getOdata<Type,Params>(modelName:string,route = '',query?:
    Params):Promise<AxiosResponse<{value:Type,"@odata.count":
    number}>>
{
    return API.get(`${modelName}${route}`)
}

```

```
function download<Type,Params>(modelName:string,route = '',query?:
    Params):Promise<AxiosResponse<{value:Type}>>
{
    return API.get(`${modelName}${route}`,{responseType:'blob'})
}

function post<Type,Response>(modelName:string,data:Type,route =
    ''):Promise<AxiosResponse<Response>>
{
    return API.post(`${modelName}${route}`,data)
}

function put<Type,Response>(modelName:string,data:Type,route = '')
    :Promise<AxiosResponse<Response>>
{
    return API.put(`${modelName}${route}`,data)
}

function remove(modelName:string,route = ''):Promise<undefined>
{
    return API.delete(`${modelName}${route}`)
}
```

- **MuseumRecordService**: O arquivo MuseumRecordService.tsx realiza a comunicação com a Controller da API de MuseumRecords, possui um Get para registros públicos e outro para registros privados, além de um Save,Delete e Update. No trecho abaixo podemos ver como é utilizado a BaseService para efetuar as operações:

```

      async function SaveRecord(record:MuseumRecord) {
    var response = await BaseService.post<MuseumRecord,
   MuseumRecord>(route,record);
    return response.data.id
   }

   async function DeleteRecord(record:MuseumRecord){
      await BaseService.remove(`${route}/${record.id}`)
   }

   async function UpdateRecord(record: MuseumRecord){
      await BaseService.put(route,record);
```

No trecho abaixo, a implementação de GetRecords, que mostra todos os registros, incluindo privados, pode-se notar a utilização da biblioteca odata-query-builder para definir o *count*,*top*,*skip* e *expand* e convertê-los para uma *string* com a*query* construída. Realizamos o expand("images") para obter as imagens relacionadas ao registro.

```
async function  GetRecords(top:number,skip:number):Promise<{data:
    MuseumRecord[],count:number}> {
   const query = new QueryBuilder()
   .count()
   .top(top)
   .skip(skip)
   .expand("Images")
   .toQuery()
   const response = await BaseService.getOdata(route+query);
   return {data:response.data.value as MuseumRecord[],count:
    response.data["@odata.count"]};
  };
```

5.1.2 Entities

Temos apenas duas entidades, uma para Registros de museu e outra para Imagens. Temos todas propriedades de taxonomia do Registro de Museu, além de informações internas, data de criação, booleano para definir se é público e um array com as imagens. Todas são strings, com exceção do Id e da images, o Id é do tipo *number*, e images é do tipo RecordImage ou *undefined*, pois pode acontecer de não termos uma imagem.

```
export interface MuseumRecord {
    id: number;
    name: string;
    species: string;
    genre: string;
    family: string;
    order: string;
    researchLinks: string;
    description: string;
    createdDate: string;
    isPublic : boolean;
    internalInformation:string
```

```
13      references:string
14      images: (RecordImage | undefined)[];
15    }
```

Para as imagens, temos o Id da imagem, a representação em Bytes como Blob na linha 3 e a URL da imagem que é construída internamente através do Blob com Javascript, essa é a URL utilizada para visualizar a imagem na página.

```
1 export interface RecordImage {
2     id:number;
3     imageBytes: Blob | undefined
4     imageBlobUrl: string | undefined
5 }
```

5.1.3 Stores

As stores são responsáveis por chamar a Service para obter ou salvar os dados, e atualizar as Páginas com esses dados atualizados. Temos um arquivo na store para cada operação CRUD necessária, para cada Entidade.

Como necessitamos carregar a imagem junto com os registros, o arquivo da Store GetDataForGallery.tsx ficou complexo.

Primeiramente carregamos o estado atual na linha 2, definimos que o estado está com *loading* na linha 3 para poder exibir na página um indicativo de que estamos buscando os dados, então verificamos se estamos autenticados na linha 5, para decidir se é chamado o endpoint que busca todos os dados ou somente registros públicos.

Então, com o resultado, iteramos sobre os registros obtidos na linha 7 para obter o Id da imagem e fazer uma requisição de download de cada imagem na linha 8 e 9.

No final, encerramos o *loading* na linha 16, e atualizamos o estado com todos os valores na linha 19.

```
1 async function  GetDataForGallery(top:number,skip:number):Promise<void>
      {
2   const state = DataGalleryState.getTree();
3   state.loading = true;
4   DataGalleryState.update()
5   const data = LocalStorageHelper.isAuthenticated()?  await
     MuseumRecordService.GetRecords(top,skip) : await
     MuseumRecordService.GetPublicRecords(top,skip)
6
```

```
7   const dataWithImages = await Promise.all(data.data.map(async record
     =>{
8     record.images = await Promise.all(record.images.map(async image =>{
9         var imageBytes = await RecordImagesService.GetImageById(image!.
     id);
10        return  {id:image?.id,imageBytes:imageBytes} as RecordImage
11     }))
12     return {...record}
13  }))
14
15  state.data = dataWithImages
16  state.count = data.count
17  state.loading=false
18
19  DataGalleryState.update()
20  }
```

5.1.4 Pages

Nas páginas, temos a Galeria, onde são efetuadas operações administrativas, a Página Inicial, onde é possível visualizar registros públicos, e a página de Login.

As operações administrativas são realizadas através de modais que abrem na página de Galeria.

No trecho abaixo temos um exemplo de como uma modal para editar é gerenciada na galeria. É possível perceber na linha 1, 3 a 6, 8 a 10 a lógica utilizada para gerenciar a *modal* de Update de um registro, onde manipulamos o estado para definir com qual registro estamos interagindo. Na linha 13 temos a modal para atualizar os dados, ela somente é exibida se a variável isUpdateModalOpen estiver como *true*, isso é gerenciado no método showUpdateModal na linha 3.

```
1  const [isUpdateModalOpen, setIsUpdateModalOpen] = useState(false);
2         ...
3   const showUpdateModal = (record:MuseumRecord) => {
4     setInteractingRecord(record)
5     setIsUpdateModalOpen(true);
6   };
7
8     const handleCancelUpdate = () => {
```

```
9      setIsUpdateModalOpen(false);
10   };
11 return <div id="gallery">
12        ...
13     <UpdateData record={interactingRecord!} handleCancel={
       handleCancelUpdate} handleOk={handleOkUpdate} isModalOpen={
       isUpdateModalOpen}></UpdateData>
14 </div>
15
16
```

No trecho abaixo temos um exemplo de como uma modal para inserir um registro é implementada, É possível perceber na linha 3 a 2 a lógica utilizada para salvar o registro, onde salvamos na linha 8 e salvamos as imagens na linha 11, da linha 15 a 41 temos o trecho React que específica o layout do formulário, é possível notar da linha 18 a 22 um *input* do tipo texto para o nome do registro.

```
1 function  RegisterData({isModalOpen,handleCancel,handleOk,record}:Props
     ) {
2         ...
3 const saveRecord = async () =>{
4   if(museumRecord)
5   {
6     museumRecord.id = record.id;
7     (museumRecord as any).photo = undefined
8     await EditMuseumRecord(museumRecord)
9
10    if(fileList?.length > 0)
11       await SaveMuseumRecordImages(museumRecord.id,fileList.filter(
     file => file.isNew).map(file => file.file))
12  }
13        ...
14
15 <Modal forceRender title="Atualizar" width="80%" open={isModalOpen}
     okText="Salvar" cancelText="Cancelar" onCancel={handleCancel} onOk
     ={saveRecord}>
16          <Form>
17
18        <Form.Item
19          name="name"
20          label="Nome Popular"
21          rules={[{ required: true, message: 'Por favor, insira um nome
```

```
    para o registro' }]}
22          >
23            <Input />
24          </Form.Item>
25          ...
26          </Form>
27 </Modal>
```

5.1.5 Router

O componente de Router implementa a funcionalidade do React de gerenciar as rotas de navegação, foram implementadas 3 rotas e uma *default*, que redireciona para os componentes das páginas específicas de cada rota baseado no caminho do navegador.

```
1  const RouterComponent = () => {
2    return (
3
4      <Routes>
5      <Route path="/login" element={<Login />} />
6      <Route path="/gallery" element={<DataGallerySetup />} />
7      <Route path="/home" element={<HomePageSetup />} />
8      <Route
9        path="*"
10       element={<Navigate to="/home" replace={true} />}
11     />
12   </Routes>
13   );
14 };
```

5.1.6 Helpers

Temos duas classes helpers, uma para converter Blob em URL de Imagem e outra que gerencia a Autenticação na Local Storage do Navegador. Ambas usam métodos nativos do Javascript para efetuar essas operações.

5.2 Backend

A estrutura de diretórios do projeto Backend .NET está específicada no capítulo 9.2.

5.2.1 Domain

Na Domain, declaramos as classes da entidades e classes de configuração dessas entidades, que serão mapeadas para tabelas do banco de dados.

Abaixo está a declaração da classe MuseumRecord, que irá gerar a tabela MuseumRecord como definida na configuração. Além disso, temos um array de RecordImage, que é outra entidade do sistema, isso irá resultar numa relação 1:Many que será gerada automaticamente no banco de dados.

```
1     public class MuseumRecord
2     {
3         public int Id { get; set; }
4         public string Name { get; set; } = string.Empty;
5         public string Species { get; set; } = string.Empty;
6         public string Genre { get; set; } = string.Empty;
7         public string Family { get; set; } = string.Empty;
8         public string Order { get; set; } = string.Empty;
9         public string ResearchLinks { get; set; } = string.Empty;
10        public string Description { get; set; } = string.Empty;
11        public string Class { get; set; } = string.Empty;
12        public string InternalInformation { get; set; } = string.Empty;
13        public string References { get; set; } = string.Empty;
14        public bool IsPublic { get; set; } = true;
15        public DateTimeOffset CreatedDate { get; set; }
16        public IEnumerable<RecordImage>? Images { get; set; }
17    }
```

Abaixo temos parte da configuração da classe MuseumRecord, é possível definir se as colunas mapeadas serão nullable, o tamanho máximo da string, qual a chave primária e como são relacionadas as chaves estrangeiras.

```
1 public class MuseumRecordConfiguration : IEntityTypeConfiguration<
    MuseumRecord>
2 {
3     public void Configure(EntityTypeBuilder<MuseumRecord> entity)
```

```
4       {
5           entity.ToTable("MuseumRecord");
6           entity.Property(e => e.Name)
7               .IsRequired()
8               .HasMaxLength(128);
9           entity.Property(e => e.Order)
10              .IsRequired(false)
11              .HasMaxLength(128);
12          entity.Property(e => e.Species)
13               .IsRequired(false)
14              .HasMaxLength(128);
15              ...
16      }
17  }
```

5.2.2 Repository

A função da repository é efetuar no banco de dados, como buscar e alterar registros de determinada entidade. Temos um *repository* para cada entidade do sistema.

Neste trecho temos a implementação de RecordImagesRepository, bastante simplificado pois herda todos os métodos da classe base Repository, que será explica em mais detalhes na subseção de Design Patterns.

```
1      public sealed class RecordImagesRepository : Repository<RecordImage
    >, IRecordImagesRepository
2       {
3           public RecordImagesRepository(MuseuDoMarContext context):base(
    context)  { }
4       }
5   }
```

5.2.3 Services

As Services realizam operações complexas manipulando entidades, mas utilizando a Repository para obtê-las ou salvá-las.

Abaixo temos a implementação do serviço MuseumRecordsService, junto com seus métodos de Get,Add e Update.

É possível notar que temos validação de valores Null antes de efetuar operações para controlar os erros. Em cada Add e Update nos chamamos unitOfWork.SaveChangesAsync(), seguindo o *pattern* de Unit of Work para salvar qualquer alteração.

```
1       public MuseumRecordsService(IMuseumRecordsRepository repository
   , IUnitOfWork unitOfWork, IMapper mapper,IRecordImagesService
   recordImagesService)
2       {
3           this.repository = repository;
4           this.unitOfWork = unitOfWork;
5           this.mapper = mapper;
6           this.recordImagesService = recordImagesService;
7       }
8       ...
9               public IQueryable<MuseumRecord> GetQueryable()
10      {
11          return repository.GetQueryable();
12      }
13
14      public async Task<MuseumRecord> AddAsync(MuseumRecord record)
15      {
16          record.CreatedDate = DateTime.UtcNow;
17          repository.Add(record);
18          await unitOfWork.SaveChangesAsync();
19          return record;
20      }
21
22      public async Task<MuseumRecord> UpdateAsync(MuseumRecord
   updatedRecord)
23      {
24          var record = await repository.GetQueryable().Where(record
   => record.Id == updatedRecord.Id).FirstOrDefaultAsync();
25          ArgumentNullException.ThrowIfNull(record);
26          mapper.Map(updatedRecord, record);
27          repository.Update(record);
28          await unitOfWork.SaveChangesAsync();
29          return record;
30      }
```

5.2.4 Controllers

As Controllers disponibilizam as rotas da API, permitindo definir os métodos de GET, POST, PUT e DELETE para cada rota.

A seguir temos a implementação da controller de PublicMuseumRecords, utilizando como base o ODataController, limitando os resultados de cada página a 25 registros.

```
1    [ApiController]
2    [Route("PublicMuseumRecords")]
3    public class PublicMuseumRecordsController : ODataController
4    {
5        ...
6
7        [HttpGet]
8        [EnableQuery(PageSize = 25)]
9        public ActionResult<IEnumerable<PublicMuseumRecordViewModel>>
   Get()
10       {
11           var results = museumRecordsService.GetQueryable();
12           var filteredResult = results.Where(i => i.IsPublic).Include
   (x => x.Images).Select(mapper.Map<PublicMuseumRecordViewModel>);
13           return Ok(filteredResult);
14       }
15   }
```

5.2.5 Settings

Nesta pasta temos as classes de Setting, que armazenam dados relativos a variáveis de ambiente do sistema, como ImageSettings, que contêm o diretório raiz das imagens e JwtOptions, que armazena a chave privada do Token JWT.

5.2.6 OData

Nesta pasta temos o arquivo EdmModelProvider.cs, onde é configurado cada rota que utiliza Odata junto com a Entidade desta rota.

5.2.7 Migrations

Pasta que armazena os arquivos incrementais das migrations, geradas após atualizar as Entities e Configurations e rodar o seguinte comando no CLI: *dotnet ef migrations add NomeDaMigração*

Abaixo está um pequeno trecho de uma migration:

```
1      migrationBuilder.CreateTable(
2          name: "DataImages"
3          columns: table => new
4          {
5              Id = table.Column<int>(type: "int", nullable: false)
6                  .Annotation("SqlServer:Identity", "1, 1"),
7              RecordId = table.Column<int>(type: "int", nullable: false),
8              Image = table.Column<string>(type: "nvarchar(max)",
    nullable: false)
9          },
10         constraints: table =>
11         {
12             table.PrimaryKey("PK_DataImages", x => x.Id);
13             table.ForeignKey(
14                 name: "FK_DataImages_MuseumRecord_RecordId",
15                 column: x => x.RecordId,
16                 principalTable: "MuseumRecord",
17                 principalColumn: "Id",
18                 onDelete: ReferentialAction.Cascade);
19         });
20         migrationBuilder.CreateIndex(
21         name: "IX_DataImages_RecordId",
22         table: "DataImages",
23         column: "RecordId");
```

5.2.8 Mappings

Esta pasta armazena as configurações do Automapper para as Entidades, no trecho abaixo, configuramos alguns mapeameamentos de PublicMuseumRecordViewModel para MuseumRecord e vice-versa, ignorando propriedades como CreatedDate e InternalInformation para ocultar do usuário final.

```
            CreateMap<PublicMuseumRecordViewModel, MuseumRecord>()
                .ForMember(m => m.CreatedDate, opts => opts.Ignore())
                .ForMember(m => m.InternalInformation, opts => opts.
    Ignore())
                .ReverseMap();
```

5.2.9 Helpers

foram utilizados métodos Helpers para gerar Token, e gerar e autenticar o hash da senha. Abaixo temos uma das classes Helpers para gerar o *hash* da senha e verificar se a senha equivale ao *hash*, para estas operações, foi utilizada a biblioteca BCrypt.

```
public class PasswordHashHelper
{
    public static PasswordSalt HashPassword(string password)
    {
        string salt = BCrypt.Net.BCrypt.GenerateSalt();

        string hashedPassword = BCrypt.Net.BCrypt.HashPassword(password
    , salt);

        return new PasswordSalt { Password = hashedPassword, Salt =
    salt };
    }

    public static bool VerifyPassword(string password, string
    hashedPassword)
    {
        return BCrypt.Net.BCrypt.Verify(password, hashedPassword);
    }
}
```

5.3 Patterns

É possível utilizar diversos Design Patterns nativamente em .NET, já outros necessitam de algumas extensões, alguns dos utilizados são:

5.3.1 Repository Pattern

Conforme descrito previamente, temos a camada de Repository, mas utilizados uma classe base de Repository para fazer todas as operações, assim, cada classe de Repository específica que herdar desta classe

Base não precisará reimplementar operações básicas.

No exemplo abaixo, a classe base realiza as operações no contexto de uma entidade específica (TEntity) (MICROSOFT, 2024e)

```
public class Repository<TEntity> : IRepository<TEntity> where TEntity :
    class
{
    private readonly MuseuDoMarContext context;
    private readonly DbSet<TEntity> dbSet;

    public Repository(MuseuDoMarContext context)
    {
        this.context = context;
        dbSet = context.Set<TEntity>();
    }

    public IQueryable<TEntity> GetQueryable()
    {
        return dbSet.AsQueryable();
    }

    public void Add(TEntity entity)
    {
        dbSet.Add(entity);
    }
```

5.3.2 Unit Of Work

Um pattern que complementa o Repository Pattern, ao invés de vários repositórios acessarem contextos diferentes, é utililizado um único contexto para todos, permitindo gerenciar as transações, otimizar as instâncias e impedir concorrência por múltiplas atualizações de contextos. (MICROSOFT, 2024e)

5.3.3 Dependency Injection

Permite alcançar Inversão de Controle entre as classes e suas dependências, para todos Repositories e Services, foram criadas Interfaces e Classes que implementam essas interfaces, caso seja necessário uma nova classe, basta alterar no container de injeção de dependência.

Na inicialização do projeto, definimos o mapeamento entre classes e interfaces. Por exemplo, definimos que a classe MuseumRecordsRepository será injetada em todos lugares que utilizam IMuseumRecordsRepository.(MICROSOFT, 2024a)

```
builder.Services.AddTransient<IEdmModelProvider, EdmModelProvider>();

builder.Services.AddTransient<IMuseumRecordsRepository,
    MuseumRecordsRepository>();
builder.Services.AddTransient<IRecordImagesRepository,
    RecordImagesRepository>();
builder.Services.AddTransient<IUsersRepository, UsersRepository>();

builder.Services.AddTransient<IMuseumRecordsService,
    MuseumRecordsService>();
builder.Services.AddTransient<IRecordImagesService, RecordImagesService
    >();
builder.Services.AddTransient<ILoginService, LoginService>();

builder.Services.AddTransient<IUnitOfWork, UnitOfWork>();
```

Então, no construtor das classes, injetamos a interface desejada.

```

public MuseumRecordsService(IMuseumRecordsRepository repository,
    IUnitOfWork unitOfWork, IMapper mapper,IRecordImagesService
    recordImagesService)
{
    this.repository = repository;
    this.unitOfWork = unitOfWork;
    this.mapper = mapper;
    this.recordImagesService = recordImagesService;
}
```

5.3.4 Deploy

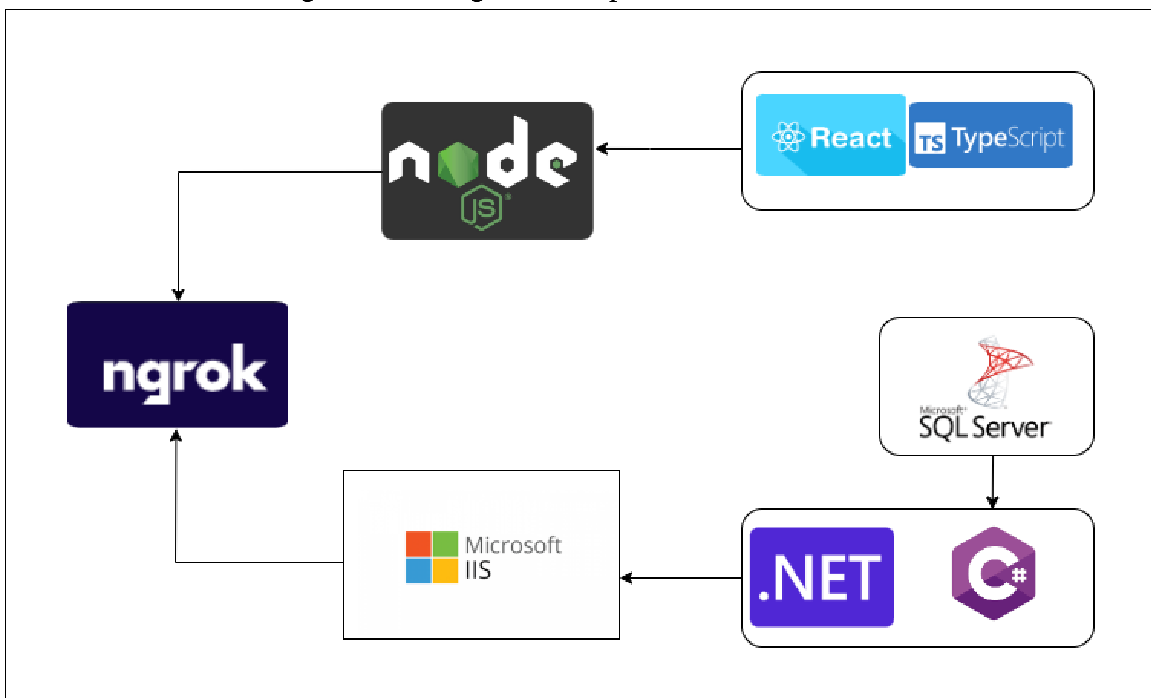
O sistema foi disponibilizado para teste na máquina local, entretanto, foi possível acessar remotamente o endereço da máquina via navegador pois foi utilizada a ferramenta Ngrok, que permite expôr aplicações locais para a internet.

Abaixo está o código de configuração Ngrok para expor o *frontend* e o *backend* para serem acessados externamente, este comando mapeia as portas 5001(.NET) e 3000(React) para um endpoint próprio público:

```
version: 2
authtoken: tokenDoUsuarioNgrok
tunnels:
  first:
    proto: http
    addr: 5001
  second:
    proto: http
    addr: 3000
```

Na figura 5.1 abaixo, é mostrada a solução atual, que disponibiliza a aplicação Frontend através de NodeJS e o backend através do Microsoft IIS, ambos disponíveis para acessar pela internet via ngrok.

Figura 5.1: Diagrama Simplificado do Sistema



Fonte: do autor.

6 MANUAL DO USUÁRIO

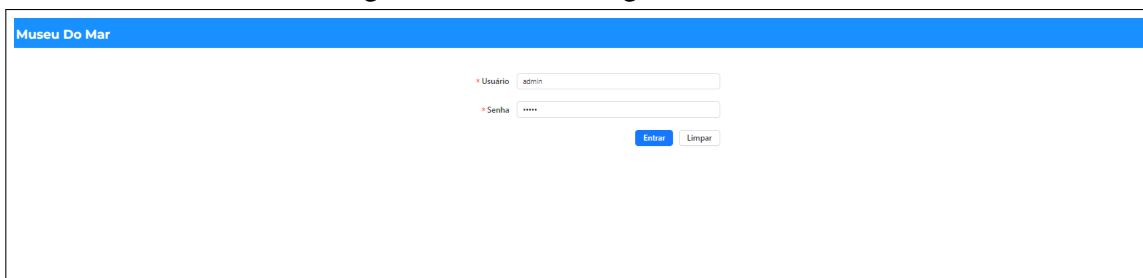
6.1 Login

Como a aplicação também é para o público em geral, foi decidido que a tela de login será acessada somente pela rota do navegador. Então, é necessário adicionar /login no final da URL do site.

O usuário padrão será admin e a senha será admin.

A Figura 6.1 mostra a tela de Login do sistema.

Figura 6.1: Tela de Login do sistema



Fonte: do autor.

6.2 Galeria

Após o Login, o usuário será automaticamente redirecionado para a Galeria, onde poderá efetuar as operações administrativas e visualizar todos os registros

A Figura 6.2 mostra a tela administrativa do sistema, e nas figuras 6.3,6.4 e 6.5 é exibida as ações de atualizar ou deletar registro.

6.3 Cadastro e Atualização

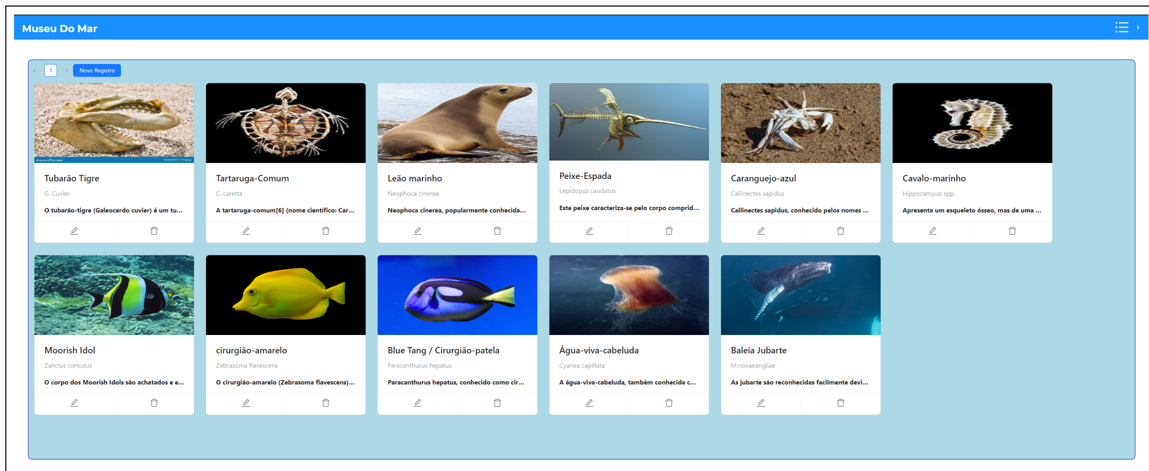
Adicione um novo registro clicando no botão Novo Registro.

Ao clicar nos ícones de um registro, é possível atualizar ou deletá-lo.

Ao clicar nos ícone de deletar, será exibido uma mensagem de confirmação.

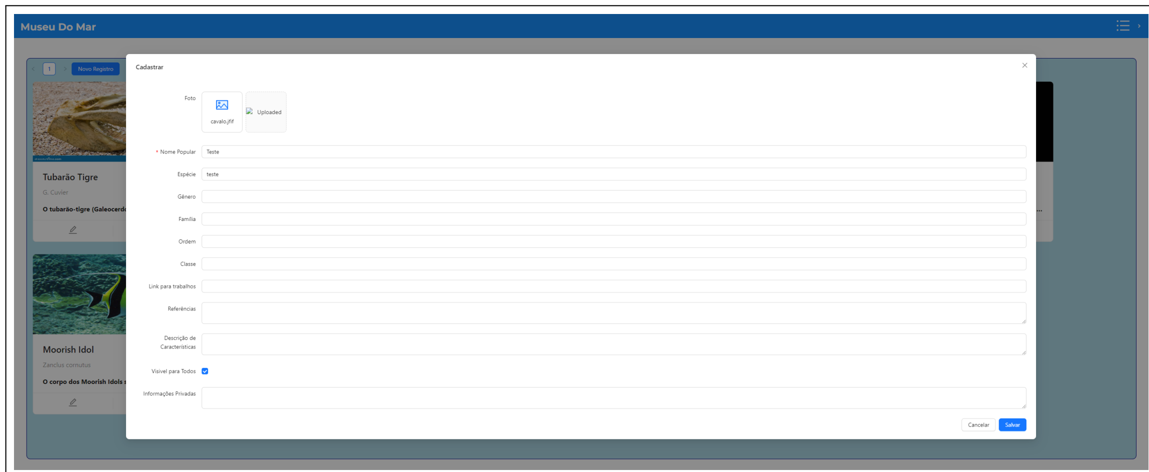
A Figura 6.6 mostra a tela de disponível para o público, visualizando somente registros públicos.

Figura 6.2: Galeria



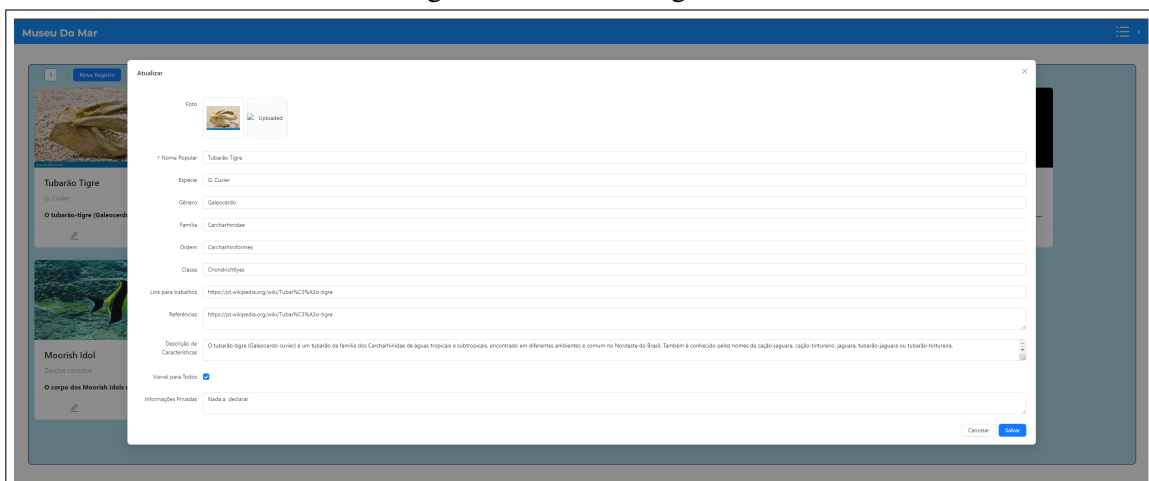
Fonte: do autor.

Figura 6.3: Novo Registro



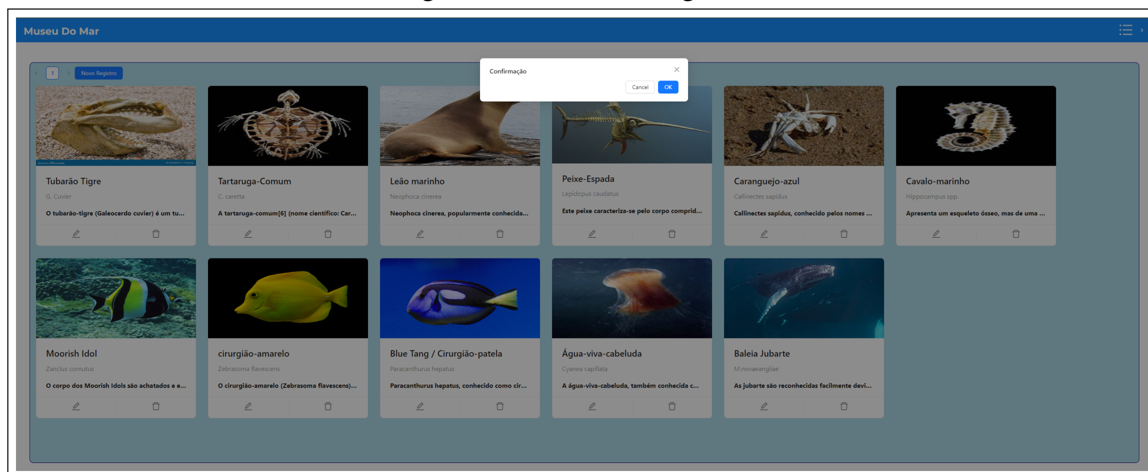
Fonte: do autor.

Figura 6.4: Editar Registro



Fonte: do autor.

Figura 6.5: Deletar Registro

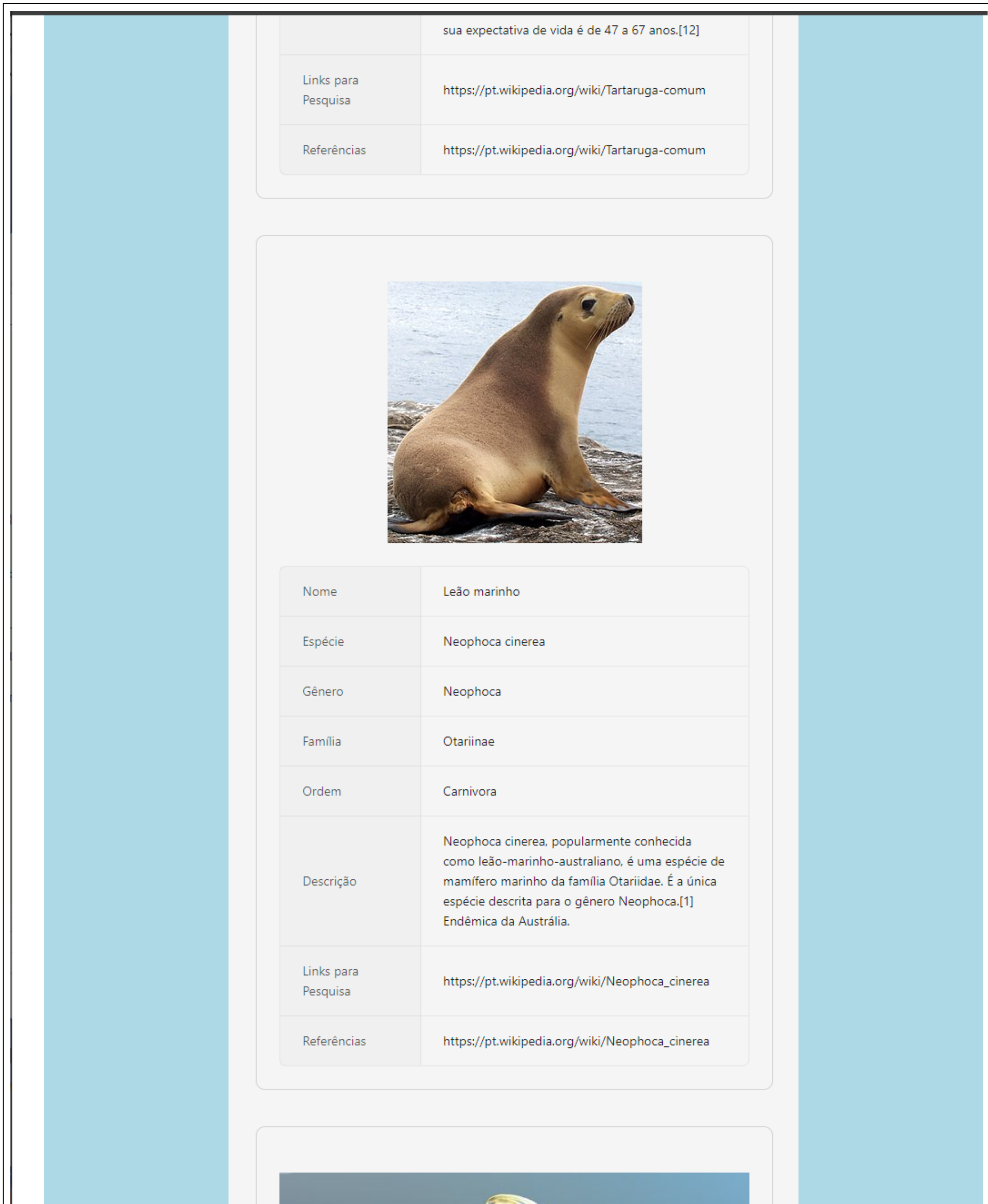


Fonte: do autor.

6.4 Página Inicial

A página inicial na figura 6.6 está disponível para o administrador e também para quem não está logado poder visualizar os registros públicos.

Figura 6.6: Página Inicial



Fonte: do autor.

7 AVALIAÇÃO E VALIDAÇÃO

Para avaliar o sistema, foi formulado um formulário de System Usability Scale e distribuido entre 5 participantes estudantes universitários, não foi possível testar com os participantes do projeto Museu do Mar..

Foi solicitado que lessem a descrição do formulário com o seguinte texto, não foi informado previamente sobre o que se tratava o sistema:

> Efetue as seguintes ações:
>
> Login no Sistema
> - Acessar a Galeria
> - Registrar Item (Ler cada campo com atenção) - Pode buscar um animal marinho no Google
> - Atualizar Item
> - Deletar Item
> - Visualizar Items na Galeria
> - Visualizar Items na Página Inicial
>
> O questionário consiste de 10 perguntas, e para cada uma delas o usuário pode responder em uma escala de 1 a 5, onde 1 significa Discordo Completamente e 5 significa Concordo Completamente.

Após realizar as tarefas, os 5 participantes responderam o formulário, onde deveriam avaliar cada questão entre 1 a 5, onde 1 significa discordar completamente, e 5 significa concordar completamente. O formulário ficou aberto para respostas durante 12 horas, nesse período os usuários realizaram os testes, cada teste durou em média 15 minutos.

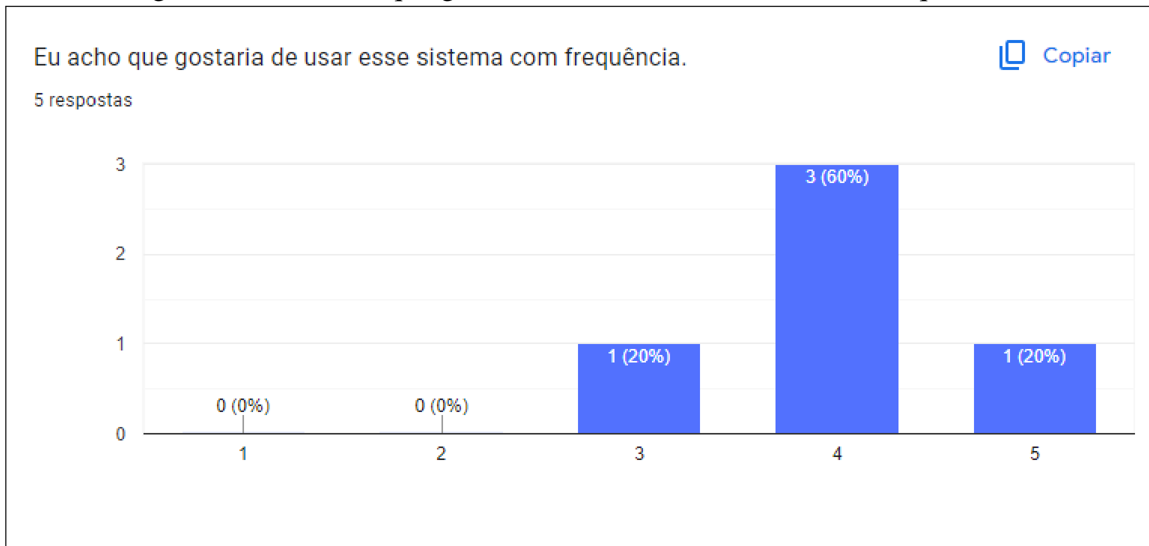
Os gráficos exibidos nas figuras 7.1, 7.2, 7.3 e 7.4, assim como nas figuras 7.5,7.6, 7.7, 7.8, 7.9 e 7.10 mostram as perguntas realizadas, juntamente com a distribuição de respostas dos participantes.

Na figura 7.1 abaixo, é possível notar que a maior parte da avaliação foi entre 4 e 5, o possível motivo para isso é de que possivelmente o usuário de imaginou como o administrador do sistema, e achou a aplicação fácil de se utilizar e útil. A nota 3 pode ser justificada pelo fato de que o usuário não é o público alvo do sistema.

Na figura 7.2 abaixo, é possível notar que ninguém achou o sistema desnecessariamente complexo, possívelmente pelo fato de que o sistema não possui nenhuma interação escondida e tudo está visível e descrito na tela. Esta mesma explicação serve para a figura 7.3, a diferença é que a nota pode ter baixado um pouco pelo fato da aplicação apresentar lentidão em determinados momentos.

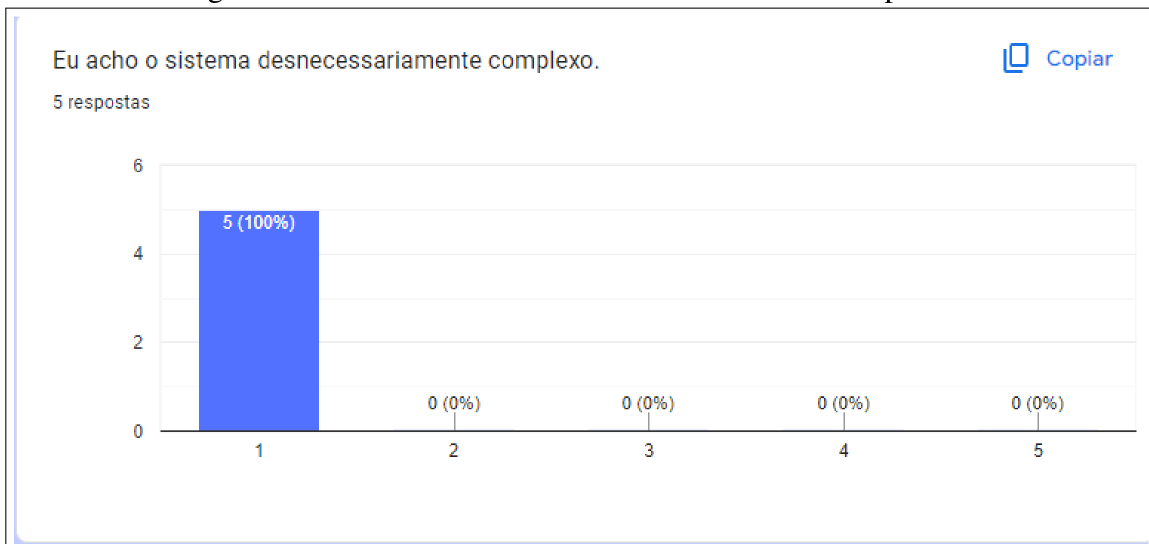
Na figura 7.4 abaixo, a pontuação ficou bastante distribuída, possivelmente porque os usuários que testaram não são o público alvo, e não possuiam os conhecimentos na área

Figura 7.1: Eu acho que gostaria de usar esse sistema com frequência.



Fonte: do autor.

Figura 7.2: Eu acho o sistema desnecessariamente complexo.



Fonte: do autor.

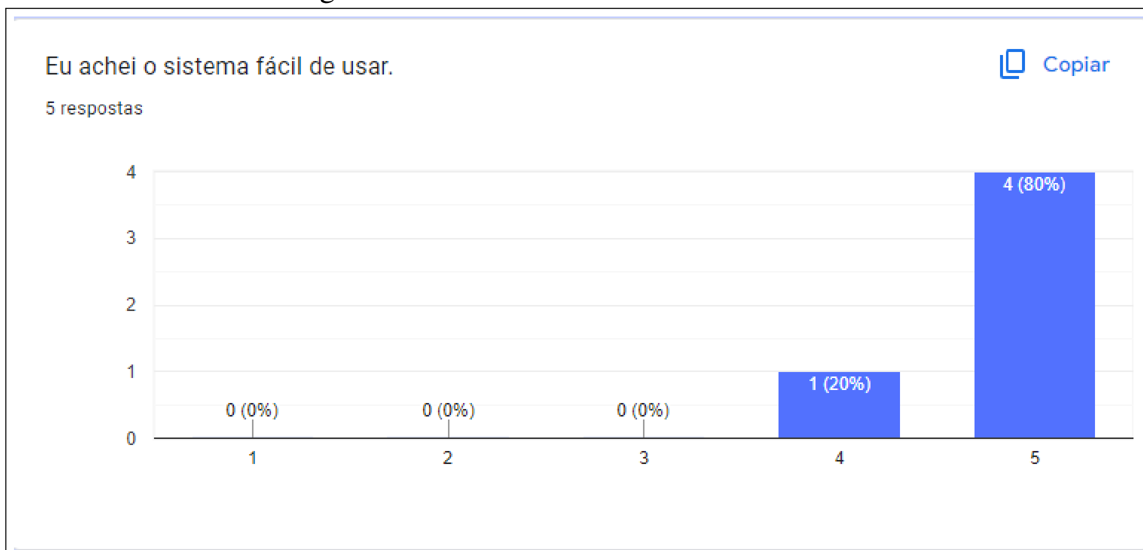
de biologia para utilizar o sistema de maneira eficiente, não foi explicada a utilidade de campos como Registro Público, Informações Internas e Referências.

Na figura 7.5 abaixo, a pontuação foi alta, possivelmente pois é fácil efetuar as ações no sistema, como inserir um registro e depois atualizá-lo.

Na figura 7.6 abaixo, é possível notar que um usuário achou o sistema inconsistente, possívelmente pelo fato de apresentar lentidão em determinados momentos ao carregar as imagens, o resto dos usuários relevou esse problema ou não teve pois foram um dos primeiros a testar o sistema.

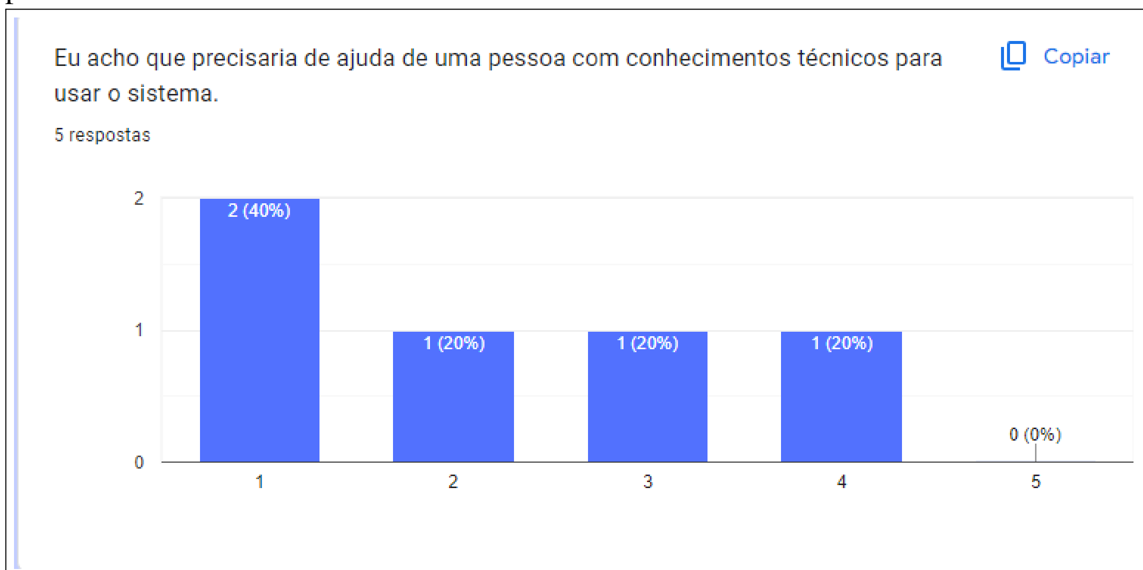
Na figura 7.7 abaixo, é possível notar que os usuários acharam o sistema fácil, e a curva de aprendizado é baixa, pois o sistema é bastante direto em apresentar as ações a

Figura 7.3: Eu achei o sistema fácil de usar.



Fonte: do autor.

Figura 7.4: Eu acho que precisaria de ajuda de uma pessoa com conhecimentos técnicos para usar o sistema.



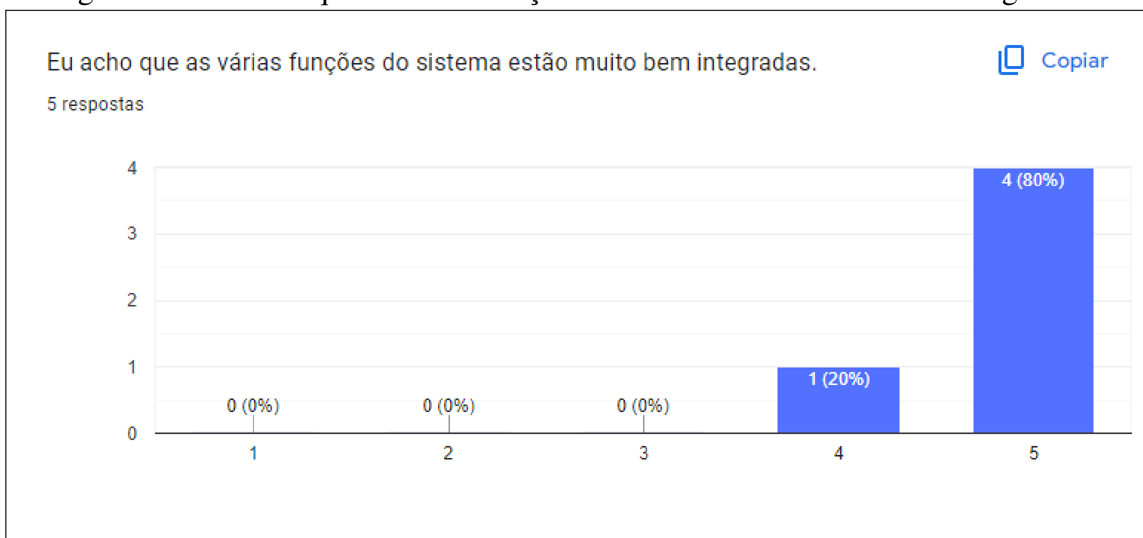
Fonte: do autor.

serem usadas.

Na figura 7.8 e 7.9 abaixo, os usuários se sentiram confiantes e não acharam o sistema atrapalhado para usar, com exceção de um usuário, novamente pelo fato da lentidão apresentada e a falta de feedback ao demonstrar que está carregando inicialmente, além da falta de confiança poder ser atribuída a necessidade de possuir conhecimentos de biologia marinha para utilizar o sistema efetivamente.

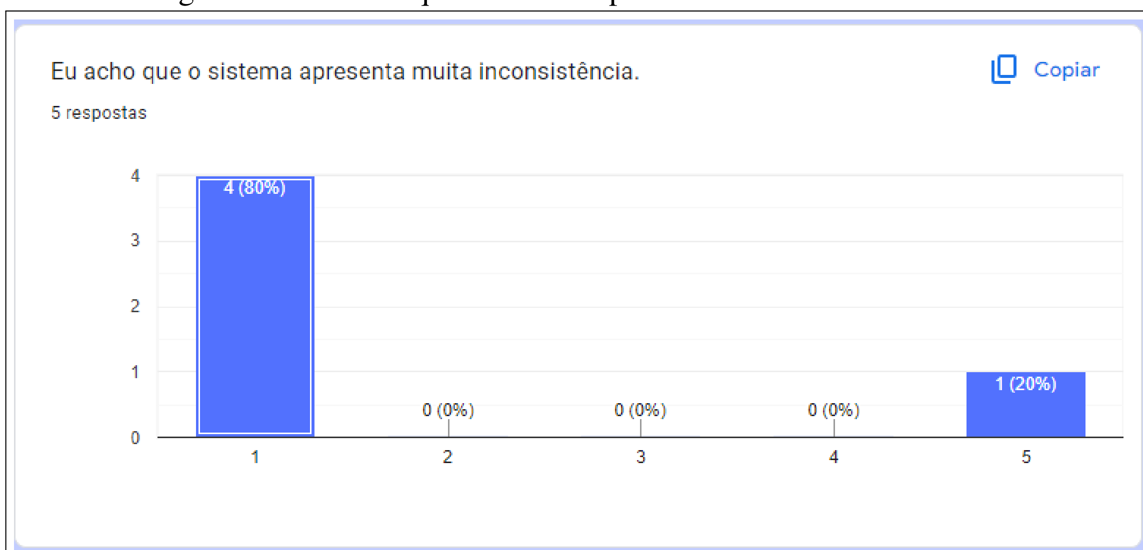
Na figura 7.10, é curioso que todos usuários consideraram que não precisaram aprender coisas novas para usar o sistema, isso conflita com algumas avaliações anteriores de confiabilidade e sistema confuso , pois provavelmente os usuários deveriam apren-

Figura 7.5: Eu acho que as várias funções do sistema estão muito bem integradas.



Fonte: do autor.

Figura 7.6: Eu acho que o sistema apresenta muita inconsistência.

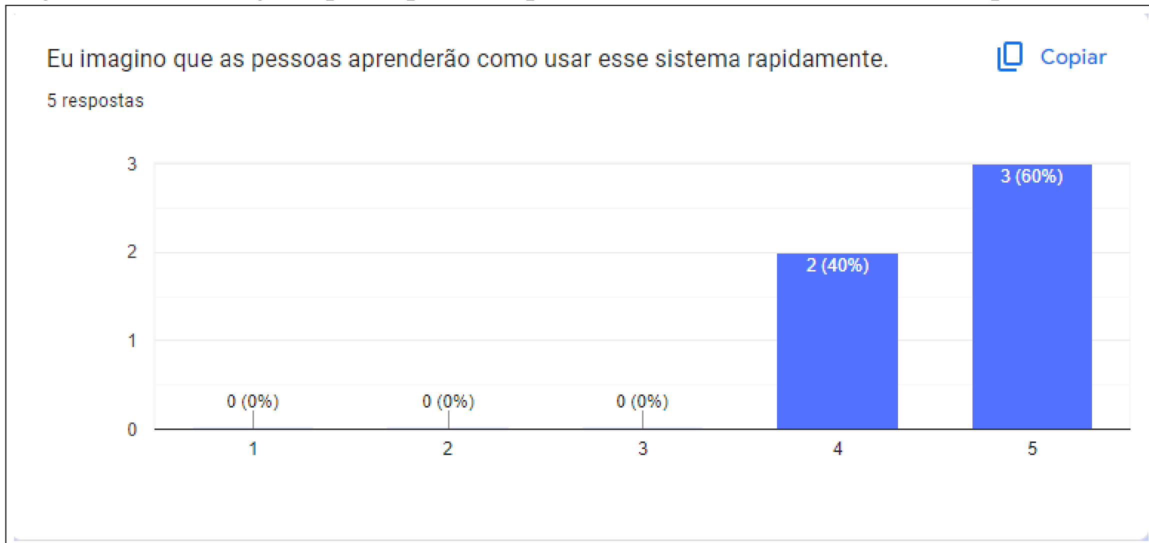


Fonte: do autor.

der e e pesquisar sobre espécies marinhas antes de usar o sistema e registrar a espécie pesquisada.

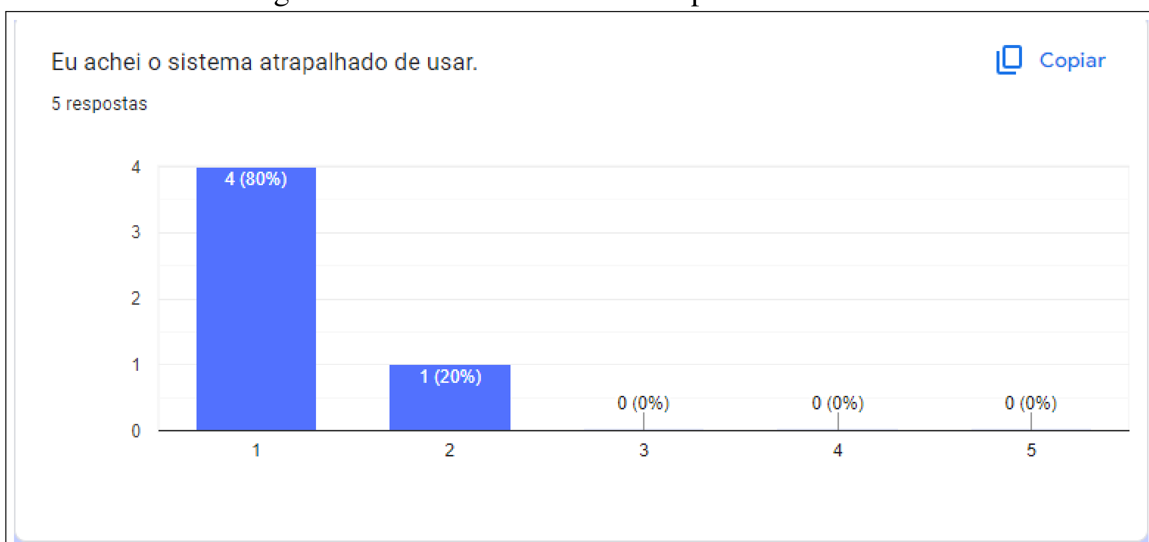
Pode-se perceber que a maioria dos participantes achou o sistema simples e fácil de utilizar, aplicando a fórmula do questionário SUS para chegar a pontuação final, chegamos a pontuação média de 89, com desvio padrão de 8.74 para os 5 participantes. (TEIXEIRA, 2015)

Figura 7.7: Eu imagino que as pessoas aprenderão como usar esse sistema rapidamente.



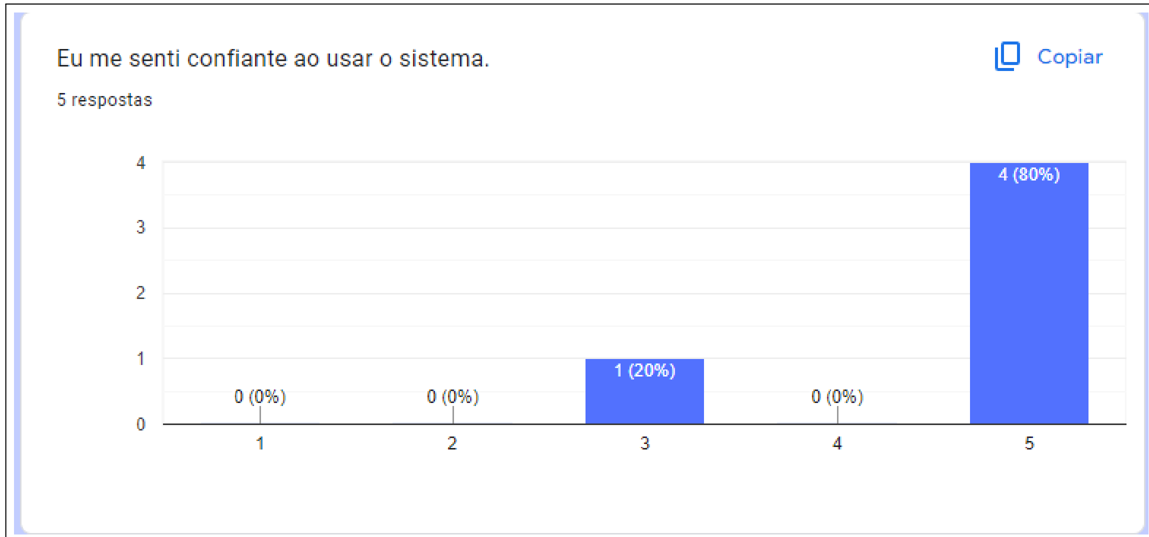
Fonte: do autor.

Figura 7.8: Eu achei o sistema atrapalhado de usar.



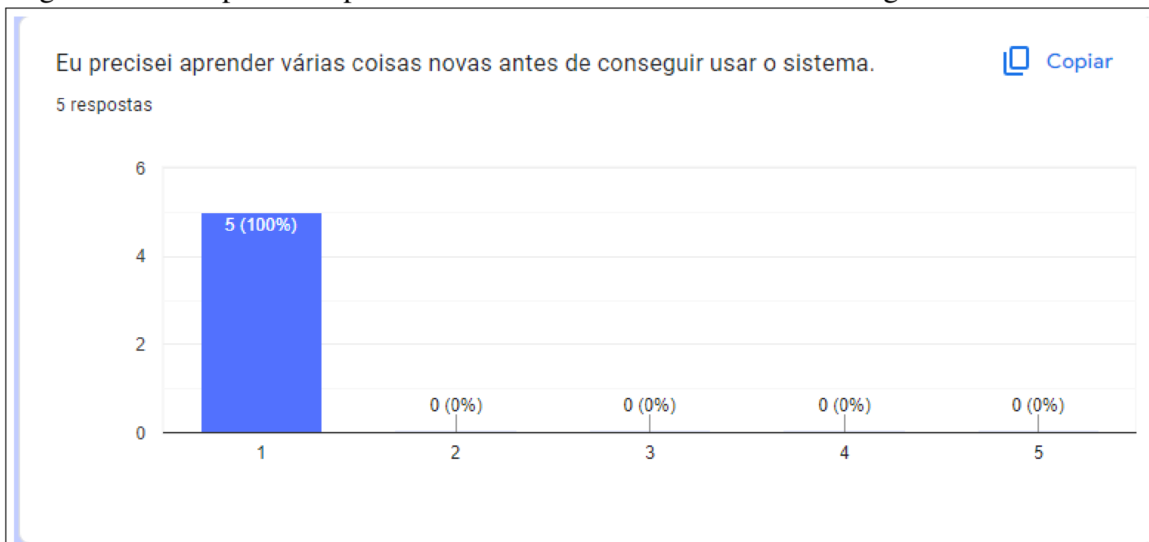
Fonte: do autor.

Figura 7.9: Eu me senti confiante ao usar o sistema.



Fonte: do autor.

Figura 7.10: Eu precisei aprender várias coisas novas antes de conseguir usar o sistema.



Fonte: do autor.

8 CONCLUSÕES

Através deste trabalho, foi possível desenvolver um software para catalogar e exibir virtualmente os itens do Museu do Mar utilizando React para o *frontend* e .NET 8 para o *backend*, além disso, o *backend* também serve como uma API para futuros projetos que possam consumir esses dados. Foram aplicada habilidades de pesquisa, levantamento de requisitos ,solução de problemas,engenharia de software e APIs RESTful para concluir este trabalho.

Nem todas as metas definidas no início foram alcançadas. A página de registro de boletim de condições do mar e a filtragem de registros não foram implementadas. Além disso, o planejamento da ordem de entregas não foram seguidos por razões técnicas, foi necessário voltar para uma funcionalidade já implementada para arrumar erros. Muitas das dificuldades técnicas foram superadas, como a implementação de autenticação e gerenciamento de imagens, mas não foi possível concluir a implementação do sistema em um ambiente *Cloud* ou utilizando Docker por causa da elevada curva de aprendizado, sendo necessário, além de configurar os containeres Docker, também adaptar as aplicações para melhor utilizar variáveis de ambiente.

Foi possível notar por meio da pesquisa com os usuários finais que a maioria deles achou o sistema simples de usar e fácil de entender. Isso destaca a eficiência da implementação dos métodos utilizados e a eficácia da aplicação desenvolvida. Porém, foram percebidos alguns problemas, como lentidão ao carregar o sistema, pelo fato do sistema tentar carregar sempre todas as imagens dos registros na tela, e também a falta de feedback quando algo está carregando.

Embora o software desenvolvido tenha cumprido parte do objetivo proposto, há sempre espaço para melhorias. Uma possível sugestão seria implementar o Registro de Boletim do Mar e utilizar *roles* para diferenciar pessoas administrativas de pescadores que queiram registrar o boletim do mar. Também seria útil implementar a aplicação em um *container* Docker, permitindo isolar o banco de dados de invasores externos.

Outra melhoria que iria elevar a experiência do usuário seria a criação de uma identidade visual para a página, junto com a identificação de melhores elementos de design de interface para serem aplicados.

Portanto, este trabalho serviu como uma excelente oportunidade para aplicar conceitos de Engenharia de Software e consolidar diversos conhecimentos importantes para o mercado de trabalho. O resultado final foi um sistema funcional que pode ser usado pelo

Museu do Mar para catalogar e exibir seus itens virtualmente, bastando apenas utilizar a aplicação em uma máquina virtual Linux ou Windows.

REFERÊNCIAS

AUTOMAPPER. **AutoMapper**. 2024. Available from Internet: <https://automapper.org/>.

AUTOMAPPER. **AutoMapper NuGet Package**. 2024. Available from Internet: <https://www.nuget.org/packages/automapper/>.

AZUREAD, M. **System.IdentityModel.Tokens.Jwt NuGet Package**. 2024. Available from Internet: <https://www.nuget.org/packages/System.IdentityModel.Tokens.Jwt/7.2.0>.

BARRY, D. K. **Representational State Transfer (REST)**. 2024. Available from Internet: <https://www.service-architecture.com/articles/web-services/representational-state-transfer-rest.html>.

CATALOGIT. **CatalogIt**. 2024. Available from Internet: <https://www.catalogit.app/>.

CECHIN, B. S. d. S. A.

Tu em Torres: auxiliando a aprendizagem sobre a fauna e flora de Torres para alunos do ensino fundamental — Universidade Federal do Rio Grande do Sul, 2019.

CHRISMCKEE. **BCrypt.Net-Next NuGet Package**. 2024. Available from Internet: <https://www.nuget.org/packages/BCrypt.Net-Next>.

FACEBOOK JOSH STORY, S. A. **React NPM Package**. 2024. Available from Internet: <https://www.npmjs.com/package/react>.

FOUNDATION, M. **MDN Web Docs - Blob**. 2024. Available from Internet: <https://developer.mozilla.org/en-US/docs/Web/API/Blob>.

FOUNDATION, M. **MDN Web Docs - Window.localStorage**. 2024. Available from Internet: <https://developer.mozilla.org/pt-BR/docs/Web/API/Window/localStorage>.

GROUP, A. D. C. A. **Ant Design Documentation**. 2024. Available from Internet: <https://ant.design/docs/react/introduce>.

GROUP, A. D. C. A. **Ant Design NPM Package**. 2024. Available from Internet: <https://www.npmjs.com/package/antd>.

Internet Engineering Task Force (IETF). **JSON Web Token (JWT)**. [S.l.], 2015. Available from Internet: <https://datatracker.ietf.org/doc/html/rfc7519>.

MAHAN, J. **OData Query Builder NPM Package**. 2024. Available from Internet: <https://www.npmjs.com/package/odata-query-builder>.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. [S.l.]: Prentice Hall, 2008. ISBN 9780132350884.

MICROSOFT. **ASP.NET Core Fundamentals - Dependency Injection**. 2024. Available from Internet: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0>.

MICROSOFT. **Entity Framework Core 8.0 Documentation**. 2024. Available from Internet: <https://learn.microsoft.com/pt-br/ef/core/what-is-new/ef-core-8.0/whatsnew>.

MICROSOFT. **Internet Information Services (IIS) Official Website**. 2024. Available from Internet: <https://www.iis.net/>.

MICROSOFT. **Introduction to LINQ Queries**. 2024. Available from Internet: <https://learn.microsoft.com/pt-br/dotnet/csharp/linq/get-started/introduction-to-linq-queries>.

MICROSOFT. **Microservices architecture, Domain-Driven Design (DDD), and Command Query Responsibility Segregation (CQRS) Patterns - Infrastructure and Persistence Layer Design**. 2024. Available from Internet: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design>.

MICROSOFT. **Microsoft SQL Server Downloads**. 2024. Available from Internet: <https://www.microsoft.com/pt-br/sql-server/sql-server-downloads>.

MICROSOFT. **.NET**. 2024. Available from Internet: <https://dotnet.microsoft.com/pt-br/>.

MICROSOFT. **.NET 8**. 2024. Available from Internet: <https://learn.microsoft.com/pt-br/dotnet/core/whats-new/dotnet-8/overview>.

MICROSOFT. **TypeScript NPM Package**. 2024. Available from Internet: <https://www.npmjs.com/package/typescript>.

MICROSOFT. **TypeScript Official Website**. 2024. Available from Internet: <https://www.typescriptlang.org/>.

MICROSOFT. **System.Linq NuGet Package**. 2026. Available from Internet: <https://www.nuget.org/packages/System.Linq/>.

MICROSOFT ASPNET, d. **Microsoft.AspNetCore.Authentication.JwtBearer NuGet Package**. 2024. Available from Internet: <https://www.nuget.org/packages/Microsoft.AspNetCore.Authentication.JwtBearer/>.

MICROSOFT ASPNET, d. E. **Microsoft.EntityFrameworkCore NuGet Package**. 2024. Available from Internet: <https://www.nuget.org/packages/Microsoft.EntityFrameworkCore/>.

MICROSOFT ODATA, d. **Microsoft.AspNetCore.OData NuGet Package**. 2024. Available from Internet: <https://www.nuget.org/packages/Microsoft.AspNetCore.OData/>.

NGROK, I. **ngrok Official Website**. 2024. Available from Internet: <https://ngrok.com/>.

ODATA. **OData**. 2024. Available from Internet: <https://www.odata.org/>.

PROVOS, N.; MAZIèRES, D. A future-adaptable password scheme. In: THE OPENBSD PROJECT. **Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference**. Monterey, California, USA, 1999.

SANTOS, R. O. S.

Implementação de um módulo de consultas geográficas para o BorbRS®2 — Universidade Federal do Rio Grande do Sul, 2011.

SARJEANT, J. J. J. **Axios Official Website**. 2024. Available from Internet: <https://axios-http.com/>.

SCHöLLER, L. M. **react-tree-store**. 2021. Available from Internet: <https://github.com/leomollmann/react-tree-store>.

SILVA, M. C. d.

Proposta de um sistema para o registro de informações de um Pescador para apoio ao Museu do Mar Virtual — Universidade Federal do Rio Grande do Sul, 2021.

SOURCE, M. O. **React Official Website**. 2024. Available from Internet: <https://react.dev/>.

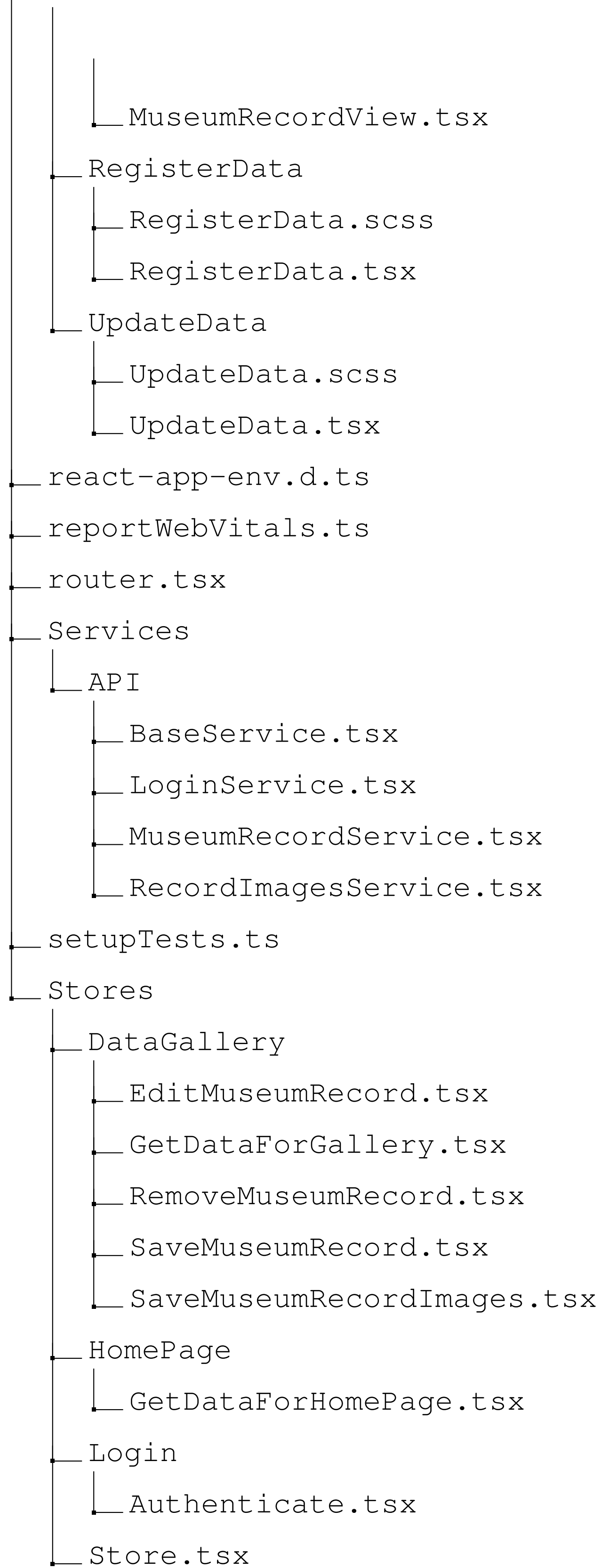
TEIXEIRA, F. **O que é o SUS (System Usability Scale) e como usá-lo em seu site**. 2015. Available from Internet: <https://brasil.uxdesign.cc/o-que-%C3%A9-o-sus-system-usability-scale-e-como-us%C3%A1-lo-em-seu-site-6d63224481c8>.

ZABRISKIE, M.; CONTRIBUTORS. **Axios NPM Package**. 2024. Available from Internet: <https://www.npmjs.com/package/axios>.

9 APÊNDICES

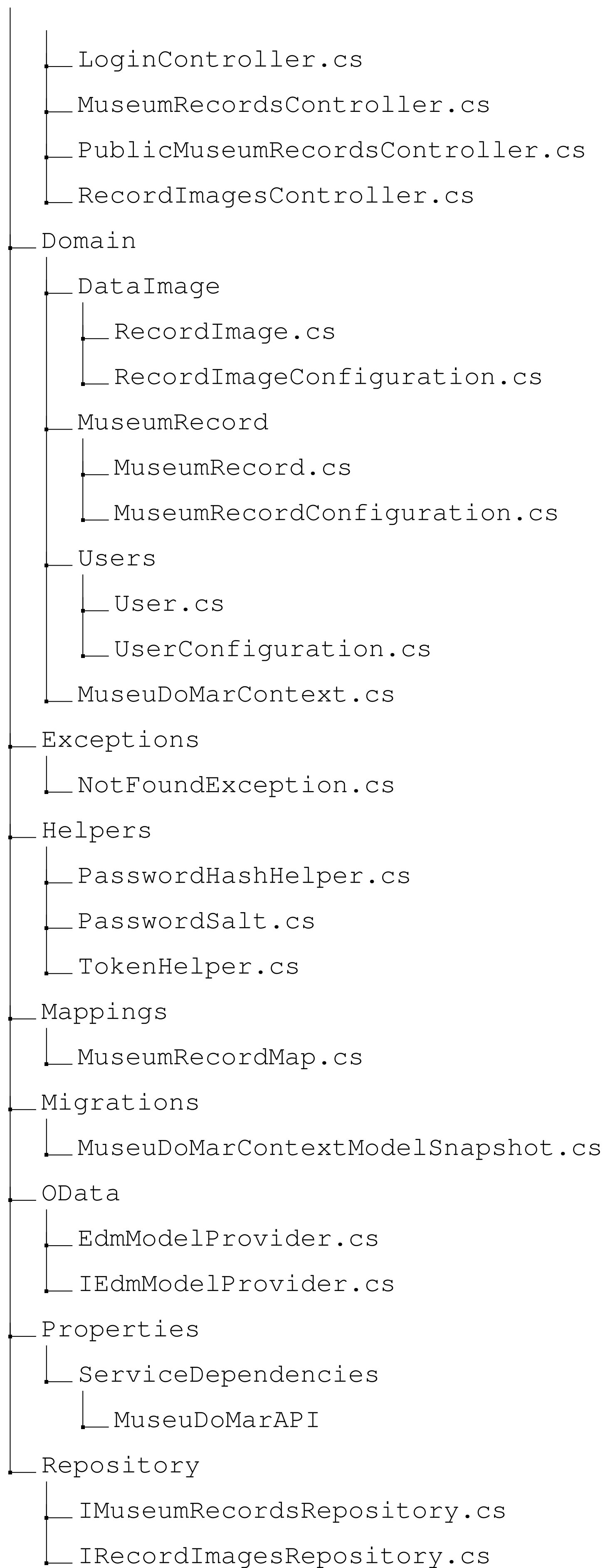
9.1 Apêndice A

```
src
├── App.css
├── App.test.tsx
├── App.tsx
├── Entities
│   ├── MuseumRecord.tsx
│   └── RecordImage.tsx
├── Helpers
│   ├── ImageBlobHelper.tsx
│   └── LocalStorageHelper.tsx
├── index.css
├── index.tsx
├── logo.svg
└── Pages
    ├── DataGallery
    │   ├── DataGallery.scss
    │   ├── DataGallery.setup.tsx
    │   └── DataGallery.tsx
    ├── Header
    │   ├── Header.scss
    │   └── Header.tsx
    ├── HomePage
    │   ├── HomePage.scss
    │   ├── HomePage.setup.tsx
    │   └── HomePage.tsx
    ├── Login
    │   ├── Login.scss
    │   └── Login.tsx
    └── MuseumRecordView
        └── MuseumRecordView.scss
```



9.2 Apêndice B

```
MuseuDoMar
└─Controllers
```


LoginController.cs
MuseumRecordsController.cs
PublicMuseumRecordsController.cs
RecordImagesController.cs
Domain
DataImage
RecordImage.cs
RecordImageConfiguration.cs
MuseumRecord
MuseumRecord.cs
MuseumRecordConfiguration.cs
Users
User.cs
UserConfiguration.cs
MuseuDoMarContext.cs
Exceptions
NotFoundException.cs
Helpers
PasswordHashHelper.cs
PasswordSalt.cs
TokenHelper.cs
Mappings
MuseumRecordMap.cs
Migrations
MuseuDoMarContextModelSnapshot.cs
OData
EdmModelProvider.cs
IEdmModelProvider.cs
Properties
ServiceDependencies
MuseuDoMarAPI
Repository
IMuseumRecordsRepository.cs
IRecordImagesRepository.cs

```
│
│  │
│  ├── IRepository.cs
│  ├── IUnitOfWork.cs
│  ├── IUsersRepository.cs
│  ├── MuseumRecordsRepository.cs
│  ├── RecordImagesRepository.cs
│  ├── Repository.cs
│  ├── UnitOfWork.cs
│  └── UsersRepository.cs
├── Services
│  ├── ILoginService.cs
│  ├── IMuseumRecordsService.cs
│  ├── IRecordImagesService.cs
│  ├── LoginService.cs
│  ├── MuseumRecordsService.cs
│  └── RecordImagesService.cs
├── Settings
│  ├── ImageSettings.cs
│  └── JwtOptions.cs
├── ViewModels
│  ├── LoginViewModel.cs
│  ├── PostLoginViewModel.cs
│  ├── PublicDataImageViewModel.cs
│  ├── PublicMuseumRecordViewModel.cs
│  ├── RecordImageViewModel.cs
│  └── SaveImageViewModel.cs
├── appsettings.Development.json
├── appsettings.json
├── Dockerfile
├── MuseuDoMar.csproj
├── MuseuDoMar.csproj.user
├── MuseuDoMar.http
└── Program.cs
```