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

MATEUS CARDOSO DA SILVA

Proposta de um sistema para o registro de informações de um Pescador para apoio ao Museu do Mar Virtual

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

Orientador: Prof. Dr. Leandro Krug Wives
Coorientador: Ms. Igor Khun

Porto Alegre
2021

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

RESUMO

Este trabalho descreve a implementação de uma aplicação web que tem como objetivo prover os meios tecnológicos para que um líder comunitário e pescador do litoral gaúcho consiga ter um maior alcance em sua contribuição social, levando o seu conhecimento para estudantes do ensino básico, para que se conscientizem sobre o meio ambiente através de uma educação ambiental focada na costa marítima, sabendo, por exemplo, quais animais podem ser encontrados, como funciona a prática da pesca, além de aprender sobre sua preservação. Dentre as funcionalidades desenvolvidas há a criação e edição de postagens, incluindo *upload* de imagens, informações sobre o clima, pescaria, lixo encontrado, entre outros. A ferramenta também permite realizar pesquisa pelos *posts* publicados, seja pelo título ou por *tags*, fazendo a ordenação dos resultados por colunas, e oferecendo a visualização dessas informações em uma aplicação responsiva, que se adapta ao dispositivo do usuário. A aplicação foi desenvolvida através de tecnologias-padrão da indústria, com o uso de React e JavaScript para os componentes de interface, Java e Spring para a implementação de uma REST API, postgreSQL como o banco de dados relacional, NGINX para a definição do *web server* e para realizar *proxy* reverso, e, para cada um dos componentes, um *container* específico foi definido através de Docker-Compose. Também foi utilizado o GitHub como ferramenta para controle das *sprints*, *backlogs* e das mudanças feitas no código.

Palavras-chave: Aplicação Web. Responsividade. React. Java Spring.

Proposal of a system for recording information from a Fisher to support the Virtual Sea Museum

ABSTRACT

This work describes the implementation of a web application that aims to provide technological means to a community leader and fisherman, letting him achieve larger social contribution and bring his knowledge to students from elementary schools. The point is to provide environmental education to those kids, amplifying their awareness about the environment, i.e., the marine coast, letting students know what kind of animals can be found, how fishery practices are and learn about its preservation. Among the developed functionalities are the creation and editing of posts containing images and information about climate, fishing, garbage found, etc. Also, it allows searching posts by publication date, title, or tags, ordering the results per column, and visualization of this information in an adaptive, responsive application. The application was developed using industry-standard technologies like React and JavaScript for the interface components, Java and Spring for a REST API, PostgreSQL for a relational database, NGINX for webserver definition, and reverse proxy. In addition, for each component, a specific container was defined using Docker-Compose. Finally, GitHub was also used to provide source code, backlog, and sprint control.

Keywords: Web Application, Responsivity, React, Java Spring.

LISTA DE FIGURAS

Figura 2.1 Visão alto nível das tecnologias empregadas na aplicação 13

Figura 3.1 Diagrama Entidade Relacionamento das tabelas de persistência do banco de dados 18
Figura 3.2 Trecho de código da entidade *PostRecord* 20
Figura 3.3 Repositório da entidade *PostRecord* 20
Figura 3.4 Objetos e pastas da aplicação de BE 21
Figura 3.5 Testes da aplicação de BE 22
Figura 3.6 Arquivo *index.js* da aplicação de FE 23
Figura 3.7 Arquivo *pageRouter.js* da aplicação de FE 23
Figura 3.8 Pasta com os componentes da tela de Criação e Edição de *Posts* 24
Figura 3.9 Pasta com os componentes da tela de Busca de *Posts* 25
Figura 3.10 Pasta com os componentes da tela de Visualização de *Posts* 25
Figura 3.11 Exemplo de chamada para requisição usando Fetch 26
Figura 3.12 Diagrama dos containers Docker 27
Figura 3.13 Trecho da classe *PostRecordController* responsável pela ordenação 30
Figura 3.14 Classe responsável pela criação das especificações 31

Figura 4.1 Tela *Home* da aplicação de FE 33
Figura 4.2 Tela *Create* da aplicação de FE 34
Figura 4.3 Tela *Search* da aplicação de FE 36
Figura 4.4 Tela *Display* da aplicação de FE 36

Figura 5.1 Respostas às perguntas 1 a 3 38
Figura 5.2 Respostas às perguntas 4 a 6 39
Figura 5.3 Respostas às perguntas 7 a 10 40
Figura 5.4 Notas calculadas para cada testador e média 41

LISTA DE ABREVIATURAS E SIGLAS

UFRGS	Universidade Federal do Rio Grande do Sul
MVP	Minimum Viable Product
SUS	System Usability Scale
CRUD	Create, Read, Update, Delete
MVC	Model View Controller
REST	Representational State Transfer
API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
BE	Backend
FE	Frontend
POM	Project Object Model
CSS	Cascading Style Sheets

SUMÁRIO

1 INTRODUÇÃO ... 9
2 CONCEITOS E TECNOLOGIAS RELACIONADOS ... 11
2.1 Desenvolvimento Ágil ... 11
2.1.1 Histórias de Usuário ... 12
2.1.2 Backlog ... 12
2.2 Tecnologias Aplicadas ... 12
2.2.1 Desenvolvimento Frontend ... 13
2.2.1.1 JavaScript ... 13
2.2.1.2 NPM ... 13
2.2.1.3 React ... 14
2.2.1.4 Semantic UI React ... 14
2.2.1.5 Outros módulos importados ... 14
2.2.2 Desenvolvimento Backend ... 14
2.2.2.1 Java ... 14
2.2.2.2 Spring ... 14
2.2.2.3 Maven ... 15
2.2.2.4 PostgreSQL ... 15
2.2.3 Infraestrutura da aplicação ... 15
2.2.3.1 Docker-Compose ... 15
2.2.3.2 NGINX ... 15
3 MODELAGEM E PROJETO ... 16
3.1 Definição de Requisitos ... 16
3.1.1 Backlog ... 17
3.2 Arquitetura ... 17
3.2.1 Backend ... 18
3.2.1.1 Modelagem de Dados ... 18
3.2.1.2 Classes e Estrutura de pastas ... 19
3.2.2 Frontend ... 22
3.2.2.1 Telas ... 22
3.2.2.2 Funções e Estrutura de pastas ... 22
3.2.2.3 Integração com Backend ... 26
3.2.3 Infraestrutura ... 26
3.2.3.1 Exposição para Internet ... 26
3.2.3.2 Definição dos containers ... 27
3.3 Funcionalidades Detalhadas ... 28
3.3.1 Criação de Posts ... 28
3.3.1.1 Upload de Imagens ... 28
3.3.1.2 Tags de Busca ... 29
3.3.1.3 Adicionar links relacionados ... 29
3.3.2 Busca de Posts ... 29
3.3.2.1 Colunas Adicionais ... 30
3.3.2.2 Ordenação por Coluna ... 30
3.3.2.3 Filtros de Busca ... 31
3.3.3 Visualização de Posts ... 32
4 GUIA DE USO ... 33
4.1 Pescador ... 33
4.2 Curador ... 35
4.3 Estudante ... 35

5 AVALIAÇÃO/VALIDAÇÃO....37
5.1 Resultados SUS....37
5.2 Feedbacks....41
6 CONCLUSÕES....44
REFERÊNCIAS....46
APÊNDICE A — QUESTIONÁRIO DE USABILIDADE....48
APÊNDICE B — DOCUMENTAÇÃO DA API....53
APÊNDICE C — BACKLOG: HISTÓRIAS DE USUÁRIO E ITENS DE BACKLOG....58

1 INTRODUÇÃO

Em um contexto em que vivemos com cada vez mais desafios para o acesso ao conhecimento, à educação de forma acessível e inclusiva, em um contexto de mudanças climáticas, de maus tratos ao meio ambiente, no Litoral Norte Gaúcho surge o Museu do Mar. Ele consiste em uma iniciativa que está montando um repositório do conhecimento e dos achados que um pescador e líder comunitário coletou durante sua vida, para que seja possível alcançar estudantes de forma remota, seja pela questão do isolamento social durante a pandemia, ou para aqueles que estão afastados do litoral. Essa iniciativa procura ser uma ponte entre a vontade de ensinar e a necessidade de se ter conteúdo que impulsione o interesse sobre o meio ambiente e sua preservação, principalmente para aqueles que estão no início da sua formação.

Hoje, este pescador já faz um trabalho de gerar conteúdo referente a condição das praias onde trabalha, coletando também “artefatos” como carcaças de animais, lixo oriundo de diversos países, e registrando tudo isso usando *posts* no Facebook, em um grupo de pescadores. No entanto, a partir do momento em que uma postagem é feita, é muito difícil que aquela informação seja encontrada novamente por um professor ou estudante que tenha interesse.

Esta monografia busca minimizar esse problema, e tem por objetivo relatar o desenvolvimento de um Mínimo Produto Viável (MVP, sigla em inglês), onde seja possível fazer a captura das informações necessárias e disponibilizar uma forma de pesquisa e visualização.

Com isso, evita-se que esse conteúdo gerado diariamente seja perdido, já que, enquanto esse conteúdo é disponibilizado somente a partir da rede social, é muito difícil de ser encontrado novamente, em meio a outras postagens que acontecem no grupo por exemplo, e nem existe a possibilidade de se encontrar algo baseado em alguma informação específica que foi colocada. Com este trabalho, a informação passa a ser agregada e disponibilizada de uma forma estruturada, sendo possível que ela seja incrementada com o tempo, que possa ser usada para alimentar pesquisas futuras, ser um recurso para professores usarem em sala de aula.

Enquanto que a mídia social tem um alcance imediato e é absorvido no instante em que é postado, a nova aplicação serve para arquivar essas postagens diárias e estruturar a informação de forma que, apesar de cada relato ser um pedaço pequeno do conhecimento, que com o tempo a quantidade de informação coletada seja possível extrair novo

conhecimento, o que seria atrativo para professores e estudantes.

O desenvolvimento foi realizado usando a práticas de desenvolvimento ágil usando alguns elementos de Scrum, como definições de histórias de usuário e *sprints* de desenvolvimento, com durações variando entre 2 ou 3 semanas, ao final de cada *sprint* ocorria uma reunião com o grupo para revisar o resultado da *sprint* e receber *feedbacks*. Terminada a implementação do MVP, foram realizados testes e coletas de *feedback* para verificar a usabilidade do sistema usando a *System Usability Scale*, ou SUS – sigla em Inglês para "Escala de Usabilidade do Sistema").

Este documento detalha as definições dos requerimentos através de histórias de usuário, as tecnologias envolvidas para o controle das atividades e para o desenvolvimento da aplicação, quais os conceitos de arquitetura, detalhes de implementação das funcionalidades e os resultados obtidos pelos testes e do questionário de usabilidade.

2 CONCEITOS E TECNOLOGIAS RELACIONADOS

Este trabalho foi realizado como um projeto de desenvolvimento de software, onde a partir de uma necessidade de um “cliente” foram determinados quais os requisitos funcionais e não-funcionais para a aplicação. Também foram definidas as tecnologias para a programação e os processos pelos quais seria feito o acompanhamento da implementação. Este capítulo detalha os conceitos utilizados durante o projeto, as ferramentas e as tecnologias.

2.1 Desenvolvimento Ágil

Metodologias ágeis de desenvolvimento, como por exemplo *Scrum*, servem para estruturar o desenvolvimento de um software ou produto de forma que a necessidade de mudanças ou de falhas que possam ocorrer durante o processo são esperadas, e por isso o desenvolvimento ocorre iterativamente, com melhorias e novas funcionalidades sendo criadas incrementalmente, onde o design e arquitetura da aplicação são maleáveis e em que um time de desenvolvimento tem as ferramentas para se comunicar e planejar os próximos passos (MERGEL, 2016).

Cada uma das iterações do software é chamada de *Sprint*, e tem a função de delimitar o escopo das funcionalidades que são implementadas em um espaço de tempo, em que o desenvolvimento é direcionado a um objetivo específico. O escopo de todas as funcionalidades de uma aplicação é chamado de *Backlog* de Produto e é onde são descritos os requisitos funcionais e não funcionais, o *Backlog* é composto por Itens de *Backlog* que descrevem as funcionalidades a serem desenvolvidas para suprir as necessidades definidas como requisito (SCHWABER; JEFF, 2020).

A cada *Sprint* existem cerimônias que ocorrem em toda sua duração: para demarcar o início de uma *Sprint*, o Planejamento da *Sprint*, onde os *backlog*s são estimados e aceitos ou não pelo time, as reuniões diárias, *Daily* em inglês, onde o time pode levantar a situação do desenvolvimento, sobre a necessidade de mudanças ou bloqueios, ao final a Revisão da *Sprint*, onde o que foi desenvolvido é apresentado, e a Retrospectiva, onde podem ser discutidos os processos internos do time (SCHWABER; JEFF, 2020).

2.1.1 Histórias de Usuário

O escopo do desenvolvimento do aplicativo foi definido utilizando o recurso de histórias de usuário, onde que a partir dos relatos daqueles que têm a necessidade do software, são extraídos os componentes essenciais para a realização das tarefas que a nova implementação precisa prover (REHKOPF, 2020).

Histórias de Usuário comumente são escritas no seguinte formato:

- "Como um..." para identificar quem é o usuário que faz uma ação
- "Eu quero..." para descrever qual a ação que o usuário precisa fazer
- "De modo que..." para descrever com que objetivo o usuário realiza a ação

2.1.2 Backlog

O *Backlog* é o detalhamento e a quebra dos requerimentos em tarefas de desenvolvimento, cada um dos Itens de *Backlog* são funcionalidades incrementais para a aplicação que podem ser realizadas durante uma *Sprint*, e que ao final dela podem ser demonstrados usando o Critério de Aceite durante a Revisão da *Sprint*. Em um time de desenvolvimento os Itens de *Backlog* são refinados para que tenham o detalhamento necessário para o entendimento das tarefas, e durante o Planejamento da *Sprint* os itens são estimados em pontos, ou em horas de desenvolvimento, para assim determinar qual será o escopo daquela *Sprint* (SCHWABER; JEFF, 2020).

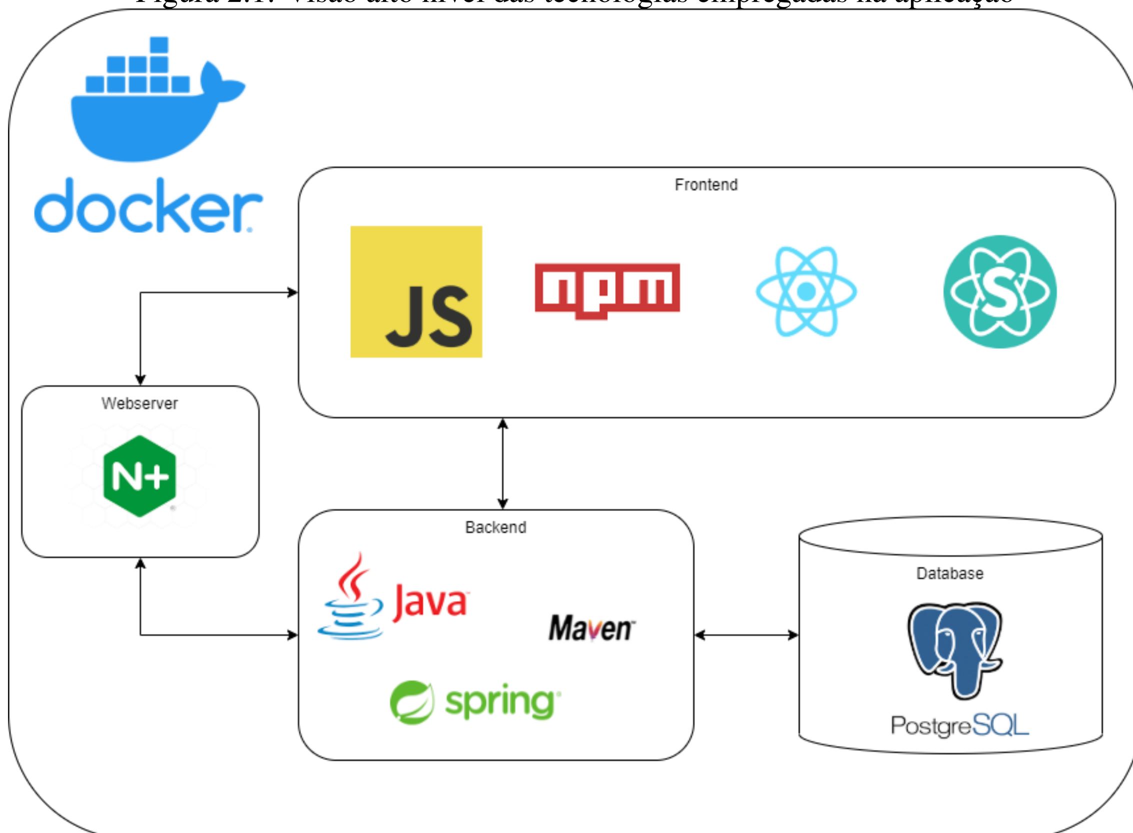
O controle do *backlog* e das *Sprints* de desenvolvimento foi feito usando do GitHub [1].

2.2 Tecnologias Aplicadas

Para o desenvolvimento da aplicação foram escolhidas tecnologias que são reconhecidamente usadas pela indústria em aplicações web, e também para cumprir com o objetivo ser fácil de implantar em um servidor sem a necessidade de configurar todas as dependências necessárias, uma característica de aplicações *Cloud-Native* orientadas a microsserviços. A figura 2.1 mostra como as tecnologias foram usadas em cada componente.

[1] <https://github.com/MateusCardoso/mar-de-informacao>

Figura 2.1: Visão alto nível das tecnologias empregadas na aplicação



2.2.1 Desenvolvimento Frontend

2.2.1.1 JavaScript

Linguagem de programação utilizada para a implementação do código que roda no navegador do usuário, também conhecido como *client-side*, e que possibilita as interações do usuário com a aplicação.

2.2.1.2 NPM

Ferramenta de empacotamento e de importação de pacotes, usado para fazer a instalação das dependências do código no servidor de *frontend* de forma automatizada a partir do arquivo *package.json* (NPM, 2009).

2.2.1.3 React

É uma biblioteca JavaScript para a criação de interfaces responsivas, sendo responsável pela renderização dos componentes da tela, pelos controles de eventos e estados da aplicação (FACEBOOK, 2013).

2.2.1.4 Semantic UI React

Uma integração com React de conceitos de Semantic UI, que disponibiliza componentes adicionais para a interface já estilizados e com configurações internas prontas para uso (SEMANTIC, 2015).

2.2.1.5 Outros módulos importados

- dotenv: possibilita definição de variáveis de ambiente em um arquivo .env (MOTTE, 2013).
- moment: *parsing* e manipulação de datas (MOMENT, 2011).
- react-router: controle de rotas da aplicação, faz a navegação entre as páginas do *app* (REMIX, 2015).
- create-react-app: fornece *scripts* para execução da aplicação React (FACEBOOK, 2016).
- react-sematinc-ui-datepickers: disponibiliza um componente no estilo de Semantic-UI para seleção de datas (DENNER, 2018).

2.2.2 Desenvolvimento Backend

2.2.2.1 Java

Linguagem de programação orientada a objetos utilizada para implementação do código de *backend*, responsável pelo tratamento das requisições HTTP e pela integração com o banco de dados.

2.2.2.2 Spring

Framework de desenvolvimento de serviços em Java, adiciona diversas anotações, classes e interfaces que podem ser utilizadas para a implementação do modelo para o

banco de dados, para o mapeamento dos métodos HTTP, como *GET*, *PATCH*, *POST*, e quais as variáveis disponíveis para estas chamadas, definições de repositórios para o acesso ao banco e suporte a definições das *queries SQL* (SPRING, 2018).

2.2.2.3 Maven

Ferramenta para definição e automação de construção da aplicação baseado em definições feitas em um arquivo POM (Objeto Modelo de Projeto em Inglês) (APACHE, 2017).

2.2.2.4 PostgreSQL

Banco de dados relacional de código aberto (POSTGRESQL, 2015).

2.2.3 Infraestrutura da aplicação

2.2.3.1 Docker-Compose

Ferramenta para definir e executar aplicações com múltiplos *containers* de execução do código. É possível separar cada camada da aplicação em um serviço individual e independente que pode ter suas próprias definições para processos de compilação, construção, implantação, com a definição de redes virtuais para a comunicação entre *containers* específicos, mapeamentos de portas e volumes na máquina *host* (DOCKER, 2013).

2.2.3.2 NGINX

Web Server utilizado para fazer balanceamento de carga e *proxy* reverso para aplicações com múltiplos serviços ou páginas (F5, 2015).

3 MODELAGEM E PROJETO

O projeto de implementação deste trabalho foi realizado tendo como princípio de que usaria boas práticas de engenharia de software, de metodologias ágeis de desenvolvimento, utilizando de tecnologias e ferramentas que representam o que é feito na indústria atualmente. São práticas que auxiliaram na entrega dos requisitos, pois possibilitam um melhor entendimento de quais eram os requerimentos, como eles poderiam ser desenvolvidos e que ajudam a deixar registrados os esforços necessários, possibilitam ter um Software que atende as necessidades atuais e que pode ser mais facilmente incrementado no futuro.

Este capítulo descreve como foram aplicados os conceitos de desenvolvimento ágil e de engenharia de software, assim como as tecnologias, ferramentas e como foi arquiteturada a solução.

3.1 Definição de Requisitos

Neste trabalho foi utilizado o conceito de *Backlog* de Produto, com as funcionalidades necessárias dividas em Histórias de Usuário e Itens de *Backlog* com o detalhamento das tarefas de desenvolvimento, cada *Sprint* teve duração de 2 ou 3 semanas e Revisões de *Sprint* foram realizadas com o grupo do Museu do Mar e com o professor orientador para a coleta de *feedbacks* e de novos requisitos, o Planejamento da *Sprint* também foi simplificado e se resumiu a apresentar para o grupo quais as novas tarefas que seriam desenvolvidas na *Sprint* seguinte, não foram utilizadas as cerimônias de *Daily* ou de Retrospectiva já que não havia um time de desenvolvimento.

As histórias de usuário levantadas foram:

- US-01 Como um “Pescador” quero poder compartilhar informações úteis de meu dia-dia para auxiliar com o aprendizado da comunidade estudantil/acadêmica
- US-02 Como um “Curador” quero poder incrementar as informações já postadas, e que sirvam de referência acadêmica para o conhecimento empírico adicionado pelo “Pescador”
- US-03 Como um “Estudante/Professor”, quero poder pesquisar por *Posts* publicados que se refiram a assuntos específicos e ver em detalhes
- US-04 Como “Administrador”, quero poder controlar quem pode criar novos *Posts*

na ferramenta

As histórias de usuário US-01, US-02 e US-03 definem os requisitos funcionais na aplicação, onde cada usuário identificado tem a intenção de usufruir da aplicação, enquanto que US-04 foi criada devido a necessidade de haver um ator que realize tarefas técnicas relacionadas com requisitos não-funcionais.

3.1.1 Backlog

Para o controle do progresso das histórias de usuário foi criado um board[1] dentro do próprio repositório do GitHub, onde foram incluídas como *cards* cada uma das histórias de usuário com um detalhamento das tarefas de desenvolvimento necessárias para cumprir o requisito. Cada tarefa também foi adicionada como um *card* para dar a visibilidade do status, e são relacionadas com as *Pull Requests* que as implementam, assim se tem também o controle de qual código implementou qual funcionalidade.

Esse Backlog foi levantando através de conversas com o Pescador e com os membros do grupo do Museu do Mar, obtendo-se os requisitos funcionais, as tarefas que seriam necessárias para desenvolver a aplicação foram descritas então usando os itens de backlog.

3.2 Arquitetura

De forma geral a arquitetura da aplicação é feita conforme o modelo MVC, *Model View Controller*, onde a *Model* é a representação dos dados no banco de dados e suas definições via classes, a *Controller* o conjunto de classes que fazem o CRUD das informações, ambos implementados no backend da aplicação, já a *View* é representada pelos componentes de frontend que possibilitam a visualização e edição dos valores na tela.

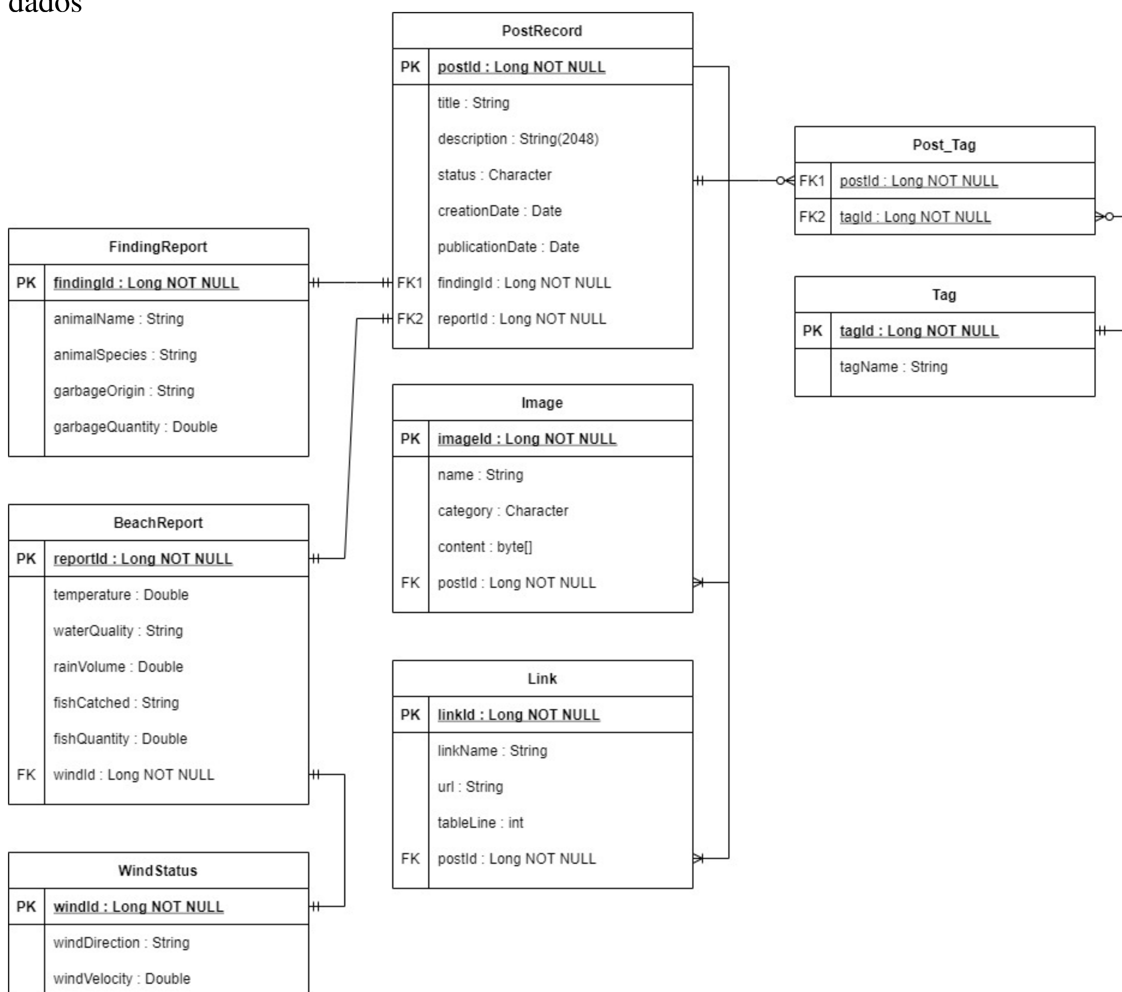
[1] <https://github.com/MateusCardoso/mar-de-informacao/projects/1>

3.2.1 Backend

3.2.1.1 Modelagem de Dados

As informações necessárias de serem armazenadas em banco de dados foram baseadas naquilo que o pescador atualmente faz em suas postagens via Facebook, e também a partir dos inputs do grupo do Museu do Mar naquilo que seria importante didaticamente, ou que poderiam auxiliar a fazer buscas mais direcionadas.

Figura 3.1: Diagrama Entidade Relacionamento das tabelas de persistência do banco de dados



O modelo de dados foi elaborado de forma que se dividiu as informações conforme os conceitos a que aquelas informações se aplicam. Na figura 3.1 podemos ver como a entidade *PostRecord* representa uma entidade central, representando o *Post* que é entrado na aplicação, como suas propriedades, os campos mais comuns de serem preenchidos, título e descrição, assim como informações para controle de status e de datas relevantes à

criação e a publicação.

As entidades *FindingReport*, *BeachReport* e, por transitividade, *WindStatus* possuem uma relação de um para um com a entidade *PostRecord*, essa definição se dá pelo fato de as informações de cada uma destas entidades serem preenchidas usualmente juntas e para que seja mais evidente aquilo que elas definem. *Image* e *Link*, são entidades com uma relação um para muitos com *PostRecord* sem a obrigatoriedade da existência destes, já que é de escolha do usuário a inserção destas informações. *Tag* é uma entidade que possui uma relação de muitos para muitos com *PostRecord* e por isso é necessária uma tabela adicional *PostTag* que armazena essa relação, essa é uma relação importante e será usada para a busca de *Posts* baseado nas *tags* a que estão relacionados.

3.2.1.2 Classes e Estrutura de pastas

A implementação do *backend* é responsável pela definição do modelo de dados no banco e das operações sobre estes dados. Conforme a especificação do *Spring* cada uma das entidades é representada por uma classe, definidas na pasta *Model*, onde os atributos da classe são traduzidos em campos das tabelas e no caso de coleções de entidades, são definidas as chaves estrangeiras necessárias para representar as associações.

Acima, na figura 3.2, podemos ver como foram definidas as propriedades da entidade *PostRecord*, e como as associações são definidas diferentemente dependendo da cardinalidade e de que entidade é responsável pela chave estrangeira usando da anotação *@JoinColumn* e no caso da entidade *Tag* como é definida a tabela de uma relação *Many to Many*.

Uma camada acima existem as definições dos repositórios, que servem para fazer a interação com o modelo mais diretamente e que tem implementação feita dinamicamente pelo Spring, sendo necessário somente a criação de interfaces específicas para cada entidade do modelo. Operações simples de CRUD são providenciadas automaticamente, e é possível também definir *queries* específicas baseadas nas entidades do modelo.

Figura 3.2: Trecho de código da entidade *PostRecord*

```
PostRecord.java ×
backend > src > main > java > com > ufrgs > inf > tcc > model > PostRecord.java > PostRecord > postId
10  @Entity
11  @JsonIgnoreProperties(value = {"links","tags", "images"})
12  public class PostRecord {
13
14      @Id
15      @GeneratedValue(strategy = GenerationType.AUTO)
16      private Long postId;
17      private String title;
18      @Column(length = 2048)
19      private String description;
20      private Character status;
21      private Date creationDate;
22      private Date publicationDate;
23
24      @OneToOne(cascade=CascadeType.ALL)
25      @JoinColumn(name = "reportId")
26      private BeachReport beachReport;
27
28      @OneToOne(cascade=CascadeType.ALL)
29      @JoinColumn(name = "findingId")
30      private FindingReport findingReport;
31
32      @OneToMany(mappedBy = "postRecord")
33      private List<Link> links;
34
35      @OneToMany(mappedBy = "postRecord")
36      private List<Image> images;
37
38      @ManyToMany(fetch = FetchType.EAGER)
39      @JoinTable(name          = "Post_Tag",
40          joinColumns          = @JoinColumn(name = "postId ", referencedColumnName = "postId"),
41          inverseJoinColumns   = @JoinColumn(name = "tagId", referencedColumnName = "tagId")
42      )
43      private List<Tag> tags;
```

Figura 3.3: Repositório da entidade *PostRecord*

```
PostRecordRepository.java ×
backend > src > main > java > com > ufrgs > inf > tcc > repository > PostRecordRepository.java > ...
1   package com.ufrgs.inf.tcc.repository;
2
3   import java.util.List;
4
5   import com.ufrgs.inf.tcc.model.Image;
6   import com.ufrgs.inf.tcc.model.Link;
7   import com.ufrgs.inf.tcc.model.PostRecord;
8   import com.ufrgs.inf.tcc.model.Tag;
9
10  import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
11  import org.springframework.data.jpa.repository.Query;
12  import org.springframework.data.repository.PagingAndSortingRepository;
13  import org.springframework.data.repository.query.Param;
14  import org.springframework.stereotype.Repository;
15
16  @Repository
17  public interface PostRecordRepository extends PagingAndSortingRepository<PostRecord, Long>,
18          JpaSpecificationExecutor<PostRecord> {
19
20      @Query("SELECT tag FROM Tag tag INNER JOIN tag.postRecords post WHERE post.id = :postId")
21      List<Tag> findTagsFromPost(
22          @Param("postId") Long postId
23      );
24
25      @Query("SELECT link FROM Link link INNER JOIN link.postRecord post WHERE post.id = :postId")
26      List<Link> findLinksFromPost(
27          @Param("postId") Long postId
28      );
29
30      @Query("SELECT image FROM Image image INNER JOIN image.postRecord post WHERE post.id = :postId")
31      List<Image> findImagesFromPost(
32          @Param("postId") Long postId
33      );
34  }
```

Alguns pontos importantes na figura 3.3, são as definições de métodos para carregamento das entidades *One to Many* e *Many to Many*, e a definição da entidade fazendo a extensão para interfaces que possibilitam ordenação e especificações, funcionalidades que terão seu uso detalhado na seção 3.3.2.

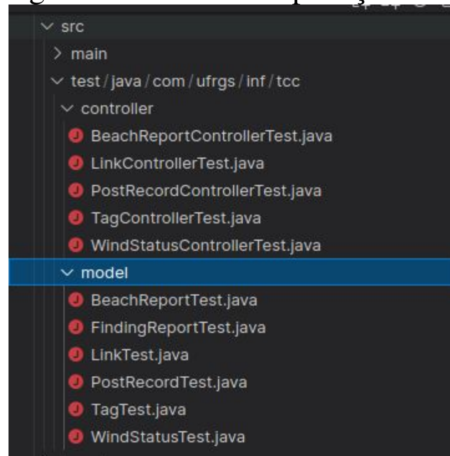
Por fim existe uma última camada de implementação de *Backend* que define a API para o tratamento de requisições HTTP, nestas classes são definidos um caminho único para o acesso àquela entidade e os mapeamentos específicos para as requisições *GET*, *PATCH*, *POST*, *DELETE*. Quando uma requisição é feita não obedecendo essas definições das classes *Controller* um erro de política de CORS é lançado automaticamente. Outras classes auxiliares para a criação de *specifications*, e para definição de exceções também são implementadas nessa camada.

Figura 3.4: Objetos e pastas da aplicação de BE

Na figura 3.4 podemos ver a lista completa de objetos produtivos separados em pastas conforme as funcionalidades. Também foram criados testes unitários para algumas das classes da pasta *model* e *controller* utilizando da ferramenta JUnit como poder ser

visto na figura 3.5 abaixo.

Figura 3.5: Testes da aplicação de BE



3.2.2 Frontend

3.2.2.1 Telas

A implementação de FE é uma única aplicação mas que pode ser subdividida em telas com funcionalidades específicas.

- *Home* página inicial com botões de acesso às outras telas.
- *Create* com as funções usadas pelos componentes editáveis na tela de criação de *Posts*.
- *Search* para definir o layout da busca de *Posts* e para a construção das requisições de filtros e ordenação.
- *Display* para definir o layout de exibição e das funções necessárias para exibir os dados salvos de um *Post*.

3.2.2.2 Funções e Estrutura de pastas

Por se tratar de uma aplicação React, o código é baseado em funções que retornam os componentes a serem renderizados na tela, e o controle de estado e a execução das requisições de integração com o BE é feito utilizando *Hooks* que facilitam a implementação por evitar o uso classes. Seguindo o fluxo de execução, a aplicação é iniciada através da definição no arquivo *index.js*, e essa por sua vez executa a renderização do controle de rotas do arquivo *pageRouter.js*, e dependendo de qual caminho é usado na URL do navegador a aplicação e direcionada para uma das telas do app.

Figura 3.6: Arquivo *index.js* da aplicação de FE

index.js

frontend > src > index.js > ...

```javascript
import 'semantic-ui-css/semantic.min.css'
import React from "react";
import ReactDOM from "react-dom";
import { Container} from "semantic-ui-react";
require('dotenv').config();

import PageRouter from "./common/pageRouter"

const App = ({ children }) => (
  <Container style={{ margin: 20 }}>


    {children}
  </Container>
);

ReactDOM.render(
  <App>
    <PageRouter />
  </App>,
  document.getElementById("root")
);

```

Figura 3.7: Arquivo *pageRouter.js* da aplicação de FE

pageRouter.js

frontend > src > common > pageRouter.js > PageRouter

```javascript
import {
    BrowserRouter as Router,
    Switch,
    Route
  } from "react-router-dom";

  import NewPost from "../create/newPost"
  import DisplayPost from "../display/displayPost"
  import Home from "./home"
  import SearchPost from "../search/searchPost";
  import EditPost from "../create/editPost";

export default function PageRouter(){
    return (
        <Router>
            <div>
                <Switch>
                    <Route path="/Create">
                        <NewPost />
                    </Route>
                    <Route path="/Edit/:postId">
                        <EditPost />
                    </Route>
                    <Route path="/Display/:postId">
                        <DisplayPost />
                    </Route>
                    <Route path="/Search">
                        <SearchPost />
                    </Route>
                    <Route path="/">
                        <Home />
                    </Route>
                </Switch>
            </div>
        </Router>
    );
}
```

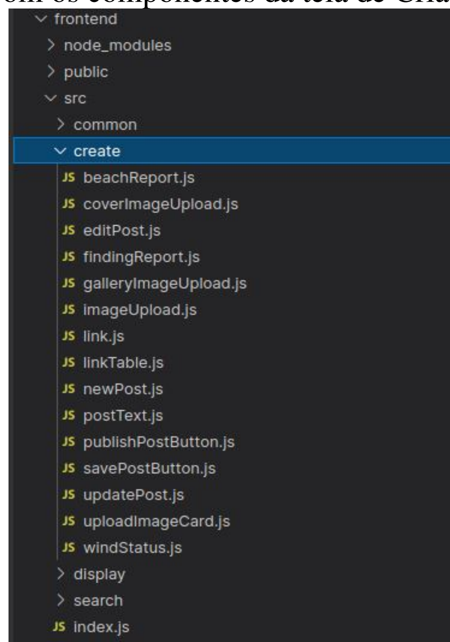
Como é possível ver nas Figuras 3.6 e 3.7 a aplicação é redirecionada automaticamente para a tela correspondente ao caminho requisitado na URL. Normalmente a aplicação é acessada pelo link[2] do servidor do Museu do Mar e por isso é encaminhada para a *Home* do aplicativo no caminho “/”.

[2]<http://musmar.inf.ufrgs.br/>

A *Home* do aplicativo foi implementada com o objetivo de mostrar as opções para o usuário de quais funcionalidades estão disponíveis. Nesta tela existem dois botões que fazem a navegação para as outras telas. O botão “Novo Post” faz a navegação “/Create” na URL, o que faz o router da aplicação passar a renderizar o componente *NewPost*, já o botão “Buscar Posts” faz a navegação para “/Search” e dispara a renderização do componente *SearchPost*, a figura 4.1 mostra cada um desses botões na página inicial.

O componente *NewPost* serve para executar a requisição de criação de um *Post*, e então a partir do novo identificador criado, fazer a navegação diretamente para o componente *EditPost* com o caminho “/Edit/:postId”, sendo *:postId* a forma de identificar uma variável na URL que é substituída pelo valor do novo identificador, por exemplo o primeiro *Post* criado teria uma navegação “/Edit/1”. Na tela de edição existem os componentes usados para renderizar cada seção e as funções para fazer as requisições *GET* durante o carregamento da página para manter os valores já salvos, e funções para requisições *POST*, *PATCH* e *DELETE* quando o *Post* é salvo.

Figura 3.8: Pasta com os componentes da tela de Criação e Edição de *Posts*

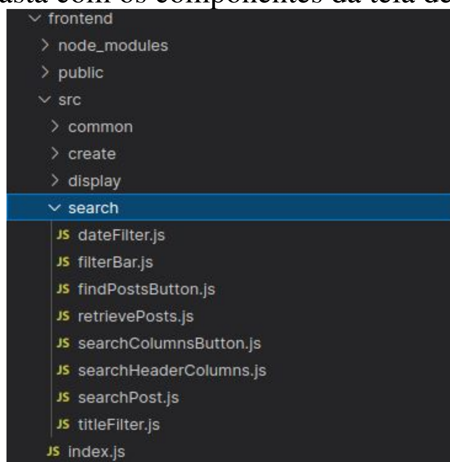


Na figura 3.8 estão listados todos os componentes utilizados para renderizar e controlar a tela de Edição de *Posts*, incluindo seções de campos, botões e funções de integração com o BE através de requisições HTTP.

O componente *SearchPost* faz a renderização da tela de busca de *Posts*, mostrando os *Posts* existentes em uma tabela de resultados, com funcionalidades de busca através de campos de filtro e de ordenação dos resultados usando as colunas da tabela. Na tela de buscas existem os componentes para construir as requisições usando as informações

passadas como filtros e de ordenação, para renderização da tabela e dos filtros. Clicando no título de um *Post* é executada a navegação para o componente *DisplayPost* através da navegação "/Display/:postId".

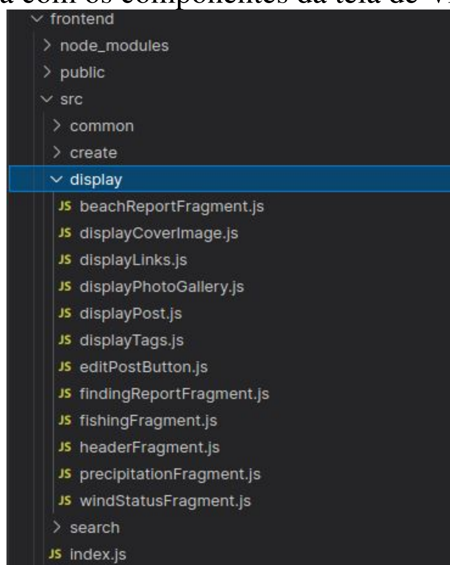
Figura 3.9: Pasta com os componentes da tela de Busca de *Posts*



Na figura 3.9 se pode ver os componentes da tela de busca, da tabela de exibição, dos filtros, de botões para customização das colunas e de atualizar os resultados, e a função para realizar as requisições *GET* utilizando as definições de ordenação e filtro.

Por fim, o componente *DisplayPost* faz a apresentação das informações salvas em um *Post* específico, entre as funcionalidades está a possibilidade de abrir as imagens em tamanho original ao clicar nas imagens e nos links colocados como referência, e os campos estarem visíveis ou não dependendo se há valores salvos. Quando disponível também existe um botão para a edição do *Post*.

Figura 3.10: Pasta com os componentes da tela de Visualização de *Posts*



Na figura 3.10 pode se ver os componentes da tela de visualização. As diferentes

seções da tela foram divididos em fragmentos para melhor organizar o código, e organizados de forma que as regras de visibilidade dos campos seja implementada somente onde relevante.

3.2.2.3 Integração com Backend

A integração da aplicação com a API implementada no BE foi feita usando Fetch para montar e disparar as requisições. Foi importante também a preocupação com qual URL usar para fazer as requisições, uma vez que quando rodando localmente a aplicação de FE pode fazer requisições para a máquina local, via "http:localhost" ou pelo IP 127.0.0.1 para acessar a aplicação de BE que também roda localmente durante testes, mas quando está rodando dentro do *container* docker, é necessário usar a URL com o IP do *container* de BE. Por isso foi usado dotenv para armazenar qual a URL corrente, assim quando rodando localmente é usado o endereço local, mas quando o *container* é construído essa variável é substituída.

Figura 3.11: Exemplo de chamada para requisição usando Fetch

```
let {postId} = useParams();
const retrievePost = useCallback(async ()=>{
    const requestOptions = {
        method: 'GET'
    };
    const response = await fetch(process.env.REACT_APP_API_URL+'/posts/'+postId, requestOptions);
    const data = await response.json();
    setPost(data);
}, [postId]);
```

Na figura 3.11 temos um exemplo de requisição para acessar um *Post* e carregar todos os campos na aplicação de visualização.

3.2.3 Infraestrutura

3.2.3.1 Exposição para Internet

Para exposição da aplicação na internet, foi necessário utilizar o servidor web NGINX para administrar as requisições recebidas. Isso se dá por que com a aplicação rodando em diferentes *containers*, não seria possível fazer com que os *containers* de BE e FE escutassem a mesma porta 80, nem seria possível manter apenas o *container* de FE nessa porta, uma vez que o código da aplicação FE é enviado para o navegador do usuário e não seria possível fazer as requisições para o BE a partir do código *client-side*.

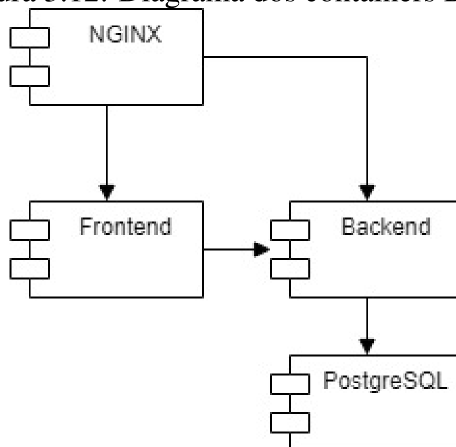
Por isso foi inserido este *container* NGINX que é responsável por fazer o proxy reverso para cada um dos *containers*. Assim o único *container* utilizando a porta 80 passa a ser o NGINX, e este quando recebe requisições na localização "/" é redirecionado para o *container* de FE, e quando a localização é "/api/v1" passa para o *container* de BE.

3.2.3.2 Definição dos containers

Ter a aplicação executando dentro de *containers* Docker é uma das funcionalidades mais importantes da aplicação, uma vez que para ser *Cloud-Native* precisaria necessariamente de microsserviços com processos automáticos de *build* e execução.

Para a definição dos *container* foi utilizada a ferramenta de Docker-Compose, que possibilita a definição de vários *containers* da mesma aplicação em um mesmo arquivo, e *scripts* para fazer o *build* e execução de forma mais direta. Para cada um dos *containers* foi necessário escolher uma imagem mais apropriada para o objetivo daquele *container*. Para o *container* NGINX foi utilizada a imagem oficial dockerizada do servidor [3], para o *container* de *frontend* foi utilizada a imagem oficial do Node[4] e um arquivo Dockerfile adicional para executar a instalação da aplicação dentro do *container*, para o *backend* a imagem Maven [5] e para o banco de dado a imagem PostgreSQL[6].

Figura 3.12: Diagrama dos containers Docker



No diagrama da figura 3.12 se tem representados os *container* responsáveis pela execução da aplicação como os módulos, as flechas direcionam no sentido em que as requisições são despachadas. No caso de o acesso ser feito externamente, ou se usada a porta 80 do servidor, as requisições passam primeiro pelo *container* NGINX e depois re-

[3]<https://hub.docker.com/_/nginx>
[4]<https://hub.docker.com/_/node>
[5]<https://hub.docker.com/_/maven>
[6]<https://hub.docker.com/_/postgres>

direcionadas para as portas onde o estão executando os *container* de FE e BE. O *container* de BE é o único com acesso ao *container* de banco de dados PostgreSQL.

3.3 Funcionalidades Detalhadas

3.3.1 Criação de Posts

A tela de criação de *Posts*, veja a figura 4.2 para a interface, permite que diversas das informações que hoje são adicionadas como texto de uma postagem do Facebook sejam também adicionadas em campos específicos na tela, isso possibilita que esses dados sejam usados depois na ordenação dos resultados de busca. Enquanto que a maioria dos campos pode ser considerado uma entrada simples de campo, seja via texto ou numérico, alguns destes campos possuem funcionalidades mais complexas e que são detalhadas nas subseções abaixo.

3.3.1.1 Upload de Imagens

O *upload* de imagens é feito a partir de uma integração entre um componente de entrada do tipo arquivo com as requisições específicas para *upload* e download de imagens definidos na API de BE.

Para a criação de um imagem é feita uma requisição de *POST* para criar de um registro na entidade *Image*, como ainda não se tem um identificador próprio para imagem é usado um método mapeado para a entidade de *PostRecord* para adicionar uma nova imagem relacionada. No corpo da requisição é utilizado de um objeto *FormData* para construir no formato *multipartImage* aceito pela API REST para enviar as informações do arquivo em si. Quando é feita uma atualização da imagem, é feita uma requisição *PATCH* somente para o identificador da imagem diretamente atualizando o arquivo binário.

A função de *upload* é usada em dois componentes da interface, *CoverImageUpload* e *GalleryImageUpload* onde existe a distinção da forma como é exibido o botão de *upload* e deleção de imagem, baseado na categoria da imagem. No caso da imagem principal *CoverImageUpload* é usado e somente é possível adicionar uma imagem com essa categoria no *Post*, já para as seções de “Animais Encontrados” e “Lixo Encontrado” é usado em cada uma o componente *GalleryImageUpload*, que permite a adição de múltiplas imagens em cada seção identificadas pela categoria da seção para determinar onde

cada imagem deve aparacer depois de uma atualização da página ou quando aberto o *Post* em modo de visualização.

3.3.1.2 Tags de Busca

Durante a criação de um *Post* é possível selecionar *Tags* de busca que sejam relevantes para o conteúdo do *Post*. Isso se é feito usando o componente *TagMultiselect* que possui o comportamento de um campo de escolha múltipla com as opções sendo carregadas do sistema, mas que também é possível adicionar novas *Tags* ao se digitar um novo nome para uma *Tag* e pressionar *Enter*.

Por causa disso existe uma complexidade no tratamento das entidades, uma vez que existe a necessidade de fazer uma requisição *POST* para criar as novas *Tags* que forem sendo adicionadas e é necessário atualizar os opções possíveis para seleção das *Tags*. Quando o usuário faz a seleção ou cria novas *Tags*, ainda não é criada a relação entre as entidades *PostRecord* e *Tag* no BE, isso somente é feito quando o botão “Salvar” é utilizado, já que é necessário atualizar as relações de todas as *Tags* que já estavam no *Post* anteriormente caso tenham sido removidas, e adicionar a relação àquelas que foram adicionadas.

3.3.1.3 Adicionar links relacionados

Para adição de múltiplos links a um *Post* foi criado um componente no formato de uma tabela editável, onde é possível adicionar e remover linhas, adicionando a informação de cada link, seu nome para exibição e URL, em cada uma das linhas.

Para esse componente a inserção dos valores acontece somente quando o *Post* é salvo, sendo possível desfazer a remoção de uma linha antes que essa mudança seja feita efetivamente, evitando retrabalho de inserir estas informações novamente caso se tenha apertado o botão por engano.

3.3.2 Busca de Posts

A tela de buscas, ver figura 4.3, foi desenvolvida para que seja possível ver as informações de várias postagens diferentes de forma consolidada, e com funcionalidades que possibilita se encontrar mais facilmente a informação. Os resultados são mostrados em uma tabela com as colunas “Título” e “Texto” do *Post* por padrão.

3.3.2.1 Colunas Adicionais

Para deixar mais customizável a visualização dos resultados, foi implementado um botão para que seja possível ver características adicionais do *Post*. O botão mostra todas as opções de campos possíveis, e fica a critério do usuário quais colunas adicionais são necessárias no resultado da busca.

Como todas as informações da tela já foram carregadas, esta mudança das colunas afeta apenas a renderização dessa tabela, já que à parte dos campos padrão “Título” e “Texto” as outras colunas são renderizadas dinamicamente dependendo do que for selecionado, não é necessária nenhuma implementação de BE para isso. As colunas disponíveis são aquelas da entidade *PostRecord* e das associações 1:1, e são exibidas na ordem que forem selecionadas pelo usuário.

3.3.2.2 Ordenação por Coluna

Cada uma das colunas da tabela de resultados pode ser usado para fazer a ordenação pelo campo descrito ao se clicar em cima do nome da coluna. Quando clicado a ordenação da tabela passa a ser em ordem decrescente para aquele campo, ao clicar novamente passa a ser crescente e na terceira vez desfaz a ordenação. Somente é possível ordenar por um campo de cada vez, e quando isso acontece é feita uma nova requisição para o API no BE para atualizar os valores considerando todos os registros no banco e não somente aqueles na tela.

Para realizar essa ordenação, é feita uma requisição específica para a API, que usando o objeto *Sort* usa o repositório da entidade de *PostRecord* para retornar os valores ordenados.

Figura 3.13: Trecho da classe *PostRecordController* responsável pela ordenação

```
@GetMapping("/orderedBy")
@ApiOperation(value = "Get Ordered Posts", nickname = "findAllOrderedBy")
public Iterable<PostRecord> findAllOrderedBy(@RequestParam(required = false) String entityName, @RequestParam("field") String field, @RequestParam("order") String order) {
    return postRecordRepository.findAll(Sort.by(Sort.Direction.fromString(order), getOrderByFieldName(entityName, field)));
}
```

Na figura 3.13 se tem a implementação do mapeamento da requisição responsável por ordenar os resultados, os valores recebidos da requisição são a ordem, se crescente ou decrescente, qual a entidade caso o campo faça parte de uma outra entidade associada, e qual o nome do campo.

3.3.2.3 Filtros de Busca

Para encontrar os *Posts* relevantes na busca foi implementada uma barra de filtros onde que pode ser usada para atualizar os resultados da tabela baseado no que for informado pelo o usuário usando os campos disponíveis. É possível filtrar os *Posts* pelo campo de “Titulo”, para buscar por palavras que podem estar em qualquer trecho no nome do *Post*, por *Tags* de busca que estiverem associadas àqueles *Posts*, e por data ou período em que os Posts foram publicados. Este filtro de datas usa um componente de seleção de datas que não estava disponível por padrão na biblioteca do Semantic UI, e foi adicionado como uma dependência adicional ao projeto. O filtro de *Tags* disponibiliza todas as *Tags* existentes no sistema para seleção, similarmente a como é usado na tela de criação, mas nesse caso é desabilitada a funcionalidade de adicionar novas *Tags*.

Para a realização dos filtros ao resultado foi importante considerar que todos os campos afetam o resultado simultaneamente, e também que us resultado filtrado precisa manter a ordenação definida pelas colunas da tabela consistente. Para isso foi definido na API requisições que fazem a combinação entre filtrar e ordenar os resultados. Para gerar as combinações dos filtros por sua vez foi utilizada a definições de especificações, que então são utilizadas dinamicamente pelos métodos do repositório da entidade *PostRecord*.

Figura 3.14: Classe responsável pela criação das especificações

```
PostRecordCombineSpecification.java
backend > src > main > java > com > ufrgs > inf > tcc > controller > PostRecordCombineSpecification.java > PostRecordCombineSpecification > combine()
1  package com.ufrgs.inf.tcc.controller;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.stream.Collectors;
6
7  import com.ufrgs.inf.tcc.model.PostRecord;
8
9  import org.springframework.data.jpa.domain.Specification;
10
11 public class PostRecordCombineSpecification {
12
13     private final List<SearchCondition> conditions;
14
15     public PostRecordCombineSpecification() {
16         conditions = new ArrayList<SearchCondition>();
17     }
18
19     public PostRecordCombineSpecification with(String fieldName, String operator, Object fieldValue) {
20         conditions.add(new SearchCondition(fieldName, operator, fieldValue));
21         return this;
22     }
23
24     public Specification<PostRecord> combine() {
25         if (conditions.size() == 0) {
26             return null;
27         }
28
29         List<Specification<PostRecord>> specs = conditions.stream()
30           .map(PostRecordSpecification::new)
31           .collect(Collectors.toList());
32
33         Specification<PostRecord> result = specs.get(0);
34
35         for (int i = 1; i < conditions.size(); i++) {
36             result = Specification.where(result)
37                     .and(specs.get(i));
38         }
39
40         return result;
41     }
42
43 }
```

Pode ser visto na figura 3.14 como são construídas as especificações. A partir

da requisição de filtro a classe *Controller* responsável cria as condições em são baseadas a especificações fazendo a chamada ao método *with* e ao fim chama o método *combine* para instanciar cada uma das especificações com as condições e combinar e uma única onde todos os predicados são ligados por uma cláusula *and*. Essa única especificação com todos os predicados criados é então passada via parâmetro para o método *findAll* do repositório, que lida automaticamente com as condições para montar a *query* SQL para o banco de dados.

3.3.3 Visualização de Posts

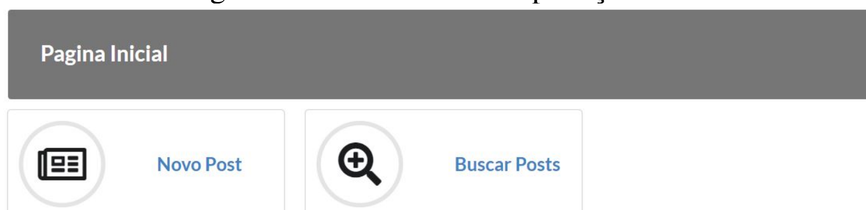
A visualização dos *Posts* não possui funcionalidades complexas como foi o caso as outras telas, tem de relevante apenas um controle de visibilidade das informações para esconder as seções e campos que não tiverem nenhuma informação corrente, deixando mais limpa a tela somente com aquilo que pé relevante. É disponibilizado um botão para fazer navegar para a tela de edição daquele *Post*.

4 GUIA DE USO

Neste capítulo se detalha o passo a passo de utilização da aplicação pelos usuários principais, o Pescador, o Curador e o Estudante. Hoje ainda não existe uma implementação que verifique os usuários e limite as funcionalidades disponíveis, então o detalhamento a seguir pode ser visto também como qual a intenção dessa funcionalidade futura.

Ao iniciar a aplicação todos os usuário são apresentados com a tela inicial, vista na figura 4.1, dependendo se o usuário teria essa autorização o botão de "Novo Post" não é disponibilizado.

Figura 4.1: Tela *Home* da aplicação de FE



4.1 Pescador

O Pescador é o usuário que começa com a criação dos *Posts*, ele faz o relato da situação daquele dia utilizando dos campos que achar necessário. Para isso ele começa clicando no botão "Novo Post" na página inicial, ver 4.1.

Então a tela passa a ser a tela de criação de um *Post*, figura 4.2, esta tela poderia ser simplificada para não mostrar campos que dificilmente o Pescador faria alguma modificação. Ele é livre para adicionar o título e o texto do *Post*, e usar os campos estruturados de informação para descrever a situação da praia naquele momento, adicionar uma imagem de capa para o *Post*.

Caso tenha feito algum achado, pode registrar de forma simples com imagens dos animais ou do lixo encontrado usando as galerias em cada uma das seções, adicionando descrições simples.

Ao fim pode usar o botão de salvar para manter as informações registradas no sistema, e pode publicar para que o *Post* fique disponível na ferramenta de busca para os Estudantes.

Figura 4.2: Tela *Create* da aplicação de FE

4.2 Curador

O Curador é o usuário que ajusta e incrementa as informações adicionadas pelo Pescador, para isso tem acesso a todas as funcionalidades da tela de edição de *Posts* da figura 4.2.

Tudo aquilo que poderia ser escondido da visualização do Pescador pode ser mostrada para este usuário complementar, como adicionar os links para outros *Posts* na aplicação, ou para artigos e documentos externos com mais informações relevantes. Pode detalhar também aquilo que o Pescador colocar como uma achado com nomes científicos, e ajustar o que for necessário para se ter um melhor entendimento do texto.

Na tela de busca, figura 4.3 esse usuário conseguiria ver também os *Posts* que tenham ficado incompletos e não foram ainda publicados para os estudantes, podendo ajustar e publicar.

4.3 Estudante

Os usuários Estudantes seriam aqueles que acessam a aplicação para consultas e pesquisas, por isso quando usam a aplicação teriam disponível somente as telas de busca e visualização, imagens 4.3 e 4.4.

Na tela de busca teria acesso a todos os *Posts* publicados pelo Pescador e pelo Curador, podendo acessar a visualização do *Post* clicando no título do *Post*. Para adicionar novas colunas de informação o botão "Mais Colunas" pode ser usado para escolher quais outras informações achar relevante.

Clicando no nome das colunas é possível ordenar os resultados em ordem crescente ou decrescente clicando mais uma vez, e removendo a ordenação no terceiro clique. Já para filtrar os resultados existem três opções:

- Por título: o usuário pode escrever qualquer frase no campo e os resultados são somente aqueles *Posts* que incluírem no título o trecho buscado.
- Por *Tag*: o usuário pode filtrar se baseando em quais *Tags* um *Post* está classificado, pode escolher múltiplas *Tags* para ser mais específico.
- Por data de publicação: o usuário pode escolher uma data específica em que um *Post* foi publicado, ou um período ao escolher uma data inicial e um data final para a busca.

Figura 4.3: Tela *Search* da aplicação de FE

Buscar Posts

Filtros:

Titulo | Tags de Busca | Data de Publicaçao

Tags... | DD MMM, YYYY

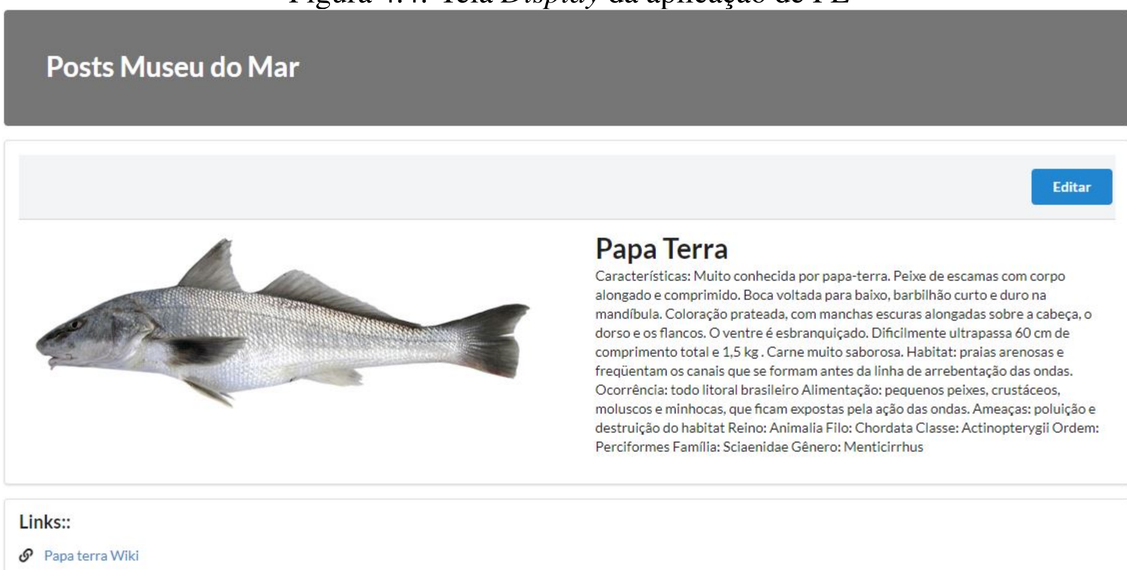
Procurar

Data de Publicaçao

Titulo	Texto	Data de Publicaçao
Papa Terra	Características: Muito conhecida por papa-terra. Peixe de escamas com corpo alongado e comprimido. Boca voltada para baixo, barbilhão curto e duro na mandíbula. Coloração prateada, com manchas escuras alongadas sobre a cabeça, o dorso e os flancos. O ventre é esbranquiçado. Dificilmente ultrapassa 60 cm de comprimento total e 1,5 kg . Carne muito saborosa. Habitat: praias arenosas e freqüentam os canais que se formam antes da linha de arrebentação das ondas. Ocorrência: todo litoral brasileiro Alimentação: pequenos peixes, crustáceos, moluscos e minhocas, que ficam expostas pela ação das ondas. Ameaças: poluição e destruição do habitat Reino: Animalia Filo: Chordata Classe: Actinopterygii Ordem: Perciformes Família: Sciaenidae Gênero: Menticirrhus	25 Oct, 2021
Animais Marinhos (Botos)	O boto é o nome que é conhecido por uma família de baleias com dentes, porque eles vivem exclusivamente em um contexto de água e têm dentes em vez de barbatanas, certamente, os grupos heterogêneos em torno de 34 espécies no mundo inteiro. Também conhecida como golfinhos do oceano. Eles são caracterizados por várias características físicas, incluindo: medindo entre 2 e 9 metros de comprimento, tem corpo fusiforme, sua cabeça é grande, focinho alongado e proprietários de um único poro, como é chamado o orifício através do qual respira e que está localizado no topo da cabeça.	25 Oct, 2021
Teste museu Quintao	Dia bom para pesca	26 Oct, 2021
O mar está		

Na tela de visualização é possível ler as informações salvas no *Post*, ver as imagens e acessar os links. Também é possível abrir as imagens em tamanho original clicando nelas.

Figura 4.4: Tela *Display* da aplicação de FE



5 AVALIAÇÃO/VALIDAÇÃO

Para validar a aplicação foi criado um questionário usando *Google Forms*[1] para capturar inputs de usuários, foram utilizadas perguntas no modelo SUS para verificar a usabilidade do sistema, e também perguntas de resposta livre para adicionar *feedbacks*. Foram descritas algumas tarefas que poderiam ser realizadas no sistema de forma que ficasse claro o que poderia ser feito, mas sem ser um passo a passo diretamente, de forma a conseguir fazer essa avaliação sem influenciar a percepção do usuário quanto à complexidade. No Apêndice A o questionário pode ser visto por completo.

O questionário foi disponibilizado para o grupo do Museu do Mar e também para outros orientandos do professor Leandro, e foram recebidas seis respostas no total. Os resultados de cada pergunta, a totalização e os *feedbacks* recebidos são detalhadas nas sessões abaixo. Se entende que os resultados podem ser afetados pelo grau de escolaridade e da familiaridade dos participantes com ferramentas online, no futuro é proposto que a ferramenta seja testada com mais usuários chave para a aplicação, como estudantes da comunidade da região, e com o uso no dia-a-dia pelo Pescador.

5.1 Resultados SUS

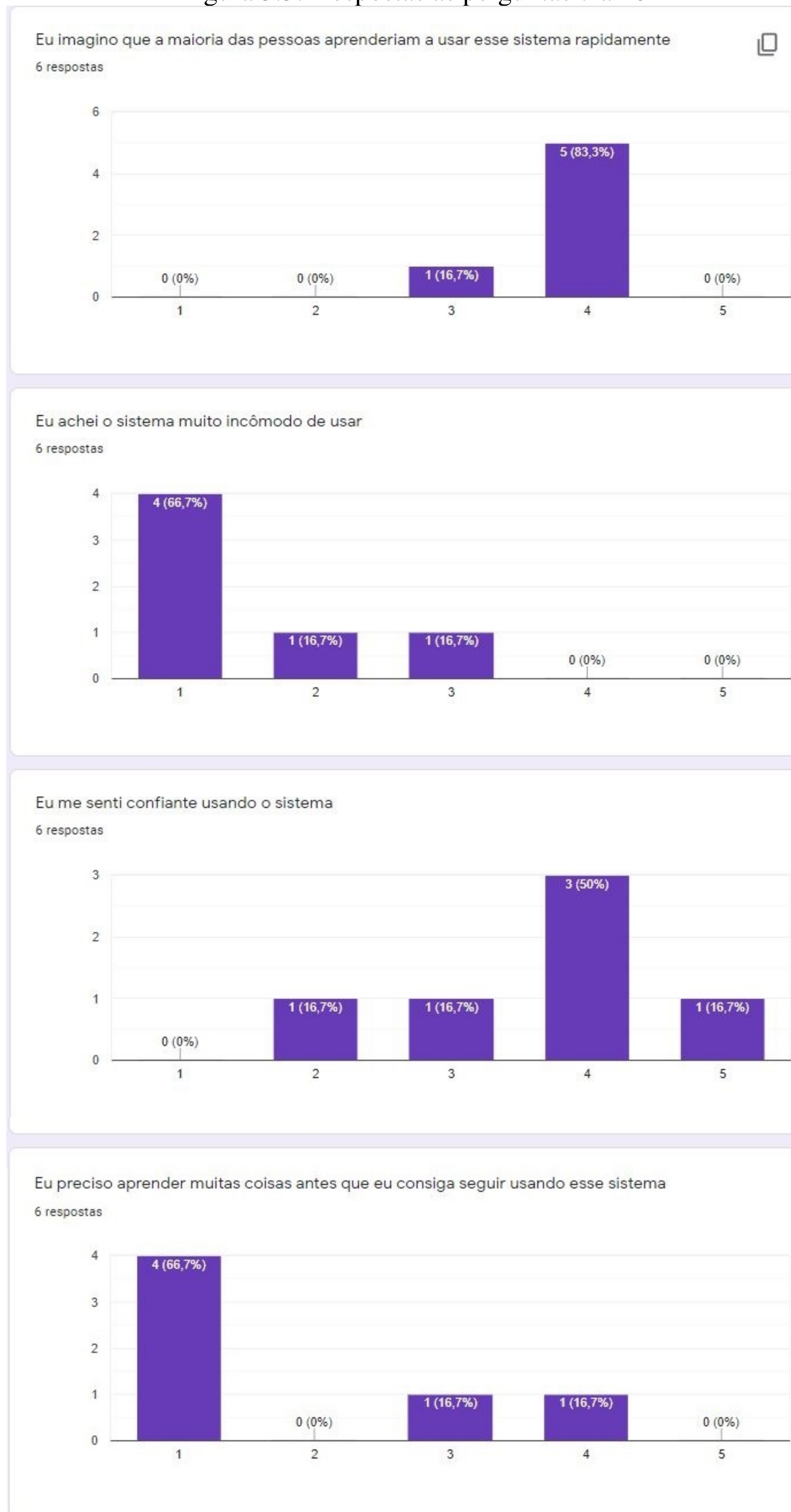
Para o questionário SUS são feitas dez perguntas usando uma escala Likert com cinco opções variando de “Discordo Fortemente” a “Concordo Fortemente”.

[1]<https://forms.gle/GPjuEsNfbK5HeQ3V8>

Figura 5.1: Respostas às perguntas 1 a 3

Figura 5.2: Respostas às perguntas 4 a 6

Figura 5.3: Respostas às perguntas 7 a 10

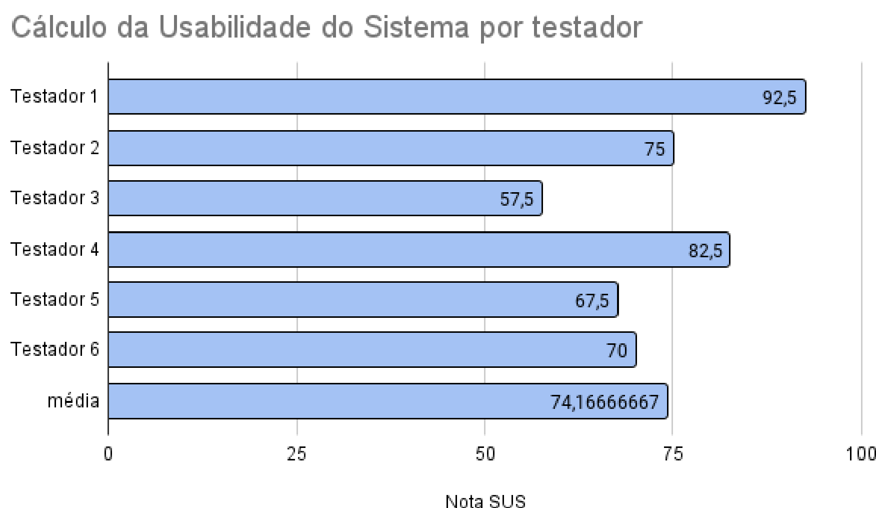


Para o cálculo da pontuação SUS é usada a seguinte regra, como mostrada em

Smyk (2020):

- Somar as notas de todas as respostas ímpares e subtrair 5
- Somar as notas de todas as respostas pares, subtrair este valor de 25
- Somar os dois valores anteriores e multiplicar por 2,5

Figura 5.4: Notas calculadas para cada testador e média



No gráfico da figura 5.4 podemos ver a distribuição das notas calculadas para cada um dos testadores, onde a menor nota foi de 57,5 pontos e a maior de 92,5 pontos, a média da pontuação foi de 74 pontos e pode se considerar que aplicação teve sucesso no teste por estar acima da média de 68 definida na literatura.

5.2 Feedbacks

Foram coletados também *feedbacks* da aplicação em texto livre, assim se pode encontrar pontos de melhoria, novas ideias para funcionalidades futuras e para entender as inconsistências encontradas. Para isso foram feitas as seguintes perguntas:

- O que você mais gostou sobre o sistema? Algo te surpreendeu positivamente?
- Encontrou algum problema? Algo específico que poderia ter sido feito de outra forma?
- Alguma sugestão de funcionalidade? Algo que poderia ser incluído no futuro e que hoje não tem como opção?

A resposta a essas perguntas não era obrigatória e por isso não se teve o mesmo

número de participantes, mas traz algumas percepções interessantes da aplicação.

Para a pergunta "O que você mais gostou sobre o sistema? Algo te surpreendeu positivamente?" se obteve as respostas:

1. Em termos de usabilidade, design *clean*
2. Simplicidade
3. Todas as funções, campos de preenchimento e espaços para inserção de links e fotos estão bem claros quanto sua função.

Para a pergunta "Encontrou algum problema? Algo específico que poderia ter sido feito de outra forma?":

1. O conceito da interface está bem pensado, de forma geral. Sugiro incluir falta um tutorial explicando o propósito do sistema. Em termos de usabilidade, a navegação é prejudicada. Em momentos, você só consegue voltar para o início clicando no botão de voltar do navegador.
2. Vários: (1) nos campos, sugerir valores, com base em valores de um dicionário ou com base em *Posts* anteriores, em especial para peixes mais capturados, direção do vento, animais encontrados, lixo, espécie. (2) botão de salvar repetido no final do formulário.(3) Dar *feedback* do botão "salvar". não dá para saber se salvou. Aliás, para que serve essa opção? não está claro se é possível salvar e continuar mais tarde, outro dia.. E onde essa edição pode ser feita se isso for possível. (4) Ao publicar, aparece a opção "editar" e mais nada. Incluir a opção de voltar ao menu inicial. (5) A página de busca apresenta *Posts* em branco. Não deixar salvar ou publicar *Posts* em branco. (6)No celular, a opção de anexar fotos tem problemas. O celular permite tirar uma foto ou anexar uma foto existente da galeria. Tirar fotos na hora e mandar não funciona inicialmente. Mas, se escolhermos uma foto da galeria ele funciona. E se escolhermos uma foto da galeria e depois tentarmos tirar uma foto, ele aceita.
3. Após criar e realizar uma postagem não há uma opção de retorno a tela inicial.
4. A função de retorno de página deveria funcionar clicando no campo superior da tela.

E para a pergunta "Alguma sugestão de funcionalidade? Algo que poderia ser incluído no futuro e que hoje não tem como opção?":

1. Publicação direto no *feed* do facebook; login e controle de usuários

2. A caixa de texto onde se clica para entrar no *Post* poderia ser de uma cor diferente do restante. O botão salvar poderia ser de outra cor, talvez um tom diferente de azul. A cor preta passa a impressão de que o botão já foi clicado. No campo de busca pelos *Posts*, no meu teste só aceitou a palavra exata fazendo distinção entre letras maiúsculas e minúsculas.

6 CONCLUSÕES

Este trabalho foi realizado com a intenção de prover um meio tecnológico diferente para aproximar os conhecimentos do Pescador e líder comunitário com os Estudantes, já que hoje as informações são disponibilizadas via rede social e têm o alcance limitado, sendo difícil de ver todas as informações consolidadas, não sendo possível fazer uma pesquisa sobre as postagens.

O objetivo da implementação é o de ser um MVP para a solução deste problema, para que possa ser utilizada para a realização das tarefas coletadas nas histórias de usuário, de uma forma fácil e intuitiva. Nesse sentido, os requisitos funcionais foram implementados com sucesso, as funcionalidades existentes foram testadas e obteve um resultado positivo de 74/100 pontos na escala de usabilidade.

Dentre os requisitos não-funcionais, é importante ressaltar que a infraestrutura da aplicação foi implementada com uma base sólida e de fácil configuração, logo pode ser facilmente convertida para um uso real e produtivo, seja num servidor próprio ou em servidor em nuvem, o que também facilita a manutenção desse código no futuro. Porém a falta de um mecanismo de logon torna inviável colocar a implementação atual em uso efetivamente, já que não há nenhuma restrição para as atividades possíveis de cada usuário.

Apesar de não ser viável o uso da aplicação no presente, ela dá um "Norte" para implementações futuras que estendam as funcionalidades hoje existentes. Algumas das funcionalidades futuras que poderiam ser adicionadas:

- Integração com Postagens ao Facebook, fazendo com que seja mais prático para o Pescador fazer a inserção dessas informações somente uma vez através da aplicação e automaticamente criar a postagem na rede social, isso também aumentaria em muito o valor percebido pelo usuário.
- Melhorias de usabilidade, como o uso de *breadcrumbs* para fazer as navegações para páginas anteriores, adicionar paginação para a tela de buscas para limitar o número de resultados, a possibilidade de filtrar os resultados por campos adicionais, exportar os resultados como planilha, mostrar sugestões de valores para adicionar aos campos durante a criação.
- Visual e apresentação, incorporar o app efetivamente em um Portal para o Museu do Mar junto de outras aplicações, ter um design consistente entre o site do Portal e a aplicação, com uso de cores, fontes e outras definições a partir de CSS.

- Aumentar escopo de interação, possibilidade de haverem comentários dentro dos *Posts*, páginas com perguntas frequentes, links entre outras aplicações de acervo do Museu.

REFERÊNCIAS

APACHE. **Maven**. DockerHub, 2017. [Online; acessado em fevereiro de 2021]. Available from Internet: <https://hub.docker.com/_/maven>.

DENNER, A. **Semantic UI Datepicker**. GitHub, 2018. [Online; acessado em outubro de 2021]. Available from Internet: <https://github.com/arthurdenner/react-semantic-ui-datepickers>.

DOCKER. **Docker-Compose**. DockerDocs, 2013. [Online; acessado em fevereiro de 2021]. Available from Internet: <https://docs.docker.com/compose/>.

F5. **NGINX**. DockerHub, 2015. [Online; acessado em outubro de 2021]. Available from Internet: <https://hub.docker.com/_/nginx>.

FACEBOOK. **React**. GitHub, 2013. [Online; acessado em março de 2021]. Available from Internet: <https://github.com/facebook/react>.

FACEBOOK. **Create React App**. GitHub, 2016. [Online; acessado em março de 2021]. Available from Internet: <https://github.com/facebook/create-react-app>.

MERGEL, I. Agile innovation management in government: A research agenda. **Government Information Quarterly**, v. 33, n. 3, p. 516–523, 2016. ISSN 0740-624X. Open and Smart Governments: Strategies, Tools, and Experiences. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0740624X16301101>.

MOMENT. **Moment**. GitHub, 2011. [Online; acessado em outubro de 2021]. Available from Internet: <https://github.com/moment/moment>.

MOTTE, S. **Dotenv**. GitHub, 2013. [Online; acessado em março de 2021]. Available from Internet: <https://github.com/motdotla/dotenv>.

NPM. **npm**. GitHub, 2009. [Online; acessado em março de 2021]. Available from Internet: <https://github.com/npm/cli>.

POSTGRESQL. **PostgreSQL**. DockerHub, 2015. [Online; acessado em fevereiro de 2021]. Available from Internet: <https://hub.docker.com/_/postgres>.

REHKOPF, M. **User stories with examples and a template**. Atlassisan, 2020. [Online; acessado em Outubro de 2021]. Available from Internet: <https://www.atlassian.com/agile/project-management/user-stories>.

REMIX. **React Router**. GitHub, 2015. [Online; acessado em abril de 2021]. Available from Internet: <https://github.com/remix-run/react-router>.

SCHWABER, K.; JEFF, S. The Scrum Guide. Scrum.Org, 2020. [Online; acessado em Outubro de 2021]. Available from Internet: <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf>.

SEMANTIC. **Semantic UI React**. GitHub, 2015. [Online; acessado em março de 2021]. Available from Internet: <https://github.com/Semantic-Org/Semantic-UI-React>.

SMYK, A. **The System Usability Scale and How It’s Used in UX**. Adobe, 2020. [Online; acessado em Novembro de 2021]. Available from Internet: <https://xd.adobe.com/ideas/process/user-testing/sus-system-usability-scale-ux/>.

SPRING. **Spring Boot**. GitHub, 2018. [Online; acessado em fevereiro de 2021]. Available from Internet: <https://github.com/spring-projects/spring-boot>.

APÊNDICE A — QUESTIONÁRIO DE USABILIDADE

Teste de usabilidade do sistema de Posts do Museu do Mar

Bem vindo!

Este formulário serve para coletar feedback da usabilidade do sistema de postagens do Museu do Mar, acesse http://musmar.inf.ufrgs.br/ em uma nova aba para começarmos com os testes.

A seguir temos alguns cenários de testes baseados nos usuários que estariam usando a aplicação:

Cenário 1: Criando um novo Post
Nesse cenário simulamos o usuário que estaria criando os posts para a ferramenta.
A partir da tela inicial, crie um novo post, adicione imagens, tags para busca, links, o que você achar que seriam informações úteis para consulta. (Obs.: evite colocar qualquer informação pessoal), salve e publique este post.

Cenário 2: Procurando e abrindo um Post
Nesse outro cenário simulamos o usuário que estaria interessado em pesquisar no banco de Posts por algum assunto específico, ou que apresentasse alguma característica.
Utilize o app de busca para buscar posts diversos, via tags ou datas, ordene por alguma coluna os resultados. Abra qualquer Post que você tenha como resultado da busca.

Agora que você se familiarizou com os recursos da ferramenta, por favor avalie as afirmações abaixo para que possamos medir a usabilidade do sistema.

Obrigado!

*Obrigatório

1. Eu acho que gostaria de usar esse sistema frequentemente *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	◯	◯	◯	◯	◯	Concordo Fortemente

2. Eu achei o sistema desnecessariamente complexo *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	◯	◯	◯	◯	◯	Concordo Fortemente

3. Eu achei o sistema fácil de usar *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	◯	◯	◯	◯	◯	Concordo Fortemente

4. Eu achei que precisaria de ajuda de uma pessoa técnica para conseguir usar esse sistema *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	◯	◯	◯	◯	◯	Concordo Fortemente

5. Eu achei que as várias funções desse sistema são bem integradas *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	◯	◯	◯	◯	◯	Concordo Fortemente

6. Eu achei que tem muitas inconsistências nesse sistema *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	⬭	⬭	⬭	⬭	⬭	Concordo Fortemente

7. Eu imagino que a maioria das pessoas aprenderiam a usar esse sistema rapidamente *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	⬭	⬭	⬭	⬭	⬭	Concordo Fortemente

8. Eu achei o sistema muito incômodo de usar *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	⬭	⬭	⬭	⬭	⬭	Concordo Fortemente

9. Eu me senti confiante usando o sistema *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	⬭	⬭	⬭	⬭	⬭	Concordo Fortemente

10. Eu preciso aprender muitas coisas antes que eu consiga seguir usando esse sistema *

Marcar apenas uma oval.

	1	2	3	4	5	
Discordo Fortemente	◯	◯	◯	◯	◯	Concordo Fortemente

Feedbacks adicionais

Agora que você respondeu quanto a usabilidade do sistema, fique a vontade para descrever um pouco mais sobre a experiência que você teve:

11. O que você mais gostou sobre o sistema? Algo te surpreendeu positivamente?

____
____
____
____
____

12. Encontrou algum problema? Algo específico que poderia ter sido feito de outra forma?

____
____
____
____
____

13. Alguma sugestão de funcionalidade? Algo que poderia ser incluído no futuro e que hoje não tem como opção?

Este conteúdo não foi criado nem aprovado pelo Google.

Google Formulários

APÊNDICE B — DOCUMENTAÇÃO DA API

Select a definition default

Mar de Informacao v1

[Base URL: localhost:8085/]
http://localhost:8085/v2/api-docs

findings Finding Report Controller

- GET `/api/v1/findings` Find all Findings
- POST `/api/v1/findings` Create Finding
- DELETE `/api/v1/findings` Delete all Findings
- GET `/api/v1/findings/{id}` Find Findings by id
- DELETE `/api/v1/findings/{id}` Delete Finding
- PATCH `/api/v1/findings/{id}` Update Finding

home-screen-controller Home Screen Controller

- GET `/` index
- HEAD `/` index
- POST `/` index
- PUT `/` index
- DELETE `/` index
- OPTIONS `/` index
- PATCH `/` index

images Image Controller

- GET `/api/v1/images` Find all Images
- DELETE `/api/v1/images/{id}` Delete Image
- GET `/api/v1/images/{imageId}` downloadImage
- PATCH `/api/v1/images/{imageId}` Update Image file
- GET `/api/v1/images/{imageId}/info` getImageInfo
- POST `/api/v1/images/postId={postId}` Create Image under Post

links Link Controller

GET **/api/v1/links** Find all Links

POST **/api/v1/links** Create Link

DELETE **/api/v1/links** Delete all Links

GET **/api/v1/links/{id}** Find Links by id

DELETE **/api/v1/links/{id}** Delete Link

PATCH **/api/v1/links/{id}** Update Link

POST **/api/v1/links/postId={postId}** Create Link under Post

posts Post Record Controller

GET **/api/v1/posts** Find all Posts

POST **/api/v1/posts** Create Post

DELETE **/api/v1/posts** Delete all Posts

GET **/api/v1/posts/{id}** Find Posts by id

DELETE **/api/v1/posts/{id}** Delete Post

PATCH **/api/v1/posts/{id}** Update Post

GET **/api/v1/posts/{id}/images** Get Post Images Info

GET **/api/v1/posts/{id}/links** Get Links from Post

GET **/api/v1/posts/{id}/tags** Get Post Tags

PATCH **/api/v1/posts/{id}/tags** Update Tag Relations

GET **/api/v1/posts/filteredBy** Get Filtered Posts

GET **/api/v1/posts/orderedBy** Get Ordered Posts

PATCH **/api/v1/posts/publish/{id}** Publish Post

reports Beach Report Controller

GET **/api/v1/reports** Find all Reports

POST **/api/v1/reports** Create Report

DELETE **/api/v1/reports** Delete all Reports

GET **/api/v1/reports/{id}** Find Reports by id

DELETE /api/v1/reports/{id} Delete Report

PATCH /api/v1/reports/{id} Update Report

tags Tag Controller

GET /api/v1/tags Find all Tags

POST /api/v1/tags Create Tag

DELETE /api/v1/tags Delete all Tags

GET /api/v1/tags/{id} Find Tags by id

PUT /api/v1/tags/{id} Update Tag

DELETE /api/v1/tags/{id} Delete Tag

windStatus Wind Status Controller

GET /api/v1/windStatus Find all Wind Status

POST /api/v1/windStatus Create Wind Status

DELETE /api/v1/windStatus Delete all Wind Status

GET /api/v1/windStatus/{id} Find Wind Status by id

DELETE /api/v1/windStatus/{id} Delete Wind Status

PATCH /api/v1/windStatus/{id} Update Wind Status

Models

```
BeachReport {
    fishCatched       string
    fishQuantity      number($double)
    id                integer($int64)
    rainVolume        number($double)
    temperature       number($double)
    waterQuality      string
    windStatus        WindStatus {...}
}

FindingReport {
    animalName        string
    animalSpecies     string
    garbageOrigin     string
    garbageQuantity   number($double)
    id                integer($int64)
}
```

```
Image {
    category            string
    content             string($byte)
    id                  integer($int64)
    name                string
}
```

InputStream

Iterable«BeachReport»

Iterable«FindingReport»

Iterable«Image»

Iterable«Link»

Iterable«PostRecord»

Iterable«Tag»

Iterable«WindStatus»

```
Link {
    id                  integer($int64)
    linkName            string
    tableLine           integer($int32)
    url                 string
}
```

```
PostRecord {
    beachReport         BeachReport {...}
    creationDate        string($date)
    description         string
    findingReport       FindingReport {...}
    id                  integer($int64)
    publicationDate     string($date)
    status              string
    title               string
}
```

```
PostRecordPartialUpdateDescription {
    description         string
    id                  integer($int64)
    title               string
}
```

```
Resource {
    description         string
    file                file
    filename            string
    inputStream         InputStream {...}
    open                boolean
    readable            boolean
    uri                 string($uri)
    url                 string($url)
}
```

```
Tag  {
   id                  integer($int64)
   tagName             string
}
```

```
WindStatus  {
   id                  integer($int64)
   windDirection       string
   windVelocity        number($double)
}
```

APÊNDICE C — BACKLOG: HISTÓRIAS DE USUÁRIO E ITENS DE BACKLOG

- US-01 Como um "Pescador", quero poder compartilhar informações úteis de meu dia-dia para auxiliar com o aprendizado da comunidade estudantil/acadêmica
  #6 by MateusCardoso was closed on Oct 24 · 7 tasks done
- BLI-01 Configuração Docker-Compose
  #7 by MateusCardoso was closed on Aug 28
- BLI-02 Banco de Dados
  #8 by MateusCardoso was closed on Aug 28
- BLI-03 API REST
  #9 by MateusCardoso was closed on Aug 28
- BLI-04 Posicionar componentes na tela de criação de Post
  #10 by MateusCardoso was closed on Aug 28
- BLI-05 Integração de componentes React com o BE na tela de criação de Post
  #11 by MateusCardoso was closed on Aug 28 · Sprint 1
- US-02 Como um "Curador", quero poder incrementar as informações já postadas, e que sirvam de referência acadêmica para o conhecimento empírico adicionado pelo "Pescador"
  #12 opened on May 30 by MateusCardoso · 4 of 6 tasks
- US-03 Como um "Estudante/Professor", quero poder pesquisar por Post publicados que ser refiram a assuntos específicos e ver em detalhes
  #13 by MateusCardoso was closed on Sep 18 · 5 tasks done · Sprint 2
- US-04 Como "Administrador", quero poder controlar quem pode criar novos posts na ferramenta
  #14 opened on May 30 by MateusCardoso · 3 tasks · Sprint 4
- BLI-06 Implementação BE para requests de busca com filtros · 1
  #16 by MateusCardoso was closed on Sep 18 · Sprint 2
- BLI-07 Layout do app de busca · 1
  #17 by MateusCardoso was closed on Sep 18 · Sprint 2
- BLI-08 Integração da busca com BE · 1
  #18 by MateusCardoso was closed on Sep 18 · Sprint 2
- BLI-09 Layout do app de visualização · 1
  #19 by MateusCardoso was closed on Sep 18 · Sprint 2
- BLI-10 Integração da visualização com BE · 1
  #20 by MateusCardoso was closed on Sep 18 · Sprint 2
- BLI-11 Implementação de edição de Post · 1
  #21 by MateusCardoso was closed on Sep 25 · Sprint 3
- BLI-12 Criação de URLs · 1
  #22 by MateusCardoso was closed on Sep 25 · Sprint 3
- BLI-13 Troca de status do Post · 1
  #23 by MateusCardoso was closed on Oct 17 · Sprint 4
- BLI-14 Catalogação de elementos
  #24 opened on Aug 28 by MateusCardoso
- BLI-15 Tela mais complexa para a edição
  #25 opened on Aug 28 by MateusCardoso · Sprint 4
- BLI-16 Integração de componentes React com BE · 1
  #26 by MateusCardoso was closed on Sep 25 · Sprint 3
- BLI-17 Configuração de usuários
  #27 opened on Aug 28 by MateusCardoso · Sprint 4
- BLI-18 Segurança do aplicativo/área reservada
  #28 opened on Aug 28 by MateusCardoso · Sprint 4
- BLI-19 Tela de Login para Admin/Curador/Pescador
  #29 opened on Aug 28 by MateusCardoso · Sprint 4
- BLI-20 Handling de imagens/videos · 1
  #30 by MateusCardoso was closed on Oct 24 · Sprint 4