Universidade Estadual do Norte Fluminense Darcy Ribeiro
Centro de Ciência e Tecnologia

Projeto POO Dev

Sistema para o Pescarte

Arthur Cézar Martins Ferreira Mól Matrícula: 20221100108

Professor: João Luiz de Almeida Filho

5 de novembro de 2025

Sumário

1 GitHub

Clique aqui para ser redirecionado ao repositório onde o projeto está.

2 Descrição do sistema

Linguagem para o back-end (e orientação a objetos) **Java** utilizando o framework **Springboot**.
Para o front-end será usado **React**.
Para o banco de dados será usado **Postgree**.

O sistema que será criado é uma plataforma interativa para o projeto PEA-Pescarte. O intuito desse sistema é criar uma plataforma onde os pescadores artesanais que participam do projeto possam logar, interagir com a plataforma e utilizar os serviços que serão fornecidos pela plataforma, como o de controle de gastos e lucro, além de informações gerais.

O sistema visa buscar fornecer serviços aos pescadores artesanais de modo a facilitar a vida deles, além de torná-los mais conectados.

As funcionalidades que estarão presentes no sistema são:

- Página **Home**: contendo as informações principais e mais relevantes do momento no site;
- Página **Login**: contendo a tela de login e criação de conta;
- Página **Serviços**: contendo serviços que podem vir a ser relevantes para os pescadores, como indicações de lojas como fábricas de gelo, lojas de artigos de pesca, informações sobre transporte, serviço de mecânico para os barcos;
- Página **Clima**: contendo informações sobre a previsão do tempo para o dia, se existe possibilidade de chuva, velocidade do vento, dentre outros;
- Página **Financeiro**: que faz um "controle"do financeiro dos pescadores, nessa página eles vão poder inserir o que pescaram e a aplicação vai fazer uma estimativa de por quanto eles deveriam vender os peixes baseado em informações estaduais e federais. Além disso, também uma opção de gerar uma planilha financeira com os gastos e lucros dos mesmos;
- Página **Sobre**: essa página vai conter as informações sobre o que é o projeto PEA-Pescarte.

2.1 O que é o projeto PEA-Pescarte

O Projeto PESCARTE tem como sua principal finalidade a criação de uma rede social regional integrada por pescadores artesanais e por seus familiares, buscando, por meio de processos educativos, promover, fortalecer e aperfeiçoar a sua organização comunitária e a sua qualificação profissional, bem como o seu envolvimento na construção participativa e na implementação de projetos de geração de trabalho e renda.

O Projeto de Educação Ambiental Pescarte (PEA Pescarte) trabalha junto às **comunidades de pesca artesanal** de Arraial do Cabo, Búzios, Cabo Frio, Campos, Carapebus, Macaé, Rio das Ostras, Quissamã, São Francisco de Itabapoana e São João da Barra. Desde 2014, atua juntamente aos pescadores artesanais e seus familiares, por meio de processos educativos, promovendo, fortalecendo e aperfeiçoando a organização comunitária e a sua qualificação profissional, bem como o seu envolvimento na construção participativa e na implementação de projetos de geração de trabalho e renda.

O processo educativo é realizado nos 10 municípios e se dá através de oficinas com temas diversos, entre eles: economia solidária, cooperativismo, políticas públicas, licenciamento ambiental, letramento digital e gestão participativa. O projeto promove, também, articulações entre os pescadores e pescadoras, com reuniões do Grupo de Trabalho, Grupo Gestor, Grupo de Acompanhamento de Obras e assembleias municipais.

3 Requisitos funcionais e não funcionais

3.1 Requisitos funcionais

Definem a **funcionalidade** que o sistema a ser desenvolvido deverá ter. Ele é um requisito relacionado com um tipo de comportamento produto de uma função do sistema.

Os requisitos funcionais deste projeto são:

- Cadastrar usuários;
- Realizar login;
- Transmitir dados;
- Exibir informações do usuário;
- Mostrar serviços;
- Gerar documentos;
- Consultar documentos.

Os requisitos funcionais se relacionam diretamente a processos[1] que o sistema deve executar. Ex: pesquisar, cadastrar, relatar, verificar, imprimir.

3.2 Requisitos não-funcionais

Indicam propriedades comportamentais que o sistema deve possuir.

Os requisitos não-funcionais deste sistema são:

- Poder acessar a aplicação usando qualquer navegador;
- Poder acessar a aplicação usando qualquer dispositivo: celular, tablet ou computador (responsividade);
- Aplicação com informações expostas de modo que pessoas com dificuldade de usar elementos da tecnologia possa acessá-lo sem problemas (acessibilidade com botões e textos grandes e intuitivos);
- Boa formatação, legibilidade e confiabilidade dos documentos gerados pelo sistema;
- Usuários poderem fazer login somente atrelados a uma Corporativa.

[1] processos = verbos

4 Diagrama de classes inicial

4.1 Diagrama de classes (UML)

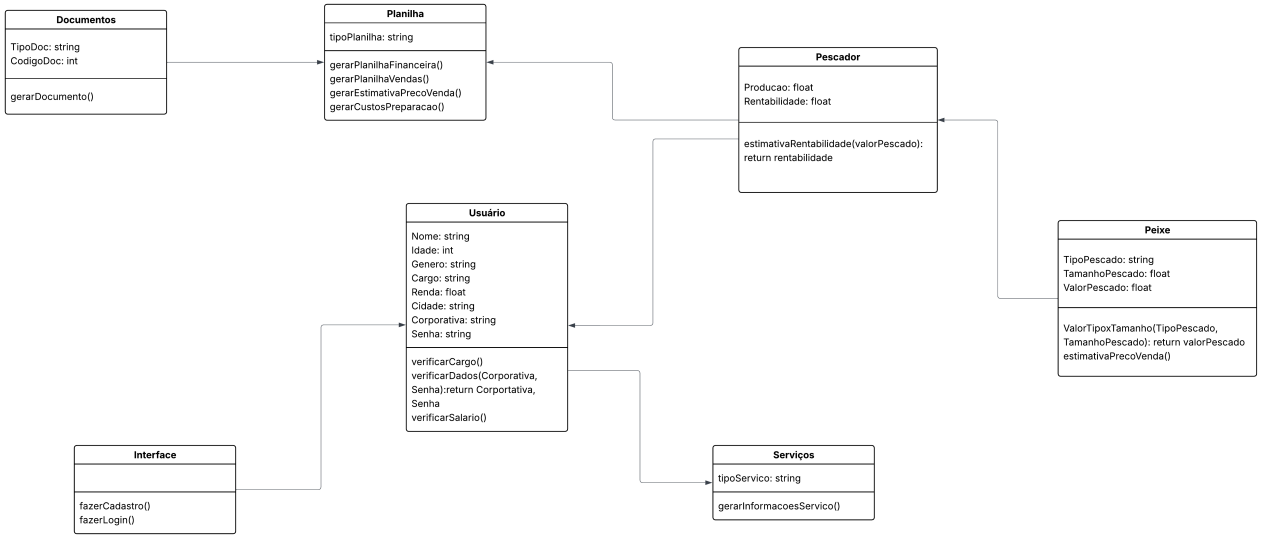

Figura 1 – Diagrama UML do projeto

4.2 Diagrama entidade relacionamento (ER)

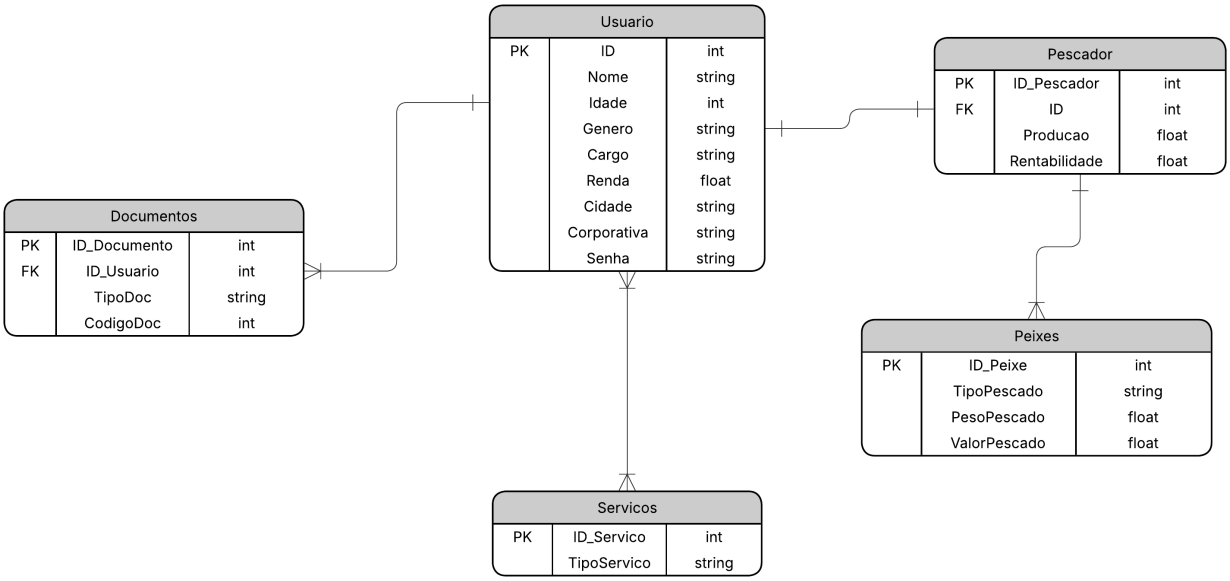

Figura 2 – Diagrama ER do projeto

5 Fluxo básico de funcionamento

5.1 Diagrama de casos de uso

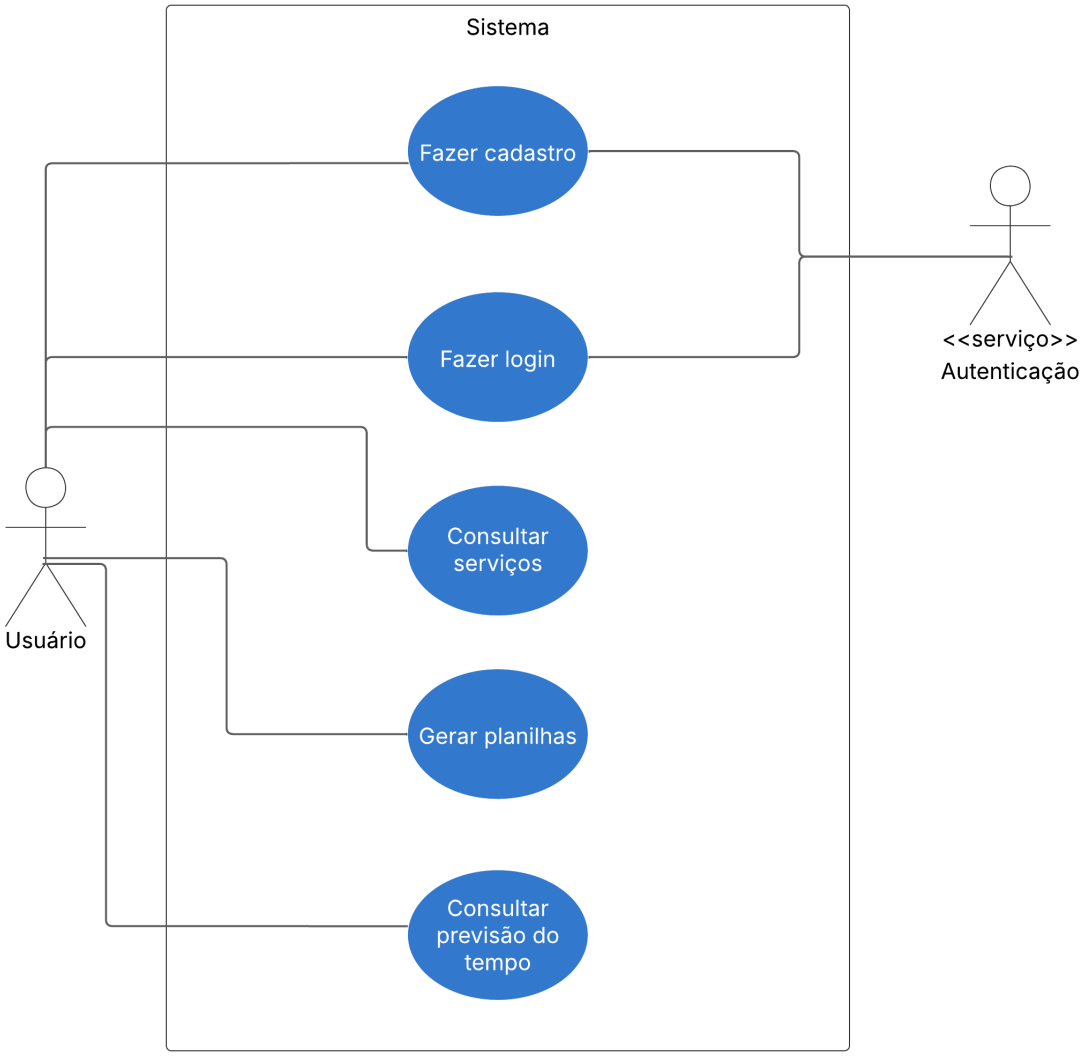

Figura 3 – Diagrama de Casos de Uso do projeto

6 Orçamento

6.1 Custo do desenvolvimento

Cobrando R$100,00 a hora. Por dia seriam utilizadas pelo menos 2 horas de trabalho, considerando que seriam trabalhados 5 dias da semana no mínimo 1 hora durante 3 meses, o custo final do desenvolvimento seria por volta de R$3000,00.

6.2 Custo de manutenção

Mensalmente seria cobrado o custo mensal da hospedagem do site, que seria cerca de R$300,00. Além do custo mensal de manutenção, que seria cerca de 2 horas de trabalho por mês exclusivamente para manutenção,

o que sairia cerca de R$200,00. Resultando um total de R$500,00 mensais.

6.3 Custo total

O preço total do projeto sairia pelo preço total R$3000,00 mais R$500,00 por mês que o cliente continuar contratando os serviços de manutenção.

7 Cronograma detalhado por semana

Semana	Datas	Atividade
1	15/09 – 19/09	Planejamento do modelo do front-end
2	22/09 – 26/09	Criação do esqueleto do front-end
3	29/09 – 03/10	Planejamento e criação do banco de dados
4	06/10 – 10/10	Planejamento do modelo do back-end
5	13/10 – 17/10	Criação do back-end
6	20/10 – 24/10	Aprimoramento do back-end
7	27/10 – 31/10	Aprimoramento do back-end
8	03/11 – 07/11	Aprimoramento do front-end
9	10/11 – 14/11	Aprimoramento do front-end
10	17/11 – 21/11	Conexão entre front-end, back-end e banco de dados
11	24/11 – 28/11	Revisão geral e melhorias pontuais
12	01/12 – 03/12	Entrega do projeto pronto

Tabela 1 – Cronograma de desenvolvimento do projeto

8 Realização do projeto

Nesta seção irei detalhar o cumprimento ou não cumprimento das tarefas propostas na seção 7.

8.1 Primeira semana

Objetivo: Planejamento do modelo front-end

Realizado: Foi feito um wireframe (figura 4) para representar uma base de como o site vai ser. Um wireframe é um esboço visual básico e de baixa fidelidade de um site ou aplicativo, que ilustra a estrutura e o layout de uma página ou tela.

Figura 4 – Wireframe do projeto

Além disso, também foi feito o que foi proposto para a Semana 2: foram criadas as pastas iniciais do projeto (Figura 5) e também foi criada uma estrutura inicial para o front-end, contendo uma NavBar funcional, que altera as páginas das seções do site corretamente e os conteúdos principais de cada página (Figura 6).

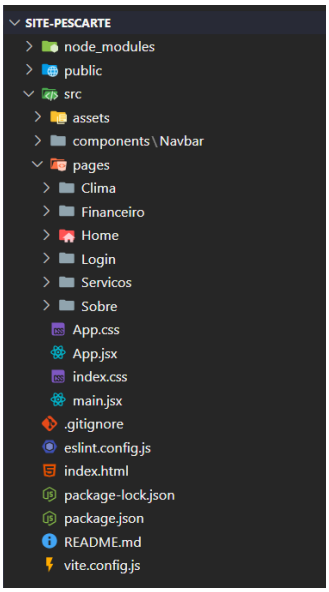

Figura 5 – Estrutura inicial das pastas do projeto

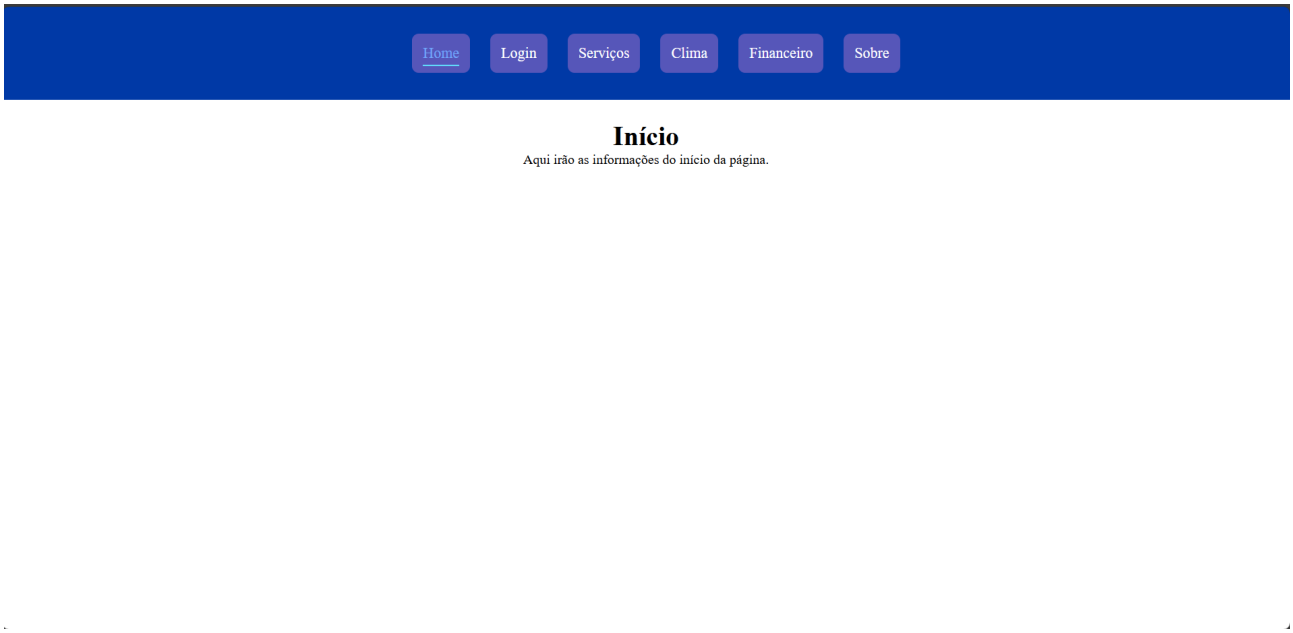

Figura 6 – Estrutura inicial do site

8.2 Segunda semana

Objetivo: Criação do esqueleto do front-end

Realizado: O que foi proposto na segunda semana já foi realizado na primeira. Na segunda semana nada foi feito em detrimento à semana de provas.

8.3 Terceira semana

Objetivo: Planejamento e criação do banco de dados

Realizado: Foi atualizado o diagrama entidade relacionamento (Figura 2). Além disso foi criada a estrutura base do banco de dados, com todas as tabelas e atributos (Figura 7).

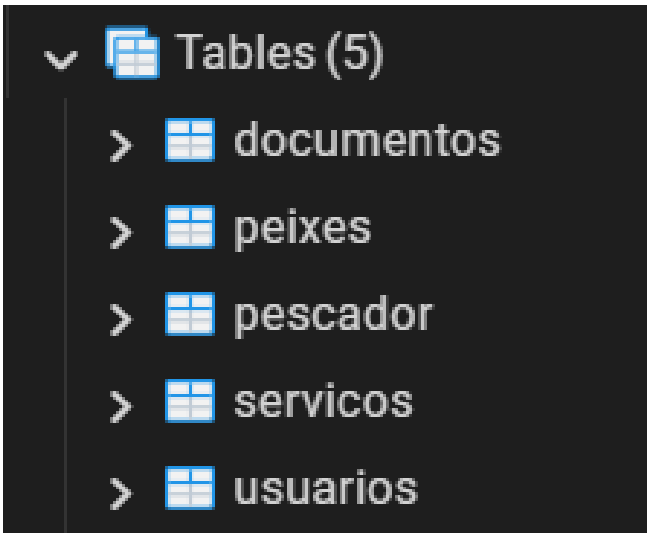

Figura 7 – Estrutura do Banco de Dados

8.4 Quarta semana

Objetivo: Planejamento do modelo do back-end

Realizado: Foi definida a estrutura do back-end, como é possível ver na figura 8.

```
src/main/java/com/pescarte/
├── config/
│   ├── SecurityConfig.java  // Configuração do Spring Security (JWT, CORS)
│   └── CorsConfig.java      // Configuração global do CORS (Conexão com React)
├── controller/
│   ├── AuthController.java      // Endpoint para /api/auth/login
│   ├── ClimaController.java     // Endpoint para /api/clima
│   └── DocumentoController.java // Endpoint para /api/documentos/gerar
├── model/
│   └── User.java              // Entidade JPA para o usuário
├── repository/
│   └── UserRepository.java      // Interface Spring Data JPA
├── service/
│   ├── ClimaService.java        // Lógica para buscar na API de clima externa
│   └── DocumentoService.java    // Lógica para gerar documentos
└── security/
    └── JwtTokenProvider.java  // (No caso de usar JWT para autenticação)
```

Figura 8 – Estrutura do Back-end

Além disso também foram criadas tuplas para começar a popular as tabelas do banco de dados.

O desenvolvimento em SpringBoot segue a arquitetura MVC (Model-View-Controller).

A camada **Model** (Modelo) é responsável por representar a lógica de negócios, os dados da aplicação e a interação com o banco de dados.

A **Repository** serve como uma interface para abstrair a camada de acesso a dados, facilitando a interação com o banco de dados sem que a lógica de negócio tenha que se preocupar com os detalhes da persistência, ele

fornece métodos para operações de CRUD (Criar, Ler, Atualizar, Excluir) e consultas, que são implementados automaticamente pelo Spring Data JPA.

A **Service** é uma camada para implementar a lógica de negócios da aplicação, atuando como intermediário entre a camada de apresentação (Controller) e a camada de persistência (Repository). Ele encapsula as regras de negócio, coordena as operações e abstrai a complexidade do acesso ao banco de dados, promovendo uma aplicação mais modular, organizada e fácil de testar.

E a **Controller** serve como o ponto de entrada para as requisições de uma aplicação, processando as solicitações HTTP e retornando respostas apropriadas para o cliente. Ela define os endpoints (as rotas da API), como GET, POST, PUT e DELETE, que expõem a funcionalidade do backend para o mundo exterior. Ao receber uma requisição, o controller chama a lógica de negócios na camada de serviço e, em seguida, converte os dados de entrada e saída para formatos como JSON ou XML.

8.5 Quinta semana

ATRASO DEVIDO À SEMANA DE PROVAS

8.6 Sexta semana

ATRASO DEVIDO À SEMANA DE PROVAS

8.7 Sétima semana

Objetivo: Aprimoramento do back-end

Realizado:

Foi feita a criação do projeto utilizando as seguintes dependências:

- Spring Web: Para criar os controladores REST (endpoints da API);
- Spring Security: Para cuidar da página de Login e proteger endpoints;
- Spring Data JPA: Para se comunicar com o banco de dados (ex: para salvar usuários);
- PostgreSQL Driver: Driver para poder conectar o banco PostgreSQL;
- Lombok: Facilita a escrita de classes (reduz boilerplate).

Foi configurada a conexão do back-end com o banco de dados, como podemos ver no código 1

```
// Configuracao do PostgreSQL
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.username=postgres
spring.datasource.password=190603
spring.datasource.url=jdbc:postgresql://localhost:5432/banco-pescarte
spring.jpa.generate-ddl=true

// Configuracao do Hibernate (JPA)
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true

// Define o schema padrao do banco de dados para o Hibernate
spring.jpa.properties.hibernate.default_schema=pescarte
```

Listing 1 – Conexão banco de dados com back-end

Criando as funcionalidades

Clima

A primeira funcionalidade adicionada foi a de Clima, visto que ela é a mais fácil de configurar pois é "externa"à aplicação, visto que ela vai buscar informações de uma API.

A API que utilizada é a OpenWeatherMap.

Primeiro de tudo eu configurei a key da API, como podemos ver no código 2.

```
1
2  // Chave do OpenWeatherMap
3  weather.api.key=${WEATHER_API_KEY}
4  // O Spring vai procurar pela variavel de ambiente chamada 'WEATHER_API_KEY' (
       seguranca da key)
5
6  // URL base da API
7  weather.api.url=https://api.openweathermap.org/data/2.5/weather
```

Listing 2 – Configuração key da API

OBS: para configurar a chave eu utilizei a variável de ambiente "WEATHER_API_KEY", porque na hora de subir o código para o GitHub ele vai subir a variável de ambiente e não a chave (que é privada). Além disso, essa "URL base da API"é a estrutura base de consulta, quando for para uma cidade específica ela ira mudar, mas baseada nela. O formato básico da URL é: <https://api.openweathermap.org/data/2.5/weather?q={CIDADE}&appid={API_KEY}&lang=pt_br&units=metric>

Após configurar a chave eu criei um Service (que contém as regras de negócio/funcionalidades) e um Controller (que é onde ficam os controladores REST) para a função clima, sendo eles ClimaService e ClimaController, respectivamente.

Configurando o ClimaService

Nele eu usei o RestTemplate, que vai chamar a API do OpenWeatherMap. O RestTemplate simplifica o processo de fazer requisições HTTP para API's externas.

Criei um AppConfig para configurar o RestTemplate que vai ser um Bean, para que ele possa ser reutilizado:

```
1      @Bean
2      public RestTemplate restTemplate() {
3          return new RestTemplate();
4      }
```

O código do ClimaService é:

```
1  @Service
2  public class ClimaService {
3      // Injetando o Bean "RestTemplate"
4      private final RestTemplate restTemplate;
5
6      // Pega os valores do application.properties
7      @Value("${weather.api.key}")
8      private String apiKey;
9
10     @Value("${weather.api.url}")
11     private String apiUrl;
```

```java

    // Injecao de dependencia (via construtor)
    public ClimaService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    // Buscando o clima de uma cidade
    // Por enquanto vai ficar estatica a cidade, depois vou passar a cidade como parametro
    // OBS: DTO = Data Transfer Object
    public ClimaDTO buscarClima() {
        String cidade = "London";

        // O UriComponentsBuilder ajuda a montar a URL de forma segura
        // Aqui esta a estrutura da URL da API
        String url = UriComponentsBuilder.fromHttpUrl(apiUrl)
                .queryParam("q", cidade)
                .queryParam("appid", apiKey)
                .queryParam("units", "metric") // Para temperatura em Celsius
                .queryParam("lang", "pt_br")   // Para descricao em portugues
                .toUriString();

        try {
            // Faz a chamada GET e pede o resultado como JsonNode
            JsonNode resposta = restTemplate.getForObject(url, JsonNode.class);

            // Cria um novo objeto para clima
            ClimaDTO clima =  new ClimaDTO();

            // Navega na arvore do JSON para pegar os dados que queremos
            clima.setCidade(resposta.get("name").asText());
            clima.setPais(resposta.get("sys").get("country").asText());
            clima.setTemperatura(resposta.get("main").get("temp").asDouble());
            clima.setUmidade(resposta.get("main").get("humidity").asInt());
            clima.setPressao(resposta.get("main").get("pressure").asInt());
            clima.setDescricao(resposta.get("weather").get(0).get("description").asText());
            clima.setVelocidadeVento(resposta.get("wind").get("speed").asDouble());
            clima.setDirecaoVento(resposta.get("wind").get("deg").asDouble());
            clima.setNebulosidade(resposta.get("clouds").get("all").asInt());

            // Retorna o DTO (clima)
            return clima;

        } catch (Exception e) {     // COLOCAR MAIS TRATAMENTO DE ERROS DEPOIS
            // Tratamento de erros simples
            e.printStackTrace();
            throw new RuntimeException("Erro ao buscar dados do clima: " + e.getMessage());
        }
    }
```

```
}
```

Como podemos ver, nele eu injeto o RestTemplate e pego a chave da API e a URL base dela. Depois crio o construtor e crio um método para buscar o clima. Por hora a cidade está estática apenas para a realização de testes, e dentro do método eu uso o UriComponentsBuilder, que é uma classe pré-definida do Springboot para a construção dinâmica da URL. Depois eu faço a chamada GET e navego na árvore JSON (que é a resposta que a API me dá) e pego os valores dela e retorno para o usuário.

Além disso também fiz um tratamento de erros simples, por enquanto.

Configurando o ClimaController

O código do ClimaController é:

```
package com.faculdade.pescarte.controller;

import com.faculdade.pescarte.dto.ClimaDTO;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.faculdade.pescarte.service.ClimaService;

@RestController
@RequestMapping("/api/clima")
@CrossOrigin(origins = "*")              // Serve para permitir conexao de dominio
    cruzado (possibilita conexao front e back)
public class ClimaController {
    private final ClimaService climaService;

    // Construtor
    public ClimaController(ClimaService climaService) {
        this.climaService = climaService;
    }

    // GET
    @GetMapping
    public ClimaDTO getClima() {
        // O Spring vai converter o ClimaDTO para JSON automaticamente
        return climaService.buscarClima();
    }
}
```

Nele eu defino o meu endpoint (que é onde essa funcionalidade vai estar na aplicação), que no caso é "api/clima".

Configurando a Segurança (SecurityConfig)

A configuração de segurança (código 3) por enquanto não está completa, ela está feita com o objetivo de permitir testes de modo mais simples. A configuração final será diferente.

Nessa classe é configurada as questões de segurança da aplicação, como a permissão de acesso à endpoints, quais requisições são permitidas e onde o front-end pode se conectar.

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                // Habilita o CORS (que vai precisar para conectar com o React)
                .cors(cors -> cors.configurationSource(corsConfigurationSource()))

                // Desabilita CSRF (Cross Site Request Forgery)
                .csrf(csrf -> csrf.disable())

                // Define a politica de sessao como STATELESS (API REST nao guarda estado)
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

                // Configura as regras de autorizacao
                .authorizeHttpRequests(authorize -> authorize
                    // Permite acesso publico ao endpoint /api/clima
                    //.requestMatchers("/api/clima/**").permitAll()

                    // Permite acesso publico ao endpoint de login
                    //.requestMatchers("/api/auth/**").permitAll()

                    // Para QUALQUER outra coisa, exige autenticacao
                    //.anyRequest().authenticated()

                        // LIBERANDO TODAS AS PERMISSOES PARA TESTAR REQUESTS (DEPOIS SERA IMPLEMENTADA A SEGURANCA)
                        .anyRequest().permitAll()
                );

    return http.build();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration corsConfiguration = new CorsConfiguration();

        // Porta que o React esta rodando
        corsConfiguration.setAllowedOrigins(Arrays.asList("http://localhost:3000"));

        // Requisicoes que ele pode realizar
        corsConfiguration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE", "OPTIONS"));
        corsConfiguration.setAllowedHeaders(Arrays.asList("Authorization", "Content-Type"));
```

```
        corsConfiguration.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource
            ();
        source.registerCorsConfiguration("/**", corsConfiguration);     // aplica a
            todos os endpoints
        return source;
    }
}
```

Listing 3 – Configuração inicial de segurança

Peixes

A próxima funcionalidade adicionada foi a de cadastrar Peixes, para testar se as requisições estão funcionando e se a conexão com o banco de dados está correta.

O software utilizado para realizar as requisições é o **Postman**.

Primeiro eu criei o Model para Peixes:

```
package com.faculdade.pescarte.model;

import jakarta.persistence.*;
import lombok.Data;

@Entity
@Table(name = "peixes")
@Data
public class Peixes {
    // Atributos (colunas do banco de dados)
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_peixe")          // A coluna do banco se chama exatamente "
        id_peixe"
    private Long id;

    // Mapeando as colunas do banco de dados no codigo usando @Column
    @Column(name = "tipopescado")       // A coluna do banco se chama exatamente "
        tipopescado"
    String tipoPescado;

    @Column(name = "pesopescado")       // A coluna do banco se chama exatamente "
        pesopescado"
    Float pesoPescado;

    @Column(name = "valorpescado")      // A coluna do banco se chama exatamente "
        valorpescado"
    Float valorPescado;
}
```

Depois eu criei o Repository para Peixes:

```
package com.faculdade.pescarte.repository;
```

```java

import com.faculdade.pescarte.model.Peixes;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface PeixesRepository extends JpaRepository<Peixes, Long> {
    // Spring JPA cria os metodos
}
```

JpaRepository é uma interface do Spring Data JPA que fornece uma maneira simplificada de interagir com bancos de dados, oferecendo métodos para operações comuns de CRUD (Create, Read, Update, Delete) sem a necessidade de escrever consultas SQL manualmente.

Depois eu criei o Service para Peixes:

```java
package com.faculdade.pescarte.service;

import com.faculdade.pescarte.model.Peixes;
import com.faculdade.pescarte.repository.PeixesRepository;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class PeixesService {

    // Criando um repositorio
    private final PeixesRepository peixesRepository;

    // Construtor da classe
    public PeixesService(PeixesRepository peixesRepository) {
        this.peixesRepository = peixesRepository;
    }

    // Metodos
    // Metodo para mostrar todos os peixes
    public List<Peixes> listarPeixes(){
        return peixesRepository.findAll();
    }

    // OPERACOES REST

    // Metodo para buscar um peixe pelo id
    public Optional<Peixes> findById(Long id) {
        return peixesRepository.findById(id);
    }

    // Metodo para salvar/cadastrar um peixe
    public Peixes salvarPeixes(Peixes peixes) {
        return peixesRepository.save(peixes);
    }
```

```
38
39      // Metodo para deletar/remover um peixe pelo id
40      public void excluirPeixes(Long id) {
41          peixesRepository.deleteById(id);
42      }
43  }
```

E por fim criei o Controller para Peixes:

```
1   package com.faculdade.pescarte.controller;
2
3   import com.faculdade.pescarte.model.Peixes;
4   import com.faculdade.pescarte.service.PeixesService;
5   import org.springframework.http.ResponseEntity;
6   import org.springframework.web.bind.annotation.*;
7
8   import java.util.List;
9
10  @RestController
11  @RequestMapping("/api/peixes")
12  public class PeixesController {
13      private final PeixesService peixesService;
14
15      // Construtor
16      public PeixesController(PeixesService peixesService) {
17          this.peixesService = peixesService;
18      }
19
20      // Metodos
21      // Metodo GET para ver a lista de peixes
22      @GetMapping // sem endpoint. Se nao passar nada aqui ele chama o metodo por
            padrao quando /api/peixes for acessada
23      public List<Peixes> listarPeixes() {
24          return peixesService.listarPeixes();
25      }
26
27      // Metodo GET para passar o ID e me retornar o peixe
28      @GetMapping("/{id}")
29      public ResponseEntity<Peixes> buscarPeixesPorId(@PathVariable Long id) {
30          return peixesService.findById(id)
31                  .map(ResponseEntity::ok)
32                  .orElse(ResponseEntity.notFound().build());
33      }
34
35      // Metodo POST para inserir no banco de dados
36      @PostMapping
37      public Peixes salvarPeixes (@RequestBody Peixes peixes) {
38          return peixesService.salvarPeixes(peixes);
39      }
40
41      // Metodo DELETE para deletar no banco de dados
42      @DeleteMapping("{id}")          // deletar por id
```

```
43      public ResponseEntity<Void> excluirPeixes(@PathVariable Long id) {
44          peixesService.excluirPeixes(id);
45          return ResponseEntity.noContent().build();
46      }
47  }
```

Após todo esse processo o endpoint "/api/peixes"está funcionando.