SOFTWARE PARA ADIÇÃO DE MORADORES EM UM RESIDENCIAL

Amanda Luiza Hoffmann Pereira; RA:12523118290, Ana Beatriz Santos Prates de Souza; RA:12523179390, Rafael Elias da Gama; RA:12523139095, Ricardo Henrique de Souza Brito; RA:12523123035 e Tiago Gonçalves de Almeida; RA:12523117709

Análise e Desenvolvimento de Software e Ciência da Computação, Paulista, Universidade Anhembi Morumbi
e-mail: amandahoffmann96@gmail.com, anabeatrizprates18@gmail.com, rafael.elias.da.gama@gmail.com, ricardohenri.brito@gmail.com e tiagoalmeida0812@gmail.com

Resumo: Java é uma linguagem de programação popular e poderosa, amplamente utilizada no desenvolvimento de aplicativos, enquanto o Spring Boot é um framework que facilita a criação de aplicativos web robustos e escaláveis. No contexto de criação de listas em Java, a classe ArrayList é comumente usada para adicionar, remover e pesquisar elementos. Além disso, a combinação do Java com JavaScript e React pode proporcionar uma experiência de desenvolvimento mais interativa, onde o JavaScript, sendo uma linguagem de programação executada no lado do cliente, permite a manipulação de elementos da página web e a interação com o servidor, e o React, um biblioteca JavaScript, é amplamente utilizado para construir interfaces de usuário modernas e reativas. Com essa combinação, os desenvolvedores podem criar aplicativos web poderosos e altamente interativos, utilizando Java no lado do servidor para processar solicitações e fornecer dados para o cliente, e JavaScript e React para criar interfaces de usuário responsivas e dinâmicas.
Palavras-chave: Java, Spring Boot, criação de listas, JavaScript, React.

Abstract: *Java is a popular and powerful programming language widely used in application development, while Spring Boot is a framework that facilitates the creation of robust and scalable web applications. In the context of creating lists in Java, the ArrayList class is commonly used to add, remove, and search for elements. Additionally, the combination of Java with JavaScript and React can provide a more interactive development experience, where JavaScript, as a client-side programming language, allows for manipulation of web page elements and interaction with the server, and React, a JavaScript library, is widely used to build modern and reactive user interfaces. With this combination, developers can create powerful and highly interactive web applications, using Java on the server-side to process requests and provide data to the client, and JavaScript and React to create responsive and dynamic user interfaces.*

Keywords: *Java, Spring Boot, list creation, JavaScript, React.*

Introdução

A tecnologia tem desempenhado um papel fundamental no avanço das soluções aplicadas à gestão de residenciais e condomínios, proporcionando maior eficiência e praticidade nas tarefas cotidianas. Nesse contexto, uma aplicação web surge como uma ferramenta indispensável, disponibilizando uma interface amigável e intuitiva para que os usuários possam gerenciar as informações dos moradores de um residencial de forma ágil e eficiente.

O objetivo principal desta aplicação é permitir que o usuário tenha controle total sobre os dados dos moradores, oferecendo recursos como adicionar, remover e filtrar informações com base no ID dos moradores. A interface da aplicação é projetada de forma a tornar essas ações simples e acessíveis, proporcionando uma experiência fluida e amigável.

No entanto, é importante ressaltar que essa aplicação não utiliza nenhum banco de dados para armazenar os dados dos moradores. Em vez disso, o back-end do sistema utiliza uma abordagem stateless, o que significa que ele não mantém nenhum estado entre as requisições. Todas as informações dos moradores são armazenadas temporariamente em memória durante a execução do programa, mas não são persistidas em um banco de dados.

Em resumo, a aplicação em questão permite ao usuário gerenciar os moradores de um residencial de forma prática e interativa. Embora não haja persistência de dados em um banco de dados, a interface proporciona uma visualização atualizada e filtrável dos moradores, utilizando tecnologias web modernas e o biblioteca React.

Materiais e métodos

Para atingir os objetivos propostos, e criar uma aplicação intuitiva foram utilizadas as seguintes

tecnologias: Java; Spring Boot; Eclipse; JavaScript; React Js; CSS; Yarn; Visual Studio Code.

O servidor, ou back-end, que é responsável por criar um novo residente e também a tabela, foi codificado na linguagem de programação Java, e para que fosse possível a criação de uma API Rest, tornou-se necessário a utilização do Spring Boot, que é um framework responsável por facilitar a comunicação com o Cliente ou front-end.

Enquanto no Cliente, ou front-end, optou-se por utilizar a linguagem de programação JavaScript, juntamente com React, que é uma biblioteca JavaScript utilizada para construção de interfaces de usuário interativas e reativas.

Resultados

A Figura 1 representa a classe "Resident", que pertence ao lado do servidor, código escrito em Java, (também disponível em: https://github.com/amandahp/cond-server/blob/main/src/main/java/condominium/apicond/methods/Resident.java), que representa um residente no sistema de gerenciamento de condomínios, armazenando suas informações pessoais. Ela possui métodos Set para definir e Get para obter os dados do residente. Ademais, a classe possui o seguintes atributos:

- id: identificador único do residente;
- count: contador estático utilizado para atribuir automaticamente um valor único ao ID de cada residente.
- name: nome do residente.
- phone: número de telefone do residente.
- appartament: número do apartamento do residente.
- parkingNumber: número da vaga de estacionamento do residente.
- emergencyContact: número de contato de emergência do residente.

```java
package condominium.apicond.methods;
public class Resident {
        private int id;
        private static int count = 0;
        private String name;
        private String phone;
        private Integer apartment;
        private String parkingNumber;
        private String emergencyContact;

        public Resident() {
                this.setId(++count);
        }

  public void setData(String name, String phone, int apartment, String parkingNumber, String
emergencyContact){
      this.name = name;
      this.phone = phone;
      this.apartment = apartment;
      this.parkingNumber = parkingNumber;
      this.emergencyContact = emergencyContact;
  }

        public String[] getData() {
                String[] data = new String[5];

                data[0] = this.name;
                data[1] = this.phone;
                data[2] = Integer.toString(this.apartment);
                data[3] = this.parkingNumber;
                data[4] = this.emergencyContact;

                return data;
        }
        public int getId() {
                return this.id;
        }
        public void setId(int id) {
                this.id = id;
        }

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getPhone() {
                return this.phone;
        }
        public void setPhone(String phone) {
                this.phone = phone;
        }
        public Integer getAppartament() {
                return this.apartment;
        }
        public void setAppartament(int apartment) {
                this.apartment = apartment;
        }
        public String getParkingNumber() {
                return this.parkingNumber;
        }
        public void setParkingNumber(String parkingNumber) {
                this.parkingNumber = parkingNumber;
        }
        public String getEmergencyContact() {
                return this.emergencyContact;
        }
        public void setEmergencyContact(String emergencyContact) {
                this.emergencyContact = emergencyContact;
        }
}
```

Figura 1: Classe "Resident", responsável por criar um residente no sistema de condomínios.

A Figura 2 (também disponível em: https://github.com/amandahp/cond-server/blob/main/src/main/java/condominium/apicond/methods/DynamicArray.java) representa a classe "DynamicArray", a qual é responsável por criar uma lista dinâmico de objetos da classe "Resident", com métodos para adicionar, remover, atualizar e filtrar elementos da lista. Possuindo um mecanismo de redimensionamento automático quando a lista atinge sua capacidade máxima. A classe possui os seguintes atributos:

- list: lista que armazena os objetos " Resident";
- maxSize: tamanho máximo do lista;
- length: tamanho atual do lista;

A classe também possui os seguintes métodos:

- DynamicArray(): construtor da classe que inicializa o lista com tamanho 1 e define a quantidade de elementos como 0.

- add(Resident item): método para adicionar um objeto "Resident" à lista. Caso a lista esteja cheia, ele é redimensionado para o dobro do tamanho atual e os elementos existentes são copiados para a nova lista.

- removeItem(int id): método para remover um objeto “Resident” da lista com base em seu ID. Utiliza um método da classe `Utils` para encontrar o índice do objeto a ser removido, desloca os elementos subsequentes para preencher o espaço vazio e decrementa a quantidade de elementos.

- updateItem(int id, Resident updatedResident): método para atualizar um objeto “Resident” na lista com base em seu ID. Utiliza um método da classe “Utils” para encontrar o índice do objeto a ser atualizado e modifica os atributos do objeto com base nos valores atualizados passados como parâmetro.

- filterItem(int id): método para filtrar um objeto “Resident” na lista com base em seu ID. Utiliza um método da classe “Utils” para encontrar o índice do objeto desejado e retorna o objeto encontrado.

- printList(): método que retorna o lista de objetos “Resident”.

- isFull(): método que verifica se a lista está cheio, ou seja, se a quantidade de elementos é igual ao tamanho máximo.

```java
package condominium.apicond.methods;
import condominium.apicond.utils.Utils;
public class DynamicArray {
        private Resident[] list;
        private int maxSize;
        private int length;
        public DynamicArray() {
                this.maxSize = 1;
                this.list = new Resident[maxSize];
                this.length = 0;
        }
        public void add(Resident item) {
                if (isFull()) {
                        maxSize = 2 * maxSize;
                        Resident[] tempList = new Resident[2 * maxSize];
                        for (int i = 0; i < length; i++) {
                                tempList[i] = list[i];
                        }
                        list = tempList;
                }
                list[length] = item;
                length++;
        }
        public void removeItem(int id) {
                Utils Utils = new Utils();
                int indexToRemove = Utils.findIndex(list, id);

                if (indexToRemove != -1) {
                        for (int i = indexToRemove; i < this.length - 1; i++) {
                                list[i] = list[i + 1];
                        }
                        list[this.length - 1] = null;
                        this.length--;
                }

        }
        public void updateItem(int id,  Resident updatedResident) {
                Utils Utils = new Utils();
                System.out.println(this.printList());
                int indexToUpdate = Utils.findIndex(this.printList(), id);

      if (updatedResident.getName() != null) {
         list[indexToUpdate].setName(updatedResident.getName());
      }
      if (updatedResident.getPhone() != null) {
        list[indexToUpdate].setPhone(updatedResident.getPhone());
      }
      if (updatedResident.getAppartament() != null) {
        list[indexToUpdate].setAppartament(updatedResident.getAppartament());
      }
      if (updatedResident.getParkingNumber() != null) {
        list[indexToUpdate].setParkingNumber(updatedResident.getParkingNumber());
      }
      if (updatedResident.getEmergencyContact() != null) {
        list[indexToUpdate].setEmergencyContact(updatedResident.getEmergencyContact());
      }
        }
        public Resident filterItem(int id) {
                Utils Utils = new Utils();
                int indexToSearch =  Utils.findIndex(list, id);
                if(indexToSearch == -1) {
                        return null;
                }
                return list[indexToSearch];
        };
        public Resident[] printList() {
                return list;
        }
        boolean isFull() {
                return length == maxSize;
        }
}
```

Figura 2: Classe “DynamicArray” responsável por criar um array dinâmico.

A Figura 3, representa a classe “ResidentController”, (também disponível em: https://github.com/amandahp/cond-server/blob/main/src/main/java/condominium/apicond/controller/ResidentController.java), define os endpoints e a lógica utilizada para realiza as operações CRUD dos objetos “Resident”, em uma API RESTful. Ela utiliza o objeto “DynamicArray” para armazenar e manipular os objetos “Resident”. A classe possui o seguintes métodos:

- register: método POST que recebe os dados de um “ResidentModel” no corpo da requisição, cria um novo objeto “Resident”, configura os atributos com base nos dados recebidos e adiciona o objeto ao array dinâmico “list”.

- list: método GET que retorna a lista completa de objetos “Resident” presentes no lista “list”.

- item: método GET que recebe o ID de um “Resident” como parâmetro na URL e retorna o objeto “Resident” correspondente com base no ID.

- update: método PUT que recebe o ID de um “Resident” como parâmetro na URL e um objeto “Resident” atualizado no corpo da requisição. Atualiza o objeto “Resident” na lista “list” com base no ID e nos dados atualizados.

- delete: método DELETE que recebe o ID de um “Resident” como parâmetro na URL. Remove o objeto “Resident” correspondente da lista “list”.

Além disso, a classe possui as anotações “@RestController” e “@RequestMapping”, indicando que ela é responsável por tratar as requisições e especificando o mapeamento da URL base para as rotas da API.

Também é utilizada a anotação “@CrossOrigin(origins = "*")”, que permite o acesso de qualquer origem para as requisições feitas a partir dessa classe.

```java
package condominium.apicond.controller;

import org.springframework.http.HttpStatus;

@RestController
@RequestMapping

public class ResidentController {
    DynamicArray list = new DynamicArray();

    @CrossOrigin(origins = "*")
    @PostMapping("/resident")
    @ResponseStatus(code = HttpStatus.CREATED)
    public void register(@RequestBody ResidentModel data) {
        Resident resident = new Resident();
        resident.setData(data.name(), data.phone(), data.appartament(), data.parkingNumber(), data.emergencyContact());
        list.add(resident);
    }

    @GetMapping("/resident")
    @ResponseStatus(code = HttpStatus.OK)
    public Object list() {
        return list.printList();
    }

    @GetMapping("/resident/{id}")
    @ResponseStatus(code = HttpStatus.OK)
    public Object item(@PathVariable int id) {
        return list.filterItem(id);
    }

    @CrossOrigin(origins = "*")
    @PutMapping("/resident/{id}")
    @ResponseStatus(code = HttpStatus.ACCEPTED)
    public String update(@RequestBody Resident data, @PathVariable int id) {
        list.updateItem(id,data);
        return "Alterado com sucesso";
    }

    @CrossOrigin(origins = "*")
    @DeleteMapping("/resident/{id}")
    @ResponseStatus(code = HttpStatus.OK)
    public void delete(@PathVariable int id) {
        list.removeItem(id);
    }
}
```

Figura 3: Classe “ResidentController” realiza as operações CRUD.

A Figura 4 representa a implementação da página inicial que busca, exibe e interage com dados de residentes usando chamadas de API e componentes reutilizáveis, foi criada utilizando Javascript e a biblioteca React, (esse código também está disponível em: https://github.com/amandahp/cond-client/blob/main/src/pages/HomePage/HomePage.js).

Dessa maneira pode-se resumir as principais partes do código da seguinte maneira:

- As importações iniciais estão trazendo os hooks e componentes necessários do React, bem como funções e bibliotecas auxiliares.

- A função “HomePage” é definida como um componente de função do React.

- São utilizados diversos hooks, como “useState”, “useEffect” e “useCallback”, para gerenciar o estado e o ciclo de vida do componente.

- O componente faz uso de funções assíncronas, como “listApi”, “listOneItemApi”, “registerApi”, “updateApi” e “deleteApi”, para realizar chamadas de API relacionadas ao CRUD de residentes.

- O estado do componente é gerenciado pelos hooks “useState”, como “data”, “error” e “users”.

- A função “fetchList” é responsável por buscar a lista de residentes da API e atualizar o estado correspondente.

- A função “fetchResident” é utilizada para buscar informações de um residente específico com base em um ID fornecido.

- O componente renderiza um cabeçalho e uma tabela usando o componente “Header” e “Table”.

- Existem funções como “openModal”, “closeModal”, “addNewRow”, “handleDelete” e ”handleEdit” que lidam com ações do usuário, como abrir e fechar um modal, adicionar uma nova linha, excluir e editar residentes na tabela.

- O componente retorna a estrutura JSX com o cabeçalho e a tabela renderizados.

```
import { useState, useEffect, useCallback } from "react";
import { Header, Table } from "../../components"
import { listApi, listOneItemApi, registerApi, updateApi,deleteApi } from
"../../services/condominiumApi";
import {isEmpty, omit} from "lodash";


export const HomePage = () => {
 const [data, setData] = useState([]);
 const [error, setError] = useState("");
 const [users, setUsers] = useState([]);

 const fetchList = useCallback(async() => {
   try{
     const response = await listApi()
     if(response.length) {
       const responseFiltered = response.filter((r) => !isEmpty(r))
       const responseFormatted = responseFiltered.map((r) => omit(r, ['data']))
       setData(responseFormatted);
       setUsers(responseFormatted);
     }


   }catch(e){
     setError(e);
   }
 },[])

 useEffect(() => {
   fetchList()
 },[fetchList])


 const fetchResident = useCallback(async(id) => {
   try{
     const user = await listOneItemApi(id)
     setUsers([user] || [])
   }catch(e){
     setError(e);
   }
 },[])

 const handleValueChange = e => {
   const value = e.target.value || undefined;
   const numericValue = value ? value.replace(/\D/g, '') : "";
   if(numericValue) {
     fetchResident(numericValue)
   }else {
     setUsers(data)
   }
   console.log(numericValue);
 };


 const [isModalOpen, setIsModalOpen] = useState(false);


const openModal = () => {
```

Figura 4: Implementação da Página Inicial da Aplicação em ReactJS

```
  setIsModalOpen(true);
];

const closeModal = () => {
  setIsModalOpen(false);
};

const addNewRow = async(data) => {
  try {
    data.appartament = Number(data.appartament)
    await registerApi(data);


    fetchList()
  } catch (e) {
    setError(e);
  }
  setIsModalOpen(false);
};

const columns =
[{Header:'Nome',accessorKey:'name'},{Header:'Telefone',accessorKey:'phone'},{Header:'Apartam
accessorKey:'appartament'},{Header:'Número da Vaga',accessorKey:'parkingNumber'},{Header:'Co
Emergência', accessorKey:'emergencyContact'}]


const handleDelete = async(data) => {
  try {
    console.log(data)
    await deleteApi(data.original.id)
    fetchList()
  }catch(e) {
    setError(e)
  }
}

const handleEdit = async (data) => {
  try {
    const payload = data.values
    await updateApi(data.row.original.id, payload)
    data.exitEditingMode()
    fetchList()
  }catch(e) {
    setError(e)
  }
}
```

Figura 4: [continuação] Implementação da Página Inicial da Aplicação em ReactJS

```
return(
  <>
  <Header />
    <Table
    handleValueChange={handleValueChange}
    columns={columns}
    data={users}
    isOpen={isModalOpen}
    onRequestClose={closeModal}
    openModal={openModal}
    addNewResident={addNewRow}
    onSubmitForm={addNewRow}
    handleDelete={handleDelete}
    handleEdit={handleEdit}
  />
</>

)
}
```

Figura 4: [continuação] Implementação da Página Inicial da Aplicação em ReactJS

Discussão

O projeto em questão teve como objetivo a criação de um sistema de gerenciamento de condomínio, utilizando uma abordagem de programação orientada a objeto. De forma resumida, o servidor do projeto, consiste em duas classes principais: uma classe que representa os residentes e uma classe responsável por uma lista de objetos dessa classe.

A primeira classe, denominada “Resident”, possui pelo menos 6 atributos, sendo um deles obrigatoriamente o ID, representado por um número inteiro. Além disso, são representados métodos getters e setters para acessar e modificar os atributos, bem como dois construtores: um padrão e outro que recebe os parâmetros necessários, exceto o ID. A propriedade do ID segue uma lógica sequencial, iniciando em um e sendo incrementada automaticamente.

A segunda classe, “DynamicArray”, é responsável por gerenciar uma lista de de classe “Resident”. Ela possui uma lista de objeto “list”, que é inicializado com um tamanho mínimo, e também mantém o controle do tamanho atual da lista por meio da propriedade “length”.

A classe “DynamicArray” é responsável pelos métodos de inserção, remoção, atualização e busca de um residente. A lógica de redimensionamento da lista garante que o sistema seja capaz de lidar com um número variável de residentes.

Para a criação da interface da aplicação, optou-se em utilizar JavaScript, juntamente com a biblioteca ReactJS, em vez de utilizar Swing. Dessa maneira, foi necessário a criação de uma API, para que fosse possível a comunicação entre o front-end e o back-end.

Para simplificar a criação do desenvolvimento da API, optou-se em utilizar Spring Boot, pois esse possui uma configuração automática e inteligente, o que garante uma alta produtividade durante o desenvolvimento da aplicação. Também possui suporte nativo para desenvolvimento a APIs RESTful, ou seja, fornece recursos para mapeamento de endpoints, serialização e deserialização de objetos em JSON.

Enquanto no desenvolvimento da parte gráfica, ou seja, interface, optou-se em utilizar Javascript e ReactJS, ao invés de Swing, pois considerou-se alguns aspectos como:

- O ReactJS, é uma biblioteca JavaScript voltada ao desenvolvimento web, dessa maneira, a aplicação poderia ser implantada em ambiente virtual, sem afetar a memória do usuário, também poderia ser acessada via celular, pois a estilização do CSS permite tornar todos os componentes responsivos, diferentemente do Swing, que é uma biblioteca Java para aplicativos Desktop.

- Também foi considerado os aspectos de componentização e reutilização da arquitetura do ReactJS, o que significa que a interface do usuário é dividida em componentes independentes e reutilizáveis.

- O ReactJs é altamente utilizado na comunidade de desenvolvimento.

Diante aos fatos apresentados, optou-se em utilizar ReactJs na criação da interface da aplicação.

Conclusão

Em suma, pode-se dizer que a criação de um sistema de gerenciamento de apartamentos baseado em uma abordagem de programação orientada a objetos usando a API em Spring Boot e a interface em ReactJS fornece uma solução moderna e eficaz para atender aos requisitos de gerenciamento de residentes em um condomínio.

Uma análise do código desenvolvido revela implementações adequadas das classes "Resident" e "DynamicArray" que permitem armazenamento, manipulação e recuperação eficientes de dados do residente. A estrutura modular e orientada a objetos oferece flexibilidade e facilidade na manutenção e ampliação do sistema no futuro.

A escolha do Spring Boot para criação de API tem várias vantagens, como facilidade de desenvolvimento, suporte a recursos avançados por meio do ecossistema Spring, gerenciamento automático de dependências e a capacidade de criar APIs RESTful escaláveis e de alto desempenho. A integração com o ReactJS garante uma comunicação eficiente entre front-end e back-end, permitindo a troca de dados e a interação com o sistema de gerenciamento de condomínio.

Por outro lado, usar o ReactJS como uma estrutura de interface do usuário tem vantagens como desenvolvimento da Web centrado no navegador, uma arquitetura baseada em componentes que promove a reutilização e a modularidade do código, atualizações de página suaves usando o DOM virtual, desempenho otimizado e um ecossistema colaborativo.

Essas tecnologias e abordagens fornecem um ambiente de desenvolvimento moderno e flexível para criar interfaces de usuário dinâmicas, responsivas e interativas. O ReactJS e o Spring Boot são amplamente usados pela comunidade de desenvolvedores, fornecendo suporte, recursos e atualizações contínuas para facilitar o desenvolvimento de sistemas de gerenciamento de residentes.

Percebe-se que usando Spring Boot para criar APIs e ReactJS para criar interfaces de usuário, o desenvolvimento do sistema de gerenciamento de apartamentos pode se beneficiar das melhores práticas, tecnologias mais recentes e abordagens orientadas a objetos para criar sistemas eficientes, escaláveis e intuitivos.

Referências

[1] Oracle. "Documentação do Java." Disponível em: https://docs.oracle.com/en/java/.

[2] Documentação do Spring Boot. Disponível em: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/.

[3] Mozilla Developer Network. "Documentação do JavaScript." Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide.

[4] Documentação do React. Disponível em: https://reactjs.org/docs/.