Trabalho Prático de Grupo

Mestrado em Engenharia Biomédica – Informática Médica

Processo Clínico e Eletrónico e
Aplicações Informáticas em Engenharia Biomédica
2º Semestre 2022/2023

Equipa docente
António Abelha, Cristiana Neto,
António Chaves

Beatriz Rodrigues Leite, PG50253
João Filipe da Costa Alves, PG50471
Mariana Rodrigues Oliveira, PG50635

ÍNDICE

Introdução

A interoperabilidade nos sistemas de saúde é essencial para permitir a comunicação eficiente e a troca de dados entre diferentes instituições de saúde. Ela possibilita a partilha e utilização das informações de forma integrada, facilitando o acesso aos dados clínicos dos pacientes em tempo real, mesmo que registados noutra instituição. Nesse contexto, a interoperabilidade semântica desempenha um papel fundamental, garantindo que os sistemas possam compreender e interpretar corretamente as informações trocadas. Tem um impacto significativo na segurança do paciente e na qualidade do atendimento, permitindo que os profissionais de saúde tenham acesso rápido às informações necessárias para tomar decisões adequadas.

Para alcançar a interoperabilidade na saúde, é necessário adotar padrões e estruturas de dados comuns. Uma dessas estruturas é o *OpenEHR*, uma especificação aberta e flexível para registos eletrónicos de saúde. Este define modelos de informação padronizados, permitindo a troca de informações entre diferentes sistemas de saúde de forma consistente e interoperável. Caso contrário, seria impossível várias organizações, cada uma com um método de registo e armazenamento de dados distinto, partilharem dados de forma eficiente.

Neste contexto, a aplicação desenvolvida no presente trabalho prático com um *Frontend* em *React* e *Backend* em *Node.js* para interoperabilidade na saúde desempenha um papel crucial. A aplicação permite a integração e troca de exames clínicos entre diferentes instituições de saúde, utilizando uma interface simples, um formulário que, desenvolvido no *Archetype Designer*, contém todos os campos essenciais para efetuar o registo de um exame, e permite o seu armazenamento de segundo as normas *OpenEHR*. Cada instituição tem a opção de efetuar um registo, ou mesmo consultar os registos efetuados de forma rápida, que pode ainda filtrar pela categoria da composição.

Arquitetura

A arquitetura da aplicação desenvolvida em *React* com Node.js e *MongoDB* seguiu uma abordagem de desenvolvimento *full-stack*, onde o *React* é usado para a criação da interface do utilizador (*Frontend*), Node.js para a implementação do servidor (*Backend*) e *MongoDB* para o armazenamento de dados.

O *Frontend* é desenvolvido usando o *React*, uma biblioteca *JavaScript* para criação de interfaces de utilizador. O *React* permite criar componentes reutilizáveis e interativas que representam diferentes partes da interface do utilizador. Foram usadas bibliotecas adicionais, como *React* Router para gerir a navegação entre páginas, e *Axios* para fazer requisições HTTP para o servidor.

O *Backend* é implementado usando o Node.js, um ambiente de execução *JavaScript* no lado do servidor. O Node.js permite criar um servidor web que lida com as requisições dos clientes e executa a lógica da aplicação.

O *MongoDB* é um banco de dados *NoSQL* orientado a documentos. Ele armazena dados em documentos JSON flexíveis, o que permite uma fácil manipulação e consulta dos dados. Esta ferramenta é amplamente utilizada com Node.js devido à compatibilidade natural com o formato de objetos *JavaScript*.

A comunicação entre o *Frontend* e o *Backend* é feita por meio de requisições HTTP. O *Frontend* envia requisições para o *Backend* para buscar ou enviar dados. O *Backend* processa essas requisições, executa a lógica e retorna uma resposta ao *Frontend*.

Por fim, refere-se a utilização do *Archetype Designer*, uma ferramenta de design de formulários que permite criar formulários personalizados com diferentes tipos de campos e validações. Ele pode ser integrado ao *Frontend* da aplicação *React* para exibir e validar os dados do formulário.

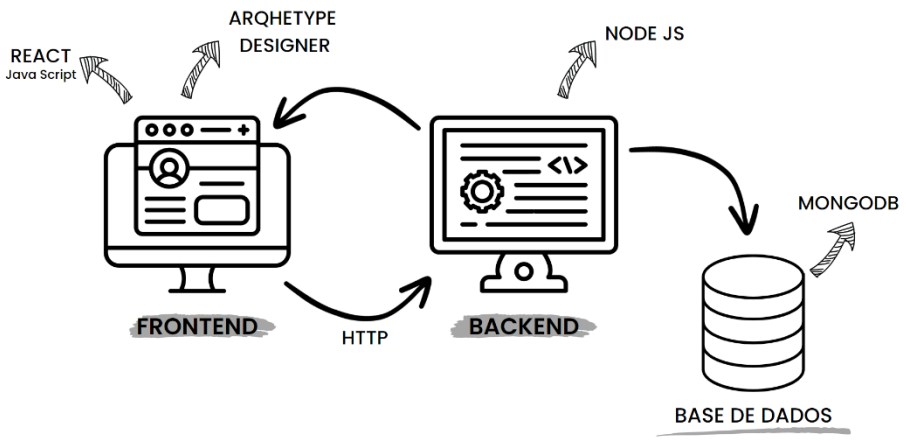

Figura 1. Arquitetura da aplicação desenvolvida.

Páginas *Web* e *Layout*

Como já referido, foi criada uma interface a ser utilizada pelo utilizador, em que se desenvolveu uma *home page*, entre outras diversas páginas web que têm como objetivo permitir a execução das principais funções da aplicação, posteriormente explicadas em detalhe. Neste sentido, para o desenvolvimento destas páginas recorreu-se à utilização de html e também *css*, definindo-se um layout para cada página e elementos comuns a todas elas, uma *navbar* e um *footer*. Isto permitiu aumentar a acessibilidade da aplicação para o utilizador.

Login e *Signin*

Um dos requisitos essenciais ao desenvolvimento do projeto foi a criação da autenticação do utilizador. Para tal, foi desenvolvida uma página de *Signin* e outra de Login.

Em relação ao *signin*, no *Frontend* define-se um formulário para a página web, o *SigninForm*, que é uma função chamada que retorna o formulário de criação de conta. Este componente utiliza o *hook useState* para definir vários estados, o nome, o email, a palavra-passe e a confirmação da palavra-passe. O código criado define várias funções manipuladoras de eventos para capturar e atualizar os valores dos campos de input. A função *handleSubmit* é chamada quando o formulário é submetido, e realiza as seguintes etapas: Verifica se as palavras-passe digitadas no campo de palavra-passe e confirmação de palavra-passe coincidem. Se não coincidirem, exibe uma mensagem de erro e interrompe o processamento; Envia uma requisição *POST* para a rota '*/signIn*' do servidor, passando o email e a palavra-passe inseridos pelo utilizador; Aguarda a resposta do servidor e verifica se a criação de conta foi feita com sucesso ou não; Redireciona o utilizador para a página de login (utilizando o *hook useNavigate*).

```
function SigninForm() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [confirmPassword, setConfirmPassword] = useState('');
  const [message, setMessage] = useState('');
  const navigate = useNavigate();

  const handleNameChange = (event) => {
    setName(event.target.value);
  };

  const handleEmailChange = (event) => {
    setEmail(event.target.value);
  };

  const handlePasswordChange = (event) => {
    setPassword(event.target.value);
  };

  const handleConfirmPasswordChange = (event) => {
    setConfirmPassword(event.target.value);
  };

  const handleSubmit = async (event) => {
    event.preventDefault();
    if (password !== confirmPassword) {
      setMessage("Passwords do not match");
      return;
    }
    try {
      const response = await axios.post('http://localhost:8080/signIn', {
        email: email,
        password: password
      });
      if (response.data.success) {
        setMessage('Account created successfully');
        navigate('/login');
      } else {
        setMessage(response.data.info);
      }
    } catch (error) {
      console.error(error);
      setMessage('An error occurred');
    }
  };
```

Figura 2. Implementação do formulário SigninForm.

O *Signin* é também efetuado na base dados, pelo que no *backend* é criada uma função *signIn* que é exportada como um módulo. Ela recebe o email e a palavra-passe como parâmetros. Dentro da função, é criada uma instância do modelo *LoginSchema* com o email e a palavra-passe fornecidos. Em seguida, essa instância é salva na base de dados usando o método *save*(). Se isto for bem-sucedido, a função retorna um objeto com a propriedade *Success* definida como *true* e o documento salvo como response. Caso ocorra algum erro durante o processo, a função retorna um objeto com a propriedade *Success* definida como false e uma mensagem de erro como *response*.

```
module.exports.signIn = async (email, password) => {
  try {
    let signIn = new LoginSchema({ email, password });
    let response = await signIn.save();
    return { Success: true, response };
  } catch (err) {
    console.log(err);
    return { Success: false, response: err };
  }
};
```

Figura 3. Módulo do SignIn.

No que toca ao Login, em termos de *Frontend*, foi criada uma página web que inclui um formulário, *LoginForm*, que contém um estado local que armazena o email, a palavra-passe e a mensagem de retorno. Também é definida uma variável *navigate* para aceder à função de navegação do *React Router*. Existem dois manipuladores de eventos, *handleEmailChange* e *handlePasswordChange*, que atualizam o estado do email e palavra-passe, respetivamente, quando o valor dos campos de entrada é alterado. O manipulador *handleSubmit* é chamado quando o formulário é enviado. Ele envia uma solicitação *POST* para a *URL http://localhost:8080/login* com as informações de email e palavra-passe fornecidas pelo utilizador. Em seguida, com base na resposta, exibe uma mensagem de sucesso ou uma mensagem de erro. O formulário inclui campos de entrada para email e senha, um botão de envio e uma mensagem de retorno.

```
function LoginForm() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [message, setMessage] = useState('');
  const navigate = useNavigate();

  const handleEmailChange = (event) => {
    setEmail(event.target.value);
  };

  const handlePasswordChange = (event) => {
    setPassword(event.target.value);
  };

  const handleSubmit = async (event) => {
    event.preventDefault();
    try {
      const response = await axios.post('http://localhost:8080/login', {
        email: email,
        password: password
      });
      if (response.data.success) {
        setMessage('Login Successful');
        navigate('/afterlog');
      } else {
        setMessage(response.data.info);
      }
    } catch (error) {
      console.error(error);
      setMessage('An error occurred');
    }
  };
```

Figura 4. Implementação do formulário LoginForm.

Aquando do *Login*, é feita uma verificação da existência do utilizador na base de dados. Tal é possível, no *Backend*, com a criação de uma função módulo para o *Login*. Esta recebe o email e a palavra-passe como parâmetros. Dentro da função, é realizada uma consulta à base de dados usando o modelo *LoginSchema.findOne()*. Ele procura um documento na base de dados que corresponda ao email e palavra-passe fornecidos. Se um documento for encontrado, significa que o *login* foi bem-sucedido, e a função retorna um objeto com a propriedade *success* definida como *true* e o documento de *login* como *response*. Caso contrário, a função retorna um objeto com a propriedade *success* definida como *false* e uma mensagem de erro como *response*. Este processo ocorre juntamente com a definição do esquema, onde é criado um objeto *LoginSchema* usando o construtor *Schema* do *mongoose*. O esquema define a estrutura dos documentos de *login* no *MongoDB*. O esquema possui duas propriedades, o email (do tipo *String*, obrigatória e

única) e a palavra-passe (do tipo *String* e obrigatória). O objeto *LoginSchema* também possui a opção {*timestamps: true*}, que adiciona automaticamente as propriedades *createdAt* e *updatedAt* ao documento, registando a data e hora de criação e atualização.

```
module.exports.login = async (email, password) => {
  try {
    let login = await LoginSchema.findOne({ email: email, password: password });
    console.log(login);
    if (login) {
      return { success: true, response: login };
    } else {
      return { success: false, response: 'Invalid email or password' };
    }
  } catch (err) {
    console.log(err);
    return { success: false, response: err };
  }
};
```

Figura 5. Módulo do Login.

Formulário

Seguido ao login, o utilizador depara-se com uma página que lhe apresenta 2 opções: o registo de uma nova composição, ou a listagem de todas as composições já existentes. No caso de selecionar a opção "Nova Composição" o utilizador depara-se com um formulário que pode preencher. Os Formulários são as interfaces para os profissionais de saúde, que os utilizam para inserir os dados dos pacientes no sistema. Estes são criados usando modelos e oferecem uma interface familiar e intuitiva para a entrada de dados. Os formulários podem ser personalizados para atender às necessidades específicas de diferentes provedores de saúde, departamentos ou fluxos de trabalho. A sua personalização, neste caso para ser aplicado a registos de exames médicos envolveu várias etapas:

- Inicialmente foram analisados os campos a preencher necessários para esta aplicação em específico. Para tal, foi analisado um exemplo de mensagem FHIR, no documento fornecido pelos docentes, "Especificações técnicas sincronização de resultados de MCDT´s e pedidos de resultados de MCDT´s ", de 03 de março de 2022. A mensagem analisada foi "Mensagem para a sincronização dos resultados de um MCDT realizado por um laboratório - Caso de um eletrocardiograma (ECG)";

```
{
    "resourceType": "Bundle",
    "id": "c3b36c20-0fff-44a0-bf48-6ef1b81f45a0",
    "meta": {
        "lastUpdated": "2017-11-15T12:16:00.358+00:00",
        "profile": [
"http://spms.min-saude.pt/pnb/profiles/mcdt_results_sync/v1.5.8"
        ]
    },
    "type": "message",
    "entry": [
        {
            "fullUrl": "http://spms.min-saude.pt/fhir/MessageHeader/ac185937-cc34-4e42-9459-ae5e1306c34d","resource": {
                "resourceType": "MessageHeader",
                "id": "ac185937-cc34-4e42-9459-ae5e1306c34d",
                "event": {
                    "system": "http://spms.min-saude.pt/pnb/events",
                    "code": "10705",
                    "display": "MCDT_RESULTS_SYNCHRONIZATION"
                },
                "destination": [
                    {
                        "name": "RequiringApp",
                        "endpoint": "SPMS/HEP"
                    },
                    {
                        "name": "BDNR",
                        "endpoint": "SPMS/BDNR"
                    }
```


Figura 6- Parte da Mensagem para a sincronização dos resultados de um ECG.

- De seguida foi feita a importação de *archetypes* existentes no *CKM* para o *Archetype Designer*, o que permitiu adicionar e combinar estes *archetypes* pré-estruturados no nosso *template*;
- Cada campo foi devidamente analisado, sendo que nem todos os que se encontravam na mensagem faziam sentido no *template* final (por serem repetitivos ou não acharmos relevantes o suficiente para serem preenchidos pelo profissional de saúde). Para os campos do tipo *Coded Text*, as opções existentes foram extraídas do guia de especificações técnicas supramencionado (que contem ligações para *hl7.org*);

Code	Display	Definition
male	Male	Male
female	Female	Female
other	Other	Other
unknown	Unknown	Unknown

Figura 7- Códigos especificados para o género do paciente através das normas hl7

- Após concluído o *template*, este é extraído no formato de OPT (*Operational Template*). Este foi posteriormente transformado em JDT (*Concrete form JSON Data Templates*). Infelizmente, durante esta transformação, alguns campos *Coded Text* perderam a funcionalidade de código. Para além disso, alguns dos campos que inicialmente se encontravam no JDT foram posteriormente eliminados antes da construção do formulário, por serem repetitivos. Esta transformação resultou em três ficheiros distintos. Um primeiro ficheiro, "jdt_imagem.json" representa uma estruturação dos dados e os campos e códigos existentes para o formulário, regras de visibilidade e outras propriedades; e um segundo ficheiro, "style_imagem.json" que especifica configurações de estilo de vários elementos do formulário, incluindo o alinhamento, tipo e tamanho

de letra, entre outros, ou seja, é utilizado para definir a aparência dos elementos do formulário. Ambos os ficheiros são posteriormente importados para o terceiro ficheiro *JavaScript* "formulário/js".

Tendo então os ficheiros que permitem a construção de um formulário estruturado e intuitivo, o ficheiro *JavaScript* utiliza o componente *Form*, importado da libraria "protected-aidaforms" para apresentar o formulário. Dentro deste objeto *Form*, existem várias variáveis para configurar o comportamento e aparência do formulário. De destacar a variável *template* e *formDesign,* que correspondem aos ficheiros importados; e a variável *onSubmit*, que é a que efetua a ligação à base de dados. Quando o formulário é submetido, chama a função *onSubmit*, que envia um pedido *POST* para a rota do *Backend* "/newcomposition", com os valores obtidos do formulário. Se o pedido for bem-sucedido, um aviso de sucesso é apresentado. Caso contrário é apresentado um aviso de erro de submissão do formulário.

```
import axios from 'axios';
import React from 'react';
import {Form} from "protected-aidaforms";
import { useNavigate } from 'react-router-dom';

let json = require('./jdt_imagem.json');
let style= require('./style_imagem.json');


const Forms = () => {

  const navigate = useNavigate();

  const onSubmit = async (values) => {
    values = JSON.parse(values)
    try {
      const response = await axios.post('http://localhost:8080/newcomposition', {values});
      console.log(values);
      console.log('POST request successful:', response.data);
      alert('Composition submitted successfully!');
      navigate('/afterlog');
    } catch (error) {
      console.error('Error submitting composition:', error);
      alert('Error submitting composition. Please try again.');
    }
  };

  return (
    <div className="Form">
      <Form
        onSubmit={(values)=> onSubmit(values)}
        onSave={(values, changedFields) => console.log("SAVED VALUES: ", values, "CHANGED FIELDS: ", changedFields)}
        onCancel={status => console.log("CANCELLED:", status)}
        dlm={{}}
        template={json}
        showPrint={true}
        editMode={true}
        professionalTasks={["Registar Pedido", "Consultar Pedido","Anular Pedido"]}
        canSubmit={true}
        canSave={true}
        canCancel={true}
        formDesign= {JSON.stringify(style)}
        submitButtonDisabled={false}
        saveButtonDisabled={false}
      />
      </div>
    );
}

export default Forms;
```

Figura 8. Código referente à página do formulário.

O registo da composição na base de dados é efetuado no *Backend* através de uma função *newComposition* que é exportada como um módulo e que lida com a manipulação das composições. Esta recebe como parâmetro um objeto chamado *composition*. E dentro desta, são executadas as seguintes etapas. Primeiro, extrai os valores da composição, que estão armazenadas na propriedade *values* do objeto *composition*. Em seguida, esta itera sobre as chaves desse objeto, através do *Object.keys(compositionValues).forEach((key) => {...})*. Durante a iteração, este verifica se o valor correspondente à chave atual é uma *string*. Em caso afirmativo, este analisa esse valor como *JSON*. Desta forma caso este *JSON* resultante possuir *blocks*, este substitui o valor original pelo texto contido nesse primeiro bloco, eliminando desta forma esta propriedade. Além disso, também verifica se o valor correspondente à chave atual é uma matriz e se possui pelo menos um elemento. Se essas condições forem verificadas, tal como efetuado acima, este tenta analisar o valor do primeiro elemento como *JSON*. Novamente, se o *JSON* resultante possuir uma propriedade chamada *blocks*, há a substituição do valor original pelo texto contido no primeiro bloco. Após manipular os valores da composição, há a criação de um novo objeto chamado *newComp* utilizando um esquema chamado *CompositionSchema* e já sem a presença de *blocks*, que é o objetivo primordial desta função.

Em seguida, essa instância é salva na base de dados usando o método *save()*. Se isto for bem-sucedido, a função retorna um objeto com a propriedade *Success* definida como *true*. Caso ocorra algum erro durante o processo, a função retorna um objeto com a propriedade *Success* definida como false e uma mensagem de erro como *response*.

Na mesma rota e caso a criação da composição seja bem-sucedida (*newCompositionResponse.Success* é verdadeiro), é efetuada a transformação da composição numa mensagem FHIR, e o seu armazenamento na base de dados.

Desta forma existe a criação de uma lista de substituições chamada *subs*. Essa lista contém pares de valores de procura e substituição, que serão utilizados posteriormente. Antes de se avançar com a restante explicação deste método, é necessário indicar que o mapeamento da *composition* para mensagem *FHIR* foi realizado através da conversão manual dos respetivos valores a alterar do ficheiro exemplo *JSON* da mensagem *FHIR* com os respetivos caminhos da *composition*. Desta forma e já com o ficheiro exemplo *FHIR* com essas indicações, foi possível utilizar valores de procura e substituição, em que este procura a indicação do respetivo *item* da *composition* criada manualmente no ficheiro *FHIR* e o substitui pelo respetivo *item* da *composition*.

Tendo isso em conta e após a criação já mencionada da lista de substituições, este chama a função *transformToFhir* para realizar a conversão da composição para o formato *FHIR*. Esta função recebe como parâmetro o ficheiro exemplo *FHIR JSON* chamado *fhir_message*, que contém um modelo de mensagem FHIR pré-definido. Este modelo é copiado para evitar a modificação direta e possíveis complicações derivadas de apontar para a mesma variável. De seguida, a função *iterate* é chamada.

A função *iterate* por sua vez, percorre recursivamente as propriedades do objeto *JSON* e realiza as substituições de valor conforme especificado na lista *subs*. Isto permite que os valores manualmente colocados quando encontrados no modelo *FHIR* sejam substituídos pelos valores correspondentes da composição original.

Por fim e de modo a que os *Ids* tanto da composição como da mensagem sejam iguais, após a conclusão da conversão para *FHIR*, é utilizado o *ID* da composição obtido da *newCompositionResponse.response.composition_id* para enviar a nova mensagem *FHIR* para o controlador *messageFHIRController.newMessageFHIR*.

Se a criação da mensagem *FHIR* for bem-sucedida (*newMessageFHIRResponse.Success* é verdadeiro) retorna uma resposta de sucesso para a requisição, indicando que a composição e a mensagem *FHIR* foram adicionadas com sucesso e há o armazenamento na base de dados.

Caso contrário retorna uma resposta indicando que a mensagem *FHIR* não foi adicionada. Se a criação da composição não for bem-sucedida (*newCompositionResponse.Success* é falso), retorna uma resposta indicando que a composição não foi adicionada.

Tabela de mensagens

A outra opção disponível após ser efetuado o login é a opção "Listar Composições". Aqui, o utilizador depara-se com uma tabela com as colunas "Data de criação", "ID", "Paciente", "Mensagem", "Formulário" e "Apagar".

Para apresentação dos resultados, numa fase inicial, através do *useEffect*() do *React*, é efetuado um pedido ao *Backend* para a rota "/listfhirs".

```
useEffect(() => {
  axios.post('http://localhost:8080/listfhirs').then((response) => {
    if (response.data.success) {
      setFhirList(response.data.data.response);
      setFilteredFhirList(response.data.data.response);
      console.log(response.data.data.response);
    } else {
      console.error('Error retrieving designations');
      setSuccess(response.data.response);
    }
  });
}, []);
```

Figura 9- Método useEffect() da página de listagem de mensagens FHIR

Esta permite, através do método *listFHIRs* do *controller "*messageFHIRController*"*, e utilizando o método find() do Node.JS, listar todas as mensagens *FHIR* existentes na base de dados e enviá-las para o *Frontend*.

Existe ainda na página um filtro para a categoria/tipo de exame, que quando alterado, recorre ao *handleCategoryChange*, que filtra os resultados obtidos do *Backend* pela categoria, apenas apresentando assim, na tabela, as mensagens da categoria correspondente.

```
const handleCategoryChange = (e) => {
  const selectedCategory = e.target.value;
  setSelectedCategory(selectedCategory);
  if (selectedCategory === '') {
    setFilteredFhirList(fhirList);
  } else {
    const filteredList = fhirList.filter(
      (fhir) => fhir.fhir.entry[2].resource.category.coding[0].display === selectedCategory
    );
    setFilteredFhirList(filteredList);
  }
};
```

Figura 10- Método handleCategoryChange() da página de listagem de mensagens FHIR

São apresentados os diferentes atributos de cada mensagem em cada coluna.

No caso das colunas "Mensagem" e "Formulário", são apresentados um *link* em cada que redireciona o utilizador para a página que apresenta a mensagem *FHIR* em questão e o formulário preenchido de acordo com as informações das mensagens, respetivamente.

Quando selecionados ambos os links, implementam duas rotas no lado do servidor e duas páginas no lado do cliente, uma rota e página para cada *link* selecionado.

Desta forma, quando o *link* de Mensagem é selecionado, recorre do lado do servidor, à rota que é *GET /mensagem/:id*. Quando essa rota é acessada, o identificador é extraído dos parâmetros de requisição *(req.params.id)*. De seguida, é feita uma chamada assíncrona à função *getJsonById* do *controller messageFHIRController*, com o identificador como argumento. O resultado é armazenado em *jsonByIdResponse*.

Se *jsonByIdResponse.success* for falso, significa que ocorreu um erro ao obter a mensagem FHIR e é lançada uma exceção com a mensagem "Erro a obter a mensagem FHIR". Caso contrário, a resposta *HTTP* é enviada ao cliente com um status de 200 (sucesso) e um objeto *JSON* que contém as propriedades *success* (verdadeiro), *info* ("Mensagem recolhida com sucesso") e *data* (o valor de *jsonByIdResponse*) com a respetiva mensagem *FHIR* do identificador requerido.

Quando o *link* de Formulário é selecionado, recorre do lado do servidor à rota *GET /mensagemcomposition/:id*, que segue uma lógica semelhante à primeira rota, mas agora a chamada assíncrona é feita à função *getJsonById* do *controller CompositionController*. A resposta HTTP retornada ao cliente é análoga à primeira rota.

Relativamente ao lado do cliente, a página relacionada com o Formulário é renderizada pelo componente Formulario. Este componente recebe o identificador por meio do *hook useParams* do *React Router*. De seguida faz uma chamada assíncrona para a rota do servidor falado acima do *GET /mensagemcomposition/:id* usando o *axios*. Após receber a resposta, isto é, o conteúdo da *composition* respetiva, o valor *jsonData* é extraído de *response.data.data.response.values* e armazenado no estado *jsonData* por meio da função *setJsonData*. O estado *loading* é usado para indicar se os dados estão a ser carregados ou não.

Este componente também possui uma função *getModifiedJDT* que retorna um objeto JSON referente ao template do *Forms* modificado com base no *jsonData* recebido. Esta função utiliza um módulo externo chamado *replace_values_jdt*, que realiza as respetivas substituições de valores no *template* original do *forms* para que este tenha já os valores retirados da *compositon* e assim consiga mostrar esses mesmos valores no formulário.

A renderização do componente depende do estado *loading*. Se *loading* for verdadeiro, é exibida a mensagem "*Loading*...". Caso contrário, é renderizado um componente *Form* com várias propriedades, incluindo o template fornecido por *getModifiedJDT*, ou seja, já com os valores respetivos da *composition* e o design do formulário definido pelo arquivo *style_imagem.json*. Este componente permite a exibição do formulário.

Quanto à página relacionada com a Mensagem é renderizada pelo componente Mensagem. Esta também recebe o identificador por meio do *hook useParams*. De seguida e tal como já referido anteriormente é feita uma chamada assíncrona para a rota do servidor *GET /mensagem/:id* usando o *axios*. Após receber a resposta, o valor *jsonData* é armazenado no estado data.

A renderização do componente exibe o título "Mensagem FHIR" e, se *data* existir, exibe o objeto *JSON* retornado pela requisição em formato de texto.

No caso da coluna “Apagar”, é apresentado para cada registo um botão, que quando selecionado recorre ao método *deleteTerm(id).* Este faz um pedido *Axios DELETE* ao *Backend* para a rota “/delete/:id”.

```
const deleteTerm = (id) => {
  axios
    .delete('http://localhost:8080/delete/' + id)
    .then((res) => {
      alert(res.data.info);
      window.location.reload();
    })
    .catch((error) => {
      setSuccess('Erro ao apagar a mensagem.');
    });
};
```

Figura 11- Método deleteTerm() da página de listagem de mensagens FHIR

Nesta rota, o identificador da mensagem *FHIR* (que é o mesmo da *composition* correspondente) é capturado através do parâmetro *id* do pedido, e é então efetuada a eliminação tanto da *composition* como da mensagem *FHIR*, através dos métodos *deleteComposition*() e *deleteMensagem*(), respetivamente. Estes métodos funcionam de forma semelhante, em *controllers* diferentes, logo com acesso a modelos diferentes. O funcionamento passa então pela utilização do método *deleteOne*() que elimina da base de dados o registo com o id fornecido. Se não ocorrer nenhum erro, a página e a tabela são então atualizadas.

Por último, é de notar que para o correto funcionamento da aplicação foi necessário a atenção a outros ficheiros da aplicação, para definição de rotas e outras especificações *React*.

Conclusão

Após a conclusão do projeto, é possível afirmar que os principais objetivos foram alcançados com sucesso. A interface da aplicação foi desenvolvida de forma eficiente, permitindo aos utilizadores realizar login, criar contas, preencher formulários e visualizar mensagens FHIR de maneira prática e intuitiva. O uso das ferramentas como o *Archetype Designer* e o *MongoDB* também contribuíram para o êxito do projeto, proporcionando recursos adicionais e suporte ao armazenamento de dados.

Uma das principais aprendizagens obtidas durante o desenvolvimento do projeto está relacionada com a transformação de composições em mensagens *FHIR*, que possibilita a comunicação e interoperabilidade entre diferentes instituições de saúde. Esta aplicação realista em um contexto hospitalar demonstra o valor e a relevância da integração de sistemas de saúde.

Algumas melhorias a implementar passariam por aprimorar a interface, averiguando maneiras de torná-la ainda mais intuitiva e amigável; aperfeiçoar a validação de dados, implementando uma validação mais abrangente nos formulários para garantir a integridade dos dados recebidos, evitando erros ou inconsistências; e por efetuar a verificação do login em cada etapa do website, com diferentes permissões para diferentes utilizadores.

Em resumo, o projeto alcançou os seus objetivos principais, proporcionando uma experiência útil e relevante na área da saúde. As sugestões de melhoria propostas visam aperfeiçoar ainda mais a aplicação, aprimorando sua utilidade e desempenho.