IPG Politécnico da Guarda
Polytechnic of Guarda

RELATÓRIO DE ESTÁGIO

Licenciatura em Engenharia Informática

Nil Silva

novembro | 2020

Escola Superior de Tecnologia e Gestão

Instituto Politécnico da Guarda

RELATÓRIO DE ESTÁGIO

SIMULAÇÃO DO CRÉDITO DE HABITAÇÃO

NIL SILVA

RELATÓRIO PARA A OBTENÇÃO DO GRAU DE LICENCIADO

EM ENGENHARIA INFORMÁTICA

11/2020

Ficha de identificação

Aluno

Nil Silva

Nº 1700774

Licenciatura: Engenharia Informática

Estabelecimento de ensino

Escola Superior de Tecnologia e Gestão

Instituto Politécnico da Guarda

Orientador de Estágio

Doutor José Fonseca

Grau académico: Doutoramento

Supervisor

Gonçalo Neto

Grau académico: Mestrado

Duração do estágio

280 horas

Inicio: 1 junho de 2020

Fim: 21 de julho de 2020

Agradecimentos

Primeiro queria agradecer à empresa Softinsa por me aceitar durante o período de estágio e ao meu supervisor, Gonçalo Neto, por me guiar ao longo do projeto mesmo com todas as dificuldades que a situação atual proporciona.

Queria também agradecer ao meu colega Fernando Lopes. Fizemos o estágio juntos, mas fizemos diferentes partes do mesmo projeto. Ele foi uma grande ajuda sempre que precisei.

Não posso deixar de agradecer a todos os professores que ao longo destes três anos me deram o conhecimento necessário para conseguir fazer um projeto com este nível de qualidade. Destes quero destacar o meu orientador, José Fonseca, pois sempre que pode foi uma grande ajuda em todo o meu percurso académico.

Queria ainda agradecer aos meus pais e irmãos pelo apoio que me deram não só durante o meu tempo no IPG, mas durante a minha vida toda. Sem as palavras deles, nem sempre boas, não teria chegado onde estou hoje.

Obrigado.

Resumo

Este documento descreve o planeamento e implementação de uma aplicação web de simulação de crédito de habitação dando aos utilizadores uma melhor ideia de quanto o banco está desposto a emprestar e quais são as condições. A aplicação foi desenvolvida durante o estágio na Softinsa, no âmbito da unidade curricular de Projeto de Informática da Licenciatura de Engenharia Informática da Escola Superior de Tecnologia e Gestão do Instituto Politécnico da Guarda.

A aplicação foi feita em conjunto com o aluno Fernando Lopes, sendo o trabalho dividido pelos dois igualmente.

A aplicação está dividida em duas partes. O *frontend* foi desenvolvido em Angular enquanto o *backend* foi desenvolvido em Node.Js. O *backend* guarda os dados inseridos pelo utilizador numa base de dados PostgreSQL utilizando o Sequelize. A aplicação tem um sistema de registo e autenticação de utilizadores. Dependendo do seu *role* o utilizador tem acesso a partes diferentes da aplicação. Os clientes podem fazer simulações e marcar visitas numa agência para falar com um representante. Os funcionários podem, para além de fazer simulações, gerir as agências disponíveis na base de dados. E, por fim, os administradores podem eliminar e alterar as permissões dos utilizadores.

O objetivo é, no final, ter uma aplicação que permite aos utilizadores fazerem o registo, autenticação e simulações do crédito de habitação. Estes objetivos foram cumpridos.

Palavras-chaves: *Web services*, crédito de habitação, Node.Js, Angular, PostgreSQL

Abstract

This document describes the planning and implementation of a web application to do mortgage credit simulations, so the users have a better idea of how much the bank is willing to lend and on what conditions. This app was developed throughout the internship at Softinsa, within the scope of the “Projeto de informática” course of the “Engenharia Informática” degree at “Escola superior de tecnologia e gestão” of the “Instituto politécnico da Guarda”.

The application was made together with the student Fernando Lopes, as such the workload was divided by the two equally.

The application is divided into two parts. The front end was developed in Angular while the back end was developed in Node.Js. The back end stores the data entered by the user in a PostgreSQL database. The application has a user authentication system. Depending on the user's permissions, what can be done changes. Customers can run simulations and schedule visits to an agency to speak to a representative. Employees can, in addition to simulations, manage the agencies available in the database. Finally, administrators can delete and change users' permissions.

The objective is by the end to have an application that allows the users to register, login and do mortgage credit simulations. These objectives were completed.

Keywords: Web services, mortgage loans, Node.Js, Angular, PostgreSQL

Índice

Índice de Figuras

Siglário

API	application programming interface
CLI	Command-line interface
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheets
DBMS	Database Management System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated development environment
JWT	JSON Web Token
ORM	Object–relational mapping
SSR	Server-side rendering
VSC	Visual Studio code
URL	Uniform Resource Locator

1 Introdução

Este relatório descreve o projeto desenvolvido em contexto de estágio na Softinsa do aluno Nil Silva, no âmbito da unidade curricular Projeto de Informática, da licenciatura em Engenharia Informática.

O estágio decorreu do dia 1 de junho a 21 de julho do ano de 2020. Durante o estágio foi desenvolvida uma aplicação de simulação do crédito de habitação.

1.1 Caraterização sumária da Softinsa

A Softinsa, é uma empresa subsidiária da IBM, e especialista em serviços de gestão e desenvolvimento de aplicações e infraestruturas.

Com 22 anos de história e experiência no mercado português, conta atualmente com uma equipa de mais de 1000 profissionais, distribuídos por Lisboa e pelos Centros de Inovação de Tomar, Viseu e Fundão (1).

1.2 Motivação

O estágio/projeto é a última etapa do curso de engenharia informática e, como tal, é a última oportunidade de ganhar experiência antes de entrar no mercado de trabalho.

A Softinsa deu a oportunidade de trabalhar com tecnologias inovadoras num contexto realista foi, por isso, uma experiência importante.

1.3 Objetivos

A aplicação desenvolvida durante o estágio consiste em fazer uma simulação do crédito de habitação e agendar uma visita a uma agência para falar com um representante. A aplicação também tem um sistema de registo e autenticação de utilizadores usando JWT[1] (*JSON Web Token*). Existem três tipos de utilizadores que são os clientes, os funcionários e os administradores. Cada tipo de utilizador tem acesso a funcionalidades diferentes. Os clientes podem fazer simulações do crédito de habitação e marcar reuniões para falar com um representante, os funcionários para além de poderem fazer as simulações podem inserir ou eliminar agências na base de dados para as quais as reuniões são marcadas e os administradores podem eliminar e alterar as permissões dos utilizadores. O *backend* foi desenvolvido usando Node.Js sendo uma API (*application programming interface*) à qual o *frontend*, desenvolvido em Angular, faz pedidos HTTP (*Hypertext Transfer Protocol*). O *backend* manipula uma base de dados PostgreSQL. No início do estágio foram convertidos uns *mockups* para HTML (*Hypertext Markup Language*) e CSS (*Cascading Style Sheets*). Os *mockups* eram cerca 72 imagens, fornecidas pela Softinsa, que mostravam como a *interface* da aplicação deveria funcionar. Estas páginas HTML são depois usadas para construir o

[1] JWT - um meio seguro e compacto de verificar a identidade uma entidade.

frontend em Angular. Por fim a aplicação deve usar SSR[2] (*Server-side rendering*). Com isto em mente a aplicação tem os seguintes objetivos:

1. Permitir ao utilizador introduzir os dados relevantes sobre a simulação do crédito de habitação;
2. Armazenar esses dados numa base de dados PostgreSQL;
3. Sistema de registo e autenticação dos utilizadores. Os utilizadores devem poder fazer *login*, *logout* e poderem alterar as informações da sua conta;
4. Usar SSR. Usando SSR a parte estática das páginas é carregada mais rapidamente, pois é renderizada no servidor, e permite ao utilizador ver o *layout* da página antes de ela ficar completamente interativa.

1.4 Estrutura do projeto

Este relatório está dividido em sete capítulos.

No primeiro é feita uma introdução ao projeto. O segundo descreve a metodologia usada no projeto. O terceiro descreve a análise de requisitos feita. O quarto descreve as ferramentas e o processo de desenvolvimento. O quinto mostra a aplicação em funcionamento usando imagens. No sexto capítulo são descritos os testes. E no sétimo é feita a conclusão e discussão dos resultados.

[2] SSR – utilizado para renderizar aplicações Angular no servidor.

2 Metodologia

Nesta secção vai ser descrita a metodologia de desenvolvimento utilizada, que foi o Scrum, e como esta foi aplicada no neste projeto. Foi escolhida esta metodologia porque é a que a Softinsa utiliza e que sugeriu ser usada.

No Scrum, os participantes pertencem a diferentes papéis:

- ***Product owner*** – é elemento responsável por documentar e organizar a lista de requerimentos por prioridade. Quem assumiu o papel de *product owner* neste estágio foram os supervisores Gonçalo Neto e Tiago Rebelo.
- ***Scrum master*** – é o elemento responsável pela implementação do processo Scrum e coordenação da equipa. Os srum masters foram os supervisores Gonçalo Neto e Tiago Rebelo
- ***Membros da equipa*** – são os elementos que completam as tarefas. Tipicamente a equipa é constituída de dois a sete membros. Os membros da equipa foram os alunos Nil Silva e Fernando Lopes.

Durante o desenvolvimento, são criados artefactos:

- ***Product backlog*** **-** uma lista de requisitos do projeto, ordenados por nível de prioridade. Isto foi determinado pelos supervisores.
- ***Sprint backlog*** **–** o *sprint backlog* são os requisitos a trabalhar no próximo *sprint*. Os requisitos são escolhidos do *product backlog*.
- **Incremento do produto** – produto final de cada *sprint*.

Devido à política de privacidade da Softinsa o *product backlog* não pode ser discutido neste relatório em grande detalhe.

O Scrum pode ser descrito nas seguintes etapas (Figura 1):

1. **Análise dos requisitos -** Os requisitos do sistema são estudados e com estes é formado o *product backlog*. Isto foi discutido no início do projeto pelos membros da equipa, pelo *product owner* e pelo *scrum master*.

2. **Planeamento do *sprint* –** Nesta reunião é formado o *sprint backlog*. É nesta fase que se decide quem faz o quê. Esta reunião era feita no início de cada *sprint* e envolvia os membros da equipa e o *scrum master*.

3. ***Sprint* –** O sprint é a etapa onde o novo incremento do produto é criado. Os sprints feitos neste projeto tinham a duração de uma a duas semanas dependendo do que era necessário fazer.

4. ***The daily Scrum* –** Uma pequena reunião diária, tipicamente de 10 a 15 minutos, com a presença dos elementos da equipa e do *scrum master*. Cada um dos participantes reporta o que fez desde a última reunião, o que vai fazer até à próxima e que problemas encontrou. Durante o estágio estas reuniões só foram feitas se necessário.

5. ***Sprint review*** **–** No final de cada sprint é necessário avaliar o que foi feito. A equipa mostra o resultado do sprint e analisam o que correu bem, o que podia ser melhorado e como trabalhar melhor em equipa no próximo sprint. Esta reunião envolvia os membros da equipa, o *product owner* e o *scrum master*.

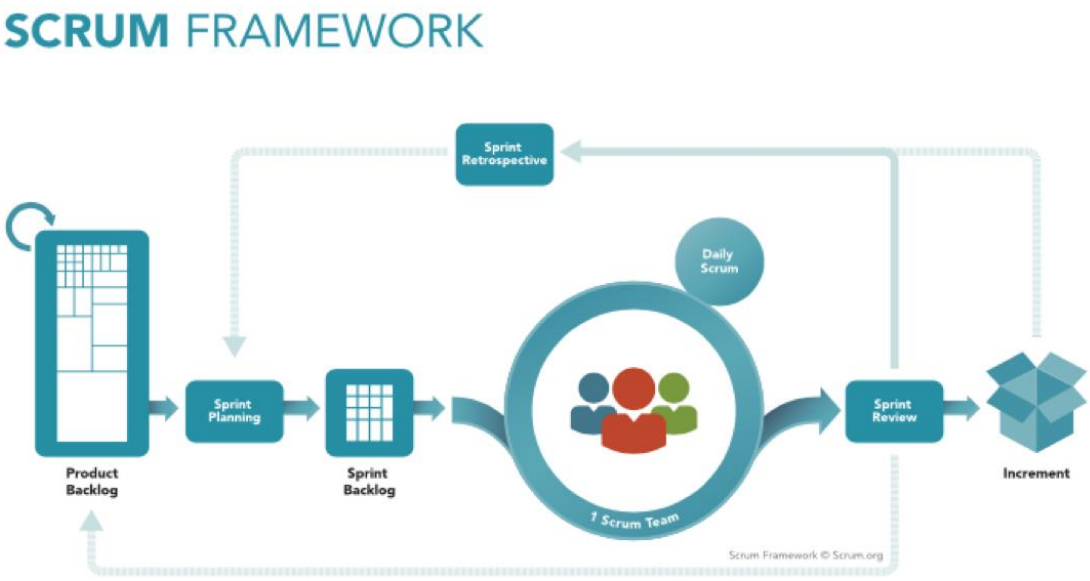

Figura 1 Visualização do SCRUM

https://www.scrum.org/resources/blog/scrum-and-projects

3 Análise de requisitos

A análise de requisitos serve para formar uma melhor ideia de como a aplicação vai funcionar. Para fazer a análise forem construídos uma serie de diagramas, em UML, e tabelas.

Durante o estágio não houve interação com o cliente por parte dos estudantes, mas mesmo assim a análise de requisitos foi discutida pelos membros da equipa e pelos supervisores que agiam como *product owner*.

Para os utilizadores poderem utilizar a aplicação será necessário fazer a autenticação de utilizadores. Depois de autenticado o utilizador será reencaminhado para a página principal. Na página principal é possível ver as simulações criadas. A barra de navegação varia dependendo das permissões do utilizador, mas duas constantes são o botão de *logout* e um botão para reencaminhar o utilizador para a página onde pode alterar os seus dados ou eliminar a sua conta.

Três dos possíveis botões da barra de navegação serão comuns aos clientes e aos funcionários. Estes são os botões para fazer um dos três tipos de simulações: obras, construção ou transferência.

Para fazer as simulações terão quatro páginas onde o utilizador insere os seus dados, uma quinta com o resumo da simulação. Em seguida o utilizador irá para a pré-análise onde poderá agendar a visita a uma agência. Depois de confirmar a hora e data da visita é reencaminhado para a página principal.

Os funcionários terão acesso a um botão adicional. Quando carrega neste botão será reencaminhado para uma página onde poderá ver, criar e eliminar agências. Só pode eliminar as que não têm registos associados na tabela das reuniões.

Os administradores terão acesso a um botão na barra de navegação que o reencaminhara para uma página onde poderão apagar utilizadores ou alterar as permissões dos utilizadores.

A Tabela 1 mostra os atores e os casos de uso.

Tabela 1 Descrição do projeto

Atores	**Casos de Uso**
Utilizador	• Criar utilizador • Editar as suas informações • Eliminar a própria conta • Criar simulações Obras, Transferência, Construção • Ver próprias simulações • Eliminar próprias simulações
Funcionário	• Criar utilizador • Editar as suas informações • Eliminar a própria conta • Criar simulações Obras, Transferência, Construção • Eliminar próprias simulações • Criar agências • Visualizar agências • Eliminar agências
Administrador	• Visualizar utilizadores • Editar Permissões de utilizadores • Eliminar utilizadores

Depois foi feito o estudo dos casos de uso. O que se segue é um exemplo de o estudo de um caso de uso.

A Tabela 2 descreve o caso de uso Editar permissões.

Tabela 2 Caso de uso "Editar permissões

Nome	Editar permissões
Descrição	Este case de uso permite a um administrador mudar as permissões dos utilizadores
Pré-condição	Login válido, permissões de administrador
Caminho principal	O utilizador clica no botão utilizadores E direcionado para uma página com algumas informações de todos os utilizadores. O utilizador altera as permissões que são necessárias alterar O sistema processa as alterações e guarda essas mesmas na base de dados O utilizador é redirecionado para uma página de sucesso
Caminhos alternativos	Se, em qualquer passo deste processo, ocorrer algum erro ou algo inesperado o utilizador é redirecionado para uma página de erro genérica

E por fim um diagrama de sequência correspondente ao caso de uso Editar permissões que pode ser visto na Figura 2.

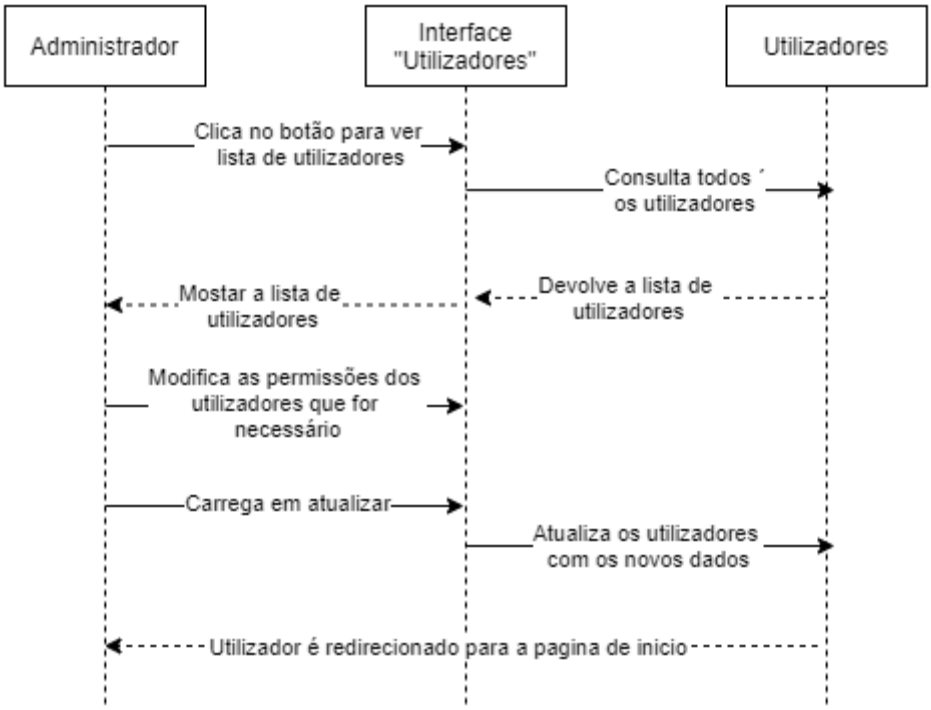

Figura 2 Diagrama de sequência

Foi ainda feito o diagrama de classes (Figura 3). O diagrama foi feito pelo Fernando Lopes, mas foi discutido pelos membros da equipa em conjunto.

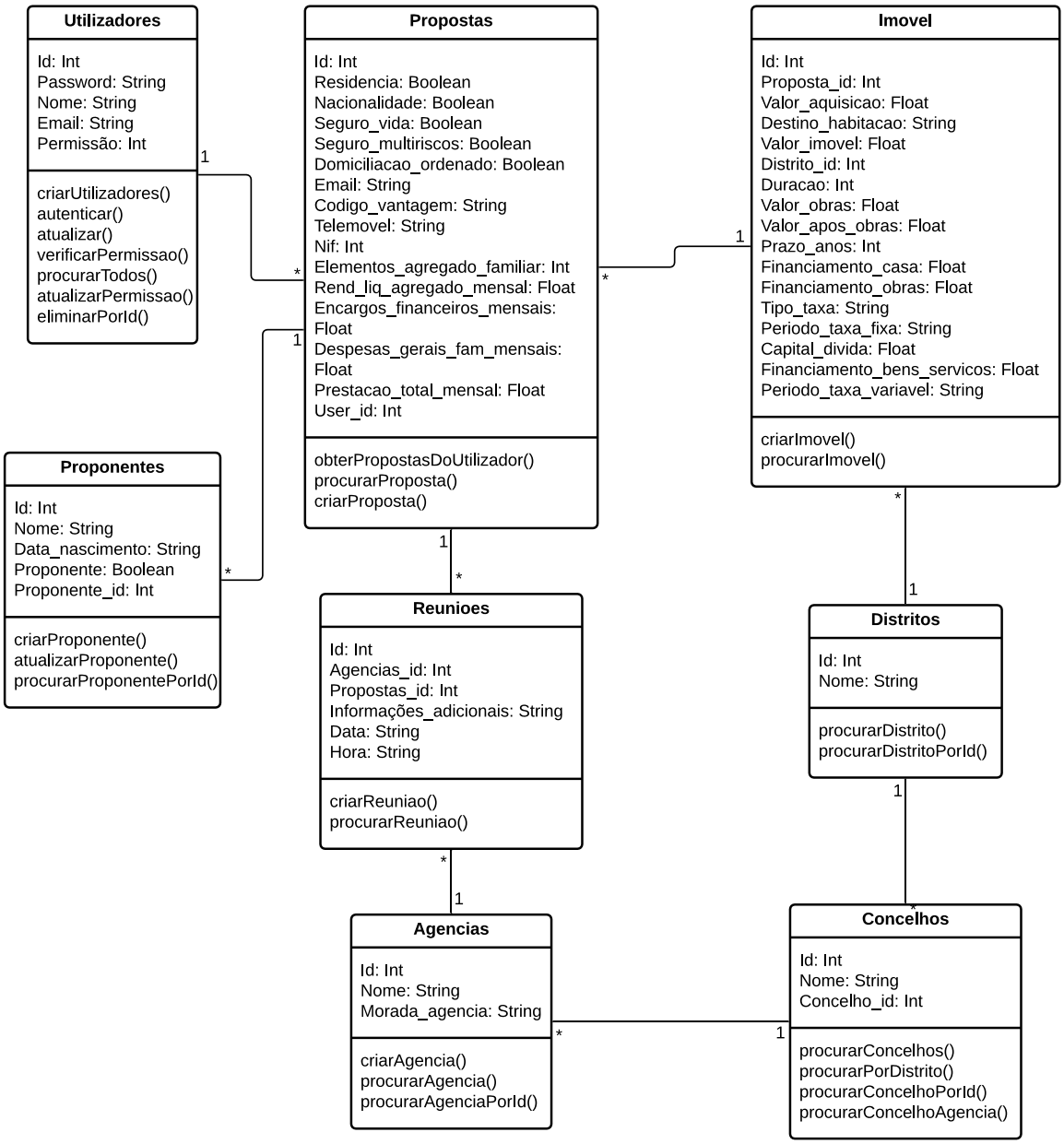

Figura 3 Diagrama de classes

Com base no diagrama de classes foi desenvolvido o modelo relacional (Figura 4).

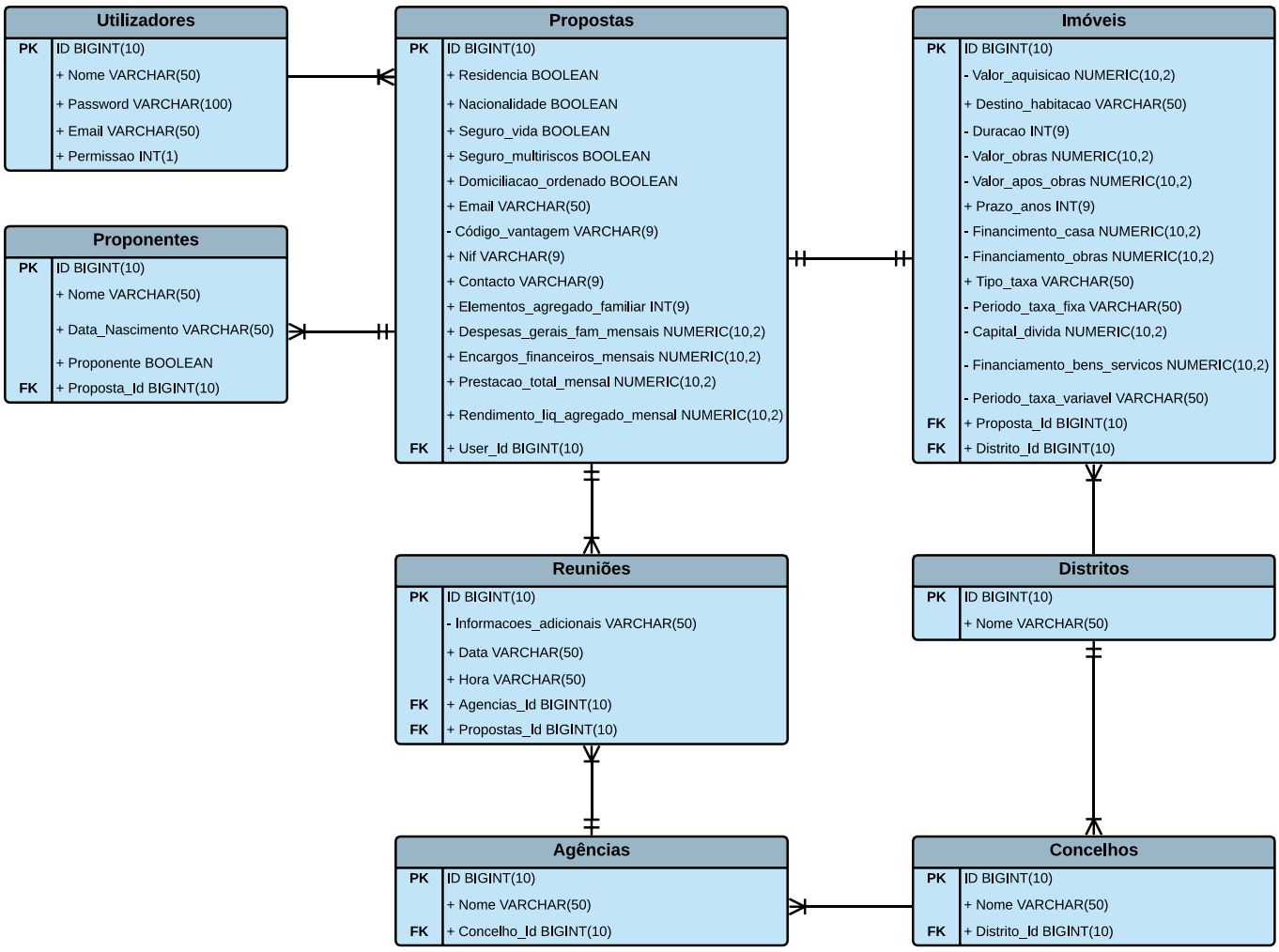

Figura 4 Modelo relacional

4 Ferramentas e desenvolvimento

Neste capítulo é feito um estudo sobre as ferramentas e tecnologias utilizadas ao longo do estágio e uma descrição do desenvolvimento da aplicação.

No início vai ser feita uma descrição das ferramentas e tecnologias utilizadas.

Depois é descrito o processo de conversão de *mockups* fornecidos pela Softinsa para HTML e CSS. Na altura em que isto foi feito ainda não tinha sido definido o projeto final e como tal não usamos Angular Material. O processo de adicionar Angular Material às páginas HTML é descrito na parte seguinte.

A seguir é feito uma breve descrição da arquitetura da aplicação de simulação do crédito de habitação. Antes de falar em mais detalhe sobre todos os aspetos envolvidos no desenvolvimento da aplicação.

4.1 Ferramentas e tecnologias

Este subcapítulo descreve as ferramentas e tecnologias utilizadas.

4.1.1 Angular

Angular é uma plataforma e *framework* para construir aplicações usando HTML e TypeScript, a linguagem em que o Angular é escrito. As funcionalidades do Angular são implementadas como um conjunto de bibliotecas TypeScript que são importadas no projeto (2).

O Angular tem acesso a uma biblioteca chamada Angular Material (3). Esta biblioteca tem um conjunto de componentes com o objetivo de construir páginas atrativas, consistentes e funcionais que aderem aos princípios de desenho modernos como portabilidade de navegador e independência de dispositivo (4). O Angular Material permite fazer isto com grande facilidade.

A escolha desta tecnologia foi exigência dos supervisores.

4.1.2 Node.Js

O Node.Js é um *runtime* JavaScript construído usando o motor v8 da Google (5). O Node.Js foi escolhido por permitir criar o servidor em JavaScript.

O Node.Js tem acesso ao express. O express é uma *framework* fácil de usar e flexível que contém um conjunto de funcionalidades robustas para desenvolver aplicações de *web* (6). Uma dessas funcionalidades é a criação básica e conveniente de um servidor web. É usado pois torna o processo de desenvolvimento mais fácil.

O Node.Js é usado em conjunto com o express para criar uma API que é usada pelo *frontend*.

4.1.3 PostgreSQL

PostgreSQL é um sistema gestor de base de dados objeto relacional (SGBD) *open source* (7). Esta tecnologia não foi escolhida pelos membros da equipa, foi uma exigência dos supervisores. Foi em PostgreSQL que foi desenvolvida a base de dados.

4.1.4 Sequelize

O Sequelize é um ORM (*Object–relational mapping*) do Node.Js baseado em promessas para o PostgreSQL e outros DBMS (*database management system*) (8).

ORM - serve para fazer *queries* a uma base de dados usando o paradigma da linguagem de programação de escolha (9).

Por exemplo a query:

```
1.  SELECT * FROM users WHERE email = 'test@test.com';
```

Usando um ORM genérico:

```
1.  var orm = require('generic-orm-libarry');
2.  var user = orm("users").where({ email: 'test@test.com' });
```

4.1.5 Microsoft Teams

O Microsoft teams é uma ferramenta que permite a colaboradores poderem conversar, através de texto ou voz e/ou vídeo e transferir ficheiros. É devido a essas funcionalidades e à pandemia durante a qual o estágio foi feito que o Teams foi muito útil, pois o estágio foi feito através de teletrabalho. Todas as reuniões foram feitas usando o Teams e também foi utilizado para partilhar documentos.

4.1.6 Azure DevOps

Azure DevOps é um serviço que fornece ferramentas de desenvolvimento e colaboração (10). Uma das várias ferramentas que fornece e a que foi usada neste projeto foi o controlo de versão. Foi usado pois é a ferramenta de controlo de versão usada na Softinsa.

4.1.7 Visual Studio code

O Visual Studio code (VSC) é um IDE (*Integrated development environment*) desenvolvido pela Microsoft. É leve e fácil de usar. Permite o desenvolvimento de aplicações em diversas linguagens.

4.2 Desenvolvimento

Este subcapítulo descreve o processo do desenvolvimento do projeto. Primeiro descreve o início do estágio antes do projeto ser definido e depois a implementação do projeto em si.

4.2.1 Conversão de *mockups* para HTML e CSS

Antes do projeto ter sido definido foram convertidos *mockups* em HTML/CSS. No total eram cerca 72 imagens das quais cerca de 32 foram convertidas por mim e as restantes pelo aluno Fernando Lopes. Estes *mockups* ilustravam como a aplicação deve ser a nível estético.

As figuras seguintes mostram um exemplo de um *mockup* (Figura 5) e a página HTML feita por mim (Figura 6).

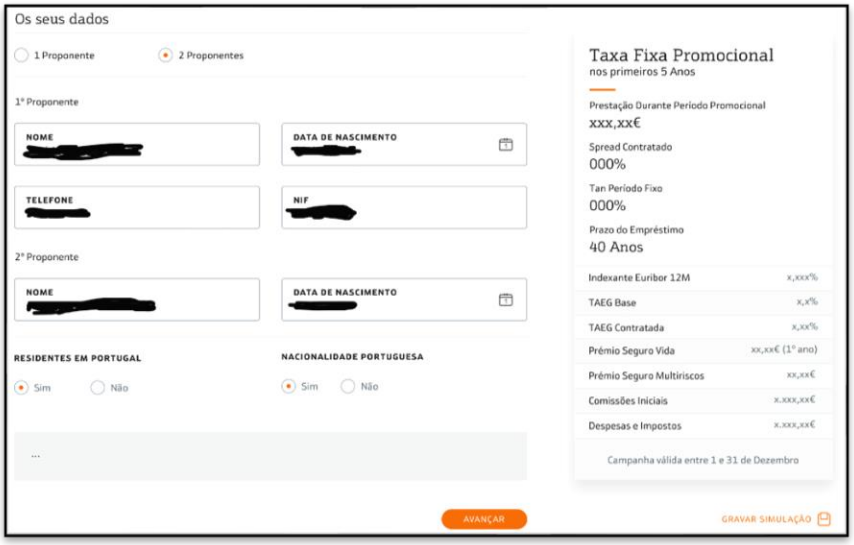

Figura 5 Exemplo de mockup

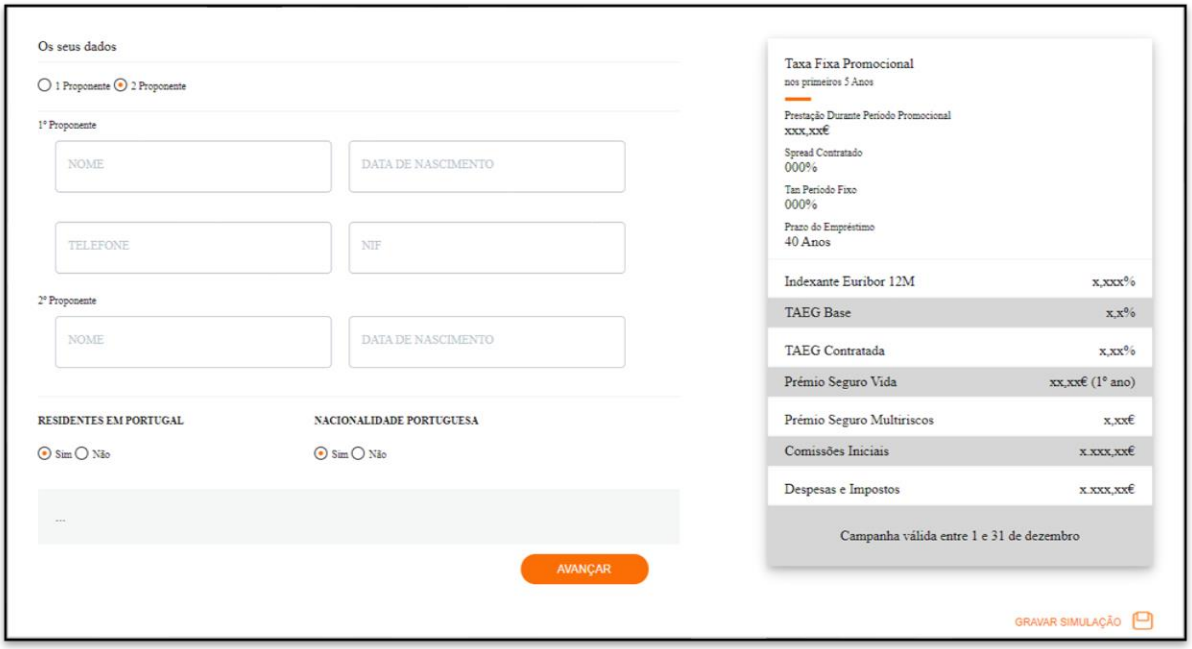

Figura 6 Mockup convertido em HTML + CSS

A conversão não é perfeita pois não tivemos acesso ao tamanho/tipo de letra ou às cores usadas no *mockup*. Mesmo assim o resultado em HTML foi semelhante ao *mockup*.

4.2.2 Adicionar Angular *material* ao código HTML

Como as páginas HTML criadas anteriormente não usavam Angular Material, pois quando foram criadas ainda não estava definido o projeto, foi adicionado quando começamos o projeto em Angular.

Em termos estéticos a diferença ao utilizar Angular Material foi mínima. Mas em termos do código a diferença foi muito maior. Pois o código é muito mais fácil de perceber.

A seguir está um exemplo de duas caixas de *input* com e sem Angular Material.

Código sem Angular Material:

HTML

```
<div class="form-div">
    <div class="floating-form">
        <div class="floating-input-div">
            <input class="floating-input" type="text" placeholder=" ">
            <span class="highlight"></span>
            <label>CONTACTO</label>
        </div>
    </div>

    <div class="floating-form">
        <div class="floating-input-div">
            <input class="floating-input" type="text" placeholder=" ">
            <span class="highlight"></span>
            <label>NIF</label>
         </div>
     </div>
</div>
```

CSS

```
.floating-form {
    float: left;
    width: 320px;
    margin-left: 20px;
}

.floating-input-div {
    position: relative;
    margin-bottom: 20px;
}

.floating-input {
    font-size: 14px;
    padding: 20px 4px 4px 14px;
    display: block;
    width: 100%;
    height: 60px;
    border: 1px solid #b2bcc6;
```

```
    border-radius: 5px;
    color: #b2bcc6
}

.floating-input-div label {
    color: #b2bcc6;
    font-size: 14px;
    font-weight: normal;
    position: absolute;
    pointer-events: none;
    left: 15px;
    top: 20px;
    transition: 0.2s ease all;
    -moz-transition: 0.2s ease all;
    -webkit-transition: 0.2s ease all;
}

.floating-input-div p {
    position: absolute;
    top: -55%;
    left: 30%;
    font-size: 12px;
    color: lightgray;
    width: fit-content;
}

.floating-input:focus~label,
.floating-input:not(:placeholder-shown)~label,
.floating-input-inv:focus~label,
.floating-input-inv:not(:placeholder-shown)~label {
    top: 15px;
    left: 15px;
    font-size: 14px;
    color: black;
}
```

Código com Angular Material:

HTML

```
<p>
    <mat-form-field appearance="outline">
        <mat-label>CONTACTO</mat-label>
        <input type="tel" matInput type="number">
    </mat-form-field>

    <mat-form-field appearance="outline">
        <mat-label>NIF</mat-label>
        <input matInput type="number">
    </mat-form-field>
</p>
```

CSS

```
.mat-form-field {
  width: 47.5%;
}
```

```
4.
5.  .mat-form-field~.mat-form-field {
6.    margin-left: 5%;
7.  }
```

Estas duas versões do código têm o mesmo resultado a nível estético, mas usando o Angular Material é muito mais fácil de perceber.

Isto foi feito para todas as páginas, não só com inputs, mas também com *selects*, *date pickers*, entre outros.

4.2.3 Arquitetura do Sistema

A aplicação está dividida em duas partes distintas *frontend* e *backend*, sendo o *frontend* a parte com qual o utilizador interage e o *backend* a parte manipula a base de dados. Neste projeto o *frontend* foi feito em Angular, o *backend* em Node.Js e a base de dados em PostgreSQL (Figura 7).

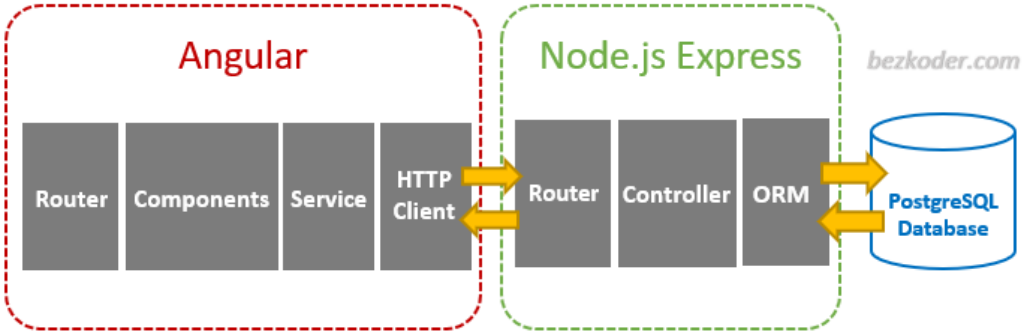

Figura 7 Arquitetura

https://bezkoder.com/Angular-10-node-express-postgresql/

No *frontend* o Angular pode ser dividido em quatro partes:

- O *router* é um serviço que escolhe qual componente mostrar usando o URL (*Uniform Resource Locator*).
- Os componentes controlam uma certa porção do ecrã chamado a *view*. Os componentes contêm lógica de aplicação, o que é feito para apoiar a *view*, mas esta deve ser mínima.
- Os serviços servem para fazer o resto da lógica necessária. Neste trabalho são usados para fazer os pedidos HTTP.
- O HTTPClient é uma biblioteca utilizada para fazer pedidos HTTP ao *backend*.

No *backend*:

- O *router* escolhe que qual função chamar dentro do respetivo controlador dependendo do pedido.

- O *controller* é um conjunto de funções. Neste trabalho usam o ORM para manipular a base de dados.
- O ORM permite fazer *queries* à base de dados facilmente.

4.2.4 Implementação

Aqui é descrito com mais detalhe a implementação da aplicação.

Primeiro é mostrada a estrutura das pastas da aplicação e a seguir é feita uma descrição de como funciona a aplicação seguindo a Figura 7 da direita para a esquerda.

Depois são explicados mais uns aspetos relevantes que não estão diretamente na figura da arquitetura.

4.2.4.1 Estrutura de pastas

A estrutura de pastas no *frontend* é criada automaticamente pelo Angular CLI (*Command-line interface*) ao iniciar um novo projeto.

No *backend* a estrutura que foi decidido usar foi a da Figura 8.

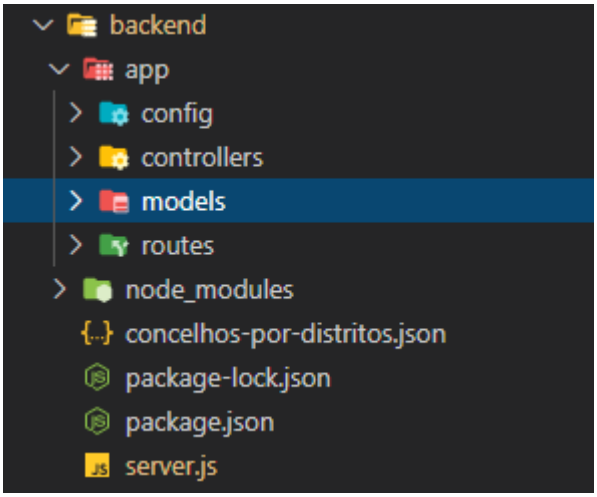

Figura 8 Estruturas backend

4.2.4.2 Base de dados e ORM

A base de dados foi criada em PostgreSQL e o ORM utilizado foi o Sequelize. O Sequelize é utilizado para configurar e manipular a base de dados de forma simples usando JavaScript.

Para exportar os parâmetros de configuração da conexão do PostgreSQL e do Sequelize foi criada a pasta *config* com um ficheiro chamado **db.config.js**. O código é o seguinte:

```
1.  module.exports = {
2.    HOST: "localhost",
3.    USER: "postgres",
4.    PASSWORD: "password",
5.    DB: "DB",
6.    dialect: "postgres"
7.  };
```

Cada tabela da base de dados tem um modelo Sequelize. São estes modelos que são utilizados para criar as tabelas. Cada tabela tem um modelo e estes modelos estão dentro da pasta *models*.

Os modelos criados são os seguintes:

- **agencia.model.js** – Modelo das agências.
- **concelho.model.js** – Modelo dos concelhos.
- **distrito.model.js** – Modelo dos distritos.
- **imovel.model.js** – Modelo dos imóveis.
- **proponente.model.js** – Modelo dos proponentes.
- **proposta.model.js** – Modelo das propostas.
- **reuniao.model.js** – Modelo das reuniões.
- **user.model.js** – Modelo dos utilizadores.

O código seguinte é o modelo **user.model.js**. Este é o que se distingue dos outros pois usa o Bcrypt para encriptar as palavras-chaves antes de criar um registo:

```
1.  const bcrypt = require("bcrypt");
2.  const saltRounds = 10;
3.
4.  module.exports = (sequelize, Sequelize) => {
5.      var User = sequelize.define("user", {
6.          password: {
7.              type: Sequelize.STRING(100),
8.              allowNull: false
9.          },
10.         email: {
11.             type: Sequelize.STRING(50),
12.             allowNull: false
13.         },
14.         nome: {
15.             type: Sequelize.STRING(50),
16.             allowNull: false
17.         },
18.         autorizado: {
19.             type: Sequelize.INTEGER,
20.             allowNull: false
21.         }
22.     });
23.
24.     User.beforeCreate(async (user, options) => {
25.         const hashedPassword = await bcrypt.hashSync(user.password, saltRounds);
26.         user.password = hashedPassword;
27.     });
28.
29.     return User;
30. };
```

Utilizando a função *beforeCreate* é possível alterar os dados antes de os inserir na base de dados, os restantes modelos não usam esta função. Aqui é feito o *hash* à palavra-chave do utilizador, isto faz a aplicação mais segura pois as palavras-chaves não são guardadas em texto simples. Para fazer isto é necessário definir os *saltRounds*. Os *saltRounds*

essencialmente controlam quanto tempo vai demorar a fazer o *hash*. Um número maior vai demorar mais tempo. Aqui o número escolhido foi 10.

O ficheiro **index.js** é utilizado para configurar a base de dados usando o Sequelize. Este ficheiro está dentro da pasta *models*.

Primeiro a base de dados é inicializada com os parâmetros exportados no ficheiro **db.config.js**:

```
const dbConfig = require("../config/db.config.js");
const Sequelize = require("sequelize");

const sequelize = new Sequelize(dbConfig.DB, dbConfig.USER, dbConfig.PASSWORD, {
    host: dbConfig.HOST,
    dialect: dbConfig.dialect,
    operatorsAliases: 0,

    pool: {
        max: dbConfig.pool.max,
        min: dbConfig.pool.min,
        acquire: dbConfig.pool.acquire,
        idle: dbConfig.pool.idle,
    },
});
```

Depois é preciso definir que modelos usar:

```
const db = {};

db.Sequelize = Sequelize;
db.sequelize = sequelize;

db.proponentes = require("./proponente.model.js")(sequelize, Sequelize);
db.propostas = require("./proposta.model.js")(sequelize, Sequelize);
db.imoveis = require("./imovel.model.js")(sequelize, Sequelize);
db.reunioes = require("./reuniao.model.js")(sequelize, Sequelize);
db.agencias = require("./agencia.model.js")(sequelize, Sequelize);
db.users = require("./user.model")(sequelize, Sequelize);
db.constantes = require("./constante.model")(sequelize, Sequelize);
db.distritos = require("./distrito.model")(sequelize, Sequelize);
db.concelhos = require("./concelho.model")(sequelize, Sequelize);
```

E finalmente definir as relações entre as tabelas:

```
db.propostas.hasMany(db.proponentes, {
    foreignKey: "propostumId",
    onDelete: "CASCADE"
});
db.proponentes.belongsTo(db.propostas);

db.propostas.hasMany(db.imoveis, {
    foreignKey: "propostumId",
    onDelete: "CASCADE"
});
db.imoveis.belongsTo(db.propostas);
```

```
12.
13. db.propostas.hasMany(db.reunioes, {
14.     foreignKey: "propostumId",
15.     onDelete: "CASCADE"
16. });
17. db.reunioes.belongsTo(db.propostas);
18.
19. db.distritos.hasMany(db.concelhos);
20. db.concelhos.belongsTo(db.distritos);
21.
22. db.distritos.hasMany(db.imoveis);
23. db.imoveis.belongsTo(db.distritos);
24.
25. db.concelhos.hasMany(db.agencias);
26. db.agencias.belongsTo(db.concelhos);
27.
28. db.agencias.hasMany(db.reunioes);
29. db.reunioes.belongsTo(db.agencias);
30.
31. db.users.hasMany(db.propostas, {
32.     foreignKey: 'userId',
33.     onDelete: 'CASCADE'
34. });
35. db.propostas.belongsTo(db.users);
```

No ficheiro **server.js** na pasta *backend* é necessário sincronizar a base de dados:

```
1.  db.sequelize.sync();
```

Ou caso seja necessário apagar os dados guardados e ressincronizar a base de dados:

```
1.  db.sequelize.sync({ force: true }).then(() => {
2.    console.log("Drop and re-sync db.");
3.  });
```

4.2.4.3 Controladores

Os controladores contêm as funções a serem executadas quando chamadas pela parte do router que é explicada no próximo subcapítulo. Neste caso as funções usam o Sequelize para manipular a base de dados.

Os controladores criados foram:

- **agencia.controller.js**
- **concelho.controller.js**
- **distrito.controller.js**
- **imovel.controller.js**
- **proponente.controller.js**
- **proposta.controller.js**
- **reuniao.controller.js**
- **user.controller.js**

Cada controlador contém as funções correspondentes do diagrama de classes (**Erro! A o rigem da referência não foi encontrada.**).

Como é no controlador do utilizador(**user.controller.js**) que é utilizado o sistema de *login* JWT, que é uma parte importante do projeto, é este mesmo controlador que vai ser utilizado como exemplo.

Primeiro é necessário importar os módulos necessários:

```
1.  const db = require("../models");
2.  const bcrypt = require("bcrypt");
3.  const jwt = require("jsonwebtoken");
4.  const express = require("express");
```

O bcrypt é uma biblioteca que serve para fazer *hash* às palavras-chaves e para depois verificar se a introduzida pelo utilizador é a correta.

O jsonwebtoken é uma implementação de *JSON web token*. JWT define uma maneira de transmitir informação no formato JSON em segurança. A informação pode verificada pois é digitalmente assinada (11).

A função *autenticarUtilizador* é a função utilizada quando um utilizador quer fazer *login*. Como para fazer *login* é o utilizador têm de estar registado na base de dados é necessário verificar se existe um utilizador com o email que vem no pedido:

```
1.  //Verificar se o utilizador com o email introduzido existe
2.  const utilizador = await User.findOne({
3.      where: {
4.          email: email
5.      }
6.  });
```

Se o utilizador não existir retornar um erro:

```
1.  res.status(403).send({ mensagem: msg });
```

Se o utilizador existir é necessário verificar se a palavra-chave esta correta. Para isso é utilizado o Bcrypt, pois as palavras chaves na base de dados estão encriptadas:

```
1.  if (bcrypt.compareSync(password, utilizador.password)) {
2.      /*
3.      A palavra-chave introduzida é a correta.
4.      Criar o token de autenticação.
5.      O token tem validade de 1 hora.
6.      O token leva codificado informações úteis e pouco sensíveis como o id e o
7.      nível de permissão.
8.      */
9.      const token = jwt.sign(
10.         {
11.             id: utilizador.id,
12.             autorizado: utilizador.autorizado
13.         },
14.         app.get("secretKey"),
```

```
        {
            expiresIn: "1h",
        }
    );

    /**
     * Enviar para o cliente as informações necessárias com o código 200.
     */
    res.status(200).json({
        token: token,
        expiresIn: "3600",
        id: utilizador.id,
        nome: utilizador.nome,
        autorizado: utilizador.autorizado
    });
} else {
    res.status(401).send({
        mensagem: msg
    });
}
```

A maioria das funções tanto neste controlador como nos outros seguem a mesma estrutura:

1. Verificar se o *token* é valido;
2. Executar uma operação na base de dados;
3. Retomar uma mensagem de sucesso/erro ou os dados relevantes.

Como pode ser visto na função *eliminarPorId*:

```
eliminarPorId: async function (req, res) {
    let id = req.params.id;
    let token = req.params.token;
    //verificar se o token dado é valido
    jwt.verify(token, app.get("secretKey"), async function (err, decoded) {
        if (err) {
            res.status(401).send({
                mensagem: "O token não é valido..."
            });
        } else {
            //Verificar se é o proprio utilizador ou um administrador a apagar a conta
            if (id == decoded.id || decoded.autorizado === 0) {
                User.destroy({
                    where: { id: id }
                }).then(_ => {
                    res.status(200).send({
                        mensagem: "Apagado!! ID - " + id
                    });
                }).catch(err => {
                    res.status(500).send({
                        mensagem: err.message || "Ocorreu algum erro a eliminar o utilizador."
                    })
                });
            } else {
                res.status(500).send({
                    mensagem: "Ocorreu algum erro a eliminar o utilizador."
```

```
                })
            }
        }
    });
}
```

A função *verify* do JWT usa a chave secreta para verificar se o *token* é valido e executa a *callback* onde retorna o estado 200 ou um erro. Se não ocorrer erro nenhum vai apagar a conta da base de dados.

4.2.4.4 Router(backend)

Quando é recebido um pedido HTTP é necessário determinar como responder isso é feito através das rotas. As rotas são definidas nos ficheiros da pasta *routes*:

- **agencia.routes.js**
- **concelho.routes.js**
- **distrito.routes.js**
- **imovel.routes.js**
- **proponente.routes.js**
- **proposta.routes.js**
- **reuniao.routes.js**
- **user.routes.js**

Estes ficheiros são todos muito semelhantes e simplesmente muda o controlador utilizado e as rotas em si. Aqui está o exemplo do **user.routes.js**:

```
module.exports = (app) => {
    const users = require("../controllers/user.controller");

    var router = require("express").Router();

    //Rota para criar um novo utilizador
    router.post("/criarUtilizador", users.criarUtilizador);

    //Rota para autenticar o utilizador
    router.post("/autenticarUtilizador", users.autenticarUtilizador);

    //Rota para fazer update ao utilizador
    router.post("/atualizar", users.atualizar);

    //Rota para ir buscar o nível de permissões de um utilizador
    router.get('/verificarPermissao/token=:token', users.verificarPermissao);

    //Rota para ir buscar todos os utilizadores
    router.get("/token=:token", users.procurarTodos);

    //Rota para atualizar permissões
    router.put("/atualizarPermissoes", users.atualizarPermissoes);

    //Rota para eliminar utilizadores usando um id
    router.delete('/eliminar/id=:id&token=:token', users.eliminarPorId);

    app.use("/api/utilizadores", router);
```

```
28. };
```

As rotas são definidas usando o router e chamando a função *post*, *get*, *delete* ou *put* e passar o padrão do URL e a função do controlador a usar. Se for preciso passar parâmetros no URL é feito usando, por exemplo, "*:id"* para passar o id, como se pode ver na linha 25.

No ficheiro **server.js** é necessário ainda importar as rotas:

```
1.  require("./app/routes/proponente.routes")(app);
2.  require("./app/routes/proposta.routes")(app);
3.  require("./app/routes/imovel.routes")(app);
4.  require("./app/routes/reuniao.routes")(app);
5.  require("./app/routes/agencia.routes")(app);
6.  require("./app/routes/user.routes")(app);
7.  require("./app/routes/constante.routes")(app);
8.  require("./app/routes/distrito.routes")(app);
9.  require("./app/routes/concelho.routes")(app);
```

4.2.4.5 Serviços e HTTPClient

Os serviços vão fazer pedidos HTTP ao *backend*.

Os serviços podem ser criados automaticamente utilizando o CLI, mas é necessário criar os métodos que esse serviço vai precisar. Como exemplo vai ser mostrado o serviço dos utilizadores.

Este serviço vai fazer ser responsável por comunicar com o backend. Para tal é necessário importar alguns módulos.

```
1.  import { HttpClient, HttpHeaders } from '@Angular/common/http';
2.  import { Observable } from 'rxjs';
```

A biblioteca *HttpClient* faz pedidos *Http*. Como os métodos desta biblioteca retornam observables também é necessário ter como os poder manipular.

Observable é umas das classes principais da biblioteca *rxjs* e os métodos da biblioteca *HttpClient* retornam *observables*.

Os *observables* servem essencialmente para ir buscar dados assincronamente, são semelhantes às promessas de JavaScript com algumas diferenças (12).

Depois de importar o *HttpClient* é necessário injetar no serviço usando o construtor.

```
1.  constructor(private http: HttpClient) { }
```

Como grande parte do URL vai ser igual para todos os métodos deste serviço.

```
1.  private backendUrl = 'http://localhost:8080/api/utilizadores';
```

Com estas preparações feitas é possível criar um método que faz pedidos à API. Estes métodos são muito semelhantes entre si, mas, regra geral, por cada função no controlador há um método no serviço.

```
autenticarUtilizador(email: string, password: string): Observable < any > {
    return this.http.post(`${this.backendUrl}/autenticarUtilizador`, {
        email: email,
        password: password,
    }, this.httpOptions).pipe(
        tap(dados => {
            this.setSession(dados);
        })
    );
}
```

Este método recebe o email e a password como parâmetros e faz um pedido Http post à API e retorna um *observable*.

Para poder aceder ao resultado sem fazer alterações é possível usar a função *tap*. Aqui é usado para poder guardar alguns dados na sessão para poderem ser acedidos facilmente quando necessário.

```
private setSession(dados) {
    const expiresAt = moment().add(dados.expiresIn, 's');
    localStorage.setItem('id_token', dados.token);
    localStorage.setItem('id', dados.id);
    localStorage.setItem('nome', dados.nome);
    localStorage.setItem("expires_at", JSON.stringify(expiresAt.valueOf()));
}
```

A biblioteca *moment* é usada para analisar, manipular e validar datas em JavaScript. Outros métodos semelhantes são utilizados para atualizar ou apagar estes valores caso o utilizador altere estas informações ou faça *logout*.

Também foram criadas algumas funções para aceder, facilmente, a estes valores.

```
getExpiration() {
    const expiration = localStorage.getItem("expires_at");
    const expiresAt = JSON.parse(expiration);
    return moment(expiresAt);
}
```

Outra função criada neste serviço foi uma para saber se o utilizador tem o *login* feito. Como os *tokens* JWT expiram depois de algum tempo uma boa forma de saber se o utilizador tem o login feito ou se tem de voltar a fazer *login* é verificar se a data em que o *token* expira é depois da data atual. Para fazer isso foi usada a biblioteca *moment* pois esta torna isto muito simples de fazer.

```
isLoggedIn() {
    return moment().isBefore(this.getExpiration());
}
```

4.2.4.6 Componente

Os componentes são os blocos da *interface* mais básicos estão divididos por um *template* HTML uma classe TypeScript que permite fazer alguma lógica de aplicação para apoiar o *template* HTML.

Estes componentes não devem ter lógica muito pesada. É para isso que servem os serviços. Para poder usar o serviço é necessário ser importado no componente correto:

```
1.  import { UserService } from '../user.service';
```

E injetar o serviço através do construtor:

```
1.  constructor(private userService: UserService) { }
```

Neste caso para fazer *login* foi criado um método:

```
1.  autenticarUtilizador() {
2.      if (this.password.length > 0 && this.email.length > 0) {
3.          this.userService.autenticarUtilizador(this.email, this.password)
4.              .subscribe(
5.                  res => { },
6.                  err => {
7.                      this.mensagem = err.error.mensagem;
8.                  },
9.                  () => {
10.                     this.password = null;
11.                     this.router.navigateByUrl('/home');
12.                 }
13.             );
14.     } else {
15.         this.mensagem = 'Introduz um email e uma palavra-chave...';
16.     }
17. }
```

Este método usa o método do serviço *user* para fazer *login*. Como usa um observable a função é assíncrona como tal o resultado pode chegar agora ou alguns minutos depois. O método *subscribe* espera pelo resultado. Este método leva como parâmetro um *observer*. Um *observer* é um objeto que define métodos *callback* para processar 3 tipos de notificações que um *observable* pode mandar.

- *next* (no código está como res) – Obrigatório. É executado todas as vezes (0 ou mais vezes).
- *error* (no código está como err) – Opcional. É executado no caso de erro. A execução do *observable* é parada.
- *complete* (no código está representado por parenteses vazios) – Opcional. É executado quando recebe a notificação de completamento. Quando a execução é completada.

Aqui quando do erro guarda a mensagem de erro na variável mensagem que é mostrada no ecrã. E quando acaba apaga o valor da variável *password* vai para a página *home*, como a notificação de erro para a execução a *callback* complete não é executada.

Para o utilizador introduzir os dados é usado HTML + CSS:

```
<mat-form-field appearance="outline">
    <mat-label>Email:</mat-label>
    <input matInput [(ngModel)]="email" name="email" type="text">
</mat-form-field>

<mat-form-field appearance="outline">
    <mat-label>Password:</mat-label>
    <input matInput [(ngModel)]="password" name="password" type="password">
</mat-form-field>
```

O *[(ngModel)]* serve para fazer *two way binding* o que significa que o HTML afeta o valor no TypeScrypt e vice-versa. Desta forma o valor introduzido nos *inputs* é passado para as variáveis.

Para correr o método autenticarUtilizador é usado um botão:

```
<button mat-raised-button type="submit"
(click)="autenticarUtilizador()">Login</button>
```

Ao carregar no botão o método é corrido.

4.2.4.7 Router(frontend)

O *router* permite navegar interpretando o URL como uma instrução para mudar o que é mostrado no ecrã. Para isto é usado um ficheiro criado automaticamente pelo CLI chamado **app-routing.module.ts**.

Para criar uma rota basta importar o componente:

```
import { LoginComponent } from './login/login.component';
```

Depois no *array routes* desse mesmo ficheiro é definido o padrão da rota e o componente a usar. Só os componentes com rotas definidas desta forma é que são acessíveis na aplicação:

```
{ path: 'login', component: LoginComponent }
```

4.2.4.8 Ficheiro agências

Para popular as tabelas concelhos e distritos foi decidido usar um ficheiro com esses dados que foi encontrado na internet (13). Nas fontes oficiais não existe, ou pelo menos não foi encontrado, um ficheiro com a informação necessária toda. O ficheiro utilizado foi validado antes de usar.

Os dados vêm no seguinte formato:

```
{
    "level": 1,
    "code": 1,
    "name": "Aveiro"
}
```

Se o *level* for 1 é distrito, se for 2 é concelho e se for 3 é freguesia. As freguesias não são necessárias para o projeto.

Para criar um ficheiro com a informação num melhor formato foi feito um *script* em python.

Para começar foi necessário importar a biblioteca necessária:

```
import json
```

A biblioteca json serve para criar e manipular objetos JSON.

Depois foi necessário criar uma variável com os dados:

```
jsonFile = json.load(open('distritos-concelhos-
freguesias.json', "r", encoding="utf8"))
```

A função *open*, neste caso leva três parâmetros. O primeiro é o ficheiro. O segundo é o modo neste caso só é necessário ler por isso foi usado o modo de leitura. E o terceiro é o tipo de codificação que foi selecionada UTF-8.

O ficheiro original vem organizado de forma a que primeiro venha o distrito. Depois vêm os concelhos desse distrito e a seguir a cada concelho vêm as freguesias desse distrito. Depois vem o distrito seguinte.

Por isso é possível simplesmente usar um ciclo para percorrer o JSON e tirar as informações necessárias:

```
distritos = []
aux = {}
flag = 0

for loc in jsonFile[:]:
    if loc['level'] == 1:
        if flag == 1:
            distritos.append(aux)
        aux = {'nome': loc['name'], 'concelhos': []}
        flag = 1

    if loc['level'] == 2:
        aux['concelhos'].append({'nome': loc['name']})
```

Como as freguesias não são precisas quando o nível é 3 não é preciso fazer nada.

Depois de ter uma variável com os dados relevantes, é uma questão de guardar os dados num ficheiro novo:

```
with open("concelhos-por-distritos.json", "w", encoding='utf8') as outfile:
    json.dump(distritos, outfile, ensure_ascii=False)
```

Agora é necessário carregar estes dados na base de dados. Para isso no ficheiro server.js do backend é preciso importar o ficheiro com os dados:

```
const distritos = require("./concelhos-por-distritos.json");
```

E para inserir os dados foi criada a seguinte função:

```
function inserirDistritos() {
    completed = false;
    return new Promise(function (resolve, reject) {
        for (let i = 0; i < distritos.length; i++) {
            Distritos.create({ nome: distritos[i].nome })
                .then(dados => {
                    for (let j = 0; j < distritos[i].concelhos.length; j++) {
                        Concelhos.create({
                            nome: distritos[i].concelhos[j].nome,
                            distritoId: dados.id
                        }).then(_ => { resolve(); }).catch(() => completed = false
)
                    }
                }).catch(() => completed = false)
        }
        if (!completed) {
            reject();
        }
    });
}
```

A função cria uma promessa e tenta inserir todos os distritos e concelhos se ocorrer algum erro chama *reject*() se não chama *resolve*(). Para popular as tabelas distritos e concelhos basta chamar a função.

4.2.4.9 Permissões

Nesta aplicação foi decidido usar um sistema simples de permissões. Cada conta tem um nível de permissão se o nível for 0 é um administrador, se for 1 é um funcionário e se for 2 é um cliente.

Por razões de segurança foi necessário limitar quem tem acesso a certas partes. Um ficheiro chamado Usando o CLI pode ser criado uma interface de *CanActivate*. Esta interface é utilizada para proteger as rotas:

```
import { Injectable } from '@Angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree } from
'@Angular/router';
```

```
3.  import { Observable } from 'rxjs';
4.
5.  @Injectable({
6.    providedIn: 'root'
7.  })
8.  export class AdminGuard implements CanActivate {
9.    canActivate(
10.     next: ActivatedRouteSnapshot,
11.     state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean |
     UrlTree> | boolean | UrlTree {
12.     return true;
13.   }
14.
15. }
```

Se o *CanActivate* retornar verdadeiro então deixa continuar a navegação se retornar falso cancela a navegação.

Para definir quais as rotas protegidas por isto é preciso modificar a rota no *app-routing*., mas primeiro é preciso importar o *AdminGuard*.

```
1.  import { AdminGuard } from './admin/admin.guard';
```

Depois é necessário adicionar o seguinte código nas rotas que se querem proteger:

```
1.  {
2.      path: 'home', component: HomepageComponent, canActivate: [AdminGuard],
3.      data: {
4.          permitido: [0, 1, 2]
5.      }
6.  }
```

O campo *CanActivate* é usado para definir qual ficheiro usar para proteger a rota e o campo *data* é usado para passar dados para a função *CanActivate*. Os dados passados para a função neste caso são os níveis de permissão que devem ter acesso à página.

```
1.  async canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): Promise<boolean
    > {
2.      if (this.userService.isLoggedIn()) {
3.          await this.userService.verificarPermissao().toPromise().then(dados => {
4.              this.autorizado = dados.autorizado;
5.          });
6.
7.          const permitido: number[] = next.data.permitido;
8.
9.          if (this.autorizado == permitido[0] || this.autorizado == permitido[1] || this.autor
    izado == permitido[2]) {
10.             return true
11.         }
12.         this.router.navigateByUrl('/home');
13.         return false;
14.     }
15.     else {
16.         this.router.navigateByUrl('/login');
17.         return false;
```

```
18.     }
19. }
```

Nesta nova versão primeiro verifica se o utilizador tem o login feito e se não tiver redireciona o utilizador para a página de login. Depois, se o utilizador tiver o *login* feito, vai verificar se a permissão do utilizador é uma das passadas no objeto data. Se tiver permissão retorna verdadeiro e deixa prosseguir, se não redireciona o utilizador para a página inicial.

5 Interface

Neste capítulo é ser mostrada, através de imagens, a aplicação em funcionamento.

Página de login pode ser vista na Figura 9.

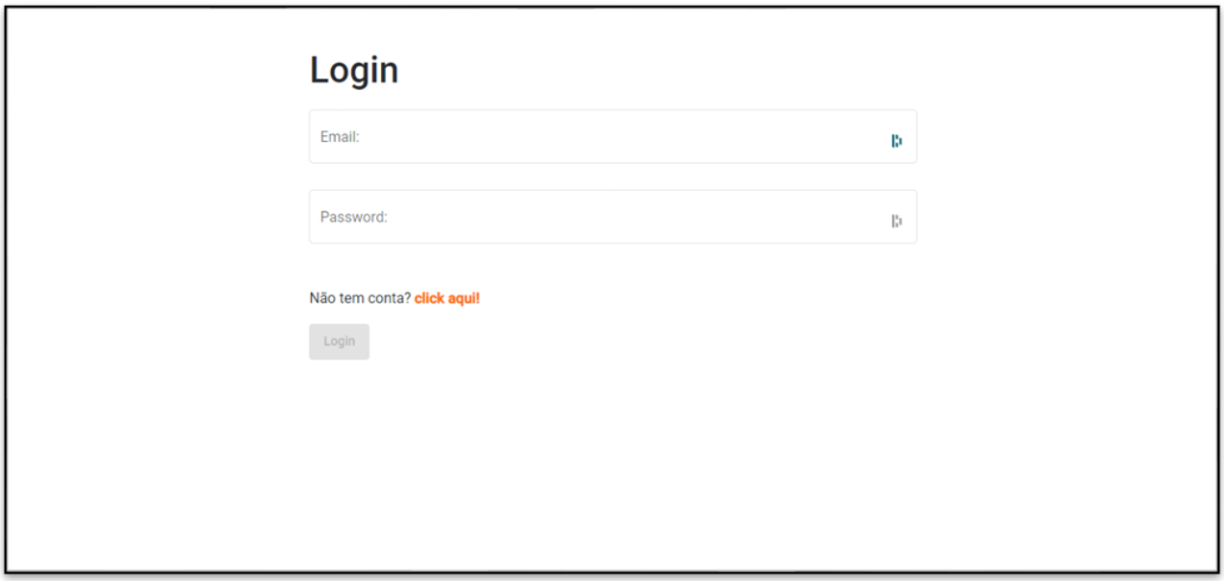

Figura 9 Interface Login

Quando o utilizado carrega em “click aqui!” o formulário muda para o formulário de registo, Figura 10.

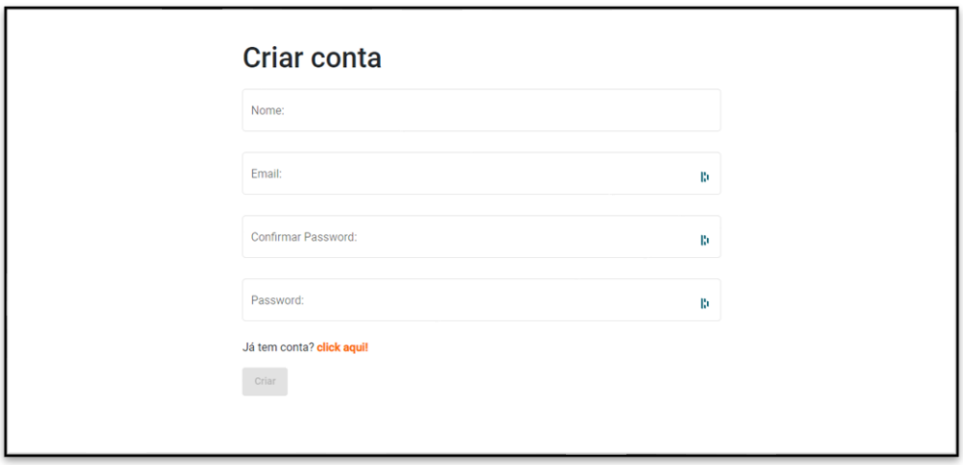

Figura 10 Interface de registo

Quando o utilizador faz o registo com sucesso volta para o formulário de *login*.

Quando faz *login* com sucesso vai para a página principal, Figura 11.

Figura 11 Interface da página inicial

Na página principal são apresentadas as propostas feitas por este utilizador (se tiver alguma feita).

Ao carregar em cima de uma proposta essa proposta expande para mostrar mais informações, Figura 12.

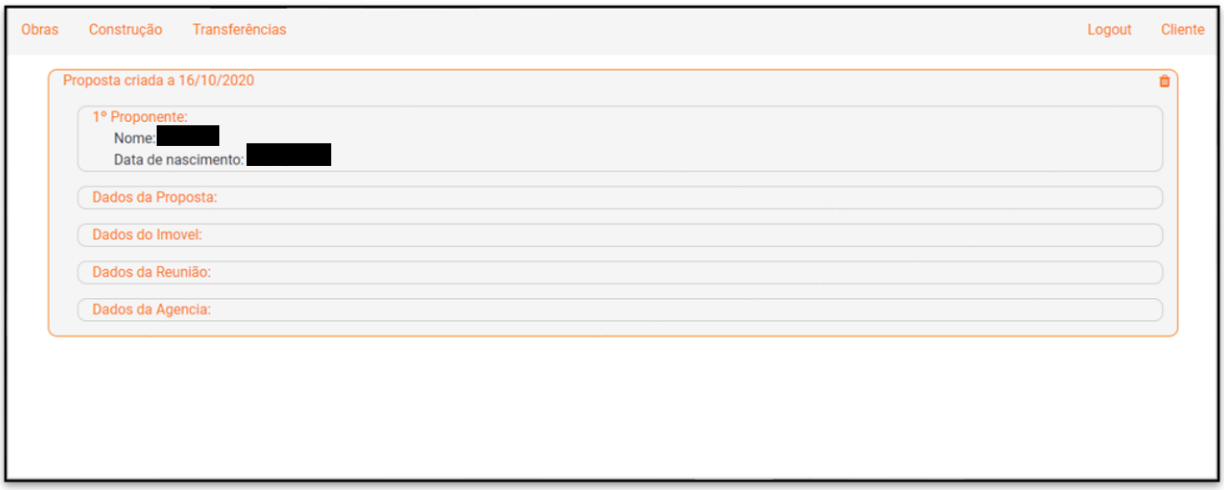

Figura 12 Interface da página inicial – campo "1º proponente" expandido

Cada uma das secções expande para mostrar as informações desse campo, Figura 13.

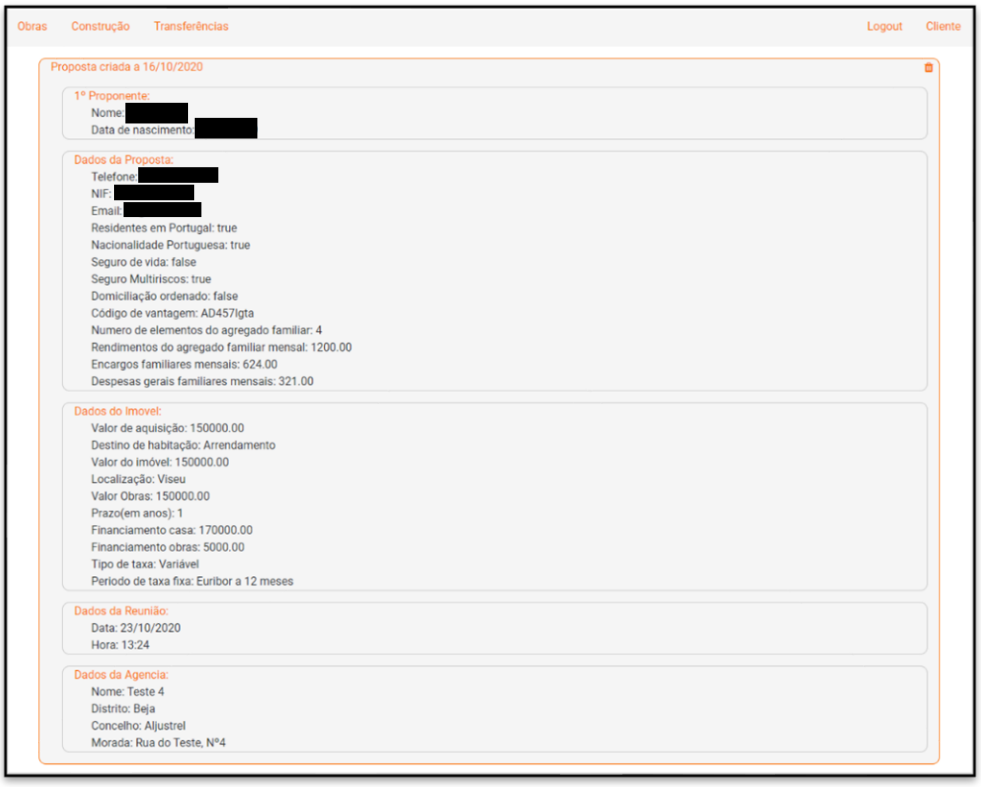

Figura 13 Interface da página inicial – todos os campos expandidos

A parte da esquerda da barra de navegação varia dependendo das permissões do utilizador. Um cliente só pode fazer as simulações, do qual há três tipos (obras, construção e transferência), um funcionário pode fazer as simulações e gerir as agências e um administrador pode gerir as permissões dos utilizadores e apagar os utilizadores.

Quando o utilizador carrega em *logout* vai para a página de *login*.

Quando carrega no seu nome vai para uma página de gestão da sua conta onde pode alterar as suas informações ou apagar a sua conta, Figura 14.

Figura 14 Interface editar conta

Todos os tipos de simulações são semelhantes, mas com alguns campos diferentes. Ao carregar em "Obras" na página inicial, o utilizador é redirecionado para a primeira página da simulação das obras, Figura 15.

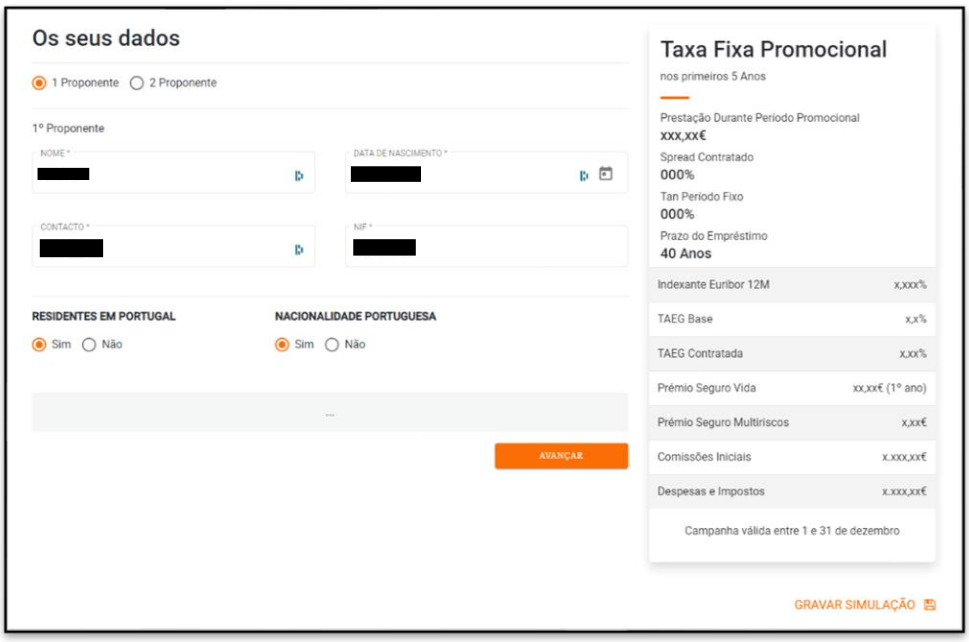

Figura 15 Interface simulação obras – dados dos proponentes

Depois de preenchido corretamente é possível carregar avançar e ir para a página seguinte.

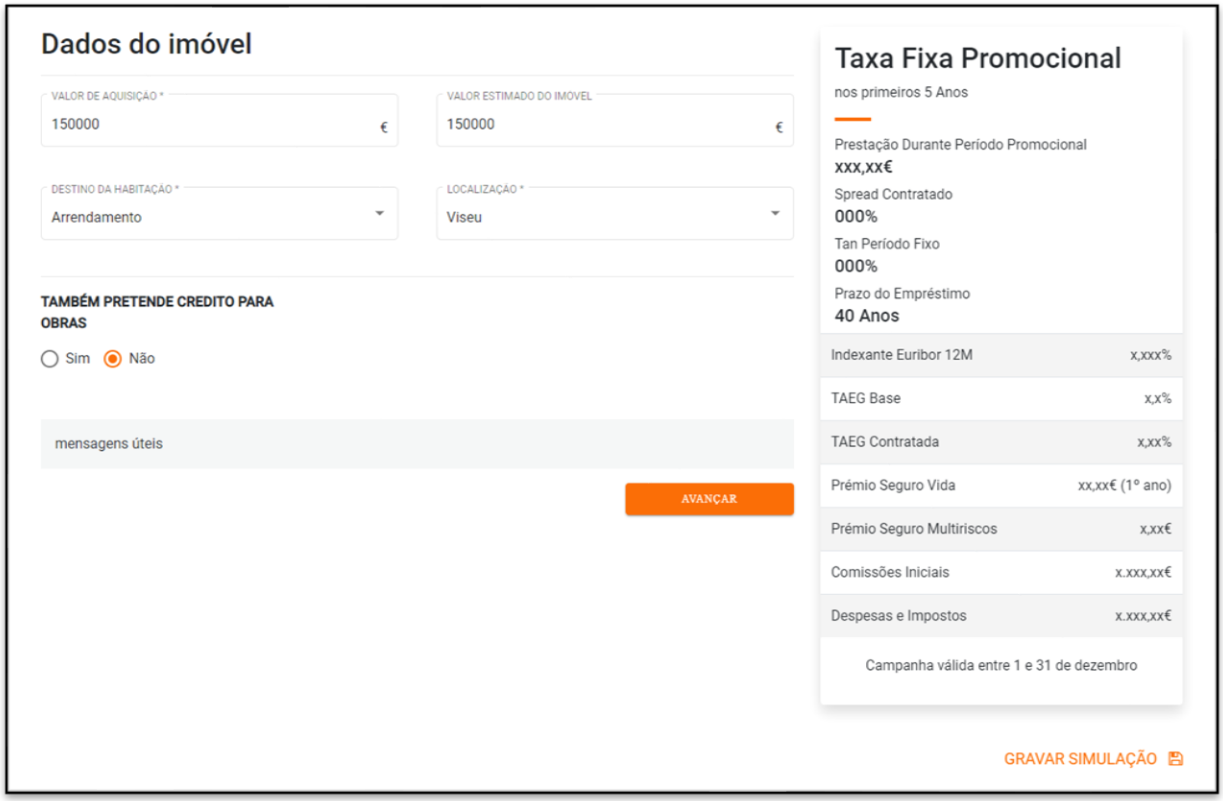

Figura 16 Interface simulação obras - dados dos imóveis

E assim sucessivamente, Figura 17 e Figura 18.

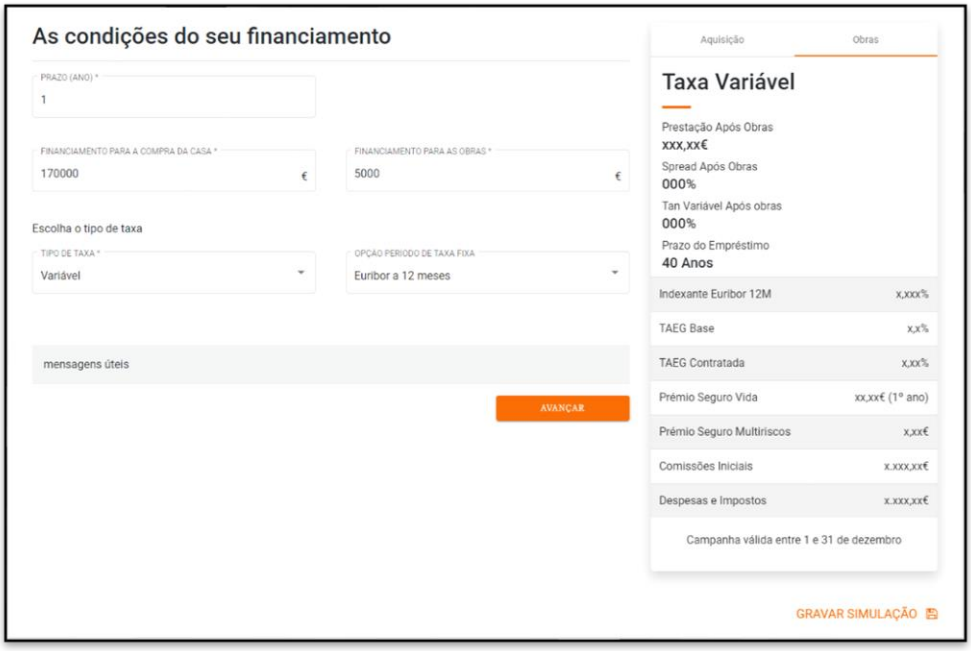

Figura 17 Interface simulação obras – condições do financiamento

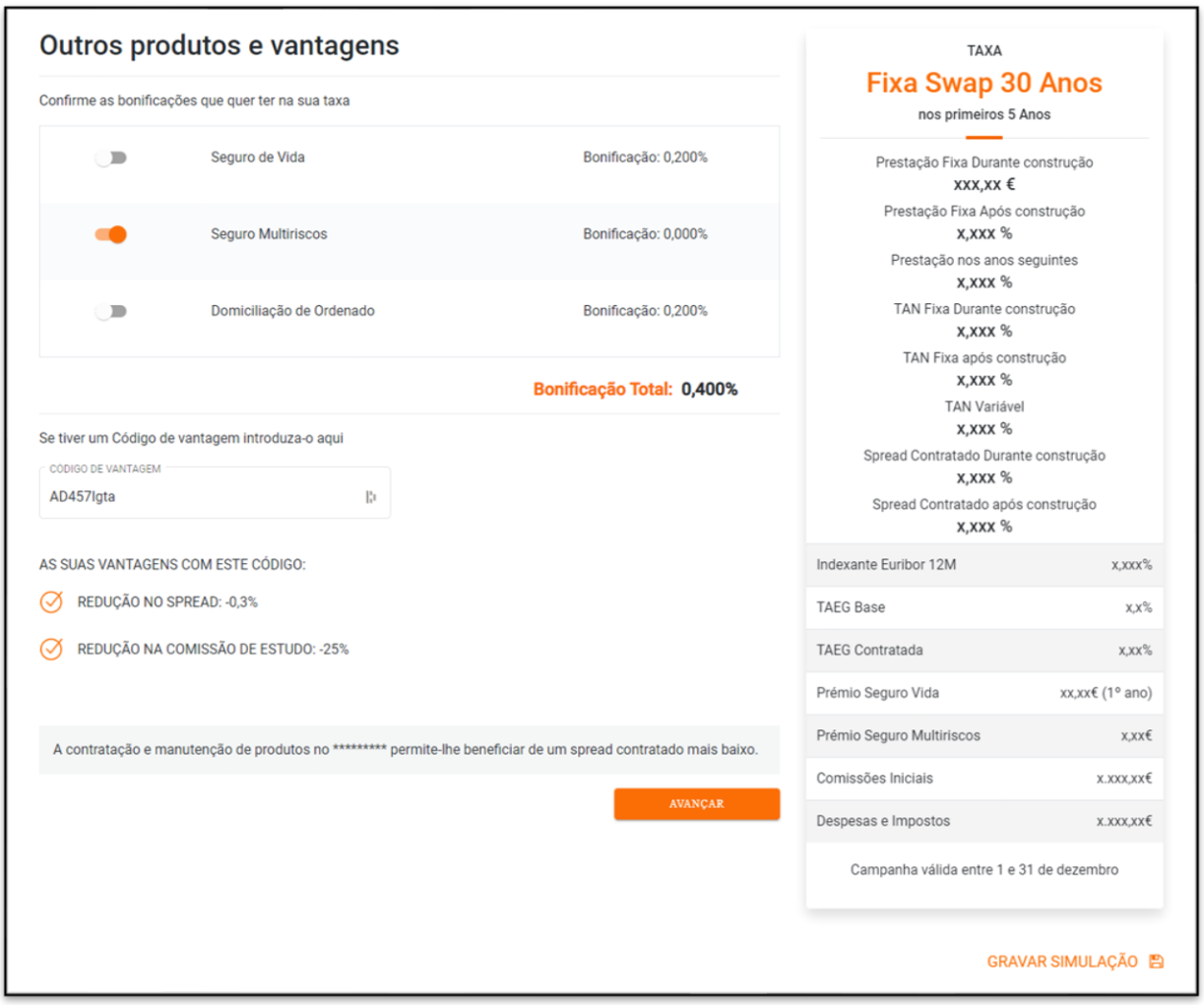

Figura 18 Interface simulação obras – outros produtos e vantagens

A Figura 19 mostra um resumo da simulação. Alguns dos dados da página são substituídos pelos dados a que temos acesso. Para avançar o utilizador carrega em "Avançar para a pré-análise".

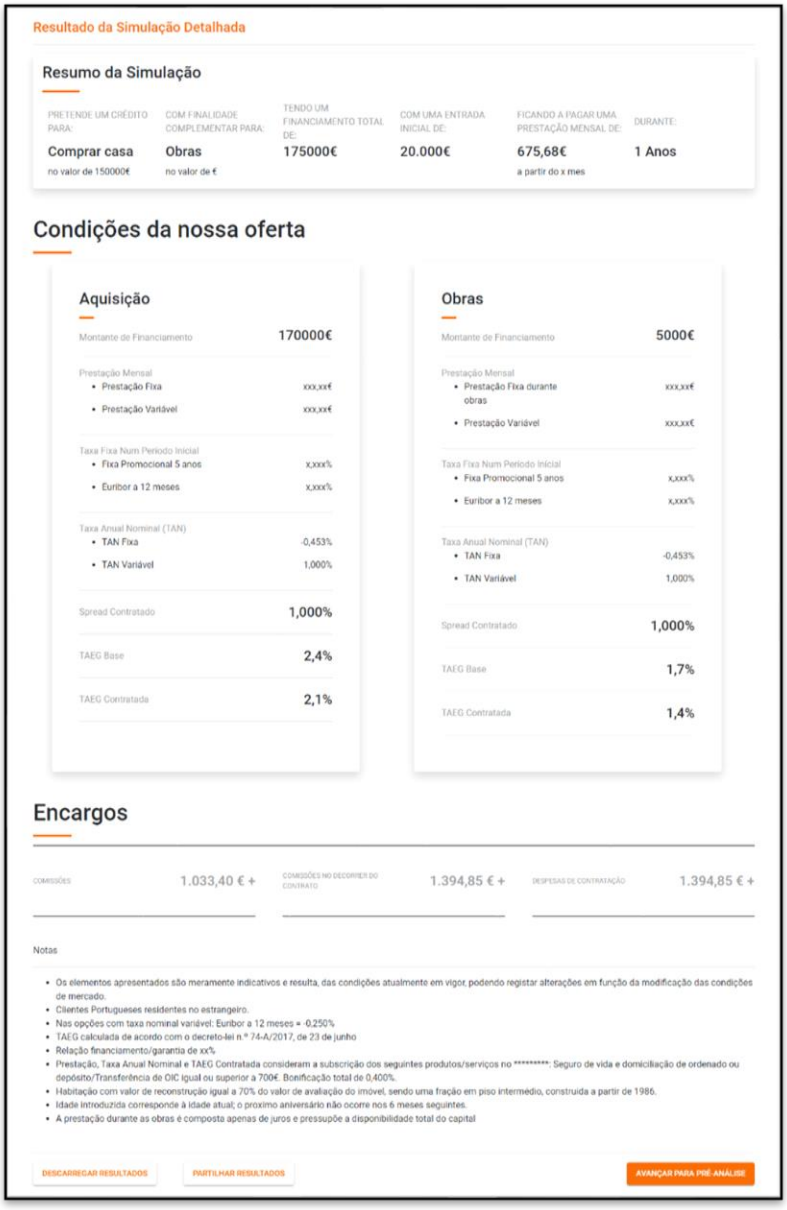

Figura 19 Interface simulação obras – resumo da simulação

Na figura 28 são introduzidos mais alguns dados e para avançar o utilizador carrega em "estou interessado", Figura 20.

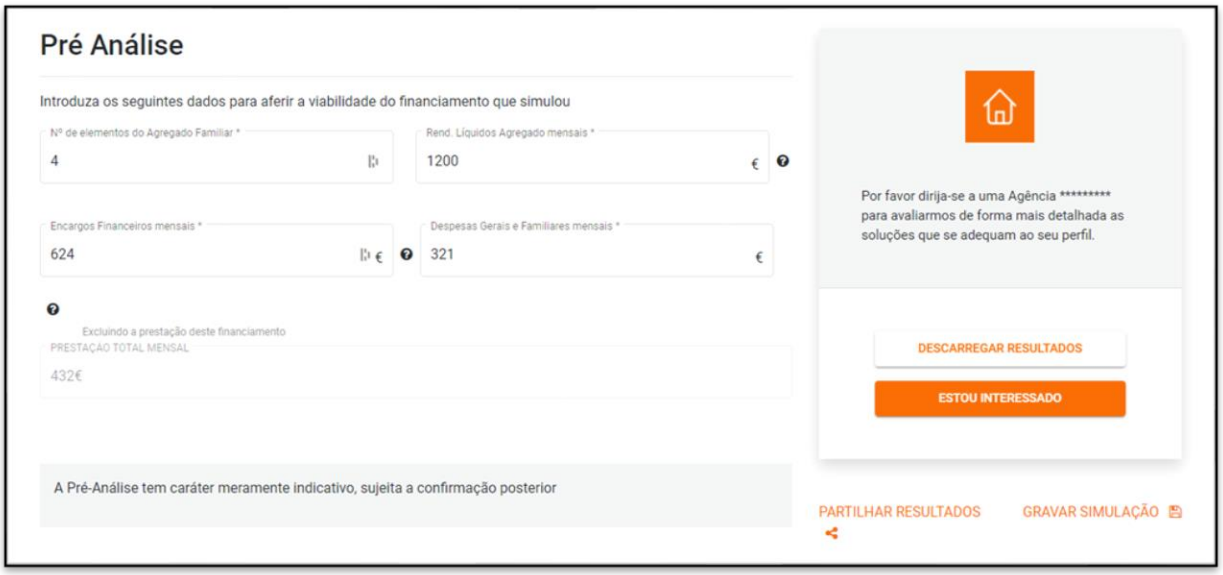

Figura 20 Interface simulação obras – pré-análise

Neste ponto todos os dados sobre a simulação estão preenchidos. O utilizador carrega em "marcar visita a agência para abrir um *overlay* onde preenche os dados sobre a visita, Figura 21.

Figura 21 Interface simulação obras – aprovação condicionada

Para marcar a visita é necessário escolher a agência e para escolher a agência é necessário saber a área geográfica para onde o utilizador pretende marcar a visita, Figura 22.

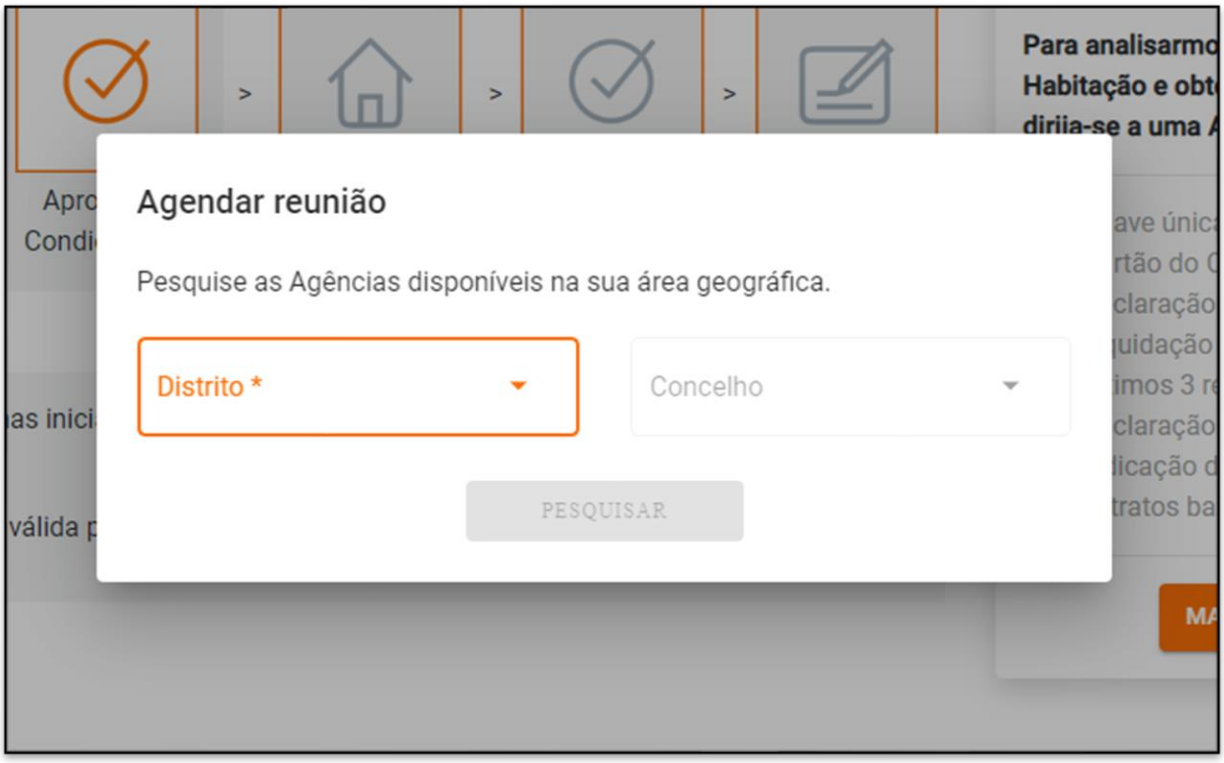

Figura 22 Interface overlay marcar visita – escolher área da agência

Depois de conhecer a área geográfica mostrar as agências disponíveis nessa área, Figura 23.

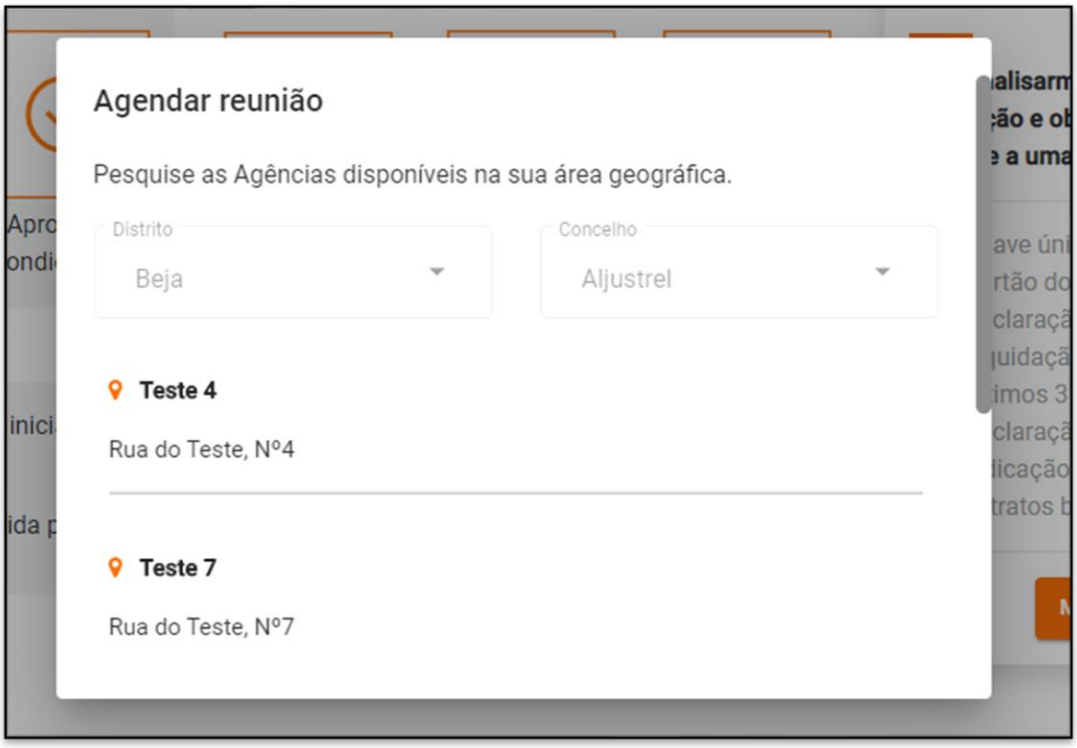

Figura 23 Interface overlay marcar visita – escolher agência

Depois de escolher a agência o utilizador preenche as informações sobre a reunião, Figura 24 e Figura 25.

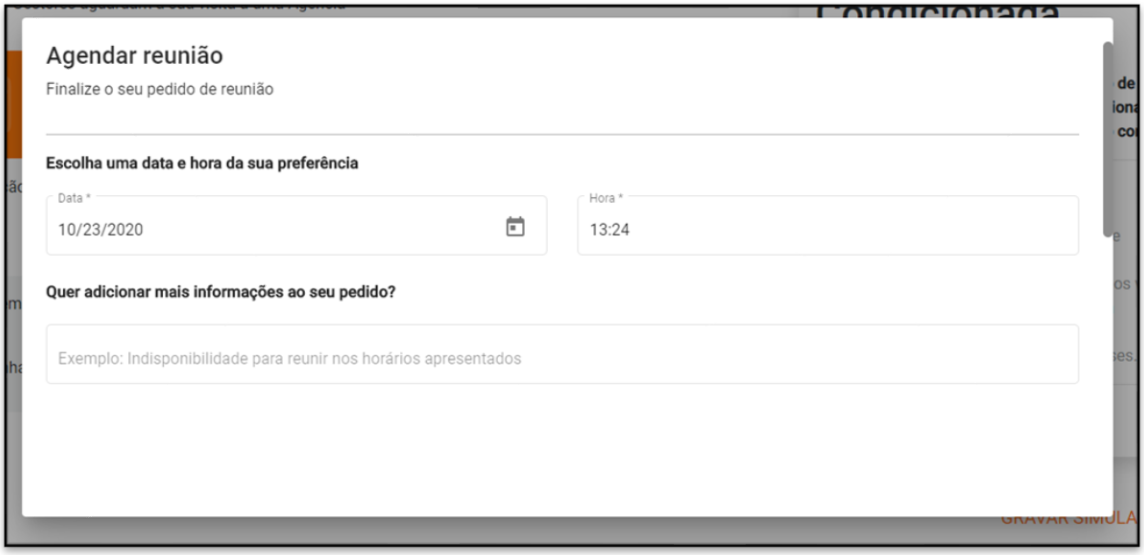

Figura 24 Interface overlay marcar visita – informações sobre a reunião

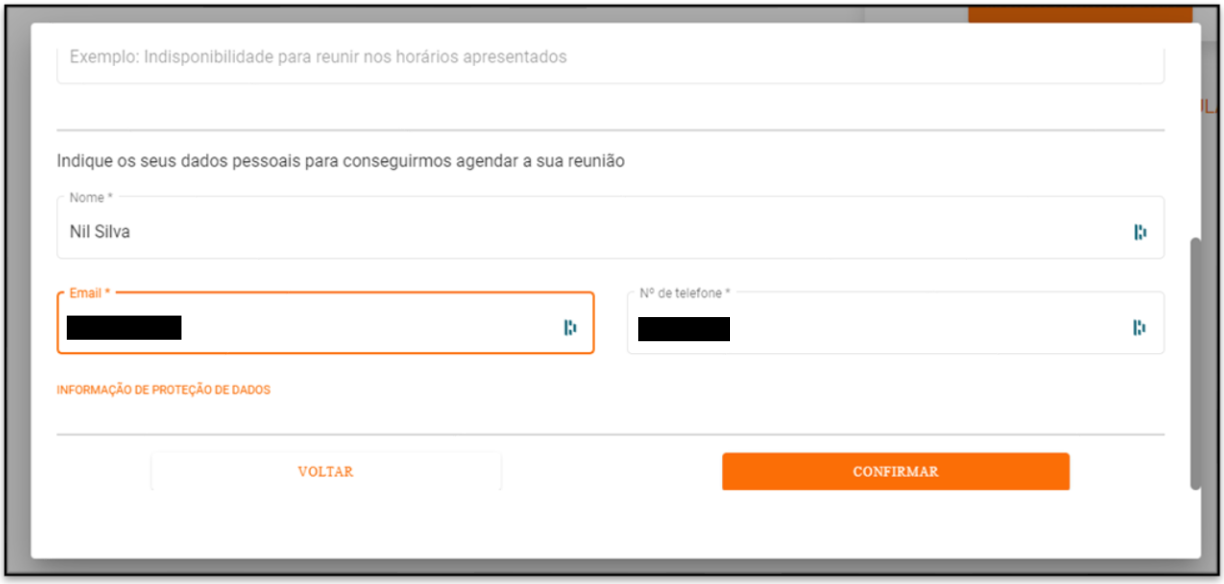

Figura 25 Interface overlay marcar visita – informações sobre a reunião (cont.)

Os funcionários na *navbar* da página inicial têm uma opção adicional. Essa opção é "agências" e quando o utilizador carrega nela vai para uma página de gestão de agências. Para apagar uma agência basta carregar no ícone do caixote do lixo da agência a apagar, Figura 26.

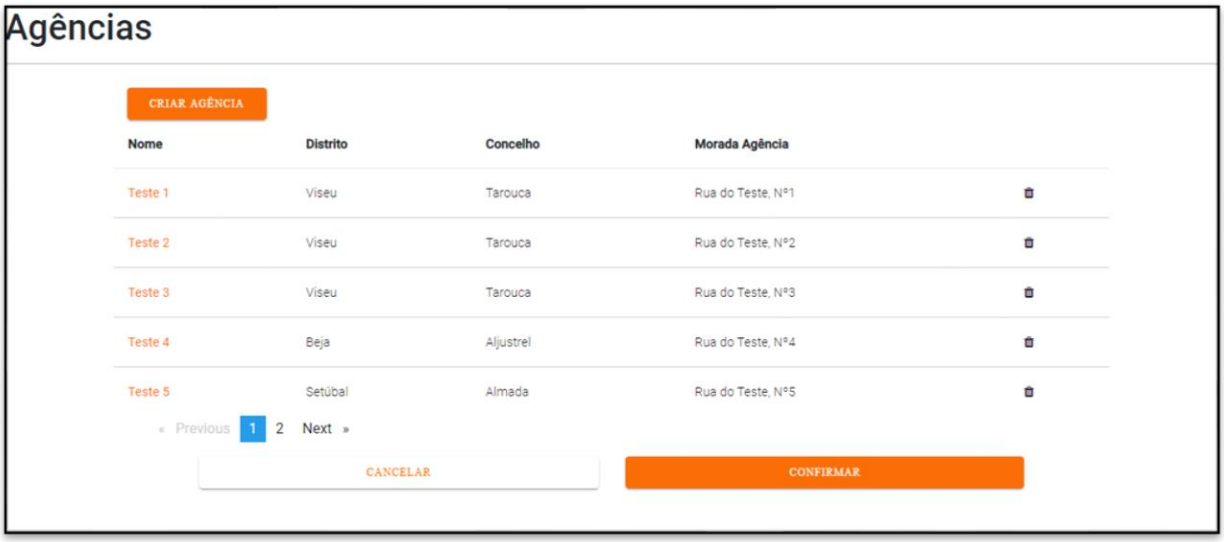

Figura 26 Interface gestão de agências

Quando o utilizador carrega no botão criar agências o formulário para criar agências aparece, Figura 27.

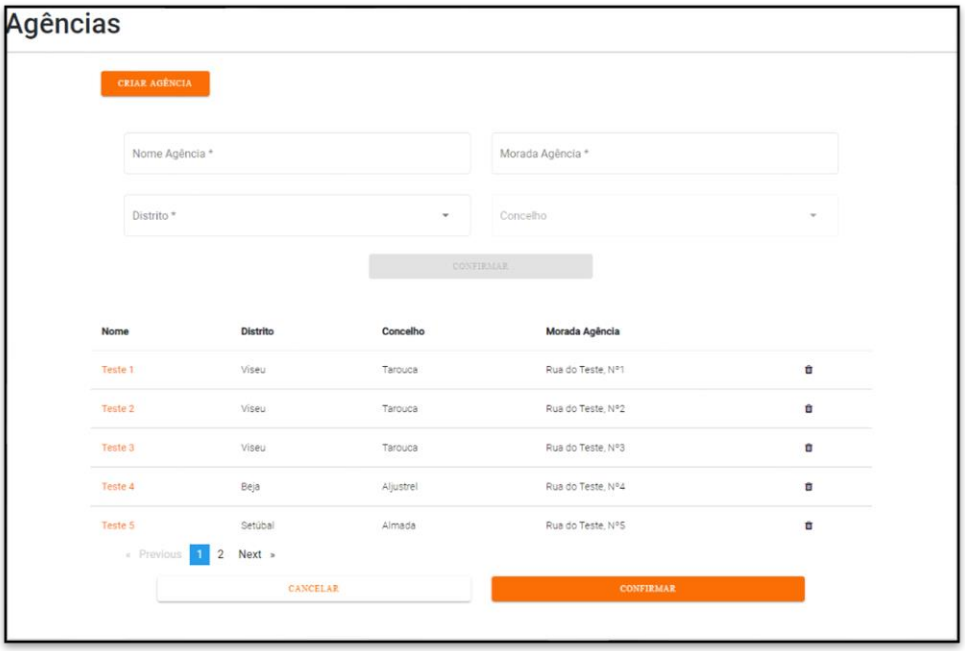

Figura 27 Interface gestão de agências – criar nova agencia

Os administradores na *navbar* da página inicial só têm uma opção. Essa opção é "utilizadores" e quando o utilizador carrega nela vai para uma página de gestão de utilizadores. Para apagar basta carregar no ícone do caixote do lixo do utilizador a apagar. E para alterar as permissões basta usar o *dropdown* do utilizador, Figura 28.

Figura 28 Interface gestão utilizadores

6 Testes

Ao desenvolver *Software* é expectável ocorrer erros. Como tal é necessário implementar testes para validar as funcionalidades implementadas.

Para testar a aplicação foi usado o Postman. O Postman é uma aplicação que permite testar APIs facilmente através de testes unitários.

O Postman permite fazer pedidos HTTP e agrupar esses pedidos em coleções. Foi criada uma coleção por cada controlador. E dentro das coleções foram criados pedidos para cada uma das funções do controlador correspondente. Assim é possível testar facilmente qualquer função. O Postman também permite adicionar *snippets* de código aos pedidos para poder verificar se não ocorreu nenhum erro (14).

A seguir está um exemplo de um *snippet* de teste utilizado no Postman:

```
pm.test("Verificar estado", function () {
    pm.response.to.have.status(200);
});

pm.test("Verificar JSON", function () {
    pm.response.to.be.json;
});
```

Neste bloco de código são aplicados dois testes o primeiro verifica que o código de estado recebido foi o 200 (sucesso). O segundo verifica se os dados enviados estão em formato JSON. Muitas das funções retornam dados neste formato por isso é importante validar isso.

Testes como o anterior foram aplicados às várias funções disponíveis para validar os resultados recebidos.

7 Conclusões

Ao longo do estágio foi desenvolvida uma aplicação de simulação do crédito de habitação. A aplicação, para além de deixar fazer a simulação do crédito de habitação e marcar uma visita a uma agência para falar com um representante, também tem um sistema de registo e autenticação de utilizadores. A aplicação tem três tipos de utilizador o cliente, funcionário e administrador tendo eles permissões para interagir com funcionalidades diferentes.

Os objetivos eram, no final, ter uma aplicação que permite aos utilizadores fazerem o registo, autenticação e simulações do crédito de habitação. Os objetivos propostos foram todos atingidos. Estando pronta a usar. O *frontend* foi a parte mais trabalhosa pois Angular era um tecnologia desconhecida até ter trabalhado nesta aplicação. Pelo contrário, o Node.Js já tinha sido trabalhado antes durante a licenciatura.

Apesar da aplicação estar a funcionar devidamente, podia ser otimizada e melhorar a velocidade. Tendo em conta a inexperiência dos membros da equipa, eu e o Fernando Lopes, e o curto tempo do estágio não pode ser evitada esta falta de otimização.

Uma adição interessante a fazer ao projeto seria o *deploy* da aplicação utilizando um serviço cloud como por exemplo o Microsoft Azure ou o Amazon Web Services.

Referências

1. **Softinsa.** *softinsa.pt.* [Online] [Citação: 25 de Julho de 2020.] https://www.softinsa.pt/pt/.

2. **Angular.** Introduction to Angular concepts. *Angular.* [Online] [Citação: 30 de Julho de 2020.] https://angular.io/guide/architecture.

3. —. Angular Material. *https://angular.io/.* [Online] [Citação: 23 de Junho de 2020.] https://material.angular.io/.

4. **Angular Material Tutorial. *tutorialspoint.* [Online] [Citação: 10 de 11 de 2020.] https://www.tutorialspoint.com/angular_material/index.htm.**

5. Node.js. *Node.js.* [Online] [Citação: 12 de 11 de 2020.] https://nodejs.org/.

6. Express - Node.js web application framework. *Express.* [Online] [Citação: 13 de 11 de 2020.] https://expressjs.com/.

7. PostgreSQL: The World's Most Advanced Open Source Relational Database. *PostgreSQL.* [Online] [Citação: 12 de 11 de 2020.] https://www.postgresql.org/.

8. *Sequelize.* [Online] [Citação: 5 de Julho de 2020.] https://sequelize.org/.

9. Hoyos, Mario. What is an ORM and Why You Should Use it. *Medium.* [Online] [Citação: 28 de Outubro de 2020.] https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a.

10. O que é o DevOps? *Microsoft Azure.* [Online] [Citação: 12 de 11 de 2020.] https://azure.microsoft.com/pt-pt/overview/what-is-devops.

11. *JSON Web Tokens.* [Online] [Citação: 9 de Julho de 2020.] https://jwt.io/.

12. Ami, Bhatt. Angular and Observable. *Medium.* [Online] [Citação: 05 de Agosto de 2020.] https://medium.com/fuzzycloud/angular-and-observable-4bf890b2a282.

13. Pina, João. distritos-concelhos-freguesias-Portugal.json. *GitHub.* [Online] [Citação: 23 de Julho de 2020.] https://gist.github.com/tomahock/a6c07dd255d04499d8336237e35a4827.

14. Postman. Writing tests. *Postman Learning Center.* [Online] [Citação: 15 de Julho de 2020.] https://learning.postman.com/docs/writing-scripts/test-scripts/.