

Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu
Departamento de Informática



Relatório de Projeto

The Softshares

Lucas Sebastião, pv22589

João Santos, estgv18725

Daniel Valpereiro, 22894

Cassamo Latif, 21373

Francisco Meneses, pv20888

Viseu, 2024

Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu
Departamento de Informática

Relatório de Projeto
Curso de Licenciatura em
Engenharia Informática

The Softshares

Ano Letivo 2023/2024

Lucas Sebastião, pv22589

João Santos, estgv18725

Daniel Valpereiro, 22894

Cassamo Latif, 21373

Francisco Meneses, pv20888

Viseu, 2024

Índice

1. Introdução	6
2. Requisitos	7
3. Desenvolvimento	12
3.1. Base de Dados	12
3.1.1. <i>Estrutura da Base de Dados</i>	12
3.1.2. <i>PgAdmin</i>	13
3.2. API.....	14
3.2.1. <i>Tecnologias Utilizadas</i>	14
3.2.2. <i>Arquitetura da Aplicação</i>	18
3.2.3. <i>Estrutura</i>	44
3.3. Web.....	46
3.3.1. <i>Estrutura do Projeto</i>	46
3.3.2. <i>Estrutura da Aplicação</i>	46
3.3.3. <i>Rotas</i>	46
3.3.4. <i>Componentes</i>	46
3.3.5. <i>Comunicação com a API</i>	52
3.3.6. <i>Páginas</i>	55
3.4. Mobile.....	71
3.4.1. <i>Estrutura do Projeto</i>	71
3.4.2. <i>Detalhes dos conteúdos</i>	71

3.4.3. Criar conteúdos (Daniel).....	80
3.4.4. Perfil (Daniel)	80
3.4.5. Editar perfil (Daniel).....	80
3.4.6. Calendário (Francisco).....	81
3.4.7. Fórum (Francisco)	81
3.4.8. Listagem por tipo (Francisco).....	81
3.4.9. Integração com o Google e Facebook (Cassamo/Francisco).....	81
3.4.10. Iniciar sessão (Cassamo).....	81
3.4.11. Comunicação com a API (Cassamo).....	81
4. Conclusão	82

Índice de tabelas

Tabela 1 - Requisitos	7
-----------------------------	---

Índice de figuras

Figura 1 - Instanciação do <i>Express</i>	18
Figura 2 - Conecta com a base de dados utilizando o <i>Sequelize</i>	18
Figura 3 - Configuração dos modelos e sincronização com a Base de Dados	19
Figura 4 - Configuração do Servidor	19
Figura 5 - Configuração das rotas e <i>middlewares</i>	19
Figura 6 - Classe <i>Controller</i>	21
Figura 7 - Classe <i>BaseController</i>	22
Figura 8 - Instanciação dos <i>middlewares</i>	43
Figura 9 - <i>Layout</i> da aplicação	46
Figura 10 - <i>PageContent</i> componente utilizado no layout	46
Figura 11 - Exemplo da definição/criação das rotas.....	46
Figura 12 - Componente <i>ProtectedRoute</i>	46
Figura 13 - Página Inicial <i>hero section</i>	55
Figura 14 – Página: Inicial listagem	55
Figura 15 – Página: Calendário	56
Figura 16 – Página: Mapa.....	57
Figura 17 - Página: Utilizadores (Backoffice).....	58
Figura 18 - Página: Conteúdos (Backoffice)	59
Figura 19 - Página: Comentários (Backoffice).....	60
Figura 20 - Página: Denúncias (Backoffice)	61

Figura 21 - Página: Tópicos (Backoffice)	62
Figura 22 - Página: Centros (Backoffice).....	63
Figura 23 - Página: Sobre	65
Figura 24 - Formulário para criar uma atividade.....	66
Figura 25 - <i>Popup</i> para escolher a localização da atividade.....	66
Figura 26 - Exemplo de um evento	67
Figura 27 - Exemplo de uma recomendação	68
Figura 28 - Página: Perfil	69
Figura 29 - Página: Perfil - Interesses	70

1. Introdução

No contexto atual de crescimento e diversificação geográfica de muitas empresas, torna-se cada vez mais importante facilitar a integração de novos colaboradores. Este projeto surge com o objetivo de desenvolver uma aplicação que possa servir como um recurso essencial para esses colaboradores, ajudando-os a adaptar-se rapidamente à nova cidade e à cultura organizacional.

A aplicação proposta visa centralizar e compartilhar informações úteis sobre as cidades onde a empresa opera, incluindo recomendações de colegas sobre restaurantes, habitação, atividades desportivas, lazer, entre outros. A ideia é criar um sistema interativo e colaborativo que permita aos novos colaboradores conhecerem melhor a região e, conseqüentemente, sentirem-se mais integrados na empresa.

Este projeto escolar envolve o desenvolvimento de um sistema de BackOffice, e uma aplicação mobile. O foco é avaliar as habilidades técnicas e a capacidade de implementar uma solução prática e funcional que atenda às necessidades descritas, promovendo um ambiente de trabalho mais acolhedor e colaborativo.

Estrutura do relatório.

2. Requisitos

Os requisitos funcionais descrevem as principais funcionalidades e comportamentos que o sistema deve ter. Especificam o que o sistema deve fazer para ir de encontro às necessidades dos utilizadores do mesmo. A tabela 1 demonstra os requisitos funcionais gerais e da parte web.

Tabela 1 - Requisitos

Requisitos Funcionais	
1	Criar Utilizadores
2	Ativar/Inativar utilizadores
3	Associar utilizadores a permissões da aplicação
4	Associar utilizadores a pelo menos um centro
5	Obrigar a atualização de password no primeiro login
6	Enviar email ao utilizador após o seu registo/criação na aplicação
7	Possibilidade de criação de áreas/categorias (Desporto, Formação, Transportes,...)
8	Possibilidade de criação de subáreas/atividades (Futebol, Boleias, Padle, ...)
9	Possibilidade de criação de centros
10	Possibilidade de criação de fóruns de discussão por categoria e/ou por atividade.
11	Associado a cada área/categoria possibilidade de criação de Álbuns.

12	Apresentação do calendário de eventos
13	Na área pessoal de um utilizador normal deve ser possível indicar/definir as áreas e subtópicos preferidos. Depois, sempre que forem publicados conteúdos nessas áreas e subtópicos, o utilizador deve ser notificado
14	Alterar o estado dos formulários a qualquer altura
15	Um administrador, em backoffice, apenas pode criar e moderar conteúdos relacionados com o centro em que é administrador. Os conteúdos moderados e publicados devem aparecer de imediato na aplicação móvel
16	Os Administradores podem editar os conteúdos dos utilizadores, quando estão a fazer a moderação. Por exemplo, podem mudar a área associada ao conteúdo
17	BackOffice adicionar os vários tipos de Categorias/Recomendações/Eventos/Atividades/Espaços. Cada tópico, terá subtópicos de acordo com as áreas de atuação.
18	Validação em backoffice de recomendações/eventos/comentários e outros conteúdos inseridos pelos utilizadores da aplicação móvel.
19	Visão geral no BackOffice em dashboard dividido por área de atuação, atividade mais comentada, atividades mais vistas, nº de tópicos abertos, e recomendações/eventos por validar, etc..
20	Um administrador só pode validar/aprovar/publicar conteúdos do seu centro
Requisitos Funcionais - APP Mobile	
21	Poder efetuar o login de 3 formas distintas

22	Possibilidade de guardar os dados de login, de forma a que não seja obrigado a efetuar o login sempre que utiliza a Aplicação
23	Possibilidade de recuperar password e de cancelar o processo de recuperar password
24	Página “Terminar Sessão” com o comportamento descrito no enunciado do trabalho.
25	Possibilidade de criação de recomendação
26	Possibilidade de criação de evento
27	Apresentação do calendário de eventos
28	Possibilidade de submeter mensagens em fóruns
29	Possibilidade de submeter fotos em álbuns
30	Possibilidade de fazer avaliações (podem ser parametrizáveis) de conteúdos. Por exemplo, de recomendações, comentários, etc.
31	Inserir comentários em cada um dos eventos disponíveis e visíveis
32	Possibilidade de denunciar comentários
33	Possibilidade de avaliar comentários
34	Visualizar eventos nos quais se encontra inscrito
35	Possibilidade de inscrever em eventos ou em grupos associados aos eventos
36	Possibilidade de partilha de uma recomendação/evento. (Exemplo, partilha de um restaurante com um colega/amigo).
37	Disponibilização de preços médios para cada recomendação partilhada

38	Na criação de eventos, recomendações, espaços ou edifícios relativos às atividades, associar geolocalização no Google Maps.
39	Disponibilizar hiperligação para partilha das atividades através das redes sociais ou na plataforma colaborativa (Teams)
40	Gerar notificações aquando de alterações de eventos ao qual se encontra inscrito
41	Gerar notificações quando são colocados comentários novos no painel dos eventos inscritos.
42	O utilizador que criou o anúncio de atividade (Ex: Jogo de Futebol, Encontro BTT, Jogo de Padle) deve receber notificação de qualquer interação que tenha sido efetuada no mesmo (inscrições, questões colocadas, etc.)
43	OS utilizadores normais podem colocar conteúdos/comentários/avaliações/mensagens em todos os centros
44	Visualizar lista de recomendações filtrada por áreas de interesse e por classificação
45	Permitir alterar os dados e acrescentar informação nos eventos do qual sou owner até esse conteúdo ser publicado por um administrador
46	Permitir inserir locais de interesse/recomendações por categoria ou área de atuação.
47	Na área pessoal de um utilizador normal deve ser possível indicar/definir as áreas e subtópicos preferidos. Depois, sempre que forem publicados conteúdos nessas áreas e subtópicos, o utilizador deve ser notificado
48	Possibilidade de alterar o estado dos formulários antes de eles serem publicados por um Administrador
Bónus	

49	A plataforma deve ter 3 línguas disponíveis (Português, Inglês e Espanhol);
50	Apresentar uma saudação ao utilizador conforme a hora
51	Disponibilizar área de Informações/Avisos: A página “Informações/Avisos” é a página onde os administradores podem criar, consultar e editar informações genéricas e avisos, que são disponibilizados imediatamente na Plataforma Web e App mobile na página “Informações/Avisos”

3. Desenvolvimento

Neste capítulo vão ser explicadas detalhadamente as aplicações mobile e Web e a infraestrutura que está por trás, Base de dados e API. Para a entrega do projeto utilizamos a plataforma render.com para hospedar as diferentes aplicações (Web, API e Base de Dados). Também com o intuito de facilitar a colaboração em equipa utilizamos o Github.

3.1. Base de Dados

3.1.1. Estrutura da Base de Dados

A base de dados foi desenhada para suportar de forma eficiente as principais funcionalidades da aplicação. Abaixo, apresento uma visão geral das principais tabelas e dos seus relacionamentos:

- **Perfil, Centro e Utilizador:** Estas tabelas armazenam as informações básicas sobre os perfis dos utilizadores, os centros aos quais podem estar associados e os próprios utilizadores. As relações entre estas tabelas permitem definir e gerir os diferentes tipos de utilizadores na aplicação.
- **Tópicos, Subtópicos e Interesses:** Estas tabelas organizam as áreas temáticas e os interesses dos utilizadores, permitindo uma personalização e filtragem de conteúdos relevantes para cada perfil.
- **Conteúdo:** Tabelas como conteúdo, álbum, comentário, e classificação armazenam e gerem todo o conteúdo gerado pelos utilizadores, incluindo uploads de imagens, comentários e avaliações.
- **Moderação e Gestão de Estado:** As tabelas estado, revisão, e denuncia são utilizadas para gerir o fluxo de trabalho de moderação, revisões de conteúdo, notificações enviadas aos utilizadores e o tratamento de denúncias.

3.1.2. PgAdmin

A implementação da base de dados foi realizada no pgAdmin, uma ferramenta robusta para administração de bases de dados PostgreSQL. As operações de criação, modificação e gestão das tabelas foram executadas através de scripts SQL, que permitem a criação de uma estrutura eficiente e escalável.

3.2. API

3.2.1. Tecnologias Utilizadas

3.2.1.1 Node.js

Node.js é uma plataforma de desenvolvimento backend baseada no motor V8 do Google Chrome, que permite a execução de JavaScript no lado do servidor. É uma tecnologia fundamental para a API, oferecendo um ambiente eficiente e escalável para o desenvolvimento de aplicações web. Node.js permite a construção de APIs que lidam com um grande número de conexões simultâneas, graças ao seu modelo assíncrono baseado em eventos.

3.2.1.2 Express.js

Express.js é um framework minimalista e flexível para Node.js, que facilita a criação de aplicações web robustas e APIs RESTful. Ele fornece uma estrutura sólida para lidar com requisições HTTP, roteamento, middlewares, e muito mais. Na tua API, o Express.js é utilizado para definir rotas, gerir middlewares (como autenticação e manipulação de erros), e estruturar a lógica de negócio de forma modular e eficiente.

3.2.1.3 Sequelize

Sequelize é um ORM (Object-Relational Mapping) para Node.js, que simplifica a interação com bases de dados relacionais, como PostgreSQL, MySQL, MariaDB, SQLite e MSSQL. Ele permite mapear modelos JavaScript a tabelas de bases de dados, facilitando operações como consultas, inserções, atualizações e exclusões de dados, tudo isso utilizando uma sintaxe simples e intuitiva. Na API, o Sequelize é utilizado para abstrair a camada de acesso a dados, permitindo que desenvolvedores trabalhem com objetos JavaScript ao invés de escrever diretamente SQL, o que acelera o desenvolvimento e reduz a probabilidade de erros.

3.2.1.4 JWT (JSON Web Token)

JSON Web Token (JWT) é uma tecnologia de autenticação que permite a comunicação segura entre cliente e servidor. É amplamente utilizada para gerir sessões de utilizadores, mantendo a segurança sem a necessidade de armazenar informações sensíveis no servidor. Na tua API, JWT é utilizado para gerar tokens que são enviados ao cliente após a autenticação bem-sucedida, permitindo que o cliente acesse rotas protegidas até que o token expire.

3.2.1.5 bcrypt

bcrypt é uma biblioteca de hashing que proporciona uma forma segura de encriptar senhas antes de armazená-las na base de dados. Ela aplica um algoritmo de hashing que é lento e resistente a ataques de força bruta, aumentando a segurança das credenciais dos utilizadores. Na API, o bcrypt é utilizado para encriptar senhas no momento da criação de utilizadores e para compará-las durante o processo de login.

3.2.1.6 Cloudinary

Cloudinary é uma solução de gestão de imagens e vídeos na nuvem, que facilita o upload, armazenamento e manipulação de arquivos multimédia. Na API, o Cloudinary é integrado para lidar com o upload e a gestão de imagens de forma eficiente, permitindo que estas sejam armazenadas na nuvem e servidas de forma otimizada para os clientes.

3.2.1.7 Crypto

Crypto é um módulo nativo do Node.js que fornece funcionalidades de criptografia e hashing. Na API, é utilizado para operações como geração de hashes, criação de tokens de segurança, e encriptação de dados sensíveis. Isso garante uma camada extra de segurança, protegendo informações críticas contra acessos não autorizados.

3.2.1.8 google-auth-library

google-auth-library é uma biblioteca utilizada para integrar autenticação e autorização via OAuth 2.0 e JWT com os serviços da Google. Na API, pode ser empregada para permitir que os utilizadores façam login utilizando as suas credenciais do Google, facilitando a autenticação através de um provedor de identidade confiável e amplamente utilizado.

3.2.1.9 multer

Multer é uma biblioteca de middleware para Express.js que facilita o tratamento de uploads de arquivos. É especialmente útil para manipular uploads de imagens, documentos e outros tipos de arquivos na API. O Multer permite definir a forma como os arquivos são armazenados e geridos, seja na memória ou diretamente em disco, garantindo uma integração eficiente com serviços de armazenamento como o Cloudinary.

3.2.1.10 nodemailer

Nodemailer é uma biblioteca para o envio de emails a partir de aplicações Node.js. Na API, é utilizado para enviar emails automatizados, como confirmações de registo, notificações, ou recuperação de senhas. O Nodemailer facilita a integração com diferentes serviços de email, permitindo que a API envie mensagens de forma confiável e segura.

3.2.1.11 pg e pgAdmin

pg é um cliente para Node.js que permite a interação direta com bases de dados PostgreSQL. Na API, o pg é utilizado para executar consultas SQL quando se prefere um acesso mais direto ou personalizado à base de dados, sem a necessidade de abstrações oferecidas por um ORM

como o Sequelize. Isso pode ser útil para operações específicas ou quando o desempenho é uma prioridade.

3.2.2. Arquitetura da Aplicação

3.2.2.1 Configuração Inicial

Este capítulo descreve as etapas envolvidas na configuração da aplicação, destacando os principais componentes e o fluxo de inicialização da API.

3.2.2.1.1 Inicialização da Aplicação Express

A primeira etapa na configuração da API é a criação de uma instância da aplicação Express. Express é um *framework* minimalista para Node.js, amplamente utilizado para construir *API's* devido à sua simplicidade e extensibilidade.

```
const app = express();
```

Figura 1 - Instanciação do *Express*

Esta linha cria uma instância da aplicação Express. Essa instância será usada para configurar e gerenciar rotas, middlewares, e para iniciar o servidor HTTP.

3.2.2.1.2 Conexão e Autenticação com a Base de Dados

A seguir, a API estabelece uma conexão com a base de dados usando Sequelize, um ORM (*Object-Relational Mapping*) que facilita a interação com bases de dados relacionais.

```
const sequelize = await DatabaseConfig.connect();
```

Figura 2 - Conecta com a base de dados utilizando o *Sequelize*

Aqui, o método `connect()` do módulo `DatabaseConfig` estabelece uma conexão com a base de dados especificada nas configurações. O Sequelize é configurado com os parâmetros de conexão (nome do banco de dados, usuário, senha, host, etc.) e tenta autenticar a aplicação junto ao banco de dados.

3.2.2.1.3 Configuração dos Modelos e Sincronização da Base de Dados

Após a conexão com a base de dados, a configuração dos modelos de dados é realizada. Esses modelos representam as tabelas no banco de dados e suas relações.

```
await ModelsConfig(sequelize);
```

Figura 3 - Configuração dos modelos e sincronização com a Base de Dados

A função `ModelsConfig` define os modelos de dados utilizando `Sequelize` e configura as associações entre eles, como relações de um-para-muitos ou muitos-para-muitos. Em seguida, sincroniza esses modelos com a estrutura atual da base de dados.

3.2.2.1.4 Configuração do Servidor

Com a base de dados configurada e os modelos sincronizados, o próximo passo é configurar o servidor HTTP que servirá a API.

```
await ServerConfig(app);
```

Figura 4 - Configuração do Servidor

A função `ServerConfig` aplica *middlewares* fundamentais, como o suporte a JSON e CORS (*Cross-Origin Resource Sharing*), configura a porta onde o servidor escutará, e inicia o processo de escuta para requisições HTTP.

3.2.2.1.5 Configuração das Rotas e Middlewares

Finalmente, as rotas e *middlewares* são configurados. Isso inclui *middlewares* de autenticação, *logging*, e tratamento de erros, além das rotas que definem os endpoints da API.

```
await RoutingConfig(app);
```

Figura 5 - Configuração das rotas e *middlewares*

A função *RoutingConfig* aplica os *middlewares* globais, como o de autenticação e de *logging*, e define as rotas da aplicação. Também inclui um middleware de tratamento de erros para garantir que qualquer exceção seja adequadamente capturada e processada.

3.2.2.2 *Controllers*

Os *Controllers* são uma parte fundamental na arquitetura de uma aplicação, pois atuam como intermediários entre a lógica de negócios e as requisições feitas pelos utilizadores. São responsáveis por processar as entradas, interagir com os serviços de negócio e devolver as respostas apropriadas aos clientes. Este capítulo descreve a implementação dos *Controllers* na nossa aplicação, com foco nas classes *Controller* e *BaseController*, que fornecem a estrutura necessária para criar controladores específicos para diferentes modelos de dados.

1. Estrutura Geral dos *Controllers*

Os *Controllers* foram implementados de forma a serem reutilizáveis e modulares, permitindo uma fácil extensão e manutenção. A estrutura básica de um *Controller* é definida na classe *Controller*, que serve como uma classe base, enquanto a classe *BaseController* fornece implementações concretas de métodos comuns, como operações *CRUD* (*Create*, *Read*, *Update*, *Delete*).

2. A Classe *Controller*

A classe *Controller* é a base sobre a qual todos os outros controladores são construídos. Ela encapsula a lógica comum de inicialização dos controladores, como a associação de um modelo de dados específico e a configuração de um serviço básico para operar com esse modelo. Cada controlador específico herda desta classe, garantindo que as operações básicas e o registo de logs sejam realizados de forma consistente em toda a aplicação.

Estrutura da Classe *Controller*

```
export class Controller {
  constructor(model, identifier = Constants.DEFAULT_IDENTIFIER) {
    this.model = model;
    this.identifier = identifier;
    this.service = new BaseService(model, identifier);
    LogUtils.log(model.name, LogUtils.TIPO.CONTROLLERS);
  }
}
```

Figura 6 - Classe *Controller*

- Inicializa a classe com um *model* e um *identifier*, que são usados para definir o contexto de operação do controlador.
- Um serviço (*BaseService*) é instanciado para gerir as operações relacionadas ao modelo.
- A classe também regista informações de log para monitorizar a operação do controlador.

3. A Classe *BaseController*

A classe *BaseController* estende a classe *Controller* e implementa métodos pré-definidos para as operações *CRUD*. Esta classe foi projetada para ser uma solução genérica que pode ser facilmente adaptada para diferentes modelos, fornecendo uma implementação padrão para as operações mais comuns.

Estrutura da Classe *BaseController*

```
export class BaseController extends Controller {
  constructor(model, identifier = Constants.DEFAULT_IDENTIFIER) {
    super(model, identifier);
  }

  async obter(req, res) {
    try {
      const { id } = req.params;
      const response = await this.service.obter(id);
      return ResponseService.success(res, response);
    } catch (error) {
      return ResponseService.error(res, error.message);
    }
  }

  //...
}
```

Figura 7 - Classe *BaseController*

Métodos CRUD:

- obter: Busca um registo específico com base no identificador fornecido.
- simples_obter, criar, listar, atualizar, remover: Métodos adicionais que cobrem as operações básicas de uma API RESTful.
- Cada método segue uma estrutura padrão: tenta realizar a operação com a ajuda de um serviço e retorna uma resposta apropriada ao cliente.

4. Implementação de um *controller* (classe *UtilizadorController*)

A classe *UtilizadorController* é uma extensão da classe *BaseController* e foi projetada para lidar com operações específicas para o modelo de utilizador, incluindo o upload de imagens e a manipulação de dados associados ao utilizador. Esta classe demonstra como um controlador pode ser personalizado para gerir funcionalidades específicas além das operações CRUD básicas fornecidas pela *BaseController*.

Explicação dos Métodos

- Método `imagem_atualizar`:

Objetivo: Atualizar a imagem de um utilizador no sistema.

Funcionamento: Utiliza o serviço *UploadService* para fazer o upload da imagem para o serviço de armazenamento em nuvem (*Cloudinary*).

Após o upload, a URL da imagem é devolvida e é utilizada para atualizar o registo do utilizador com a nova imagem.

A resposta é retornada utilizando o *ResponseService*, que fornece uma estrutura consistente para respostas de sucesso ou erro.

- Método `criar`:

Objetivo: Criar um utilizador.

Funcionamento: O método utiliza o *AuthLoginService* para criar um utilizador com base nos dados fornecidos no corpo da requisição.

A resposta é gerida e retornada pelo *ResponseService*.

- Método `obter`:

Objetivo: Obter detalhes de um utilizador específico.

Funcionamento: Busca o utilizador com base no ID fornecido na URL.

Inclui informações adicionais definidas pelo método estático `#modelos_adicionais`, que especifica quais associações e modelos relacionados devem ser incluídos na resposta.

- Método `simples_listar`:

Este método é uma reimplementação do método `simples_listar` do *BaseController*.

Objetivo: Listar utilizadores com informações simplificadas.

Funcionamento: Lista todos os utilizadores com base nos critérios fornecidos no corpo da requisição. Utiliza o método estático `#modelos_adicionais_simplificado` para determinar quais associações simplificadas devem ser incluídas na resposta.

- `modelos_adicionais`: Define associações detalhadas que devem ser incluídas nas respostas quando se obtém ou lista utilizadores. Inclui o modelo `interesse` e suas associações com `subtopico`.

3.2.2.3 *Models*

Os *models* são o componente central da camada de dados de uma aplicação. Eles definem a estrutura das entidades no sistema, como um utilizador, um comentário ou uma notificação, mapeando essas entidades para tabelas no banco de dados relacional. Em *frameworks* como o *Sequelize*, os *models* permitem trabalhar com dados de forma orientada a objetos, facilitando operações como criação, leitura, atualização e exclusão (CRUD).

1. Estrutura dos *Models*

Os *models* são configurados para representar várias entidades, como Utilizador, Conteúdo, Comentário, entre outros. Cada um destes modelos tem as suas próprias propriedades, que refletem os atributos específicos da entidade, como o nome do utilizador, o texto do comentário, ou a data de criação de um conteúdo.

2. Relações Entre *Models*

Um aspeto importante dos *models* é a definição das relações entre diferentes entidades. Por exemplo, um utilizador pode ter muitos conteúdos, e cada conteúdo pode ter muitos comentários. Estas relações são configuradas utilizando associações como *hasMany*, *belongsTo*, entre outras, garantindo que as operações no banco de dados mantenham a integridade referencial.

3.2.2.4 Routes

As Routes (rotas) são uma parte crucial na arquitetura de uma aplicação web, pois definem como as requisições HTTP são mapeadas para as funções dos Controllers. Na aplicação descrita, as rotas são implementadas utilizando o framework Express.js e seguem um padrão modular e reutilizável, garantindo que cada Controller tenha um conjunto de rotas claramente definido. Esta seção descreve a implementação das rotas na aplicação, focando na função `BaseRoutes`, nas rotas específicas como `ComentarioRoutes` e na configuração geral das rotas com `RoutesConfig`.

Estrutura Geral das Routes

As Routes são definidas de forma modular, utilizando funções que geram instâncias de routers do Express.js. Estas funções recebem como parâmetro a classe do Controller que será responsável por lidar com as requisições. Dessa forma, cada rota mapeia uma URL específica para um método correspondente do Controller, como obter, criar, listar, atualizar, e remover.

A Função BaseRoutes

A função `BaseRoutes` é responsável por criar um conjunto padrão de rotas para qualquer entidade que siga as operações CRUD (Create, Read, Update, Delete). A função recebe uma classe de Controller como parâmetro e retorna um objeto router configurado com as seguintes rotas:

- `GET Constants.URL_NAMING.GET`: Mapeia para o método `obter` do Controller. Este método é usado para buscar um registo específico baseado no identificador fornecido na URL.
- `GET Constants.URL_NAMING.SIMPLE_GET`: Mapeia para o método `simples_obter` do Controller, que busca informações simplificadas de um registo.
- `POST Constants.URL_NAMING.CREATE`: Mapeia para o método `criar`, utilizado para criar um novo registo no sistema.

- `POST Constants.URL_NAMING.LIST`: Mapeia para o método `listar`, que retorna uma lista de registros baseados em critérios fornecidos na requisição.
- `POST Constants.URL_NAMING.SIMPLE_LIST`: Mapeia para o método `simples_listar`, que lista registros com informações simplificadas.
- `PUT Constants.URL_NAMING.UPDATE`: Mapeia para o método `atualizar`, responsável por atualizar os dados de um registro existente.
- `DELETE Constants.URL_NAMING.DELETE`: Mapeia para o método `remover`, que exclui um registro do sistema.

Esta estrutura permite que qualquer novo *Controller* que siga o padrão CRUD seja rapidamente configurado com um conjunto de rotas consistente, reduzindo a repetição de código e facilitando a manutenção.

A Função *ComentarioRoutes* (exemplo)

A função *ComentarioRoutes* segue a mesma estrutura de *BaseRoutes*, mas adiciona uma rota específica para a entidade Comentário. Além das operações CRUD padrão, essa função define uma rota adicional:

- `POST /revisao/listar`: Mapeia para o método `revisao_listar` do *Controller*, que lista revisões associadas a comentários.

Com esta implementação as Routes podem ser facilmente estendidas para acomodar funcionalidades específicas de cada entidade, sem comprometer a organização geral da aplicação.

Configuração Geral das Rotas (RoutesConfig)

A função *RoutesConfig* é responsável por inicializar todas as rotas da aplicação. Ela recebe como parâmetro a instância da aplicação *Express* (`app`) e utiliza a função `app.use()` para associar cada conjunto de rotas a um caminho base (prefixo de URL).

Exemplo de configuração de rotas:

- `app.use("/album", BaseRoutes(new AlbumController(models.album)));`
- `app.use("/autenticacao", AutenticacaoRoutes(new AutenticacaoController(models.utilizador)));`
- `app.use("/comentario", ComentarioRoutes(new ComentarioController(models.comentario)));`

Neste exemplo, as rotas associadas ao `AlbumController` serão acessíveis a partir do caminho `/album`, enquanto as rotas associadas ao `ComentarioController` estarão disponíveis em `/comentario`. Essa abordagem modular permite que a aplicação seja facilmente escalável, com novas rotas sendo adicionadas conforme necessário, mantendo a coesão e a organização.

3.2.2.5 Services

Os Services (serviços) na aplicação desempenham um papel crucial na lógica de negócios, atuando como intermediários entre os Controllers e os Models. Eles encapsulam a lógica de manipulação de dados e interações com o banco de dados, garantindo que as operações de criação, leitura, atualização e exclusão (CRUD) sejam realizadas de maneira consistente e segura. Nesta seção, abordaremos a implementação das classes Service e BaseService, que fornecem a estrutura necessária para criar serviços específicos para diferentes modelos de dados.

3.2.2.5.1 Classe Service

A classe Service é a classe base para todos os serviços na aplicação. Ela fornece a infraestrutura básica para interagir com um modelo de dados específico.

```
export class Service {  
  constructor(model, identifier = Constants.DEFAULT_IDENTIFIER) {  
    this.model = model;  
    this.identifier = identifier;  
  }  
}
```

Construtor: O construtor da classe Service recebe dois parâmetros:

- **model:** O modelo de dados que o serviço manipulará. Esse modelo representa uma tabela específica no banco de dados.
- **identifier:** Um identificador padrão (geralmente uma chave primária como id) que será utilizado para buscar registros no banco de dados. Se nenhum identificador for fornecido, será usado um identificador padrão definido em Constants.DEFAULT_IDENTIFIER.

Objetivo: Esta classe é utilizada para inicializar qualquer serviço específico com o modelo de dados correspondente, garantindo que todas as operações subsequentes sejam realizadas no contexto do modelo associado.

3.2.2.5.2 Classe *BaseService*

A classe *BaseService* estende a funcionalidade da classe *Service*, implementando métodos concretos para as operações CRUD. Esta classe serve como uma solução genérica que pode ser facilmente adaptada a diferentes modelos, fornecendo métodos padrões para as operações mais comuns.

a. Método buscar

```
async buscar(id, identifier = null, manual_models = []) {
  try {
    const response = await this.model.findOne({
      where: { [identifier ? identifier : this.identifier]: id },
      include: [...manual_models],
    });
    if (!response) throw new NotFoundException("Objeto não encontrado.");
    return response;
  } catch (e) {
    throw new ServerException(e);
  }
}
```

Objetivo: Buscar um registo específico baseado no identificador fornecido. O método aceita identificadores personalizados (*identifier*) e pode incluir associações com outros modelos (*manual_models*).

Funcionamento: Se o registo não for encontrado, uma exceção *NotFoundException* é lançada. Caso contrário, o registo é retornado.

b. Método obter

```
async obter(id, manual_models = []) {
  try {
    const response = await this.model.findOne({
      where: { [this.identifier]: id },
      include: [...ControllersUtils.modelsDirectlyAssociated(this.model), ...manual_models],
    });
    if (!response) throw new NotFoundException("Objeto não encontrado.");
    return response;
  } catch (e) {
    throw new ServerException(e);
  }
}
```


Objetivo: Similar ao método buscar, mas com a inclusão automática de modelos diretamente associados ao modelo principal, facilitando a recuperação de dados relacionados.

c. Método simples_obter

```
async simples_obter(id, manual_models = []) {  
  try {  
    const response = await this.model.findOne({  
      where: { [this.identifier]: id },  
      include: [...manual_models],  
    });  
    if (!response) throw new NotFoundException("Objeto não encontrado.");  
    return response;  
  } catch (e) {  
    throw new ServerException(e);  
  }  
}
```

Objetivo: Recuperar um registro específico com um conjunto simplificado de associações. Esse método é útil quando informações detalhadas não são necessárias.

d. Método criar

```
async simples_obter(id, manual_models = []) {  
  try {  
    const response = await this.model.findOne({  
      where: { [this.identifier]: id },  
      include: [...manual_models],  
    });  
    if (!response) throw new NotFoundException("Objeto não encontrado.");  
    return response;  
  } catch (e) {  
    throw new ServerException(e);  
  }  
}
```

Objetivo: Criar um novo registro no banco de dados.

Funcionamento: Antes de criar o registro, os dados são validados utilizando a utilidade `ModelsUtils.validateModelData`. Se a criação for bem-sucedida, o novo registro é retornado.

e. e. Método listar

```
async listar(query, manual_models = [], removeDirectlyAssociatedModels = false) {
  try {
    const includeModels = removeDirectlyAssociatedModels
      ? [...manual_models]
      : [...ControllersUtils.modelsDirectlyAssociated(this.model), ...manual_models];

    const response = await this.model.findAll({
      where: { ...query },
      include: includeModels,
      order: [
        ["data_criacao", "DESC"],
        ["id", "DESC"],
      ],
    });
    if (!response) throw new NotFoundException("Objeto não encontrado.");
    return response;
  } catch (e) {
    throw new ServerException(e);
  }
}
```

Objetivo: Listar registros com base em uma consulta específica (query).

Funcionamento: Este método pode incluir ou excluir associações com modelos relacionados, dependendo do valor de *removeDirectlyAssociatedModels*. Os resultados são ordenados por data de criação e ID em ordem decrescente.

f. Método simples_listar

```
async simples_listar(query, manual_models = []) {  
  try {  
    const response = await this.model.findAll({  
      where: { ...query },  
      include: [...manual_models],  
      order: [  
        ["data_criacao", "DESC"],  
        ["id", "DESC"],  
      ],  
    });  
    if (!response) throw new NotFoundException("Objeto não encontrado.");  
    return response;  
  } catch (e) {  
    throw new ServerException(e);  
  }  
}
```

Objetivo: Similar ao método listar, mas retorna um conjunto simplificado de registros, ideal para situações em que menos detalhes são necessários.

g. Método atualizar

```
async atualizar(id, data) {  
  try {  
    const response = await this.model.update(data, {  
      where: { [this.identifier]: id },  
    });  
    if (!response) throw new NotFoundException("Objeto não encontrado.");  
    return response;  
  } catch (e) {  
    throw new ServerException(e);  
  }  
}
```

Objetivo: Atualizar um registo existente com base no identificador fornecido e nos dados novos.

Funcionamento: Se a atualização não for bem-sucedida, uma exceção é lançada.

h. Método remover

```
async remover(id) {
  try {
    const response = await this.model.destroy({
      where: { [this.identifier]: id },
    });
    if (!response) throw new NotFoundException("Objeto não encontrado.");
    return response;
  } catch (e) {
    throw new ServerException(e);
  }
}
```

Objetivo: Remover um registo específico do banco de dados.

Funcionamento: O método utiliza o identificador para localizar e excluir o registo. Caso a operação falhe, uma exceção é lançada.

i. Método `remover_query`

```
async remover_query(query) {
  try {
    const response = await this.model.destroy({
      where: { ...query },
    });
    if (!response) throw new NotFoundException("Objeto não encontrado.");
    return response;
  } catch (e) {
    throw new ServerException(e);
  }
}
```

Objetivo: Remover múltiplos registos que correspondam a uma determinada consulta (query).

Funcionamento: Este método é útil para exclusões em massa, onde vários registos precisam ser removidos ao mesmo tempo.

3.2.2.5.3 Classe *AuthService*

A `AuthService` centraliza toda a lógica relacionada à autenticação e autorização de utilizadores na aplicação. Lida com a criação e verificação de tokens JWT (JSON Web Tokens) para várias finalidades, como autenticação de acesso, recuperação de senha e atualização de senha.

Manipulação de Tokens JWT:

- Criação de Tokens: Métodos como `createAuthToken`, `createForgetPasswordToken`, e `createAtualizarPasswordToken` são responsáveis por gerar tokens JWT usando diferentes segredos, conforme necessário. Estes tokens são usados para autenticar utilizadores ou realizar ações seguras, como redefinir senhas.
- Verificação de Tokens: Métodos como `verifyAuthToken`, `verifyForgetPasswordToken`, e `verifyAtualizarPasswordToken` garantem que os tokens recebidos nas requisições são válidos e não foram adulterados, o que é crucial para manter a segurança das operações sensíveis.

Integração com Serviços Externos:

O método `verifyGoogleLoginToken` lida com a verificação de *tokens* de login do Google, facilitando o login de utilizadores através de contas do Google, uma funcionalidade comum em muitas aplicações modernas.

Comparação de Senhas:

O método `comparePassword` usa `bcrypt` para comparar as senhas fornecidas pelos utilizadores com as senhas armazenadas na base de dados de forma segura, garantindo que apenas utilizadores autenticados possam aceder à aplicação.

Controlo de Acesso: O método `hasPermission` serve como uma base para a verificação de permissões de utilizador, permitindo uma camada adicional de segurança que pode ser adaptada conforme as necessidades da aplicação.

3.2.2.5.4 Classe *AuthLoginService*

A *AuthLoginService* complementa a *AuthService* ao focar nas operações de login e criação de utilizadores.

Login de Utilizadores:

O método `entrar` é responsável por autenticar utilizadores com base nas suas credenciais (e-mail/tag e senha). Também suporta logins externos, onde a senha pode ser temporariamente definida ou ignorada, facilitando a integração com provedores de login como Google ou Facebook.

Criação de Utilizadores:

O método `criar` lida com a criação de novos utilizadores, incluindo a definição de uma senha temporária e o envio de um e-mail de boas-vindas. Este método automatiza e simplifica o processo de registo de novos utilizadores, garantindo que todas as etapas necessárias sejam seguidas (como envio de e-mail e geração de senhas).

Utilidade da *AuthLoginService*:

Esta classe facilita a gestão de login e criação de utilizadores, garantindo que as operações sejam seguras e eficientes, com suporte para autenticação tradicional e integração com provedores externos.

3.2.2.5.5 Classe *CloudStorageService*

A *CloudStorageService* gere o upload de ficheiros para um serviço de armazenamento na nuvem (neste caso, *Cloudinary*).

Upload de Ficheiros:

O método `upload` é responsável por enviar múltiplos ficheiros para uma pasta específica na nuvem. Ele utiliza um método privado (`#uploader`) para fazer o upload individual de cada ficheiro, garantindo que a operação seja tratada de forma consistente e segura.

Tratamento de Erros:

Em caso de falhas durante o upload, a classe lança exceções personalizadas (`CloudinaryException`), facilitando a identificação e o tratamento de problemas durante o processo de upload.

Utilidade da *CloudStorageService*:

Este serviço abstrai a complexidade da integração com serviços de armazenamento na nuvem, oferecendo uma interface simples e reutilizável para o upload de ficheiros.

3.2.2.5.6 Classe *MulterService*

A `MulterService` utiliza o middleware `Multer` para gerir o upload de ficheiros diretamente através de requisições HTTP.

Upload com Multer:

O método `upload` encapsula a configuração e execução do upload de ficheiros utilizando o `Multer`, permitindo que múltiplos ficheiros sejam processados de forma segura e eficiente.

Promessa para Upload:

O método retorna uma promessa, o que facilita a integração com a lógica assíncrona da aplicação, permitindo que os desenvolvedores tratem os ficheiros carregados com facilidade.

Utilidade da *MulterService*:

Fornece uma solução pronta para o upload de ficheiros no servidor, simplificando o processo e permitindo uma fácil integração com a lógica de negócio da aplicação.

3.2.2.5.7 Classe *UploadService*

A `UploadService` combina a funcionalidade de dois serviços distintos, `MulterService` e `CloudStorageService`, para facilitar o upload de ficheiros tanto para armazenamento local quanto para a nuvem.

Processamento de Ficheiros:

Upload Local e para a Nuvem: A `UploadService` lida com o upload de ficheiros, primeiro utilizando o `MulterService` para armazenar os ficheiros localmente, e depois usando o `CloudStorageService` para enviá-los para um serviço de armazenamento na nuvem, como o Cloudinary. Isto permite que a aplicação tenha uma solução híbrida de armazenamento, garantindo que os ficheiros estejam disponíveis tanto localmente quanto na nuvem.

Formatação de Caminhos de Ficheiros:

O método `formatPathsArray` é utilizado para extrair e formatar os caminhos dos ficheiros carregados, facilitando a gestão dos mesmos em operações subsequentes.

Tratamento de Exceções:

Caso ocorra algum erro durante o processo de upload, a classe lança uma `UploadException`, garantindo que os problemas possam ser capturados e tratados de forma eficaz.

Utilidade da `UploadService`:

Esta classe simplifica o processo de upload, oferecendo uma interface unificada para o carregamento de ficheiros em múltiplos destinos, enquanto abstrai a complexidade subjacente.

3.2.2.5.8 Classe *ResponseService*

A `ResponseService` oferece uma interface padronizada para a formatação de respostas HTTP enviadas ao cliente, melhorando a consistência e a clareza das respostas da API.

Respostas de Sucesso e Erro:

Success: O método `success` envia uma resposta de sucesso padrão, incluindo os dados retornados pela operação.

Error: O método `error` trata erros e envia uma resposta apropriada, registando também o erro no log e exibindo o stack trace, o que facilita o diagnóstico de problemas.

Respostas de Acesso Não Autorizado:

Unauthorized: O método `unauthorized` lida com situações em que o utilizador não tem permissão para aceder a um recurso, retornando uma resposta HTTP adequada.

Mensagens Personalizadas:

Message: Permite o envio de mensagens personalizadas juntamente com uma resposta, seja ela de sucesso ou falha, dando flexibilidade na comunicação de informações específicas ao cliente.

Utilidade da ResponseService:

Esta classe garante que as respostas da API sejam consistentes e bem estruturadas, melhorando a experiência do utilizador e facilitando a depuração e manutenção da aplicação.

3.2.2.5.9 Classe ScheduleService

A ScheduleService gere eventos agendados e o envio de notificações ou e-mails associados a esses eventos.

Verificação de Eventos Próximos:

O método `verificaProximoEventoOuAtividade` percorre os conteúdos armazenados para identificar atividades ou eventos que estão próximos de ocorrer. Quando encontra um evento próximo, notifica os utilizadores subscritos, assegurando que eles estão cientes do evento a tempo.

Envio de E-mails Relacionados a Alterações de Conteúdo:

`enviaEmailDeAlteracoes`: Este método envia e-mails a todos os utilizadores associados a um conteúdo quando há atualizações, mantendo-os informados sobre mudanças importantes.

Notificação de Novas Participações:

`enviaEmailNovaParticipacao`: Quando um novo participante é adicionado a um conteúdo, este método notifica o criador do conteúdo via e-mail, promovendo a interação e colaboração dentro da aplicação.

Notificação de Interessados:

`enviaEmailParaInteressados`: Quando há atualizações em conteúdos de interesse, este método envia e-mails aos utilizadores que expressaram interesse, garantindo que eles não percam informações relevantes.

Utilidade da `ScheduleService`:

Este serviço é essencial para a gestão e comunicação de eventos e atualizações, automatizando o processo de notificação e assegurando que os utilizadores são informados de maneira oportuna.

3.2.2.6 *Middleware*

Os middlewares são componentes fundamentais na arquitetura de uma aplicação Express.js. Eles interceptam e processam as requisições HTTP antes que estas cheguem às rotas finais, permitindo a implementação de funcionalidades transversais como autenticação, logging e tratamento de erros. A seguir, analisamos três middlewares específicos: `LoggerMiddleware`, `AuthMiddleware`, e `ErrorMiddleware`, discutindo sua utilidade e como são instanciados dentro da aplicação.

Utilidade dos Middlewares

- **Interceção e Processamento:** Os *middlewares* permitem a interceção de requisições HTTP antes que estas sejam processadas pelas rotas principais, possibilitando a aplicação de lógica adicional, como verificação de autenticação, logging, manipulação de dados da requisição, ou mesmo a interrupção da requisição se alguma condição não for cumprida.
- **Reutilização de Código:** Ao encapsular lógica comum em middlewares, evita-se a repetição de código em múltiplas rotas. Por exemplo, verificar a autenticidade de um token JWT ou registrar logs pode ser feito de forma centralizada, aplicando o middleware a todas as rotas que necessitam dessa funcionalidade.
- **Fluxo de Execução Controlado:** Os middlewares controlam o fluxo da aplicação através do método `next()`, que permite que a execução continue para o próximo middleware ou rota. Isto é crucial para garantir que as operações sejam realizadas na ordem correta e que erros possam ser capturados e tratados adequadamente.

Análise dos Middlewares

1. *AuthMiddleware*

Propósito: O *AuthMiddleware* é responsável pela autenticação dos utilizadores. Ele extrai o token de autenticação do cabeçalho da requisição, verifica a sua validade e, se o token for

válido, associa o utilizador correspondente ao objeto *req* para que ele esteja disponível nas rotas subsequentes.

Funcionamento: Primeiro, o middleware tenta obter o token de autenticação através do método *getTokenHeader* da *AuthService*. Se o token estiver presente, ele é verificado usando o método *verifyAuthToken*. Se a verificação for bem-sucedida, o utilizador correspondente é recuperado através de *getUserById* e anexado ao objeto *req*. Isto permite que as rotas seguintes acessem ao utilizador autenticado diretamente através de *req.user*.

Utilidade: Este middleware é crucial para proteger rotas que requerem autenticação, garantindo que apenas utilizadores autenticados possam aceder a determinados recursos.

Instanciação: O *AuthMiddleware* é instanciado e aplicado globalmente à aplicação através da função *RoutingConfig*, que o adiciona ao pipeline de *middlewares* da aplicação.

2. **ErrorMiddleware**

Propósito: O *ErrorMiddleware* lida com erros que ocorrem durante o processamento das requisições. Ele é normalmente o último middleware na cadeia de execução, capturando qualquer erro que não tenha sido tratado anteriormente.

Funcionamento: Quando um erro ocorre, este middleware é ativado, recebendo o erro como parâmetro. Ele utiliza o *ResponseService* para formatar e enviar uma resposta de erro ao cliente, garantindo que o erro seja reportado de forma consistente.

Utilidade: Ao centralizar o tratamento de erros, este middleware melhora a robustez da aplicação, evitando que exceções não tratadas resultem em comportamentos inesperados ou quebras na aplicação.

Instanciação: O *ErrorMiddleware* também é instanciado dentro da função *RoutingConfig* e é colocado após as rotas da aplicação, garantindo que qualquer erro durante o processamento das requisições seja capturado e tratado.

3. **LoggerMiddleware**

Propósito: O *LoggerMiddleware* é responsável por registrar informações sobre as requisições que chegam à aplicação, como método HTTP, URL no ficheiro *access.log*.

Funcionamento: Geralmente, um middleware de *logging* captura informações sobre cada requisição, como o método HTTP, a rota, o status de resposta, e o tempo de processamento, registrando estas informações em *logs* para posterior análise.

Utilidade: Este middleware é essencial para monitorar o tráfego e desempenho da aplicação, bem como para a depuração, ao permitir que os desenvolvedores acompanhem o comportamento da aplicação em tempo real ou analisem *logs* históricos.

Instanciação: Assim como os outros *middlewares*, o *LoggerMiddleware* é instanciado dentro de *RoutingConfig*, sendo provavelmente o primeiro a ser executado para garantir que todas as requisições sejam registadas.

Instanciação dos Middlewares

Todos os *middlewares* são instanciados dentro da função *RoutingConfig*. Esta função é responsável por configurar a aplicação Express.js, aplicando os *middlewares* na ordem correta e garantindo que eles estejam ativos para processar as requisições.

Ordem de Execução: A ordem dos *middlewares* é crucial. Primeiro, é aplicado o *LoggerMiddleware* para registar a requisição. Em seguida, o *AuthMiddleware* verifica a autenticação do utilizador. As rotas são então processadas, e finalmente, qualquer erro que ocorra é capturado pelo *ErrorMiddleware*.

```
export const RoutingConfig = async (app) => {  
  app.use(LoggerMiddleware);  
  app.use(AuthMiddleware);  
  
  RoutesConfig(app);  
  
  app.use(ErrorMiddleware);  
  
  LogUtils.log("Middlewares inicializados!", LogUtils.TIPO.MIDDLEWARES);  
};
```

Figura 8 - Instanciação dos *middlewares*

3.2.3. Estrutura

A API está estruturada da seguinte forma:

- **config:** Este diretório geralmente contém ficheiros de configuração, como variáveis de ambiente, configuração de base de dados e outras definições globais que a aplicação possa necessitar. Essas configurações são utilizadas por outras partes da aplicação para garantir que as mesmas informações e parâmetros sejam usados de forma consistente.
- **constants:** O diretório constants armazena constantes que são usadas em toda a aplicação, como códigos de erro, mensagens padrão, e quaisquer outros valores imutáveis que precisam estar disponíveis globalmente.
- **controllers:** Em controllers estão as funções responsáveis por manipular as requisições e respostas HTTP. Eles fazem a ponte entre as rotas e a lógica de negócio. Um controller normalmente chama métodos de serviços para realizar operações e devolver os resultados ao cliente.
- **exceptions:** Este diretório é destinado ao tratamento de exceções e erros personalizados. Aqui podem estar definidos tipos específicos de erros que são lançados e manipulados em diferentes partes da aplicação.
- **helpers:** helpers contém funções utilitárias que ajudam a realizar tarefas repetitivas ou comuns, como manipulação de datas, formatação de dados, entre outros. Essas funções são reutilizáveis e podem ser chamadas a partir de qualquer parte do código.
- **middlewares:** Em middlewares, encontram-se as funções que interceptam as requisições antes que elas cheguem aos controllers. Elas podem realizar tarefas como autenticação, validação de dados, logging, etc. Os middlewares são uma parte fundamental do fluxo de requisições em aplicações Express.
- **models:** O diretório models armazena as definições das entidades da aplicação, normalmente mapeando para tabelas de base de dados. Estes modelos representam a estrutura dos dados e contêm métodos para realizar operações de base de dados, como criar, ler, atualizar e eliminar registos.
- **routes:** Em routes estão definidos os endpoints da API. Cada rota está associada a um controller que executa a lógica necessária para responder às requisições. As rotas organizam como as requisições HTTP são tratadas, direcionando-as para os controllers adequados.

- **services:** O diretório `services` contém a lógica de negócio da aplicação. Enquanto os `controllers` lidam com a interface HTTP, os serviços realizam operações mais complexas que podem envolver múltiplos modelos ou interações externas, como chamadas a APIs de terceiros.
- **utils:** `utils` é semelhante a `helpers`, mas costuma armazenar funções utilitárias mais genéricas que não estão diretamente ligadas à lógica de negócio. Pode incluir, por exemplo, funções de manipulação de ficheiros, criptografia, etc.
- **app.js:** Este é geralmente o ponto de entrada da aplicação. O ficheiro `app.js` configura o servidor, carrega os `middlewares` globais, define as rotas principais e inicia a aplicação. É aqui que o servidor Express é configurado e inicializado.

3.3. Web

3.3.1. Estrutura do Projeto

3.3.2. Estrutura da Aplicação

3.3.3. Rotas

3.3.4. Componentes

Neste capítulo vão ser apresentados os componentes mais utilizados/importantes. Vai ser explicado como foi implementado e a sua utilidade.

3.3.4.1 Header e Drawer

Para implementar o *Header* e *Drawer*, como são componentes que vão aparecer em todas as páginas foram implementados no *PageLayout* (que foi documentado e explicado no capítulo 3.3.2).

O componente *Drawer* foi implementado para fornecer um menu de navegação lateral que adapta seu comportamento com base no estado de abertura e no tamanho da janela. Abaixo, destaco os principais pontos da implementação:

Estruturação dos Componentes:

- A implementação é dividida em dois componentes principais: *Drawer* e *Item*.
- O componente *Item* é responsável por renderizar individualmente os itens do menu, enquanto o componente *Drawer* gerencia o conjunto desses itens e o comportamento do menu como um todo.

Componente *Item*:

- Este componente recebe como props a rota (route), o ícone (icone) e o rótulo (label) do item de navegação.
- Utiliza o hook `useLocation` do React Router para monitorar a rota atual e determinar se o item está selecionado.
- O estilo do item muda dinamicamente com base no estado de seleção, utilizando classes CSS condicionais.
- Se o drawer estiver aberto, o texto associado ao item de navegação é exibido ao lado do ícone; caso contrário, apenas o ícone é mostrado.

Componente Drawer:

- O Drawer gerencia o estado de abertura do menu através do hook customizado `useDrawerStatus`.
- O menu ajusta sua largura dinamicamente entre dois valores (`DRAWER_CLOSE_WIDTH` e `DRAWER_OPEN_WIDTH`), com uma transição suave de 0,3s para melhorar a experiência do usuário.
- Um hook de redimensionamento (`useOnResize`) foi utilizado para fechar o drawer automaticamente quando a largura da janela é menor que 1300 pixels, tornando a interface mais responsiva.
- O conteúdo do menu é renderizado dinamicamente através da função `Render` do objeto `NavItems`, que agrupa todos os itens de navegação.

3.3.4.2 Mapa

O componente de Mapa Interativo foi criado para permitir que os usuários escolham uma localização no mapa e obtenham o endereço exato desse local. Ele também oferece a opção de buscar um endereço digitando-o em uma caixa de texto. Abaixo, explico como funciona:

Configuração do Mapa:

- O mapa que aparece na tela é gerado usando uma ferramenta chamada react-leaflet, que permite a visualização de mapas e a interação com eles diretamente no navegador. O mapa utiliza dados da OpenStreetMap, uma fonte de mapas gratuita e aberta.

Seleção de Localização:

- Quando o usuário clica em algum ponto do mapa, o sistema registra as coordenadas (latitude e longitude) desse ponto. Essas coordenadas são como "endereços" em formato numérico que o mapa entende.
- Após clicar no mapa, ele se centraliza automaticamente no ponto selecionado, mostrando exatamente onde o usuário clicou.

Busca de Endereço:

- O usuário pode digitar um endereço na caixa de texto fornecida. Quando o usuário clica no botão de busca, o sistema envia esse endereço para um serviço online que tenta encontrar as coordenadas correspondentes.
- Se o endereço for encontrado, o mapa se ajusta automaticamente para mostrar a localização desse endereço. Caso o endereço não seja encontrado, o sistema informa ao usuário que a busca falhou.

Confirmação da Localização:

- Depois de selecionar uma localização no mapa, o usuário pode confirmar essa escolha clicando no botão "Confirmar". O sistema então obtém o endereço detalhado dessa localização (como o nome da rua, bairro, cidade, etc.) e envia essa informação de volta ao sistema que está usando o componente de mapa.
- Esse endereço e as coordenadas selecionadas são armazenados e podem ser usados para outras finalidades, como preencher um formulário ou atualizar um cadastro.

Interface Simples e Intuitiva:

- O componente foi projetado para ser fácil de usar. Além do mapa, há uma barra de pesquisa para encontrar endereços e botões claros para realizar ações como buscar um endereço e confirmar a localização escolhida.
- Tudo foi feito para que o usuário possa interagir com o mapa de forma natural, sem precisar de conhecimentos técnicos.

3.3.4.3 Tabelas

O componente Tabela foi desenvolvido para fornecer uma tabela flexível e paginável, com colunas personalizáveis, utilizando componentes do Material UI. Abaixo, explico os principais aspectos da implementação:

Estrutura do Componente:

- O componente Tabela recebe vários parâmetros que permitem a personalização das colunas, das linhas, das opções de número de linhas por página e do número padrão de linhas por página.
- As props permitem que o componente seja reutilizado em diferentes contextos com diferentes configurações.

Estado Interno:

- Dois estados principais são gerenciados dentro do componente: `page`, que controla a página atual da tabela, e `rowsPerPage`, que define quantas linhas são exibidas por página.
- O `useState` do React é utilizado para gerenciar esses estados, garantindo que a tabela responda corretamente às interações do usuário, como mudanças de página ou de número de linhas por página.

Manipulação de Páginas e Linhas por Página:

- Funções de manipulação (`handleChangePage` e `handleChangeRowsPerPage`) são implementadas para atualizar os estados `page` e `rowsPerPage` conforme o usuário navega pelas páginas ou altera o número de linhas exibidas por página.
- Essas funções garantem que a tabela seja re-renderizada adequadamente com os dados corretos para a página selecionada.

Estruturação da Tabela:

- O componente utiliza os elementos `Table`, `TableHead`, `TableBody`, `TableRow` e `TableCell` do Material UI para construir a tabela.
- As colunas são geradas dinamicamente com base na prop `columns`, que contém as definições das colunas, incluindo o ID, alinhamento (`align`), largura mínima (`minWidth`) e o rótulo (`label`).
- As linhas são fatiadas de acordo com a página atual e o número de linhas por página, e são renderizadas dentro do `TableBody`. Cada célula (`TableCell`) é preenchida com os dados correspondentes, e, se um formato especial for definido para a coluna (por exemplo, para números), ele é aplicado.

Paginação:

- A paginação da tabela é gerenciada pelo componente `TablePagination` do Material UI, que exibe os controles de navegação entre as páginas e de ajuste do número de linhas por página.
- A configuração de `rowsPerPageOptions` permite que o usuário escolha entre diferentes quantidades de linhas a serem exibidas.

Estilização e Layout:

- A tabela é renderizada dentro de um Paper do Material UI, com estilo que garante que a tabela ocupe 100% da largura disponível e que seu conteúdo seja rolável se exceder a altura máxima (maxHeight).
- A propriedade stickyHeader do componente Table é utilizada para manter os cabeçalhos das colunas fixos no topo durante o scroll, melhorando a usabilidade em tabelas grandes.

3.3.5. Comunicação com a API

Na aplicação, a comunicação com a API é fundamental para o funcionamento correto, permitindo a interação com a api para operações como autenticação, criação e manipulação de dados. Abaixo está uma explicação geral sobre a implementação das classes *AutenticacaoRequest* e *ApiRequest*, que gerenciam essa comunicação de forma estruturada e eficiente.

3.3.5.1 Classe *AutenticacaoRequest*

A classe *AutenticacaoRequest* é responsável por gerir todas as operações relacionadas à autenticação de utilizadores na aplicação. Esta classe facilita o processo de iniciar sessão, terminar sessão, atualizar *tokens* de acesso e recuperar ou redefinir senhas. Cada método utiliza a função *myAxios* para enviar requisições à API, garantindo que todas as comunicações sejam seguras e incluam o token de autenticação necessário.

- *entrar(login, senha)*: Envia uma requisição POST para autenticar o utilizador com as credenciais fornecidas. Se a autenticação for bem-sucedida, o token recebido é armazenado para futuras requisições.
- *terminar_sessao()*: Remove o token de autenticação do armazenamento local, efetivamente encerrando a sessão do utilizador.
- *obterUtilizadorAtual()*: Obtém as informações do utilizador atualmente autenticado, utilizando o token armazenado.
- *externalLogin(token)*: Gere o processo de autenticação com um token externo, útil para integrações com sistemas de login de terceiros.
- *atualizarToken()*: Atualiza o token de autenticação para prolongar a sessão do utilizador sem a necessidade de um novo login.
- *forgotPassword(email)*: Envia uma requisição para iniciar o processo de recuperação de senha para o email fornecido.
- *resetPassword(token, senha, confirmacao_senha)*: Permite ao utilizador redefinir sua senha utilizando um token de recuperação.

- `atualizarPasse(token, senha, senha_old)`: Atualiza a senha do utilizador, validando o token e a senha antiga antes de definir a nova senha.

3.3.5.2 Classe `ApiRequest`

A classe `ApiRequest` lida com operações CRUD (Criar, Ler, Atualizar, Apagar) gerais na aplicação, para diferentes tipos de dados. Esta classe proporciona métodos reutilizáveis para interagir com a API, independentemente do tipo de dado ou operação.

- `atualizar(endpoint, id, data)`: Atualiza um recurso específico identificado pelo id, enviando os novos dados para o endpoint correspondente.
- `criar(endpoint, data)`: Cria um novo recurso no backend, enviando os dados para o endpoint especificado.
- `upload_user_image(id, imagem)`: Faz o upload de uma imagem de perfil do utilizador, utilizando multipart/form-data para enviar o arquivo ao servidor.
- `criar_with_files(endpoint, data, file_key)`: Similar ao método `criar`, mas permite a inclusão de arquivos (como imagens ou documentos) nos dados enviados.
- `listar(endpoint, data, token)`: Lista recursos com base nos critérios fornecidos em data. Este método é utilizado para operações de busca ou filtragem.
- `obter(endpoint, id)`: Obtém detalhes de um recurso específico identificado pelo id.
- `remover(endpoint, id)`: Remove um recurso do servidor com base no id fornecido.

3.3.5.3 Integração com `Axios` e `Interceptores`

As classes acima utilizam *axios* como cliente HTTP para realizar as requisições à API. O *axios* é configurado para usar uma base URL comum (`API_URL`), e um intercetor de requisições é implementado para garantir que cada requisição inclua automaticamente o token de autenticação, se estiver disponível. Esse intercetor simplifica o processo de autorização, permitindo que o token seja gerido de forma centralizada pela aplicação.

3.3.5.4 Função *myAxios*

A função *myAxios* serve como um *wrapper* personalizado para *axios*, facilitando a configuração das requisições e o tratamento de erros. Ela recebe parâmetros como URL, método HTTP, dados a serem enviados, token adicional, e cabeçalhos personalizados, e retorna a resposta da API já processada ou um tratamento adequado de erros.

3.3.6. Páginas

3.3.6.1 Página Inicial

A página inicial contém uma *hero section* onde tem uma pequena introdução à aplicação e ao fazer scroll tem uma lista de conteúdos separado por tipos para os utilizadores poderem interagir. Na Figura 13 está ilustrada a página inicial e na Figura 14 está ilustrado a listagem por tipo dos conteúdos. Esta página está disponível apenas para utilizadores autenticados.

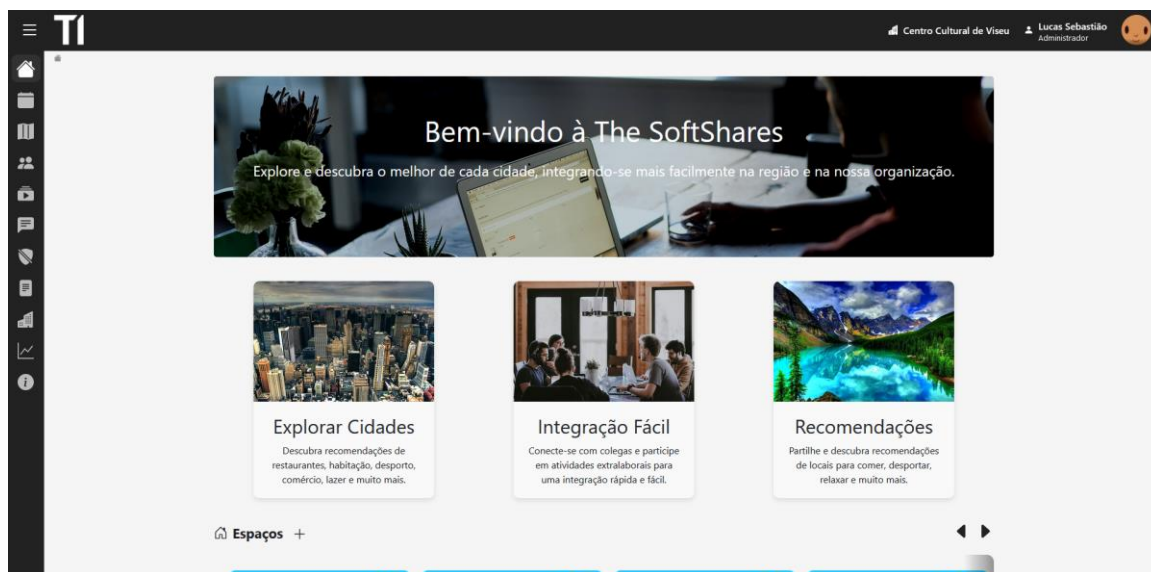


Figura 13 - Página Inicial *hero section*

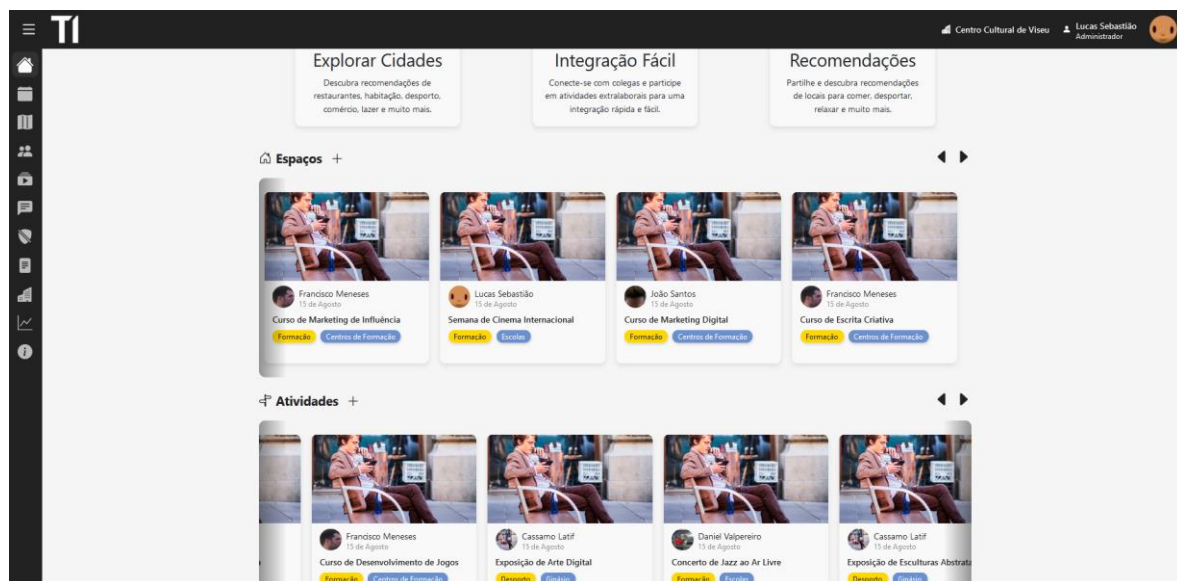


Figura 14 – Página: Inicial listagem

3.3.6.2 Calendário

No calendário aparecem apenas os conteúdos em que o utilizador está a participar. Qualquer utilizador autenticado pode ver esta página. Na Figura 15 está ilustrada a página.

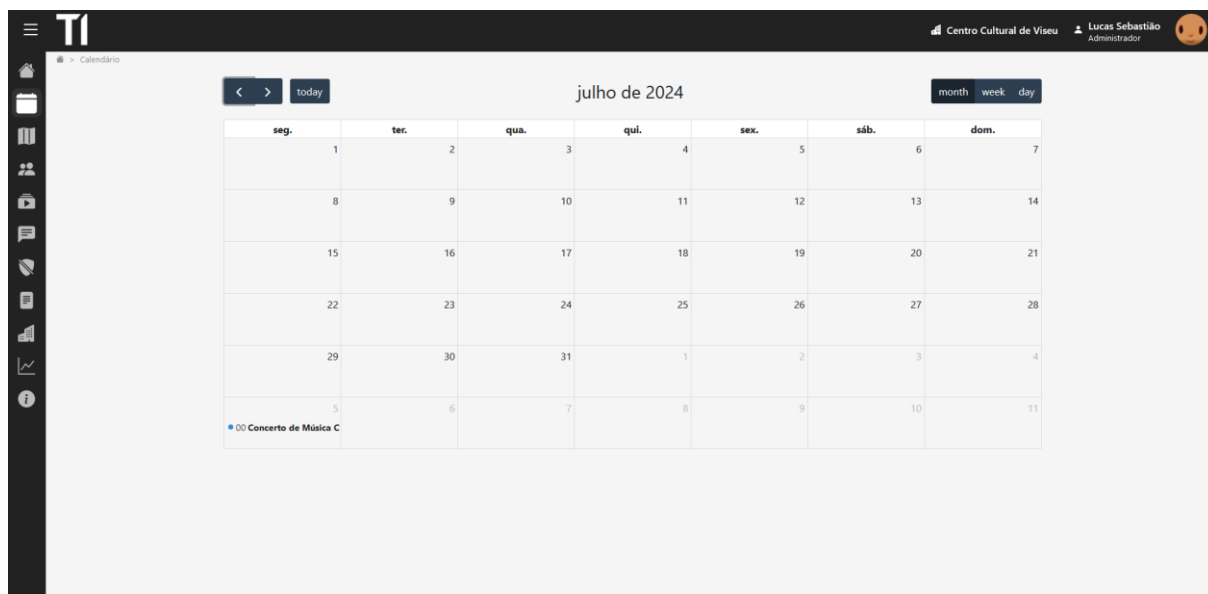


Figura 15 – Página: Calendário

3.3.6.3 Mapa

Nesta página é possível ver todos os conteúdos por localidade. Assim como o calendário, o mapa, está disponível para todos os utilizadores autenticados. Na Figura 16 está ilustrada a página.

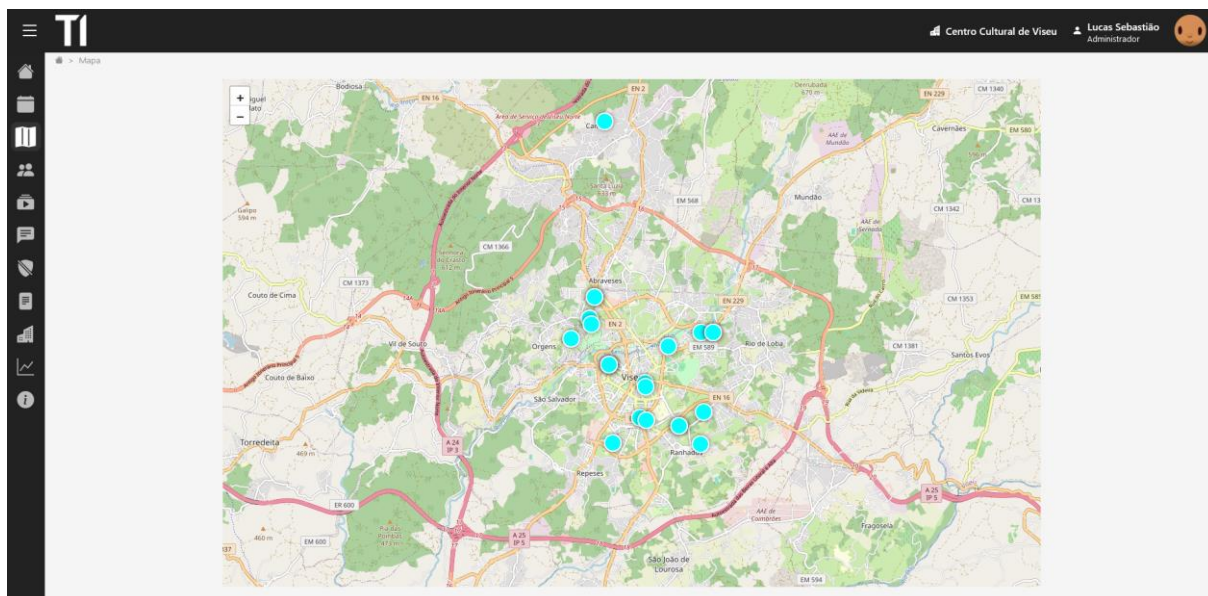
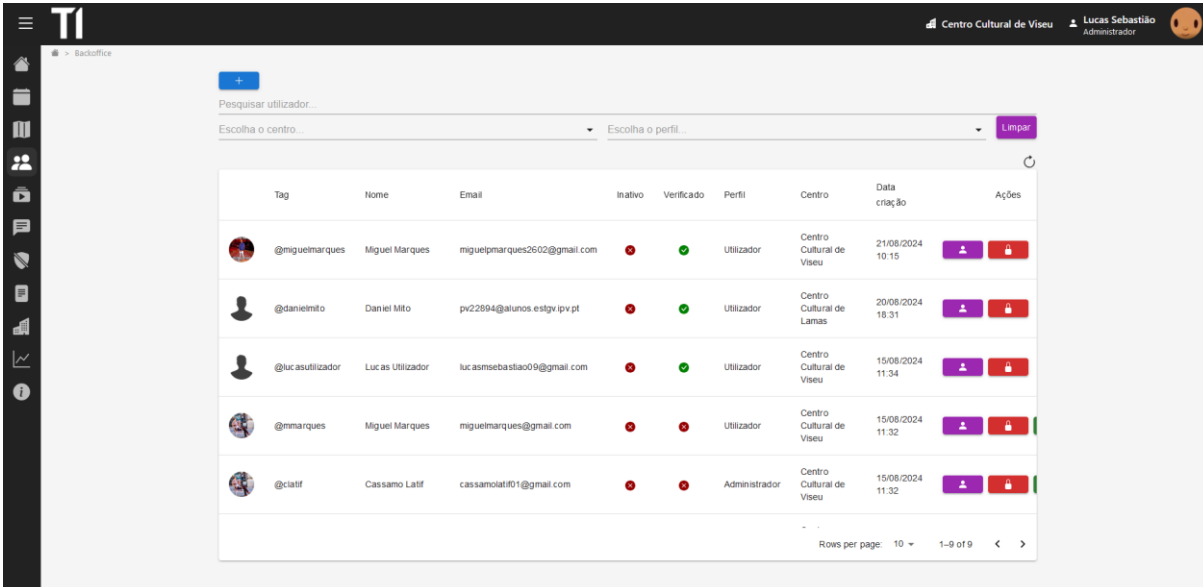


Figura 16 – Página: Mapa

3.3.6.4 Utilizadores (Backoffice)

Está página é possível gerir os utilizadores, como inativar a conta, verificar la (deixa de precisar de alterar a palavra-passe na primeira autenticação) e tem as informações básicas do utilizador, como nome, *tag*, email, perfil e centro. Também é possível filtrar os utilizadores, seja diretamente por texto através da *TextBox* ou pelo centro e perfil, através das *ComboBox* centro e perfil. Na Figura 17 está ilustrada a página.



TI Backoffice

Centro Cultural de Viseu | Lucas Sebastião Administrador

Pesquisar utilizador...

Escolha o centro... Escolha o perfil... Limpar

Tag	Nome	Email	Inativo	Verificado	Perfil	Centro	Data criação	Ações
@miguelmarques	Miguel Marques	miguelmarques2602@gmail.com	✗	✓	Utilizador	Centro Cultural de Viseu	21/08/2024 10:15	
@danielmito	Daniel Mito	pv22894@alunos.estgv.ipv.pt	✗	✓	Utilizador	Centro Cultural de Lamas	20/08/2024 18:31	
@lucasutilizador	Lucas Utilizador	lucasmsebastiao09@gmail.com	✗	✓	Utilizador	Centro Cultural de Viseu	15/08/2024 11:34	
@mmarques	Miguel Marques	miguelmarques@gmail.com	✗	✗	Utilizador	Centro Cultural de Viseu	15/08/2024 11:32	
@clatif	Cassano Latif	cassamolatif01@gmail.com	✗	✗	Administrador	Centro Cultural de Viseu	15/08/2024 11:32	

Rows per page: 10 1-9 of 9

Figura 17 - Página: Utilizadores (Backoffice)

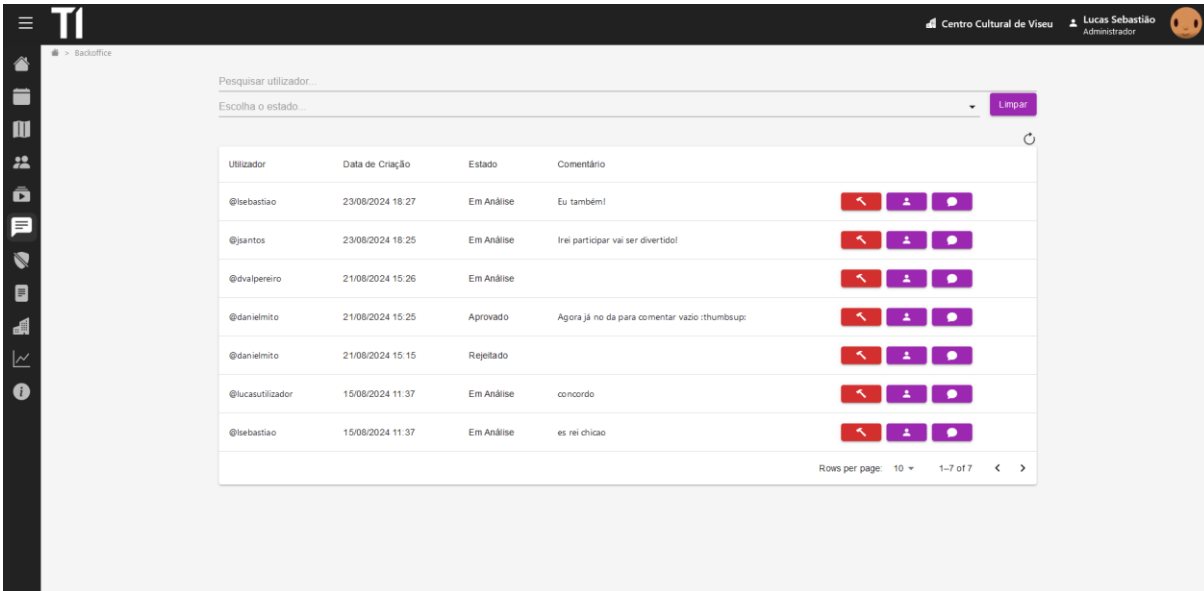
3.3.6.5 Conteúdos (Backoffice)

Nesta página é possível gerir os conteúdos e fazer a sua revisão. Para facilitar a revisão é possível filtrar os conteúdos pelo estado da revisão, através da *ComboBox*. Na coluna de ações existem várias opções, sendo possível, rever o conteúdo, aceitando ou não, ir para o perfil do utilizador que criou o conteúdo e por último ir para os detalhes do conteúdo. Na Figura 18 está ilustrada a página.

Tag	Data de Criação	Estado	Título	Tipo	Subtópico	Ações
@jsantos	23/08/2024 16:23	Aprovado	Jogo de futebol	Atividade	Outros...	
@lucasutilizador	23/08/2024 17:42	Em Análise	fdg	Espaço	Transportes públicos	
@lucasutilizador	23/08/2024 17:41	Em Análise	fdg	Espaço	Transportes públicos	
@lucasutilizador	23/08/2024 17:41	Em Análise	adsf	Espaço	Cinema	
@lucasutilizador	23/08/2024 17:39	Em Análise	teste	Espaço	Parques	
@sebastiao	21/08/2024 16:34	Aprovado	teste red	Atividade	Restaurantes	
@dvalpereiro	21/08/2024 16:24	Aprovado	teste	Atividade	Cinema	
@dvalpereiro	21/08/2024 15:33	Em Análise	asdasd	Evento	Quartos para arrendar	

Figura 18 - Página: Conteúdos (Backoffice)






















3.3.6.6 Comentários (Backoffice)



Pesquisar utilizador...

Escolha o estado ...

Limpar









Utilizador	Data de Criação	Estado	Comentário	
@lusebastiao	23/08/2024 18:27	Em Análise	Eu também!	  
@jsantos	23/08/2024 18:25	Em Análise	Irei participar vai ser divertido!	  
@dvalpereiro	21/08/2024 15:26	Em Análise		  
@danielmito	21/08/2024 15:25	Aprovado	Agora já não dá para comentar vazio :thumbsup:	  
@danielmito	21/08/2024 15:15	Rejeitado		  
@lucasutilizador	15/08/2024 11:37	Em Análise	concordo	  
@lusebastiao	15/08/2024 11:37	Em Análise	es rei chicoa	  

Rows per page: 10 1-7 of 7

Figura 19 - Página: Comentários (Backoffice)

3.3.6.7 Denúncias (Backoffice)

Quando um comentário é denunciado é adicionado na tabela na Figura 20, onde está ilustrada a página denúncias. Nesta página é possível filtrar os dados e gerir as denúncias, aceitar ou rejeitar, também é possível ir diretamente para o comentário em questão através da coluna ações.

Motivo	Data de Criação	Estado	Comentário	Denunciado	Denunciou	Ações
Denunciado 2	2024-08-25T13:12:28.648Z	Em Análise	concordo	@lucasutilizador	@lsebastiao	 
Denunciado	2024-08-25T13:12:16.016Z	Em Análise	Agora já não dá para comentar vazio :thumbsup:	@danielmito	@lsebastiao	 
a	2024-08-25T13:04:13.314Z	Em Análise	Irei participar vai ser divertido!	@jsantos	@lsebastiao	 
x	2024-08-25T13:04:10.549Z	Em Análise	Eu também!	@lsebastiao	@lsebastiao	 

Rows per page: 10 1-4 of 4

Figura 20 - Página: Denúncias (Backoffice)

3.3.6.8 Tópicos (Backoffice)

Na Figura 21 está ilustrada a página tópicos, onde é possível visualizar os tópicos e subtópicos existentes e caso necessário criar tópicos/subtópicos.

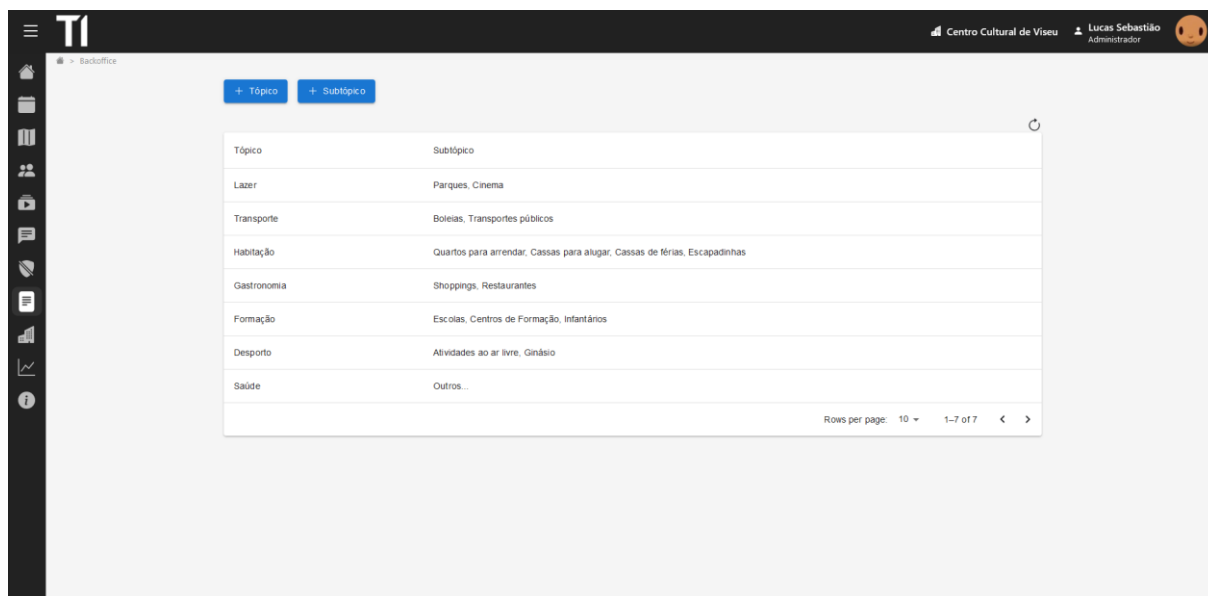


Figura 21 - Página: Tópicos (Backoffice)

3.3.6.9 Centros (Backoffice)

Parecida com a página tópicos, no capítulo 3.3.6.8, nesta página também é possível visualizar os centros e caso necessário criá-los.

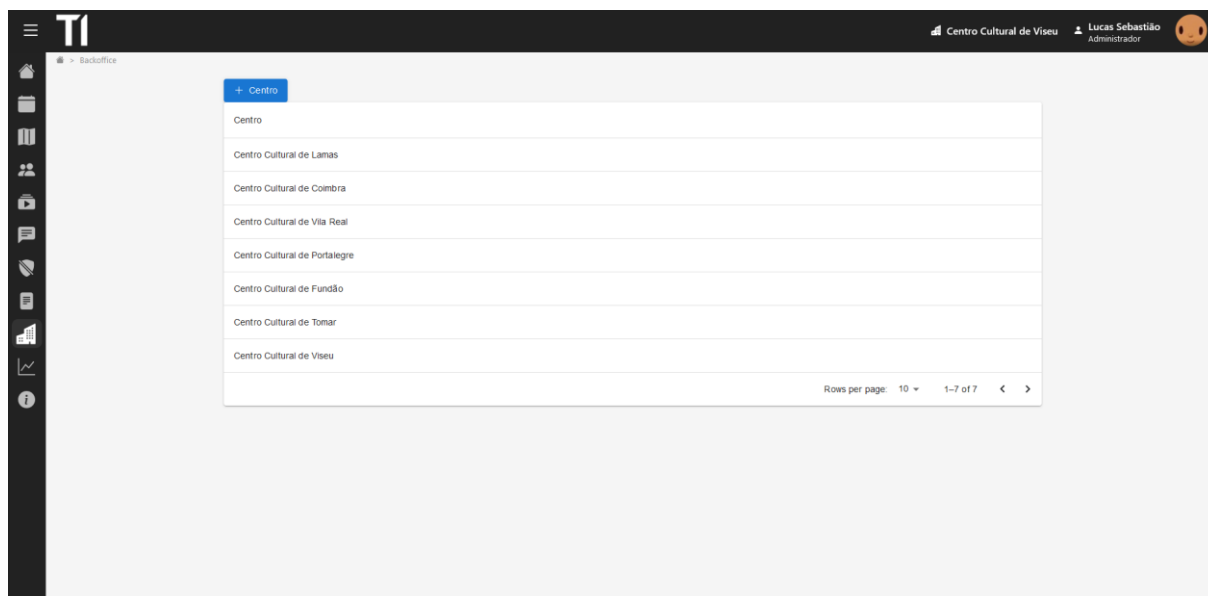


Figura 22 - Página: Centros (Backoffice)

3.3.6.10 Estatísticas/Reporting (Backoffice)

3.3.6.11 Sobre

Está página é o que foi pedido para ter as informações básicas do projeto, como credenciais, link para download da aplicação, relatório, link para a web app e uma pequena vídeo. Na Figura 23 está ilustrada a página sobre.

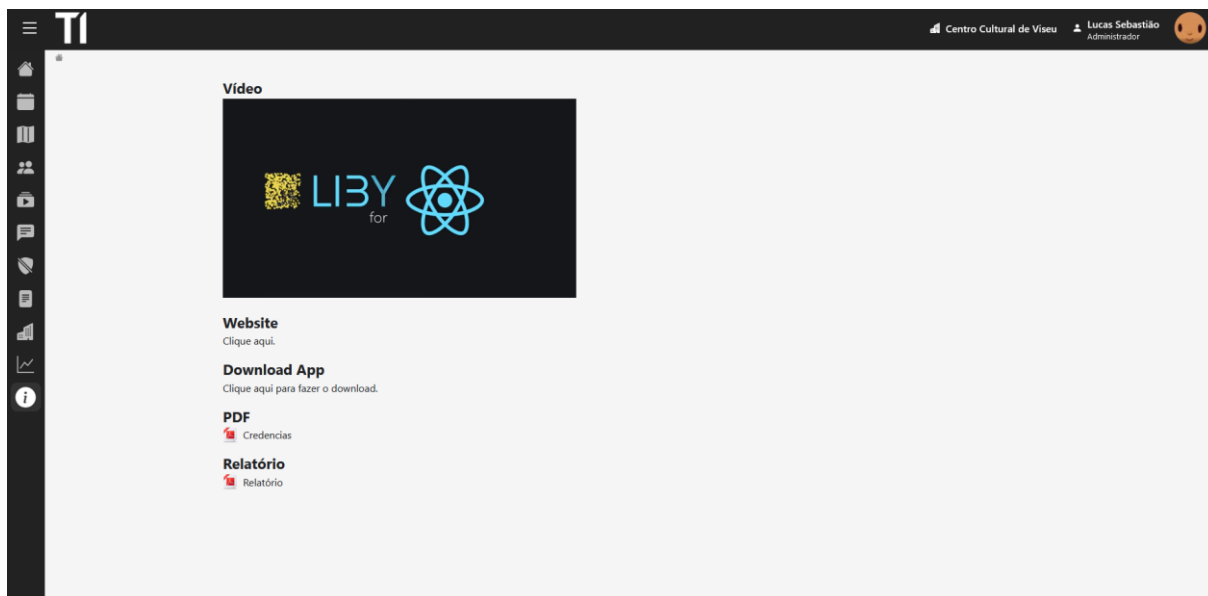
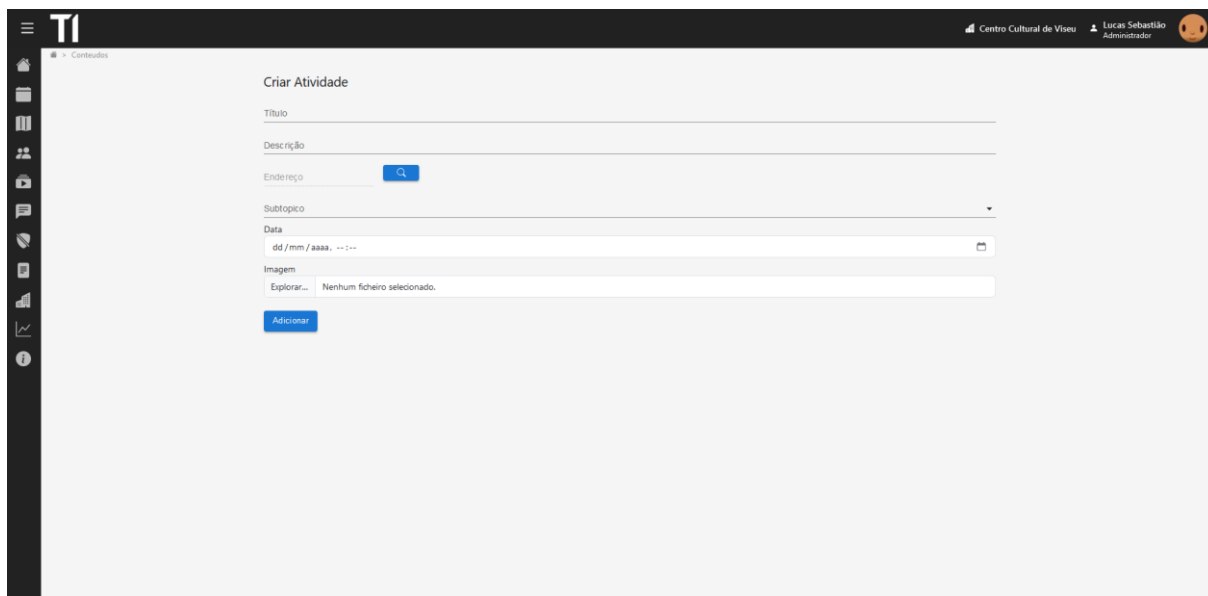


Figura 23 - Página: Sobre

3.3.6.12 Criar Conteúdos

Na criação do conteúdo, dependendo do tipo (atividade/recomendação/espço/evento), são visíveis apenas os campos necessários. Para escolher a localização foi utilizado a biblioteca *leafletmap*, que foi documentado e detalhado no capítulo 3.3.4.2.



The screenshot shows a web application interface for creating an activity. The header includes the logo 'TI' and the text 'Centro Cultural de Viteu' and 'Lucas Sebastião Administrador'. The main form is titled 'Criar Atividade' and contains the following fields: 'Título' (Title), 'Descrição' (Description), 'Endereço' (Address) with a search icon, 'Subtopico' (Subtopic) with a dropdown arrow, 'Data' (Date) with a date picker showing 'dd/mm/aaaa', and 'Imagem' (Image) with a file explorer button. A blue 'Adicionar' (Add) button is at the bottom of the form.

Figura 24 - Formulário para criar uma atividade

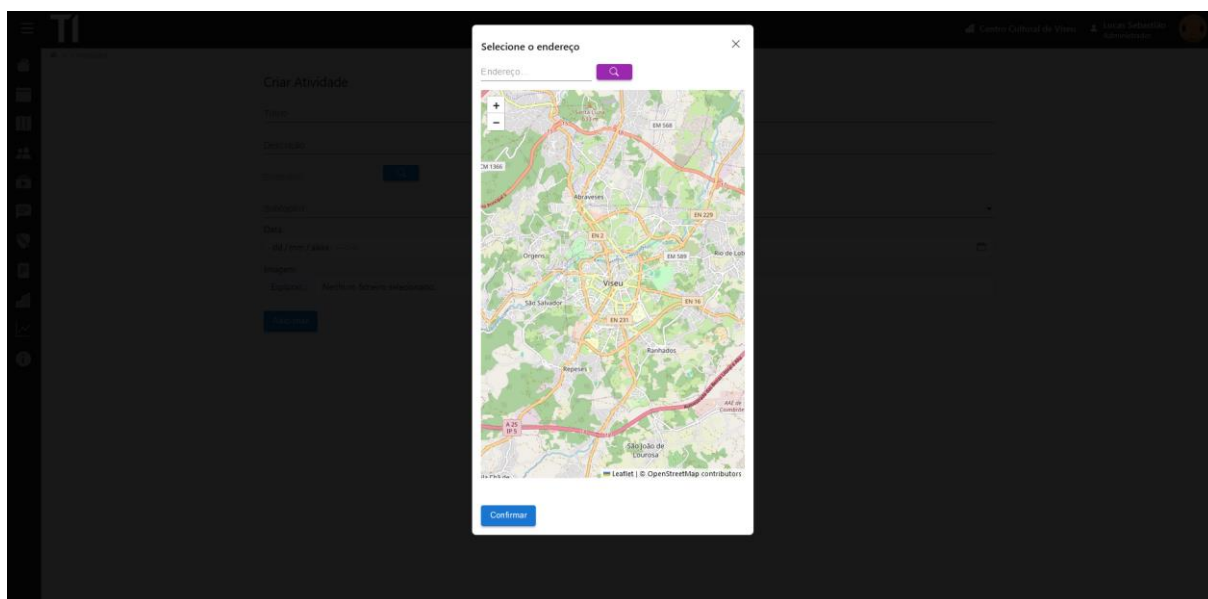


Figura 25 - *Popup* para escolher a localização da atividade

3.3.6.13 Detalhes dos conteúdos

A página de detalhes tem bastantes funcionalidades:

- **Participar:** caso o utilizador esteja interessado ele pode se inscrever no evento/atividade.
- **Editar:** enquanto o conteúdo está em revisão, tanto o utilizador que o criou quanto os administradores podem editar o conteúdo.
- **Rever:** a qualquer altura um administrador pode aprovar/rejeitar ou mover o conteúdo para análise.
- **Copiar link:** caso o utilizador queira partilhar o conteúdo com alguém fora da app pode copiar o link e enviar para quem quiser.
- **Classificar:** é possível classificar o conteúdo utilizando um sistema de 5 estrelas.
- **Comentar:** é possível fazer um comentário, seja a fazer perguntas, ou para dar a sua opinião.
- **Álbum:** caso o utilizador queira pode adicionar imagens ao álbum do conteúdo.

Na Figura 26 e Figura 27 estão ilustrados um evento e uma recomendação, respetivamente.

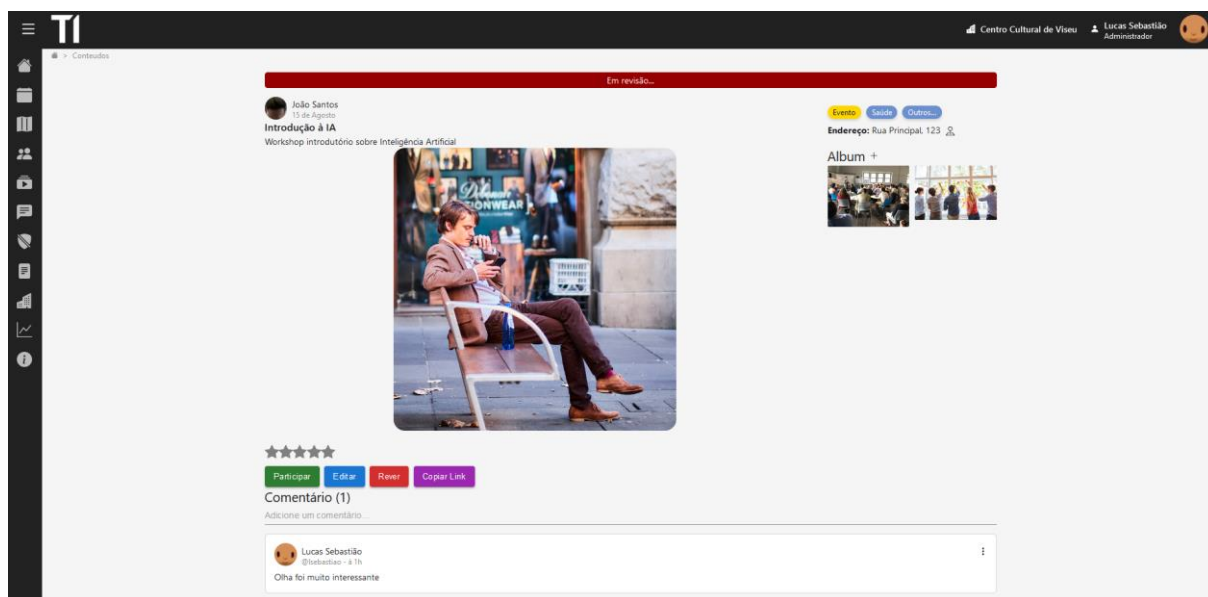


Figura 26 - Exemplo de um evento

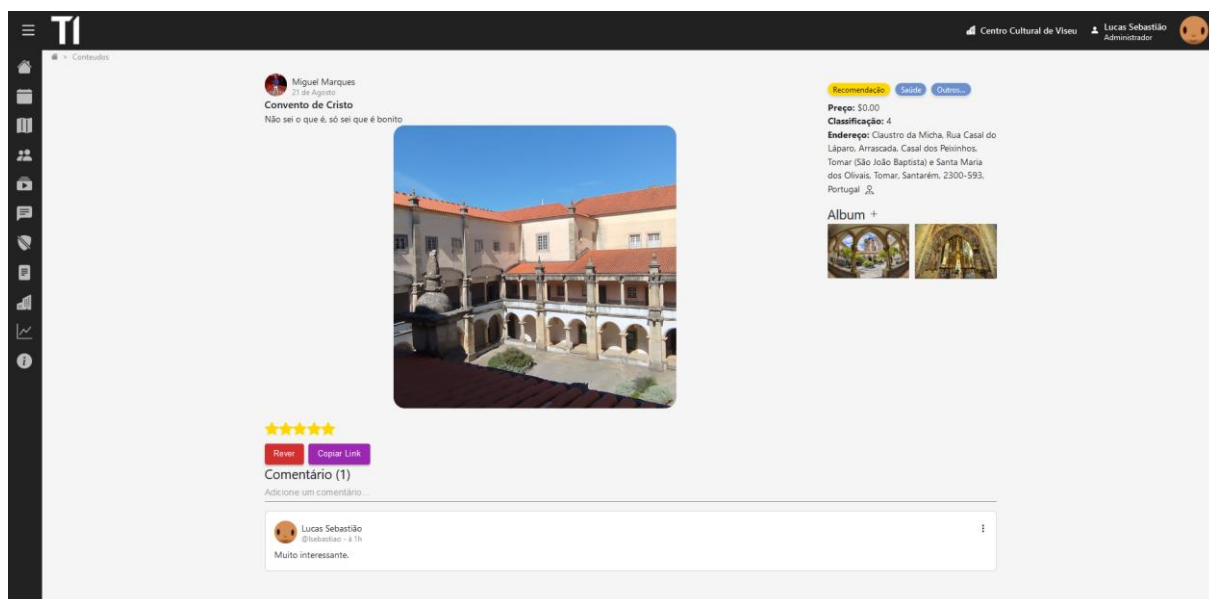


Figura 27 - Exemplo de uma recomendação

3.3.6.14 Perfil

No perfil do utilizador tem as várias informações sobre o mesmo, como é possível observar na Figura 28, as informações disponíveis são:

- Nome e sobrenome;
- *Tag*;
- Data de criação;
- Centro a que pertence;
- Interesses;
- Redes sociais;
- Perfil;
- Conteúdos criados;
- Inscrições

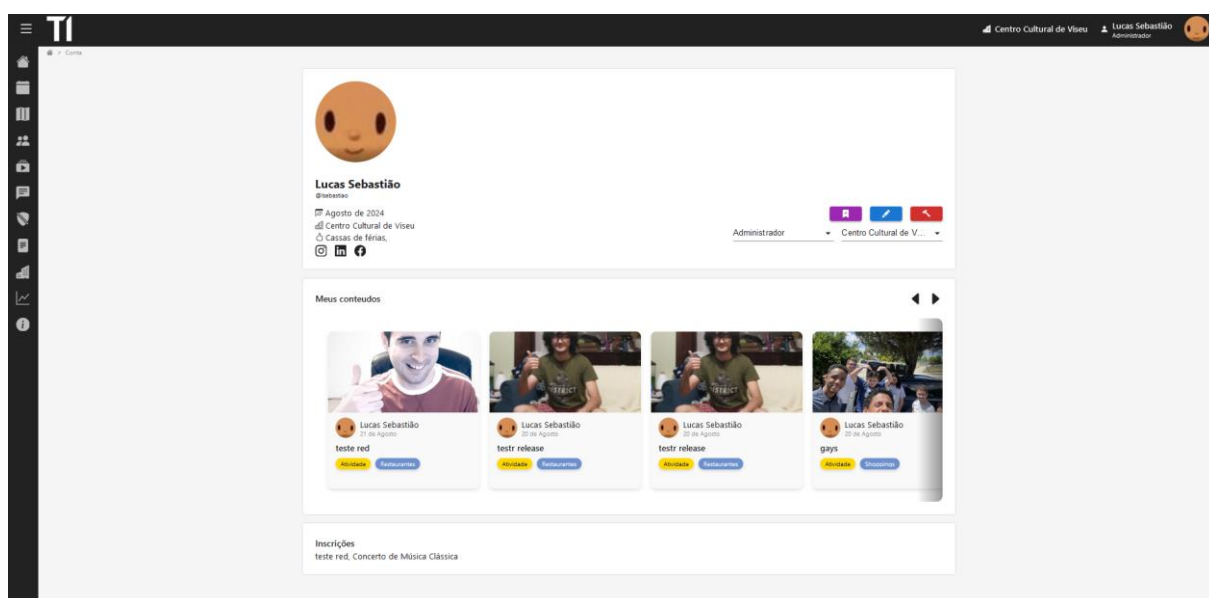


Figura 28 - Página: Perfil

O utilizador pode adicionar tópicos que está interessado, como é possível ver a interface na Figura 29. Ao adicionar um tópico como interesse, sempre que um conteúdo for adicionado com esse tópico, o utilizador será notificado.

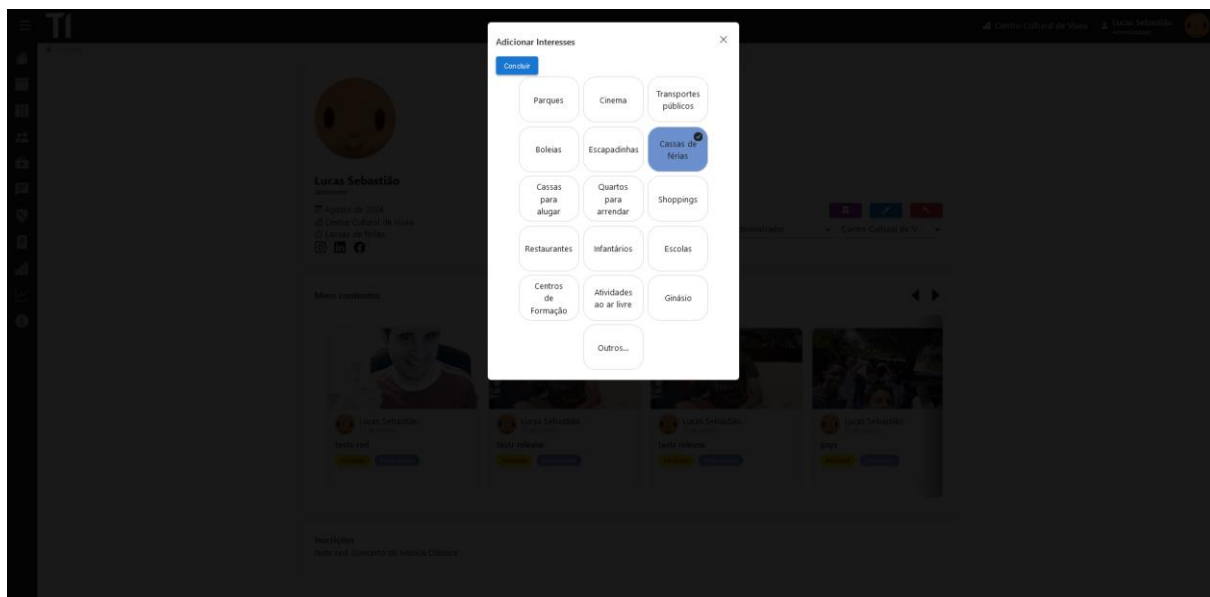


Figura 29 - Página: Perfil - Interesses

3.4. Mobile

3.4.1. Estrutura do Projeto

O projeto foi organizado em diferentes pastas para garantir uma estrutura organizada, que facilitasse o desenvolvimento contínuo pelos diferentes membros do grupo e a compreensão do código em melhorias futuras.

A estrutura está separada da seguinte forma:

- **assets** – este diretório foi usado para guardar imagens e logos.
- **backend** – esta pasta contém todos os ficheiros relacionados a funções de comunicação com a API.
- **widgets** – aqui encontram-se os ficheiros com o código dos widgets que foram sendo reutilizados em diferentes partes do projeto.
- **utils** – contém todas as funções uteis que são usadas em vários pontos da aplicação o que evita a repetição de código.
- **lib** – esta pasta está subdividida em vários diretórios. Cada um contém ficheiros correspondentes a funcionalidades, páginas ou conteúdos diferentes como por exemplo: atividade, espaço, recomendação, evento, calendário e perfil.

3.4.2. Detalhes dos conteúdos

A página dos detalhes dos conteúdos apresenta a seguinte estrutura base:



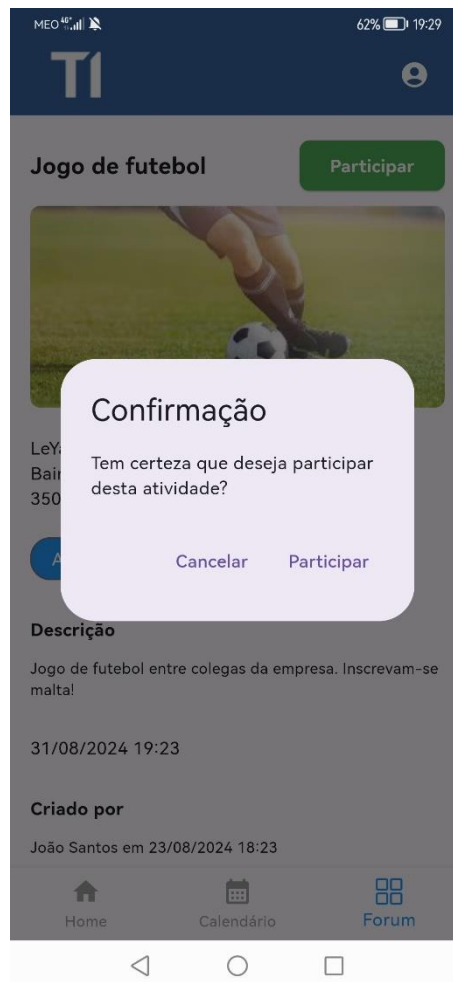
Nesta página são mostrados os detalhes de um conteúdo: título, imagem, local, tópico e subtópicos a que pertencem, descrição, data/hora, o utilizador que criou bem como a data e hora

de criação. Tem também um botão de “participar”, um álbum de imagens, uma opção de partilha, classificação e uma secção de comentários. Estas funcionalidades são explicadas a seguir.

É de notar que de acordo com o tipo de conteúdo esta página será ligeiramente diferente, sendo mostrados ou escondidos campos como data/hora, preço.

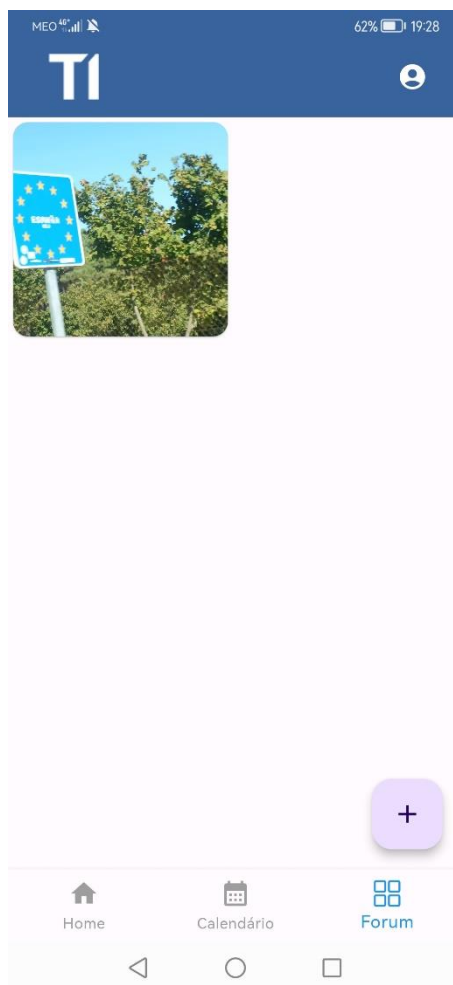
Funcionalidades/Requisitos

- **Participar na atividade** – Ao carregar no botão de “Participar” no topo da página o utilizador cria uma participação no conteúdo em questão. Este botão estará oculto para o caso de o conteúdo ser uma recomendação ou espaço.

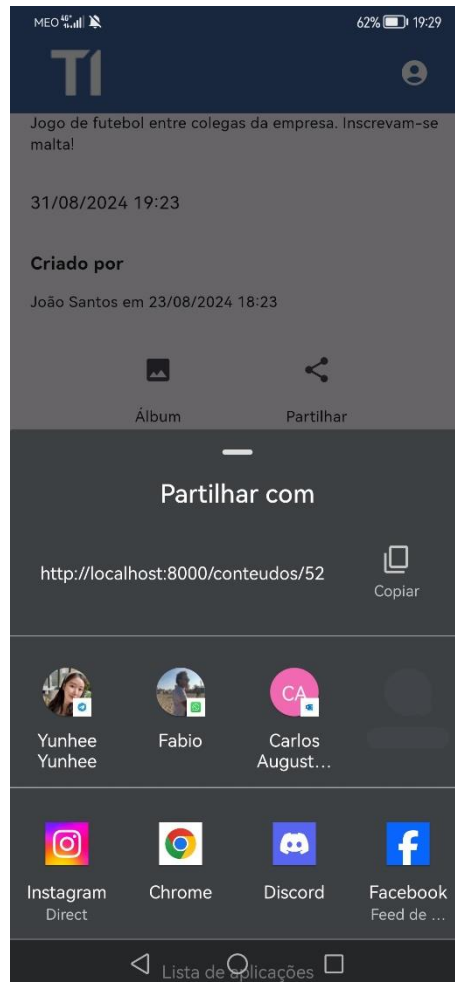




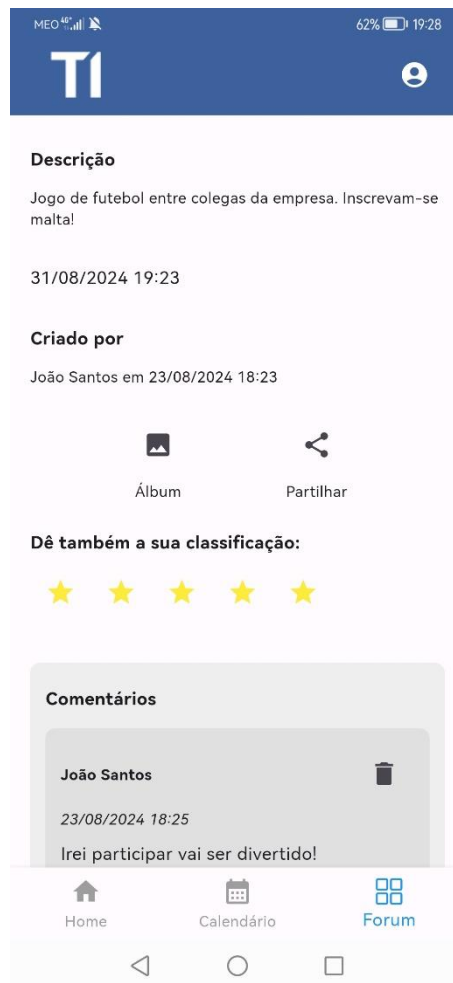
- **Álbum de imagens** – Se o utilizador clicar no botão do “Álbum” será redirecionado para outra página onde poderá ver e adicionar mais imagens ao álbum.



- **Partilhar** – Por sua vez o botão de “Partilhar” copia para a área de transferência o link do conteúdo da página web. Ao mesmo tempo aparecem as aplicações do telemóvel onde o utilizador poderá partilhar esse link.



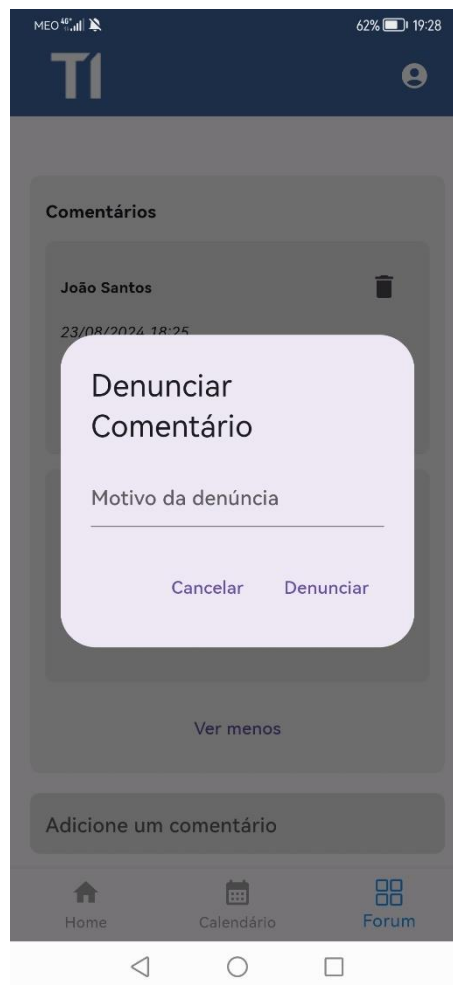
- **Classificação** – Tal como pretendido também é possível para um utilizador atribuir uma classificação de 1-5 (representado por estrelas) ao conteúdo.



- **Comentários** – Existe também uma secção de comentários onde os utilizadores podem interagir acerca do conteúdo publicado.

O utilizador consegue adicionar comentários ao conteúdo e eliminá-los caso estes não tenham nenhuma denuncia ou classificação. Pode ainda denunciar e classificar comentários de outros users.





3.4.3. Criar conteúdos (Daniel)

3.4.4. Perfil (Daniel)

3.4.5. Editar perfil (Daniel)

3.4.6. Calendário (Francisco)

3.4.7. Fórum (Francisco)

3.4.8. Listagem por tipo (Francisco)

3.4.9. Integração com o Google e Facebook (Cassamo/Francisco)

3.4.10. Iniciar sessão (Cassamo)

Falar da comunicação com a api (não é preciso explicar o que acontece na api) como é feita (aproveitando e mencionando o capítulo comunicação com a api) e mencionar também o esqueceu passe

3.4.11. Comunicação com a API (Cassamo)

Explicar para que serve e como funcionam as classes AuthService, ApiService e MyHttp

4. Conclusão

Através da implementação de um sistema de BackOffice, e uma aplicação mobile, foi possível criar uma solução prática e funcional com o intuito de facilitar a adaptação dos novos colaboradores a diferentes localidades, ao mesmo tempo em que promove uma maior interação e partilha de experiências entre os membros da organização.

Durante o desenvolvimento, foram utilizadas tecnologias modernas e amplamente adotadas no mercado, como *Node.js*, *Express.js*, *Sequelize*, *PostgreSQL*, entre outras, garantindo a robustez e escalabilidade da aplicação. Além disso, a integração de serviços como *JWT* para autenticação, *Cloudinary* para gestão de imagens, e a utilização de metodologias de desenvolvimento ágeis contribuíram para a qualidade e segurança da solução final.

Este projeto não só permitiu aplicar e consolidar os conhecimentos adquiridos ao longo do curso, mas também demonstrou a importância de uma abordagem integrada entre diferentes componentes de um sistema. Através do trabalho realizado, ficou evidente que a combinação entre um *backend* robusto, uma interface web intuitiva, e uma aplicação mobile eficiente pode resultar em uma solução completa e de grande valor para o usuário final.

Por fim, a aplicação desenvolvida cumpre os objetivos propostos, proporcionando uma ferramenta útil para a integração de colaboradores e, ao mesmo tempo, fomentando um ambiente mais colaborativo dentro da organização. O projeto serviu como uma experiência enriquecedora, preparando os alunos para futuros desafios no desenvolvimento de sistemas complexos e interativos.