Universidade Aberta

Universidade de Trás-os-Montes e Alto Douro

MEIW – Mestrado em Engenharia Informática e Tecnologia Web

22285 – Programação Web Avançada

Projeto Final – Gestão de Sócios da Banda Musical de Monção

Ano letivo 2023/2024

Docentes:

Professor Doutor Luís Barbosa

Professor Doutor Ricardo Batista

Discente

André Pereira [2302569]

Índice

Introdução: 3
Planeamento 4
Diagrama de casos de uso 4
Mockups 5
Modelo dos dados 9
Dependências e bibliotecas utilizadas 11
Operações CRUD e API REST 12
Views do projeto 17
- Home.vue 17
- Inscrição.vue 17
- Noticias.vue 17
- Acerca.vue 17
- Contactos.vue 17
- Login.vue 18
- Conta.vue 18
- Admin.vue 18
- Notificacao.vue 18

Router 19
- Index.js 19
- Router.js 19

Controllers 20
Publicação online 21
Links do projeto publicado online 21
Conclusão 21
Referências Bibliográficas 22

Introdução:

No contexto da Unidade Curricular de Programação Web Avançada, chega-se à fase final: a implementação de uma aplicação Web como prova de conceito. Esta etapa exige a aplicação prática de recursos full-stack, envolvendo tanto o front-end quanto o back-end e baseados na linguagem JavaScript. A minha proposta de projeto apresentada para esta Unidade Curricular é a criação de um sistema de gestão para as inscrições de sócios na instituição Banda Musical de Monção. Esta aplicação será full-stack, com um front-end construído em Vue.js, um back-end em Node.js com Express, e a base de dados armazenada no MongoDB. A autenticação será implementada para sócios e administradores, garantindo um acesso seguro às funcionalidades do sistema.

No front-end, os sócios têm um sistema de login, um formulário de inscrição, uma área exclusiva para administradores, a confirmação ou rejeição de inscrições, a visualização da lista de sócios inscritos e a remoção de sócios. Além disso, haverá um sistema de criação de notificações de eventos diversos, enviadas por email ou postadas na página de notícias e uma área personalizada para sócios gerirem e atualizarem seus dados. No back-end, serão implementados sistemas de autenticação, registo de inscrições de sócios, operações de edição, remoção e alteração de dados de sócios, registo de eventos e notificações para comunicação eficaz com os sócios, e um sistema de envio de emails para notificar sócios sobre eventos e atualizações.

O avanço da tecnologia e a crescente importância da presença online para organizações como a Banda Musical de Monção tornam este projeto não apenas uma prova de conceito técnico, mas também uma ferramenta para a organização e proximidade com os seus sócios. A aplicação será uma forma eficaz de gerir inscrições, comunicar eventos e manter os sócios informados, tudo de forma mais atual eficiente.

Planeamento

Vamos analisar em detalhe os 3 pilares do desenvolvimento usados: os diagramas de caso de uso, os mockups e o modelo dos dados

Diagrama de casos de uso

O diagrama de casos de uso descreve as principais interações entre os atores da aplicação (sócios e administradores) e os componentes do sistema.

Os sócios podem aceder ao sistema através do Login e, se não tiverem conta, podem registar-se na Inscrição. Após autenticados, têm acesso à sua página (Página do sócio), onde podem ver e editar as suas informações e aceder à gestão de notícias (Notícias). Recebem notícias em forma de notificações sobre eventos e atualizações.

Os administradores gerem todas as informações e funcionalidades, incluindo a gestão de sócios (Gestão de sócios), a criação e gestão de notícias (Notícias), e o envio de notificações (Envio de Notificações). Podem adicionar, editar e remover Sócios. Podem criar, editar e remover eventos (Gestão de Eventos), essenciais para a organização e gestão das atividades. Podem enviar notificações sobre eventos e atualizações para os Sócios, mantendo-os informados e envolvidos. Os Sócios podem visualizar informações detalhadas sobre os eventos (Detalhes do Evento), garantindo a participação e o sucesso dos mesmos.

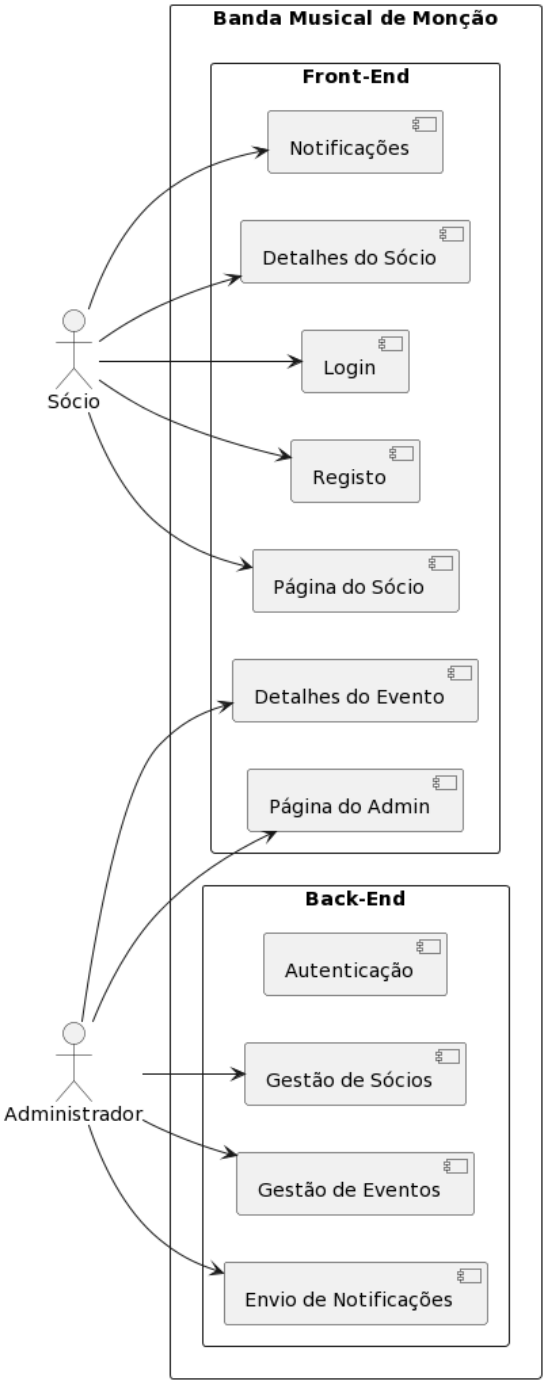

Figura 1 - Diagrama de Casos de Uso

Mockups

A transformação digital chegou recentemente à Banda Musical de Monção, tendo esta implementado a criação da marca Banda Musical de Monção, adotando um novo logotipo e uma imagem mais profissional. Aproveitando esta onda de inovação decidi aproveitar a nova palete de cores e fontes adotada por esta marca.

Figura 2 - Palete de cores escolhida

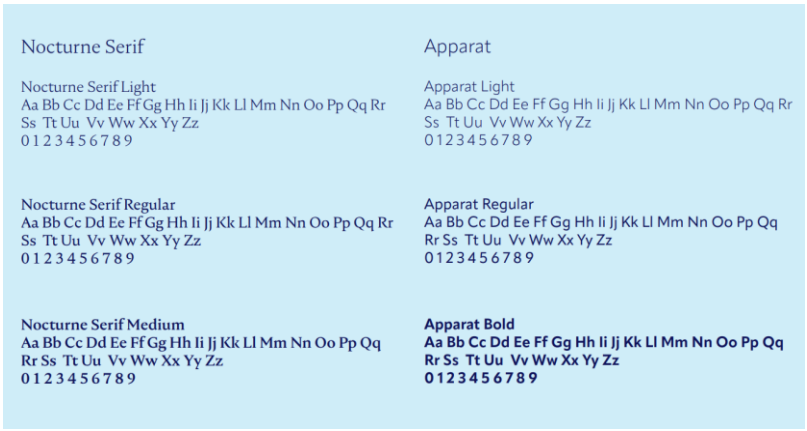

Figura 3 - Palete de fontes escolhida

Não foi possível nesta fase implementar as fontes pretendidas pelo que mesmo depois no desenvolvimento foi usada a fonte PT Serif pela sua similaridade. Idealizaram-se assim os seguintes mockups:

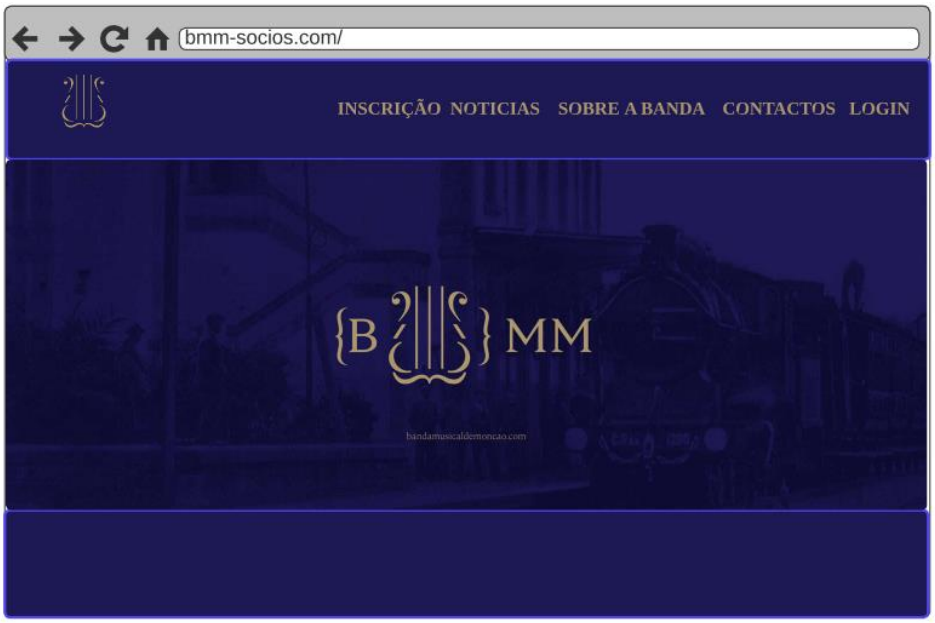

Figura 4 - Mockup Página Inicial

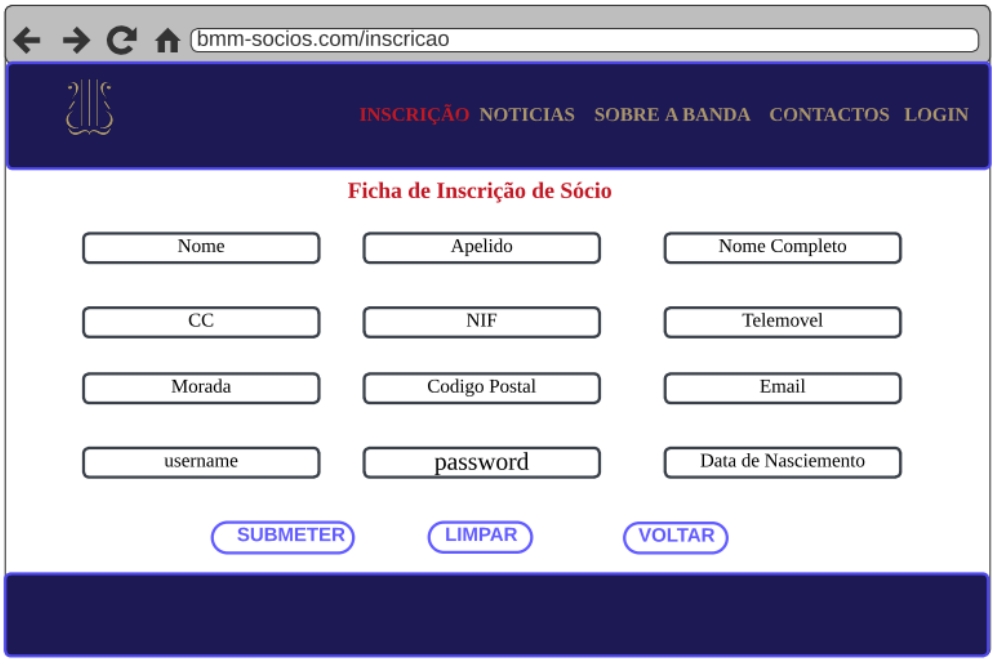

Figura 5 - Mockup da inscrição de sócio

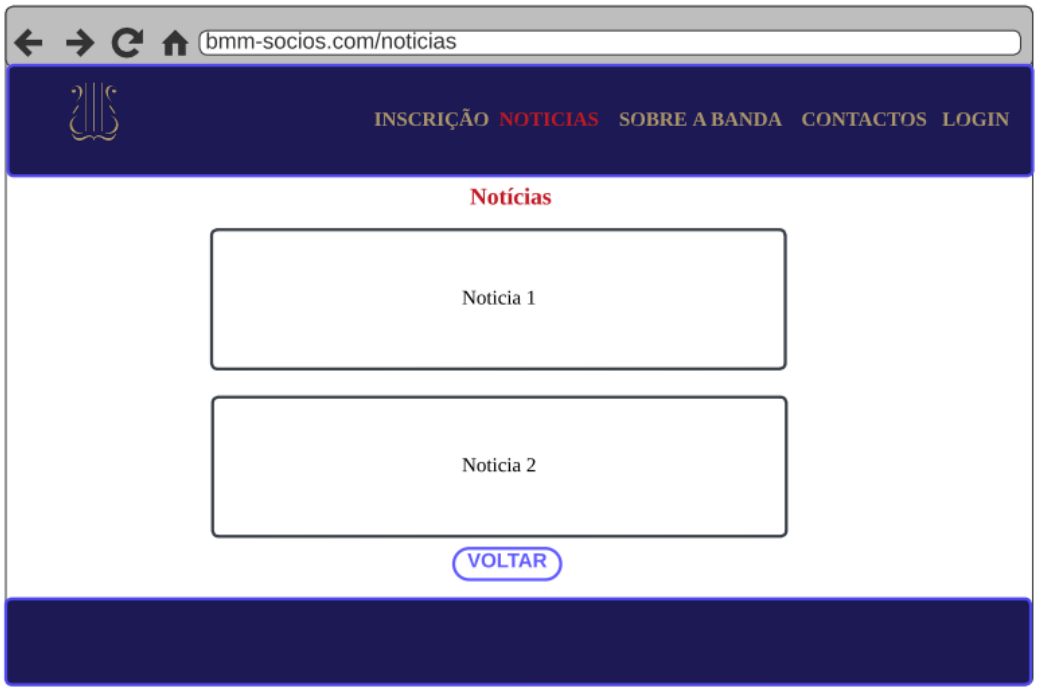

Figura 6 - Mockup das notícias

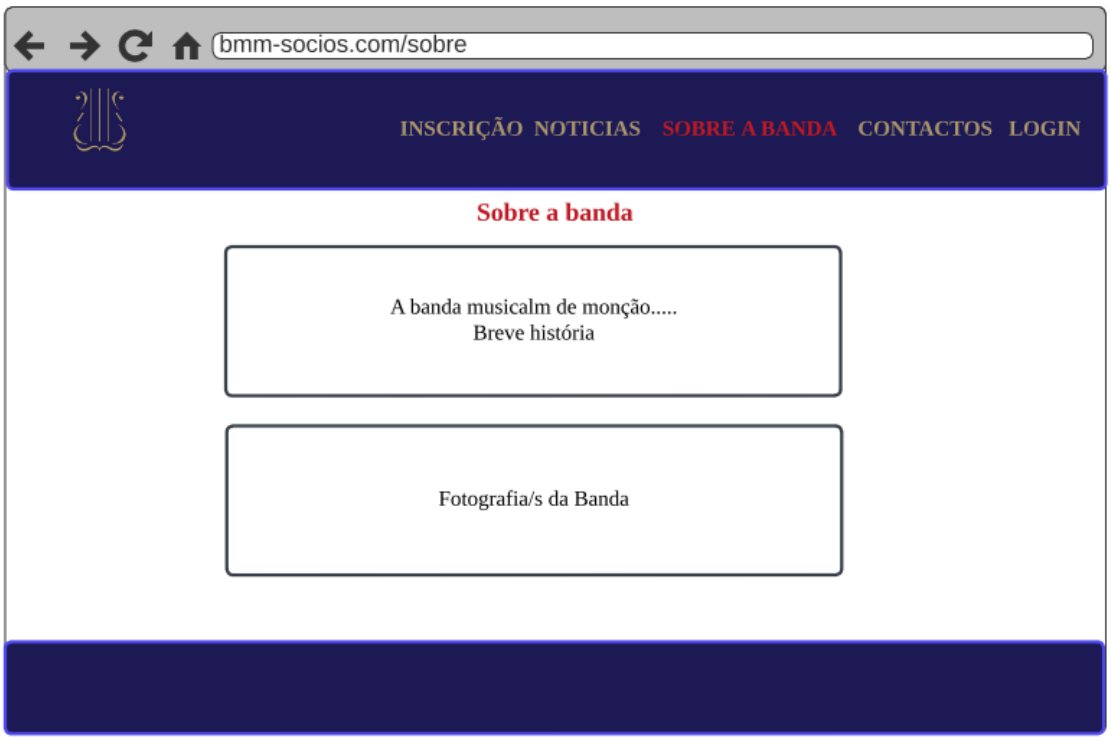

Figura 7 - Mockup acerca da banda

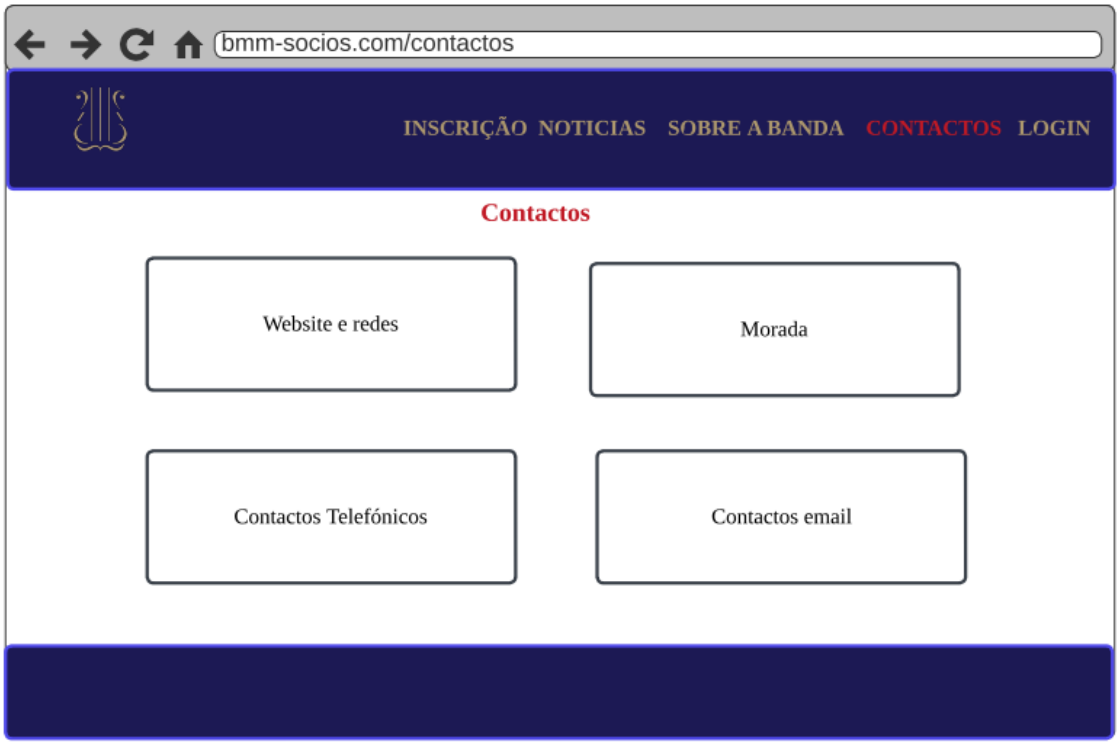

Figura 8 - Mockup dos contactos

Figura 9 - Mockup visão admin´

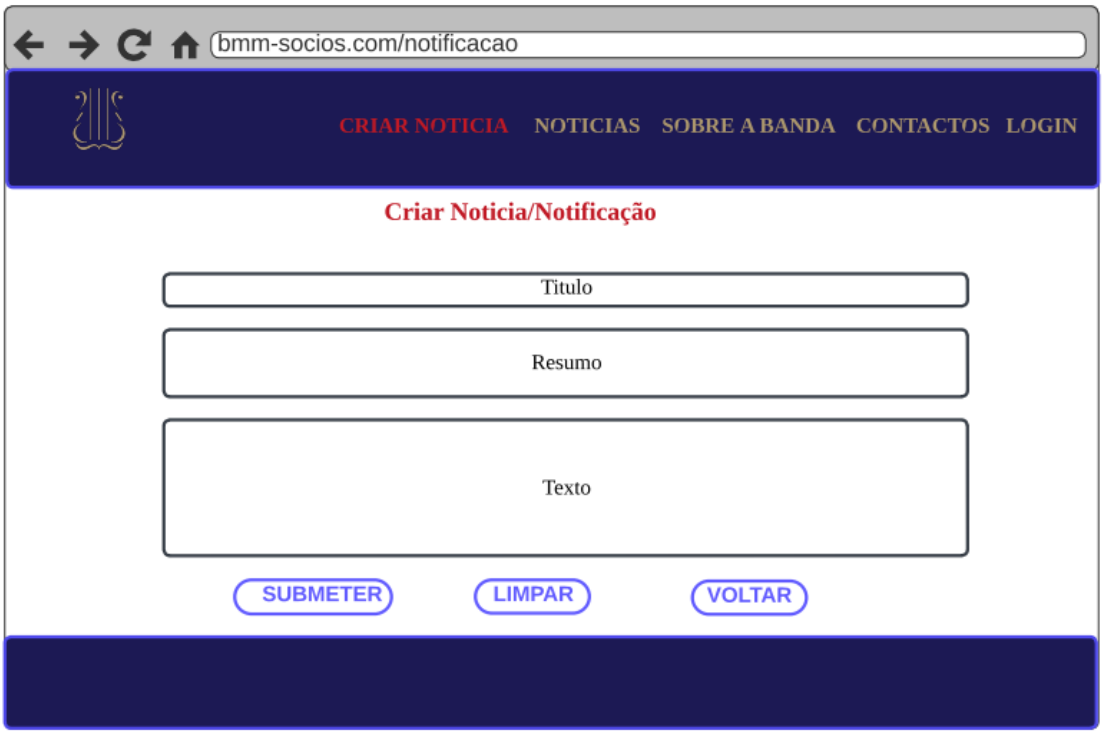

Figura 10 - Mockup criação de notícia (admin)

Figura 11 - Mockup da edição de conta (socio e admin)

Modelo dos dados

A Base de dados é composta por apenas 2 tabelas que representam as entidades principais do sistema (Utilizador e Notificação).

O modelo de dados para o utilizador contém informações pessoais dos utilizadores, como nome, morada, contacto, entre outros. Além disso, possui campos para indicar o nível de acesso do utilizador (sócio ou administrador) e a aceitação de notificações por email. Este modelo também regista a data de registo do utilizador e possui campos únicos para o nome de utilizador e palavra-passe.

O modelo de dados para notificações contém informações sobre as notificações enviadas aos utilizadores, como título, resumo e texto completo. Além disso, possui campos para indicar se a notificação deve ser enviada por email ou publicada na página de notícias. Este modelo também regista a data de registo da notificação.

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const notificationSchema = new Schema({
  title: String,
  summary: String,
  text: String,
  notifEmail: Boolean,
  notifPage: Boolean,
  registration_date: {
    type: Date,
    default: Date.now
  }
})

module.exports = mongoose.model('notification', notificationSchema);
```

Figura 12 - Model das notificações

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const userSchema = new Schema({
    firstname: String,
    lastname: String,
    name: String,
    cartaocidadao: String,
    nif: String,
    morada: String,
    codigopostal: String,
    freguesia: String,
    bdate: String,
    email: String,
    mobile: Number,
    accepted: Boolean,
    level: String,
    notifications: Boolean,
    username: {
        type: String,
        unique: true
    },
    password: String,
    registration_date: {
        type: Date,
        default: Date.now
    },

});

module.exports = mongoose.model('user', userSchema);
```

Figura 13 - Model do Utilizador

Por defeito e criando funções próprias, criamos um administrador e uma notificação por defeito que têm a seguinte estrutura e tabela já na BD:

```
_id: ObjectId('65ce17e4d964381ea3a6b9f2')
firstname : "Admin"
lastname : "User"
name : "Admin User"
nif : "12345678"
morada : "Rua do Admin"
codigopostal : "1234-567"
freguesia : "Dsae"
email : "admin@example.com"
accepted : true
level : "admin"
notifications : true
username : "admin"
password : "admin"
registration_date : 2024-02-10T13:55:48.360+00:00
```

```
_id: ObjectId('65ce17e4d964381ea3a6b9f0')
title : "Bem vindo ao espaço sócio!"
summary : "Um espaço de intregação e modernização da nossa banda."
text : "Um espaço que permetirá a gestão eficiente e comunicação com os sócios…"
notifEmail : false
notifPage : true
registration_date : 2024-02-15T13:55:48.341+00:00
```

Figuras 14 e 15 - Tabelas iniciais na Base de Dados

Dependências e bibliotecas utilizadas

As dependências e bibliotecas utilizadas no projeto podem ser consultadas na seguinte tabela:

	Bibliotecas	*Descrição*
BACK-END	bcrypt	Biblioteca usada para criptografia de passwords.
	cookie-parser	Biblioteca para manipulação de cookies (autenticação).
	cors	Biblioteca que permite que o frontend faça requisições ao backend (política de mesma origem (CORS))
	express	Framework web para Node.js, cria e gere rotas e controladores.
	express-validator	Biblioteca que valida dados no Express, garante a integridade dos dados recebidos pelo backend.
	handlebars	Mecanismo de template para Node.js, renderiza as páginas HTML do frontend.
	moongoose	Biblioteca de modelagem de objetos MongoDB, define e manipula os modelos de dados do sistema.
	nodemailer	Biblioteca de envio de emails, notifica os utilizadores sobre eventos e atualizações.
FRONT-END	axios	Biblioteca de pedidos HTTP, comunica com o backend.
	bootstrap-icons	Biblioteca que adiciona ícones às páginas do frontend.
	core-js	Biblioteca que fornece funcionalidades atuais do JavaScript e garante a compatibilidade com diferentes navegadores (mesmo os mais antigos).
	express	Framework web para Node.js, cria e gere rotas e também controladores.
	path	Biblioteca que manipula caminhos de ficheiros e diretórios no frontend.
	serve-static	Biblioteca que serve ficheiros estáticos como os ficheiros HTML, CSS e JavaScript do frontend.
	vue	Framework de construção de interfaces de utilizador, cria as páginas do frontend.
	vue-router	Biblioteca para roteamento de aplicações Vue.js, define as rotas do frontend.
	vuex	Biblioteca para gestão do estado global do sistema em aplicações Vue.js.

Tabela 1 - Dependências e bibliotecas do projeto

Operações CRUD e API REST

No backend, as operações CRUD foram implementadas utilizando a framework Express e o ORM Mongoose para interagir com a base de dados MongoDB. Vamos utilizar por exemplo as operações no utilizador. Para criar um utilizador, é necessário validar os dados recebidos do pedido de inscrição e verificar se já existe um utilizador com o mesmo email. Se não existir, é criado um utilizador com os dados fornecidos, que são armazenados na base de dados.

```
exports.create = (req, res) => {
    const errors = validationResult(req).array();
    if (errors.length > 0) return res.status(406).send(errors);

    User.findOne({ 'email': req.body.email })
        .exec()
        .then((user) => {
            if (user) res.status(userMessages.success.s3.http).send(userMessages.success.s3);
            else {
                const newuser = new User({
                    firstname: req.body.firstname,
                    lastname: req.body.lastname,
                    name: req.body.name,
                    cartaocidadao: req.body.cartaocidadao,
                    nif: req.body.nif,
                    email: req.body.email,
                    bdate: req.body.bdate,
                    mobile: req.body.mobile,
                    accepted: req.body.accepted,
                    notifications: req.body.notifications,
                    morada: req.body.morada,
                    codigopostal: req.body.codigopostal,
                    freguesia: req.body.freguesia,
                    level: req.body.level,
                    username: req.body.username,
                    password: req.body.password

                });
                newuser.save()
```

Figura 16 - código de criação do utilizador

Para ler todos os utilizadores, é feita uma consulta à base de dados que retorna todos os utilizadores existentes.

```
exports.get = ((req, res) => {
    User.find()
        .exec()
        .then((user, error) => {
            if (error) throw error;
            let message = userMessages.success.s1;
            message.body = user;
            return res.status(message.http).send(message);
        })
        .catch((error) => {
            console.log(error);
            return res.status(500).send({ error: 'Internal Server Error' });
        });
});
```

Figura 17 - Código da leitura dos utilizadores

Para atualizar um utilizador, é feita uma consulta à base de dados para encontrar o utilizador que se pretende atualizar e, em seguida, são atualizados os campos que escolhi com os novos valores fornecidos no pedido.

```
exports.update = (req, res) => {
    User.findOneAndUpdate({ '_id': { $eq: req.params.id } }, {
            $set: {
                'firstname': req.body.firstname,
                'lastname': req.body.lastname,
                'name': req.body.name,
                'cartaocidadao': req.body.cartaocidadao,
                'nif': req.body.nif,
                'bdate': req.body.bdate,
                'mobile': req.body.mobile,
                'morada': req.body.morada,
                'codigopostal': req.body.codigopostal,
                'freguesia': req.body.freguesia,
            }
        }, { new: true })
        .exec()
        .then((user) => {
            if (!user) return res.status(userMessages.error.e0.http).send(userMessages.error.e0);
            let message = userMessages.success.s1;
            message.body = user;
            return res.status(message.http).send(message);
        })
        .catch((error) => {
            console.log(error);
            return res.status(500).send({ error: 'Internal Server Error' });
        });
};
```

Figura 18 - Código do update do utilizador

Para apagar um utilizador, é feita uma consulta à base de dados para encontrar o utilizador que se pretende apagar e, em seguida, o utilizador é removido da base de dados.

```
exports.delete = (req, res) => {
    User.deleteOne({ '_id': { $eq: req.params.id } })
        .exec()
        .then((user) => {
            if (user.deletedCount <= 0) return res.status(userMessages.error.e0.http).send(userMessages.error.e0);
            let message = userMessages.success.s4;
            return res.status(message.http).send(message);
        })
        .catch((error) => {
            console.log(error);
            return res.status(500).send({ error: 'Internal Server Error' });
        });
};
```

Figura 19 - Código do Delete do utilizador

Todas estas operações são realizadas de forma assíncrona, utilizando promises, e são tratadas as exceções para garantir a integridade do sistema. Além disso, são devolvidas mensagens de erro ou sucesso, dependendo do resultado da operação.

No frontend, as operações CRUD são realizadas utilizando a biblioteca Axios para fazer requisições HTTP ao backend. Para criar um utilizador, é feita um pedido POST ao endpoint correspondente, com os dados do utilizador a serem criados.

```
await axios
  .post("http://localhost:3000/user", postData)
  .then((response) => {
    if (response.data.http == 201) {
      this.message.type = "success";
      this.message.msg = "Utilizador criado com sucesso. A sua inscrição fica pendente de aprovação.";
      this.isShow = true
    } else if (response.data.http == 200) {
      this.message.type = "warning";
      this.message.msg = "Utilizador existente.";
      this.isShow = true
    } else {
      this.message.type = "danger";
      this.message.msg = "Ocorreu um problema, tente de novo...";
      this.isShow = true
    }
  })
```

Figura 20 - Pedido POST no frontend

Para ler todos os utilizadores, é feita um pedido GET ao endpoint correspondente, que retorna todos os utilizadores existentes.

```
async getUsers() {
  this.usersAccepted = [];
  this.usersToAccept = [];
  await axios
    .get("http://localhost:3000/user", {
      headers: {
        Authorization: this.token,
      },
    })
    .then((response) => {
      let users = response.data.body;
      for (let i = 0; i < users.length; i++) {
        if (users[i].accepted == true && users[i].level == "socio") {
          this.usersAccepted.push({
```

Figura 21 - pedido GET no frontend

Para atualizar um utilizador, normalmente é feito um pedido PUT ao endpoint correspondente, mas como apenas queremos atualizar alguns campos como morada ou número de telefone usamos o método PATCH, com os dados do utilizador a serem atualizados. Utilizamos o método PUT quando o administrador aceita uma inscrição e passa de não aceite para aceite.

```
await axios
  .patch("http://localhost:3000/user/" + this._id, postData, {
    headers: {
      Authorization: this.token,
    },
  })
  .then((response) => {
    if (response.data.http == 200) {
      this.message.type = "success";
      this.message.msg = "Alterações registadas com sucesso."
      this.isShow = true
    } else {
      this.message.type = "danger";
      this.message.msg = "Ocorreu um problema, tente de novo...";
    }
  })
```

Figura 22 - Código do pedido PATCH no frontend

```
async acceptStd(_id) {
  await axios
    .put("http://localhost:3000/user/" + _id, {
      headers: {
        Authorization: this.token,
      },
    })
    .then(() => {
      this.message.msg = "Utilizador aceite!";
      this.message.type = "success";
      this.isShow=true
      this.getUsers();
    })
```

Figura 23 - Código do pedido PUT no frontend

Para apagar um utilizador, é feita um pedido DELETE ao endpoint correspondente, com o id do utilizador a ser apagado.

```
async deleteStd(_id) {
  await axios
    .delete("http://localhost:3000/user/" + _id, {
      headers: {
        Authorization: this.token,
      },
    })
    .then(() => {
      this.message.msg = "Utilizador eliminado!";
      this.message.type = "success";
      this.isShow=true
      this.getUsers();
    })
```

Figura 15 - Código do pedido DELETE no frontend

Todas estas operações são realizadas de forma assíncrona e são tratadas as exceções para garantir a integridade do sistema. Além disso, são devolvidas mensagens de erro ou sucesso, dependendo do resultado da operação.

Views do projeto

Home.vue

A página Home.vue é a página inicial da aplicação, onde os visitantes são recebidos com uma imagem de fundo da banda a barra de navegação com as hipóteses de exploração, um footer com as redes sociais e uma pequena mensagem. A imagem de fundo é uma representação da nova identidade assumida recentemente pela banda.

Inscrição.vue

A página de Inscrição é onde os visitantes podem-se inscrever como sócios da instituição. A página é composta por um formulário com campos para preencher, como nome, apelido, NIF, número do cartão de cidadão, email, número de telemóvel, data de nascimento, morada, código postal, freguesia, nome de utilizador e palavra-passe. Há também uma opção para subscrever notificações. Os visitantes podem preencher os campos e submeter o formulário, que irá criar um registo de utilizador no sistema. Se houver algum erro ou campo em falta, será exibida uma mensagem de erro. Os visitantes também têm a opção de limpar o formulário ou voltar à página inicial.

Noticias.vue

A página de Notícias é onde os utilizadores podem ler as últimas notícias e atualizações da instituição. A página é composta por uma lista de cartões, cada um contendo o título, um resumo e o texto completo da notícia. Os utilizadores podem percorrer a lista e ler as notícias.

Acerca.vue

A página Acerca fornece uma visão geral da história e realização da Banda Musical de Monção. A página inclui um texto detalhado que destaca os eventos e marcos importantes na história da banda, bem como a sua evolução ao longo dos anos. A página também inclui uma imagem da banda

Contactos.vue

A página Contactos fornece informações detalhadas sobre como entrar em contato com a Banda Musical de Monção. A página inclui a morada da banda, números de telefone para diferentes departamentos e membros da banda, um link para o endereço web da banda e endereços de e-mail para diferentes departamentos. Estas informações são apresentadas de forma clara e organizada, facilitando os utilizadores a encontrarem as informações de contacto necessárias.

Login.vue

A página Login é a que permite aos utilizadores autenticarem-se na aplicação. A página inclui um formulário de login com campos para o nome de utilizador e a senha, bem como um botão de submissão. Quando um utilizador tenta fazer login, o formulário é validado e, se os dados fornecidos estiverem corretos, o utilizador é redirecionado para a página inicial ou para a página de administração, dependendo do nível de acesso do utilizador. Se os dados fornecidos estiverem incorretos, uma mensagem de erro é exibida. A página é estilizada de forma simples e limpa, com um fundo branco e bordas azuis.

Conta.vue

A página Conta permite ao utilizador visualizar e atualizar os detalhes da sua conta. A página inclui um formulário com campos para o primeiro nome, último nome, nome completo, cartão de cidadão, NIF, e-mail, telemóvel, data de nascimento, morada, código postal, freguesia, e a opção para subscrever notificações. O utilizador pode preencher os campos com as informações atualizadas e clicar no botão "ALTERAR" para submeter as alterações. Após submeter as alterações, uma mensagem de sucesso ou erro é exibida, dependendo do resultado da operação.

Admin.vue

Esta página é a de gestão de sócios, onde o administrador pode visualizar os sócios inscritos e os pedidos pendentes de novos sócios. A página inclui duas tabelas: uma para os sócios inscritos e outra para os pedidos pendentes. Cada tabela mostra o nome, NIF e cartão de cidadão dos sócios, bem como botões para aceitar ou rejeitar os pedidos pendentes e para ver os detalhes dos sócios inscritos. Quando o administrador clica no botão "Detalhes", é aberta uma modal que mostra mais informações sobre o sócio, incluindo o nome completo, email, contacto móvel, data de nascimento, morada, código postal, freguesia e se está ou não inscrito para receber notificações.

Notificacao.vue

A componente Notificacao.vue é uma página onde os administradores podem criar uma notificação ou notícia. A página inclui um formulário com campos para inserir o título, resumo e texto da notificação, bem como opções para enviar a notificação por email e/ou publicar na página de notícias.

Router

Index.js

O ficheiro index.js é responsável por configurar e definir as rotas da aplicação Vue.js. Ele importa os componentes de visualização (views) e os componentes de armazenamento (store), e define as rotas da aplicação. A função createRouter do Vue Router é usada para criar um objeto router, que recebe um histórico de navegação baseado em createWebHistory. Este histórico de navegação é criado com base na variável de ambiente process.env.BASE_URL.

As rotas da aplicação são definidas no array routes, onde cada rota é um objeto com as seguintes propriedades:

Propriedades	Descrição
path	O caminho da URL que a rota deve corresponder.
name	O nome da rota.
component	O componente de visualização que deve ser processado quando a rota é acedida
meta	Um objeto que contém metadados sobre a rota. Neste caso, a propriedade auth é usada para indicar se a rota requer autenticação (false ou true)

Tabela 2 - Propriedades e descrição das rotas da aplicação

Se a rota requer autenticação e o usuário não está autenticado, ele é redirecionado para a página de login (next('/login')). Caso contrário, a navegação continua normalmente (next()).

Finalmente, o objeto router é exportado para ser usado na aplicação Vue.js.

Router.js

No backend, as rotas são como caminhos que o servidor segue para lidar com diferentes tipos de pedidos. Cada rota é como um endereço específico que o servidor conhece.

router.js: Este ficheiro diz ao servidor onde encontrar as rotas para cada tipo de recurso. Liga às rotas de autenticação, notificação e utilizadores

auth.routes.js: trata de tudo relacionado à autenticação de utilizadores.

notification.routes.js: trata de tudo relacionado à criação e visualização de notificações.

user.routes.js: Este arquivo trata de tudo relacionado à criação, visualização, atualização e exclusão de utilizadores (operações CRUD).

Controllers

Os controllers servem para gerir cada rota no backend. Eles recebem os pedidos, processam os dados e enviam uma resposta de volta. Existem 4 controllers neste projeto, para a autenticação, os emails, as notificações e os utilizadores.

auth.controller.js: Este ficheiro lida com tudo relacionado à autenticação de utilizadores (ex: verificar as credenciais e responder se elas são ou não válidas).

user.controller.js: Este ficheiro lida com tudo relacionado aos utilizadores. Por exemplo, se alguém tentar criar um utilizador, o controller sabe como guardar essas informações no banco de dados.

notification.controller.js: Este ficheiro lida com assuntos relacionados às notificações. Por exemplo, se alguém tentar criar uma notificação, o controller sabe como guardar essas informações no banco de dados e enviar e-mails para os utilizadores inscritos.

email.controller.js: Este ficheiro lida com o envio de e-mails. Por exemplo, se alguém tentar criar uma notificação e marcar para enviar por e-mail, o controller sabe como enviar esses e-mails para os utilizadores inscritos.

Publicação online

Para a publicação o projeto numa plataforma online também foi um desafio, pois encontrar alternativas gratuitas é difícil. A plataforma escolhida para este efeito foi o Render, pois dá para fazer deploy diretamente de um repositório existente. Sendo assim criaram-se 2 repositórios, uma para o backend (definimos como "webservice") e outro para o frontend (deginimos como "static page" publicação online da aplicação, dividida entre o segmento backend, designado como "webservice", e o frontend, classificado como "static site". Também troquei a base de dados local que estava a usar pelo MongoDB Compass para o serviço Atlas Cloud.

Links do projeto publicado online

Os links da publicação online do projeto são os seguintes:
Backend: https://gestao-socios-bmm-api.onrender.com
Frontend: https://gestao-socios-bmm-front.onrender.com

credenciais	*username*	*password*	*descrição*
administrador	admin	admin	Admin único
sócio	andre	andre	Sócio exemplo

Nota: Usar credenciais para visualizar Views correspondentes.

Conclusão

A realização do projeto de gestão de sócios da BMM foi desafiante. A utilização de tecnologias modernas como o Vue.js, Express.js e MongoDB permitiu a criação de uma aplicação web dinâmica e responsiva, que atende às necessidades de gestão de sócios de forma eficiente. O projeto implementou um CRUD completo para a gestão de sócios, permitindo a criação, leitura, atualização e eliminação de registos de sócios de forma eficiente e intuitiva.

A integração com a plataforma Render para a publicação online simplificou o processo de deploy, garantindo que as atualizações feitas no repositório sejam refletidas automaticamente na aplicação online. Além disso, a migração da base de dados local para o MongoDB Atlas Cloud proporcionou uma infraestrutura mais robusta e escalável.A adoção de boas práticas de segurança e monitorização contínua do desempenho da aplicação são fundamentais para garantir um ambiente de produção seguro e uma experiência do utilizador otimizada.

Conclui-se que o projeto de gestão de sócios da BMM é um exemplo de como a tecnologia pode ser aplicada de forma eficaz para resolver problemas do mundo real, proporcionando uma gestão mais eficiente e uma melhor experiência para os utilizadores.

Referências Bibliográficas

- Filipe Portela, Ricardo Queirós (2020). Desenvolvimento Avançado para a Web - Do front-end ao back-end. 1ª edição, FCA, ISBN: 978-972-722-915-4. (Wook/ Portugal/ FCA)
- Derek M. Powazek, (2002). Design for Community: The Art of Connecting Real People in Virtual Places. New Riders, ISBN: 9780735710757. (Amazon/EUA) (Amazon/Italia)
- PREECE, J., ROGERS, Y., SHARP, H. (2005). "Design de Interação". Bookman Companhia Edição 1.
- VIANNA, Maurício...[et al]. Design Thinking - Inovação em Negócios. Rio de Janeiro: MJC Press, 2012. 162p.).
- Peters, C. (2017). Building rich internet applications with node. js and express. js. Rich Internet Applications w/HTML and Javascript, 15.
- Barsoti, N., & Gibertoni, D. (2020). IMPACTO QUE O SEQUELIZE TRAZ PARA O DESENVOLVIMENTO DE UMA API CONSTRUÍDA EM NODE. JS COM EXPRESS. JS. Revista Interface Tecnológica, 17(2), 231-243
- Nguyen, S. (2023, December 5). Understanding the connection between databases and APIs. Dreamfactory software. https://blog.dreamfactory.com/understanding-the-connectionbetween-databases-and-apis/
- DIVE INTO HTML5 BY MARK PILGRIM: http://diveinto.html5doctor.com/
- HTML5 – A vocabulary and associated APIs for HTML and XHTML by Hickson, I., Berjon, R., Faulkner, et al. (orgs.) (2014): https://www.w3.org/TR/html5/
- MDN Javascript Reference: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference
- MDN Document Object Model (DOM) Reference: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model