
PAW – Trabalho Grupo

<MachadoTickets>

Versão 1.0

Grupo13

Índice

Conteúdo

1. Introdução – 1ª Milestone.....	1
1.1 Propósitos/Objetivos.....	1
1.2 Rotas	1
1.3 Modules	2
1.4 Views.....	2
1.5 Controllers	3
1.6 Planeamento	3
2. Descrição	5
2.1 Perspetiva do Produto.....	5
2.2 Funcionalidades do Produto	6
2.3 CSS	6
3. Organização de dados.....	6
3.1 MongoDB.....	6
4. Introdução – 2ª Milestone.....	7
4.1 Propósitos/Objetivos.....	7
4.2 API's usadas	7
4.3 Models	8
4.4 Components	8
4.5 Referências	13

Revision History

Name	Date	Reason For Changes	Version
Início do relatório	30/04/2023	-----	1.0
Conclusão do relatório	02/05/2023	Últimas alterações	2.0

1. Introdução – 1ª Milestone

1.1 Propósitos/Objetivos

Este projeto tem como principal objetivo desenvolver uma aplicação web que permita promover e gerir o património português. Para isso, a aplicação irá gerir eventos, locais e a compra de bilhetes associados aos mesmos, permitindo que o administrador adicione novos locais e eventos e que altere os preços dos bilhetes.

Neste primeiro milestone o nosso foco foi o desenvolvimento do backoffice da aplicação, sendo que implementamos as seguintes funcionalidades:

- Registo de funcionários com o papel de administradores;
- Gestão de eventos, permitindo adicionar e alterar informação e estabelecer preços de bilhetes;
- Registo de utilizadores e emissão de bilhetes para a conta do utilizador;

Esta aplicação será depois integrada com o frontoffice da aplicação, que será o foco no segundo milestone.

1.2 Rotas

Uma rota é basicamente um caminho que pode ser seguido dentro de uma aplicação web para aceder a diferentes recursos ou conteúdos.

O Express.js é uma framework node.js que permite a criação de aplicações web de forma simples e rápida, com suporte para rotas. Para criar uma rota no Express.js, é necessário definir uma url para a rota e um callback que será executado quando a rota for acedida.

Dividimos este projeto em 5 rotas principais: “index.js”, “adm.js”, “events.js”, “local.js”, “store.js”, sendo que utilizamos operações crud como “list”, “show”, “create”, “update”, “edit” e “delete”, que funcionam praticamente de igual forma em todas as rotas. Quanto ao “delete”,

- **index.js:** é responsável por redirecionar para a main page;
- **adm.js:** é responsável pela gestão de funcionários do sistema, sendo que utiliza operações CRUD definidas no controlador AdmController.js;
- **events.js:** é responsável pela gestão de eventos do sistema, sendo que utiliza operações CRUD definidas no controlador EventsController.js;
- **local.js:** é responsável pela gestão de locais do sistema, sendo que utiliza operações CRUD definidas no controlador LocalController.js;

-**store.js**: é responsável pela gestão de bilhetes, sendo que utiliza operações CRUD definidas no controlador `StoreController.js`;

Temos também a rota “**auth.js**” para o login mas vamos deixar para a próxima milestone.

1.3 Modules

Dividimos o projeto em 4 módulos de modo a facilitar o entendimento e organizar o código da melhor forma possível, sendo eles “Adm”, “Events”, “Local” e “Store”.

Um funcionário é caracterizado pelo nome, email, endereço , palavra-passe, um botão do tipo rádio para verificar se é administrador ou não e notas.

Quanto aos eventos, um evento é caracterizado pelo nome, data, local (já tem que estar criado), preço dos bilhetes (criança, adulto e idoso), número máximo de tickets, nome do ficheiro que é inserido diretamente no site, uma descrição e o ficheiro.

Quanto aos locais, utilizamos a api “geoapify” e fornecendo a latitude e a longitude, a api obtém a rua, o país, o código postal e o nome da localidade.

Quanto ao bilhetes (Store) , têm nome, o evento a que estão associados (já tem que ter um evento criado), e o funcionário responsável por comprar o bilhete.

1.4 Views

As views são um componente crucial de qualquer projeto Express que utilize EJS. Sendo que o nosso projeto envolve operações CRUD como create, edit, list e show, as views assumem um papel ainda mais importante, uma vez que são responsáveis por exibir informações para os utilizadores e permitir que eles executem as operações. Ou seja, permitem que sejam criadas páginas web dinâmicas capazes de se adaptarem a diferentes situações e dados e ainda a diferentes dispositivos e tamanhos de tela.

Ou seja, as views são renderizadas e utilizadas nos controllers respetivos para realizar as funções dos mesmos.

Neste sentido temos as views “create.ejs”, “edit.ejs”, “list.ejs” e “show.ejs”, utilizadas para os funcionários, eventos, locais e compra de bilhetes, sendo que para este último temos ainda a view “listTickets.ejs”.

1.5 Controllers

Neste projeto temos 4 controllers, “AdmController.js”, “EventsController.js”, “LocalController.js”, StoreController.js”. Para além disso temos o “auth.js” para a autenticação através do login mas decidimos abordar isso apenas na próxima milestone.

Nos controllers realizamos as funções referidas acima nas rotas, ou seja, as rotas vão usar os controladores respetivos, fazendo assim todas as operações do sistema.

1.6 Planeamento


O processo de desenvolvimento de um site pode ser complexo, especialmente quando se trata de criar uma interface que seja intuitiva e agradável para os utilizadores. Neste sentido, o planeamento através de mockups é uma etapa fundamental que se deve ter em conta.

Mockups são modelos visuais que representam a estrutura e o layout do site. Ao criar um mockup é possível ter uma visão geral do projeto e identificar possíveis problemas antes de começar ajudando ainda a identificar a melhor maneira de organizar o conteúdo e as funcionalidades do site. Para além disso contribui positivamente para encontrar a melhor maneira de estruturar as views.

Assim sendo, depois de lermos o enunciado, achamos vantajoso realizar mockups e utilizamos a plataforma “Visily ai” para o fazer.


Screen 1

PAW Home Funcionários Eventos Locais Comprar




Funcionários

Selecionar




Eventos

Selecionar



Locais

Selecionar



Comprar Bilhetes

Selecionar

@2023 Copyright

Screen 3

PAW Home Funcionários Eventos Locais Comprar

Create Event

Name

Date

Local

Prices

Kids

Adults

Old

Max.Bilhetes

Selecionar Todos

Save

Eventos List

Screen 2

PAW Home Funcionários Eventos Locais Comprar

Create Employee

Nome

Address

Email

Password

Screen 4

PAW Home Funcionários Eventos Locais Comprar

Coldplay - concerto

Final da Champions

Screen 4

PAW Home Funcionários Eventos Locais Login

Torre dos Clérigos

Estádio da Luz

Screen 5

PAW Home Funcionários Eventos Locais Comprar

Event List

Name

Local

Eventos

Estádio da Luz

Save

Locais List

Screen 5

PAW Home Funcionários Eventos Locais Comprar

Create Local

Latitude

Longitude

Save

Locais List

Screen 3

PAW Home Funcionários Eventos Locais Comprar

Edit Event

Name

Date

Local

Prices

Kids

Adults

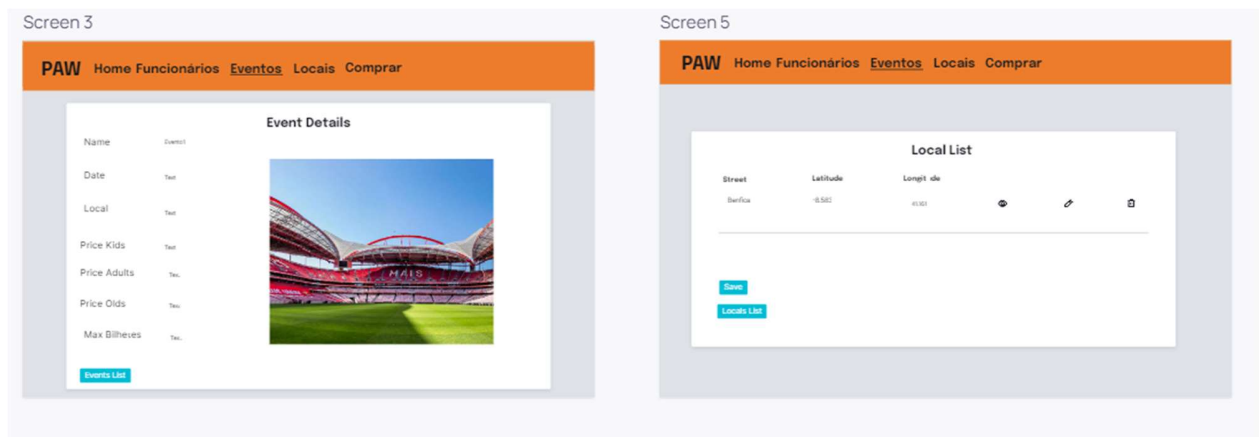
Old

Max.Bilhetes

Selecionar Todos

Save

Eventos List



Neste sentido, os mockups serviram de certa forma como uma espécie de rascunho que foi depois aprimorado e posto em prática para o desenvolvimento do site.

2. Descrição

2.1 Perspetiva do Produto

Nesta 1ª milestone tivemos como foco o backoffice da aplicação.

Assim sendo, debruçamo-nos sobre as operações crud dos funcionários, locais e eventos e sobre a compra dos bilhetes.

Toda a nossa aplicação funciona da seguinte forma: É possível ao funcionário comprar um bilhete mas, para isso, já tem de existir um evento criado. Neste sentido, caso não existam bilhetes comprados para um determinado evento, é possível eliminar todos os dados relativos ao mesmo. No caso de já existirem bilhetes comprados, o evento não é totalmente eliminado, apenas aparece a mensagem que aquele evento foi eliminado mas alguns dos seus parâmetros continuam disponíveis na base de dados.

Relativamente aos locais, recorreremos à API “geoapify”, e apenas fornecendo a latitude e longitude do local a api vai buscar a informação relativa a nome, rua, país, código postal e uma imagem relativa ao local, que foi conseguido através da api e da api key.

Pensamos em colocar já a opção de comprar bilhetes separada do menu dos eventos de modo a, de certa forma, facilitar a abordagem quando passarmos para o frontoffice da aplicação.

2.2 Funcionalidades do Produto

- Crud de funcionários;
- Crud de eventos;
- Crud de locais;
- Serviço de bilheteira

2.3 CSS

Relativamente ao CSS, apesar de pesquisarmos bastante, acabamos por nos guiarmos maioritariamente pelos exemplos apresentados no [bootstrap](#), sendo que depois ajustávamos ao nosso código de forma a ficar o mais adequado possível.

3. Organização de dados

3.1 MongoDB

MongoDB é uma base de dados que permite gerir e guardar um grande volume de dados de forma fácil e eficiente.

Neste projeto utilizamos o Mongo para guardar e gerir os dados do nosso site. Tal foi possível através da utilização do seguinte código:

```
var mongoose = require('mongoose');  
var EmployeeSchema = new mongoose.Schema({
```

4. Introdução – 2ª Milestone

4.1 Propósitos/Objetivos

Nesta segunda milestone foram efetuadas algumas alterações no backoffice, tendo em conta as recomendações propostas pelos docentes, como por exemplo: metemos data de início e de fim dos eventos com as várias restrições associadas, atribuímos roles aos funcionários, atribuímos o NIF como um atributo dos funcionários e colocamos a opção para os locais serem reconhecidos como um evento, caso seja possível comprar bilhete para aquele local.

No entanto, o maior foco desta segunda milestone é a implementação do frontoffice, recorrendo ao Angular, de modo a gerir o website do cliente e compras online. Para a realização do frontend é utilizado o mesmo projeto do backend, desenvolvido no primeiro milestone, sendo que foram acrescentadas algumas funcionalidades do backend para dar suporte ao frontend. Utilizamos a RestAPI para conectar o backoffice ao frontoffice e o STRIPE para gerir a compra de bilhetes.

Ou seja, a ligação entre o frontend e o backend é estabelecida por meio das requisições HTTP enviadas pelo frontend para os endpoints do backend, sendo que as respostas do backend são processadas pelo frontend para realizar as ações apropriadas na interface do cliente.

4.2 API's usadas

Nesta segunda milestone foram utilizadas 2 apis: a RestAPI, utilizada de forma a ser estabelecida a ligação entre o backoffice e o frontoffice e o “Stripe”, uma API que foi utilizada para a compra dos bilhetes.

A classe “RestAPIService” é um serviço do Angular responsável por estabelecer a comunicação entre o frontoffice e o backoffice através de requisições HTTP. Por exemplo o “getEvents()” realiza uma requisição get para o endpoint presente no backend que é utilizado para obter eventos do backend, sendo chamado pelo frontend para exibir uma lista de eventos.

Ou seja, através do “HttpClient” do Angular, os métodos do “RestAPIService” enviam as requisições HTTP para os endpoints definidos no backend, que está configurado para responder a essas solicitações. Resumidamente, a classe “RestAPIService” fornece métodos para realizar requisições HTTP ao backend com o objetivo de obter os dados relativos a eventos, locais e bilhetes, que são necessários para a interface do cliente.

Quanto à API “Stripe”, é uma plataforma de pagamento online que permite a integração de serviços de pagamento em aplicativos e sites. Funciona da mesma forma que a RestAPI, ou seja, são usadas funções no frontend que são conseguidas através de requisições HTTP para os endpoints respetivos no backend. No entanto, apesar de tentarmos usar as funcionalidades desta API para simular o método de pagamento, não nos foi possível por falta de tempo. Neste sentido, temos o

botão para adicionar as compras ao carrinho que acaba por não dar para simular pagamento e temos outro botão para guardar os bilhetes na base de dados, que funciona corretamente.

4.3 Models

Em Angular, os modelos (models) são classes ou interfaces usadas para representar a estrutura e os tipos de dados de um objeto.

Para este projeto achamos importante dividir em 5 models: “events.ts”, “locals.ts”, “object.ts”, “tickets.ts” e “cart.model.ts”.

Os módulos vão receber construtores com os parâmetros respetivos, parâmetros estes que estão associados a cada um desses módulos no backend da aplicação.

O model “object.ts” recebe um array de eventos, locals e tickets, tudo numa função.

Desta forma, é possível manipular a informação de forma mais fácil.

4.4 Components

Relativamente aos componentes, que geramos através do comando “ng generate component”.

- events-list.component

Este componente é responsável por exibir uma lista de eventos numa tabela através da biblioteca DataTables, sendo que são obtidos os dados dos eventos do backend, através da RestAPI.

No método ngOnInit() é chamada a função getEvents() para obter a lista de eventos, utilizando a RestAPIService.

O método “ngOnDestroy” é para evitar o desperdício de memória, através do cancelamento da inscrição do dtTrigger ao destruir o componente.

O método construtor deste componente é utilizado para as dependências dos serviços RestAPIService, ActivatedRoute e Router.

A função view(id: string) é chamada ao clicar em um evento da tabela. Ela navega para a página de visualização do evento com base no ID, utilizando o serviço Router e permite a seleção dos bilhetes para a compra.

- headercomponent

Este componente é responsável por exibir o cabeçalho, que é utilizado em diversas páginas da aplicação.

Neste caso, o componente possui a função “logout()”, que é chamada quando o usuário clica no botão de logout.

No construtor do componente, são utilizados dois serviços: Router e AuthRestServiceService. O serviço Router é utilizado para realizar a navegação entre as páginas, enquanto que o serviço “AuthRestServiceService” é responsável por lidar com a autenticação.

```
<li class="nav-item"><a href="/events-list" class="nav-link">Comprar Bilhetes</a></li>
    <li class="nav-item"><a href="/user-details" class="nav-link">Perfil</a></li>
    <li class="nav-item"><a href="#" class="nav-link" (click)="logout()">Log Out</a></li>
</ul>
```

O Header, que terá as opção de LogOut, Comprar Bilhetes e Perfil funciona da seguinte forma: ao clicar em “Comprar Bilhetes” o utilizador é redirecionado para a página da lista de eventos, através da referência introduzida; ao clicar em Perfil o utilizador é redirecionado para a página dos detalhes relativos ao utilizador e ao clicar em LogOut, uma vez que “href=#”, ou seja, o link não tem nenhuma rota definida, quando o usuário clicar no botão será chamada a função logout(), chamada no “header.component.ts” e definida no “auth-rest-service-service” (apresentado na figura abaixo), que remove os dados relativos ao “currentUser” e redireciona novamente para a página de login.

```
logout() {
  localStorage.removeItem('currentUser');
}
```

-loginComponent

Este componente é responsável pela exibição e lógica da página do login.

O construtor do componente recebe o “Router” e o serviço “AuthRestServiceService” como parâmetros, o que permite que o componente utilize esses recursos para a navegação e autenticação.

A função login é chamada quando o utilizador clica no botão de login, sendo chamado o método login do serviço AuthRestServiceService passando o email e a senha fornecidos pelo utilizador (figura abaixo).

```
login(email: string, password: string): Observable<any> {  
    return this.http.post<any>('http://localhost:3000/api/v1/auth/login', new  
    LoginModel(email, password));
```

Em seguida, através do método subscribe a função é inscrita para receber a resposta do serviço. Caso o usuário seja autenticado com sucesso (verificado pela presença de um token no objeto de resposta), os detalhes do usuário, bem como o token respetivo, são armazenados no localStorage. O NIF e o email também são armazenados no localStorage para uso posterior. Por fim, o usuário é redirecionado para a página "/events-list", onde é apresentada a lista de eventos disponíveis para possível compra.

Se ocorrer um erro no login, é exibido um alerta informando o erro.

A função "register()" funciona da mesma forma que a função login() mas, neste caso, é chamada quando o utilizador aciona o botão de SignUp.

Resumidamente, este componente é responsável por gerir a lógica da autenticação dos utilizadores, permitindo que estes façam se registem, façam login e efetuem logout, interagindo com o serviço "AuthRestServiceService" para realizar estas operações e utiliza o serviço "Router" para navegar entre as várias páginas.

- ticket-create-component

Este componente é responsável pela exibição e lógica da página de criação de bilhetes.,

O construtor do componente recebe o "CartService", o "Router", o "ActivatedRoute" e a "RestAPIService" como parâmetros, o que permite que o componente utilize esses recursos para manipular o carrinho de compras, navegar entre as páginas e fazer solicitações à RestAPI de modo a comunicar com o backend.

O método "ngOnInit" é executado durante a inicialização do componente e obtém o Id do evento a partir dos parâmetros da rota atual usando "route.snapshot.params['id']". De seguida, faz uma solicitação ao serviço "RestAPIService" de modo a obter os detalhes do evento com o Id correspondente.

A função "navigateToItems()" é chamada quando o usuário clica num botão para navegar para a página de lista de itens, utilizando o Router para redirecionar o usuário para a página desejada ("/item-list").

A função "calcularPrecoTotal()" é chamada sempre que há uma alteração na quantidade selecionada de bilhetes, calculando o preço total com base nas quantidades selecionadas e nos preços definidos no backend. O resultado é armazenado na variável resultado.

A função “adicionarAoCarrinho()” é chamada quando o usuário clica em um botão para adicionar o ingresso ao carrinho de compras. Ela cria um objeto evento do tipo “CartItem”, preenchendo os detalhes do bilhete, como o nome, preço, quantidade selecionada e Id do evento. Em seguida, utiliza o serviço “CartService” para adicionar o evento ao carrinho de compras e redireciona o utilizador para a página do carrinho (“/cart”).

A função add é chamada quando o utilizador clica em um botão para adicionar o bilhete ao sistema. Ela cria um objeto ticket do tipo Ticket, preenchendo os detalhes do bilhete, como nome do evento, quantidades selecionadas, preço total, Id do evento e o NIF do utilizador que adicionou o bilhete obtido através do localStorage. De seguida, faz uma solicitação ao serviço “RestAPIService” para adicionar o ticket ao sistema e, se a operação for concluída com sucesso, exibe um alerta informando que a compra foi realizada.

- userDetailsComponent

Este componente é responsável por exibir os detalhes relativos ao utilizador.

O construtor deste componente recebe o serviço “AuthRestServiceService” e o serviço “ActivatedRoute” como parâmetros, o que permite que o componente utilize esses serviços para autenticação e manipulação de rotas.

A variável “currentUser” é do tipo Adm | null, o que significa que pode conter um objeto do tipo Adm ou o valor null e inicialmente é definida como null.

O método “ngOnInit()” é executado durante a inicialização do componente e chama o método “getUserDetails()”, método que é responsável por obter os detalhes do utilizador através do serviço “authService”. Obtém o valor do NIF do usuário pelo localStorage e, em seguida, faz uma solicitação ao serviço getUser passando esse NIF. O resultado da solicitação é armazenado na propriedade currentUser.

- ticket-list-component

Este componente é responsável por exibir a lista de bilhetes associada a um funcionário específico, através da comunicação com o serviço de autenticação e com o serviço REST para obter os detalhes do utilizador e os bilhetes associados.

O construtor do componente recebe o “Router” e o serviço “AuthRestServiceService” como parâmetros, o que permite que o componente utilize esses recursos para a navegação e autenticação.

O método ngOnInit() é implementado para ser executado quando o componente é inicializado e chama os métodos “getUserDetails()” e getTickets()”.

O método “getUserDetails()” é utilizado para obter os detalhes do utilizador que tem login efetuado, através do serviço “AuthRestServiceService”, e do Nif do utilizador presente no LocalStorage.

```
getCurrentUser(): any {
    const currentUser = localStorage.getItem('currentUser');
    return currentUser ? JSON.parse(currentUser) : null;
}

getUser(id: string): Observable<Adm> {
    const currentUser = this.getCurrentUser();

    if (currentUser && currentUser.id === id) {
        // Returns an Observable that emits the current user stored in the service
        return of(currentUser);
    } else {
        // Gets the user details from the backend using the provided ID
        return this.http.get<Adm>(`${endpoint.replace(':id', id)}`, httpOptions);
    }
}
```

O método “getTickets()” é utilizado para obter os bilhetes associados ao nif do utilizador que efetuou a compra dos bilhetes, utilizando mais uma vez o serviço “AuthRestServiceService” (figura abaixo) e o nif do utilizador presente no localStorage.

```
getTickets(id: string): Observable<Ticket[]> {
    const currentUser = this.getCurrentUser();

    if (currentUser && currentUser.id === id) {
        // Returns an Observable that emits the current user stored in the service
        return of(currentUser);
    } else {
        // Gets the user details from the backend using the provided ID
        return
this.http.get<Ticket[]>(`http://localhost:3000/api/v1/auth/tickets/`+id,
httpOptions);
    }
}
```

Os bilhetes são armazenados na matriz Ticket[].

4.5 Referências

Quando surgiam dúvidas procuramos e pesquisamos por informação que nos pudesse ser útil, sendo que recorremos maioritariamente aos seguintes sites:

- [W3Schools Online Web Tutorials](#)
- [Stack Overflow - Where Developers Learn, Share, & Build Careers](#)
- [Bootstrap · The most popular HTML, CSS, and JS library in the world. \(getbootstrap.com\)](#)

4.6 Observações

Relativamente aos requisitos do enunciado e devido a algumas dúvidas que fomos tendo não tivemos tempo suficiente para os abordar a todos. Neste sentido, decidimos fazer aqueles que achamos ser de maior importância.

Relativamente ao tópico de consultar todos os eventos com a possibilidade de filtrar de acordo com determinados parâmetros utilizamos a biblioteca “DataTables” e é nos possível filtrar os eventos da lista pelo nome.

Quanto ao tópico de consultar a conta corrente de pontos ganhos, é nos possível ver a quantidade de pontos de cada utilizador com login e compras efetuadas, sendo que a quantidade de pontos é diretamente proporcional ao preço total que o utilizador despendeu na compra dos bilhetes.

Para além disso conseguimos consultar os bilhetes adquiridos por cada utilizador através do “ticket-list”.

Assim sendo, apenas não conseguimos utilizar os pontos para efetuar descontos nas compras, por falta de tempo.