



Escola Secundária Domingos Sequeira

ESDS

Projeto de Software



CarsGallery

O lar dos carros.

Projeto realizado por:

Roberto

**Curso Profissional de
Gestão e Programação de Sistemas Informáticos**

Cofinanciado por:



UNIÃO EUROPEIA
Fundo Social Europeu



RELATÓRIO DO PROJETO

Cofinanciado por:



UNIÃO EUROPEIA
Fundo Social Europeu



RELATÓRIO DO PROJETO

Índice

Introdução	8
Memória Descritiva	9
Enquadramento Teórico	10
Metodologia	11
Tecnologias/Recursos	12
Constituição do projeto	14
Modelo da base de dados e o Prisma	16
O que é Vue? E porquê?	24
Estrutura do projeto	25
Ficheiro env	28
Outros ficheiros	29
Como é que o frontend comunica com o backend?	30
Vuex - Gerenciamento do estado em Vue	32
Rotas backend	34
Rotas frontend	39
Rotas inexistentes	40
Autenticação Bearer	41
Middlewares	42
Como funciona o sistema de autenticação?	43
Temas	46
Traduções	49
Criação de utilizador	52
Criação de “Posts”	57
Posts	60
Comentários	63
Likes	68
Seguidores	72
Procurar utilizadores	76



RELATÓRIO DO PROJETO

Definições da conta	79
Deteção de erros	81
Layout	81
Conclusão	94
Bibliografia/Referências Bibliográficas Eletrónicas	100

Índice de figuras

Figura 1- Gráfico de Gantt	11
Figura 2 - Comunicação Frontend-Backend	14
Figura 3 - Exemplo de um teste à rota login (POST) com o Insomnia	14
Figura 4 - Exemplo de uma “query” à tabela dos users com o Prisma	15
Figura 5 - scheme.prisma	16
Figura 6 - Tabela Post	17
Figura 7 - Tabela PostLike	18
Figura 8 - Tabela Comment	19
Figura 9 - Tabela Profile	20
Figura 10 - Tabela Tag	20
Figura 11 - Tabela User	21
Figura 12 - Diagrama das relações das tabelas, geradas pelo Prisma	23
Figura 13 - Logotipo Vue	24
Figura 14 - Estrutura de uma página utilizando a lógica do Vue	24
Figura 15 - Estrutura do projeto (frontend e backend)	25
Figura 16 - Estrutura frontend	26
Figura 17 - Estrutura backend	27
Figura 18 - Ficheiro env do frontend	28
Figura 19 - Ficheiro env do backend	28
Figura 20 - Comunicação entre backend e frontend pelo Vuex	30
Figura 21 - Página register frontend	31
Figura 22 - Dados enviados para a rota	31
Figura 23 - Gerenciamento do estado pelo Vuex	32
Figura 24 - Action "login" no vuex.js	33
Figura 25 - Váriaveis do "state" no vuex.js	33
Figura 26 - Mutation "SET_LOADING_STATUS" no vuex.js	33
Figura 27 - Componente Loading no "Login.vue"	33
Figura 28 - Rotas GET PUT POST e DELETE	34
Figura 29 - Rotas utilizadores	35

RELATÓRIO DO PROJETO

Figura 30 - Rotas posts	36
Figura 31 - Rotas tags	37
Figura 32 - Rotas seguidores	37
Figura 33 - Rotas comentários	38
Figura 34 - Rotas do frontend	39
Figura 35 - Exemplo (função que apaga um utilizador no frontend vuex.js)	41
Figura 36 - Rota no backend (routes.ts)	41
Figura 37 - Ficheiros com os middlewares	42
Figura 38 - Ações que ocorrem durante o login	43
Figura 39 - Fluxograma descrevendo um login	44
Figura 40 - Chamada ao backend (api) pela rota login, enviando os dados, e a resposta obtida	44
Figura 41 - Resposta caso os dados não sejam válidos	45
Figura 42 - Popup sucesso (dados válidos)	45
Figura 43 - Popup erro (dados inválidos)	45
Figura 44 - vuex.js	46
Figura 45 - Comments.vue	46
Figura 46 - Comments.vue	46
Figura 47 - ProfileSettings.vue	47
Figura 48 - ProfileSettings.vue	47
Figura 49 - Mutation "SET_DARK_THEME" em vuex.js	47
Figura 50 - Exemplo Dark Mode (tema escuro) no Profile.vue	48
Figura 51 - Exemplo Dark Mode (tema claro) no Profile.vue	48
Figura 52 - Chave "profile" em pt.json	49
Figura 53 - Chave "profile" em en.json	49
Figura 54 - Profile.vue	49
Figura 55 - Criação de uma constante com vários dados necessários a cada chamada	50
Figura 56 - Chave "email_notfound" em en.json	50
Figura 57 - Figura 39 - Chave "email_notfound" em pt.json	50
Figura 58 - Função login, dentro de UserController.ts	51
Figura 59 - Erro caso email seja inválido	51
Figura 60 - Rota chamada	51
Figura 61 - Campos a serem introduzidos para a criação de um novo utilizador	52
Figura 62 - Dados enviados para a rota	52
Figura 63 - Utilização de "regras" para os campos, neste caso, para o nome	53
Figura 64 - Verificação se um nome é demasiado curto	53
Figura 65 - Verificação de um email inválido (sem @)	53
Figura 66 - Ficheiro countries.json com todos os países	54
Figura 67 - Armazenamento dos países e siglas em arrays no mounted do Register.vue	54
Figura 68 - Código do componente com todos os países	55
Figura 69 - Componente com todos os países	55
Figura 70 - Verificação da password	55
Figura 71 - Password válida (barra verde)	56
Figura 72 - Regras para os campos ao criar conta em Register.vue	56

RELATÓRIO DO PROJETO

Figura 73 - Criar post (rota /uploadpost)	57
Figura 74 - Campos do novo post	57
Figura 75 - Preview do post no perfil com os campos	58
Figura 76 - Função onPost dentro de UploadPost.vue	58
Figura 77 - Função "createPost", em vuex.js	59
Figura 78 - Sucesso na criação do Post	59
Figura 79 - Componente PostViewer	60
Figura 80 - Componente PostViewer quando clicamos no Post	60
Figura 81 - Profile.vue	61
Figura 82 - Componente PostFeed	61
Figura 83 - Home.vue	62
Figura 84 - Tabela dos comentários	63
Figura 85 - Tabela dos posts	63
Figura 86 - Rotas utilizadas para comentários (backend routes.ts)	64
Figura 87 - Função getPostsComments, utilizado na rota "GET /post/:id/comments"	64
Figura 88 - Resposta da rota "GET /post/:id/comments"	64
Figura 89 - Componente Comments.vue	65
Figura 90 - Comentário introduzido	65
Figura 91 - Componente utilizado dentro de outro componente	66
Figura 92 - Função quando um comentário é introduzido	66
Figura 93 - Função "commentOnPost" que chama a rota do backend que adiciona um comentário ao post. (rota POST "/post/:id/comments")	66
Figura 94 - PostViewer.vue	67
Figura 95 – Post sem comentários	67
Figura 96 - Post com comentários	67
Figura 97 - Código do componente Like.vue	68
Figura 98 - Componente se não tivermos like	69
Figura 99 - Componente se tivermos like	69
Figura 100 - Quando o componente é criado, verifica se o utilizador tem like no post ou não	69
Figura 101 - Action "DoILikeThePost" chamada quando o componente é criado	69
Figura 102 - Função userLikedPost no backend, que verifica se na tabela PostLike o utilizador tem like no post ou não	70
Figura 103 - Tabela PostLike	70
Figura 104 - Componente PostViewer a usar o componente Like	71
Figura 105 - Parte visual a que o código da figura acima se refere	71
Figura 106 - Perfil do utilizador	72
Figura 107 - Popup ao seguir utilizador	72
Figura 108 - Função handleFollowClick (Profile.vue)	72
Figura 109 - actions followUser e unfollowUser do vuex.js	73
Figura 110 - action doFollowTheUser	73
Figura 111 - Perfil de um utilizador que não sigo	74
Figura 112 - Perfil de um utilizador que sigo	74
Figura 113 - Código que ordena os posts	74

RELATÓRIO DO PROJETO

Figura 114 - Barra para pesquisar utilizadores (componente SearchUsers.vue)	76
Figura 115 - Função que é executada cada vez que a barra recebe input	77
Figura 116 - Componente autocomplete (que é a barra de pesquisa)	77
Figura 117 - Componente autocomplete para a pesquisa de utilizadores	78
Figura 118 - Componente autocomplete para a pesquisa de utilizadores	78
Figura 119 - Função que redireciona para o perfil selecionado	78
Figura 120 - ProfileSettings.vue	79
Figura 121 - ProfileSettings.vue	79
Figura 122 - Dialog que pede o novo campo (ProfileSettings.vue)	79
Figura 123 - Função handleDone, que chama a action "updateUserProfile"	80
Figura 124 - Action "updateUserProfile"	80
Figura 125 - Função handleError em functions.js	81
Figura 126 - Popup com erro	81
Figura 127 - Página inicial para visitantes	82
Figura 128 - Página inicial para visitantes	82
Figura 129 - Página login	83
Figura 130 - Página login telemóvel	83
Figura 131 - Página registrar	84
Figura 132 - Página registrar telemóvel	84
Figura 133 - Página inicial	85
Figura 134 - Página inicial telemóvel	85
Figura 135 - Página perfil de um utilizador	86
Figura 136 - Página perfil de um utilizador telemóvel	87
Figura 137 - Página para criação de post	87
Figura 138 - Página para criação de post telemóvel	88
Figura 139 - Campos na criação do post	88
Figura 140 - Campos na criação do post	88
Figura 141 - Definições gerais	89
Figura 142 - Definições gerais telemóvel	89
Figura 143 - Definições perfil	90
Figura 144 - Definições conta	90
Figura 145 - Ver posts	91
Figura 146 - Procura de posts por tag	92
Figura 147 - Procura de posts por tags telemóvel	92
Figura 148 - Página inicial para visitantes	93
Figura 149 - Página login (tema claro)	93
Figura 150 - Página register (tema claro)	94
Figura 151 - Página inicial para utilizadores (tema claro)	94
Figura 152 - Perfil de utilizador (tema claro)	95
Figura 153 - Página para publicar novo post (tema claro)	95
Figura 154 - Página para editar os campos do novo post (tema claro)	96
Figura 155 - Página para procura de posts por tag (tema claro)	96
Figura 156 - Configurações da conta (tema claro)	97



RELATÓRIO DO PROJETO

Figura 157 - Configurações do perfil (tema claro)	97
Figura 158 - Configurações gerais (tema claro)	98
Figura 159 - Comentários num post (tema claro)	98



RELATÓRIO DO PROJETO

Introdução

A criação e evolução da informática possibilitou um avanço das atividades relacionadas a esta área na quase totalidade das atividades humanas, iniciando pelas Engenharias e atingindo os mais diversos setores.

A PAP (Projeto de Aptidão Profissional) é um projeto pessoal com o objetivo de aplicar os conhecimentos adquiridos ao longo do curso.

A minha PAP consiste numa rede social destinada só à partilha de fotos de carros. Investi nesta ideia visto que é um conceito que ainda não existe e acredito que pode ser interessante especialmente para pessoas fascinadas pelo tema. Claro que no futuro poderão ser adicionadas novas funcionalidades ou até o código ser usado para projetos diferentes.



RELATÓRIO DO PROJETO

Memória Descritiva

CarsGallery tem como principal conceito a partilha de fotos de carros da maneira mais simples e prática possível. Esta plataforma inclui várias ferramentas que permitem uma melhor experiência na procura de carros específicos. Este projeto é considerado MEVN uma vez que enquadra várias tecnologias como MySQL, ExpressJS, VueJS e NodeJS.

O produto final esperado é um site funcional com várias ferramentas essenciais de uma rede social, como postagem de fotos/vídeos, adicionar comentários, adicionar ou remover likes, seguir ou deixar de seguir utilizadores, entre outras. Infelizmente não irá ter “TODAS” as ferramentas de uma rede social, pois a falta de tempo e conhecimento limitou-me em alguns aspetos, mas espero futuramente adicionar essas ferramentas a este projeto e quem sabe, o site poder ser utilizado em ambiente real ou até o código ser aproveitado e fazerem outro tipo de rede social com base neste projeto.

Com a realização deste projeto, espero conseguir melhorar o meu conhecimento acerca de desenvolvimento web, não só em termos visuais, mas também em termos de segurança, e no geral, ter um conhecimento mais “profundo” de um desenvolvimento full stack.



RELATÓRIO DO PROJETO

Enquadramento Teórico

Durante estes três anos de curso, consegui adquirir vários conhecimentos que me facilitou a realização deste trabalho, como as linguagens de programação SQL, PHP e JavaScript, e também outras ferramentas como HTML e CSS.

Os objetivos deste projeto são:

- Criação de utilizadores
- Autentificação de utilizadores
- Criação/remoção de “posts” (fotos ou vídeos)
- Adicionar/remover “likes” (gostos)
- Adicionar comentários
- Adicionar/remover “tags”
- Sistema de procura de posts por tags
- Seguir/deixar de seguir utilizadores
- Base de dados
- Traduções para o site inteiro
- Tema escuro e claro (para o site)
- Configuração de campos do utilizador (email, password, avatar, biografia, etc....)

RELATÓRIO DO PROJETO

Metodologia

Calendarização das atividades:

- Planear e estruturar todas as ideias (29 abril – 3 maio)
- Criação da base de dados (3 maio – 8 maio)
- Desenvolvimento backend (10 maio – 14 junho)
- Desenvolvimento do frontend (27 maio – 1 julho)
- Relatório (15 maio – 3 julho)
- Teste e depuração de erros (15 junho – 3 julho)
- Finalização do projeto (3 julho)

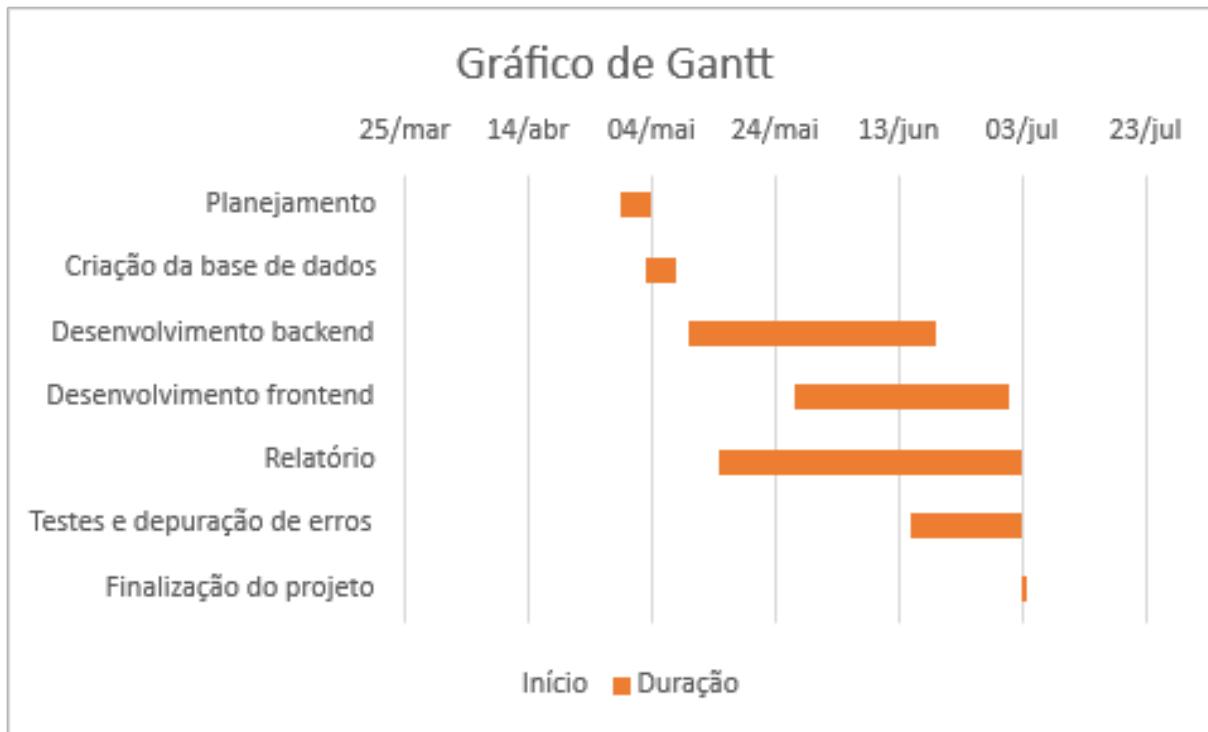


Figura 1- Gráfico de Gantt



RELATÓRIO DO PROJETO

Tecnologias/Recursos

Na realização deste projeto, foram utilizados diversos programas e tecnologias necessários para um melhor desenvolvimento do mesmo.

Recursos utilizados:

Recurso	Descrição	Utilização no projeto
Visual Studio Code	Editor de texto	Escrita/programação do código fonte do site
GIMP	Ferramenta de manipulação de imagem	Edição de diversas imagens para o site
MySQL	Sistema de administração de base de dados.	Criação das diversas bases de dado
JavaScript	Linguagem de programação de alto nível, usada para aplicações web	Programação geral do site
Npm	Gerenciador de pacotes JavaScript	Instalação de dependências
TypeScript	Um superset da linguagem JavaScript para facilitar/organizar escrita de código.	Programação do backend.
ExpressJS	Framework de JavaScript para desenvolvimento de aplicações JavaScript e de web APIs (backend)	Backend do site, ou seja, a sua API
VueJS	Framework de Javascript para o desenvolvimento de interfaces (frontend)	Frontend do site, ou seja, o design e a interface visual

RELATÓRIO DO PROJETO

NodeJS	Ambiente de execução de JavaScript server-side	Correr a web API (backend) do site
SCSS + SASS (Syntactically Awesome Style Sheets)	Linguagem de folhas de estilo	Desenvolvimento da parte visual do site
Vuetify	Framework de CSS (Cascading Style Sheets)	Desenvolvimento da parte visual do site
HTML (HyperText Markup Language)	Linguagem de marcação	Desenvolvimento do site
LibreOffice Writer	Software de edição de texto	Realização do relatório
Eslint	Ferramenta de análise de código	Formatação do código JavaScript e deteção de erros
Firefox	Browser/navegador	Testagem do site
Bcrypt	Método de criptografia do tipo hash para passwords	Criptografar as passwords dos utilizadores
Prisma	Ferramenta de mapeamento objeto-relacional	Programação e manutenção de base de dados MySQL, sem usar a linguagem SQL (buscar/criar dados, criar tabelas, etc...)
Insomnia	Ferramenta para testagem de web APIs	Testagem das rotas simples do backend
Git	Hospedagem de código aberto	Hospedagem do código do site

RELATÓRIO DO PROJETO

Constituição do projeto

Este projeto divide-se em 2 partes principais, o frontend, a parte visual do site onde o utilizador interage, e a web API (servidor backend) que processa toda a informação necessária. Assim, tudo o que é armazenamento/processamento ou validação de dados é feito através do backend. Já tudo o que se trata de páginas web e design, está relacionado com frontend.

O frontend foi desenvolvido em VueJS e é corrido no navegador do utilizador, enquanto que o servidor backend foi desenvolvido em ExpressJS, mas usa-se o NodeJS para correr a web API sem recurso a um navegador. No sistema de login, por exemplo, o frontend trata da apresentação visual da página com caixas para introdução de dados do utilizador e outros mecanismos, enquanto que o backend é responsável por validar esses dados recorrendo à base de dados.

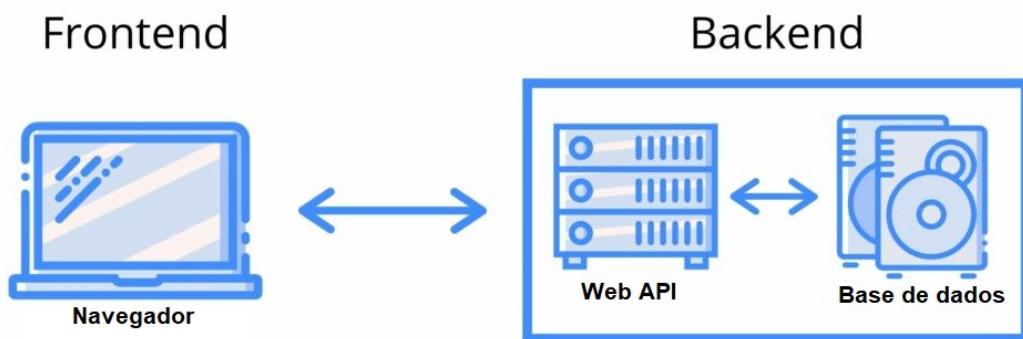


Figura 2 - Comunicação Frontend-Backend

São também usadas outras ferramentas como por exemplo o Prisma, que permite trabalhar com a base de dados, sem necessidade de escrever código SQL, ou também Insomnia, que ajudou na testagem das rotas da web API tendo uma interface mais acessível e de fácil uso, sendo assim mais fácil a depuração da mesma.

POST ▾	http://localhost:3000/login	Send	404 Not Found	174 ms	64 B
JSON ▾	Auth ▾	Query	Header 1	Preview ▾	Header 17
1 ▾ { 2 "email": "roby@gmail.com", 3 "password": "teste123" 4 }				1 ▾ { 2 "message": "The email does not exist", 3 "error": "Email not found" 4 }	Cookie

Figura 3 - Exemplo de um teste à rota login (POST) com o Insomnia



RELATÓRIO DO PROJETO

```
// verifica se email é valido, se for, retorna os dados
// do utilizador e também o numero de seguidores e seguindo
const userExists = await database.user.findUnique({
  where: {
    email: userEmail
  },
  include: { following: true, followedBy: true }
})
```

Figura 4 - Exemplo de uma “query” à tabela dos users com o Prisma



RELATÓRIO DO PROJETO

Modelo da base de dados e o Prisma

Como não sou muito fã de base de dados, o meu conhecimento sobre o assunto é mínimo. Depois de alguma pesquisa, encontrei esta ferramenta “Prisma” que me cativou bastante a atenção, pois, a sua utilização é bastante fácil e direta, facilitando-me assim a gestão e programação da base de dados, sem usando SQL.

O prisma funciona à base do seu “esquema”. Ao executar o comando para iniciar o projeto prisma “npx prisma init”, irá criar um ficheiro chamado “schema prisma”. Esse ficheiro é importante e o ponto central pois é lá que irei criar todas as tabelas e relações. Com esse ficheiro criado, ao executar “npx prisma migrate dev”, ele cria a base de dados e as tabelas no servidor MySQL.

A partir daí, posso utilizar o prisma e as suas ferramentas para a gestão da base de dados sem ter que escrever/programar em SQL.

Agora irei mostrar as tabelas e as relações entre os vários campos

Para começar, “datasource db” irá ser onde se define os dados da base de dados, no meu caso, irei usar MySQL, e o link da base de dados está no ficheiro env ([explicado mais à frente](#)).

```
schema.prisma ×
prisma > schema.prisma
1 // This is your Prisma schema file,
2 // learn more about it in the docs: https://pris.ly/d/prisma-schema
3
4 datasource db {
5   provider = "mysql"
6
7   url      = env("DATABASE_URL")
8 }
9
```

Figura 5 - schema.prisma

RELATÓRIO DO PROJETO

Tabela dos posts:

```

16 model Post {
17
18     id      Int      @default(autoincrement()) @id
19
20     createdAt DateTime @default(now())
21
22     updatedAt DateTime @updatedAt
23
24     title    String   @db.VarChar(255)
25
26     content  String?
27
28     likes    PostLike[]
29
30     published Boolean @default(false)
31
32     author   User     @relation(fields: [authorId], references: [id])
33
34     authorId Int
35
36     file     PostFile? @relation(fields: [fileId], references: [id])
37
38     fileId   Int?
39
40     tags     Tag[]
41
42     comments Comment[]
43 }
44

```

Figura 6 - Tabela Post

id	ID único do post (que incrementa automaticamente)
createdAt e updatedAt	Data da criação do post
title	Título do post
content	Descrição do post
likes	Array com a quantidade de likes do post (campo interligado com a tabela “PostLike”)
published	Bool que indica se foi publicado com sucesso
author	Campo ligado à tabela User (que se refere ao autor do post)
authorId	ID único do autor do post

RELATÓRIO DO PROJETO

file	Campo ligado à tabela PostFile (que se refere ao ficheiro)
fileId	ID único do ficheiro
tags	Array com as tags (campo interligado com a tabela Tag)
comments	Array com os comentários (campo interligado com a tabela Comment)

Tabela “PostLike”, que servirá para a relação entre um like de um utilizador num post:

```

45  model PostLike {
46    id      Int      @default(autoincrement()) @id
47
48    user    User    @relation(fields: [userId], references: [id])
49
50    userId  Int
51
52    post    Post    @relation(fields: [postId], references: [id])
53
54    postId  Int
55  }

```

Figura 7 - Tabela PostLike

id	ID único da relação
user	Campo ligado à tabela User (que se refere ao utilizador que deu like)
userId	ID único do utilizador que deu like
post	Campo ligado à tabela Post (que se refere ao Post que o utilizador deu like)
postId	ID único do post

RELATÓRIO DO PROJETO

Tabela “Comment”, que servirá para o armazenamento dos comentários:

```

67 model Comment {
68
69     id      Int      @default(autoincrement()) @id
70
71     author  User    @relation(fields: [authorId], references: [id])
72
73     authorId Int
74
75     postId   Int
76
77     post     Post    @relation(fields: [postId], references: [id])
78
79     content  String
80
81     createdAt DateTime @default(now())
82
83     updatedAt DateTime @updatedAt
84 }
85

```

Figura 8 - Tabela Comment

id	ID único do comentário
author	Campo ligado à tabela User (que se refere ao utilizador que comentou)
authorId	ID único do utilizador que comentou
postId	ID único do post
post	Campo ligado à tabela Post (que se refere ao Post que o utilizador deu like)
content	Comentário em si
createdAt e updatedAt	Data de criação do comentário

RELATÓRIO DO PROJETO

Tabela “Profile”, que servirá para o armazenamento de alguns campos do perfil do utilizador:

```

86  model Profile {
87
88    id      Int      @default(autoincrement()) @id
89
90    photo  Bytes?   @db.LongBlob
91
92    user   User     @relation(fields: [userId], references: [id])
93
94    userId Int      @unique
95
96

```

Figura 9 - Tabela Profile

id	ID único do perfil
photo	Foto de avatar do utilizador
user	Campo ligado à tabela User (que se refere ao utilizador do perfil)
userId	ID único do utilizador

Tabela “Tag”, que servirá para o armazenamento de tags:

```

134  model Tag {
135    id Int @default(autoincrement()) @id
136
137    name String @unique
138
139    posts Post[]
140
141

```

Figura 10 - Tabela Tag

id	ID único da tag
name	A tag em si
posts	Array com os posts com a tag (campo ligado à tabela posts)

RELATÓRIO DO PROJETO

Tabela “User”, que servirá para o armazenamento de utilizadores:

```

98  model User {
99
100    id      Int      @default(autoincrement()) @id
101
102    email   String   @unique
103
104    password String
105
106    name    String
107
108    username String  @unique
109
110    birthday DateTime
111
112    country String?
113
114    bio     String?
115
116    posts   Post[]
117
118    postsCommented Comment[]
119
120    postsLiked PostLike[]
121
122    profile  Profile?
123
124    followedBy User[]  @relation("UserFollows", references: [id])
125
126    following User[]  @relation("UserFollows", references: [id])
127
128    createdAt DateTime @default(now())
129
130    updatedAt DateTime @updatedAt
131
132  }
133

```

Figura 11 - Tabela User

id	ID único do utilizador
email	Email único do utilizador
password	Password do utilizador
name	Nome do utilizador
username	Nome de utilizador do utilizador



RELATÓRIO DO PROJETO

birthday	Data de nascimento do utilizador
country	País do utilizador
bio	Biografia do utilizador
posts	Array com os posts do utilizador (campo interligado à tabela Posts)
postsCommented	Array com todos os comentários a posts do utilizador (campo interligado à tabela Comments)
postsLiked	Array com todos os likes a posts do utilizador (campo interligado à tabela PostLike)
profile	Campo com alguns dados do utilizador usados para o perfil (campo interligado à tabela Profile)
followedBy	Seguidores do utilizador (campo interligado à tabela User)
following	Utilizadores que o utilizador segue (campo interligado à tabela User)
createdAt e updatedAt	Data da criação do utilizador

RELATÓRIO DO PROJETO

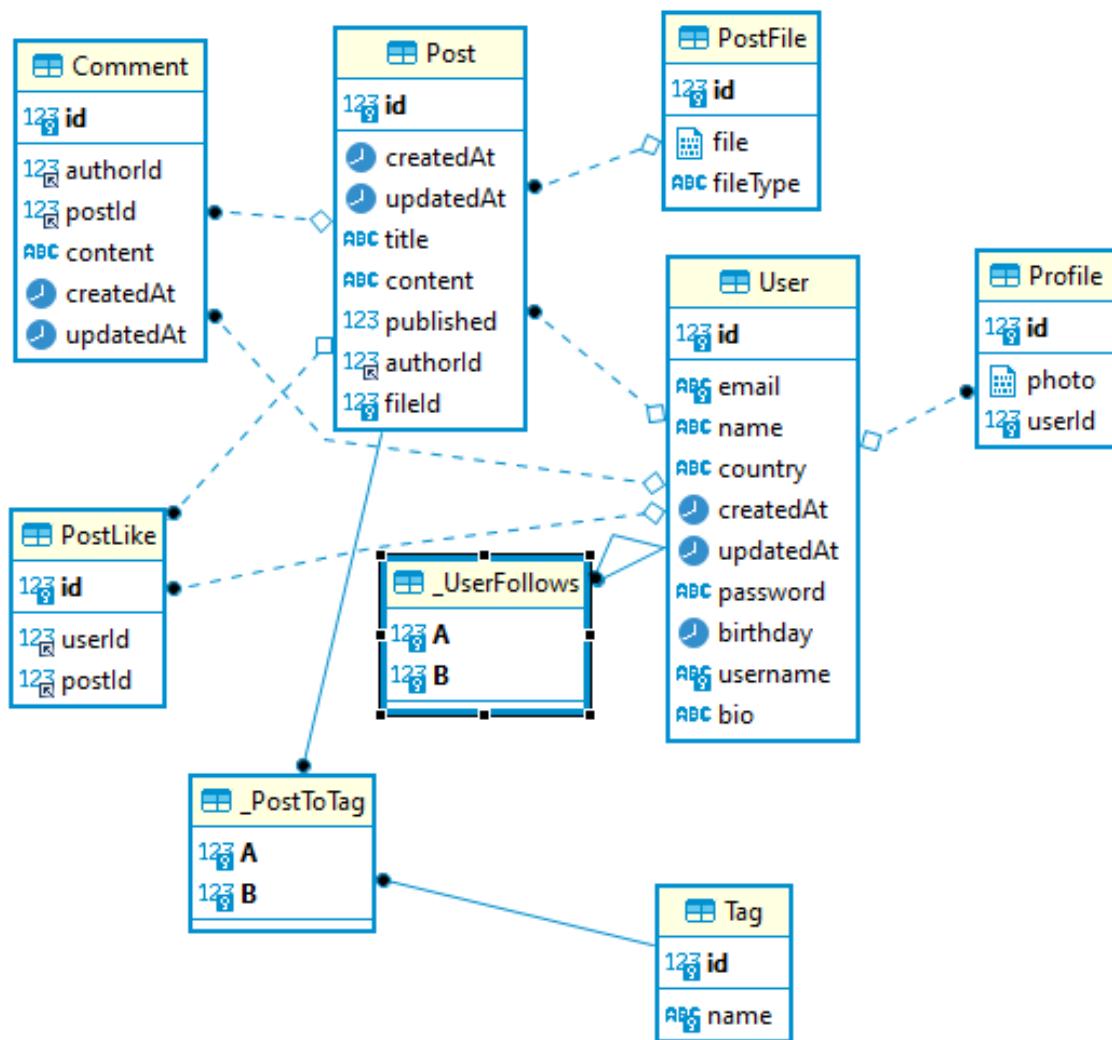


Figura 12 - Diagrama das relações das tabelas, geradas pelo Prisma

RELATÓRIO DO PROJETO

O que é Vue? E porquê?



Figura 13 - Logotipo Vue

Vue é uma framework de JavaScript, usada no desenvolvimento de interfaces web (frontend), que ao longo dos anos se tem tornado cada vez mais influente. Outras frameworks conhecidas de Javascript são por exemplo React, Angular e Jquery.

Neste projeto escolhi trabalhar em Vue como framework uma vez que o meu conhecimento de desenvolvimento web não é muito vasto, permitindo-me aprender mais e tornando o projeto mais desafiante e enriquecedor.

Uma das características que mais me agradou no Vue foi a sua estrutura. Aplicações que utilizam Vue são criadas a partir de componentes (.Vue). Estes componentes permitem trabalhar HTML, CSS e JavaScript num único ficheiro e podem ser reutilizados em várias páginas, evitando redundância no código. Vue tem componentes ou views, as views são páginas estáticas por assim dizer, não reutilizáveis, enquanto que os componentes podem ser utilizados em várias páginas.

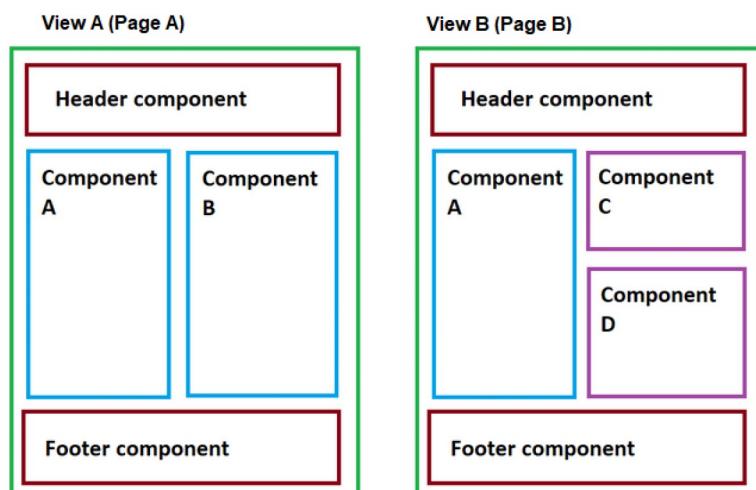


Figura 14 - Estrutura de uma página utilizando a lógica do Vue

RELATÓRIO DO PROJETO

Estrutura do projeto

Na minha opinião, uma das características mais importantes de um projeto (de código aberto ou não) é sua estrutura e organização. Neste projeto tentei ser o máximo organizado e tentar separar as coisas de modo a que uma pessoa que olhe para o código e queira contribuir ou só aprender, o consiga fazer sem dificuldade (ou com pouca).

Como já referido, o projeto divide-se em 2 partes, o backend e o frontend. A estrutura do projeto é simples:

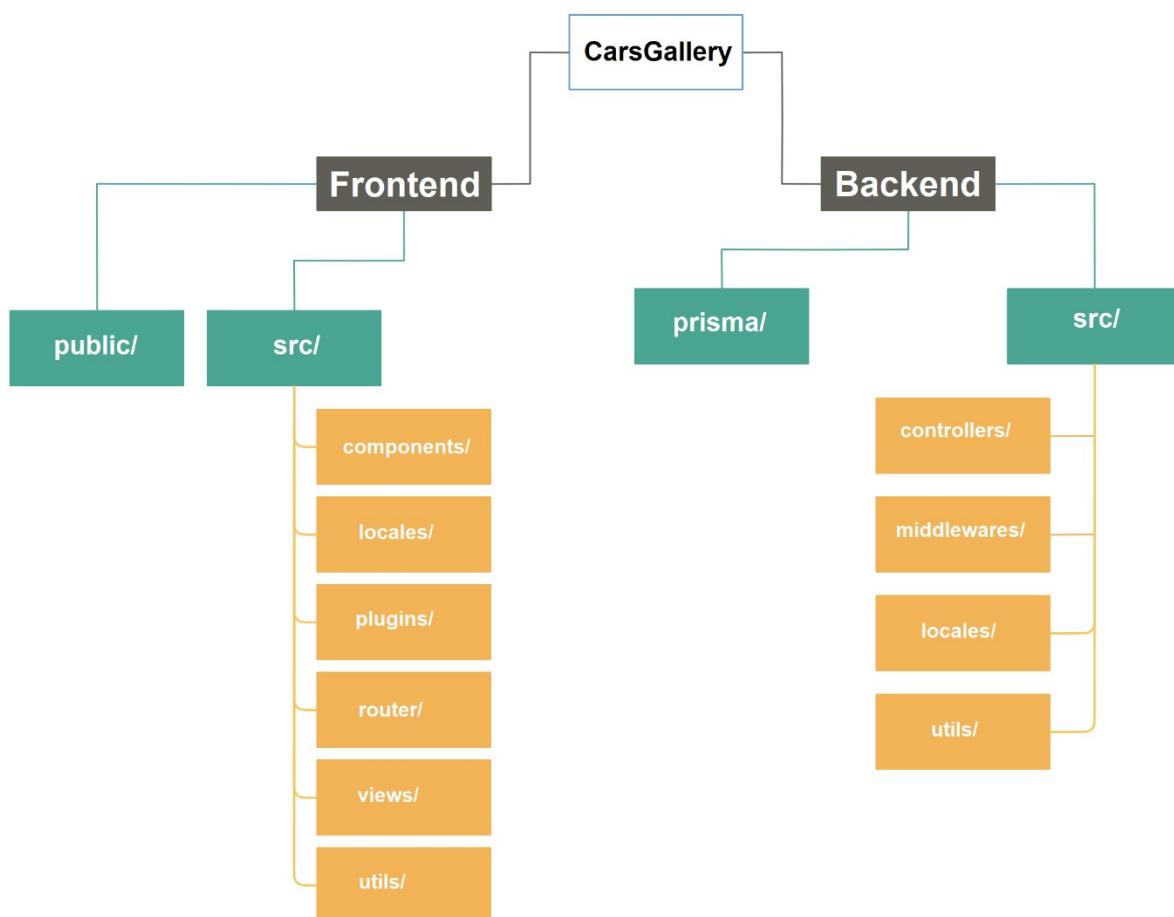


Figura 15 - Estrutura do projeto (frontend e backend)

RELATÓRIO DO PROJETO

Frontend:

- public/ - Conteúdo estático do site (imagens)
- src/ - Código do site
 - components/ - Componentes Vue do site
 - locales/ - Ficheiros com traduções (PT/ENG)
 - plugins/ - Ferramentas para integrar com o Vue
 - router/ - Rotas do frontend
 - views/ - Páginas
 - utils – Utilidades pouco usadas

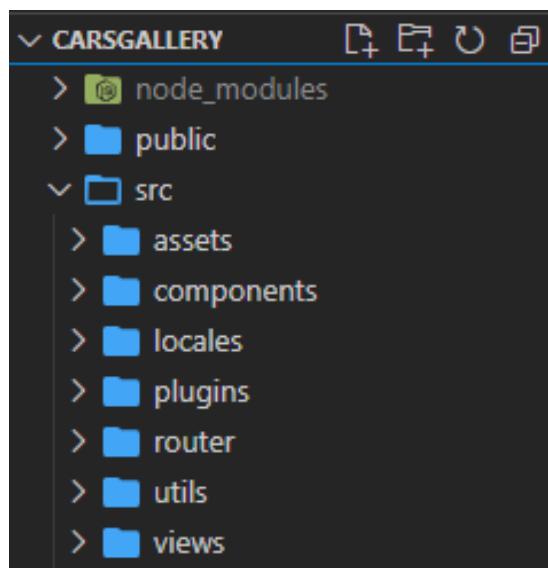


Figura 16 - Estrutura frontend

Backend:

- prisma/ - Ficheiros relacionados à base de dados

RELATÓRIO DO PROJETO

- src/ - Código do backend
 - controllers/ - Componentes Vue do site
 - locales/ - Ficheiros com traduções (PT/ENG)
 - middlewares/ - Camada entre as rotas
 - utils/ – Utilidades

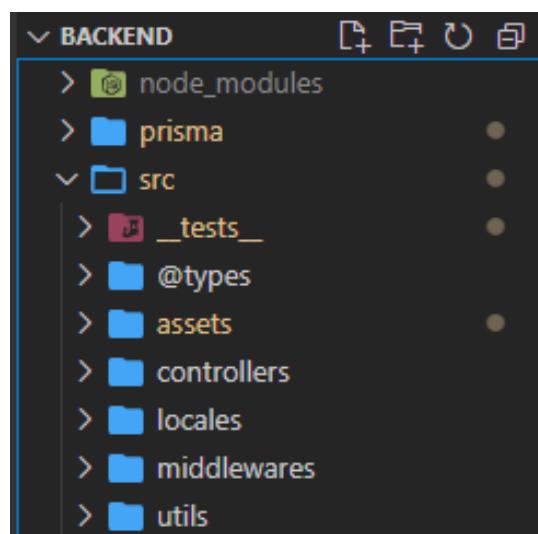


Figura 17 - Estrutura backend

Nota: Tanto no frontend como no backend, existe uma pasta chamada “node_modules” e um ficheiro package.json, a pasta e o ficheiro contêm todas as dependências usadas no projeto. (usando o gerenciador de pacotes NPM)



RELATÓRIO DO PROJETO

Ficheiro env

Em todos os projetos de web, há sempre um ficheiro “env”. Esse ficheiro é basicamente um “ficheiro secreto” com algumas variáveis que não devem ser expostas para o público, ou podemos também usar para criação de variáveis estáticas globais. Este ficheiro pode servir para, por exemplo, definir os dados de uma base de dados.

No ficheiro env do frontend as variáveis que defini foram as seguintes:

- VUE_APP_I18N_LOCALE – linguagem padrão do plugin i18n (para as traduções)
- VUE_APP_I18N_FALLBACK_LOCALE – caso a variável de cima esteja vazia, linguagem padrão será esta
- VUE_APP_API_URL – link do nosso backend (web api)

```
.ENV .env X  
.ENV .env  
1 VUE_APP_I18N_LOCALE=en  
2 VUE_APP_I18N_FALLBACK_LOCALE=en  
3 VUE_APP_API_URL="http://localhost:3000"  
4 |
```

Figura 18 - Ficheiro env do frontend

Já no ficheiro env do backend as variáveis que defini foram as seguintes:

- DATABASE_URL – link da base de dados com os dados, que servirá para o prisma
- PORT – porta do backend
- SECRET – um tipo de “password” para ser usada em rotas com dados sensíveis

```
.ENV .env X  
.ENV .env  
1 DATABASE_URL="mysql://root:123@127.0.0.1:3306/carsgallery"  
2 PORT=3000  
3 SECRET=carsgallerysecret|
```

Figura 19 - Ficheiro env do backend



RELATÓRIO DO PROJETO

Outros ficheiros

Existem outros ficheiros “soltos” no projeto, irei explicar agora cada um deles.

Backend:

- config.ts – Define variáveis globais (buscando-as ao ficheiro env)
- database.ts – Define e exporta o objeto “prisma” para podermos usa-lo
- index.ts – Ficheiro base do backend que define todas as dependências que irão ser utilizadas
- routes.ts – Todas as rotas do backend
- server.ts – Ficheiro que é executado quando o backend é corrido

Frontend:

- App.vue – Ficheiro “principal” que controla a execução do site
- main.js – Importação dos componentes que iremos utilizar com o Vue
- vue.config.js – Configuração de alguns plugins do Vue
- index.html – Definição de todas as bibliotecas CSS para o site

RELATÓRIO DO PROJETO

Como é que o frontend comunica com o backend?

O processo de comunicação é simples: o backend terá várias rotas, logo, para guardarmos ou requisitarmos dados, o frontend fará essas tais “chamadas” ao backend pela respectiva rota. Para isso é usada uma ferramenta do Vue, o Vuex, que inclui as “Actions”, que servem para fazer as chamadas assíncronas (ao backend), e as “Mutations”, para atualizar os dados do “State”, onde armazenamos os dados (ver tópico seguinte).

As Actions fazem as chamadas à API utilizando o “Axios”, que é uma biblioteca HTTP que serve como “ponte” entre o frontend e o backend, ou seja, que interliga os dois.

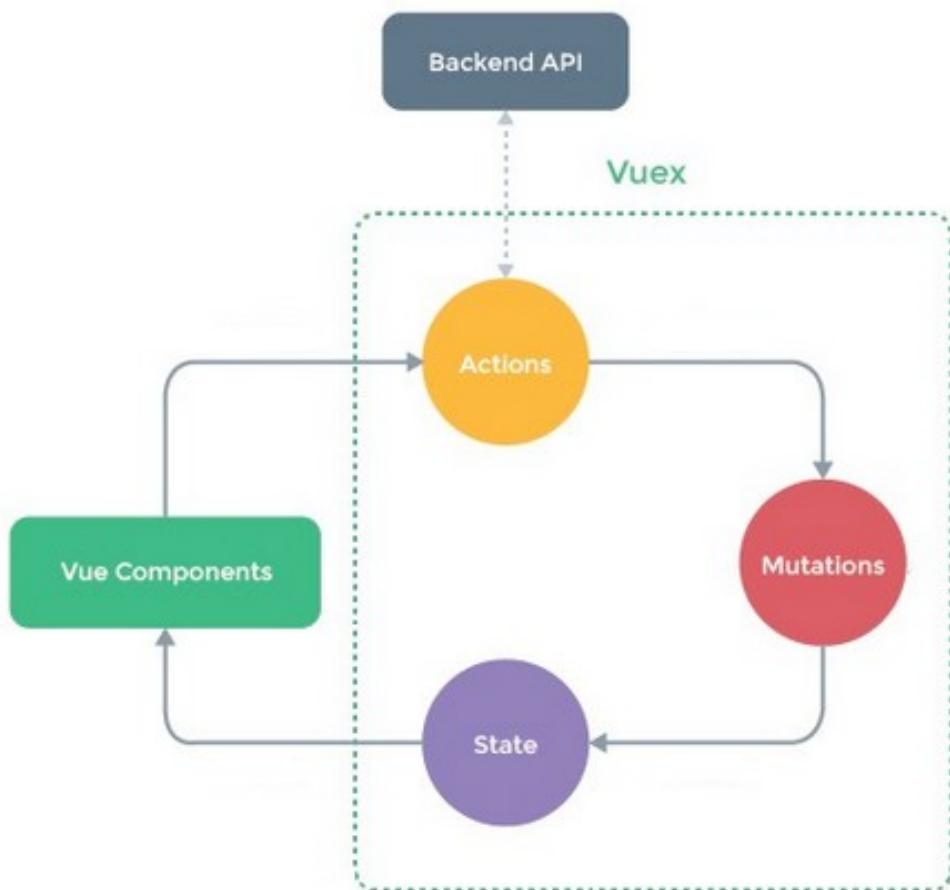


Figura 20 - Comunicação entre backend e frontend pelo Vuex



RELATÓRIO DO PROJETO

Exemplo: Quando um utilizador se regista, o frontend envia os dados do registo pela rota “/register”.

New Account

Name
teste

Username
teste

Email
@ teste@gmail.com

Birthday date
2020-05-13

Country
Portugal

Password

REGISTER

Figura 21 - Página register frontend

```
JSON
birthday: "2020-05-13"
country: "PT"
email: "teste@gmail.com"
name: "teste"
password: "testetestete"
username: "teste"
```

Figura 22 - Dados enviados para a rota

RELATÓRIO DO PROJETO

Vuex - Gerenciamento do estado em Vue

O Vue tem uma ferramenta muito útil já referida anteriormente, chamada Vuex, que, resumindo, guarda o estado de alguns dados importantes, e comunica com o backend.

Basicamente, poderíamos caracterizá-la como um “armazém” com variáveis/funções globais, onde podemos pedir dados ao backend e guardá-los e usá-los em vários componentes.

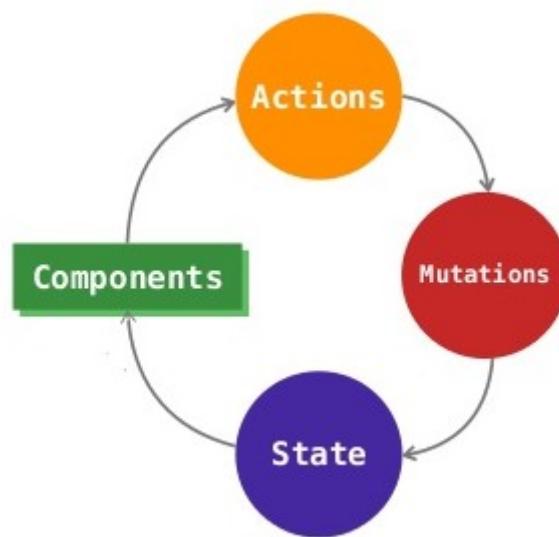


Figura 23 - Gerenciamento do estado pelo Vuex

Para atualizarmos dados, usamos as “Actions” que fazem as chamadas ao backend, para salvar esses dados, chamamos uma “Mutation” que os atualiza/armazena, e assim esses dados tornam-se utilizáveis em Componentes/Views.

Exemplo 1: Um bom exemplo para explicar isto, é a animação “loading” quando o frontend aguarda uma resposta do backend (ou seja, enquanto os dados estão a ser processados e a página está a carregar). Sempre que faço uma chamada assíncrona (pelas actions), meto a variável loadingStatus do state como verdadeira, e no componente uso um componente do Vue Loading que só irá ser visível na página se esse loadingStatus do state for verdadeiro. Como os componentes do Vue são reativos, assim que recebemos resposta e metermos o loadingStatus de novo a falso, a animação para de ser visível.

Exemplo 2: Quando o utilizador faz login, o frontend tem que esperar para que o backend verifique se os dados estão válidos, logo, assim que a chamada é feita, meto o loading a verdadeiro, e assim que receber a resposta, meto-o de novo a falso.



RELATÓRIO DO PROJETO

```
actions: {
  login({ commit }, user) {
    commit("SET_LOADING_STATUS", true) ←

    return api
      .post("/login", user)
      .then((response) => {
        commit("SET_LOADING_STATUS", false) ←
        commit("SET_USER", response.data.user)
      })
  }
}
```

Figura 24 - Action "login" no vuex.js

```
state: {
  user: null,
  token: localStorage.getItem("token") || "",
  loadingStatus: false, ←
},
```

Figura 25 - Variáveis do "state" no vuex.js

```
mutations: [
  SET_LOADING_STATUS(state, status) {
    state.loadingStatus = status
  },
]
```

Figura 26 - Mutation "SET_LOADING_STATUS" no vuex.js

```
<Loading
  :active.sync="loadingStatus"
  color="#3e3fa6"
  :opacity="0.1"
  blur
  :can-cancel="false"
  :is-full-page="true"
/>
```

Figura 27 - Componente Loading no "Login.vue"

"active.sync" é a propriedade que define se o componente é mostrado ou não, se loadingStatus for falso, o componente não é mostrado.

RELATÓRIO DO PROJETO

Rotas backend

O backend é responsável pela “lógica” e pelo armazenamento/processamento dos dados. Para isso acontecer, são definidas várias rotas às quais podemos enviar ou pedir dados. Há então vários tipos de rotas: GET, POST, PUT e DELETE.

As rotas GET são as rotas que devolvem dados.

As rotas POST armazenam dados.

As rotas PUT atualizam os dados.

As rotas DELETE como o nome indica, apagam dados.



Figura 28 - Rotas GET PUT POST e DELETE

RELATÓRIO DO PROJETO

O backend tem várias rotas, estas rotas estão todas definidas no ficheiro routes.ts:

1. Rotas para utilizadores:

- GET:
 - /users – Retorna todos os utilizadores
 - /user/:id - Retorna os dados de um utilizador (pelo seu id)
 - /user/:id/avatar - Retorna o avatar de um utilizador (pelo seu id)
 - /authenticate - Retorna os dados de um utilizador pelo token
 - /user/:id/posts – Retorna os posts de um utilizador (pelo seu id)
- POST:
 - /register - Cria um utilizador
 - /login - Verifica se utilizador existe
 - /avatar – Atualiza o avatar do utilizador
- PUT:
 - /user - Atualizar dados do utilizador (por exemplo, mudar a password)
- DELETE:
 - /user - Apagar a conta do utilizador

```

37  // User routes
38  router.get('/users', UserController.index)
39  router.get('/user/:id', UserController.getUser)
40  router.get('/user/:id/avatar', UserController.getAvatar)
41  router.get('/authenticate', isAuthenticated, UserController.authenticate)
42  router.get('/user/:id/posts', UserController.getUserPosts)
43  router.post('/register', UserController.register)
44  router.post('/login', UserController.login)
45  router.post(
46    '/avatar',
47    isAuthenticated,
48    upload.single('avatar'),
49    UserController.uploadProfilePicture
50  )
51  router.put('/user', isAuthenticated, UserController.updateProfile)
52  router.delete('/user', isAuthenticated, UserController.deleteUser)
53

```

Figura 29 - Rotas utilizadores

RELATÓRIO DO PROJETO

2. Rotas para posts:

- GET:
 - /posts – Retorna todos os posts
 - /file/:id – Retorna o ficheiro associado ao post (pelo seu id)
 - /post/:id – Retorna o conteúdo do post (pelo seu id)
 - /post/:id/like – Retorna se o utilizador tem like no post (pelo seu id)
- POST:
 - /posts – Criar post
- PUT:
 - /post/:id/like – Dar like num post (pelo seu id)
 - /post/:id/file – Associar um ficheiro a um post (pelo seu id)
 - /post/:id – Atualizar post
- DELETE:
 - /post/:id/like – Tirar o like de um post (pelo seu id)
 - /post/:id – Eliminar o post (pelo seu id)

RELATÓRIO DO PROJETO

```

54 // Post routes
55 router.get('/posts', optionalAuthenticated, PostController.index)
56 router.get('/file/:id', PostController.getFile)
57 router.get('/post/:id', isAuthenticated, PostController.getPost)
58 router.get('/post/:id/like', isAuthenticated, PostController.userLikedPost)
59 router.post('/posts', isAuthenticated, PostController.create)
60 router.put('/post/:id/like', isAuthenticated, PostController.likePost)
61 router.put(
62   '/post/:id/file',
63   isAuthenticated,
64   upload.single('file'),
65   PostController.uploadFile
66 )
67 router.put('/post/:id', isAuthenticated, PostController.updatePost)
68 router.delete('/post/:id/like', isAuthenticated, PostController.unlikePost)
69 router.delete('/post/:id', isAuthenticated, PostController.deletePost)
70

```

Figura 30 - Rotas posts

3. Rotas para Tags:

- GET:
 - /tags – Retorna todos os posts com certas tags

```

71 // Tag routes
72 router.get('/tags', TagController.index)
73

```

Figura 31 - Rotas tags

4. Rotas para seguidores:

- GET:
 - /user/:id/following – Retorna os utilizadores que o utilizador segue (pelo seu id)
 - /user/:id/followedBy – Retorna os seguidores do utilizador (pelo seu id)
 - /user/:id/isFollowing – Retorna se o utilizador segue outro utilizador
- POST:

RELATÓRIO DO PROJETO

- /follow/:id – Seguir um utilizador (pelo seu id)
- DELETE:
 - /unfollow/:id – deixar de seguir um utilizador

```
73 // Follow routes
74 router.get('/user/:id/following', FollowController.getUserFollowing)
75 router.get('/user/:id/followedBy', FollowController.getUserFollowers)
76 router.get(
77   '/user/:id/isFollowing',
78   isAuthenticated,
79   FollowController.checkIfFollowing
80 )
81 router.post('/follow/:id', isAuthenticated, FollowController.follow)
82 router.delete('/unfollow/:id', isAuthenticated, FollowController.unfollow)
83
84
```

Figura 32 - Rotas seguidores

5. Rotas para comentários:

- GET:
 - /post/:id/comments – Retorna os comentários de certo post (pelo seu id)
- POST:
 - /post/:id/comments – Comentar num post (pelo seu id)
- PUT:
 - /comments/:id – Atualizar comment (pelo id do comentário)
- DELETE:
 - /comments/:id – Apagar comentário (pelo id do comentário)

RELATÓRIO DO PROJETO

```
85 // Comment routes
86 router.get('/post/:id/comments', CommentController.getPostComments)
87 router.post(
88   '/post/:id/comments',
89   isAuthenticated,
90   CommentController.commentOnPost
91 )
92 router.put('/comments/:id', isAuthenticated, CommentController.updateComment)
93 router.delete('/comments/:id', isAuthenticated, CommentController.deleteComment)
```

Figura 33 - Rotas comentários

Os dois pontos antes de uma variável na rota (por exemplo “:id”), significam que a rota espera que seja feita uma chamada, e que um id passe na rota (como argumento), caso contrário, os dados não serão encontrados. As rotas do backend estão todas definidas no ficheiro routes.ts

Exemplo: Ao aceder a um perfil de outro utilizador, é usada a rota GET /user/:id que pode ser aplicada da seguinte maneira: GET /user/4 para que sejam devolvidos os dados do utilizador que tem o id 4 na base de dados.

RELATÓRIO DO PROJETO

Rotas frontend

Como já foi referido anteriormente, o frontend é responsável pela parte visual do site. As rotas do frontend tem então a utilidade de redirecionar o utilizador para uma determinada página e exibir o seu conteúdo. Assim, em Vue, cada rota tem um “componente” associado.

```
19 const routes = [
20   { path: "*", component: NotFound },
21   { path: "/", component: Home },
22   { path: "/home", component: Home },
23   { path: "/login", component: Login },
24   { path: "/register", component: Register },
25   { path: "/logout", component: Logout },
26   {
27     path: "/profile/:id?",
28     component: Profile,
29   },
30   { path: "/profilesettings", component: ProfileSettings },
31   { path: "/uploadpost", component: UploadPost },
32   { path: "/tagsearch/:tags", component: TagSearch},
33 ]
34 ]
```

Figura 34 - Rotas do frontend

- * - Rota para páginas não existentes
- / - Página inicial
- /home – Página inicial
- /login – Página do login
- /register – Página do register
- /logout – Sair da conta
- /profile/:id? – Ir para o perfil de um certo utilizador (pelo seu id)
- /profilesettings – Ir para as definições do utilizador
- /uploadpost – Carregar novo post
- /tagsearch/:tags – Procurar posts com certas tags

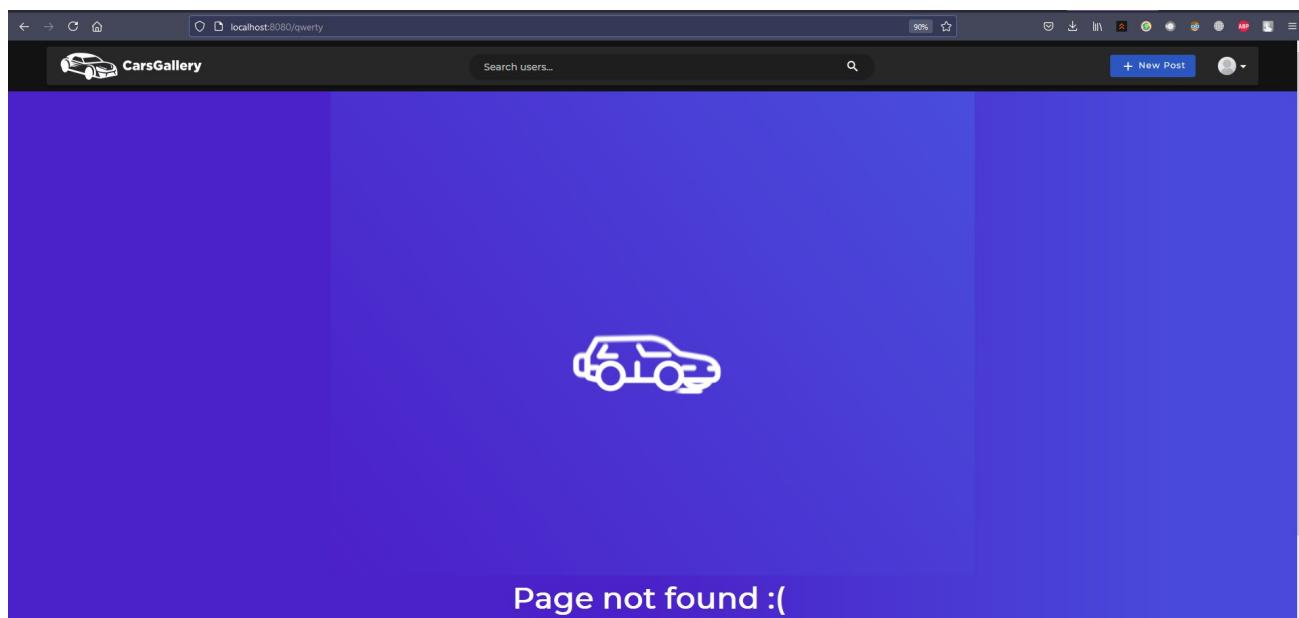


RELATÓRIO DO PROJETO

Rotas inexistentes

E se por acaso o utilizador tentar aceder a rotas não definidas? O que acontecerá?

Simples, no backend simplesmente não existe resposta, já no frontend somos redirecionados para uma página especial. Vou experimentar e tentar aceder à rota “/qwerty”.



O site por si indica que esta página não existe.

RELATÓRIO DO PROJETO

Autenticação Bearer

Para evitar problemas, maior parte das rotas necessita de um tipo especial de autenticação, esse tipo é o “Bearer”, basicamente temos que enviar o nosso token para rota privada com um header: "Authorization": "Bearer <token>" - e a API verifica o header "Authorization" - Caso não tenha um token dá erro, se tiver guarda a token, descriptografa a token e usa os dados do utilizador.

```

181  deleteUser({ commit }) {
182    // delete /user (+token)
183    commit("SET_LOADING_STATUS", true);
184    api
185      .delete("/user", {
186        headers: {
187          Authorization: "Bearer " + localStorage.getItem("token"),
188        },
189      })
190      .then((response) => {
191        commit("SET_LOADING_STATUS", false);
192        Vue.swal.fire(i18n.t("success"), response.data.message, "success");
193        router.push("/home");
194      })
195      .catch((error) => {
196        commit("SET_LOADING_STATUS", false);
197        setTimeout(() => {
198          handleError(error);
199        }, 900);
200      });
201  },
202

```

Figura 35 - Exemplo (função que apaga um utilizador no frontend vuex.js)

```

52 |
53   router.delete("/user", isAuthenticated, UserController.deleteUser)
54

```

Figura 36 - Rota no backend (routes.ts)

Podemos ver que a rota usa um middleware, que é a função “isAuthenticated”. Esse middleware é uma função que irá verificar se o token é valido, e se for, executa a função do 3º parâmetro (UserController.deleteUser neste caso).



RELATÓRIO DO PROJETO

Cofinanciado por:



UNIÃO EUROPEIA
Fundo Social Europeu

RELATÓRIO DO PROJETO

Middlewares

Os middlewares são nada mais nada menos que funções que são executadas antes da “callback” da rota, isto é útil em situações como, por exemplo, um utilizador querer apagar a sua conta, para termos a certeza que é o utilizador a fazer esta chamada à API, usamos o método já referido anteriormente “Bearer”, que passando um token, o middleware “isAuthenticated” irá verificar se o token é valido e é mesmo o utilizador a fazer essa chamada.

Middlewares que usei no projeto:

- isAuthenticated – verifica se o token do utilizador é valido
- generateToken (auth.js) – cria um token (JSON Web Token)
- checkLocale (locale.ts) – funcionalidade do i18n, que irá verificar a linguagem que devemos retornar as mensagens
- optionalAuthenticated – middleware optional usado para algumas rotas

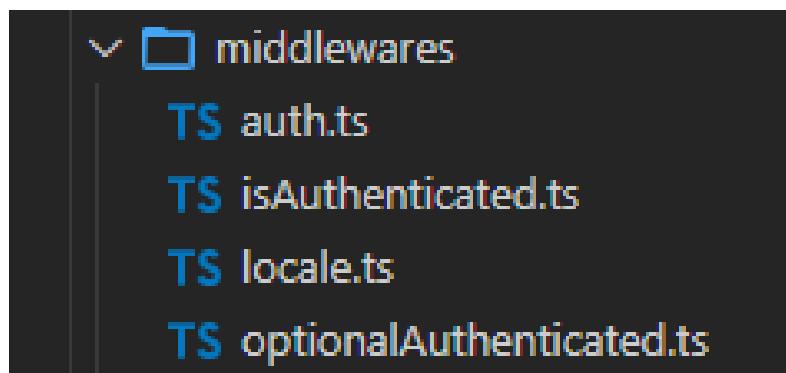


Figura 37 - Ficheiros com os middlewares

RELATÓRIO DO PROJETO

Como funciona o sistema de autenticação?

No sistema de autenticação são usadas várias ferramentas/tecnologias que garantem segurança ao utilizador. Alguns exemplos são, Bcrypt, usado para criptografar as passwords e JWT (Json Web Tokens), um token que armazenará os dados únicos do utilizador (criptografados) permitindo a validação do utilizador no site. Este token é guardado no localStorage e terá uma data de validade. O localStorage consiste num sistema local no browser onde se podem guardar os dados. O token é guardado aqui uma vez que localStorage não é limpo depois de o browser ser fechado (o Vuex é limpo por exemplo). Assim, quando o utilizador fecha e volta a abrir o navegador, se o token ainda for válido, é autenticado automaticamente.

Exemplo: Quando o utilizador se tenta loggar pelo browser, o frontend envia uma chamada ao backend transmitindo os dados introduzidos (e-mail e password). De seguida, o backend irá validar estes dados segundo a base de dados. Se os dados forem válidos, são devolvidos todos os dados do utilizador da base de dados e um token temporário, caso sejam inválidos, retorna erro.

Resumindo:



Figura 38 - Ações que ocorrem durante o login

RELATÓRIO DO PROJETO

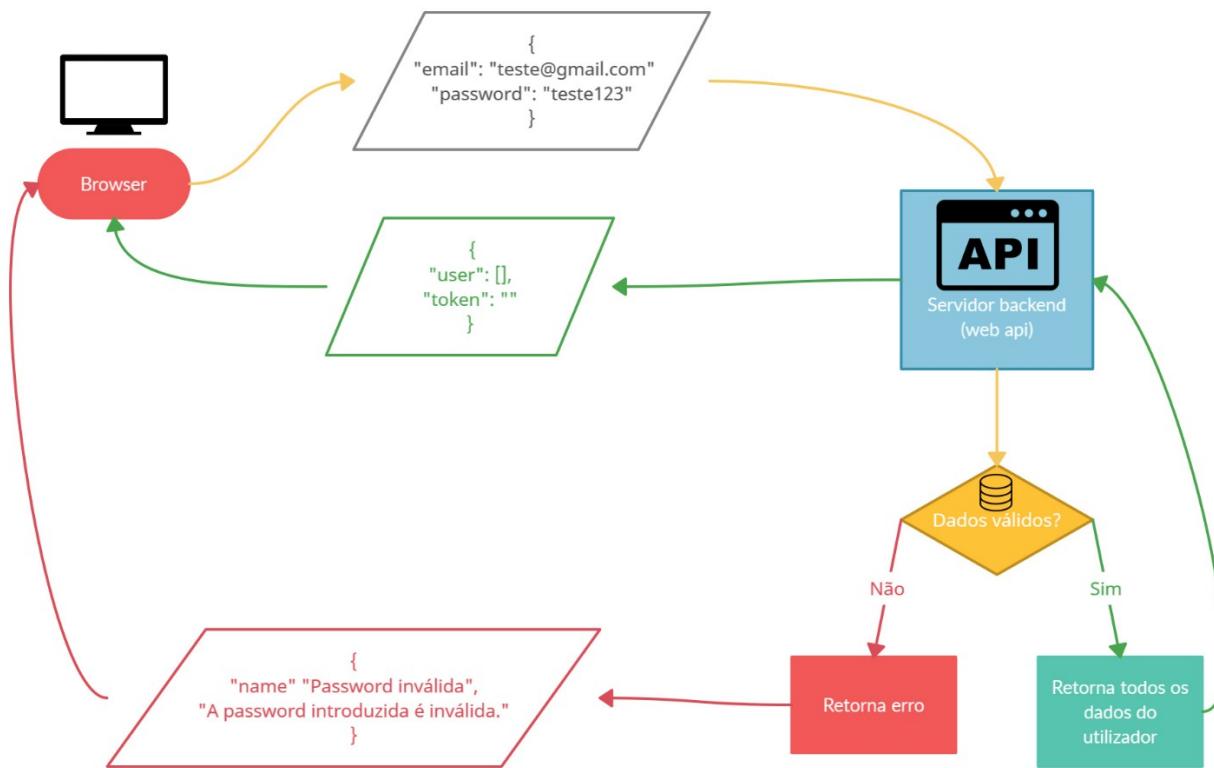


Figura 39 - Fluxograma descrevendo um login

POST ▾ http://localhost:3000/login		Send	200 OK	86.7 ms	503 B	Just Now ▾	
JSON ▾	Auth ▾	Query	Header 1	Preview ▾	Header 17	Cookie	Timeline
<pre> 1 <pre>{ 2 "email": "roby@gmail.com", 3 "password": "robyroby" 4 }</pre> </pre> <pre> 1 <pre>{ 2 "user": { 3 "id": 12, 4 "email": "roby@gmail.com", 5 "password": "\$2b\$10\$KUDrKkt4wsY0JwubksANF.viG0mXJligWBiQ2o2xWodAU/6trzvMS", 6 "name": "roby", 7 "username": "roby", 8 "birthday": "2020-02-06T00:00:00.000Z", 9 "country": "PT", 10 "bio": null, 11 "createdAt": "2021-06-22T01:57:26.380Z", 12 "updatedAt": "2021-06-22T01:57:26.394Z", 13 "following": 0, 14 "followedBy": 0 15 }, 16 "token": 17 "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIsImVtYWlsIjoi cm9ieUBnbWFpbC5jb20iLCJpYXQiOjE2MjQ2NTU4Mjg5ImV4cCI6MTYyNTI2MDYyOH0.ZcmY1My1u_0bWEVixz5nnfLSVtKnLMjsWeok4Y9FmJw" 18 }</pre> </pre>							

Figura 40 - Chamada ao backend (api) pela rota login, enviando os dados, e a resposta obtida



RELATÓRIO DO PROJETO

Como podemos ver na imagem acima, se os dados forem válidos, o servidor retorna os nossos dados da base de dados e o tal token. Já na imagem abaixo, caso os dados estiverem incorretos, o servidor retorna erro.

The screenshot shows a POST request to `http://localhost:3000/login`. The JSON payload is:

```
1 {  
2   "email": "admin@admin.com",  
3   "password": "123"  
4 }
```

The response status is **401 Unauthorized**, with a response time of 88.9 ms and a body size of 58 B. The response body is:

```
1 {  
2   "message": "Password is wrong",  
3   "error": "Invalid password"  
4 }
```

Figura 41 - Resposta caso os dados não sejam válidos

E aqui tão os “pop-ups” que aparecem na tela depois do login, indicando sucesso ou erro.

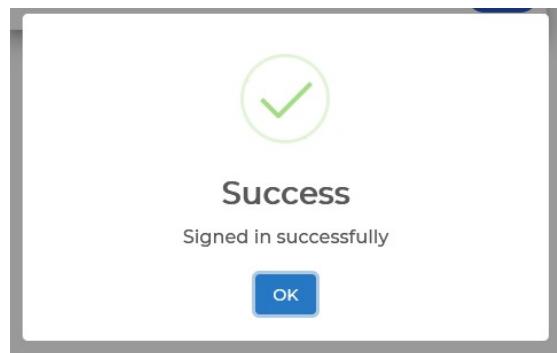


Figura 42 - Popup sucesso (dados válidos)

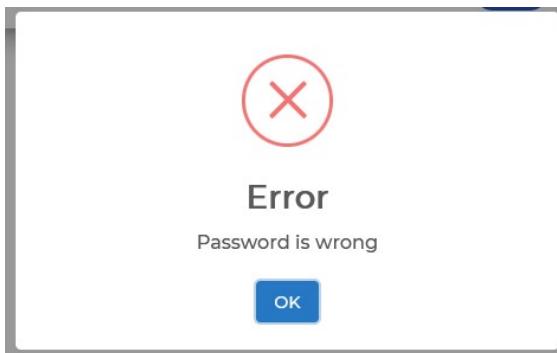


Figura 43 - Popup erro (dados inválidos)

RELATÓRIO DO PROJETO

Temas

Uma configuração que muitos sites têm hoje em dia é o tema, que é basicamente uma ferramenta visual do site que o utilizador pode escolher à sua preferência, normalmente existem dois tipos de tema, o “Light Mode”, que é um tema claro, e depois o “Dark Mode” que é mais focado em cores escuras.

A implementação desta ferramenta foi relativamente fácil graças ao Vuetify, pois ele muda as cores automaticamente dependendo do tema, a única coisa que eu fiz foi guardar o tema no Vuetify, localStorage e no “State” do Vuex, e ter uma função que retorna se o tema está em Light Mode ou Dark Mode”, pelo Vuetify, para as cores mudarem automaticamente.

```
state: {
  user: null,
  token: localStorage.getItem("token") || "",
  darkTheme: localStorage.getItem("theme") || false, // 1 = dark, 0 = light
  loadingStatus: false,
},
```

Figura 44 - vuex.js

```
isDarkTheme() {
  return this.$vuetify.theme.dark;
},
```

Figura 45 - Comments.vue

```
<div
  :class="[
    $vuetify.breakpoint.width < 600
    ? 'commentDivMobile'
    : 'commentDivDesktop',
    isDarkTheme ? 'commentsDarkmode' : 'commentsLightmode',
  ]"
>
  {{ c.content }}
</div>
```

Figura 46 - Comments.vue

Por exemplo, neste código, estou a associar uma “class CSS” a uma “div” dependendo se o tema do utilizador está em Light Mode ou Dark Mode.

O utilizador pode mudar o seu tema nas configurações gerais.



RELATÓRIO DO PROJETO

The screenshot shows a user interface for profile settings. On the left, there is a sidebar with three items: 'DEFINIÇÕES GERAIS' (selected), 'DEFINIÇÕES PERFIL', and 'DEFINIÇÕES CONTA'. The main area has two dropdown menus: 'Línguagem' set to 'Português' and 'Tema' set to 'Claro'. A red arrow points to the lightbulb icon next to the 'Claro' button.

Figura 47 - ProfileSettings.vue

Ao clicarmos neste botão (lâmpada), mudamos o tema pelo Vuetify, e também no “state” do Vuex e no localStorage.

```
toggleTheme() {
  this.$vuetify.theme.dark = !this.$vuetify.theme.dark;
  this.$store.commit('SET_DARK_THEME', this.$vuetify.theme.dark);
},
```

Figura 48 - ProfileSettings.vue

```
SET_DARK_THEME(state, theme) {
  state.darkTheme = theme; // 1 = darkmode, 0 = lightmode
  localStorage.setItem("theme", theme);
},
```

Figura 49 - Mutation "SET_DARK_THEME" em vuex.js

RELATÓRIO DO PROJETO

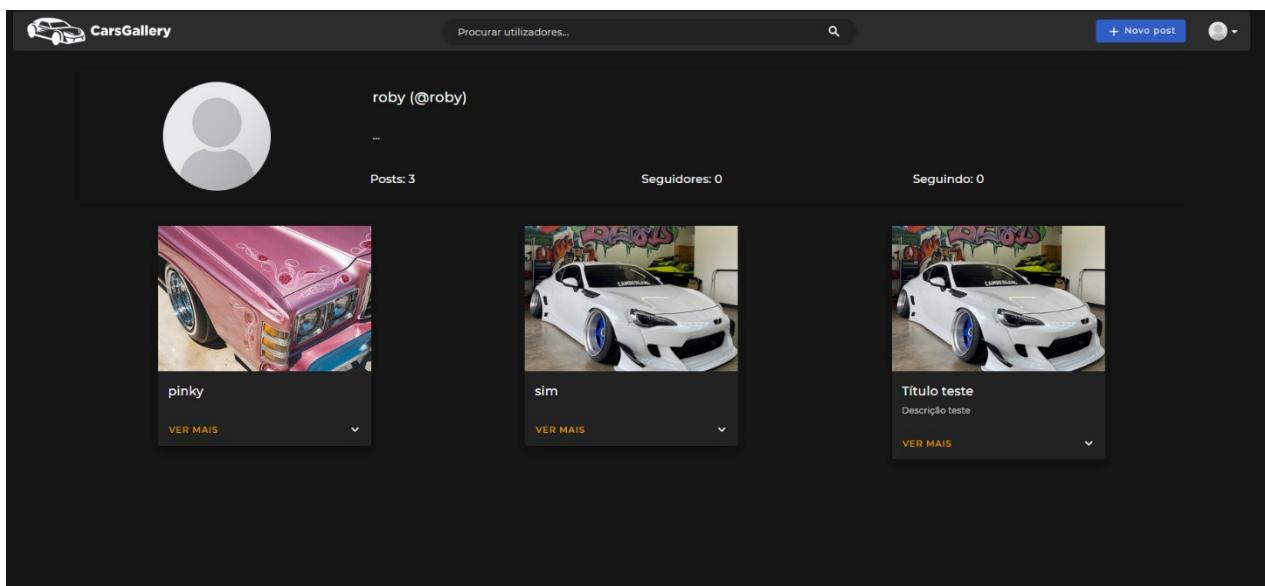


Figura 50 - Exemplo Dark Mode (tema escuro) no Profile.vue

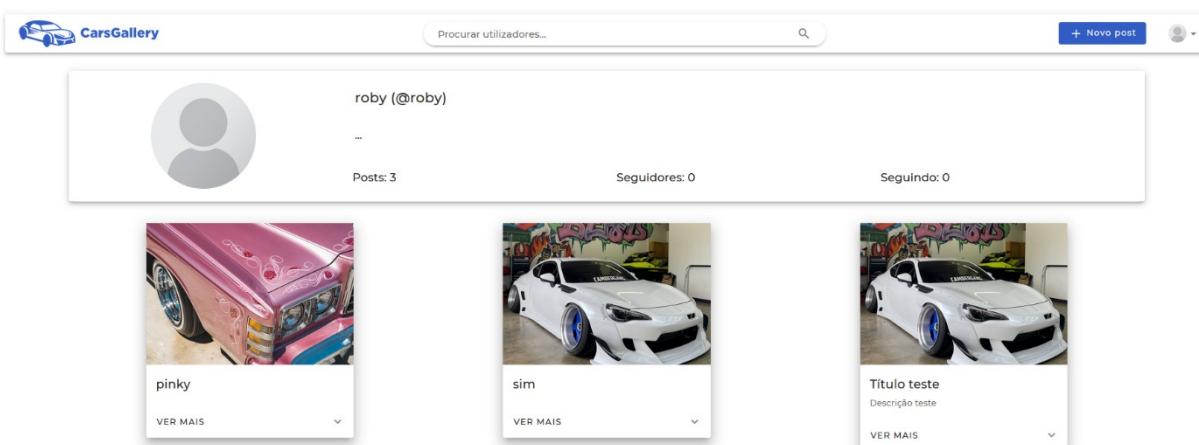


Figura 51 - Exemplo Dark Mode (tema claro) no Profile.vue



RELATÓRIO DO PROJETO

Traduções

Quando decidi que ia dar disponibilizar este projeto publicamente (código aberto) quis que pudesse ser usado por pessoas de outros países. Por isso, achei importante adicionar pelo menos suporte para Inglês. A implementação foi relativamente fácil, foi utilizada uma ferramenta chamada i18n, tanto para o backend e frontend. A linguagem escolhida pelo utilizador é armazenada em localStorage, assim, quando o site carrega, o i18n traduz os componentes dependendo do localStorage. A linguagem padrão é inglês, mas o utilizador pode alterar a linguagem nas definições gerais.

As traduções estão armazenadas em 2 ficheiros JSON (pt.json e en.json) e cada “chave” tem a sua tradução. Depois no componente, eu chamo essa tal chave, que o i18n traduz automaticamente. *Exemplo da utilização do i18n e das traduções, no frontend:*

```
"profile": {  
    "followers": "Seguidores",  
    "following": "Seguindo",  
    "posts": "Posts"  
},
```

Figura 52 - Chave "profile" em pt.json

```
"profile": {  
    "followers": "Followers",  
    "following": "Following",  
    "posts": "Posts"  
},
```

Figura 53 - Chave "profile" em en.json

```
<v-col  
| >{{ this.$t("profile.posts") }}:  
| {{ this.user.posts.length }}  
</v-col>  
<v-col  
| >{{ this.$t("profile.followers") }}:  
| {{ this.user.followedBy }}  
</v-col>  
<v-col  
| >{{ this.$t("profile.following") }}:  
| {{ this.user.following }}</v-col  
>
```

Figura 54 - Profile.vue



RELATÓRIO DO PROJETO

No backend o i18n é utilizado na mesma, porém, quando é feita uma chamada (GET, POST, PUT ou DELETE), é enviado como argumento a linguagem que o utilizador está a usar no frontend, para o backend saber em que linguagem devolver a mensagem de sucesso ou erro.

Aqui é definida a variável api, que irá usar o “axios” já referido anteriormente, e cria uma “configuração”, evitando assim que se repitam os parâmetros cada vez que fazemos uma chamada à API. Esta configuração tem então a linguagem do utilizador (localStorage), por isso, sempre que for feita uma chamada ao backend pelo frontend, um dos parâmetros será a linguagem.

```
const api = axios.create({
  baseURL: "http://localhost:3000",
  params: {
    lng: localStorage.getItem("lang"),
  },
})
```

Figura 55 - Criação de uma constante com vários dados necessários a cada chamada

Também temos as traduções separadas em 2 ficheiros JSON json no backend, então a utilização acaba por ser da mesma maneira.

```
"email_notfound": {
  "name": "Email not found",
  "message": "The email does not exist"
},
```

Figura 56 - Chave "email_notfound" em en.json

```
"email_notfound": {
  "name": "Email não encontrado",
  "message": "O email introduzido não existe"
},
```

Figura 57 - Figura 39 - Chave "email_notfound" em pt.json

RELATÓRIO DO PROJETO

```
// se email for invalido, retorna erro
if (!userExists) {
    return new AppError(res, {
        name: req.t("user_controller.email_notfound.name"),
        message: req.t("user_controller.email_notfound.message"),
        statusCode: 404,
    })
}
```

Figura 58 - Função login, dentro de UserController.ts

Se tentarmos dar login com um email que não existe e tivermos o site em português, o backend retorna erro.

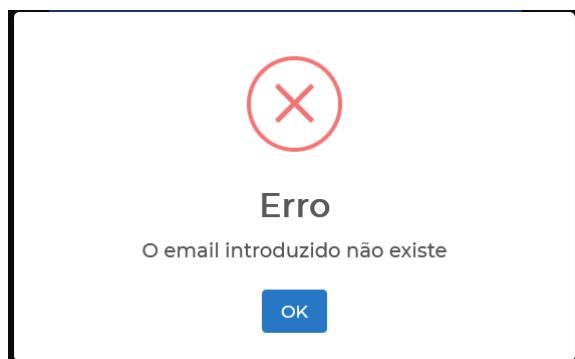


Figura 59 - Erro caso email seja inválido

E aqui está a rota com o respetivo parâmetro da linguagem atual do utilizador no site:

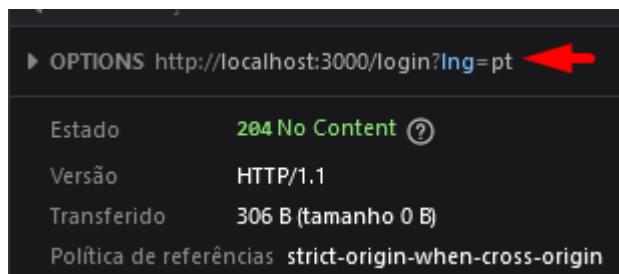


Figura 60 - Rota chamada

Caso o parâmetro não exista na rota, o backend vai assumir que a linguagem é inglês.



RELATÓRIO DO PROJETO

Criação de utilizador

Para o utilizador criar uma conta, utiliza-se a rota “/register”.

Exemplo: Quando um utilizador se regista, o frontend envia os dados do registo pela rota “/register”.

New Account

Name
 teste

Username
 teste

Email
 @ teste@gmail.com

Birthday date
 2020-05-13

Country
 Portugal

Password

REGISTER

Figura 61 - Campos a serem introduzidos para a criação de um novo utilizador

```
JSON
{
  "birthday": "2020-05-13",
  "country": "PT",
  "email": "teste@gmail.com",
  "name": "teste",
  "password": "testetestete",
  "username": "teste"
}
```

Figura 62 - Dados enviados para a rota

Os campos têm uma verificação para ter a certeza que os campos são introduzidos.

RELATÓRIO DO PROJETO



```

<!-- NAME -->
<v-text-field
  :label="$t('register.name')"
  :rules="[rules.required, rules.name]"
  @input="handleInput($event, 'name')"
  prepend-icon="mdi-account"
  type="text"
  color="reverse"
  :hint="$t('register.name_hint')"
>
</v-text-field>

```

Figura 63 - Utilização de "regras" para os campos, neste caso, para o nome

Na figura acima, podemos ver que para termos esta verificação, temos que ter uma “regra” para o campo. Com essa regra definida, o componente do Vuetify trata do resto sozinho.

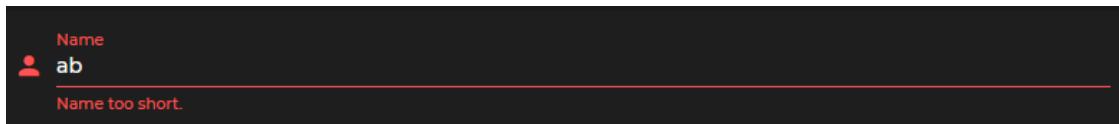


Figura 64 - Verificação se um nome é demasiado curto

O campo do email utiliza um padrão “regex” para termos a certeza que é um email valido (basicamente tem que ter um @gmail.com por exemplo)



Figura 65 - Verificação de um email inválido (sem @)

No campo dos países, aparecem todos os países do mundo, fiz isto com um simples ficheiro json que contém todos os nomes dos países e a sua sigla.

RELATÓRIO DO PROJETO



```

{
  "AF": "Afghanistan",
  "AX": "Åland Islands",
  "AL": "Albania",
  "DZ": "Algeria",
  "AS": "American Samoa",
  "AD": "Andorra",
  "AO": "Angola",
  "AI": "Anguilla",
  "AQ": "Antarctica",
  "AG": "Antigua and Barbuda",
  "AR": "Argentina",
  "AM": "Armenia",
  "AW": "Aruba",
  "AU": "Australia",
  "AT": "Austria",
  "AZ": "Azerbaijan",
  "BS": "Bahamas",
  "BH": "Bahrain",
  "BD": "Bangladesh",
  "BB": "Barbados",
  "BY": "Belarus",
  "BE": "Belgium",
  "BZ": "Belize",
  "BJ": "Benin",
  "BM": "Bermuda",
  "BT": "Bhutan",
  "BO": "Bolivia, Plurinational State of",
  "BQ": "Bonaire, Sint Eustatius and Saba",
  "BA": "Bosnia and Herzegovina",
  "BW": "Botswana",
  "BV": "Bouvet Island",
  "BR": "Brazil",
  "IO": "British Indian Ocean Territory",
  "BN": "Brunei Darussalam",
  "BG": "Bulgaria",
  "BF": "Burkina Faso",
  "BT": "Burundi"
}

```

Figura 66 - Ficheiro countries.json com todos os países

Quando a página é carregada, o que eu faço é, um ciclo pelo ficheiro todo e armazeno as siglas e os nomes em 2 arrays separados:



```

mounted() {
  // get countries from json
  var json = require("../utils/countries.json");
  for (var key in json) {
    if (Object.prototype.hasOwnProperty.call(json, key)) {
      this.c_siglas.push(key);
      this.c_names.push(json[key]);
    }
  }
},

```

Figura 67 - Armazenamento dos países e siglas em arrays no mounted do Register.vue

RELATÓRIO DO PROJETO

Depois simplesmente uso um componente do Vuetify para criar uma lista select com todos os países do array.

```
<!-- COUNTRY -->
<v-autocomplete
  cache-items
  flat
  hide-no-data
  hide-details
  item-text
  v-model="select_country"
  :items="c_names" <----- Red arrow here
  :search-input.sync="searchCountry"
  :label="$t('register.country')"
  :rules="[rules.required]"
  @input="handleInput($event, 'country')"
  prepend-icon="mdi-flag"
  type="text"
  color="reverse"
  class="mb-5"
>
```

Figura 68 - Código do componente com todos os países



Figura 69 - Componente com todos os países

A password também tem uma regra que é, tem que ter mais que 8 caracteres.



Figura 70 - Verificação da password



RELATÓRIO DO PROJETO

Também contém uma barra que vai mudando de cor, esta barra é também um componente do Vuetify, eu só programei e defini o “mínimo de caracteres” para a password ser válida.

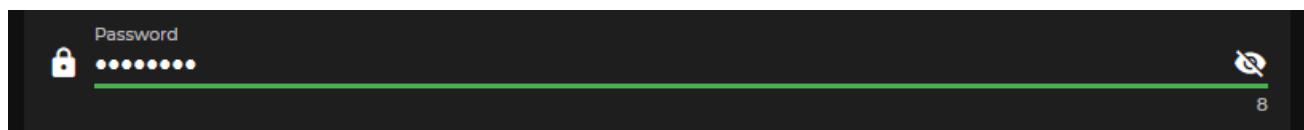


Figura 71 - Password válida (barra verde)

Resumindo, os campos para a criação de uma conta têm as seguintes regras:

```
rules: {
    required: (value) => !!value || this.$t("login.required"),
    name: (value) =>
        (!value && value.length >= 4) || this.$t("register.name_short"),
    email: (value) => {
        const pattern = /^(([^<><>()[]\\.,;:\\s@"]+)(\\.[^<>()[]\\.,;:\\s@"]+)*)(\\.[^.+])+$/;
        return pattern.test(value) || this.$t("register.invalid_email");
    },
    password: (value) => {
        return (!value && value.length >= 8) || this.$t("register.pw_short");
    },
},
```

Figura 72 - Regras para os campos ao criar conta em Register.vue

- required – preenchimento obrigatório (aplicado a todos os campos)
- name – verificação do tamanho (aplicado ao nome e ao username)
- email – verificação do email (aplicado ao campo do email)
- password – verificação do tamanho (aplicado ao campo da password)

RELATÓRIO DO PROJETO

Criação de “Posts”

A criação de Posts é feita na view UploadPost.vue.

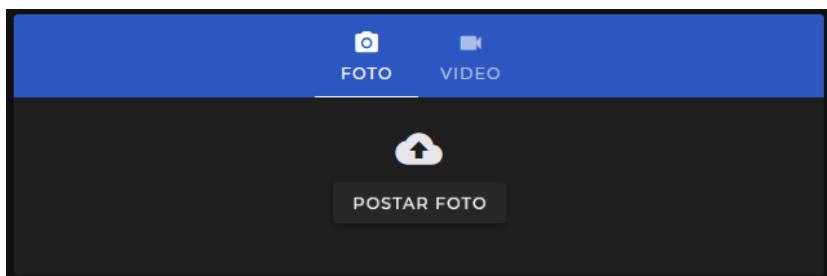


Figura 73 - Criar post (rota /uploadpost)

Ao dar upload de uma foto, irá aparecer os campos a preencher.

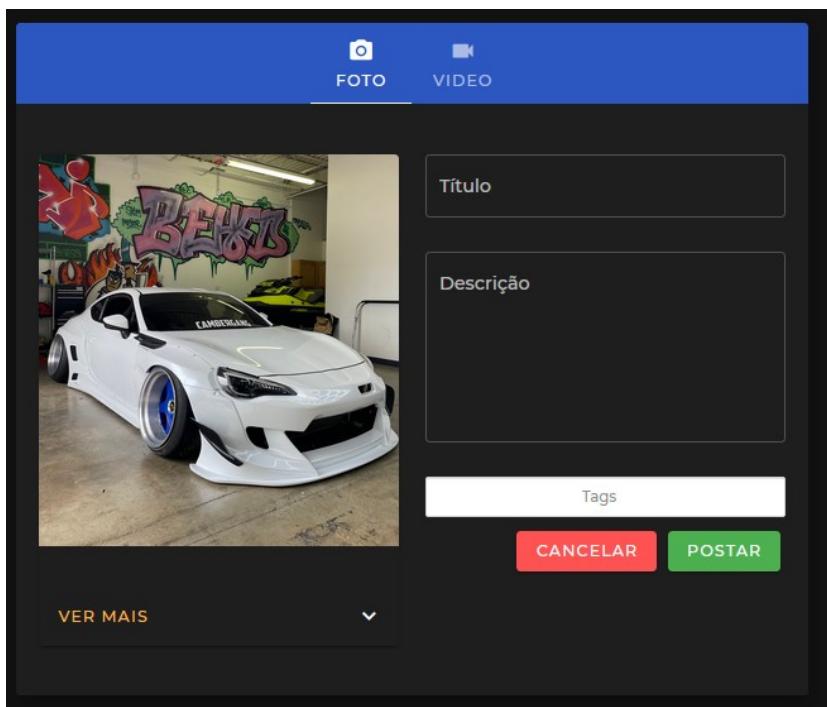


Figura 74 - Campos do novo post

O único campo obrigatório é o Título. Ao preencher os campos, o “card do post preview” da esquerda atualiza e mostra como irá ficar no perfil. *Exemplo foto:*

RELATÓRIO DO PROJETO

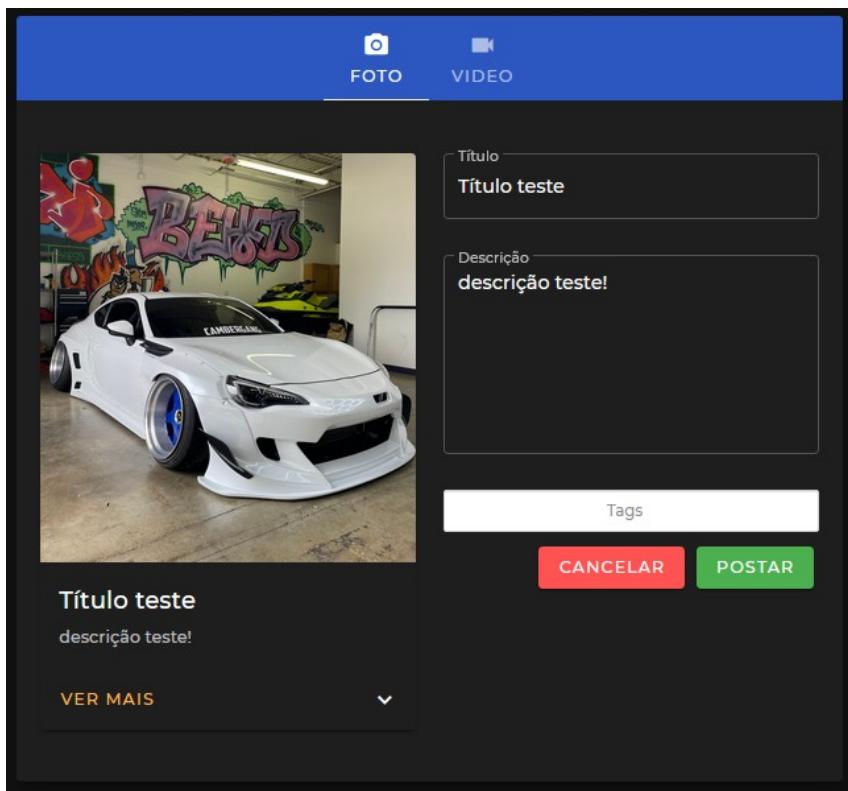


Figura 75 - Preview do post no perfil com os campos

Ao clicar em “Postar”, ele irá executar esta função “onPost”, que basicamente, primeiro verifica se o campo obrigatório está preenchido, se não estiver retorna erro, se estiver, irá chamar uma Action do Vuex “createPost”.

```

378    |     async onPressed() {
379    |       if (!this.title) {
380    |         this.errorSwal(this.$t("no_title"));
381    |         this.$refs["postTitle"].$refs.input.focus();
382    |         return;
383    |       }
384
385    |       this.$store.dispatch("createPost", {
386    |         title: this.title,
387    |         content: this.description,
388    |         file: this.img.image || this.vid.video,
389    |         tags: this.tags,
390    |       });
391    |

```

Figura 76 - Função onPressed dentro de UploadPost.vue

RELATÓRIO DO PROJETO

Esta função/Action “createPost” irá chamar 2 rotas ao backend:

- “/posts” - Criar um novo “post” na base de dados com os dados (título, descrição, tags..)
- “post/:id/file” - Associar um ficheiro (imagem por exemplo) ao id do Post criado anteriormente.

```

275 |     async createPost({ commit }, { title, file, content, tags }) {
276 |       commit("SET_LOADING_STATUS", true);
277 |       const formData = new FormData();
278 |       formData.append("file", file);
279 |
280 |       api.post("/posts", { title, content, tags }, {
281 |         headers: {
282 |           Authorization: `Bearer ${localStorage.getItem("token")}`,
283 |         },
284 |       })
285 |         .then((response) => {
286 |           const post = response.data;
287 |           api
288 |             .put("/post/" + post.id + "/file", formData, {
289 |               headers: {
290 |                 Authorization: `Bearer ${localStorage.getItem("token")}`,
291 |                 "Content-Type": "multipart/form-data",
292 |               },
293 |             })
294 |             .then((res) => {
295 |               Vue.swal(i18n.t("success"), res.data.message, "success");
296 |               router.push("/profile");
297 |             })
298 |             .catch((error) => {
299 |               setTimeout(() => {
300 |                 handleError(error);
301 |               }, 900);
302 |             });
303 |         })
304 |         .catch((error) => {
305 |           setTimeout(() => {
306 |             handleError(error);
307 |           }, 900);
308 |         });
309 |       }
310 |     .catch((error) => {
311 |       setTimeout(() => {
312 |         handleError(error);
313 |       }, 900);
314 |     });
315 |   }
316 | 
```

Figura 77 - Função “createPost”, em vuex.js

Se tudo correr bem no backend, o post é criado e somos redirecionados para o nosso perfil.

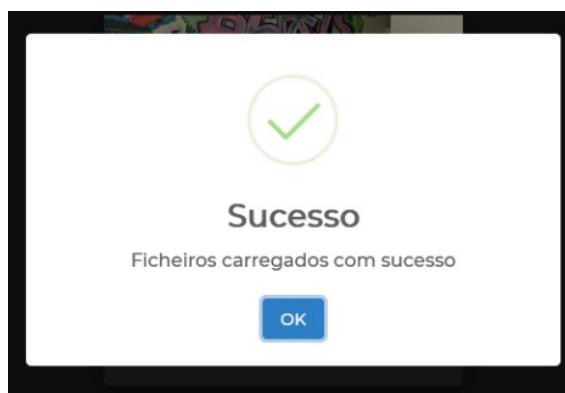


Figura 78 - Sucesso na criação do Post

RELATÓRIO DO PROJETO

Posts

Existem vários componentes destinados aos posts, esses componentes servem para mostrar os Posts (mostrar a foto, título, descrição, etc....)

Os componentes destinados a isso são: PostViewer e PostFeed. Claro que há outros componentes que vão ser chamados e utilizados dentro destes, como por exemplo Likes.vue (gostos), Comments.vue (comentários) e Tags.

PostViewer é o componente usado para mostrar os posts do perfil.

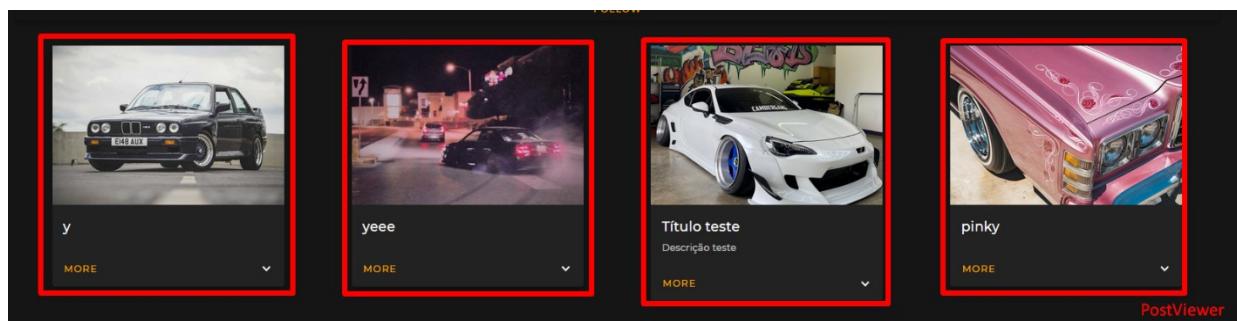


Figura 79 - Componente PostViewer

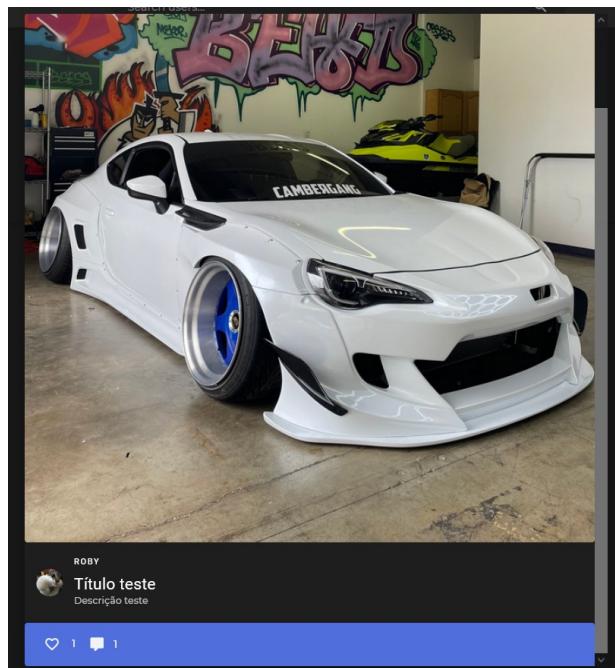


Figura 80 - Componente PostViewer quando clicamos no Post

RELATÓRIO DO PROJETO

Quando o perfil é carregado, um ciclo é executado e todos os posts do utilizador são enviados para este componente que irá mostrar o post e os detalhes.

```
<!-- POSTS -->
<v-container class="pa-4 mb-15">
  <v-row wrap>
    <template v-for="(post, index) in posts">
      <v-col :key="'post' + index">
        <PostViewer :post="post" />
      </v-col>
    </template>
  </v-row>
</v-container>
```

Figura 81 - Profile.vue

O PostFeed é usado para mostrar os Posts da página inicial (home):

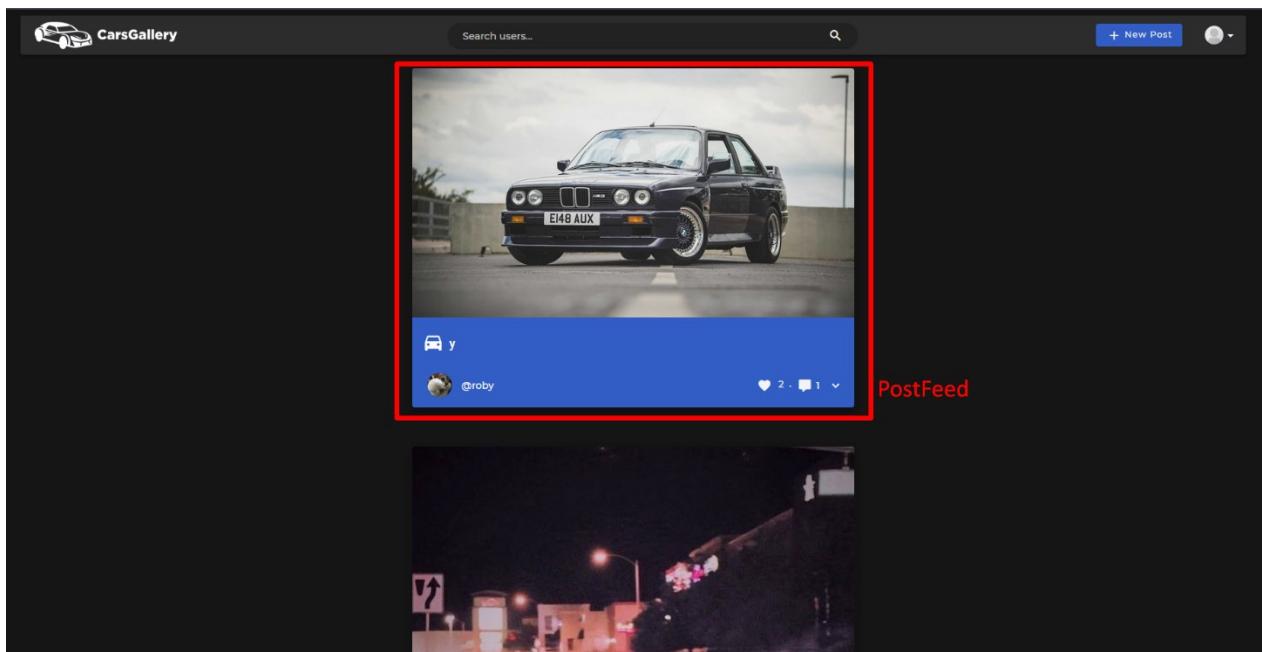


Figura 82 - Componente PostFeed



RELATÓRIO DO PROJETO

A lógica é a mesma, quando o utilizador vai para a página inicial, o frontend vai buscar todos os posts de utilizadores e mostrar neste componente.

```
<v-container fill-height>
  <v-row justify="center" align="center">
    <v-col cols="12" md="5" sm="8" v-if="posts.length > 0">
      <template v-for="(post, index) in posts">
        <PostFeed :key="index" :post="post" />
      </template>
    </v-col>
  </v-row>
</v-container>
```

Figura 83 - Home.vue



RELATÓRIO DO PROJETO

Comentários

A implementação dos comentários dos Posts no backend foi relativamente fácil, para isso criei simplesmente uma tabela (já referida anteriormente) para armazenar o comentário:

```
67 model Comment {  
68     id      Int      @default(autoincrement()) @id  
69     author  User    @relation(fields: [authorId], references: [id])  
70     authorId Int  
71     postId   Int  
72     post     Post    @relation(fields: [postId], references: [id])  
73     content  String  
74     createdAt DateTime @default(now())  
75     updatedAt DateTime @updatedAt  
76 }  
77 
```

Figura 84 - Tabela dos comentários

E depois na tabela dos posts, faço a relação (1-N):

```
16 model Post {  
17     id      Int      @default(autoincrement()) @id  
18     createdAt DateTime @default(now())  
19     updatedAt DateTime @updatedAt  
20     title   String   @db.VarChar(255)  
21     content String?  
22     likes   PostLike[]  
23     published Boolean @default(false)  
24     author  User    @relation(fields: [authorId], references: [id])  
25     authorId Int  
26     file    PostFile? @relation(fields: [fileId], references: [id])  
27     fileId   Int?  
28     tags    Tag[]  
29     comments Comment[]  
30 }  
31 
```

Figura 85 - Tabela dos posts



RELATÓRIO DO PROJETO

```
79 // Comment routes
80 router.get("/post/:id/comments", CommentController.getPostComments)
81 router.post(
82   "/post/:id/comments",
83   isAuthenticated,
84   CommentController.commentOnPost
85 )
86 router.put("/comments/:id", isAuthenticated, CommentController.updateComment)
87 router.delete("/comments/:id", isAuthenticated, CommentController.deleteComment)
88 |
```

Figura 86 - Rotas utilizadas para comentários (backend routes.ts)

```
try {
  const post = await database.post.findUnique({
    where: { id: Number(postId) },
  })

  if (!post) {
    return new AppError(res, {
      name: req.t("post_controller.post_not_found.name"),
      message: req.t("post_controller.post_not_found.message"),
      statusCode: 404,
    })
  }

  const comments = await database.comment.findMany({
    where: { post: { id: Number(postId) } },
    include: { author: { select: { id: true, username: true } } },
  })

  res.send(comments)
}
```

Figura 87 - Função getPostsComments, utilizado na rota "GET /post/:id/comments"

```
1 [
2 {
3   "id": 3,
4   "authorId": 12,
5   "postId": 10,
6   "content": "uaa",
7   "createdAt": "2021-06-28T00:46:33.068Z",
8   "updatedAt": "2021-06-28T00:46:33.069Z",
9   "author": {
10     "id": 12,
11     "username": "robby"
12   }
13 },
14 {
15   "id": 4,
16   "authorId": 12,
17   "postId": 10,
18   "content": "sque",
19   "createdAt": "2021-06-28T00:47:05.727Z",
20   "updatedAt": "2021-06-28T00:47:05.727Z",
21   "author": {
22     "id": 12,
23     "username": "robby"
24   }
25 }
```

Figura 88 - Resposta da rota "GET /post/:id/comments"

RELATÓRIO DO PROJETO

No frontend, criei um componente “Comments.vue”, que aberto através de um Post, irá servir para adicionar novos comentários e também para mostrar os comentários já feitos.

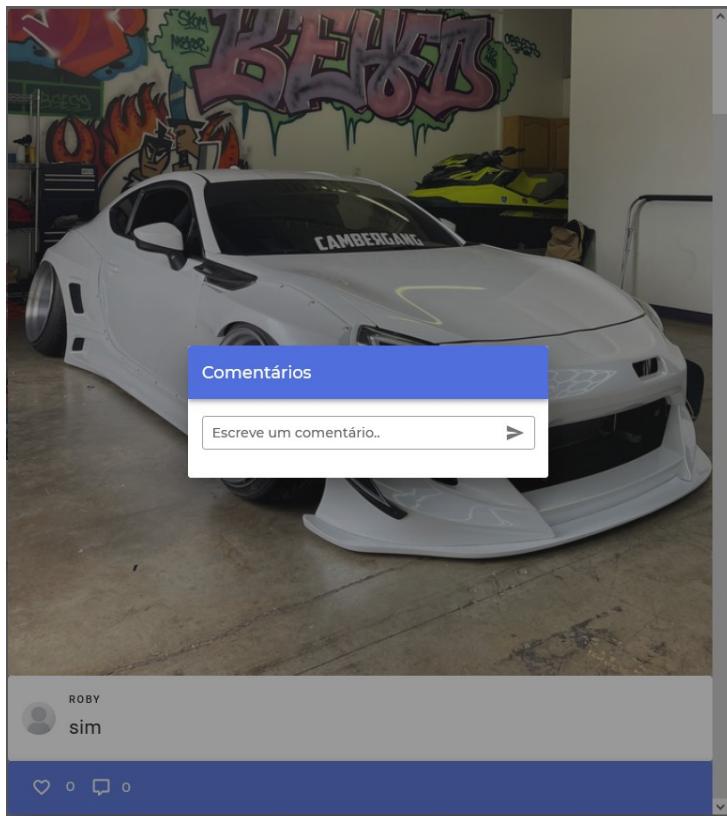


Figura 89 - Componente Comments.vue

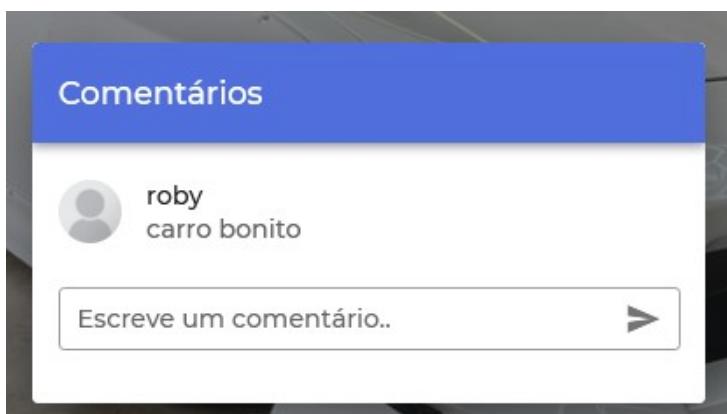


Figura 90 - Comentário introduzido

RELATÓRIO DO PROJETO

Para chamar este componente, tenho que passar vários parâmetros

```
51 | <Comments :dialogC.sync="dialogComments" :comments="comments" :postid="post.id"/>
52 |
53 |
```

Figura 91 - Componente utilizado dentro de outro componente

“dialogComments” é uma variável bool que define se o dialog/janela dos comentários tem que estar aberta, “comments” é o array dos comments do post, “post.id” é o id do post.

Ao fazer um comentário, irei chamar a função “commentOnPost” do vuex e adicionar ao array “comments” o meu comentário, o meu id e o comentário em si.

```
85 |     onSendComment() {
86 |       if (this.comment) {
87 |         this.$store.dispatch("commentOnPost", {
88 |           post: this.postid,
89 |           comment: this.comment,
90 |         });
91 |
92 |         this.comments.push({
93 |           author: {
94 |             id: this.$store.state.user.id,
95 |             username: this.$store.state.user.username,
96 |           },
97 |           content: this.comment,
98 |         });
99 |         this.comment = "";
100 |         document.activeElement.blur();
101 |       }
102 |     },
103 |   },
```

Figura 92 - Função quando um comentário é introduzido

```
224 |
225 |   commentOnPost({ commit }, { post: postId, comment: content }) {
226 |     // post /post/:id/comments
227 |     api.post(`/post/${postId}/comments`, { content: content }, {
228 |       headers: {
229 |         Authorization: "Bearer " + localStorage.getItem("token"),
230 |       },
231 |     })
232 |     .catch((error) => {
233 |       setTimeout(() => {
234 |         handleError(error);
235 |       }, 900);
236 |     });
237 |   },
238 | }
```

Figura 93 - Função “commentOnPost” que chama a rota do backend que adiciona um comentário ao post. (rota POST “/post/:id/comments”)



RELATÓRIO DO PROJETO

Os comentários na parte visual têm só um pequeno ajuste, que é, se o post tiver comentários, aparece um ícone diferente se não tiver (com uma simples operação ternária).

```
<v-btn @click="dialogComments = true" icon color="white">
  <v-icon>
    {{ post._count.comments
      ? "mdi-comment"
      : "mdi-comment-outline"
    }}
  </v-icon>
</v-btn>
```

Figura 94 - PostViewer.vue

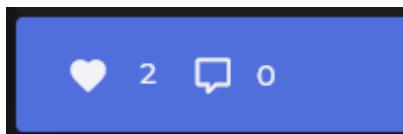


Figura 95 – Post sem comentários

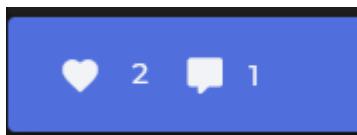


Figura 96 - Post com comentários

RELATÓRIO DO PROJETO

Likes

Os likes são os “gostos” de cada post. Qualquer utilizador pode dar like em qualquer post. A lógica por trás é simples

```

1  <template>
2    <div style="display: inline">
3      <v-btn icon color="white" @click="iLike">
4        <v-icon> {{ like ? "mdi-heart" : "mdi-heart-outline" }} </v-icon>
5      </v-btn>
6      <span class="subheading mr-2 white--text"> {{ likes }} </span>
7    </div>
8  </template>
9
10 <script>
11 export default {
12   name: "Like",
13   props: {
14     postid: Number,
15     likes: Number
16   },
17   data() {
18     return {
19       like: false
20     };
21   },
22   async mounted() {
23     this.like = await this.$store.dispatch("DoILikeThePost", this.postid)
24   },
25   methods: {
26     iLike() {
27       const amountOfLikes = this.likes;
28
29       if (!this.like) {
30         this.$emit("update:likes", amountOfLikes + 1);
31         this.$store.dispatch("likePost", this.postid);
32       } else {
33         this.$emit("update:likes", amountOfLikes - 1);
34         this.$store.dispatch("unlikePost", this.postid);
35       }
36
37       this.like = !this.like;
38     }
39   }
40 };

```

Figura 97 - Código do componente Like.vue

Basicamente, sempre que chamo o componente, tenho que passar por argumento o ID do post e a quantidade de likes, assim este componente vai controlar a adição de novos likes.

A programação visual verifica se temos like ou não, para mostrar um dos seguintes dois tipos de ícone:

RELATÓRIO DO PROJETO

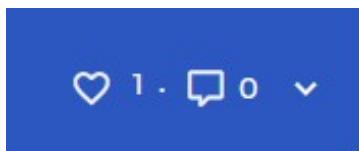


Figura 98 - Componente se não tivermos like



Figura 99 - Componente se tivermos like

Esta verificação é feita através da função “DoILikeThePost”, que chama uma action e esse action por si chama uma rota ao backend (GET /post/:id/like) e que retorna verdadeiro se eu (o utilizador do site) tiver like num específico post.

```
async mounted() {
  this.like = await this.$store.dispatch("DoILikeThePost", this.postid)
},
```

Figura 100 - Quando o componente é criado, verifica se o utilizador tem like no post ou não

```
DoILikeThePost({ commit }, postid) {
  // GET /post/:id/like
  commit("SET_LOADING_STATUS", true);

  return new Promise((resolve, reject) => {
    api
      .get(`/post/${postid}/like`)
      .headers: {
        Authorization: `Bearer ${localStorage.getItem("token")}`,
      }
    .then((res) => {
      commit("SET_LOADING_STATUS", false);
      resolve(res.data.liked);
    })
    .catch((error) => {
      commit("SET_LOADING_STATUS", false);

      setTimeout(() => {
        handleError(error);
      }, 900);

      reject(null);
    });
  });
},
```

Figura 101 - Action "DoILikeThePost" chamada quando o componente é criado

RELATÓRIO DO PROJETO

```

userLikedPost: async function (req: Request, res: Response) {
  const { id: postId } = req.params
  const { id: userId } = req.user as User

  try {
    const post = await database.post.findUnique({
      where: { id: Number(postId) },
    })

    if (!post || (!post.published && post.authorId !== userId)) {
      return new AppError(res, {
        name: req.t("post_controller.post_not_found.name"),
        message: req.t("post_controller.post_not_found.message"),
        statusCode: 404,
      })
    }

    const liked = await database.postLike.findFirst({
      where: { postId: post.id, userId },
    })

    res.send({ liked: !!liked }) ←
  } catch (error) {
    new AppError(res, {
      name: req.t("server_internal"),
      message: error.message,
      statusCode: 500,
    })
  }
}

```

Figura 102 - Função `userLikedPost` no backend, que verifica se na tabela `PostLike` o utilizador tem like no post ou não

```

45  model PostLike {
46    id          Int      @default(autoincrement()) @id
47
48    user        User     @relation(fields: [userId], references: [id])
49
50    userId     Int
51
52    post        Post     @relation(fields: [postId], references: [id])
53
54    postId     Int
55  }

```

Figura 103 - Tabela `PostLike`

RELATÓRIO DO PROJETO

Este componente dos likes é utilizado em outros componentes, mais especificamente nos componentes dos Posts, porque é aí que os likes são necessários.

```
<v-toolbar color="primary lighten-1">
  <Like :postid="post.id" :likes.sync="post._count.likes" />

  <v-btn @click="dialogComments = true" icon color="white">
    <v-icon>
      {{
        post._count.comments
        ? "mdi-comment"
        : "mdi-comment-outline"
      }}
    </v-icon>
  </v-btn>
  <span class="subheading mr-2 white--text">
    {{ post._count.comments }}
  </span>
  <v-spacer></v-spacer>

  <template v-if="mySelf(post.author.id)">
    <v-btn @click="handleDeletePost" icon color="white">
      <v-icon> mdi-delete </v-icon>
    </v-btn>
  </template>
</v-toolbar>
```

Figura 104 - Componente PostViewer a usar o componente Like

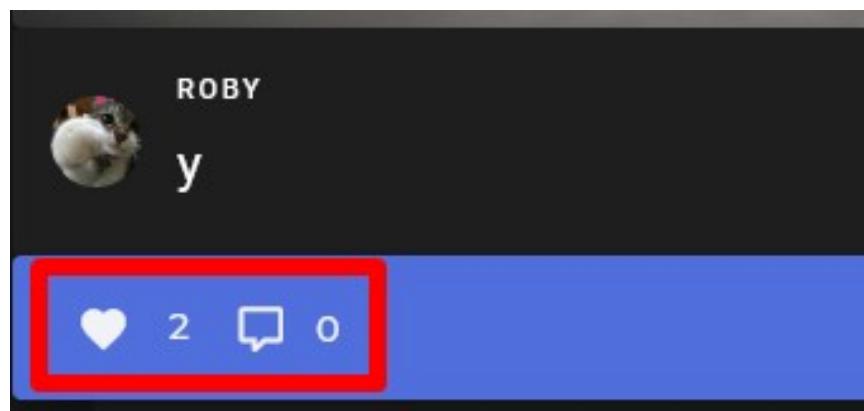


Figura 105 - Parte visual a que o código da figura acima se refere

RELATÓRIO DO PROJETO

Seguidores

Para seguir um utilizador, basta ir ao perfil do mesmo e clicar no botão “Seguir”.

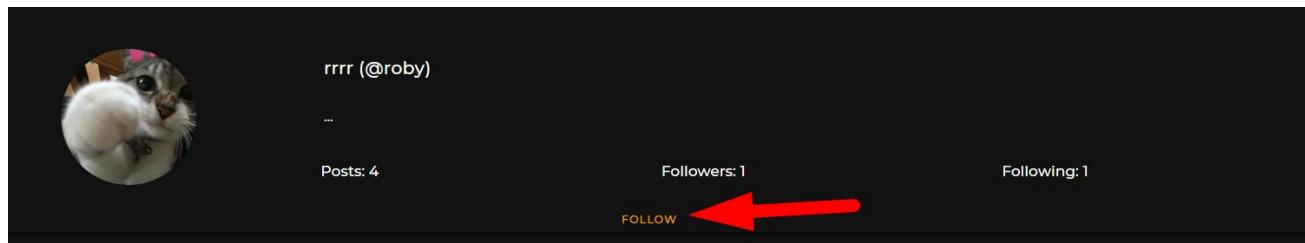


Figura 106 - Perfil do utilizador

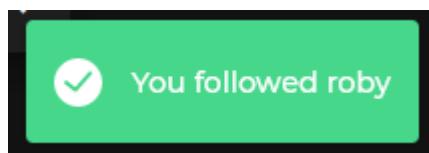


Figura 107 - Popup ao seguir utilizador

O botão executa a função “handleFollowClick”, que basicamente chama uma das actions dependendo se vamos seguir ou deixar de seguir o utilizador.

```
handleFollowClick() {
    if (this.doIFollowTheUser) { // unfollow
        this.$store.dispatch("unfollowUser", this.user.id);
        this.user._count.followedBy = this.user._count.followedBy - 1;
        this.$toast.info(this.$t("you_unfollowed") + this.user.username)
    } else { // follow
        this.$store.dispatch("followUser", this.user.id);
        this.user._count.followedBy = this.user._count.followedBy + 1;
        this.$toast.success(this.$t("you_followed") + this.user.username)
    }

    this.doIFollowTheUser = !this.doIFollowTheUser;
},
```

Figura 108 - Função handleFollowClick (Profile.vue)

Estas actions basicamente vão chamar 2 rotas: “POST /follow/:id” se for para seguir o utilizador, ou “DELETE /unfollow/:id” para deixar de seguir o utilizador.

RELATÓRIO DO PROJETO

```

449 followUser({ commit }, userid) {
450   // post /follow/:id
451
452   api
453     .post(`/follow/${userid}`, undefined, {
454       headers: {
455         Authorization: "Bearer " + localStorage.getItem("token"),
456       },
457     })
458     .catch((error) => {
459       setTimeout(() => {
460         handleError(error);
461       }, 900);
462     });
463   },
464
465 unfollowUser({ commit }, userid) {
466   // delete /unfollow/:id
467
468   api
469     .delete(`/unfollow/${userid}`, {
470       headers: {
471         Authorization: "Bearer " + localStorage.getItem("token"),
472       },
473     })
474     .catch((error) => {
475       setTimeout(() => {
476         handleError(error);
477       }, 900);
478     });
479   },
480

```

Figura 109 - actions `followUser` e `unfollowUser` do vuex.js

O frontend sabe qual é a action que vamos executar pois, temos outra action que se chama “`DolFollowTheUser`”, que nos retorna verdadeiro se seguimos um certo utilizador ou falso se não. Esta função é útil pois ao clicar no botão o frontend tem que saber se já seguimos o utilizador para deixar de seguir ou se não seguimos o utilizador passamos a seguir.

```

doIFollowTheUser({ commit }, userId) {
  commit("SET_LOADING_STATUS", true);

  return new Promise((resolve, reject) => {
    api
      .get(`/user/${userId}/isFollowing`, {
        headers: {
          Authorization: `Bearer ${localStorage.getItem("token")}`,
        },
      })
      .then((res) => {
        commit("SET_LOADING_STATUS", false);
        resolve(res.data.following);
      })
      .catch((error) => {
        commit("SET_LOADING_STATUS", false);
        setTimeout(() => {
          handleError(error);
        }, 900);

        reject(null);
      });
  });
},

```

Figura 110 - action `dolFollowTheUser`

RELATÓRIO DO PROJETO

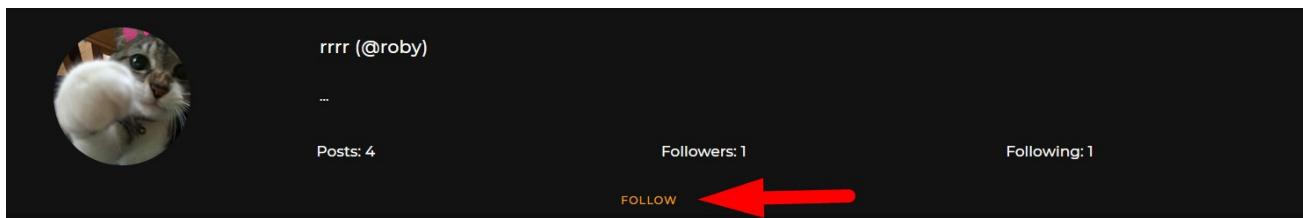


Figura 111 - Perfil de um utilizador que não sigo



Figura 112 - Perfil de um utilizador que sigo

Quero também referir que, na página inicial, os posts são ordenados, os primeiros que aparecem são dos utilizadores que “seguimos” os restantes de outros utilizadores.

```

if (user) {
  posts.sort([(postA, postB) => {
    const { followedBy: followedByA } = postA.author
    const { followedBy: followedByB } = postB.author

    const isFollowedByA = followedByA.find(
      (follower) => follower.id === user?.id
    )
    const isFollowedByB = followedByB.find(
      (follower) => follower.id === user?.id
    )

    if (
      (isFollowedByA && postA.createdAt > postB.createdAt) ||
      !isFollowedByB
    )
      return -1
    else if (
      (isFollowedByB && postA.createdAt < postB.createdAt) ||
      !isFollowedByA
    )
      return 1
    else return 0
  }])
}

```

Figura 113 - Código que ordena os posts

RELATÓRIO DO PROJETO

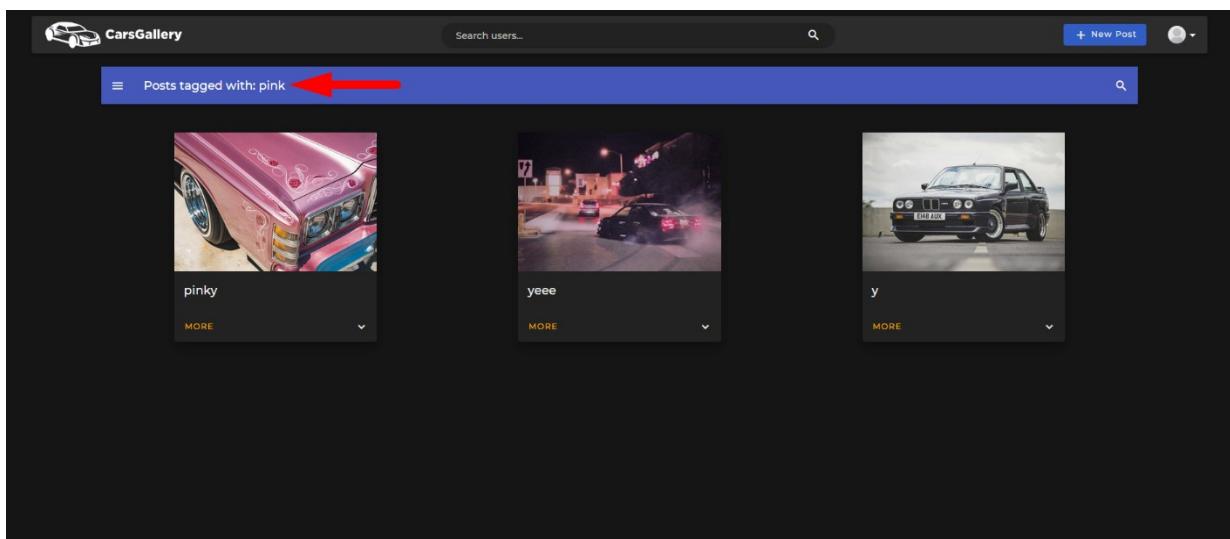
Tags

As tags são um tipo de “palavras chaves” para uma certa imagem que o utilizador pode adicionar.

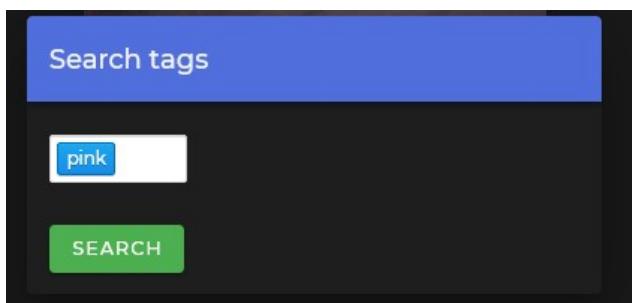
Um exemplo, se eu tirar foto a um carro da marca X e da cor Y, posso adicionar as tags X e Y.

Mas para que é que as tags são úteis? As tags são úteis pois, podemos pesquisar TODOS os posts com certa tag. Por exemplo, se eu procurar pela tag “pink”, aparecerão todos os posts com essa tag.

TagSearch.vue:



Se clicarmos na lupa, podemos pesquisar por mais tags.



Para obtermos todos os posts com certa tag, usamos a função “getPostsFromTag” (chamada quando a página TagSearch.vue é carregada) que usa a rota “GET /posts”, passando as tags por argumento (no url) e sendo assim o backend procura na base de dados e retorna os posts com a tag.

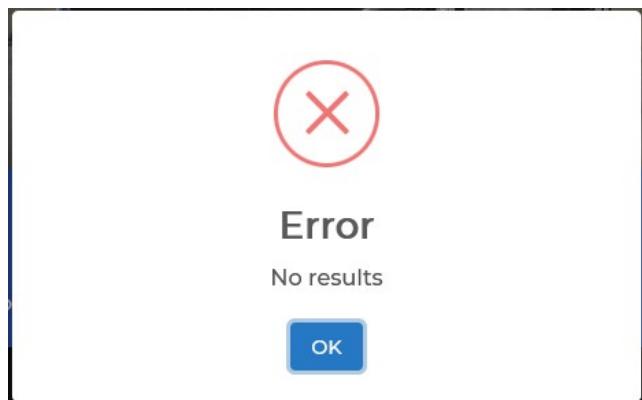
RELATÓRIO DO PROJETO

```
async beforeRouteUpdate(to, from, next) {
  this.dialogTags = false
  this.tags = to.params.tags.split(",");
  this.posts = await this.$store.dispatch("getPostsFromTag", this.tags);

  if (!this.posts.length) {
    this.$swal(this.$t("error"), this.$t("no_results"), "error");
    return this.$router.push("/home");
  }

  next()
},
```

Caso não haja resultados, retorna erro e redireciona para a página principal.



Procurar utilizadores

Para procurar outros utilizadores, temos esta barra na navbar:

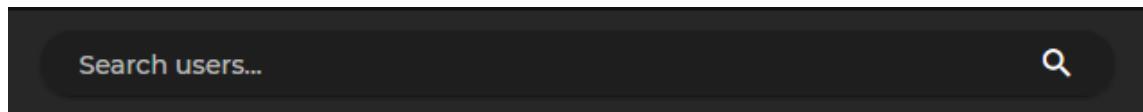


Figura 114 - Barra para pesquisar utilizadores (componente SearchUsers.vue)

Quando o componente é carregado, ele vai armazenar todos os utilizadores (pela rota “GET /users”), esses utilizadores são armazenados num array. Quando introduzimos algum valor na barra, ele atualiza o array, chamando a rota de novo.

RELATÓRIO DO PROJETO

```

watch: {
  search(val) {
    if (this.users.length > 0) return;
    if (this.isLoading) return;
    this.isLoading = true;

    api.get("/users")
      .then(res => {
        this.count = res.data.length;
        this.entries = res.data;
        this.users = res.data
      })
      .catch(err => {
        console.log(err.response);
      })
      .finally(() => (this.isLoading = false));
  }
}
  
```

Figura 115 - Função que é executada cada vez que a barra recebe input

Graças ao Vuetify, existe um componente perfeito para este caso, que é o componente autocomplete, basicamente, cria um “menu select” com várias opções, neste caso, as opções serão os dados do array (que são os utilizadores), ao escrever algo na barra, ele elimina os dados irrelevantes, e guarda só os que começam pelas mesmas letras.

```

<v-autocomplete
  filled
  rounded
  dense
  solo
  color="reverse"
  v-model="model"
  :items="users"
  :loading="isLoading"
  :search-input.sync="search"
  hide-no-data
  hide-selected
  item-text="username"
  item-value="id"
  :label="$t('navbar.search')"
  append-icon="mdi-magnify"
  maxlength="32"
  return-object
  @change="(event) => redirectUser(event, model.id)"
>
</v-autocomplete>
  
```

Figura 116 - Componente autocomplete (que é a barra de pesquisa)



RELATÓRIO DO PROJETO

Como podemos ver, ao escrever roby, todos os utilizadores que têm o username começado por roby irão aparecer, ao clicar num deles, somos redirecionados para o perfil do mesmo.

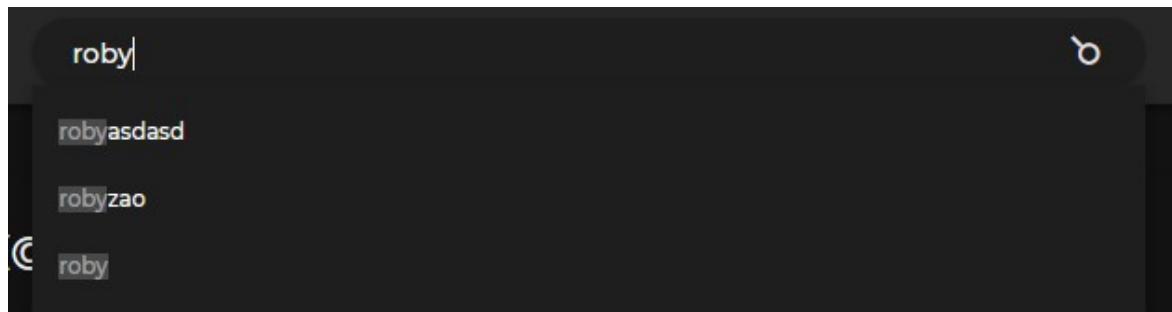


Figura 117 - Componente autocomplete para a pesquisa de utilizadores

Ao clicar num dos items da lista, o evento “@change” ocorre, logo a função “redirectUser” é chamada.

```
<template>
  <v-autocomplete
    filled
    rounded
    dense
    solo
    color="reverse"
    v-model="model"
    :items="users"
    :loading="isLoading"
    :search-input.sync="search"
    hide-no-data
    hide-selected
    item-text="username"
    item-value="id"
    :label="$t('navbar.search')"
    append-icon="mdi-magnify"
    maxlength="32"
    return-object
    @change="(event) => redirectUser(event, model.id)"
  >
</v-autocomplete>
</template>
```

Figura 118 - Componente autocomplete para a pesquisa de utilizadores

Esta função irá redirecionar para a rota do perfil do utilizador selecionado.

```
redirectUser() {
  this.$router.push('/profile/' + this.model.id)
  this.$router.go()
```

Figura 119 - Função que redireciona para o perfil selecionado



RELATÓRIO DO PROJETO

Definições da conta

Ao aceder às configurações, podemos mudar vários campos, como a foto, a biografia, o nome, o username, o email, a password ou até podemos apagar a conta.

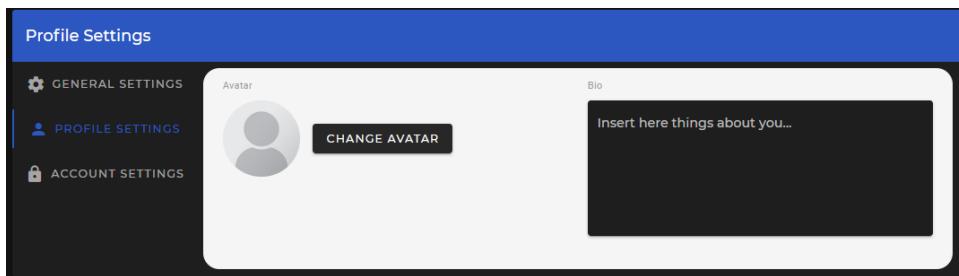


Figura 120 - ProfileSettings.vue

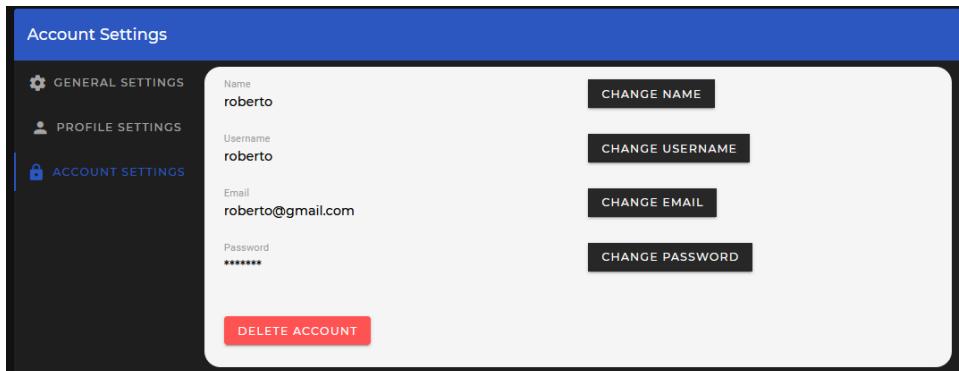


Figura 121 - ProfileSettings.vue

Ao clicarmos num dos botões aparece um “dialog”, que pede o novo campo.

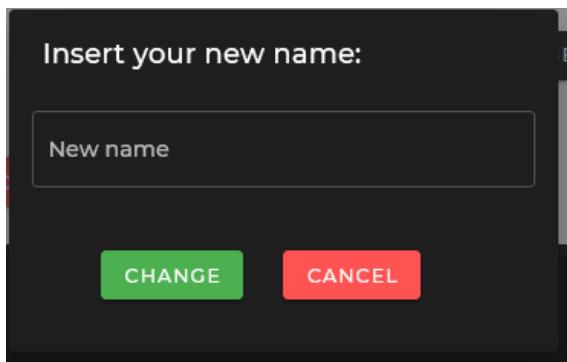


Figura 122 - Dialog que pede o novo campo (ProfileSettings.vue)

RELATÓRIO DO PROJETO

Independentemente do campo que for atualizado, a função chamada é “updateUserProfile”, que passando o novo campo atualizado e uma chave para o mesmo, fazemos uma chamada ao backend (rota PUT /user), com o novo campo (variável “data”).

Exemplo ao atualizarmos o nome:

```

501     async handleDone() {
502       if (this.changeFieldType === "password" && !this.newFieldInput) {
503         this.errorSwal(this.$t("profile_settings.no_input"));
504         return;
505       }
506
507       switch (this.changeFieldType) {
508         case "nome":
509         case "name": [
510           if (this.newFieldInput.length < 4) {
511             this.errorSwal(this.$t("register.name_short"));
512             return;
513           }
514           this.user = await this.$store.dispatch("updateUserProfile", { name: this.newFieldInput });
515           break;
516         ]
517       >         case "email": ...
518       >         case "username": ...
519       >         case "senha":
520       >         case "password": ...
521       >       }
522       this.changeFieldDialog = false
523     },
524   
```

Figura 123 - Função `handleDone`, que chama a action “`updateUserProfile`”

```

updateUserProfile({ commit }, data) {
  // PUT /user
  commit("SET_LOADING_STATUS", true);

  return new Promise((resolve, reject) => {
    api.put("/user", data, {
      headers: {
        Authorization: "Bearer " + localStorage.getItem("token"),
      },
    })
    .then((response) => {
      commit("SET_LOADING_STATUS", false);
      commit("SET_USER", response.data.user);
      Vue.$toast.success(response.data.message);
      resolve(response.data.user);
    })
    .catch((error) => {
      commit("SET_LOADING_STATUS", false);
      setTimeout(() => {
        handleError(error);
      }, 900);
      reject(null);
    });
  });
}

```

Figura 124 - Action “`updateUserProfile`”

RELATÓRIO DO PROJETO

Deteção de erros

Para detetar erros, não tive nenhuma maneira específica, no backend por exemplo utilizei o método “try catch” e mostrar o erro na consola, no frontend simplesmente usei a consola para ver se existem erros e usei uma função “handleErro” que mostra um popup com erro.

```

6  function handleError(error) {
7    console.dir(error)
8    if (error.response) {
9      // chamada feita e resposta
10     Vue.swal(i18n.t("error"), error.response.data.message, "error")
11     console.log(error.response.data.message)
12   } else if (error.request) {
13     // chamada feita mas sem resposta
14     Vue.swal(error.message, i18n.t("api_no_response"), "error")
15   } else {
16     Vue.swal(i18n.t("error"), i18n.t("smt_went_wrong"), "error")
17     console.log(error.message)
18   }
19 }
20

```

Figura 125 - Função handleError em functions.js

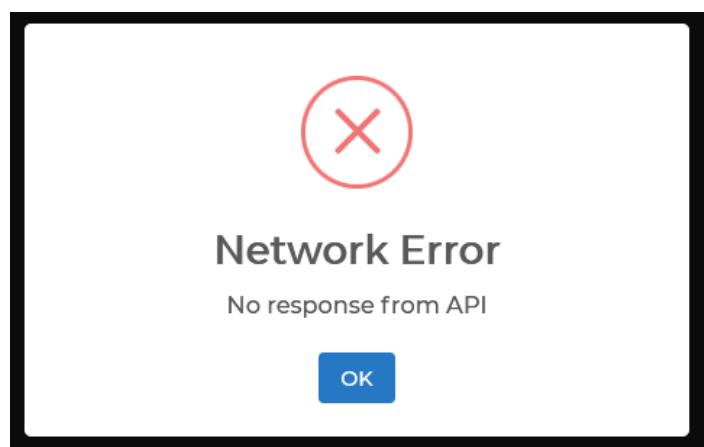


Figura 126 - Popup com erro

Layout

Agora irei mostrar o site todo (páginas, componentes, etc....)

Quero também referir que todo o layout do frontend é minimamente responsivo, sendo assim a aplicação/site utilizável para telemóvel também.

RELATÓRIO DO PROJETO

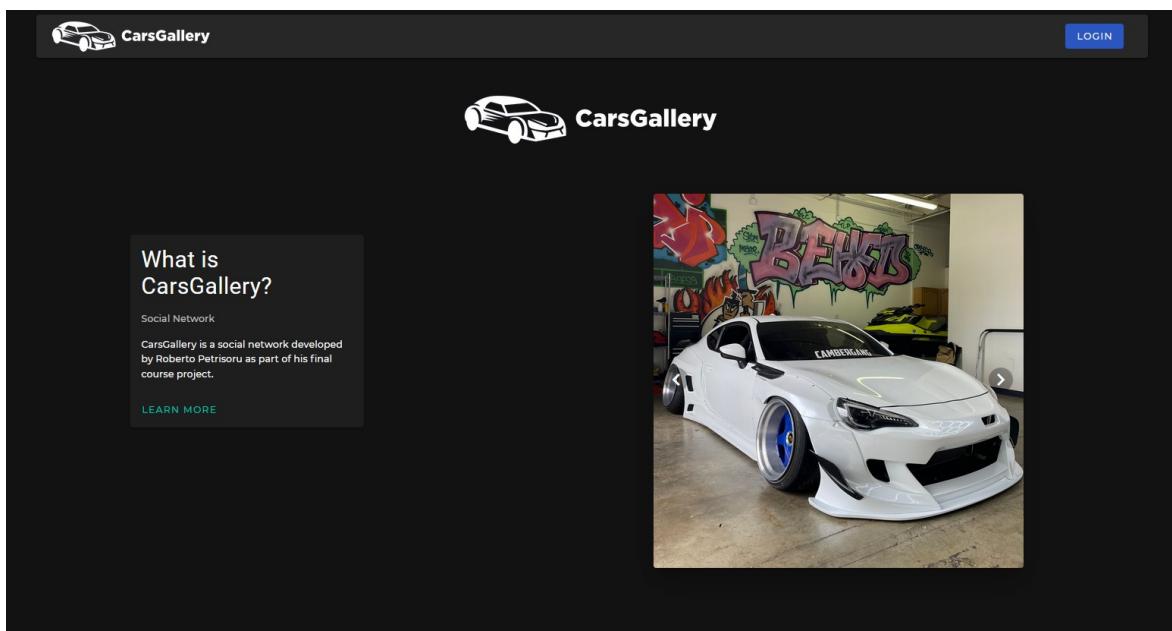


Figura 127 - Página inicial para visitantes

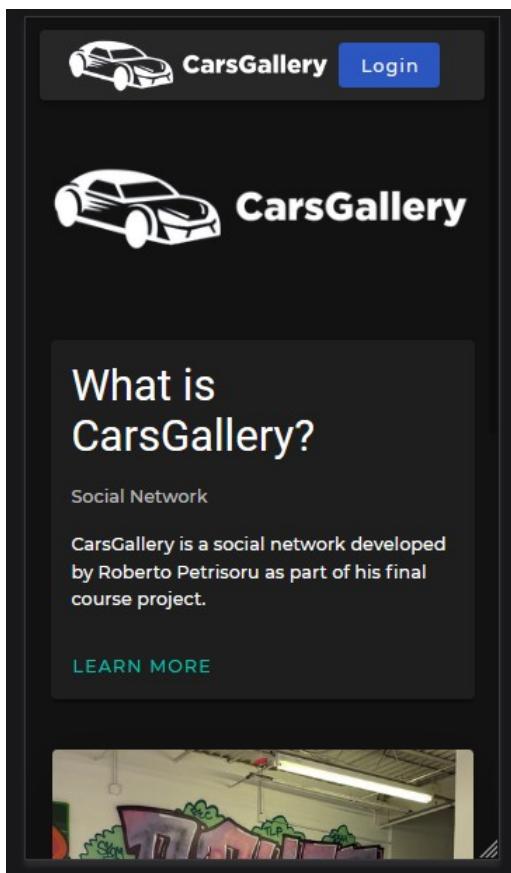


Figura 128 - Página inicial para visitantes

Cofinanciado por:



RELATÓRIO DO PROJETO

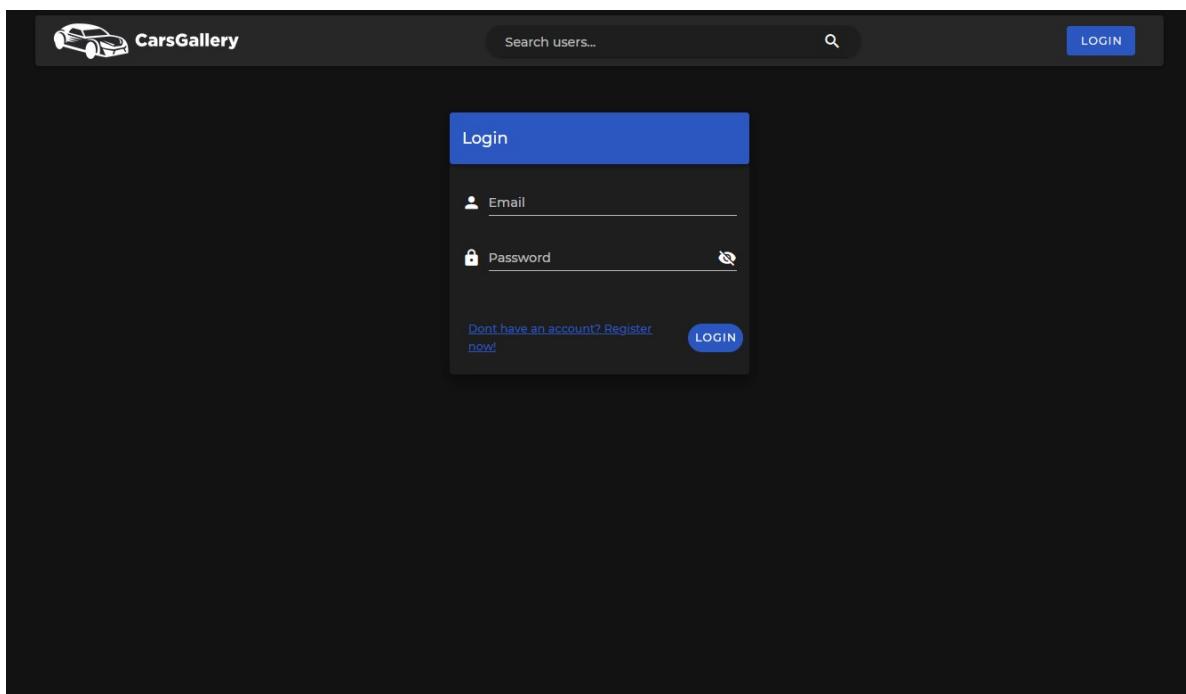


Figura 129 - Página login

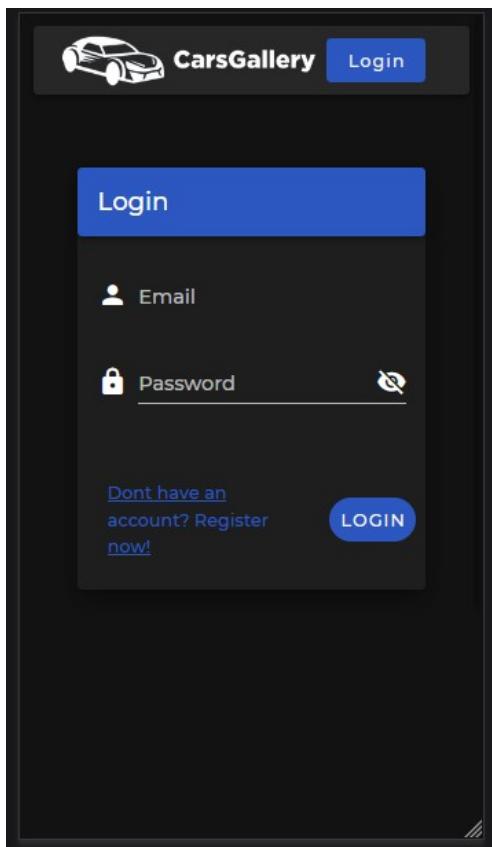
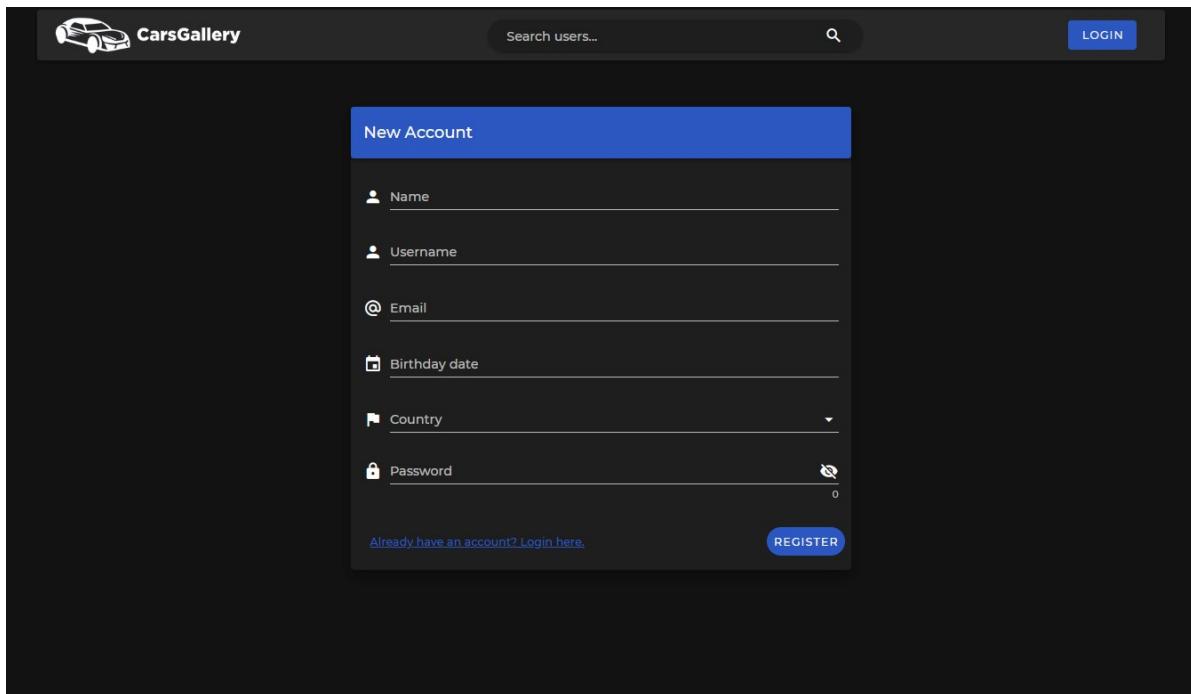


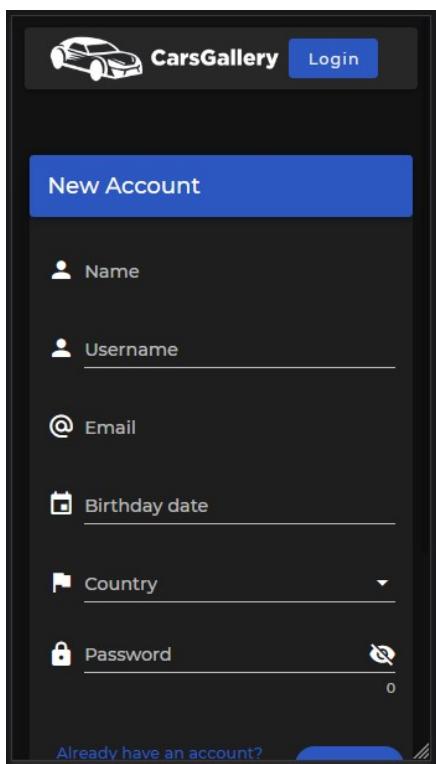
Figura 130 - Página login telemóvel

RELATÓRIO DO PROJETO



The screenshot shows a dark-themed web page for 'CarsGallery'. At the top left is a car icon and the text 'CarsGallery'. To the right is a search bar with placeholder text 'Search users...' and a magnifying glass icon. Further right is a blue 'LOGIN' button. Below the header is a blue rectangular box containing the text 'New Account'. Underneath this box are several input fields: 'Name' (with a person icon), 'Username' (with a person icon), 'Email' (with an '@' icon), 'Birthday date' (with a calendar icon), 'Country' (with a flag icon and a dropdown arrow), and 'Password' (with a lock icon). To the right of the password field is a character counter showing '0'. At the bottom of the form is a link 'Already have an account? Login here.' and a blue 'REGISTER' button.

Figura 131 - Página registrar



This screenshot shows a mobile-optimized version of the 'CarsGallery' website. The top navigation bar includes a car icon, the 'CarsGallery' logo, and a 'Login' button. The main content area features a blue header bar with the text 'New Account'. Below it are the same six registration fields as the desktop version: Name, Username, Email, Birthday date, Country, and Password. The password field includes a character count of '0' and a visibility toggle icon. At the bottom of the form is a link 'Already have an account?' and a blue 'REGISTER' button.

Figura 132 - Página registrar telemóvel

RELATÓRIO DO PROJETO

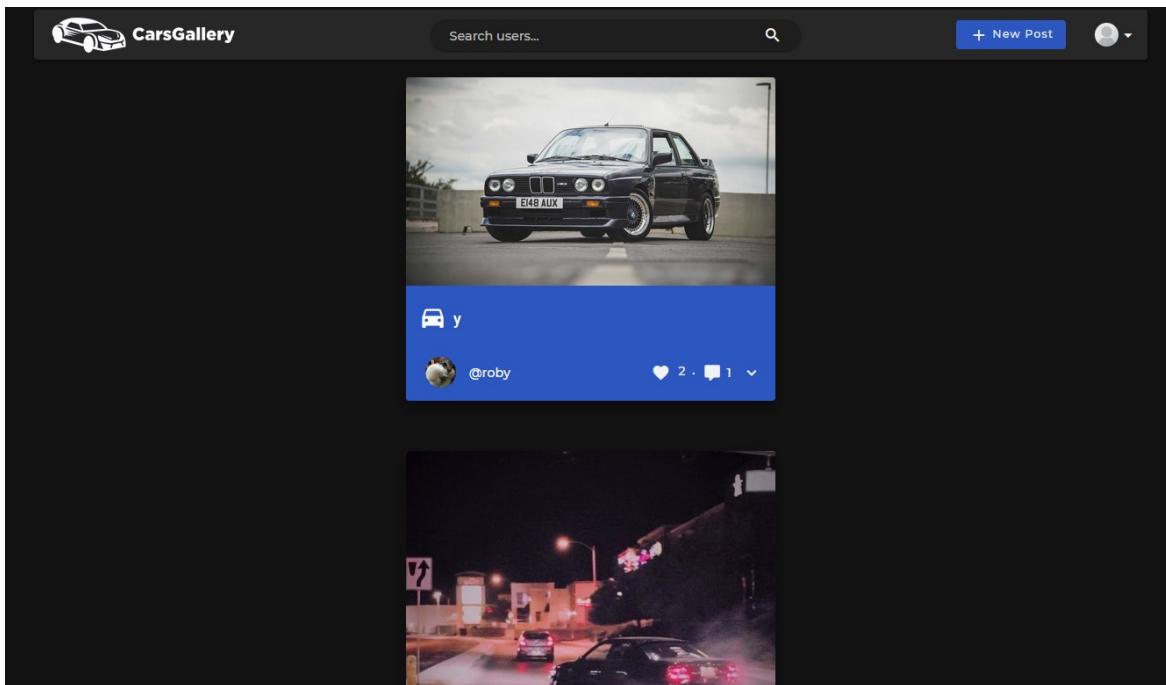


Figura 133 - Página inicial

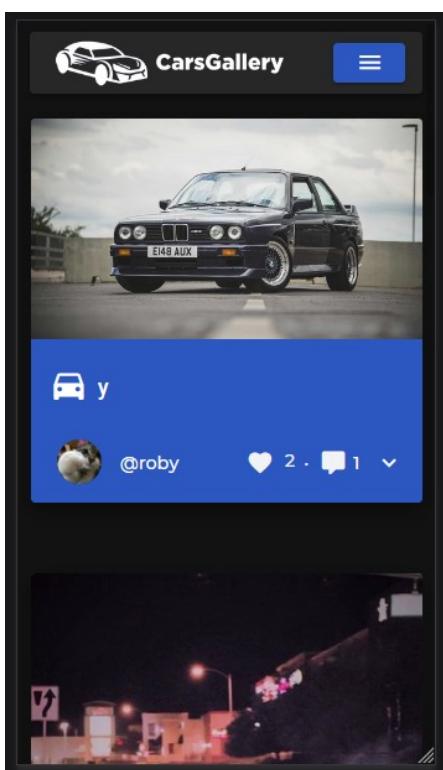


Figura 134 - Página inicial telemóvel

RELATÓRIO DO PROJETO

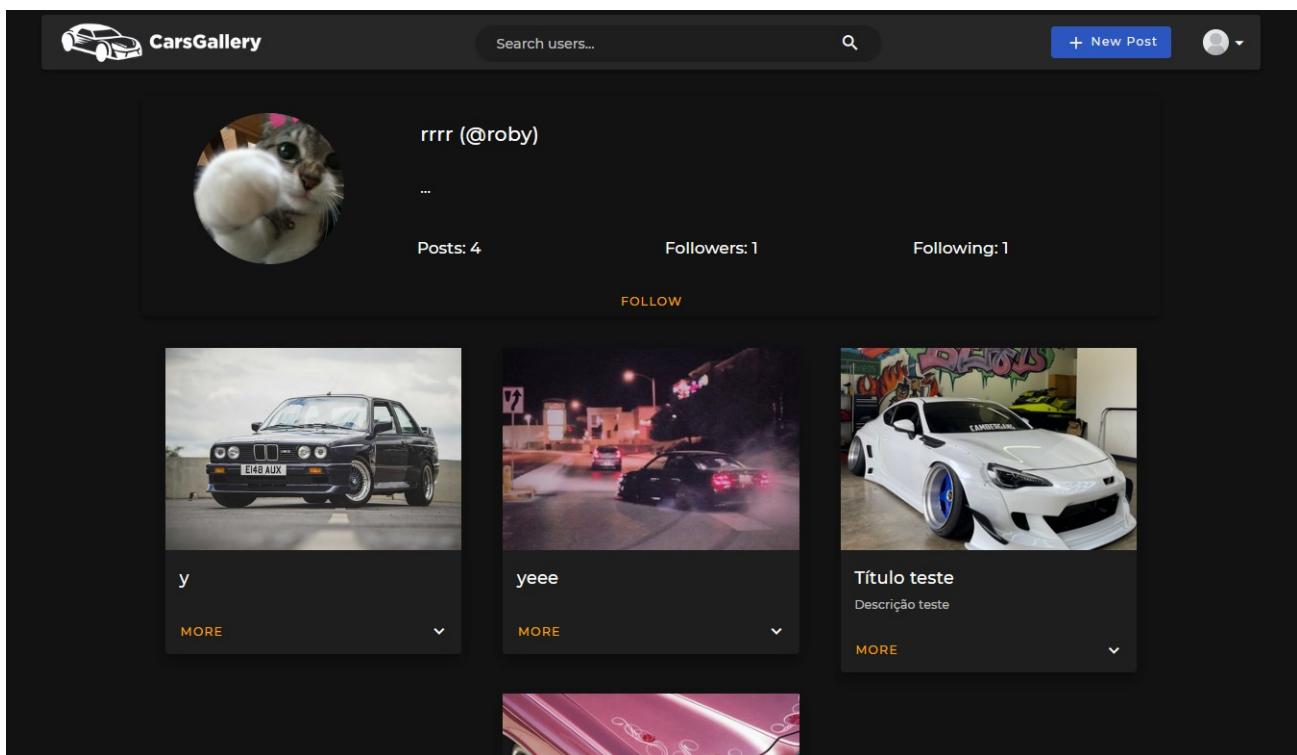


Figura 135 - Página perfil de um utilizador





RELATÓRIO DO PROJETO

Figura 136 - Página perfil de um utilizador telemóvel

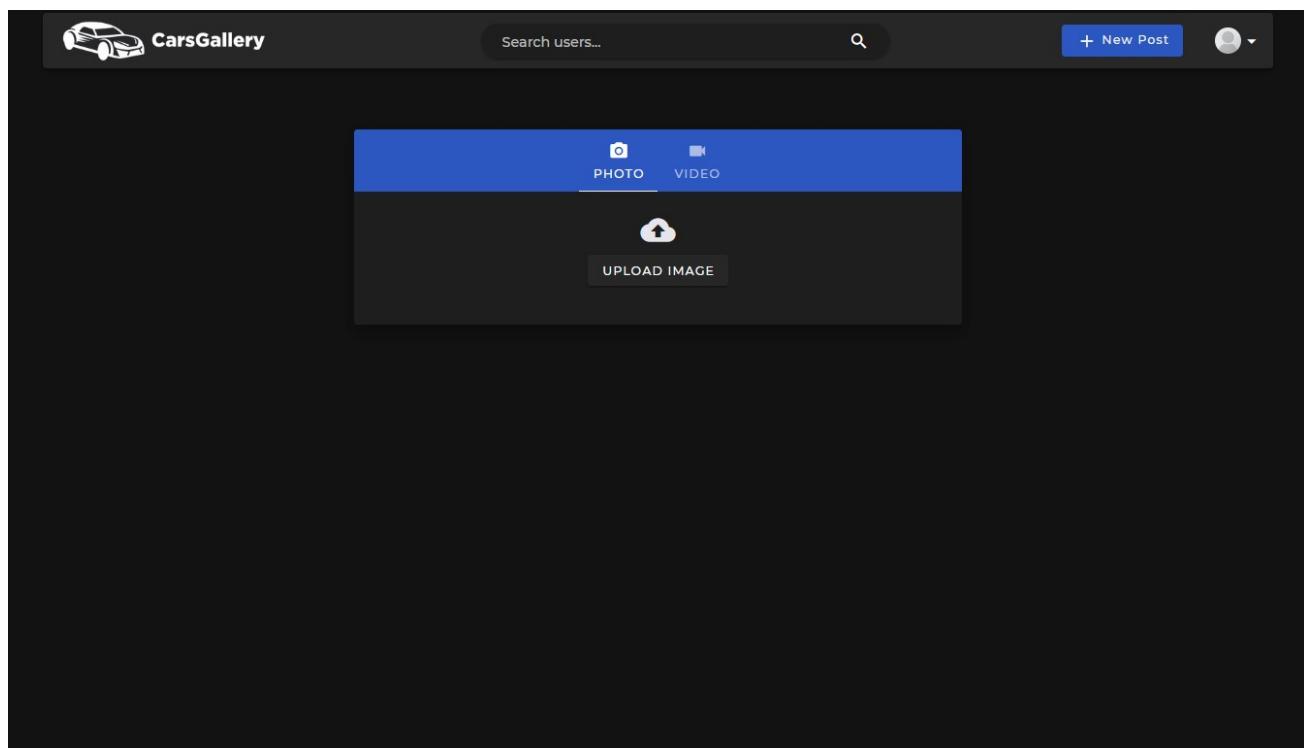
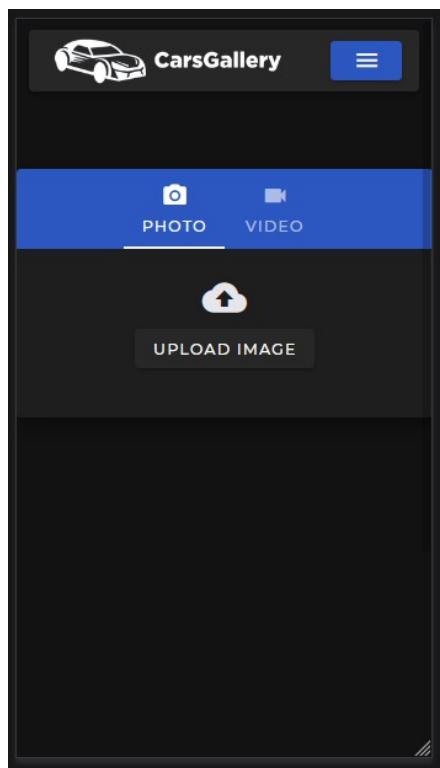


Figura 137 - Página para criação de post



RELATÓRIO DO PROJETO

Figura 138 - Página para criação de post telemóvel

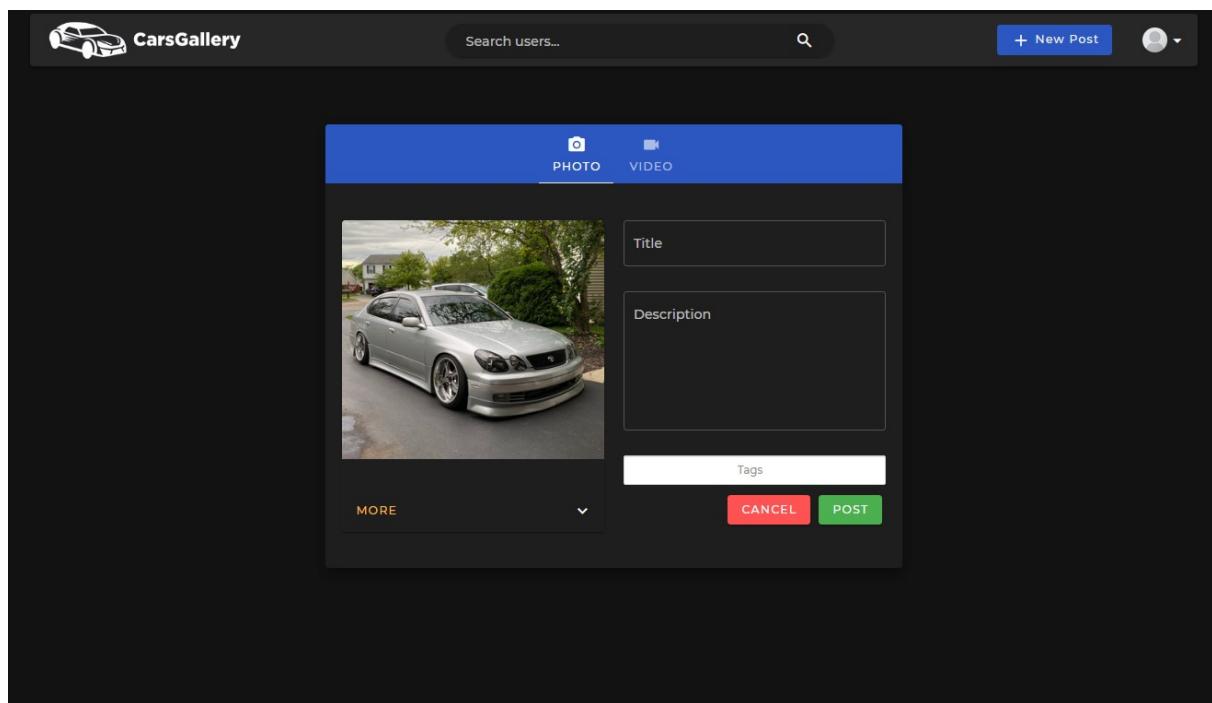


Figura 139 - Campos na criação do post

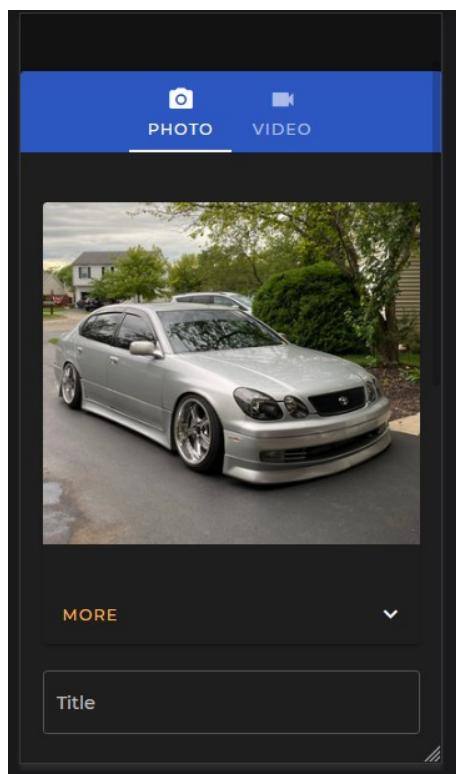


Figura 140 - Campos na criação do post

Cofinanciado por:

RELATÓRIO DO PROJETO

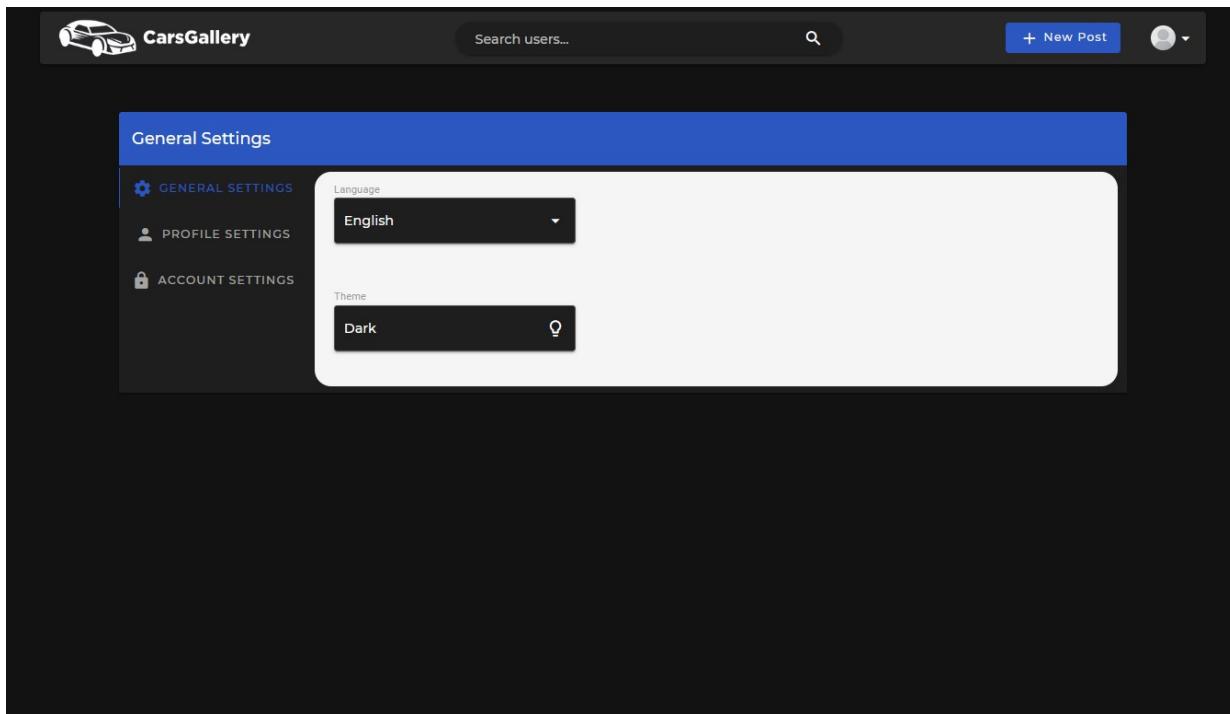


Figura 141 - Definições gerais

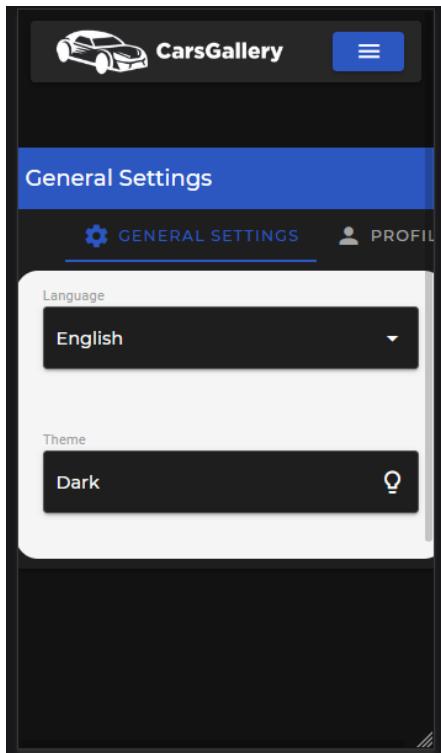


Figura 142 - Definições gerais telemóvel

RELATÓRIO DO PROJETO

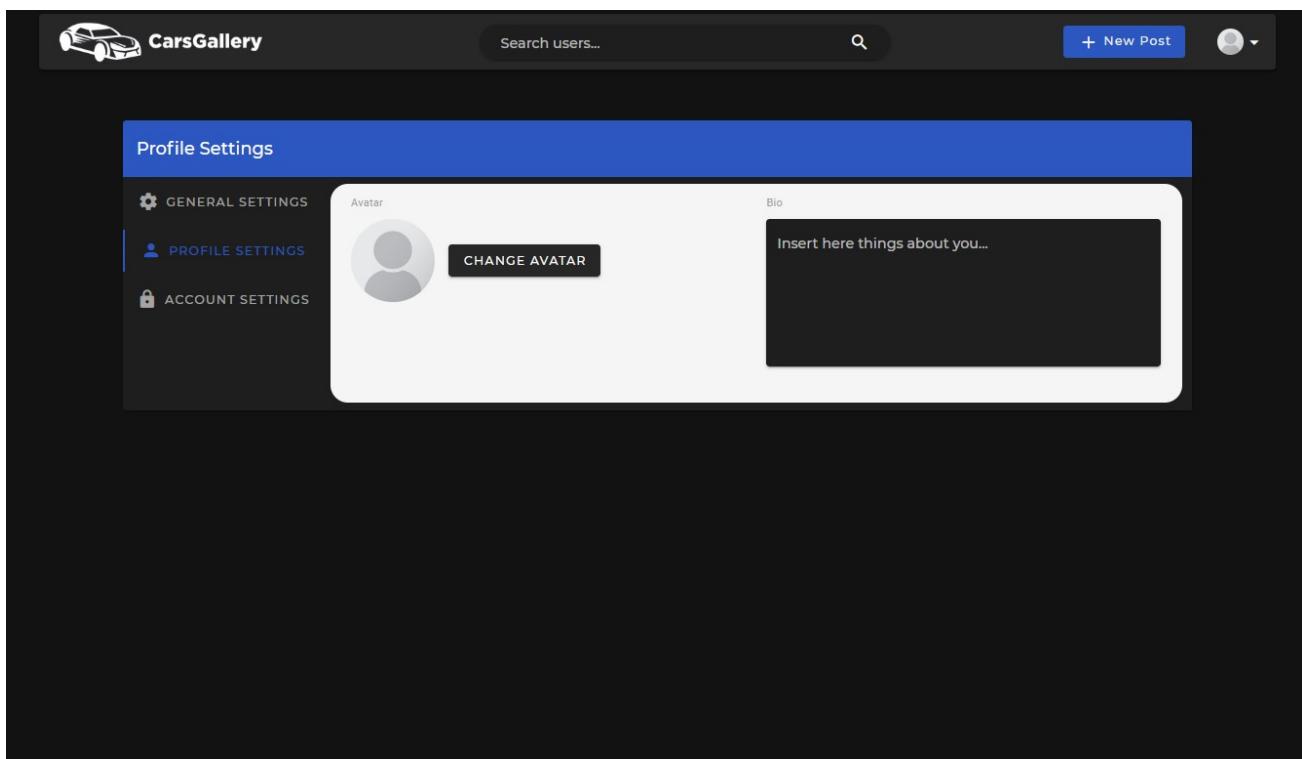


Figura 143 - Definições perfil

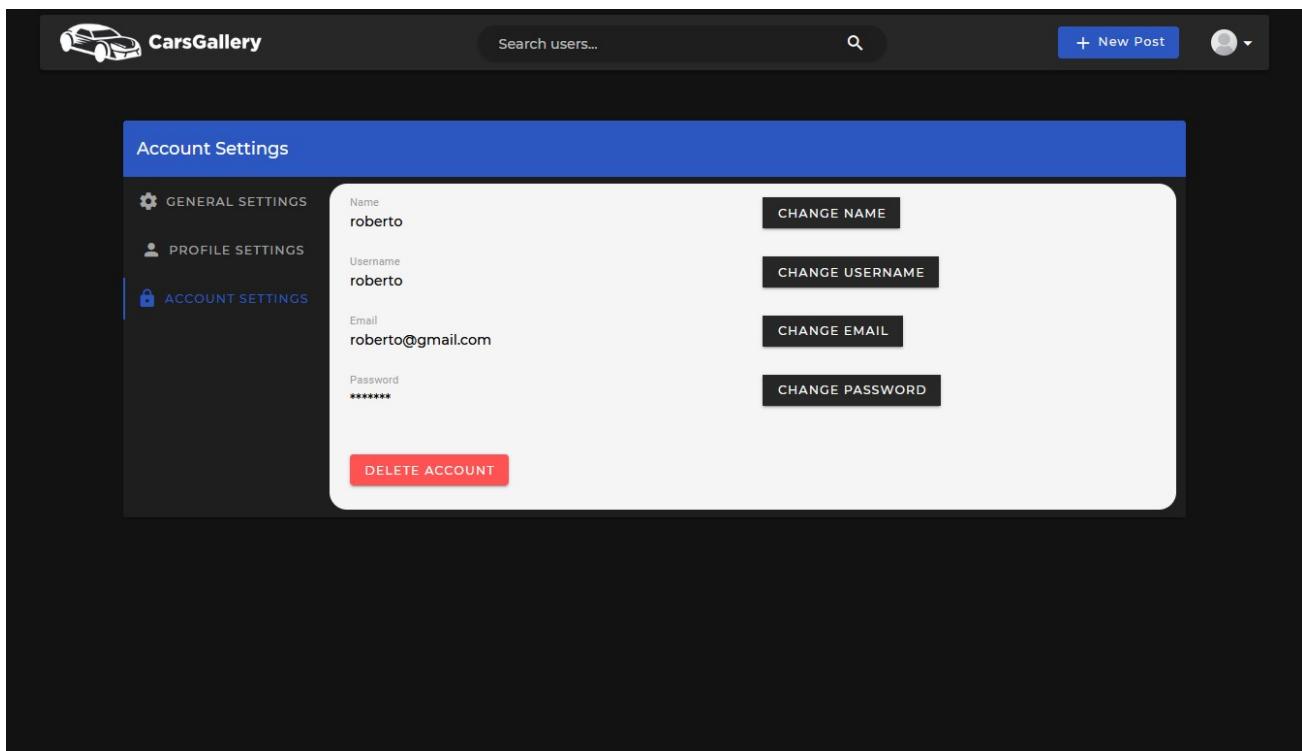


Figura 144 - Definições conta

RELATÓRIO DO PROJETO

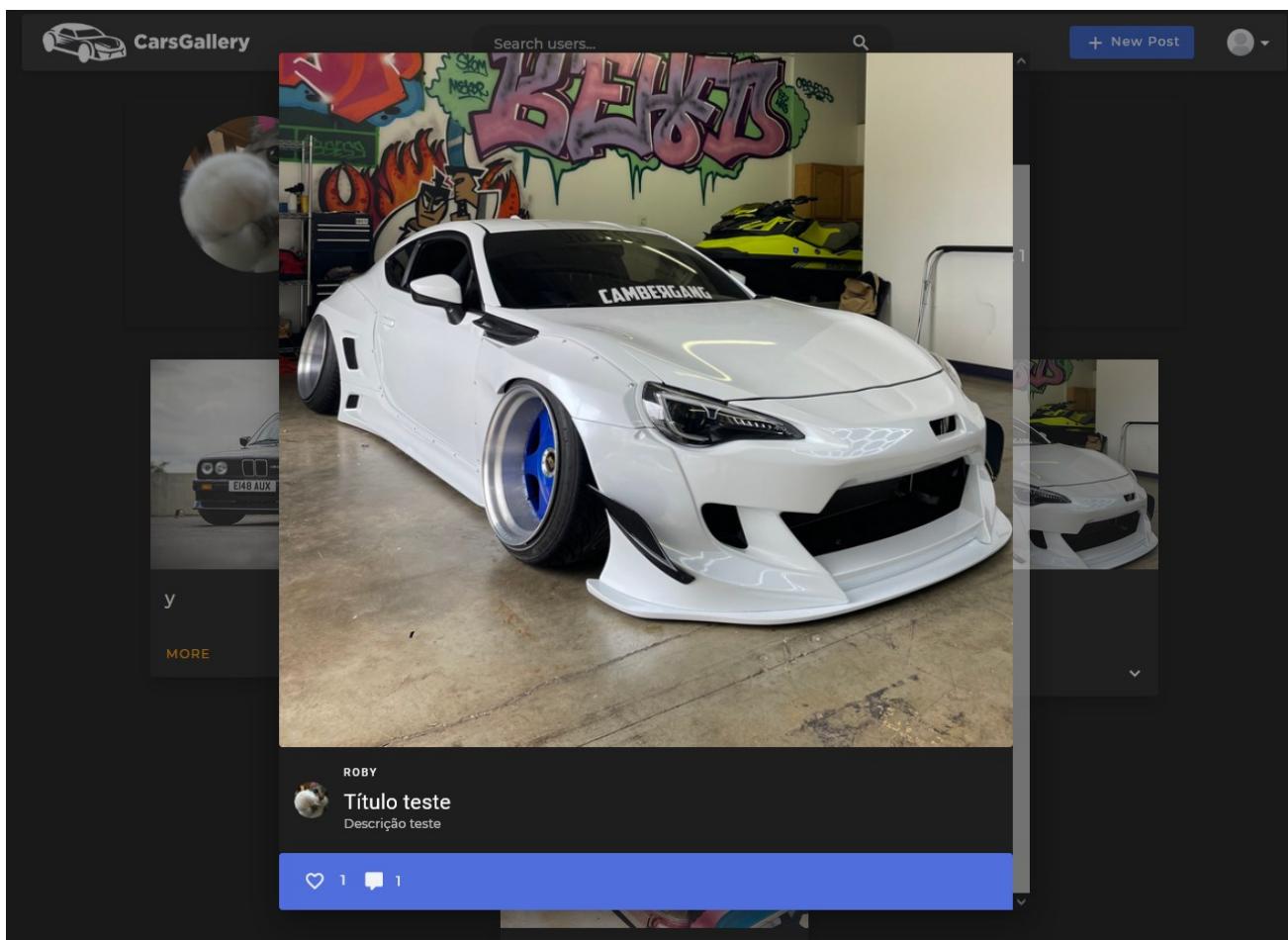


Figura 145 - Ver posts

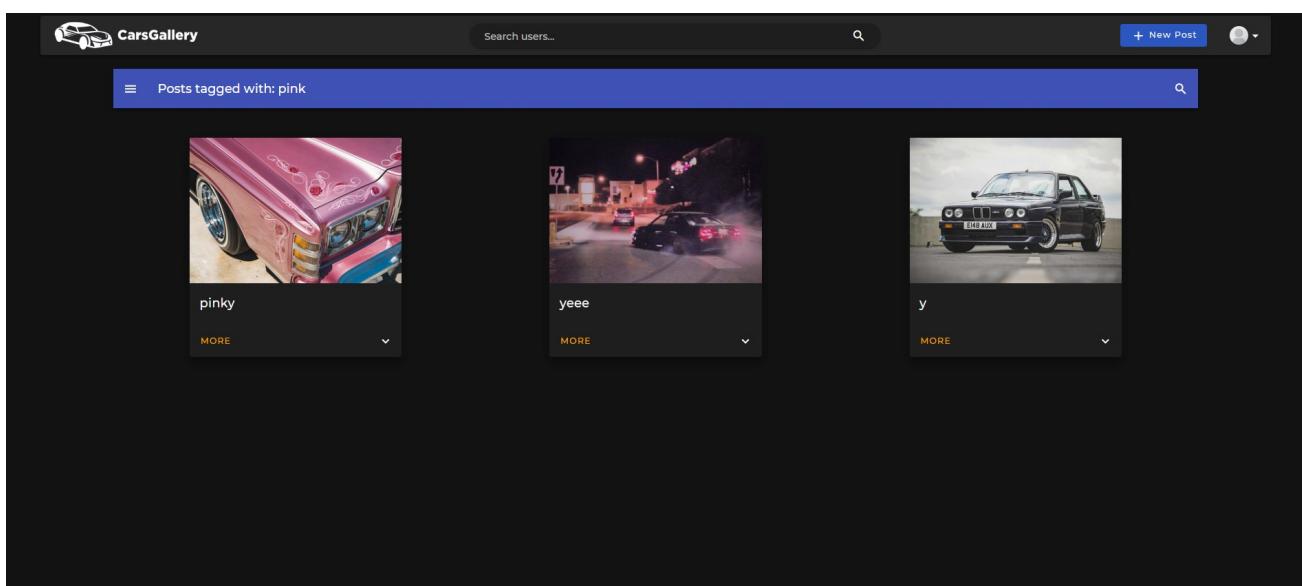


Figura 146 - Procura de posts por tag

RELATÓRIO DO PROJETO

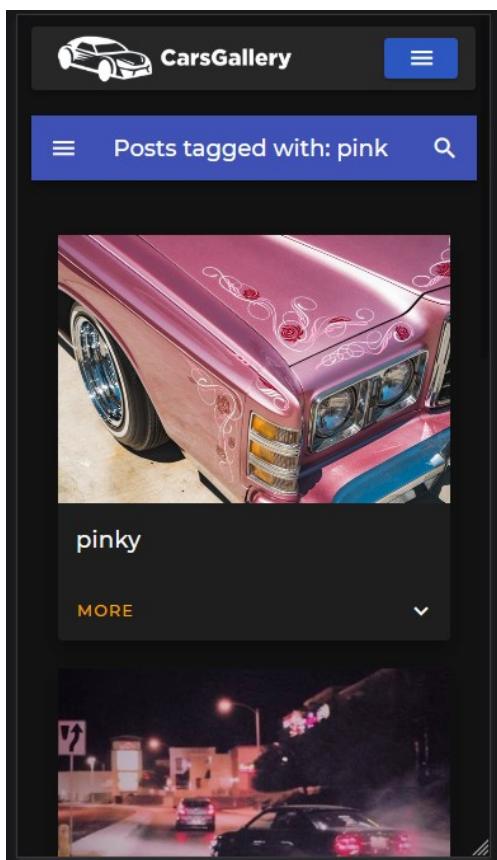


Figura 147 - Procura de posts por tags telemóvel

E agora com o tema claro:



RELATÓRIO DO PROJETO

What is
CarsGallery?

Social Network

CarsGallery is a social network developed by Roberto Petrisor as part of his final course project.

LEARN MORE

Figura 148 - Página inicial para visitantes

Figura 149 - Página login (tema claro)

Cofinanciado por:



UNIÃO EUROPEIA
Fundo Social Europeu



RELATÓRIO DO PROJETO

The screenshot shows a registration form titled 'New Account'. It includes fields for Name, Username, Email, Birthday date, Country (with a dropdown menu), and Password. There is also a link for existing users to log in and a 'REGISTER' button.

Figura 150 - Página register (tema claro)

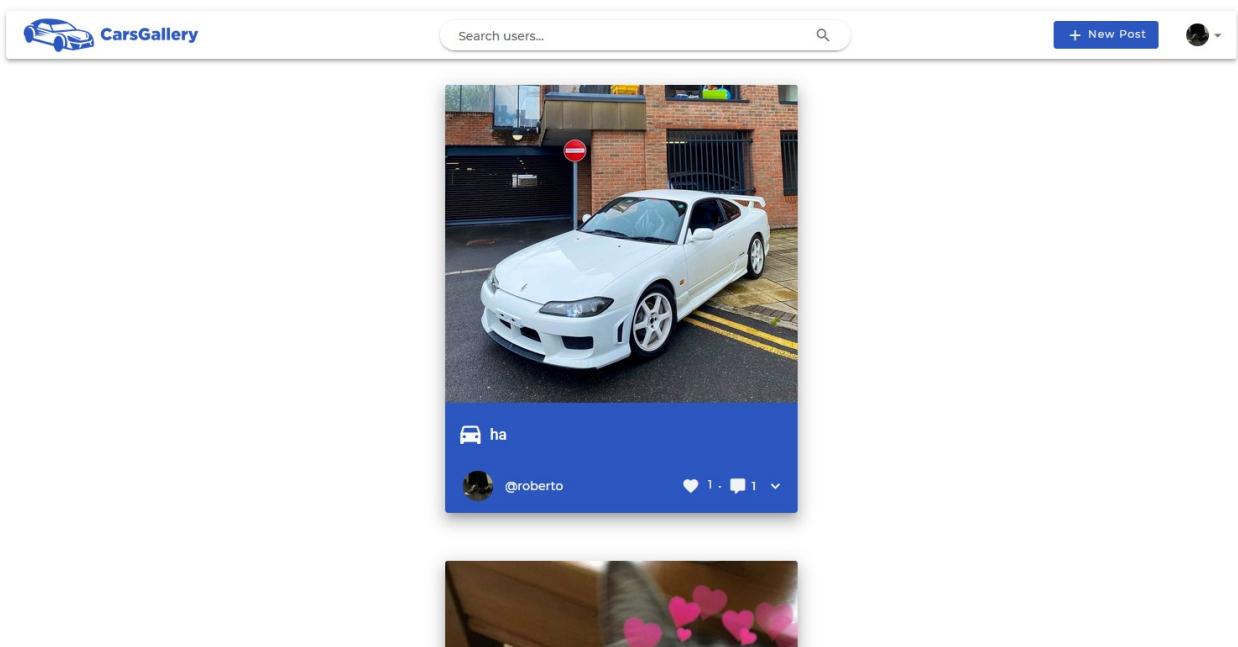


Figura 151 - Página inicial para utilizadores (tema claro)

RELATÓRIO DO PROJETO

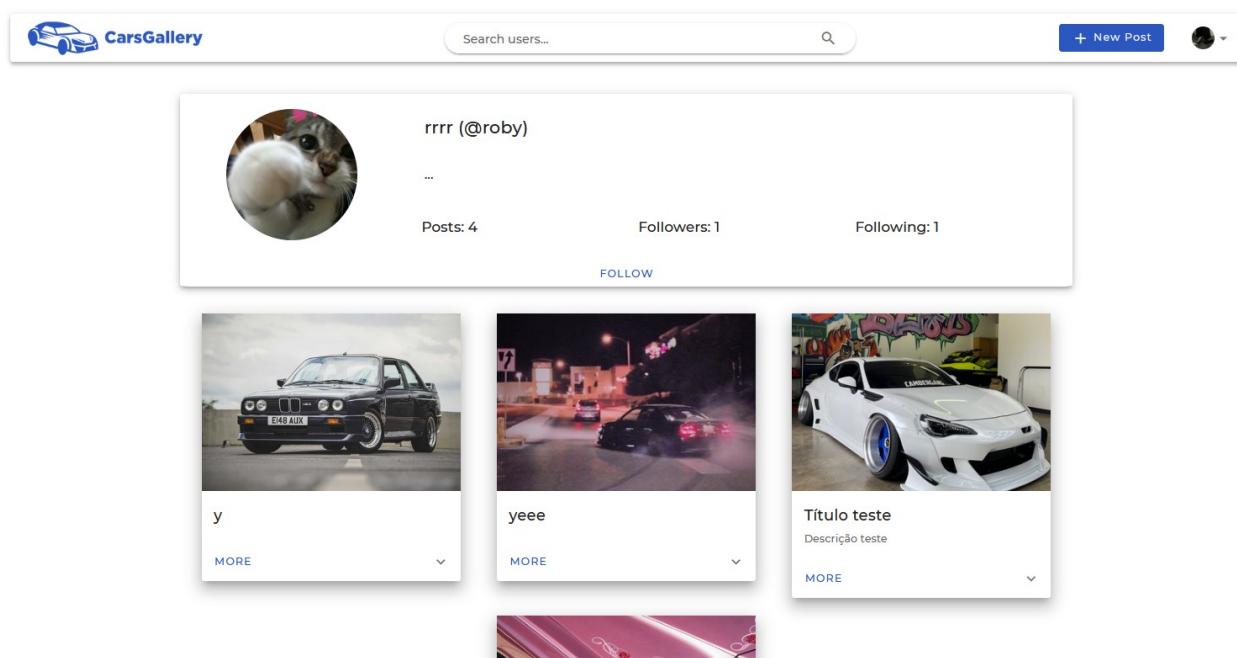


Figura 152 - Perfil de utilizador (tema claro)

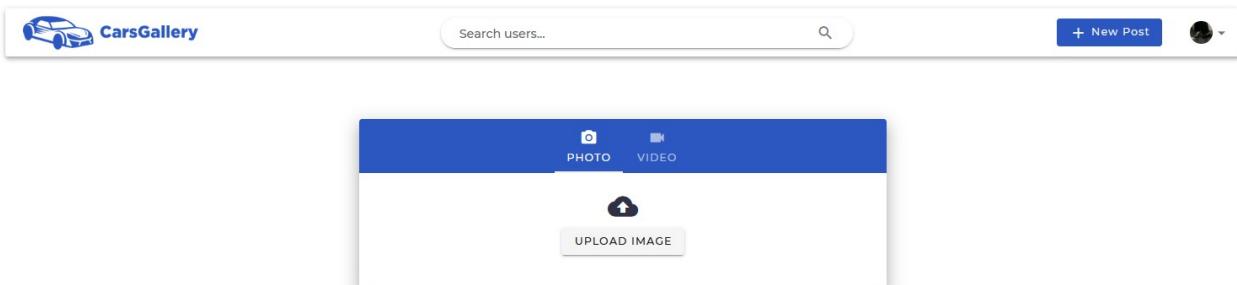


Figura 153 - Página para publicar novo post (tema claro)

RELATÓRIO DO PROJETO

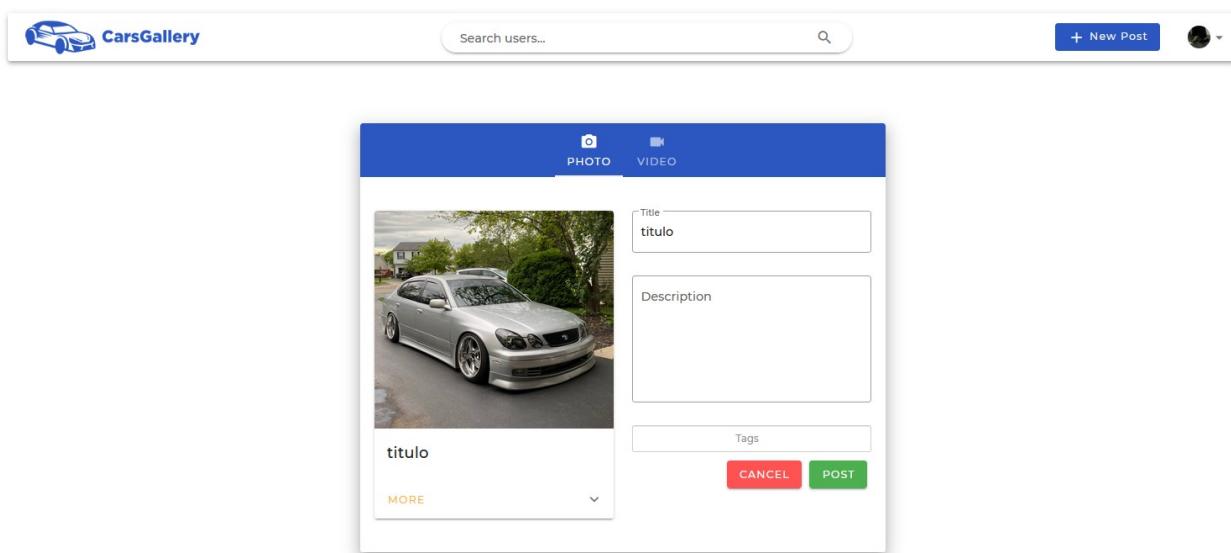


Figura 154 - Página para editar os campos do novo post (tema claro)

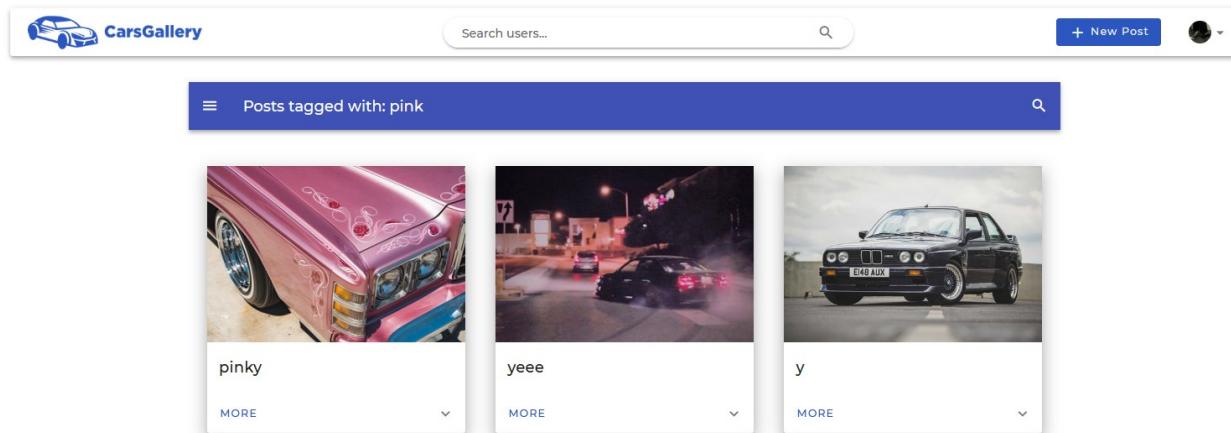


Figura 155 - Página para procura de posts por tag (tema claro)



RELATÓRIO DO PROJETO

Account Settings

GENERAL SETTINGS

PROFILE SETTINGS

ACCOUNT SETTINGS

Name: roberto

Username: roberto

Email: robby@gmail.com

Password: *****

CHANGE NAME

CHANGE USERNAME

CHANGE EMAIL

CHANGE PASSWORD

DELETE ACCOUNT

Figura 156 - Configurações da conta (tema claro)

CarsGallery

Search users...

+ New Post

Profile Settings

GENERAL SETTINGS

PROFILE SETTINGS

ACCOUNT SETTINGS

Avatar

Bio: chefe

CHANGE AVATAR

Figura 157 - Configurações do perfil (tema claro)



RELATÓRIO DO PROJETO

The screenshot shows the 'General Settings' page of a web application. At the top, there is a header bar with a car icon, the text 'CarsGallery', a search bar, and a 'New Post' button. Below the header, the main content area has a blue header bar labeled 'General Settings'. Underneath, there are three main sections: 'GENERAL SETTINGS' (with a gear icon), 'PROFILE SETTINGS' (with a person icon), and 'ACCOUNT SETTINGS' (with a lock icon). Each section contains a dropdown menu. The 'GENERAL SETTINGS' section currently has 'English' selected under 'Language' and 'Light' selected under 'Theme'.

Figura 158 - Configurações gerais (tema claro)

The screenshot shows a post on the CarsGallery application. The main image is a dark-colored sports car parked in a wet, reflective parking lot at night. A 'Comments' overlay box is centered over the image, containing a text input field with the placeholder 'Write a comment...'. Below the input field, there is a small arrow pointing to the right. At the bottom of the post, there is a user profile for 'ROBERTO mwaa' with a black circular profile picture. The post has 0 likes and 0 comments. The overall theme of the application interface is light.

Figura 159 - Comentários num post (tema claro)



RELATÓRIO DO PROJETO

Conclusão

Ao acabar este projeto, senti-me satisfeito pois consegui superar algumas dificuldades que tive no passado e acabei por adquirir mais conhecimento nesta área, mais especificamente a área web.

Tive algumas dificuldades no desenvolvimento do backend pois era tudo um bocado novo para mim, mas acabei por superar com muita pesquisa e prática.

Infelizmente o site não tem tudo que uma rede social tem, graças à falta de conhecimento e tempo, mas acho que está uma ideia bem estruturada e realizada, dando assim para adaptar este código a outras necessidades caso seja preciso.

Claro que no futuro, graças ao código ser aberto, poderão ser adicionadas novas funcionalidades, tanto por mim ou por outra pessoa que queira adaptar este projeto às suas necessidades.

Link do projeto (código): https://github.com/awyxx/pap_carsgallery



RELATÓRIO DO PROJETO

Bibliografia/Referências Bibliográficas Eletrónicas

<https://vuejs.org/v2/api/>

<http://expressjs.com/en/api.html>

<https://vuetifyjs.com/en/api/v-app/>

<https://stackoverflow.com/>

<https://www.prisma.io/docs/reference/api-reference/prisma-client-reference>

<https://www.prisma.io/docs/concepts/components/prisma-schema>

<https://sass-lang.com/documentation>

<https://materialdesignicons.com/>

<https://support.insomnia.rest/>

<https://sweetalert2.github.io/>

<https://coolors.co/>

<https://github.com/ankurk91/vue-loading-overlay>

<https://www.countryflags.io/>

<https://youtu.be/mbsmsi7l3r4>

<https://docs.github.com/pt>