Documentação Front-end

Participar da Aliança pela Mobilidade Sustentável

Ingenico do Brasil

Bahia, BA

Versão 1.0

Novembro de 2024

Identificação

Título: Documentação Front-end
Projeto: Participar da Aliança pela Mobilidade Sustentável
Data: Novembro 2024
Local: Bahia, BA
Versão: 1.0

Revisões

Data	Alterações / Comentário	Revisores
20.09.2024	Criação do documento.	João Vitor N. Ramos e Albert Silva de Jesus

Equipe de Desenvolvimento

Tutor

Marcelo Silva

Tutor

Rogério de Jesus

Líder Técnico

João Vitor Nascimento Ramos

Desenvolvedores

Albert Silva de Jesus

Danilo da Conceição Santos

Everlan Santos de Rosário

Leonardo Ribeiro Barbosa Santos

Sumário

Lista de abreviaturas e siglas

1 Introdução

1.1 Objetivo do E-Drive Navigator

O *E-Drive Navigator* é um sistema projetado para facilitar o planejamento e a navegação em viagens utilizando veículos elétricos. Ele oferece funcionalidades avançadas, como cálculo de rotas, análise dinâmica de estações de recarga e gerenciamento eficiente da autonomia do veículo. O objetivo principal do sistema é proporcionar uma experiência otimizada e confiável para os motoristas, garantindo que suas jornadas sejam realizadas de forma segura e sustentável, independentemente da distância ou das condições da viagem.

1.2 Importância do Sistema

Com o crescimento da mobilidade elétrica, a necessidade de ferramentas que auxiliem no planejamento de viagens tornou-se fundamental. O *E-Drive Navigator* atende a essa demanda, fornecendo informações precisas sobre a autonomia do veículo, estações de recarga e rotas mais eficientes. O sistema reduz a ansiedade de autonomia, melhora a eficiência energética e promove a adoção de veículos elétricos. Além disso, sua interface intuitiva e funcionalidade robusta garantem que tanto motoristas quanto empresas possam utilizar o sistema de forma eficaz.

1.3 Escopo do Documento

Este documento abrange a documentação completa do *front-end* do sistema *E-Drive Navigator*. Ele inclui informações detalhadas sobre a estrutura do código, componentes principais, serviços implementados, integração com *APIs* externas e diretrizes para personalização e manutenção do sistema. O escopo também aborda as práticas recomendadas de desenvolvimento e documentação, garantindo que a equipe tenha uma base sólida para colaborar e evoluir o sistema ao longo do tempo.

2 Estrutura

2.1 Arquitetura do Sistema

A Figura 1 ilustra a estrutura global de arquivos do front-end do sistema *E-Drive Navigator*, organizada desde a raiz do projeto. Essa organização segue práticas modernas de desenvolvimento, visando modularidade, reutilização de código e manutenção simplificada.

A estrutura destaca as principais pastas e arquivos:

Figura 1 – Estrutura global de arquivos do front-end

2.1.1 Descrição das pastas e arquivos principais

- *.angular*: Diretório utilizado para armazenar o cache do Angular CLI, incluindo dados relacionados à compilação incremental e otimização. Este cache acelera o processo de build e serve para melhorar o desempenho em desenvolvimentos subsequentes.
- *node_modules*: Pasta onde ficam armazenadas todas as dependências do projeto, instaladas pelo gerenciador de pacotes (como npm ou Yarn). Ela contém os pacotes necessários para que o projeto funcione corretamente, incluindo bibliotecas e frameworks de terceiros.

- *src*: Diretório principal onde o código-fonte do projeto está localizado. Contém os arquivos do aplicativo, incluindo componentes, serviços, modelos, estilos e outros elementos fundamentais para o desenvolvimento do sistema.
- *angular.json*: Arquivo de configuração principal do Angular CLI. Ele define a estrutura do projeto, scripts de build, configurações de servidor de desenvolvimento e opções de compilação.
- *package.json*: Arquivo que define as dependências do projeto, scripts de execução (como `npm start` e `npm build`), informações do projeto (como nome e versão) e outras configurações essenciais.
- *package-lock.json*: Arquivo gerado automaticamente pelo npm para garantir a consistência das dependências instaladas. Ele registra as versões exatas das bibliotecas no momento da instalação, garantindo que o projeto funcione da mesma maneira em diferentes máquinas.
- *proxy.config.json*: Arquivo de configuração utilizado para redirecionar requisições da aplicação para diferentes endpoints, como a API do Google Maps ou o backend. Isso é útil para evitar problemas de CORS durante o desenvolvimento.
- *pest.config.ts*: Arquivo de configuração para o framework de testes Jest. Ele define como os testes devem ser executados e configurados no ambiente do projeto.
- *setup-jest.ts*: Arquivo de configuração adicional para o Jest, utilizado para definir comportamentos ou inicializações antes da execução dos testes, como mock de módulos ou configurações de ambiente.
- *tsconfig.json*: Arquivo principal de configuração do TypeScript no projeto. Ele define opções globais, como versão da linguagem, paths e configurações de compilação.
- *tsconfig.app.json*: Arquivo de configuração específico para o código do aplicativo. Ele é usado para compilar apenas os arquivos do aplicativo em si, excluindo testes ou outras partes.
- *tsconfig.spec.json*: Arquivo de configuração usado exclusivamente para compilar e executar os arquivos de teste no projeto.

2.1.2 Diretório `src`

O diretório `src` contém os principais arquivos e pastas responsáveis pelo desenvolvimento e funcionamento do front-end do sistema *E-Drive Navigator*. A Figura 2 ilustra a estrutura interna desse diretório.

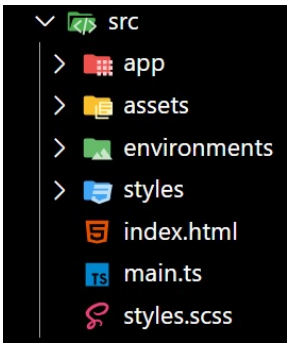

Figura 2 – Estrutura global de arquivos do diretório `src`

2.1.3 Descrição das pastas e arquivos principais

- *app*: É a pasta principal que contém todo o código relacionado à lógica do aplicativo. Inclui os componentes, serviços, módulos, diretivas e outros elementos essenciais. Sua organização reflete a modularidade do sistema, facilitando a manutenção e a escalabilidade.
- *assets*: Diretório destinado a armazenar recursos estáticos, como imagens, ícones, fontes e outros arquivos que não fazem parte do código. Por exemplo, as imagens utilizadas no sistema são mantidas nessa pasta.
- *environments*: Contém arquivos de configuração de ambiente, como `environment.ts` (usado no desenvolvimento) e `environment.prod.ts` (usado em produção). Esses arquivos armazenam variáveis como as chaves da API do Google Maps, URLs para comunicação com o backend e certificados de segurança, como `server.crt`, `server.key` e `server.p12`, utilizados para conexões seguras entre o front-end e os serviços externos.
- *styles*: Pasta que armazena arquivos de estilo global do projeto. Inclui CSS ou SCSS utilizados em todo o sistema, garantindo consistência visual entre as páginas.
- *index.html*: Arquivo HTML principal do projeto. Ele serve como ponto de entrada para o aplicativo Angular, onde os scripts e estilos compilados são injetados automaticamente durante o processo de build.

- *main.ts*: Arquivo TypeScript que funciona como o ponto de partida do aplicativo Angular. Ele inicializa o módulo raiz (`AppModule`) e configura a execução da aplicação.
- *styles.scss*: Arquivo de estilo global, escrito em SCSS (uma extensão do CSS). Ele permite definir estilos que se aplicam a toda a aplicação, oferecendo maior flexibilidade e organização ao código de estilo.

2.1.4 app

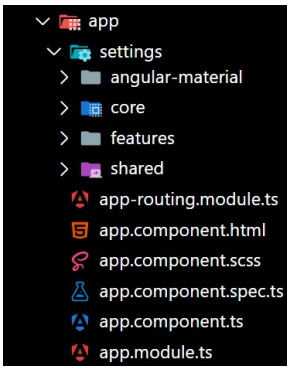

Figura 3 – Estrutura global de arquivos do diretório src/app

O diretório src/app é o coração do aplicativo Angular, contendo os principais arquivos e pastas que estruturam a lógica e o funcionamento do sistema *E-Drive Navigator*.

2.1.4.0.1 Arquivos principais

- *app.component.html*: Arquivo de template HTML principal do aplicativo, responsável por renderizar o layout base da aplicação.
- *app-routing.module.ts*: Arquivo de configuração das rotas do aplicativo. Ele define o mapeamento entre URLs e componentes, permitindo a navegação entre as diferentes páginas.
- *app.component.scss*: Arquivo de estilo específico do AppComponent, utilizado para estilizar o layout principal.
- *app.component.spec.ts*: Arquivo de teste automatizado para o AppComponent, garantindo o funcionamento correto de sua lógica.
- *app.component.ts*: Arquivo TypeScript que implementa a lógica do AppComponent, o componente raiz do aplicativo.

- *app.module.ts*: Arquivo principal de configuração do Angular. Ele define o AppModule, importando e declarando os módulos e componentes necessários para o funcionamento do sistema.

2.1.4.0.2 Estrutura das pastas

- *settings*: Pasta que contém os arquivos e módulos principais do projeto. Dentro dela, estão organizadas outras pastas importantes:
  - *angular-material*: Diretório dedicado à configuração dos componentes do Angular Material. Contém um módulo específico para importar e exportar todos os componentes necessários da biblioteca Angular Material.
  - *core*: Diretório que armazena serviços, interceptores, e funcionalidades globais do sistema, como autenticação, autorização e configuração de comunicação com o backend.
  - *features*: Pasta que agrupa os módulos e componentes específicos das funcionalidades do sistema, como cadastro de usuários, planejamento de rotas, entre outros.
  - *shared*: Diretório utilizado para armazenar componentes, pipes e diretivas reutilizáveis em diferentes partes do sistema, promovendo a reutilização de código e consistência no projeto.

2.1.4.1 core

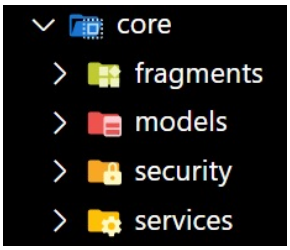

Figura 4 – Estrutura global de arquivos do diretório src/app/core

2.1.4.1.1 fragments

Dentro da pasta *core*, encontra-se a pasta *fragments*, que organiza componentes específicos utilizados em diferentes partes do aplicativo:

- *BottomBarComponent*: Componente responsável pela barra inferior da interface do usuário, visível para o usuário logado. Ele fornece opções de navegação ou informações rápidas, como acesso ao perfil ou configurações.
- *FaqPopupComponent*: Componente que exibe o pop-up com perguntas frequentes (FAQ), ajudando os usuários a encontrar respostas rápidas para dúvidas comuns no sistema.
- *FooterComponent*: Componente que renderiza o rodapé da tela principal do aplicativo, geralmente contendo informações de copyright, links úteis e outros dados institucionais.
- *NavbarIntroComponent*: Componente que exibe a barra de navegação para páginas de introdução, como páginas de registro, login e informações sobre os serviços do sistema, facilitando o acesso a essas funcionalidades para os usuários.

2.1.4.1.2 models

Na pasta *models*, ficam as interfaces necessárias para a construção dos objetos utilizados no sistema. As interfaces presentes são as seguintes:

- *api-response.ts*: Interface que define a estrutura da resposta da API.
- *autonomy.ts*: Interface relacionada à autonomia do veículo.
- *brand.ts*: Interface que define as informações sobre as marcas de veículos.
- *category.ts*: Interface que descreve as categorias dos veículos.
- *DataCategoryAvgAutonomyStats.ts*: Interface para armazenar as estatísticas de autonomia média por categoria.
- *geocoding-response.ts*: Interface que define a estrutura da resposta de geocodificação.
- *inter-Address.ts*: Interface para representar o endereço de um local.
- *inter-Login.ts*: Interface para os dados de login do usuário.
- *model.ts*: Interface genérica para representar um modelo no sistema.
- *pageable.ts*: Interface para representar parâmetros de paginação.

- *paginatedResponse.ts*: Interface que define a resposta paginada de uma consulta.
- *propulsion.ts*: Interface relacionada à propulsão do veículo.
- *role.ts*: Interface que define os diferentes papéis de usuário no sistema.
- *step.ts*: Interface que descreve os passos em um plano de viagem.
- *user.ts*: Interface que representa os dados do usuário no sistema.
- *user-vehicle.ts*: Interface para representar o vínculo entre o usuário e o veículo.
- *vehicle.ts*: Interface que define as informações do veículo.
- *vehicle-type.ts*: Interface para categorizar os tipos de veículos.
- *vehicle-with-user-vehicle.ts*: Interface que combina as informações de veículo com o vínculo do usuário.

2.1.4.1.3 security

Na pasta *security*, concentram-se várias funcionalidades de segurança e autenticação do sistema, organizadas nas seguintes pastas e arquivos:

1. *guards*: Contém os guards, que são responsáveis por proteger rotas no sistema. Eles verificam se o usuário tem permissão para acessar determinadas páginas, com base em sua autenticação ou autorização. Nela se concentram os seguintes componentes:
    - *CanMatchGuard*: Guarda que verifica se o usuário tem permissão para acessar determinadas rotas com base em suas credenciais e autorização.
    - *AuthGuard*: Guarda responsável por verificar a autenticação do usuário antes de permitir o acesso às rotas protegidas.
2. *interceptors*: Armazena os interceptores, que permitem interceptar e modificar requisições ou respostas HTTP. São utilizados para adicionar cabeçalhos de autenticação, tratamento de erros, entre outras funções, garantindo a segurança e a integridade das comunicações do sistema.

3. *login*: Contém componentes relacionados ao processo de login do usuário, como o controle de sessão, autenticação e validação de credenciais. A pasta é dividida em duas subpastas:

    - *user-login*: Contém o componente *UserLoginComponent*, responsável pela interface de login e autenticação do usuário.
    - *recovery-password*: Contém os seguintes componentes relacionados à recuperação de senha:
        - *ModalRecoveryPasswordComponent*: Exibe um modal para iniciar o processo de recuperação de senha.
        - *ResetPasswordComponent*: Componente para permitir que o usuário redefina sua senha.
        - *ConfirmAccount*: Componente para confirmar a conta do usuário, caso necessário.

    Esses componentes são englobados no *LoginModule*, que gerencia toda a lógica e a funcionalidade do processo de login e recuperação de senha.

4. *services*: Na pasta *services*, encontram-se os diversos serviços responsáveis pela comunicação entre o front-end e as funcionalidades do sistema, organizados por domínio ou funcionalidade específica. Cada serviço geralmente contém dois arquivos: um arquivo de implementação (`*.service.ts`) e um arquivo de testes (`*.service.spec.ts`). Abaixo estão os principais serviços organizados por categoria:

    - *Address*: Contém o serviço `address.service.ts` para gerenciar operações relacionadas a endereços.
    - *Alertas*: Contém o serviço `alertas.service.ts`, que gerencia os alertas do sistema.
    - *Apis*: Agrupa serviços para interação com APIs externas, incluindo:
        - *Country*: `country.service.ts`, para gerenciar operações relacionadas a países.
        - *Geocoding*: `geocoding.service.ts`, para conversão de endereços em coordenadas geográficas.
        - *Location*: `location.service.ts`, para gerenciar a localização do usuário.
        - *Postal Code*: `postal-code.service.ts`, para operações relacionadas a códigos postais.

- *Brand*: `brand.service.ts`, que gerencia informações sobre marcas de veículos.
- *Category*: `category.service.ts`, para manipulação das categorias de veículos.
- *Category Avg Autonomy Stats*: `category-avg-autonomy-stats.service.ts`, que trata da média de autonomia por categoria.
- *Map*: `map.service.ts`, responsável pela integração com mapas.
- *Modal*: `modal.service.ts`, para exibição de modais no sistema.
- *Model*: `model.service.ts`, para manipulação de modelos de veículos.
- *Propulsion*: `propusion.service.ts`, que gerencia os dados de propulsão dos veículos.
- *Trip Planner Maps*: `trip-planner-maps.service.ts`, serviço para planejar rotas no mapa.
- *Type Vehicle*: `type-vehicle.service.ts`, para manipulação de tipos de veículos.
- *User*: Organizado em subpastas para funcionalidades específicas de usuários:
  - *Userdata*: `user-data.service.ts`, para gerenciar dados do usuário.
  - *User*: `user.service.ts`, serviço principal para manipulação de informações do usuário.
  - *User Vehicle*: `user-vehicle.service.ts`, para gerenciar os veículos associados ao usuário.
- *Vehicle*: `vehicle.service.ts`, para gerenciar as operações relacionadas aos veículos.
- *Version*: `version.service.ts`, para gerenciar as informações sobre a versão do sistema.

2.1.4.2 `features`

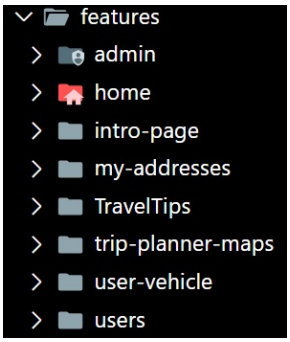

Figura 5 – Estrutura global de arquivos do diretório `src/app/feature`

Este diretório contém diversas funcionalidades do aplicativo, divididas em subdiretórios responsáveis por diferentes módulos e componentes do sistema.

2.1.4.2.1 Admin

O diretório `admin` contém os módulos relacionados à administração do sistema, com subdiretórios para os recursos de `brand`, `model` e `vehicle`. Cada um desses subdiretórios tem componentes específicos para lidar com as operações de criação, edição e exibição desses recursos.

- *admin.module.ts*: Define o módulo principal de administração.
- *admin-routing.module.ts*: Contém a configuração de rotas para a área administrativa.
- *brand/components*: Contém os componentes para exibir e editar marcas, como `brand-list.component.ts`, `modal-details-brand.component.ts`, e `modal-form-brand.component.ts`.
- *model/components*: Contém os componentes para exibir e editar modelos de veículos, como `modal-details-model.component.ts`, `modal-form-model.component.ts`, e `model-list.component.ts`.
- *vehicle/components*: Contém os componentes para exibir e editar veículos, como `modal-details-vehicle.component.ts`, `modal-form-vehicle.component.ts`, e `vehicle-list.component.ts`.

2.1.4.2.2 Home

O diretório `home` é responsável pelos componentes relacionados à página inicial e à navegação do sistema. Ele contém dois componentes principais: o `dashboard` e o `navbar`.

- *home.module.ts*: Define o módulo principal da página inicial.
- *home-routing.module.ts*: Contém a configuração de rotas para a página inicial.
- *components/dashboard*: Contém o componente do dashboard (`dashboard.component.ts`), que exibe informações de resumo para o usuário.
- *components/navbar*: Contém o componente da barra de navegação (`navbar.component.ts`), que permite a navegação entre as páginas do sistema.

2.1.4.2.3 Intro-page

O diretório `intro-page` contém os componentes relacionados à página de introdução do sistema, que pode ser usada como uma tela de boas-vindas ou de login.

- *intro-page.component.ts*: Define o comportamento do componente da página de introdução.
- *intro-page.component.html*: Define o layout da página de introdução.
- *intro-page.component.scss*: Contém os estilos específicos para a página de introdução.
- *module/intro-page.module.ts*: Define o módulo da página de introdução, incluindo a configuração de rotas.

2.1.4.2.4 My-address

O diretório `my-addresses` é responsável pelos componentes que permitem ao usuário gerenciar seus endereços, exibindo uma lista de endereços, detalhes e permitindo a edição ou criação de novos.

- *my-addresses.module.ts*: Define o módulo principal de gerenciamento de endereços.
- *my-addresses-routing.module.ts*: Contém a configuração de rotas para a funcionalidade de endereços.
- *components/list-my-addresses*: Contém o componente que exibe a lista de endereços (`list-my-addresses.component.ts`).
- *components/modal-details-address*: Contém o componente para exibir os detalhes de um endereço (`modal-details-address.component.ts`).
- *components/modal-form-my-addresses*: Contém o componente para editar ou adicionar um novo endereço (`modal-form-my-addresses.component.ts`).

2.1.4.2.5 Trip-planner

O diretório `trip-planner` é responsável pelos componentes e funcionalidades relacionadas ao planejamento de viagens dentro do sistema. Ele inclui módulos e componentes que permitem ao usuário traçar rotas, visualizar as etapas da viagem, e verificar informações sobre os pontos de parada, como estações de carregamento.

- *trip-planner.module.ts*: Define o módulo principal do planejador de viagens.
- *trip-planner-routing.module.ts*: Contém a configuração de rotas para as páginas do planejador de viagens.
- *components/route-steps*: Contém o componente que exibe as etapas da viagem (`route-steps.component.ts`).
- *components/charging-stations*: Contém o componente para exibir as estações de carregamento ao longo do caminho (`charging-stations.component.ts`).
- *components/trip-summary*: Contém o componente de resumo da viagem (`trip-summary.component.ts`), que exibe um resumo das etapas, distâncias e estações de carregamento.
- *services/trip-planner.service.ts*: Contém o serviço que lida com a lógica de planejamento de rotas e cálculo de paradas em estações de carregamento.

2.1.4.2.6 trip-planner-maps

O diretório `trip-planner-maps` é responsável pelo planejamento da viagem, focado em estações de recarga e planejamento de rotas. Ele contém os seguintes componentes e módulos:

- *map-stations*: Contém os componentes responsáveis por exibir as estações de recarga no mapa e gerenciar modais relacionados. Inclui:
  - *map-stations.component.ts*: Componente principal para exibir as estações no mapa.
  - *scss*: Contém os arquivos de estilo específicos, como `map-stations.details-modal.scss`, `map-stations-map.scss`, e `map-stations.modal.scss`.
- *modal-form-vehicle-battery*: Componente para o formulário de baterias dos veículos. Contém:
  - *modal-form-vehicle-battery.component.ts*: Componente para o formulário de veículo e bateria.
  - *modal-form-vehicle-battery.component.scss*: Estilos específicos para o formulário de bateria.
- *modal-select-address*: Componente para selecionar o endereço de partida e destino. Contém:
  - *modal-select-address.component.ts*: Componente para seleção de endereço.
- *planning-trip*: Componente para gerenciar o planejamento da viagem, mostrando as etapas do trajeto e as estações de recarga ao longo do caminho. Contém:
  - *planning-trip.component.ts*: Componente para planejar a viagem.
- *map-stations.module.ts*: Define o módulo principal do `trip-planner-maps`.
- *map-stations-routing.module.ts*: Configuração de rotas para o módulo `trip-planner-maps`.

2.1.4.2.7 users

O diretório `users` é responsável pelas funcionalidades relacionadas ao gerenciamento de usuários, como a atualização de perfil e alteração de senha. Ele contém:

- *user-password-modal*: Componente para a alteração de senha do usuário.
- *user-perfil*: Componente para exibição e edição do perfil do usuário.
- *user-registration-form*: Componente para o registro de novos usuários.
- *user-update*: Componente para a atualização de dados do usuário.
- *users.module.ts*: Define o módulo principal para o gerenciamento de usuários.
- *users-routing.module.ts*: Contém a configuração das rotas para a área de usuários.

2.1.4.2.8 user-vehicle

O diretório `user-vehicle` contém os componentes responsáveis pela gestão dos veículos dos usuários, incluindo a exibição e a edição dos veículos registrados.

- *modal-details-vehicle*: Componente para exibir os detalhes de um veículo.
- *modal-form-vehicle*: Componente para editar ou adicionar um novo veículo.
- *user-vehicle-list*: Componente para exibir a lista de veículos do usuário.
- *user-vehicle.module.ts*: Define o módulo principal para o gerenciamento de veículos dos usuários.
- *user-vehicle-routing.module.ts*: Contém a configuração de rotas para o gerenciamento de veículos dos usuários.

2.1.4.3 `shared`

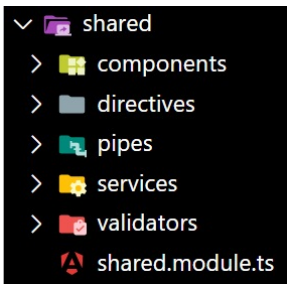

Figura 6 – Estrutura global de arquivos do diretório `src/app/shared`

O diretório `shared` contém recursos reutilizáveis em toda a aplicação, organizados em subdiretórios conforme sua funcionalidade. A seguir, apresenta-se a estrutura resumida, destacando apenas os arquivos principais de cada pasta:

- `components`: Contém componentes reutilizáveis na aplicação.
  - `lgpd-modal/lgpd-modal.component.ts`: Componente para exibição de um modal relacionado à LGPD.
  - `ui-button/ui-button.component.ts`: Componente para botões reutilizáveis.
- `directives`: Contém diretivas personalizadas.
  - `dynamic-mask.directive.ts`: Diretiva para aplicação de máscaras dinâmicas em campos de entrada.
  - `email-pattern-validator.directive.ts`: Diretiva para validação de padrões de e-mail.
- `pipes`: Inclui pipes reutilizáveis.
  - `phone-mask.pipe.ts`: Pipe para formatação de números de telefone.
- `services`: Contém serviços que oferecem utilitários compartilhados.
  - `FormUtils/form-utils.service.ts`: Serviço para manipulação de utilitários relacionados a formulários.
- `validators`: Inclui validadores personalizados para formulários.
  - `confirm-password.validators.ts`: Validador para verificação de confirmação de senha.
  - `country-code.validators.ts`: Validador para códigos de país.

    - `email-exists.validator.ts`: Validador para verificação de e-mails já cadastrados.
    - `future-year-validator.ts`: Validador para anos futuros.
    - `generic-text-validator.ts`: Validador para texto genérico.
    - `no-numbers.validator.ts`: Validador para evitar números em entradas de texto.
    - `number-validator.ts`: Validador para entradas numéricas.
    - `password-field.validator.ts`: Validador para campos de senha.
    - `password-visibility-toggle.ts`: Lógica para alternar a visibilidade de senha.
- `shared.module.ts`: Módulo principal que centraliza a importação e exportação dos recursos compartilhados.

3 Detalhamento Técnico

3.1 Estrutura de Rotas

O arquivo `app-routing.module.ts` define as rotas da aplicação, controlando a navegação entre os módulos de funcionalidades. Ele organiza as rotas e permite o carregamento sob demanda dos módulos, utilizando Lazy Loading. A seguir, temos um exemplo da configuração das rotas para as funcionalidades de login, registro de usuários e dashboard de administração.

```
const routes: Routes = [
  {
    path: 'e-driver',
    children: [
      {
        path: 'intro-page',
        loadChildren: () => import('./settings/features/intro-page/module/intro-page.module').then(m => m.IntroPageModule)
      },
      {
        path: 'login',
        loadChildren: () => import('./settings/core/security/login/login.module').then(m => m.LoginModule)
      },
      {
        path: 'users/registration',
        loadChildren: () => import('./settings/features/users/users.module').then(m => m.UsersModule),
      },
    ]
  },
```

3.2 Guards de Rota

As rotas podem ser protegidas por guardas de autenticação, como o `authGuard`, e por guardas de verificação de compatibilidade de versão, como o `canMatchGuard`. Eles são responsáveis por verificar se o usuário está autenticado ou se a versão da aplicação é compatível para acessar determinados módulos.

Abaixo, um exemplo de como as rotas são protegidas:

```
  {
    path: 'e-driver',
    canActivate: [authGuard],
    canMatch: [canMatchGuard],
    children: [
      {
        path: 'dashboard',
        loadChildren: () => import('./settings/features/home/
            home.module').then(m => m.HomeModule),
      },
      {
        path: 'map',
        loadChildren: () => import('./settings/features/trip-
            planner-maps/map-stations.module').then(m => m.
            MapStationsModule),
      },
      {
        path: 'admin',
        loadChildren: () => import('./settings/features/admin/
            admin.module').then(m => m.AdminModule)
      },
    ]
  },
  { path: '', redirectTo: 'e-driver/intro-page', pathMatch: '
      full' },
];
```

3.3 Cobertura de Testes com Jest

Os testes são realizados com `Jest`, uma ferramenta popular de teste para JavaScript e TypeScript. Para visualizar a cobertura dos testes, o comando `npx jest -coverage` deve ser executado. Isso gera uma pasta `coverage`, contendo um arquivo `index.html`, que pode ser aberto em um navegador para visualizar o relatório de cobertura.

Exemplo de comando para ver a cobertura:

```
npx jest --coverage
```

O Jest gera um relatório detalhado que indica quais partes do código foram testadas e quais não foram, ajudando a garantir que todas as funcionalidades da aplicação estejam cobertas por testes.

Abaixo está um exemplo de configuração do Jest, que define as opções para coleta de cobertura, formatos de relatório e arquivos a serem ignorados durante a cobertura.

```
import type { Config } from 'jest';
const config: Config = {
  coverageDirectory: 'coverage',
  coverageReporters: ['html', 'text', 'lcov'],
  collectCoverageFrom: [
    "**/*.{js,jsx,ts,tsx}","!**/node_modules/**", "!**/vendor
       /**","!**/dist/**","!**/*.spec.{js,jsx,ts,tsx}","!**/.
       angular/**", "!**/coverage/**","!<rootDir>/setup-jest.
       ts", "!**/*.config.{js,ts}",
  ],
  preset: 'jest-preset-angular',
  setupFilesAfterEnv: ['<rootDir>/setup-jest.ts'],
  testPathIgnorePatterns: [
    '<rootDir>/node_modules/','<rootDir>/dist/',
  ],
  transform: {
    '^.+\\.(ts|js|html)$': ['jest-preset-angular', {
      tsconfig: '<rootDir>/tsconfig.spec.json',
         stringifyContentPathRegex: '\\.html$',
    }],
  },
};
export default config;
```

Este arquivo de configuração do Jest garante que a cobertura de testes seja coletada de todos os arquivos relevantes, exceto aqueles que estão na pasta `node_modules`, `dist`, ou que sejam arquivos de configuração e teste. O relatório de cobertura será gerado nos formatos `html`, `text` e `lcov`.

3.4 Comunicação entre Componentes e Serviços

Os componentes da aplicação se comunicam com os serviços que, por sua vez, interagem com o backend. O código abaixo ilustra como o componente `VehicleListComponent` utiliza o `VehicleService` para obter dados do backend e como ele também interage com o serviço `AlertasService` para mostrar mensagens de alerta ao usuário.

```
@Component({
  selector: 'app-vehicle-list',
  templateUrl: './vehicle-list.component.html',
  styleUrls: ['./vehicle-list.component.scss']
})
export class VehicleListComponent {
  constructor(
    private vehicleService: VehicleService, private dialog:
       MatDialog, private alertServise: AlertasService
  ) {}
  ngOnInit() {
    this.loadVehicles();
  }
  loadVehicles() {
    this.vehicleService.getAll().subscribe({
      next: (response) => {
        this.vehicles = response.content;
      },
      error: (error) => {
        this.handleError(error);
      }
    });
  }
  handleError(error: HttpErrorResponse) {
    this.alertServise.showError("Erro !!", error.message);
  }
}
```

O `VehicleService` é responsável por fazer a requisição HTTP para o backend, utilizando o ambiente configurado no `environment.ts`. A comunicação com o backend é feita por meio do `HttpClient`, utilizando métodos como `getAll()` para obter a lista de veículos.

3.5 Backend e Variáveis de Ambiente

A configuração do ambiente é crucial para determinar a URL base do backend e a chave da API do Google. O serviço `HttpClient` usa a URL base definida no arquivo `environment.ts` para fazer as requisições HTTP. O arquivo `environment.ts` contém as variáveis específicas para o ambiente de desenvolvimento, enquanto o `environment.prod.ts` define as variáveis para o ambiente de produção.

3.5.1 Exemplo de configuração do ambiente para desenvolvimento

```
export const environment = {
  production: false,
  apiUrl: 'http://localhost:8080/api',
  googleApiKey: 'sua-chave-da-api-google-aqui'
};
```

3.5.2 Exemplo de configuração do ambiente para produção

```
export const environment = {
  production: true,
  apiUrl: 'https://api.suaaplicacao.com/api',
  googleApiKey: 'sua-chave-da-api-google-para-producao-aqui'
};
```

Na versão de produção, a URL do backend é alterada para o endereço real da aplicação na nuvem ou servidor. A chave da API do Google também é configurada de forma segura para que a aplicação possa interagir com os serviços do Google, como Maps ou Geocoding, no ambiente de produção.

3.5.3 Exemplo do uso da URL no Serviço `VehicleService`

No serviço `VehicleService`, a URL base é concatenada com os endpoints específicos para realizar as operações no backend. O código abaixo mostra como

o `VehicleService` utiliza a URL base definida no `environment.ts` para realizar requisições HTTP.

```
@Injectable({
  providedIn: 'root'
})
export class VehicleService {

  private apiUrl = environment.apiUrl + '/vehicles';  % Usando a URL base configurada

  constructor(private http: HttpClient) {}

  getAll(pageIndex: number, pageSize: number): Observable<PaginatedResponse<Vehicle>> {
    return this.http.get<PaginatedResponse<Vehicle>>(`${this.apiUrl}?page=${pageIndex}&size=${pageSize}`);
  }
}
```

Neste exemplo, o serviço `VehicleService` usa a URL definida em `environment.apiUrl` e a concatena com o endpoint `/vehicles` para obter dados dos veículos.

3.5.4 Configuração da chave da API do Google

Para interagir com os serviços da Google, como o Google Maps, Geocoding, entre outros, é necessário configurar a chave da API do Google no arquivo `environment.ts`. Essa chave é utilizada para autenticar as requisições feitas para os serviços do Google.

Exemplo de utilização da chave da API no `GeocodingService`:

```
@Injectable({
  providedIn: 'root'
})
export class GeocodingService {

  private apiUrl = "https://maps.googleapis.com/maps/api/
     geocode/json?key=${environment.googleApiKey}";

  constructor(private http: HttpClient) {}

  geocode(address: string): Observable<GeocodingResponse> {
    return this.http.get<GeocodingResponse>(`${this.apiUrl}&
       address=${address}`);
  }
}
```

No código acima, observe que, ao invés de usar aspas duplas ("), devemos utilizar crase (‘) para criar strings do tipo `template string` em TypeScript. Isso é necessário para permitir interpolação de variáveis dentro da string, como o `googleApiKey`.