UTILIZANDO OS DADOS OBD2 PARA UMA MELHOR CONDUÇÃO

Adnan Silva Pedroso, Lance Armstrong Ferreira, Valeria Maria Volpe

e-mail:
adnan.pedroso@fatec.sp.gov.br; lance.armstrong@fatec.sp.gov.br;
valeria.volpe@fatec.sp.gov.br;

Resumo: O objetivo deste projeto foi desenvolver um aplicativo para ajudar na condução de veículos que dispõe de uma porta OBD, visando proporcionar maior economia de combustível e diminuir a emissão de poluentes. O aplicativo obtém dados através da porta OBD e, após analisá-los, exibe em tempo real ao motorista quais ajustes podem ser feitos em seu estilo de condução. Todas as interações são feitas através de um *smartphone,* do qual necessita estar conectado a um adaptador OBD via *bluetooth*. Para a implementação do projeto foi utilizado o *framework* Maui para a criação da interface gráfica.

Palavras-chave: OBD. Estilo de condução. Economia de combustível. Veículos automotores.

Abstract: *The objective of this project was to develop an application to help driving vehicles that have an OBD port, aiming to provide greater fuel economy and reduce the emission of pollutants. The application obtains data through the OBD port and, after analyzing them, shows the driver in real time which adjustments can be made to his driving style. All interactions are made through a smartphone, which needs to be connected to an OBD adapter via bluetooth. For the implementation of the project, the Maui framework was used to create the graphic interface.*

Keywords: *OBD. Driving style. fuel economy. Motor vehicles.*

1 Introdução

Com o avanço da tecnologia, os veículos começaram a usar as vantagens de um computador. A partir de 1995 a ideia da injeção eletrônica começou a ganhar popularidade por ser capaz de controlar todo o funcionamento do motor de um veículo através de um computador conectado a diversos sensores e atuadores.

Os veículos com um computador a bordo tornaram-se cada vez mais eficientes e confiáveis, visto que o computador era capaz de gerenciar o motor de forma extremamente precisa. Como a tecnologia está em constante evolução, os computadores que acompanham os veículos passaram a se tornar cada vez menores e mais inteligentes, e com toda essa evolução tecnológica, foi criado um sistema de autodiagnostico.

Esse sistema de autodiagnostico está presente em todos os veículos modernos, e é conhecido com OBD. O OBD é capaz de fornecer uma lista de erros que aconteceram e/ou

estão acontecendo. Cada erro se refere a algum mau funcionamento de alguma parte de veículo, facilitando assim a manutenção e garantindo segurança para os ocupantes.

Além dos erros, o OBD também fornece uma grande quantidade de dados em tempo real, provenientes da leitura de diversos sensores usados pela injeção eletrônica e outros módulos espalhados pelo veículo.

O aplicativo irá se beneficiar de todos esses dados fornecidos pelo OBD para realizar uma análise em tempo real da forma em que o veículo está sendo conduzido, e assim, mostrar ao motorista uma maneira mais eficiente de dirigir. Com isso, uma maior economia de combustível poderá ser alcançada, e consequentemente, menos poluição será causada pela queima de combustíveis fosseis.

2 Justificativa

Recentemente com o grande aumento no preço dos combustíveis, os motoristas estão cada vez mais preocupados com a eficiência e autonomia de seus veículos, porém apenas uma pequena parte dos motoristas sabem que é possível ter um aumento na eficiência de seus veículos apenas executando pequenos ajustes em suas formas de condução.

Como esses ajustes podem ser muito sutis, apenas motoristas experientes são capazes de realizá-los com excelência, pois é necessário conhecer e entender perfeitamente o veículo que está sendo dirigido. Entretanto, ao analisar os dados fornecidos pelo veículo através da porta OBD, é possível mostrar ao motorista de forma precisa e em tempo real quais ajustes podem ser realizados para alcançar uma maior eficiência de seu veículo.

3 Objetivo(s)

Este projeto tem como objetivo estudar como os dados fornecidos por um veículo podem ser utilizados para ajudar na condução dele, e desenvolver um aplicativo capaz de processar esses dados e exibir *feedbacks* para o motorista. Com o aplicativo será possível visualizar em tempo real o que deve ser feito para melhorar a forma em que se conduz o veículo.

4 Fundamentação Teórica

Neste tópico abordaremos as ferramentas e tecnologias utilizadas para o desenvolvimento deste projeto.

4.1 Bluetooth

O *bluetooth* é uma tecnologia de comunicação sem fio, que permite a troca de informações entre dispositivos próximos através de ondas de rádio.

Criado em 1994 pela empresa Ericsson, que permitisse comunicação entre aparelhos sem a necessidade de cabos e de baixo custo. Alguns aparelhos que podem ser usados pelo *bluetooth* são celulares, teclados, mouses, fone sem fio, caixas de som etc.

O *bluetooth* funciona semelhante ao *wi-fi*, a tecnologia usa ondas de rádio de curto alcance para enviar e receber informações entre aparelhos e não precisa estar conectado na internet. (ALECRIM, 2008).

4.2 GitHub

Desenvolvido em 2005 por Linus Torvalds, Git é um software para gerar projetos onde vários desenvolvedores podem trabalhar ao mesmo tempo no mesmo projeto.

O GitHub a base da plataforma é armazenar códigos de programação, produzidos por programadores do mundo todo e compartilhá-los como se fosse uma rede social comum. Assim é possível que qualquer usuário cadastrado na plataforma divulgue seus trabalhos e outros membros da plataforma podem fazer contribuições. (BARROS, 2021).

4.3 OBD

OBD é um sistema de autodiagnostico presente em todos os veículos modernos. OBD é uma sigla que vem do inglês, e significa *On-Board Diagnostic (Diagnostico a Bordo)*. Através de uma porta geralmente localizada na parte inferior do painel do veículo, é possível se comunicar diretamente com o computador central do veículo, podendo ler diversas informações importantes como por exemplo códigos de erro e o número do chassi do veículo. (OLIVEIRA, 2022)

4.5 Figma

O Figma é um editor online com ênfase em prototipagem de interfaces gráficas e estruturas de design de experencia de usuário, sendo possível modelar o projeto com mais de uma pessoa ao mesmo tempo. (VILLAIN, 2022)

4.6 .NET Maui

O .NET Maui é uma SDK – *Software Development Kit* (kit de Desenvolvimento de Software) criado pela Microsoft, que permite desenvolver um único aplicativo para funcionar em diversas plataformas de forma rápida e prática, ou seja, uma única base de código é capaz de funcionar em sistemas Android, iOS, MacOS e Windows. (MACORATTI, 2021)

O .NET Maui utiliza a linguagem C# para programação e XAML - *Extensible Application Markup Language* (Linguagem de Marcação de Aplicativos Extensíveis) para a criação de telas. A comunicação do XAML com código C# é feito a partir de eventos, dessa forma cria-se uma amarração permitindo que as duas linguagens possam interagir entre si.

O .NET Maui funciona através da plataforma .NET 6, que por sua vez consegue fazer a interação entre o aplicativo e o sistema operacional abstraindo todo o código para uma linguagem intermediaria. Uma vez que o aplicativo é iniciado, o código resultante da linguagem

intermediaria é compilado em tempo real para *assembly* nativo do sistema operacional que o aplicativo está sendo executado.

4.7 Machine Learning

Machine learning é um ramo da inteligência artificial que se concentra no desenvolvimento de algoritmos e técnicas que permitem aos computadores aprenderem a partir de dados. Em vez de seguir instruções específicas, os sistemas de *machine learning* são capazes de identificar padrões nos dados e tomar decisões ou fazer previsões com base nesses padrões. Essa capacidade de aprender e se adaptar torna o *machine learning* uma ferramenta poderosa em uma variedade de aplicações, desde reconhecimento de voz e imagem até recomendações personalizadas em plataformas online.

Os algoritmos de *machine learning* são alimentados por dados, e a qualidade e quantidade desses dados desempenham um papel fundamental na eficácia do modelo resultante. Os dados são divididos em conjuntos de treinamento e teste, e o modelo é ajustado iterativamente para melhorar seu desempenho.

4.8 Interface ELM327

A interface ELM327 é um microcontrolador programado que traduz comandos utilizados pelo protocolo OBD para que possam ser entendidos por um dispositivo de diagnóstico, como um computador ou smartphone. O ELM327 permite a leitura de códigos de erro, monitoramento de dados em tempo real e outros diagnósticos de veículos, facilitando a comunicação entre o veículo e o usuário.

ELM327 é um pequeno dispositivo usado para diagnóstico de automóveis. Ele se conecta à porta OBD do seu veículo. Seu principal benefício depende do aplicativo ou software conectado, mas o mais importante é ler e compreender os códigos de erro do seu carro. Isso torna mais fácil identificar e corrigir problemas. (ROBINSON, 2023)

5 Trabalhos Similares

Após uma ampla pesquisa, foram encontrados alguns trabalhos com propostas similares, que mesmo sendo similares apresentam particularidades e propostas diferentes.

O primeiro trabalho é o *Reading And Interpreting Diagnostic Data From Vehicle OBDII System* (Lendo e Interpretando Dados de Diagnostico do Sistema OBDII de um Veículo), o qual aborda como podemos conectar ao sistema OBD de um veículo e obter os dados fornecidos por ele. É apresentado também alguns dos protocolos e padrões necessários para realizar a comunicação. (DZHELEKARSKI, 2005).

O segundo trabalho é o Leitura OBD2 Através de Smartphone, qual aborda o desenvolvimento de um sistema para *smartphone* capaz de realizar leituras via OBD, baseando-se em tecnologias de baixo custo e de fácil acesso. (OSS, 2018)

O terceiro trabalho é o *OBD2 PID Reader* (Leitor de PID de OBD2), que mostra o caminho percorrido pelo autor para ler os dados OBD de um veículo Chevrolet Volt. É abordado desde a arquitetura do hardware necessário até a exibição dos dados em um computador. (FONG, 2013).

6 Metodologia

Para auxiliar no desenvolvimento desse projeto foram utilizados trabalhos semelhantes que serão apresentados a seguir:

Reading And Interpreting Diagnostic Data From Vehicle OBDII: esse trabalho propõe um sistema de leitura e interpretação de dados diagnósticos do sistema OBD de veículos. Ele exibe informações como velocidade do veículo, rotações por minuto (RPM) do motor e consumo de combustível, entre outras informações. O trabalho é relevante para esse projeto porque utiliza o mesmo protocolo de comunicação do OBD para obter os dados do veículo.

Leitura OBD2 Através de Smartphone: esse trabalho apresenta uma solução para a leitura de dados do sistema OBD de veículos utilizando um *smartphone*. O sistema utiliza um adaptador de OBD para *bluetooth* para poder estabelecer uma comunicação entre o veículo e o *smartphone*, exibindo as informações em um aplicativo específico. O trabalho foi relevante por utilizar a mesma tecnologia de comunicação, no caso o *bluetooth*, para transferência de dados.

OBD2 PID Reader: Esse trabalho apresenta um *software* para leitura de dados do protocolo OBD de veículos utilizando a linguagem de programação Java. O *software* é capaz de exibir informações como temperatura do motor, RPM do motor e velocidade do veículo. O trabalho é relevante porque irá auxiliar na leitura dos dados do sistema OBD.

Com a análise desses trabalhos semelhantes, percebe-se que existem diversas formas para a leitura de dados do sistema OBD de veículos utilizando diferentes tecnologias e linguagens de programação. Entretanto, o presente projeto se diferencia dos trabalhos apresentados por utilizar a tecnologia .NET Maui para o desenvolvimento da interface gráfica e inteligência artificial analisar os dados obtidos e entregar resultados precisos para alcançar o objetivo do projeto. Além disso o projeto busca oferecer uma solução mais completa, onde não terá apenas informações sobre performance do veículo, mas também fornecerá ao condutor um *feedback* sobre como ele está dirigindo, podendo realizar ajustes visando a economia de combustível.

7 Desenvolvimento

Para o desenvolvido da aplicação foi utilizado o *framework* Maui (*Multi-platform App UI*) da Microsoft. Este *framework* possibilita a criação de aplicações multiplataformas a partir de um único código fonte. A interface gráfica é definida a partir da linguagem de marcação XAML (*Extensible Application Markup Language*), ela separa a lógica da criação dos componentes visuais da lógica de negócios, proporcionando um baixo acoplamento entre a parte gráfica e a parte logica da aplicação. A figura 1 mostra um trecho em XAML referente a uma das telas da aplicação.

Figura 1 Exemplo de código XAML

```
<VerticalStackLayout VerticalOptions="CenterAndExpand">
    <Label
        FontSize="Body"
        HorizontalTextAlignment="Center"
        Opacity="0.7"
        Text="Nenhum dispositivo encontrado" />

    <Label
        HorizontalTextAlignment="Center"
        Opacity="0.7"
        Text="Verifique sua conexão bluetooth e tente novamente" />
</VerticalStackLayout>
```

Fonte: autoria própria

Para assegurar a qualidade da aplicação, diversos padrões e técnicas de desenvolvimentos foram adotados. Um dos principais padrões adotados é a injeção de dependência, este poderoso padrão proporciona a inversão de controle entre as classes e serviços da aplicação.

A injeção de dependências é uma forma de implementar a inversão de controle, onde a responsabilidade de fornecer as dependências necessárias para uma classe é transferida para um componente externo. Isso significa que uma classe não cria suas próprias dependências, mas as recebe de fora através de um construtor.

Em C#, há um *framework* fornecido nativamente pela linguagem. Para utilizá-lo, primeiramente precisamos registrar todas as dependências de nossa aplicação, para isso, o *framework* expõe um *container* com métodos específicos, permitindo o registro da dependência e o seu ciclo de vida. A figura 2 mostra o registro de todas as dependências da aplicação.

Figura *2* Dependências da aplicação

```
#if ANDROID
        builder.Services.AddSingleton<IBluetoothConnector, Platforms.Android.AndroidBluetoothConnector>();
#endif

        builder.Services.AddMachineLearning();

        builder.Services.AddSingleton<IDialogManager, DefaultDialogManager>();
        builder.Services.AddSingleton<IOBDEncoder, DefaultOBDEncoder>();
        builder.Services.AddSingleton<IDataPuller, DefaultDataPuller>();
        builder.Services.AddSingleton<IDataFactory, DefaultDataFactory>();
        builder.Services.AddSingleton<IRequestProcessor, DefaultRequestProcessor>();
        builder.Services.AddSingleton<IVehicleProvider, DefaultVehicleProvider>();
        builder.Services.AddSingleton<IVehicleRepository, DefaultVehicleRepository>();
        builder.Services.AddSingleton<IWiseCalculations, DefaultWiseCalculations>();
        builder.Services.AddSingleton<IReportGenerator, DefaultReportGenerator>();
        builder.Services.AddSingleton<ITemperatureAlert, DefaultTemperatureAlert>();
        builder.Services.AddSingleton<IDriverFeedback, DefaultDriverFeedback>();

        builder.Services.AddSingleton(AudioManager.Current);

        builder.Services.AddTransient<MainPage>();

        builder.Services.AddTransient<MainPageViewModel>();
        builder.Services.AddTransient<MainViewViewModel>();
        builder.Services.AddTransient<DataViewModel>();
        builder.Services.AddTransient<ConnectionViewModel>();
```

Fonte: autoria própria

Outro padrão utilizado é o padrão MVVM (*Model View View Model*), que consiste em separar a interface de usuário de toda a lógica da aplicação. Este padrão utiliza ligações dinâmicas (*bindings*) para realizar a comunicação entre a interface de usuário (*View*) e a parte lógica (*View Model*).

Um dos grandes desafios foi conciliar este padrão com a injeção de dependências. Para alcançar este objetivo, registramos a página principal da aplicação no container de injeção de dependências, e em seguida registramos todas as v*iew models* – como pode ser observado na figura 2. Depois disso, podemos injetar todas as *view models* conforme necessário, como pode ser observado na figura 3.

Figura *3* Injetando dependências

```
public MainPageViewModel(MainViewViewModel mainViewModel,
    DataViewModel dataViewModel,
    ConnectionViewModel connectionViewModel)
{
    MainViewModel = mainViewModel;
    DataViewModel = dataViewModel;
    ConnectionViewModel = connectionViewModel;
}
```

Fonte: autoria própria

Após concluirmos a infraestrutura básica do projeto, estávamos prontos para começar a desenvolver os requisitos específicos da aplicação, mas nesse momento notamos alguns pontos que deveriam ser definidos antes de prosseguirmos.

Ao analisarmos que estamos trabalhando com um *framework* multiplataformas, percebemos que cada plataforma tem suas devidas particularidades, então não podemos depender diretamente de implementações concretas, isto é, não devemos expor classes com métodos específicos para uma determinada plataforma, pois cada plataforma requer uma implementação diferente. Mesmo que o foco do trabalho seja unicamente para Android, adotamos a estratégia de depender apenas de abstrações através de interfaces, pois caso futuramente seja necessário a expansão para outras plataformas, podemos facilmente substituir suas implementações sem a necessidade de refazer todo o projeto.

Outra decisão a ser feita era em relação a responsividade da aplicação, uma vez que operações que envolvam *bluetooth* naturalmente tem uma latência maior para serem processadas, isso poderia ocasionar travamentos na interface de usuário e proporcionar uma experiencia negativa. Para evitar este problema, adotamos o extensivo uso de eventos junto ao recurso de tarefas assíncronas presente na própria linguagem de programação. Tal uso se provou extremamente eficiente, pois podemos reagir assincronamente a eventos evitando processamentos na interface de usuário.

O último ponto a ser considerado antes de iniciarmos o desenvolvimento foi em relação a arquitetura do projeto. Decidimos utilizar serviços para agrupar métodos e procedimentos relacionados em um único lugar, pois isso proporciona uma maior flexibilidade e robustez a aplicação, pois assim conseguimos aplicar com facilidade os princípios SOLID. Todos os serviços devem ser expostos através de uma interface que deve ser registrada na injeção de dependências como *singleton* para ser injetado conforme necessário.

Com tudo isso claro e definido, começamos a implementar o primeiro serviço da aplicação, o serviço em questão seria responsável por gerenciar a conexão com o dispositivo *bluetooth.*

Figura *4* Definição do serviço de conexão bluetooth

```
public interface IBluetoothConnector
{
    event EventHandler? DeviceConnected;

    event EventHandler? DeviceDisconnected;

    event EventHandler? DeviceChanged;

    IBluetoothDevice? ConnectedDevice { get; }

    bool IsConnected { get; }

    Task<IEnumerable<string>> GetAvailableDevices();

    Task<IBluetoothDevice> Connect(string deviceName);

    Task Disconnect();
}
```

Fonte: autoria própria

Como pode ser observado na figura 4, o serviço expõe métodos e eventos uteis relacionados a conexão e gerenciamento do dispositivo *bluetooth*. Uma vez que um dispositivo for conectado com sucesso, ele fica exposto através da propriedade ConnectedDevice. Esta propriedade fornece acesso ao nome do dispositivo e a métodos utilizados para enviar e receber dados do dispositivo, como pode ser visto na figura 5.

Figura *5* Definição da interface do dispositivo bluetooth

```
public interface IBluetoothDevice
{
    string Name { get; }

    Task SendAsync(byte[] data, CancellationToken cancellation = default);

    Task<byte[]> ReadAsync(CancellationToken cancellation = default);
}
```

Fonte: autoria própria

Com isso já é possível enviar e receber dados via *bluetooth*. Então começamos a parte relacionada ao OBD. Durante o desenvolvimento da aplicação, tivemos acesso apenas ao veículo Ford Fiesta 2012 1.6 que está de acordo com o protocolo e normas ISO 14230-4 KWP, portanto todo desenvolvimento e explicação está de acordo com esta norma. Para melhor entendimento do desenvolvimento, vamos explicar breve e resumidamente como funciona o protocolo OBD.

Tabela *1* Estrutura de dados OBD

Identificador	# Bytes	Modo	PID	A	B	C	D	Terminador

Considerando a estrutura de dados apresentada na tabela 1, temos:

- Identificador: responsável por identificar o módulo a receber ou que enviou a resposta;
- # Bytes: informa o tamanho da mensagem;
- Modo: informa o tipo da requisição ou resposta;
- PID: código que informa qual dado está sendo requisitado ou respondido;
- A: Dado a ser recebido;
- B: Dado a ser recebido;
- C: Dado a ser recebido;
- D: Dado a ser recebido;
- Terminador: indica o final da requisição ou resposta.

Esta estrutura de dados completa pode ser chamada de quadro ou *frame*. Com exceção do identificador e do terminador, todos os dados devem ter tamanho de 2 bytes representados de forma hexadecimal. O identificador deve ter 3 bytes representados de forma hexadecimal e o terminador deve ter 1 byte representado por um caractere da tabela ASCII. Esta estrutura é utilizada tanto em requisições quanto em respostas.

O identificador indica qual o alvo da mensagem ou qual módulo está respondendo, ou seja, em uma requisição podemos utilizar o valor 7DF que funciona como um *broadcast* e deve propagar a mensagem para todos os módulos do veículo. O módulo que aceitar a requisição

deve responder com seu identificador, como por exemplo, o módulo de gerenciamento do motor geralmente possui o identificador 7E8. A partir desse ponto podemos nos comunicar diretamente com ele enviando o identificador dele na requisição.

O número de bytes quantos bytes de dado estão sendo enviados ou recebidos, isto é necessário pois este número pode variar de acordo com o que for solicitado, por exemplo, a requisição referente a velocidade do veículo possui apenas 1 byte de dados, mas a requisição referente ao RPM possui 2 bytes de dados.

O modo informa qual modo de requisição ou resposta está sendo enviada ou recebida, por exemplo, o modo 01 é utilizado para informações de leitura dos sensores, o modo 09 é utilizado para informações referente ao veículo.

O PID indica qual dado está sendo requerido ou enviado, por exemplo, para o RPM o PID é o 0C, para a velocidade o PID é o 0D e para a carga do motor o PID é o 04. Os PID disponíveis podem variar de fabricante para fabricante e de acordo com os sensores disponíveis no veículo.

A região referente a A, B, C e D carregam os dados conforme necessário, em requisições esses dados não são necessários, mas em respostas são obrigatórias. Para obter o valor, equações são utilizadas de acordo com o PID, por exemplo, na equação a seguir temos a equação utilizada para obter o valor da RPM do veículo.

$$\frac{256A+B}{4} \quad (1)$$

O terminador indica o final da mensagem, na norma do veículo que utilizamos seu valor é 0D, que na tabela ASCII representa o caractere de *carriage return*.

Então, para obtermos a RPM do veículo, precisamos fazer uma requisição como a representada na tabela 2.

Tabela *2* Requisição do RPM

7DF	02	01	0C	00	00	00	00	/r

Como resposta, teríamos a seguinte como representada na tabela 3.

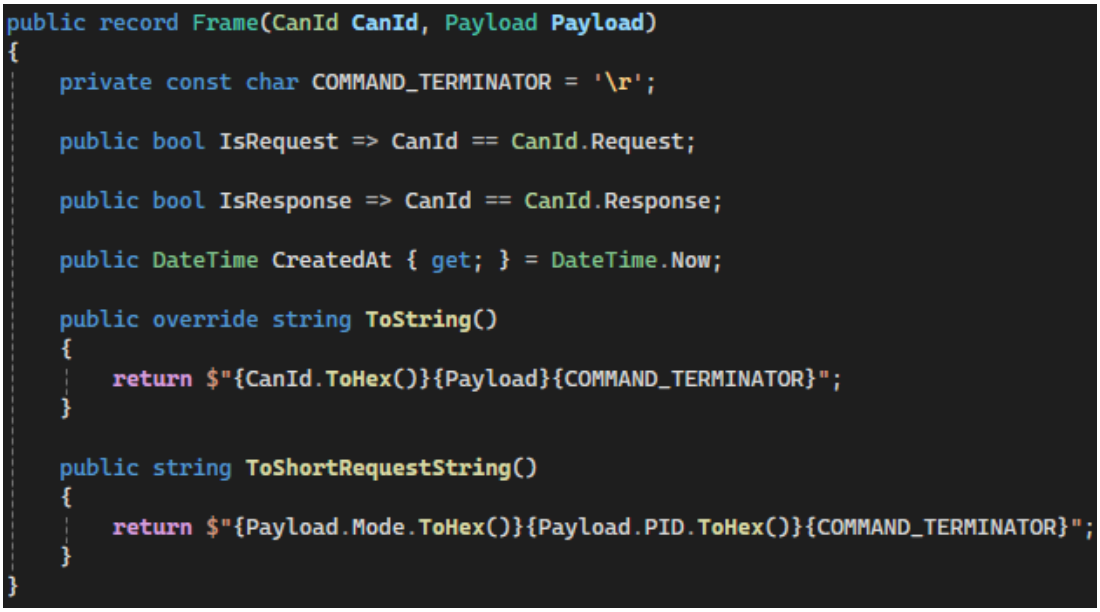
Tabela *3* Resposta do RPM

7E8	04	41	0C	12	34	00	00	/r

Aplicando a equação nos dados recebidos, obtemos o valor de 1165, isso significa que no momento da requisição o RPM era de 1165.

Na aplicação, esta estrutura está representada em uma classe como mostra a figura 6.

Figura *6* Representação de um quadro OBD

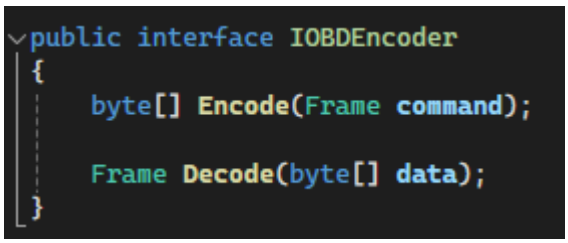


```
public record Frame(CanId CanId, Payload Payload)
{
    private const char COMMAND_TERMINATOR = '\r';

    public bool IsRequest => CanId == CanId.Request;

    public bool IsResponse => CanId == CanId.Response;

    public DateTime CreatedAt { get; } = DateTime.Now;

    public override string ToString()
    {
        return $"{CanId.ToHex()}{Payload}{COMMAND_TERMINATOR}";
    }

    public string ToShortRequestString()
    {
        return $"{Payload.Mode.ToHex()}{Payload.PID.ToHex()}{COMMAND_TERMINATOR}";
    }
}
```

Fonte: autoria própria

Com a estrutura devidamente definida, precisamos transformar os dados puros que são recebidos pelo *bluetooth* para a estrutura definida na figura 6 e vice-versa. Para isso foi criado um serviço capaz de codificar e decodificar esses dados, este serviço pode ser observado na figura 7. Aqui podemos observar uma das vantagens de depender apenas de abstrações, pois se necessário se adequar a outras normas e protocolos OBD, será necessário apenas substituir a implementação deste serviço.

Figura *7* Serviço de codificação e decodificação



```
public interface IOBDEncoder
{
    byte[] Encode(Frame command);

    Frame Decode(byte[] data);
}
```

Fonte: autoria própria

Figura *8* Implementações do serviço

```
public byte[] Encode(Frame frame)
{
    var data = frame.ToShortRequestString();
    var buffer = Encoding.ASCII.GetBytes(data);

    return buffer;
}

public Frame Decode(byte[] buffer)
{
    var command = Encoding.ASCII.GetString(buffer);
    if (command.Contains('>'))
    {
        var commands = command.Split('>');

        foreach (var possibleCommand in commands.Reverse())
        {
            if (string.IsNullOrEmpty(possibleCommand))
                continue;

            command = possibleCommand;
            break;
        }
    }

    var bytes = GetBytes(command);

    var canId = GetCanId(bytes);
    var mode = GetMode(bytes);
    var pid = GetPid(bytes);
    var fragments = CreateFragments(bytes);

    var data = Data.CreateFrom(fragments);
    var payload = new Payload(mode, pid, data);
    var frame = new Frame(canId, payload);

    return frame;
}
```

Fonte: autoria própria

Para coordenar a requisição dos dados via OBD de forma assíncrona, foi necessário a criação de mais dois serviços, um dos serviços é responsável por enfileirar todos as requisições e remover requisições duplicadas, o outro é responsável por processar as requisições. Esta separação se mostrou necessária pois o há uma grande latência no processamento das requisições OBD, por isso o processamento precisa ser feito em uma *thread* completamente isolada. Por outro lado, não há latência nem alta demanda de poder computacional no enfileiramento das requisições.

Abaixo podemos ver as definições dos serviços citados anteriormente.

Figura 9 Definição do serviço de enfileiramento

```
public interface IDataPuller
{
    Task<TType> PullDataAsync<TType>(CancellationToken cancellationToken = default)
        where TType : class, IOBDData;
}
```

Fonte: autoria própria

Figura *10* Definição do serviço de processamento

```
public interface IRequestProcessor
{
    event FrameReceivedEventHandler FrameReceived;

    void QueueRequest<TType>(TType request) where TType : class, IOBDData;
}
```

Fonte: autoria própria

A implementação desses serviços foi um grande desafio, pois o processamento dos dados não é assíncrono por natureza. Para casos desse tipo, podemos contornar a situação de algumas maneiras. No nosso projeto, optamos por utilizar a classe *TaskCompletionSource,* que é nativa do ecossistema .NET, em conjunto com eventos e *callbacks.*

A classe *TaskCompletionSource* permite criar e manipular tarefas assíncronas que podem ser manualmente completadas, oferecendo métodos para sinalizar suas possíveis conclusões. Isso é útil para criar operações assíncronas personalizadas, facilitando o controle e a coordenação de fluxos de trabalho assíncronos complexos, onde o resultado da tarefa pode ser determinado posteriormente, em vez de ser concluído automaticamente por uma operação assíncrona.

Estes dois serviços podem ser considerados o coração da aplicação, pois coordenam e ordenam todo o acesso ao sistema OBD, fornecendo os dados de maneira eficaz e assegurando sua integridade.

Apesar de nosso projeto ter como foco apenas um único veículo, visando uma maior flexibilidade para expansões futuras, definimos mais dois serviços que são responsáveis por fornecer informações do veículo atual. Estes serviços são bem mais simples se comparados aos outro citados anteriormente, pois trata-se apenas de um provedor e um repositório. O repositório é responsável por fornecer dados de veículos vindos de uma fonte externa, como por exemplo, um banco de dados. O serviço provedor detecta o veículo atual a partir do número de chassis fornecido pelo sistema OBD, e utilizando essa informação, ele busca o veículo no repositório e guarda em memória todas as informações relacionadas ao veículo, evitando assim que uma nova consulta ao banco de dados seja feita toda vez que seja necessário obter alguma informação do veículo.

Outro serviço simples, porém, de suma importância, é o sistema de monitoramento da temperatura de funcionamento do motor. Apesar de estar fora do escopo inicial do projeto, este serviço se provou necessário, pois durante o desenvolvimento da aplicação, o veículo de testes apresentou um superaquecimento em situações especificas. Como isso pode levar a falhas catastróficas, decidimos incluir este serviço no projeto, pois assim estaríamos mais seguros para seguir com o desenvolvimento. Este serviço funciona de maneira simples, uma vez que o sistema se conecta ao dispositivo OBD via *bluetooth*, ele continuamente checa a temperatura do motor, e caso a temperatura exceda os padrões normais, ele soa um alarme sonoro para o motorista.

Como a natureza da nossa aplicação exige diversos cálculos, criamos também um serviço para unificar todos os cálculos que serão utilizados para processar os dados obtidos através do sistema OBD. Este serviço fornece métodos capazes de calcular a marcha atual do veículo através da velocidade e do RPM do motor, o consumo de combustível, o fluxo da massa de ar que é admitida pelo motor, a variação de diversos parâmetros e outros.

Figura *11* Definição do serviço de cálculos

```
public interface IWiseCalculations
{
    int GetCurrentGear(double rpm, double speed);

    double GetVolumetricEfficiency(double rpm, double maf, double intakeTemperature, double map);

    double GetImap(double rpm, double map, double intakeAirTemperature);

    double GetCalculatedMaf(double imap, double volumetricEfficiency);

    double GetFuelComsumption(double predictedFuelComsumption, double speed, double maf, int fuelStatus);

    double GetFuelEfficiency(double fuelComsumption, bool isOnHighway);

    double GetAverageFuelComsumption(IEnumerable<double> comsumptions);

    double GetComsumptionVariance(IEnumerable<double> comsumptions);

    TrafficCondition GetTrafficCondition(double averageSpeed, bool isOnHighway);

    double GetAverageSpeed(IEnumerable<double> speeds);

    double GetRpmVariance(IEnumerable<double> rpms);

    double GetSpeedVariance(IEnumerable<double> speeds);

    double GetTpsVariance(IEnumerable<double> tps);
}
```

Fonte: autoria própria

Finalmente chegamos ao serviço considerado o cérebro da aplicação, pois ele obtém e processa todos os dados fornecidos pelo sistema OBD. Apesar de ter uma definição simples, sua implementação tem um nível de complexidade elevado. A filosofia por trás deste serviço é relativamente simples, consiste em gerar relatórios e publicá-los através de um evento, disponibilizando também os relatórios gerados anteriormente através de uma coleção. Podemos ver sua definição na figura 12 e as propriedades que compõe um relatório na figura 13.

Figura *12* Serviço gerador de relatórios

```
public interface IReportGenerator
{
    event ReportGeneratedEventHandler? ReportGenerated;

    IReadOnlyCollection<Report> Reports { get; }
}
```

Fonte: autoria própria

Figura *13* Estrutura e propriedades do relatório

```
public record Report()
{
    public required DateTime CreatedAt { get; init; }
    public required double Speed { get; init; }
    public required double AverageSpeed { get; init; }
    public required double SpeedVariation { get; init; }
    public required double Rpm { get; init; }
    public required double CoolantTemperature { get; init; }
    public required double EngineLoad { get; init; }
    public required double IntakeAirTemperature { get; init; }
    public required double IntakePressure { get; init; }
    public required double ThrottlePosition { get; init; }
    public required double FuelTrim { get; init; }
    public required int FuelStatus { get; init; }
    public required double MassAirFlow { get; init; }
    public required double VolumetricEfficiency { get; init; }
    public required int Gear { get; init; }
    public required double FuelConsumption { get; init; }
    public required double AverageFuelConsumption { get; init; }
    public required DrivingStyle DrivingStyle { get; init; }
    public required double DrivingEfficiency { get; init; }
    public required double AverageDrivingEfficiency { get; init; }
    public bool IsOnHighway => AverageSpeed > 60;
    public bool IsVehicleMoving => Speed > 0;
    public bool IsEngineRunning => Rpm > 0;
}
```

Fonte: autoria própria

Como observado na figura 13, o relatório informa todas as leituras de sensores que podem ser realizadas pelo sistema OBD e algumas informações que são obtidas através de cálculos específicos.

A implementação do serviço gerador de relatórios se baseia em um temporizador que é iniciado quando um dispositivo *bluetooth* é conectado. Este temporizador aciona o método responsável pela geração dos relatórios a uma frequência de dez vezes por segundo, este método, por sua vez, verifica se há a algum relatório sendo construído. Se nenhum relatório estiver sendo construído, um novo relatório é iniciado, caso contrário, nenhuma ação é tomada e a execução do método é terminada. Quando a geração do relatório é concluída com sucesso, o método adiciona o relatório a coleção de relatórios e logo em seguida emite um evento, para que todas as dependências que utilizam o relatório possam ser atualizadas.

O primeiro passo para a geração do relatório é ler todos os sensores disponíveis, e em sequência podemos realizar processamentos básicos com os dados obtidos, como por exemplo, calcular a marcha atual do veículo. Mas para processamentos mais avançados é necessário a leitura de um sensor que não está disponível em todos os veículos, este sensor é conhecido com sensor MAF (*Mass Air Flow*), ele é responsável por medir a quantidade de ar admitida pelo motor. Este sensor é de extrema importância para o objetivo do projeto, pois através dos valores fornecidos por ele podemos obter dados importantes, como por exemplo, o consumo de combustível, pois a quantidade de combustível utilizado pelo motor é diretamente proporcional a quantidade de ar admitida, então, ao obtermos a quantidade de ar admitida, podemos facilmente aproximar a quantidade de combustível consumida em tempo real.

Como citado anteriormente, este sensor não está disponível em todos os veículos, e como esperado, ele não está disponível no nosso veículo de testes. Para contornar essa situação, utilizamos inteligência artificial através da técnica de aprendizado de máquina. O algoritmo é capaz de prever um possível valor para o sensor MAF a partir da leitura de todos os outros sensores disponíveis com uma precisão de até aproximadamente 98%. No decorrer do desenvolvimento, esta estratégia se provou extremamente eficiente e precisa, superando nossas expectativas.

Outro ponto chave do gerador de relatórios é o cálculo do consumo de combustível, para isso utilizamos outro algoritmo de aprendizado de máquina capaz de prever o consumo de combustível em quilômetros por litros através dos valores obtidos pelos sensores, incluído o MAF, que foi previsto através de outro algoritmo de aprendizado de máquina. A precisão deste algoritmo foi de aproximadamente 91%, e durante o desenvolvimento, também se provou eficiente e precisa. Uma das principais vantagens da adoção dessa estratégia foi a simplificação significativa da nossa base de códigos.

Ao obtermos o consumo de combustível aproximado, podemos calcular a eficiência aproximada do veículo em tempo real. Alcançamos isso através da divisão do número obtido pela previsão do algoritmo de aprendizado de máquina pelo número divulgado pela fabricante do veículo, por exemplo, a autonomia divulgada para o nosso veículo de teste é de 8.4 quilômetros por litro e, através do algoritmo, obtemos uma autonomia de 7.8 quilômetros por litro. Ao realizarmos o cálculo, obteríamos uma eficiência resultante de aproximada de 92%.

Nosso último grande desafio no gerador de relatórios foi definir o estilo de condução do motorista. Para isso, realizamos a média do coeficiente de variação do consumo, RPM, velocidade e posição do acelerador. Realizamos este cálculo com base nos dados obtidos nos últimos 10 segundos. Caso a média do coeficiente de variação seja superior a 25%, atribuímos ao motorista um estilo de condução agressivo, caso contrário, um estilo de condução contínuo.

Figura *14* Código responsável por calcular o coeficiente de variação

```
var consumptionVariance = wiseCalculations.GetComsumptionVariance([.. lastSecondsReports.Select(r => r.FuelConsumption), fuelComsumption]);
var rpmVariance = wiseCalculations.GetRpmVariance([.. lastSecondsReports.Select(r => r.Rpm), rpm]);
var speedVariance = wiseCalculations.GetSpeedVariance([.. lastSecondsReports.Select(r => r.Speed), speed]);
var tpsVariance = wiseCalculations.GetTpsVariance([.. lastSecondsReports.Select(r => r.ThrottlePosition), throttlePosition]);
var averageVariance = (consumptionVariance + rpmVariance + speedVariation + tpsVariance) / 4;
```

Fonte: autoria própria

Com isso, podemos avançar para nosso último serviço, o serviço responsável por analisar os relatórios e gerar um *feedback* para o motorista. A definição deste serviço expõe apenas um evento que será invocado quando um novo *feedback* for gerado. A implementação deste serviço se baseia em um temporizador de 10 segundos, que é iniciado assim que um dispositivo *bluetooth* é conectado. Então, a cada 10 segundos, é invocado um método capaz de analisar os relatórios gerados nos últimos 10 segundos, e através de uma série de comparações, escolhe qual o *feedback* mais apropriado para o motorista. Assim que a decisão do *feedback* é

concluída, o evento presente na definição do serviço é disparado, e carrega como argumento o *feedback* gerado. Na figura 15 podemos ver todos os possíveis feedbacks gerados pelo serviço.

Figura *15* Enumeração contendo os possíveis resultados

```
public enum CurrentFeedback
{
    HighThrottle,
    HighGear,
    LowGear,
    HighSpeed,
    HighRpm,
    HighVariation,
    None
}
```

Fonte: autoria própria

Assim, concluímos todos os principais serviços necessários para o funcionamento da aplicação, restando apenas a implementação da interface gráfica. Como adotamos o padrão MVVM, esta parte foi extremamente simples, pois através da injeção de dependências, bastou apenas injetar os serviços requeridos para cada tela, se inscrevendo nos eventos necessários e utilizando as propriedades expostas pelas definições dos serviços. Podemos ver isso em prática na figura 16

Figura *16* Injeção de dependências na tela principal

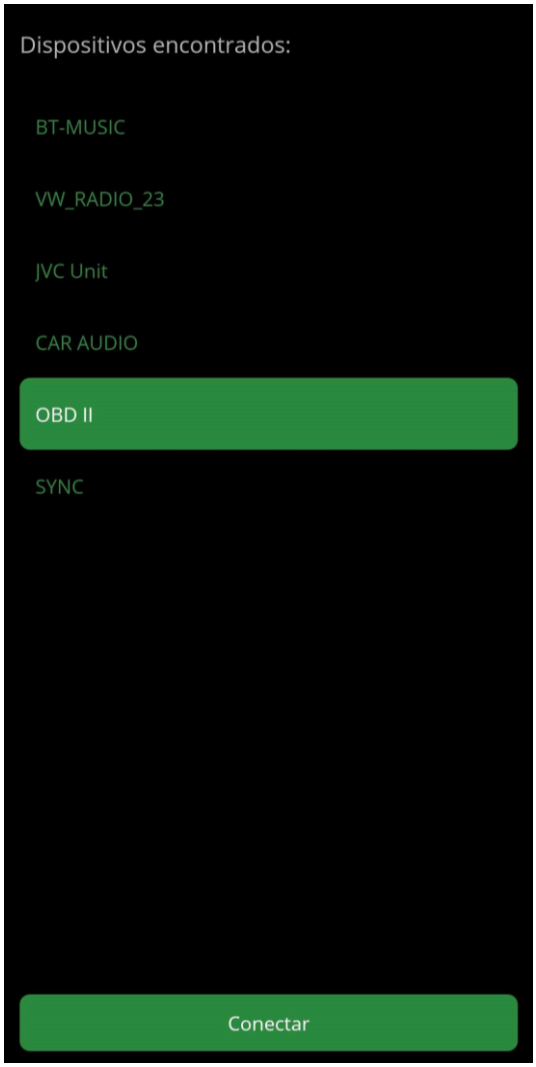
```
public MainViewViewModel(IReportGenerator reportGenerator,
    IVehicleProvider vehicleProvider,
    IDriverFeedback driverFeedback)
{
    this.reportGenerator = reportGenerator;
    this.vehicleProvider = vehicleProvider;

    vehicleProvider.VehicleChanged += OnVehicleChanged;
    reportGenerator.ReportGenerated += OnReportGenerated;
    driverFeedback.FeedbackReceived += OnFeedbackReceived;
}
```

Fonte: autoria própria

8 Resultados e Discussões

Ao iniciar o aplicativo, a tela de gerenciamento de conexões será exibida, exibindo uma lista com os dispositivos *bluetooth* encontrados. Por padrão, o aplicativo pré-seleciona o dispositivo mais provável de ser utilizado, mas o usuário é livre para selecionar outro dispositivo conforme necessário.

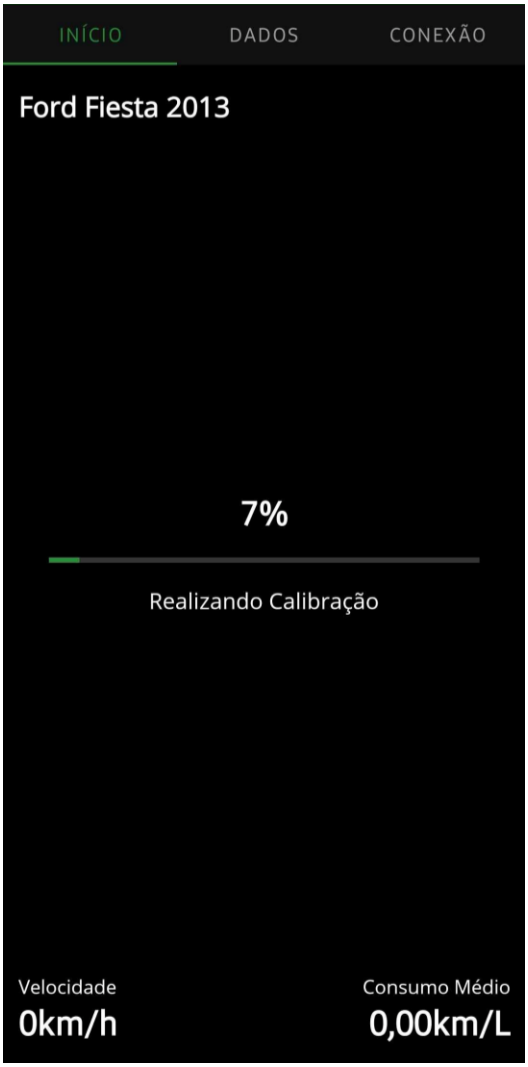
Figura *17* Tela de gerenciamento de conexões



Fonte: autoria própria

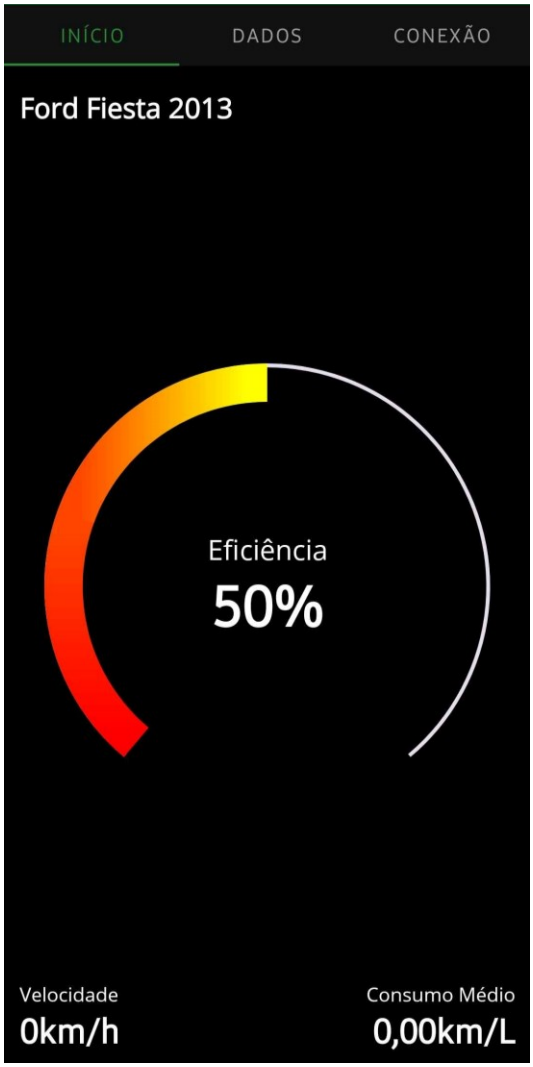
Após a conexão ser bem-sucedida, o usuário será levado a tela principal da aplicação, responsável por exibir eficiência do veículo e as informações e ações necessárias para o motorista. Ao realizar a conexão, é necessário realizar uma calibração no aplicativo por um breve período, o progresso da calibração pode ser acompanhado nesta tela. Esta tela também exibe avisos caso a temperatura do motor do veículo esteja fora da faixa de funcionamento ideal.

Figura *18* Tela inicial durante a fase de calibração



Fonte: autoria própria

Figura *19* Tela inicial durante o uso



Fonte: autoria própria

Há também uma tela capaz de mostrar as informações em tempo real, que pode ser útil para aferições ou até mesmo para a detecção de problemas que possam estar ocorrendo no veículo. Esta tela juntamente com a de gerenciamento de conexões somente podem ser exibidas caso o veículo não esteja em movimento. Caso o usuário esteja em uma dessas telas e o veículo entre em movimento, o usuário será levado a tela principal e não poderá retornar a essas telas até o veículo estar completamente parado.

Figura *20* Tela de exibição dos dados

Fonte: autoria própria

Por fim, ao utilizarmos o aplicativo no veículo de testes, pudemos constatar um aumento na eficiência do veículo, aumentado sua autonomia em cerca de 28%, de aproximadamente 6.2 quilômetros por litros, para aproximadamente 7.9 quilômetros por litros. Os testes foram realizados pelo mesmo motorista em um trajeto de 22 quilômetros, utilizando o mesmo combustível e em condições gerais similares. A validação dos dados pode ser obtida através do computador de bordo presente no veículo de testes.

O resultado obtido se provou satisfatório e atendeu as expectativas, pois, no veículo de testes, a distância máxima que pode ser alcançada com o tanque de combustível em sua capacidade máxima de 54 litros, passou de 334 quilômetros para 426 quilômetros. Um aumento de 92 quilômetros com a mesma quantidade de combustível.

9 Conclusões

Com a utilização contínua do aplicativo, concluímos que o aplicativo se provou capaz de cumprir o objetivo proposto, capaz de auxiliar motoristas a alcançarem uma maior eficiência de seus veículos e proporcionando um menor consumo de combustível.

Além do objetivo principal, também foi obtido grande êxito no planejamento e execução da interface gráfica, alcançando uma interface gráfica simples e elegante, intuitiva e muito fácil de ser utilizada, e principalmente, clara e direta com as informações que devem ser mostradas, sendo capaz de transmitir com maestria as ações a serem tomadas para o motorista.

Entretanto, é notável a oportunidade de melhoria em alguns pontos. Primeiramente, adicionar suporte para diversos protocolos OBD pode ampliar a compatibilidade do aplicativo com uma gama maior de veículos e sistemas, permitindo que mais usuários possam se beneficiar de suas funcionalidades.

Além disso, a implementação um banco de dados para armazenar informações sobre os veículos utilizados pelo usuário será fundamental para a evolução do aplicativo. Esse banco de dados permitirá o armazenamento de dados específicos de cada veículo, possibilitando análises mais detalhadas e personalizadas. Além disso, facilitará a gestão de múltiplos veículos, especialmente para frotas, oferecendo um controle centralizado e mais eficiente.

Por fim, a integração de dados de GPS pode melhorar significativamente a precisão e qualidade das ações recomendadas. O uso do GPS permitirá o aplicativo obter informações sobre a localização do veículo, e considerar fatores como o tráfego, velocidade máxima da via, topografia e condições da estrada, proporcionando resultados ainda mais precisos e personalizadas para otimizar ainda mais o consumo de combustível e a eficiência da condução.

Em conclusão, o projeto revelou-se um grande êxito desde a concepção da ideia até o planejamento e a execução. Através da implementação de uma interface gráfica intuitiva e eficiente, e de arquitetura e funcionalidades robustas, aliado ao uso de inteligência artificial, conseguimos não apenas atingir, mas superar os objetivos propostos. As oportunidades de melhoria identificadas apontam para um futuro ainda mais promissor, consolidando o aplicativo como uma ferramenta indispensável para diversos motoristas.

Agradecimentos

Gostaria de expressar meus sinceros agradecimentos aos professores, amigos e familiares que, com seu apoio incondicional e constante motivação, tornaram este projeto possível. A orientação dos professores foi fundamental para o sucesso deste trabalho. Aos amigos, agradeço a colaboração e incentivo. E aos familiares, sou grato pelo suporte emocional e compreensão ao longo de todo o processo. A todos, meu mais profundo agradecimento por acreditarem e contribuírem para a realização deste trabalho.

Referências

ALECRIM, Emerson. Bluetooth: o que é, como funciona e versões, 30 jan. 2008. Disponível em: <https://www.infowester.com/bluetooth.php>. Acesso em: 16 jun. 2024.

BARROS, Pedro. Qual a diferença entre Git e GitHub?, 15 nov. 2021. Disponível em: <https://www.driven.com.br/blog/qual-a-diferenca-entre-git-e-github>. Acesso em: 16 jun. 2024.

VILLAIN, Mateus. Figma: o que é a ferramenta, Design e uso, 20 out. 2022. Disponível em: <https://www.alura.com.br/artigos/figma>. Acesso em: 16 jun. 2024.

OLIVEIRA, Ricardo de. OBD II: Conectividade ajuda a saber mais sobre o carro, 04 mar. 2022. Disponível em: <https://www.noticiasautomotivas.com.br/obd-ii-conectividade ajuda-motorista-a-saber-mais-sobre-o-carro>. Acesso em: 04 dez. 2022.

MACORATTI, Jose. NET MAUI - Criando seu primeiro projeto, 13 out. 2021. Disponível em: <https://www.macoratti.net/21/10/maui_primproj1.htm>. Acesso em: 16 jun. 2024.

DZHELEKARSKI, Peter. READING AND INTERPRETING DIAGNOSTIC DATA FROM VEHICLE OBDII SYSTEM. 2005. 7 f. - Faculty of Electronic Engineering and Technologies, Technical University of Sofia. Disponível em: <http://ecad.tu-sofia.bg/et/2005/pdf/Paper098-P_Dzhelekarski2.pdf>. Acesso em: 16 jun. 2024.

FONG, Andrew. OBD2 PID READER. Disponível em: <https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1089&context=cpesp>. Acesso em: 16 jun. 2024.

OSS, Marcelo. LEITURA OBD2 ATRAVÉS DE SMARTPHONE. 2018. 79 f. –Universidade Tecnológica Federal Do Paraná, Curso De Especialização Em Sistemas Embarcados Para Indústria Automotiva. Disponível em: <http://riut.utfpr.edu.br/jspui/bitstream/1/19812/1/CT_CESEB_IV_2018_07.pdf>. Acesso em: 4 dez. 2022.

NEPOMUCENO DE CASTILHO, Guilherme. et al. ESTUDO DETALHADO DA EVOLUÇÃO DO TURBOCOMPRES- SOR E COMO A SOBREALIMENTAÇÃO AUMENTA A EFICIÊN- CIA VOLUMÉTRICA DO MOTOR DE COMBUSTÃO. [s.l: s.n.]. Disponível em: <https://fatecsantoandre.edu.br/upload/663027c208538.pdf>. Acesso em: 15 jun. 2024.

OBD2 Frame Format or Message structure | OBD2 PIDs table. Disponível em: <https://www.rfwireless-world.com/Terminology/OBD2-Frame-format.html>. Acesso em: 15 jun. 2024.

Real-Time Fuel Consumption Monitoring from the OBD System. Disponível em: <https://www.windmill.co.uk/fuel.html>. Acesso em: 15 jun. 2024.

X-ENGINEER.ORG. How to calculate wheel and vehicle speed from engine speed – x-engineer.org. Disponível em: <https://x-engineer.org/calculate-wheel-vehicle-speed-engine-speed>. Acesso em: 15 jun. 2024.

BARNHILL, Brian. Volumetric Efficiency 101. Disponível em: <https://tunertools.com/pages/ve101>. Acesso em: 15 jun. 2024.

BARNHILL, Brian. Load Control and Calculation. Disponível em: <https://tunertools.com/pages/load-control-and-calculation>. Acesso em: 15 jun. 2024.

LIGHTNER, Bruce. MAP- and MAF-Based Air/Fuel Flow Calculator. Disponível em: <https://www.lightner.net/obd2guru/IMAP_AFcalc.html>. Acesso em: 15 jun. 2024.

OBD2 Explained - A Simple Intro (2023). Disponível em: <https://www.csselectronics.com/pages/obd2-explained-simple-intro>. Acesso em: 15 jun. 2024.

OBD2 PID Overview [Lookup/Converter Tool, Table, CSV, DBC]. Disponível em: <https://www.csselectronics.com/pages/obd2-pid-table-on-board-diagnostics-j1979>. Acesso em: 15 jun. 2024.

ELM327: What Is The Best Genuine ELM327 Adapters 2024 (Bluetooth/Wifi/USB)?. Disponível em <https://www.obdadvisor.com/elm327/>. Acesso em: 29 jun. 2024.