Aplicações Empresariais com

.NET MAUI

/

Luís Simões da Cunha, 2025

Índice

Parte 1: Fundamentos de .NET MAUI e MVVM

Bem-vindo à Parte 1 deste tutorial! Aqui, vamos explorar os fundamentos do **.NET MAUI** e o padrão **MVVM**, usando exemplos práticos e simples para construir uma base sólida. Nosso objetivo é apresentar esses conceitos de forma clara, reduzindo a carga cognitiva e ajudando você a ganhar confiança inicial no desenvolvimento de aplicações multiplataforma. Vamos criar uma aplicação básica de lista de tarefas para ilustrar cada tópico.

Introdução a .NET MAUI

O que é .NET MAUI?

O **.NET MAUI** (Multi-platform App UI) é um framework de código aberto desenvolvido pela Microsoft que permite criar aplicações nativas para múltiplas plataformas — como Android, iOS, macOS e Windows — a partir de uma única base de código. Ele é a evolução do Xamarin.Forms e integra-se ao ecossistema .NET, oferecendo uma maneira eficiente de compartilhar código enquanto mantém a aparência e o comportamento nativos de cada plataforma.

Por que usar .NET MAUI?

- **Produtividade**: Escreva o código uma vez e reutilize-o em várias plataformas, economizando tempo e esforço.
- **Desempenho**: Acesse APIs nativas para garantir um desempenho otimizado em cada sistema operacional.
- **Suporte**: Conte com uma vasta comunidade de desenvolvedores e o respaldo oficial da Microsoft.

Com o .NET MAUI, você pode construir desde aplicativos simples até soluções empresariais complexas, tudo com uma abordagem unificada.

Padrão MVVM

O padrão **MVVM** (Model-View-ViewModel) é uma arquitetura de design que separa a lógica de negócio da interface do usuário, tornando o código mais organizado, testável e fácil de manter. Ele é composto por três componentes principais:

- **Model**: Representa os dados e a lógica de negócio. É independente da interface.
- **View**: É a interface do usuário (geralmente definida em XAML), responsável por exibir os dados e capturar interações do usuário.
- **ViewModel**: Faz a ponte entre o Model e a View, contendo a lógica de apresentação e expondo dados e comandos para a interface.

Exemplo Básico: Lista de Tarefas

Imagine uma aplicação simples onde o usuário pode adicionar tarefas a uma lista. Vamos usar este exemplo para entender o MVVM na prática:

- O **Model** será uma classe `Tarefa` com uma propriedade `Descricao`.
- O **ViewModel** gerenciará a lista de tarefas e os comandos para adicionar novas tarefas.
- A **View** exibirá uma caixa de texto para inserir tarefas, um botão para adicioná-las e uma lista para mostrá-las.

Vinculação de Dados

A **vinculação de dados** (data binding) é o mecanismo que conecta a interface aos dados do ViewModel. Quando os dados no ViewModel mudam, a interface é atualizada automaticamente, e vice-versa, graças ao suporte do .NET MAUI a esse recurso.

Exemplo: Caixa de Texto para Nome

Suponha que queremos uma caixa de texto onde o usuário insere seu nome. No ViewModel, criamos uma propriedade `Nome` que será vinculada a essa caixa:

- No ViewModel: Uma propriedade que notifica mudanças.

- Na View: Um campo de texto (como um `Entry`) vinculado a essa propriedade. Isso permite que o nome digitado seja refletido no ViewModel sem código manual na interface.

Comandos e Comportamentos

Comandos

Os **comandos** permitem que ações do usuário, como clicar em um botão, sejam vinculadas a métodos no ViewModel, mantendo a lógica fora da camada de interface. No .NET MAUI, usamos a interface `ICommand` para isso.

Exemplo: Botão para Adicionar Tarefa

Vamos criar um botão "Adicionar Tarefa" que, ao ser clicado, chama um método no ViewModel para incluir a tarefa na lista. O comando será vinculado ao botão na View, eliminando a necessidade de código no code-behind.

Comportamentos

Os **comportamentos** (behaviors) adicionam funcionalidades a controles sem subclassificá-los. Por exemplo, podemos usar um comportamento para conectar um evento (como um toque) a um comando no ViewModel, aumentando a flexibilidade e a reutilização do código.

Exemplo Prático: Criar uma Lista de Tarefas

Agora, vamos construir a aplicação de lista de tarefas passo a passo. O usuário poderá inserir uma tarefa numa caixa de texto, clicar em "Adicionar" e ver a lista atualizada.

1. Model

Crie uma classe simples para representar uma tarefa:

```
public class Tarefa
{
    public string Descricao { get; set; }
}
```

2. ViewModel

Crie um ViewModel para gerenciar a lista de tarefas, com uma propriedade para a nova tarefa e um comando para adicioná-la:

```
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Windows.Input;

public class ListaTarefasViewModel : INotifyPropertyChanged
{
    private string _novaTarefa;
    public ObservableCollection<Tarefa> Tarefas { get; } = new
ObservableCollection<Tarefa>();

    public string NovaTarefa
    {
        get => _novaTarefa;
        set
        {
            _novaTarefa = value;
            OnPropertyChanged(nameof(NovaTarefa));
```

```
        }
    }

    public ICommand AdicionarTarefaCommand { get; }

    public ListaTarefasViewModel()
    {
        AdicionarTarefaCommand = new Command(AdicionarTarefa,
PodeAdicionarTarefa);
    }

    private void AdicionarTarefa()
    {
        if (!string.IsNullOrWhiteSpace(NovaTarefa))
        {
            Tarefas.Add(new Tarefa { Descricao = NovaTarefa });
            NovaTarefa = string.Empty; // Limpa o campo após adicionar
        }
    }

    private bool PodeAdicionarTarefa()
    {
        return !string.IsNullOrWhiteSpace(NovaTarefa); // Só habilita o
botão se houver texto
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

3. View (XAML)

Crie uma página com uma caixa de texto, um botão e uma lista para exibir as tarefas:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ListaTarefasApp.ListaTarefasPage">
    <StackLayout>
        <Entry Text="{Binding NovaTarefa}" Placeholder="Nova Tarefa" />
        <Button Text="Adicionar" Command="{Binding
AdicionarTarefaCommand}" />
        <CollectionView ItemsSource="{Binding Tarefas}">
            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding Descricao}" />
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>
    </StackLayout>
</ContentPage>
```

Explicação

- O `Entry` está vinculado à propriedade `NovaTarefa` do ViewModel, permitindo que o texto digitado seja capturado.
- O `Button` usa o comando `AdicionarTarefaCommand` para adicionar a tarefa à lista.
- O `CollectionView` exibe as tarefas da coleção `Tarefas`, atualizando-se automaticamente quando novos itens são adicionados, graças ao uso de `ObservableCollection`.

Desafio: Adicionar um Botão para Limpar a Lista

Vamos adicionar um botão "Limpar Lista" que remove todas as tarefas. Este desafio ajuda a refletir sobre como o ViewModel gerencia o estado da aplicação.

Solução

1. **No ViewModel**: Adicione um novo comando para limpar a lista:

```
public ICommand LimparTarefasCommand { get; }

public ListaTarefasViewModel()
{
    AdicionarTarefaCommand = new Command(AdicionarTarefa,
PodeAdicionarTarefa);
    LimparTarefasCommand = new Command(LimparTarefas); // Novo comando
}

private void LimparTarefas()
{
    Tarefas.Clear();
}
```

2. **Na View**: Adicione um botão vinculado ao novo comando:

```
<Button Text="Limpar Lista" Command="{Binding LimparTarefasCommand}" />
```

Reflexão

- O **ViewModel** mantém a responsabilidade de gerenciar a lógica (limpar a lista), preservando a separação entre a interface e a lógica de negócio.
- A **View** reflete automaticamente a mudança na coleção `Tarefas` devido à vinculação de dados e ao uso de `ObservableCollection`, sem necessidade de intervenção manual na interface.
- Isso demonstra como o MVVM facilita a manutenção e a escalabilidade do código.

Conclusão

Nesta Parte 1, exploramos os fundamentos do **.NET MAUI** e do padrão **MVVM**, criando uma aplicação prática de lista de tarefas. Aprendemos como usar vinculação de dados para conectar a interface ao ViewModel e como implementar comandos para ações do usuário, tudo sem poluir a camada de interface com lógica. O desafio de adicionar um botão "Limpar Lista" reforçou esses conceitos, mostrando a flexibilidade do padrão.

Este tutorial estabelece uma base sólida para tópicos mais avançados, como injeção de dependência e navegação, que abordaremos nas próximas partes. Continue praticando com o exemplo fornecido e experimente adicionar mais funcionalidades, como remover tarefas específicas ou marcar como concluídas!

Parte 2: Injeção de Dependência e Comunicação entre Componentes

Nesta parte, vamos explorar dois conceitos fundamentais para criar aplicações modulares e fáceis de manter no .NET MAUI, utilizando o padrão MVVM: **Injeção de Dependência** e **Comunicação entre Componentes**. Vamos aplicar esses conceitos em um exemplo prático com uma tela de login, introduzindo um serviço de autenticação simulado e usando o **Messenger** para atualizar a interface de forma desacoplada.

O objetivo é mostrar como organizar o código de maneira acessível, evitando dependências rígidas e promovendo a reutilização, tudo isso com exemplos práticos que você pode seguir.

Injeção de Dependência

O que é Injeção de Dependência?

Injeção de Dependência (DI) é um padrão de design que permite que uma classe receba suas dependências (como serviços ou objetos que ela precisa) de fora, em vez de criá-las internamente. Isso reduz o acoplamento entre as partes da aplicação, tornando o código mais flexível, testável e fácil de modificar.

Por exemplo, em vez de um ViewModel criar diretamente um serviço de autenticação, ele recebe esse serviço pronto para uso. Se no futuro quisermos trocar a implementação do serviço (como mudar de uma autenticação simulada para uma real), basta alterar a configuração, sem mexer no ViewModel.

Como usar Injeção de Dependência em .NET MAUI?

No .NET MAUI, a DI é nativamente suportada por meio do contêiner de dependências integrado. A configuração é feita no arquivo `MauiProgram.cs`, onde registramos os serviços e ViewModels que a aplicação usará. O framework então "injeta" automaticamente essas dependências onde elas forem necessárias.

Vamos ver isso na prática mais adiante, no exemplo do serviço de autenticação.

Serviços e Interfaces

Para criar código reutilizável e modular, é uma boa prática definir **interfaces** para os serviços e implementá-las em classes separadas. Isso permite que diferentes partes da aplicação usem o serviço sem depender de uma implementação específica.

Exemplo: Serviço de Autenticação

Vamos criar um serviço simples para gerenciar autenticação. Primeiro, definimos uma interface chamada `IAuthService`:

```
public interface IAuthService
{
    bool Authenticate(string username, string password);
}
```

Depois, criamos uma implementação simulada chamada `MockAuthService`, que aceita qualquer username desde que a senha seja "123":

```
public class MockAuthService : IAuthService
{
    public bool Authenticate(string username, string password)
    {
        // Simula autenticação: aceita qualquer username com senha "123"
        return password == "123";
    }
}
```

Essa abordagem nos permite substituir o `MockAuthService` por um serviço real (como um que consulta uma API) no futuro, sem alterar o resto do código.

Comunicação entre Componentes

Em aplicações MVVM, diferentes partes do código (como ViewModels ou Views) frequentemente precisam se comunicar. Para evitar dependências diretas, usamos o **Messenger**, uma ferramenta do **CommunityToolkit.Mvvm** que permite enviar mensagens entre componentes de forma desacoplada.

Por exemplo, após um login bem-sucedido, podemos enviar uma mensagem para atualizar a interface, como mudar um texto de boas-vindas, sem que o ViewModel de login precise conhecer o ViewModel da tela principal.

Exemplo Prático: Implementar um Serviço de Autenticação

Vamos criar uma tela de login simples para nossa aplicação. O usuário insere um nome de usuário e senha, clica em um botão de "login" e, se as credenciais forem válidas, exibe uma mensagem de sucesso.

1. Configurar a Injeção de Dependência

No arquivo `MauiProgram.cs`, registramos o serviço de autenticação e o ViewModel:

```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts =>
            {
                fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            });
```

```
        // Registrar o serviço de autenticação como Singleton
        builder.Services.AddSingleton<IAuthService, MockAuthService>();
        // Registrar o ViewModel como Transient (nova instância a cada
uso)
        builder.Services.AddTransient<LoginViewModel>();

        return builder.Build();
    }
}
```

2. Criar o LoginViewModel

O `LoginViewModel` usa o serviço injetado para verificar as credenciais e exibe uma mensagem de resultado:

```
using CommunityToolkit.Mvvm.ComponentModel;
using CommunityToolkit.Mvvm.Input;

public partial class LoginViewModel : ObservableObject
{
    private readonly IAuthService _authService;

    [ObservableProperty]
    private string _username;

    [ObservableProperty]
    private string _password;

    [ObservableProperty]
    private string _message;

    public IRelayCommand LoginCommand { get; }

    // O serviço é injetado via construtor
```

```
    public LoginViewModel(IAuthService authService)
    {
        _authService = authService;
        LoginCommand = new RelayCommand(Login);
    }

    private void Login()
    {
        if (_authService.Authenticate(Username, Password))
        {
            Message = "Login bem-sucedido!";
        }
        else
        {
            Message = "Credenciais inválidas.";
        }
    }
}
```

Aqui, usamos o `[ObservableProperty]` do CommunityToolkit.Mvvm para gerar propriedades automáticas com notificação de mudanças, simplificando o código.

3. Criar a View de Login

No arquivo `LoginPage.xaml`, adicionamos os campos de entrada e o botão:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MinhaApp.LoginPage">
    <StackLayout Padding="20">
        <Entry Text="{Binding Username}" Placeholder="Username" />
        <Entry Text="{Binding Password}" Placeholder="Password"
IsPassword="True" />
        <Button Text="Login" Command="{Binding LoginCommand}" />
        <Label Text="{Binding Message}" />
```

```
    </StackLayout>
</ContentPage>
```

No code-behind (`LoginPage.xaml.cs`), definimos o BindingContext:

```
public partial class LoginPage : ContentPage
{
    public LoginPage(LoginViewModel viewModel)
    {
        InitializeComponent();
        BindingContext = viewModel;
    }
}
```

4. Configurar a Navegação Inicial

No `App.xaml.cs`, definimos a `LoginPage` como página inicial e usamos a DI para instanciá-la:

```
public partial class App : Application
{
    public App(IServiceProvider serviceProvider)
    {
        InitializeComponent();
        MainPage = new
NavigationPage(serviceProvider.GetService<LoginPage>());
    }
}
```

Com isso, ao rodar a aplicação, você verá a tela de login. Insira qualquer username e a senha "123" para ver a mensagem de sucesso.

Desafio: Enviar uma Mensagem com o Messenger para Atualizar a UI Após o Login

Agora, vamos expandir o exemplo para usar o **Messenger** e atualizar um texto de boas-vindas em outra parte da aplicação após o login.

1. Definir a Mensagem

Crie uma classe para a mensagem de login bem-sucedido:

```
using CommunityToolkit.Mvvm.Messaging.Messages;

public class LoginSuccessMessage : ValueChangedMessage<string>
{
    public LoginSuccessMessage(string username) : base(username) { }
}
```

2. Modificar o LoginViewModel

Atualize o método `Login` para enviar a mensagem usando o `WeakReferenceMessenger`:

```
private void Login()
{
    if (_authService.Authenticate(Username, Password))
    {
        Message = "Login bem-sucedido!";
        // Envia a mensagem com o username
        WeakReferenceMessenger.Default.Send(new
LoginSuccessMessage(Username));
    }
    else
    {
        Message = "Credenciais inválidas.";
    }
}
```

3. Criar um MainViewModel para Receber a Mensagem

Crie um `MainViewModel` que escuta a mensagem e atualiza o texto de boas-vindas:

```
using CommunityToolkit.Mvvm.ComponentModel;
using CommunityToolkit.Mvvm.Messaging;

public partial class MainViewModel : ObservableObject,
IRecipient<LoginSuccessMessage>
{
    [ObservableProperty]
    private string _welcomeMessage;

    public MainViewModel()
    {
        WelcomeMessage = "Por favor, faça login.";
        // Registra este ViewModel para receber mensagens do tipo
LoginSuccessMessage

WeakReferenceMessenger.Default.Register<LoginSuccessMessage>(this);
    }

    public void Receive(LoginSuccessMessage message)
    {
        WelcomeMessage = $"Bem-vindo, {message.Value}!";
    }
}
```

4. Criar a MainPage

No `MainPage.xaml`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MinhaApp.MainPage">
    <StackLayout>
        <Label Text="{Binding WelcomeMessage}" HorizontalOptions="Center"
VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

No `MainPage.xaml.cs`:

```
public partial class MainPage : ContentPage
{
    public MainPage(MainViewModel viewModel)
    {
        InitializeComponent();
        BindingContext = viewModel;
    }
}
```

5. Registrar o MainViewModel no MauiProgram.cs

Adicione:

```
builder.Services.AddTransient<MainViewModel>();
builder.Services.AddTransient<MainPage>();
```

6. Navegar Após o Login

Atualize o `LoginViewModel` para navegar para a `MainPage` após o login:

```
private readonly IAuthService _authService;
private readonly IServiceProvider _serviceProvider;

public LoginViewModel(IAuthService authService, IServiceProvider
serviceProvider)
{
    _authService = authService;
    _serviceProvider = serviceProvider;
    LoginCommand = new RelayCommand(Login);
}

private async void Login()
{
    if (_authService.Authenticate(Username, Password))
    {
        Message = "Login bem-sucedido!";
        WeakReferenceMessenger.Default.Send(new
LoginSuccessMessage(Username));
        // Navega para a MainPage
        var mainPage = _serviceProvider.GetService<MainPage>();
        await
Application.Current.MainPage.Navigation.PushAsync(mainPage);
    }
    else
    {
        Message = "Credenciais inválidas.";
    }
}
```

Agora, ao fazer login com a senha "123", a aplicação navegará para a `MainPage` e exibirá "Bem-vindo, [username]!".

Conclusão

Nesta Parte 2, vimos como:

- **Injeção de Dependência** organiza o código ao fornecer serviços de forma desacoplada, configurados no `MauiProgram.cs`.
- **Serviços e Interfaces** promovem reutilização, como no caso do `IAuthService` e `MockAuthService`.
- **Comunicação entre Componentes** com o **Messenger** permite atualizar a UI sem dependências rígidas.
- No **exemplo prático**, implementamos um botão de login com autenticação simulada e usamos o Messenger para atualizar a interface.

Esses conceitos são a base para aplicações escaláveis e manuteníveis. Experimente adicionar mais funcionalidades, como validar entradas ou navegar para outras páginas, para praticar!

Parte 3: Navegação e Validação em .NET MAUI

Nesta parte do tutorial, vamos abordar dois conceitos fundamentais no desenvolvimento de aplicações com .NET MAUI: **navegação entre páginas** e **validação de dados**. Usaremos o padrão MVVM para manter a separação de responsabilidades e construiremos um exemplo prático: um formulário de registo com validação de campos (nome e e-mail obrigatórios) que, ao ser submetido, navega para uma página de confirmação exibindo uma saudação personalizada. Vamos tornar esses conceitos fáceis de entender com exemplos que simulam fluxos reais.

Navegação em .NET MAUI

O que é Navegação?

A navegação em .NET MAUI permite que os utilizadores se movam entre diferentes páginas da aplicação, como de uma tela de login para uma tela principal após autenticação. O framework suporta vários tipos de navegação, mas aqui vamos focar na **navegação hierárquica**, que usa uma pilha de páginas para avançar e retroceder.

Como Implementar Navegação Hierárquica

Para gerenciar a navegação hierárquica, usamos a classe `NavigationPage`, que mantém uma pilha de páginas. Vamos configurar a navegação inicial e navegar entre páginas.

2. **Configurar a Navegação Inicial:** No ficheiro `App.xaml.cs`, definimos a página inicial dentro de um `NavigationPage`:

```
public App()
{
    InitializeComponent();
    MainPage = new NavigationPage(new RegistroPage());
}
```

3. **Navegar para Outra Página:** Para manter o padrão MVVM, criaremos um serviço de navegação que será usado pelo ViewModel para acionar a navegação.

Serviço de Navegação

Para separar a lógica de navegação da View, criamos um serviço de navegação injetável.

4. **Interface do Serviço:**

```
public interface INavigationService
{
    Task NavigateToAsync(string route);
}
```

5. **Implementação:**

```
public class NavigationService : INavigationService
{
    public async Task NavigateToAsync(string route)
    {
        await Shell.Current.GoToAsync(route);
    }
}
```

Nota: Aqui usamos o `Shell` para navegação. Se não estiver usando Shell, pode-se usar `Navigation.PushAsync` com `NavigationPage`.

Passagem de Parâmetros

Muitas vezes, precisamos enviar dados entre páginas, como o nome do utilizador após o registo. Vamos usar query strings para isso.

Exemplo: Passar o Nome do Utilizador

6. **Navegar com Parâmetros:** No ViewModel, ao acionar a navegação:

```
await
_navigationService.NavigateToAsync($"confirmacao?nome={nome}");
```

7. **Receber Parâmetros:** Na página de destino (ViewModel da confirmação), usamos o atributo `[QueryProperty]`:

```
public class ConfirmacaoViewModel : INotifyPropertyChanged
{
    private string _nome;
    [QueryProperty(nameof(Nome), "nome")]
    public string Nome
    {
        get => _nome;
        set
        {
            _nome = value;
            OnPropertyChanged();
        }
    }

    public string Saudacao => $"Olá, {Nome}!";

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged([CallerMemberName] string
propertyName = null)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

Validação de Dados

A validação garante que os dados inseridos pelo utilizador são corretos. No MVVM, essa lógica fica no ViewModel. Vamos criar uma classe base para validação e aplicar regras simples, como verificar se um campo está vazio.

Classe Base para Validação

```
public abstract class ValidatableObject : INotifyPropertyChanged
{
    private string _value;
    public string Value
    {
        get => _value;
        set
        {
            _value = value;
            OnPropertyChanged();
            Validate();
        }
    }

    public bool IsValid { get; private set; }
    public string ErrorMessage { get; private set; }

    protected abstract void Validate();

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged([CallerMemberName] string
propertyName = null)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

Regras de Validação

- **Nome:** Não pode estar vazio.
- **E-mail:** Não pode estar vazio e deve conter "@".

Exemplo Prático: Formulário de Registo com Validação

Vamos criar um formulário onde o utilizador insere nome e e-mail, com validação que exibe mensagens de erro se os campos estiverem vazios ou inválidos, e navega para uma página de confirmação ao submeter.

1. ViewModel do Registo

```
public class RegistroViewModel : INotifyPropertyChanged
{
    private readonly INavigationService _navigationService;

    public RegistroViewModel(INavigationService navigationService)
    {
        _navigationService = navigationService;
        Nome = new NomeValidatableObject();
        Email = new EmailValidatableObject();
        RegistrarCommand = new Command(Registrar, PodeRegistrar);
    }

    public ValidatableObject Nome { get; }
    public ValidatableObject Email { get; }
    public ICommand RegistrarCommand { get; }

    private void Registrar()
    {
        if (Nome.IsValid && Email.IsValid)
        {

_navigationService.NavigateToAsync($"confirmacao?nome={Nome.Value}");
        }
        else
        {
            // Mostrar mensagem de erro (ex.: via alert)
        }
    }

    private bool PodeRegistrar()
    {
        return Nome.IsValid && Email.IsValid;
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged([CallerMemberName] string
propertyName = null)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

2. Classes de Validação

```
public class NomeValidatableObject : ValidatableObject
{
    protected override void Validate()
    {
        if (string.IsNullOrWhiteSpace(Value))
        {
            IsValid = false;
            ErrorMessage = "O nome é obrigatório.";
        }
        else
        {
            IsValid = true;
            ErrorMessage = string.Empty;
        }
    }
}
```

```
public class EmailValidatableObject : ValidatableObject
{
    protected override void Validate()
    {
        if (string.IsNullOrWhiteSpace(Value))
        {
            IsValid = false;
            ErrorMessage = "O e-mail é obrigatório.";
        }
        else if (!Value.Contains("@"))
        {
            IsValid = false;
            ErrorMessage = "E-mail inválido.";
        }
        else
        {
            IsValid = true;
            ErrorMessage = string.Empty;
        }
    }
}
```

3. View do Registo (XAML)

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MinhaApp.RegistroPage">
    <StackLayout>
        <Entry Text="{Binding Nome.Value}" Placeholder="Nome" />
        <Label Text="{Binding Nome.ErrorMessage}" TextColor="Red" />
        <Entry Text="{Binding Email.Value}" Placeholder="E-mail" />
        <Label Text="{Binding Email.ErrorMessage}" TextColor="Red" />
        <Button Text="Registrar" Command="{Binding RegistrarCommand}" />
    </StackLayout>
</ContentPage>
```

4. View da Confirmação (XAML)

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MinhaApp.ConfirmacaoPage">
    <StackLayout>
        <Label Text="{Binding Saudacao}" />
    </StackLayout>
</ContentPage>
```

Desafio: Página de Confirmação

O desafio foi adicionar uma página de confirmação que recebe o nome inserido e exibe uma saudação personalizada. Isso foi cumprido no exemplo acima:

- O `RegistroViewModel` navega para a página de confirmação passando o nome como parâmetro.
- O `ConfirmacaoViewModel` recebe o nome via `[QueryProperty]` e exibe a saudação "Olá, [nome]!".

Conclusão

Nesta Parte 3, aprendemos:

- **Navegação**: Como mover-se entre páginas com um serviço de navegação e passar parâmetros usando query strings.
- **Validação**: Como adicionar regras simples no ViewModel para verificar dados e exibir mensagens de erro.

Com esses conceitos, você pode criar fluxos de aplicação intuitivos e garantir que os dados inseridos sejam válidos. Experimente expandir o exemplo adicionando mais campos ao formulário ou validações mais complexas, como verificar o formato do e-mail com expressões regulares!

Parte 4: Acesso a Dados Remotos e Autenticação

Acesso a APIs REST

Para consumir dados de um serviço web (API REST), usamos o `HttpClient` no .NET MAUI. Vamos criar um serviço que busca uma lista de produtos de uma API simulada.

8. **Modelo de Produto:** Crie uma classe simples para representar os dados:

```
public class Produto
{
    public int Id { get; set; }
    public string Nome { get; set; }
    public decimal Preco { get; set; }
}
```

9. **Serviço de API:** Defina uma interface e uma implementação para buscar os produtos:

```
public interface IApiService
{
    Task<List<Produto>> GetProdutosAsync(string token);
}

public class ApiService : IApiService
{
    private readonly HttpClient _httpClient;

    public ApiService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public async Task<List<Produto>> GetProdutosAsync(string token)
    {
        _httpClient.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", token);
        var response = await
_httpClient.GetAsync("https://api.mock.com/produtos");
        response.EnsureSuccessStatusCode();
        var json = await response.Content.ReadAsStringAsync();
        return JsonSerializer.Deserialize<List<Produto>>(json);
    }
}
```

Aqui, o token é enviado no cabeçalho da requisição para autenticar o acesso à API.

Autenticação e Autorização

Usaremos tokens JWT (JSON Web Tokens) para proteger o acesso à API. Após o login, o token será armazenado localmente e usado em todas as requisições.

1. **Serviço de Autenticação:** Crie uma interface e uma classe para gerenciar login e logout:

```
public interface IAuthService
{
    Task<string> LoginAsync(string username, string password);
    void Logout();
    Task<string> GetTokenAsync();
}

public class AuthService : IAuthService
{
    private readonly HttpClient _httpClient;
    private string _token;

    public AuthService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public async Task<string> LoginAsync(string username, string
password)
    {
        var loginData = new { Username = username, Password =
password };
        var json = JsonSerializer.Serialize(loginData);
        var content = new StringContent(json, Encoding.UTF8,
"application/json");
        var response = await
_httpClient.PostAsync("https://api.mock.com/login", content);
        response.EnsureSuccessStatusCode();
        var responseJson = await
response.Content.ReadAsStringAsync();
        var tokenResponse =
JsonSerializer.Deserialize<TokenResponse>(responseJson);
        _token = tokenResponse.Token;
```

```
        await SecureStorage.SetAsync("auth_token", _token); //
Armazena o token de forma segura
        return _token;
    }

    public void Logout()
    {
        _token = null;
        SecureStorage.Remove("auth_token"); // Remove o token ao
fazer logout
    }

    public async Task<string> GetTokenAsync()
    {
        if (string.IsNullOrEmpty(_token))
        {
            _token = await SecureStorage.GetAsync("auth_token");
        }
        return _token;
    }
}

public class TokenResponse
{
    public string Token { get; set; }
}
```

O `SecureStorage` é usado para persistir o token de forma segura no dispositivo.

Cache de Dados

Para melhorar a performance, vamos armazenar os produtos localmente usando o `Preferences`. Isso permite carregar os dados rapidamente sem depender da rede o tempo todo.

1. **Serviço de Cache:**

```
public class CacheService
{
    public void SaveProdutos(List<Produto> produtos)
    {
        var json = JsonSerializer.Serialize(produtos);
        Preferences.Set("produtos_cache", json);
    }

    public List<Produto> GetProdutos()
    {
        var json = Preferences.Get("produtos_cache", string.Empty);
        if (string.IsNullOrEmpty(json))
        {
            return new List<Produto>();
        }
        return JsonSerializer.Deserialize<List<Produto>>(json);
    }
}
```

Exemplo Prático: Lista de Produtos Após Autenticação

Vamos criar uma página que exibe os produtos após o login, com botões para carregar da API ou do cache.

ViewModel da Lista de Produtos:

```
public class ProdutosViewModel : INotifyPropertyChanged
{
    private readonly IApiService _apiService;
    private readonly AuthService _authService;
    private readonly CacheService _cacheService;
    private List<Produto> _produtos;

    public ProdutosViewModel(IApiService apiService, AuthService
authService, CacheService cacheService)
    {
        _apiService = apiService;
        _authService = authService;
        _cacheService = cacheService;
        CarregarProdutosCommand = new Command(async () => await
CarregarProdutosAsync());
        CarregarDoCacheCommand = new Command(CarregarDoCache);
    }

    public List<Produto> Produtos
    {
        get => _produtos;
        set
        {
            _produtos = value;
            OnPropertyChanged();
        }
    }

    public ICommand CarregarProdutosCommand { get; }
    public ICommand CarregarDoCacheCommand { get; }

    private async Task CarregarProdutosAsync()
    {
        var token = await _authService.GetTokenAsync();
        if (string.IsNullOrEmpty(token))
        {
            // Redirecionar para login (exemplo: await
Shell.Current.GoToAsync("//login"))
```

```
            return;
        }
        Produtos = await _apiService.GetProdutosAsync(token);
        _cacheService.SaveProdutos(Produtos);
    }

    private void CarregarDoCache()
    {
        Produtos = _cacheService.GetProdutos();
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged([CallerMemberName] string
propertyName = null)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

View da Lista de Produtos (XAML):

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MinhaApp.ProdutosPage">
    <StackLayout>
        <Button Text="Carregar Produtos" Command="{Binding
CarregarProdutosCommand}" />
        <Button Text="Carregar do Cache" Command="{Binding
CarregarDoCacheCommand}" />
        <CollectionView ItemsSource="{Binding Produtos}">
            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <StackLayout>
                        <Label Text="{Binding Nome}" />
                        <Label Text="{Binding Preco,
StringFormat='Preço: {0:C}'}" />
                    </StackLayout>
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>
    </StackLayout>
</ContentPage>
```

Desafio: Implementar Logout

Vamos adicionar um logout que limpe o token e retorne ao ecrã de login.

1. **Adicionar Comando de Logout no ViewModel:**

```
public class ProdutosViewModel : INotifyPropertyChanged
{
    // ... (outros membros existentes)
    private readonly INavigationService _navigationService;
    public ICommand LogoutCommand { get; }

    public ProdutosViewModel(IApiService apiService, AuthService
authService, CacheService cacheService, INavigationService
navigationService)
    {
        _apiService = apiService;
        _authService = authService;
        _cacheService = cacheService;
        _navigationService = navigationService;
        CarregarProdutosCommand = new Command(async () => await
CarregarProdutosAsync());
        CarregarDoCacheCommand = new Command(CarregarDoCache);
        LogoutCommand = new Command(Logout);
    }

    private void Logout()
    {
        _authService.Logout();
        _navigationService.NavigateToAsync("login"); // Exemplo de
navegação
    }

    // ... (outros métodos existentes)
}
```

Nota: `INavigationService` é uma abstração para navegação (pode ser implementada com `Shell` ou outra abordagem).

2. **Adicionar Botão de Logout na View:**

```
<Button Text="Logout" Command="{Binding LogoutCommand}" />
```

3. **Reflexão sobre Segurança:**

    - **Limpeza do Token:** Ao fazer logout, removemos o token do `SecureStorage` para evitar acessos não autorizados.
    - **Armazenamento Seguro:** Usar `SecureStorage` protege o token contra acesso direto no dispositivo.
    - **Expiração do Token:** Em um cenário real, verifique a validade do token e force reautenticação se estiver expirado.

Conclusão

Nesta Parte 4, você aprendeu:

- **Acesso a APIs REST**: Como buscar dados de uma API com `HttpClient`.
- **Autenticação e Autorização**: Uso de tokens JWT para segurança.
- **Cache de Dados**: Armazenamento local com `Preferences` para performance.
- **Exemplo Prático**: Uma lista de produtos integrada com uma API simulada.
- **Desafio**: Logout seguro com limpeza de token e navegação.

Com esses conceitos, você pode criar aplicativos .NET MAUI que interagem com serviços web de forma eficiente e segura. Experimente adicionar mais funcionalidades, como detalhes de produtos ou um carrinho de compras, para aprofundar o aprendizado!

Referência:

Stonis, M. (2022). *Enterprise application patterns using .NET MAUI* (v2.0). Microsoft Developer Division, .NET, and Visual Studio product teams. https://aka.ms/maui-ebook