# Cross-Plattform-Apps mit .NET MAUI entwickeln

Mit C# für Android, iOS, macOS und Windows programmieren

# 10 Navigation und die Shell

Mobile Apps bestehen fast immer aus mehr als einer Seite. Um dem Benutzer den Wechsel der Seiten innerhalb der App zu ermöglichen, gibt es verschiedene Navigationsmöglichkeiten. Diese sehen wir uns in diesem Kapitel an.

Wir starten mit einem kurzen Blick auf die verschiedenen Navigationsstrategien und beschäftigen uns dann mit den Hilfsmitteln, die .NET MAUI zur Implementierung dieser Strategien mitbringt. Anschließend lernen Sie die Shell kennen. Die .NET MAUI Shell ist ein moderner Ansatz zur Lösung unterschiedlicher Navigationsszenarien.

Zum Abschluss des Kapitels widmen wir uns der konkreten Umsetzung der Shell in unserer Beispiel-App.

## ■ 10.1 Navigation in mobilen Apps

Die gängigsten Navigationsstrategien für mobile Apps sind Registerkarten (Tabs), hierarchische Navigation und Navigation über die Seitenleiste, auch Navigation Drawer, Burger-Menü, Side Navigation oder Flyout genannt. In der Praxis werden die verschiedenen Ansätze häufig kombiniert.

### 10.1.1 Registerkarten

Registerkarten (engl. *Tabs*) ermöglichen eine schnelle Navigation zwischen verschiedenen Seiten der App mit einem Klick. Je Betriebssystem wird die horizontale Registerkartenleiste automatisch am oberen Bildschirmrand (Android und Windows) oder am unteren Bildschirmrand (iOS) angezeigt.

Jede Registerkarte hat einen Titel und ein optionales Bild. Beim Klick auf einen Tab wird automatisch innerhalb des Inhaltsbereichs der dem Tab zugewiesene Inhalt, in der Regel eine eigene Seite, angezeigt. Sie müssen dazu keinen eigenen Navigationscode schreiben.

Aufgrund des begrenzten horizontalen Platzes auf dem Bildschirm von mobilen Endgeräten bietet sich die Navigation mit Tabs immer dann an, wenn Sie nur wenige Menüpunkte in Ihrer Navigationsleiste platzieren möchten. Registerkartenleisten sind zwar scrollbar,

wenn der horizontale Platz zur Darstellung aller Optionen nicht ausreicht, für den Benutzer ist die Bedienung einer scrollbaren Tab-Navigation allerdings nicht komfortabel.

Bild 10.1 zeigt Registerkarten unter Android, iOS und Windows. Wie bereits beschrieben, befinden sich die Registerkarten unter iOS unten und unter Android und Windows oben. Der Grund hierfür ist, dass Android-Handys im unteren Bildschirmbereich häufig eine Navigationsleiste zur gerätespezifischen Navigation haben (Zurück, Home, letzte Apps). Um das Risiko zu minimieren, beim Anwählen der Registerkarten versehentlich die gerätespezifische Navigationsleiste anzutippen und dadurch die App zum Beispiel zu verlassen, wurde die Registerkartenleiste unter Android am oberen Bildschirmrand platziert.

Als Entwickler müssen Sie sich nicht um die Platzierung der Registerkartenleiste kümmern. Da .NET MAUI unter der Haube die nativen Oberflächenelemente der jeweiligen Betriebssysteme rendert, werden Tabs automatisch an der richtigen Position dargestellt.


**Bild 10.1** Registerkarten unter Android, iOS und Windows

Mit .NET MAUI können Tabs über zwei Wege erzeugt werden, über die `TabbedPage` und über die .NET MAUI Shell. In diesem Abschnitt sehen wir uns die `TabbedPage` an. Die Shell wird später in diesem Kapitel noch ausführlich beschrieben.

Die .NET-MAUI-TabbedPage ist eine Containerseite, die weitere Seiten in den Inhaltsbereich lädt. Das bedeutet, dass Sie neben den eigentlichen Detailseiten noch eine zusätzliche Seite benötigen, die `TabbedPage`. Um das Beispiel aus Bild 10.1 zu erzeugen, wurden ausgehend

von einem neuen Projekt im ersten Schritt zwei Detailseiten mit dem Namen *Page1.xaml* und *Page2.xaml* angelegt.

Beide Seiten sind einfache Inhaltsseiten. Die einzige Besonderheit ist die Angabe der Eigenschaften `Title` und `IconImageSource` (beide sind in Listing 10.1 fett gedruckt), die die Beschriftung der Registerkarten auf der `TabbedPage` steuern.

**Listing 10.1** Eine einfache ContentPage, vorbereitet zur Darstellung in einer TabbedPage

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       x:Class="TabsSample.Page1"
       Title="Startseite"
       IconImageSource="home.png">
  <ContentPage.Content>
    <StackLayout>
      <Label Text="Startseite" FontSize="Title"
        VerticalOptions="CenterAndExpand"
        HorizontalOptions="CenterAndExpand" />
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

**Icons in Registerkarten**

Wie Sie in Listing 10.1 sehen, wird das Icon einer Registerkarte über die Eigenschaft `IconImageSource` der darzustellenden Seite gesteuert. Dieser Eigenschaft weisen Sie den Namen eines Bilds zu, das im Ordner *Resources\Images* liegen muss. Den detaillierten Umgang mit Icons und Bildern lernen Sie in einem späteren Kapitel. Wenn Sie jetzt schon mehr erfahren möchten, dann werfen Sie einfach einen Blick in das Beispielprojekt *TabsSample* dieses Kapitels. ■

Um über Registerkarten auf die beiden angelegten Inhaltsseiten navigieren zu können, benötigen wir nun noch eine `TabbedPage`. Bei der `TabbedPage` handelt es sich um einen speziellen .NET-MAUI Seitentyp. Da es keine eigene Dateivorlage für eine `TabbedPage` gibt, wurde im vorliegenden Beispiel einfach der Quelltext der von der Projektvorlage generierten Datei *MainPage.xaml* durch den Code aus Listing 10.2 ersetzt.

**Listing 10.2** Definition von Registerkarten in einer TabbedPage

```
<?xml version="1.0" encoding="utf-8" ?>
<TabbedPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:local="clr-namespace:TabsSample "
       x:Class="TabsSample.MainPage">
  <local:Page1/>
  <local:Page2/>
</TabbedPage>
```

Wie Sie anhand der fett gedruckten Zeilen sehen können, wurde als Wurzelelement statt einer `ContentPage` eine `TabbedPage` genutzt. Außerdem wurde ein zusätzlicher Namens-

raum mit dem Namen `local` deklariert. In diesem Namensraum befinden sich die beiden zuvor angelegten Seiten *Page1.xaml* und *Page2.xaml*. Beide Seiten wurden als Kindelemente zum Element `TabbedPage` hinzugefügt. Das vorgestellte Markup reicht bereits aus, um die vollständig funktionsfähige Registerkartennavigation aufzubauen. Da die Datei *MainPage.xaml* nun allerdings keine `ContentPage` mehr ist, sondern eine `TabbedPage`, muss im Code-Behind die Zeile `public partial class MainPage : ContentPage` wie folgt geändert werden: `public partial class MainPage : TabbedPage`. Außerdem muss sämtlicher durch die Projektvorlage angelegter Quellcode abgesehen vom Konstruktor aus der Klasse gelöscht werden.

Da die Startseite einer neu angelegten App standardmäßig eine Shell ist, und eine TabbedPage kein Kindelement einer Shell sein darf, muss in der Datei App.xaml.cs außerdem die Zeile `MainPage = new AppShell();` wie folgt geändert werden: `MainPage = new MainPage();`.Fehlt diese Änderung, dann kommt es zur Laufzeit zu einer Exception, wodurch die App abstürzt.

## 10.1.2 Hierarchische Navigation

Bei der hierarchischen Navigation navigiert der Benutzer linear von einer Seite zur nächsten und wieder zurück. Dabei baut er einen Navigationsstapel auf, auf den die Seiten draufgelegt (push) oder heruntergenommen werden (pop).

In Bild 10.2 sehen Sie, wie der Navigationsstapel sich durch die Vorwärtsnavigation verändert. Jede neue Seite wird auf den Stapel draufgelegt. Diese Operation nennt man *push*. Der Ausschnitt des iPhone über dem jeweiligen Stapel zeigt, wie sich die Darstellung der Navigationsleiste der App während der Navigation verändert.

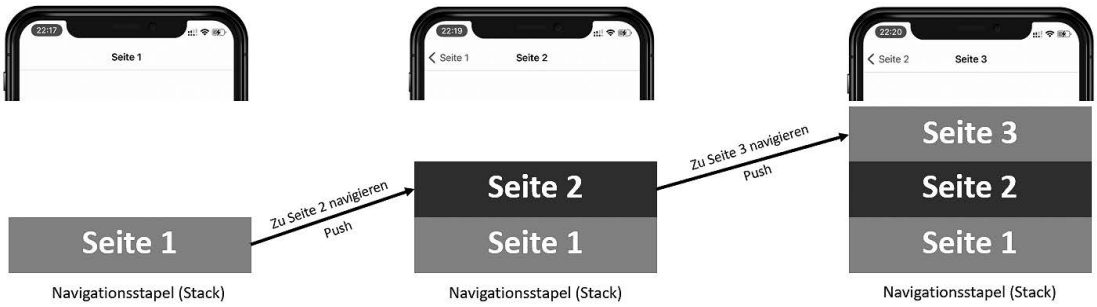


**Bild 10.2** Aufbau des Navigationsstapels durch das Drauflegen von Seiten

Navigiert der Anwender wieder zurück, wird die oberste Seite wieder vom Stapel genommen. Wir sprechen hier von einer *pop*-Operation. Bild 10.3 verdeutlicht die Last-In-First-Out (LIFO)-Arbeitsweise des Stacks.

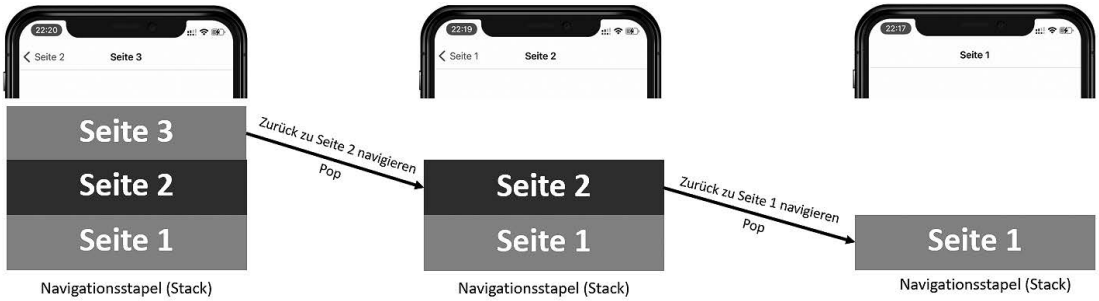


**Bild 10.3** Abbau des Navigationsstapels beim Zurücknavigieren

Unter .NET MAUI lässt sich eine hierarchische Navigation mit einfachen Mitteln umsetzen. Ähnlich wie bei der Registerkartennavigation gibt es auch bei der hierarchischen Navigation eine Containerseite, die das Navigationsgrundgerüst herstellt, die sogenannte `NavigationPage`. Im Gegensatz zur `TabbedPage` müssen Sie für die `NavigationPage` jedoch keine eigene XAML-Datei anlegen. Stattdessen reicht es, die erste Navigationsseite im Quellcode mit einer `NavigationPage` zu umschließen. Bezogen auf das Beispiel zum Theme Registerkarten bedeutet dies, dass Sie in der Datei *App.xaml.cs* die folgenden Änderungen aus Listing 10.3 umsetzen.

**Listing 10.3** Einführung einer NavigationPage zur Umsetzung einer hierarchischen Navigation

```
public App()
{
    InitializeComponent();

    // MainPage = new AppShell();
    MainPage = new NavigationPage(new Page1());
}
```

Sobald diese Änderung vollzogen ist, können Sie an der Eigenschaft `Navigation`, die es bei jeder .NET-MAUI-Seite gibt, die Methode `PushAsync` aufrufen, um zu einer weiteren Seite zu navigieren. Im folgenden Beispielcode sehen Sie eine Ereignisbehandlungsroutine für den Klick auf einen Button im Code-Behind einer Seite, mit der Sie zu einer weiteren Seite navigieren können:

```
private async void Button_OnClicked(object sender, EventArgs e)
{
    await Navigation.PushAsync(new Page2());
}
```

Da die Methode `PushAsync` asynchron ist, sollte sie mit dem Schlüsselwort `await` aufgerufen werden. Das hat zur Folge, dass die umschließende Methode, in diesem Beispiel die Ereignisbehandlungsmethode `Button_OnClicked` mit dem Schlüsselwort `async` markiert werden muss.

Bild 10.4 zeigt die Darstellung einer hierarchischen Navigation via `NavigationPage` unter Android, iOS und Windows. Auf dem iOS-Gerät (Mitte im Bild) wurde bereits navigiert. Daher sehen Sie im linken oberen Bereich in der Navigationsleiste bereits die Schaltfläche zum Zurücknavigieren. Rechts sehen Sie, dass unter Windows auch bereits navigiert wurde.

Im Gegensatz zu iOS wird nur ein kleiner Pfeil links oben in der Titelleiste dargestellt. Eine Beschriftung des Pfeils, wie es unter iOS der Fall ist, fehlt unter Windows.


**Bild 10.4** Hierarchische Navigation unter Android, iOS und Windows

### 10.1.3 Seitenleiste

Die Navigation über eine Seitenleiste, auch als Navigation Drawer, Burger-Menü, Flyout, oder Sidenav bekannt, ist ein beliebtes Navigationsmuster in mobilen Anwendungen.

In der linken oberen Ecke befinden sich, wie man in Bild 10.5 sieht, ein Icon, welches in der Regel aus drei horizontalen Linien besteht. Die drei Linien erinnern optisch an zwei Brötchenhälften mit einer Fleischscheibe in der Mitte, also an einen Hamburger, woher auch der Name Burger-Menü für diese Art der Navigation kommt.

Ein Klick auf das Menü-Icon öffnet die zuvor verborgene Navigationsleiste und lässt sie von der Seite „reinfliegen“ (daher auch der alternative Name Flyout).

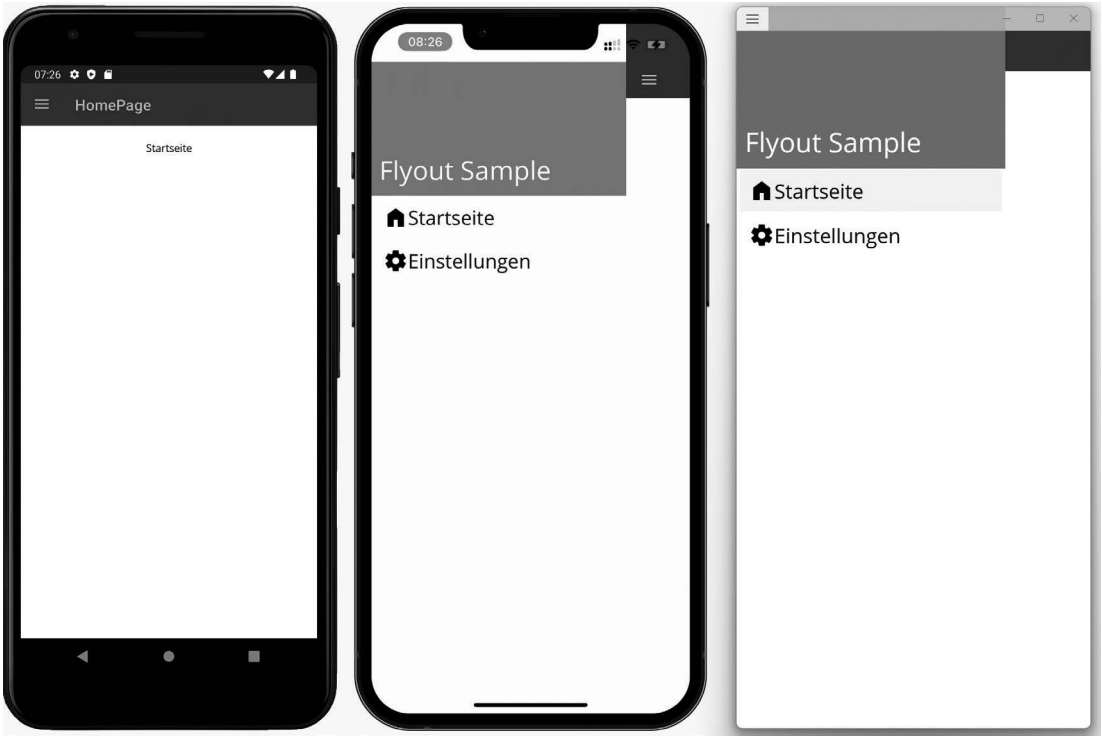


**Bild 10.5** Ein eingeklapptes (links) und aufgeklappte (mitte und rechts) Burger-Menüs

Genau wie bei den Registerkarten gibt es auch bei der seitlichen Navigationsleiste zwei Implementierungsvarianten: Die klassische Implementierung über eine FlyoutPage und die moderne Variante über die .NET MAUI Shell, mit der wir uns im nächsten Abschnitt beschäftigen werden.

Bei der klassischen Variante benötigen Sie drei oder mehr Seiten. Die erste Seite vom Typ .NET MAUI `FlyoutPage` dient als Container. Sie verfügt über zwei wichtige Eigenschaften:

- Die Eigenschaft `Flyout`, die die Seite angibt, die die Menüpunkte rendert.
- Die Eigenschaft `Detail`, über die die initiale Detailseite angegeben wird.

Den Aufbau einer FlyoutPage sehen Sie in Listing 10.4.

**Listing 10.4** Aufbau einer FlyoutPage

```
<?xml version="1.0" encoding="utf-8" ?>
<FlyoutPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:pages="clr-namespace:FlyoutSample"
       x:Class="FlyoutSample.MainPage"
      FlyoutLayoutBehavior="Popover">
  <FlyoutPage.Flyout>
    <pages:MainPageFlyout x:Name="FlyoutPage" />
  </FlyoutPage.Flyout>
```

```
  <FlyoutPage.Detail>
    <NavigationPage>
      <x:Arguments>
        <pages:HomePage />
      </x:Arguments>
    </NavigationPage>
  </FlyoutPage.Detail>
</FlyoutPage>
```

Die eigentliche Navigation wird im Code-Behind der FlyoutPage implementiert. Dazu registriert man eine Ereignisbehandlungsroutine für das Ereignis, das ausgelöst wird, wenn ein Menüpunkt ausgewählt wird. In dieser Ereignisbehandlungsroutine wird dann auf Basis der Auswahl ein entsprechendes .NET-MAUI-Seitenobjekt erzeugt und der Eigenschaft `Detail` der FlyoutPage zugewiesen. Listing 10.5 zeigt eine entsprechende Implementierung.

**Listing 10.5** Code-Behind der FlyoutPage

```
public partial class MainPage : FlyoutPage
{
  public MainPage()
  {
    InitializeComponent();
    FlyoutPage.MenuItemsCollectionView.SelectionChanged += OnSelectionChanged;
  }

  private void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
  {
    // prüfen, ob der ausgewählte Eintrag vom korrekten Typ ist
    if (e.CurrentSelection.FirstOrDefault() is MainPageFlyoutMenuItem item)
    {
      // Detailseite erzeugen und die Navigation durchführen
      Detail = new NavigationPage((Page)Activator.CreateInstance(item.TargetType));

      // Flyout wieder einklappen
      IsPresented = false;
    }
  }
}
```

Die Darstellung der Menüeinträge erfolgt im Beispiel durch die Seite `MainPageFlyout`, die der Eigenschaft `Flyout` der `FlyoutPage` zugewiesen wurde.

Listing 10.6 zeigt eine beispielhafte Implementierung. Wie Sie dem Code entnehmen können, definiert die Seite eine Eigenschaft `MenuItems` vom Typ `ObservableCollection<MainPageFlyoutMenuItem>`. Dieser Auflistung werden im Konstruktor die verschiedenen Menüeinträge hinzugefügt. Jeder Menüeintrag hat einen Titel, ein Bild und eine Eigenschaft `TargetType`. Die passenden Eigenschaften `Titel`, `Image` und `TargetType` sind in der Klasse `MainPageFlyoutItem` definiert. Anhand der Eigenschaft TargetType definieren Sie den Typ der Seite, die beim Klick auf einen Menüeintrag dargestellt werden soll. Die Eigenschaft wird in der Ereignisbehandlungsroutine der FlyoutPage ausgewertet (siehe Listing 10.5).

**Listing 10.6** Die Definition der Menüeinträge geschieht im Code-Behind der Menüseite

```
using System.Collections.ObjectModel;

namespace FlyoutSample;
public partial class MainPageFlyout : ContentPage
{
  public ObservableCollection<MainPageFlyoutMenuItem> MenuItems { get; }
  public MainPageFlyout()
  {
    InitializeComponent();
    MenuItems = new ObservableCollection<MainPageFlyoutMenuItem>(new[]
    {
      new MainPageFlyoutMenuItem
      {
        Title = "Startseite",
        TargetType = typeof(HomePage),
        Image = "home.png"
      },
      new MainPageFlyoutMenuItem
      {
        Title = "Einstellungen",
        TargetType = typeof(SettingsPage),
        Image = "settings.png"
      },
    });
    BindingContext = this;
  }
}
```

Abgesehen vom Code-Behind benötigen wir selbstverständlich auch entsprechendes Markup zum Rendern der Menüeinträge. Listing 10.7 zeigt eine beispielhafte Implementierung, die Sie nach Belieben anpassen können.

**Listing 10.7** Die Darstellung der Menüeinträge geschieht mithilfe eines CollectionView-Elements.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       x:Class="FlyoutSample.MainPageFlyout"
       xmlns:local="clr-namespace:FlyoutSample"
       Title="Flyout"
       IconImageSource="menu.png">
  <VerticalStackLayout>
    <CollectionView x:Name="MenuItemsCollectionView"
            x:FieldModifier="public"
            SelectionMode="Single"
        ItemsSource="{Binding MenuItems}">
      <CollectionView.Header>
        <Grid BackgroundColor="#2097C8"
            ColumnDefinitions="10,*,10"
            RowDefinitions="30,80,Auto,10">
          <Label
            Grid.Column="1"
            Grid.Row="2"
            Text="Flyout Sample"
            TextColor="White" FontSize="32"/>
        </Grid>
```

```
      </CollectionView.Header>
      <CollectionView.ItemTemplate>
        <DataTemplate>
          <Grid Padding="15,10"
              ColumnDefinitions="30,*"
              x:DataType="local:MainPageFlyoutMenuItem">
            <Image Source="{Binding Image}"/>
            <Label VerticalOptions="FillAndExpand"
                   Grid.Column="1"
                   Text="{Binding Title}"
                   FontSize="24"/>
          </Grid>
        </DataTemplate>
      </CollectionView.ItemTemplate>
    </CollectionView>
  </VerticalStackLayout>
</ContentPage>
```

Da Listen erst in einem späteren Kapitel behandelt werden, möchte ich an dieser Stelle noch nicht zu weit vorgreifen und nur auf einige wichtige Kernpunkte hinweisen.

Dem `CollectionView`-Element wird im Listing der Name `MenuItemsCollectionView` gegeben. Durch die Zuweisung des Werts `public` an das Attribut `x:FieldModifier` wird eine öffentliche Eigenschaft in der Klasse `MainPageFlyoutItem` definiert. Über diese Eigenschaft wird in der FlyoutPage schließlich die Ereignisbehandlungsroutine des Ereignisses `ItemSelected` registriert (siehe Konstruktor in Listing 10.5).

Die Daten für das CollectionView-Element werden über eine Datenbindung der Eigenschaft `ItemsSource` bereitgestellt.

Das Aussehen der einzelnen Listeneinträge können Sie über die Eigenschaft `ItemTemplate` im Markup der Seite definieren. Die XAML-Struktur, die Sie innerhalb der Kindeigenschaft `DataTemplate` des `ItemTemplate` definieren, wird für jede Zeile der `CollectionView` gerendert.

## ■ 10.2 Die Shell

Im Abschnitt 10.1 dieses Kapitels haben Sie gängige Navigationsmuster in mobilen Apps kennengelernt und erfahren, wie diese mit .NET MAUI implementiert werden können. In den Abschnitten zu Registerkarten und der seitlichen Navigationsleiste haben Sie außerdem erfahren, dass es zusätzlich zu den vorgestellten Implementierungen auch eine neue, modernere Implementierung gibt: die .NET MAUI Shell. Die Shell wurde im .NET-MAUI-Vorgänger Xamarin.Forms Version 4.0 eingeführt und verfolgt das Ziel, die Entwicklung von Apps zu verkürzen, indem bereits eine grundlegende Infrastruktur für gängige Aufgabenstellungen bereitgestellt wird.

Zu den Hauptfunktionen der Shell gehört unter anderem ein einheitlicher Navigations-Container, mit dem Sie sowohl eine Navigation über Registerkarten, eine hierarchische Navigation, als auch eine Navigation über eine seitliche Navigationsleiste implementieren können. Eine komplizierte Kombination aus TabbedPage, FlyoutPage und NavigationPage gehört somit der Vergangenheit an.

Ein weiteres Navigationsfeature der Shell ist die routenbasierte Navigation. Diese funktioniert ähnlich wie die Navigation in webbasierten Anwendungen, indem man Pfade zu bestimmten Seiten definiert, wie zum Beispiel `Routing.RegisterRoute("settings/wifi", typeof(WifiSettingsPage));`. Über den angegebenen Pfad – im Beispiel ist dies „settings/wifi“ – können Sie innerhalb der App von jeder beliebigen Stelle zur hinterlegten Seite – im Beispiel „WifiSettingsPage“ – navigieren, ohne sich an die vorgegebene Struktur zu halten.

Die letzte Kernfunktion der Shell ist die Anzeige eines integrierten Suchfeldes in der Navigationsleiste. In diesem Kapitel werden wir uns auf die Navigation mit der Shell konzentrieren und die Suche nicht behandeln.

### 10.2.1 Überblick über die Shell

Mit der .NET-MAUI-Shell können Sie eine seitliche Navigationsleiste, eine registerkartenbasierte Navigation, oder eine Kombination aus beidem erstellen.

Das Wurzelelement, welches Sie in Ihrer Shell anlegen, ist unabhängig von der gewählten Navigationsstrategie immer das Element `Shell`.

Als Kindelemente der Shell können Sie Elemente vom Typ `FlyoutItem` oder `TabBar` anlegen. `FlyoutItem`-Elemente werden in der seitlichen Navigationsleiste angezeigt, `TabBar`-Elemente als Registerkarten. Als Kindelemente eines `FlyoutItem` oder einer `TabBar` haben Sie immer Elemente vom Typ `Tab`, die wiederum Kindelemente vom Typ `ShellContent` haben. Ein Tab definiert die Beschriftung und das Icon eines Menüeintrags bzw. einer Registerkarte, wohingegen `ShellContent` die darzustellende Seite für den Menüeintrag/die Registerkarte angibt.

Abgesehen von `FlyoutItem` und `TabBar` gibt es mit dem `MenuItem` noch ein weiteres Kindelement. Der Unterschied zwischen `MenuItem` und den beiden Alternativen ist, dass der Klick auf ein `MenuItem` keine automatische Navigation zu einer Seite, sondern ein Click-Ereignis auslöst, dass Sie in Ihrem Quellcode behandeln können.

Die Struktur der Shell lautet demnach:

**Shell**

- FlyoutItem
  - Tab
    - ShellContent
- TabBar
  - Tab
    - ShellContent
- MenuItem

Diese doch recht aufwendige Hierarchie kann dank eingebauter Konvertierungsoperatoren wie folgt vereinfacht werden:

**Shell**

- FlyoutItem
  - ShellContent

- TabBar
  - ShellContent
- MenuItem

Oder sogar:

**Shell**

- ShellContent
- MenuItem

Es können also sowohl die Ebene des Tab als auch die Ebene des FlyoutItem/der TabBar weggelassen werden.

Zusätzlich zu den Menüeinträgen können Sie über die Shell außerdem mit den Eigenschaften `FlyoutHeader` und `FlyoutFooter` bzw. `FlyoutHeaderTemplate` und `FlyoutFooterTemplate` noch die Kopf- und Fußzeile der seitlichen Navigationsleiste konfigurieren.

### 10.2.2 Eine Navigationsstruktur mit der Shell definieren

Nachdem Sie die Grundlagen der Shell kennengelernt haben, tauchen wir nun in die Praxis ein. Der erste Schritt bei der Entwicklung einer Shell-App besteht im Anlegen des Shell-Grundgerüsts. Die Hauptarbeit übernimmt die Visual-Studio-Projektvorlage beim Erzeugen eines neuen Projekts für uns bereits. Die Vorlage legt die Datei an *AppShell.xaml* in unserem Projekt an, die eine grundlegende Shell enthält.

Sie besteht aus einer Code-Behind-Datei mit folgender Klassenstruktur `public partial class AppShell : Shell` und einer XAML-Datei, deren Markup Sie in Listing 10.8 sehen.

**Listing 10.8** Das Grundgerüst einer Shell

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  x:Class="ShellSample.AppShell"
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:ShellSample"
  Shell.FlyoutBehavior="Disabled">

  <ShellContent
    Title="Home"
    ContentTemplate="{DataTemplate local:MainPage}"
    Route="MainPage" />

</Shell>
```

Das Markup aus Listing 10.8 ist unser Grundgerüst für jede Shell. Für die nun folgenden Szenarien gehen wir stets davon aus, dass wir diesen Stand haben und nur Kindelemente zur Shell hinzufügen. Außerdem gehen wir davon aus, dass es in unserer App zwei ContentPages mit den Namen `MainPage` und `SettingsPage` sowie die beiden Bilddateien *home.svg*, *info.svg* und *settings.svg* gibt. Im aktuellen Zustand kann über die Shell noch nicht über eine seitliche Navigationsleiste navigiert werden, da die Eigenschaft `Shell.FlyoutBehavior` den Wert `Disabled` hat. Dies werden wir im weiteren Verlauf des Kapitels ändern.

### 10.2.2.1 Eine Registerkartennavigation mit der Shell anlegen

Als erstes Beispiel setzen wir die Registerkartennavigation aus Bild 10.6 um.

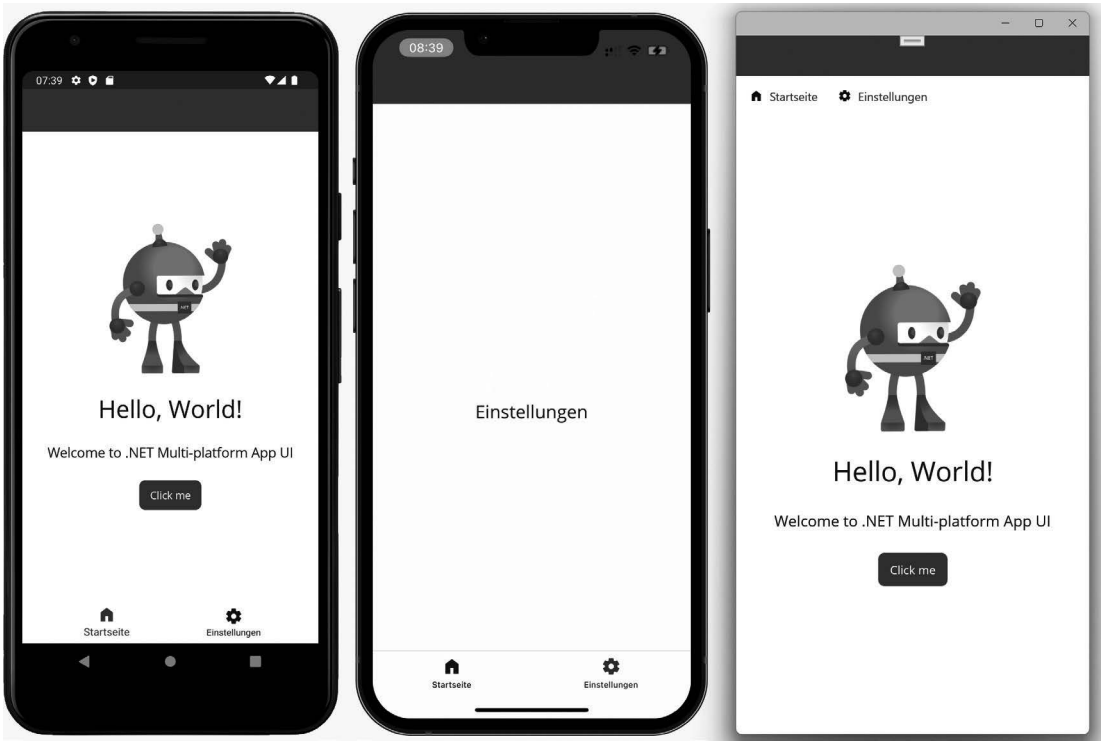


**Bild 10.6** Eine einfache Registerkartennavigation mit der Shell

Basis für die Registerkarten aus Bild 10.6 ist ein `TabBar`-Element, das als Container für Registerkarten dient. Diesem Element fügen wir zwei `Tab`-Elemente inklusive `ShellContent`-Elementen hinzu. Das Ergebnis sehen Sie in Listing 10.9.

**Listing 10.9** Eine Registerkartennavigation mit der Shell

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  x:Class="ShellSample.AppShell"
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:ShellSample"
  Shell.FlyoutBehavior="Disabled">

  <TabBar>
    <Tab Title="Startseite" Icon="home.png">
      <ShellContent ContentTemplate="{DataTemplate local:MainPage}"
        Route="MainPage" />
    </Tab>
    <Tab Title="Einstellungen" Icon="settings.png">
```

```
      <ShellContent ContentTemplate="{DataTemplate local:SettingsPage}"
        Route="SettingsPage" />
    </Tab>
  </TabBar>
</Shell>
```

Über die Eigenschaften `Title` und `Icon` des `Tab`-Elements definieren wir die Beschriftung sowie das Icon der Registerkarte.

Die Seite, die beim Klick auf die Registerkarte angezeigt wird, konfigurieren wir über die Eigenschaft `ContentTemplate` des `ShellContent`-Elements. Der Eigenschaft wird via Datenbindung ein `DataTemplate` zugewiesen, das auf den Typ der Zielseite zeigt. Der Vorteil dieser Methode ist, dass das Seitenobjekt nicht beim Applikationsstart, sondern erst beim Klick auf den Menüeintrag erzeugt werden muss, wodurch sich die Performance der App verbessert.

Im Beispiel wird noch die optionale Eigenschaft `Route` des `ShellContent`-Elements genutzt. Mit ihr legen wir die Navigationsroute fest, über die wir per Quellcode zur Zielseite navigieren können.

Aus dem Code aus Listing 10.9 können wir die `Tab`-Elemente dank der zuvor beschriebenen Konvertierungsoperatoren entfernen und ihn verkürzt auch wie folgt schreiben:

```
<TabBar>
  <ShellContent Title="Startseite" Icon="home.png"
          ContentTemplate="{DataTemplate local:MainPage}"
          Route="MainPage" />

  <ShellContent Title="Einstellungen" Icon="settings.png"
          ContentTemplate="{DataTemplate local:SettingsPage}"
          Route="SettingsPage" />
</TabBar>
```

Da die `Tab`-Elemente in Listing 10.9 für die Beschriftung und das Icon zuständig waren, müssen die Eigenschaften `Title` und `Icon` in der verkürzten Markup-Definition bei den `ShellContent`-Elementen angegeben werden. Andernfalls würden die Registerkarten weder beschriftet sein noch über ein Icon verfügen.

#### 10.2.2.2 Eine seitliche Navigationsleiste mit der Shell

Der Quellcode für eine seitliche Navigationsleiste, wie wir sie in Bild 10.7 sehen, ähnelt sehr dem Quellcode aus Listing 10.9.

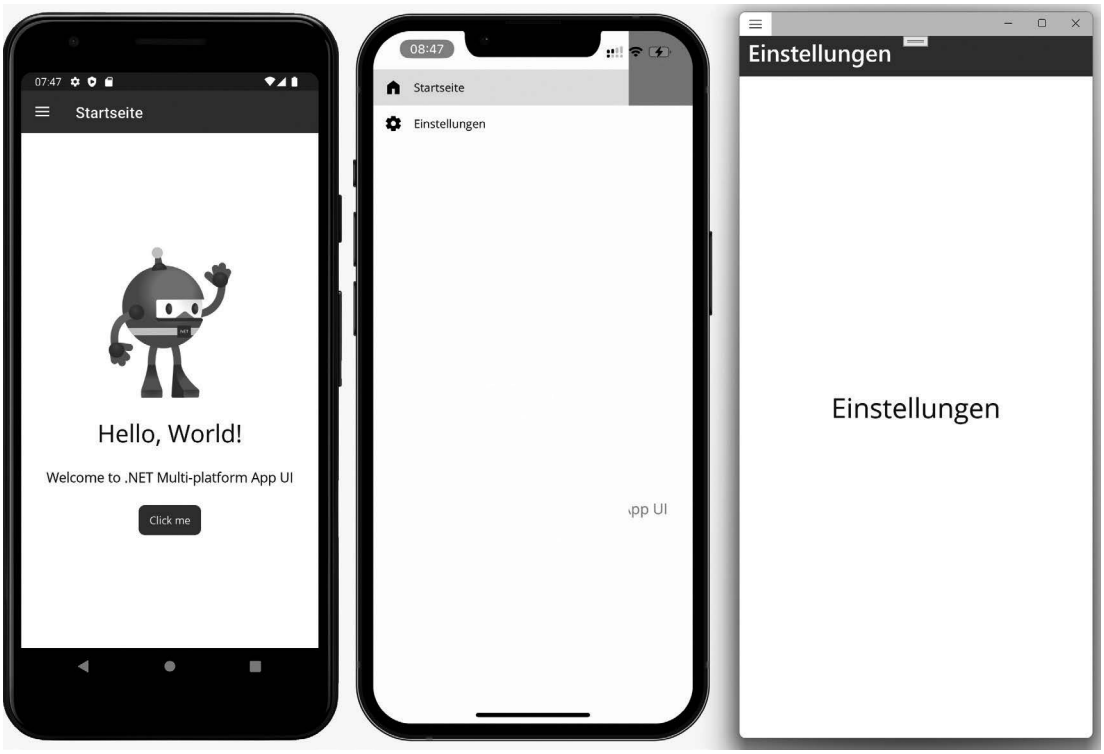


**Bild 10.7** Eine seitliche Navigation mit Burger-Menü unter Android (links, geschlossen), iOS (Mitte, geöffnet) und Windows (rechts, geschlossen nach Navigation) mithilfe der Shell

Anstelle eines `TabBar`-Elements, wie bei der Navigation über Registerkarten, benötigen wir bei der seitlichen Navigationsleiste ein `FlyoutItem`-Element je `Tab`-/`ShellContent`-Element (Listing 10.10).

**Listing 10.10** Eine seitliche Navigation mit der Shell

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  x:Class="ShellSample.AppShell"
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:ShellSample"
  Shell.FlyoutBehavior="Flyout">
  <FlyoutItem Title="Startseite" Icon="home.png">
    <Tab>
      <ShellContent
        Title="Startseite"
        ContentTemplate="{DataTemplate local:MainPage}"
        Route="MainPage" />
    </Tab>
  </FlyoutItem>
  <FlyoutItem Title="Einstellungen" Icon="settings.png">
    <Tab>
      <ShellContent
```

```
        Title="Einstellungen"
        ContentTemplate="{DataTemplate local:SettingsPage}"
        Route="SettingsPage" />
    </Tab>
  </FlyoutItem>
</Shell>
```

Die beiden größten Unterschiede zur Registerkartennavigation bestehen darin, dass wir je `Tab`/`ShellContent` ein eigenes umschließendes `FlyoutItem`-Element benötigen und dass die Beschriftung und das Icon über Eigenschaften des `FlyoutItem`-Elements und nicht des `Tab`-Elements festgelegt werden. Außerdem muss eine weitere Beschriftung über die `Title`-Eigenschaft des ShellContent-Elements erfolgen, damit eine Überschrift im oberen Bildschirmbereich sichtbar ist. Voraussetzung für die Anzeige des Menüs ist außerdem, dass die Eigenschaft `FlyoutBehavior` der Shell den Wert `Flyout` hat. Da dies der Standardwert ist, kann die Angabe der Eigenschaft auch entfallen. Wichtig ist an dieser Stelle jedoch, dass der Wert `Disabled`, der durch die Projektvorlage eingetragen wurde, aus der Eigenschaft `FlyoutBehavior` ersetzt oder die komplette Eigenschaftsdefinition entfernt wird.

Auf den ersten Blick ist das Markup für eine seitliche Navigationsleiste etwas länger als für eine Navigation über Registerkarten. In der Praxis wird allerdings meistens die verkürzte Schreibweise genutzt, die vollkommen ohne umklammernde `FlyoutItem`- und `Tab`-Elemente auskommt:

```
<TabBar>
    <ShellContent Title="Startseite" Icon="home.png" Route="MainPage"
      ContentTemplate="{DataTemplate local:MainPage}" />
    <ShellContent Title="Einstellungen" Icon="settings.png" Route="SettingsPage"
       ContentTemplate="{DataTemplate local:SettingsPage}" />
</TabBar>
```

### 10.2.2.3 Die seitliche Navigation mit Registerkarten kombinieren

Bei sehr umfangreichen Menüs kann es sinnvoll sein, die seitliche Navigationsleiste um einige Einträge zu kürzen und diese stattdessen als zusätzliche Registerkarten anzuzeigen. Eine solche Verfahrensweise bietet sich immer dann an, wenn die Menüeinträge, die in Registerkarten zusammengefasst werden, thematisch ähnlich sind.

Bild 10.8 zeigt eine solche Kombination. Im Menü, das Sie ausgeklappt auf dem rechten Mobiltelefon des Bildschirmfotos sehen, gibt es einen übergeordneten Eintrag mit der Beschriftung „Gruppe 1“. Wählt der Benutzer diesen Eintrag aus, kommt er auf die Seite, die Sie auf dem linken Mobiltelefon des Bildschirmfotos sehen. Diese Seite verfügt über die angesprochene Registerkartennavigation.

Die beiden Menüpunkte *Dummy Eintrag 1* und *Dummy Eintrag 2* der Abbildung wurden übrigens nur aufgenommen, um anzudeuten, dass das Menü noch weitere Einträge haben könnte.

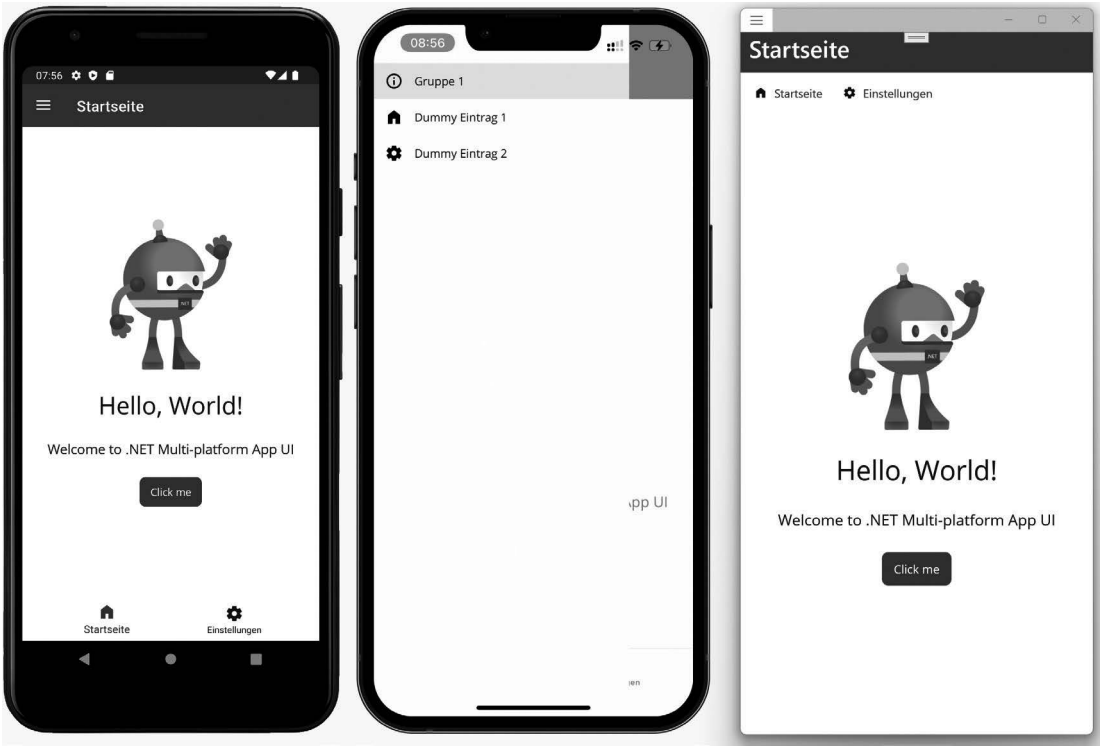


**Bild 10.8** Eine seitliche Navigationsleiste, kombiniert mit Registerkarten

In Listing 10.11 sehen Sie den notwendigen Quellcode für die Bildschirmmaske aus Bild 10.8. Das Listing zeigt, dass Sie die beiden ersten `ShellContent`-Elemente, die als Registerkarten gerendert werden, lediglich mit einem `FlyoutItem`-Element umklammern müssen.

**Listing 10.11** Eine seitliche Navigationsleiste, kombiniert mit Registerkarten

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  x:Class="ShellSample.AppShell"
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:ShellSample"
  Shell.FlyoutBehavior="Flyout">
  <FlyoutItem Title="Gruppe 1" Icon="info.png">

    <ShellContent Icon="home.png" Title="Startseite"
      ContentTemplate="{DataTemplate local:MainPage}"
      Route="MainPage" />

    <ShellContent Icon="settings.png" Title="Einstellungen"
      ContentTemplate="{DataTemplate local:SettingsPage}"
      Route="SettingsPage" />
  </FlyoutItem>

  <ShellContent Icon="home.png" Title="Dummy Eintrag 1"
```

```
    ContentTemplate="{DataTemplate local:MainPage}"
    Route="MainPage" />

  <ShellContent Icon="settings.png" Title="Dummy Eintrag 2"
    ContentTemplate="{DataTemplate local:SettingsPage}"
    Route="SettingsPage" />
</Shell>
```

Nach einer kleinen Modifikation von Listing 10.11 können die Registerkarten im Übrigen auch am oberen Bildschirmrand statt am unteren Bildschirmrand angezeigt werden. Bild 10.9 zeigt dies. Unter Windows führt diese Änderung allerdings dazu, dass die Registerkarten erst aufgeklappt werden müssen.

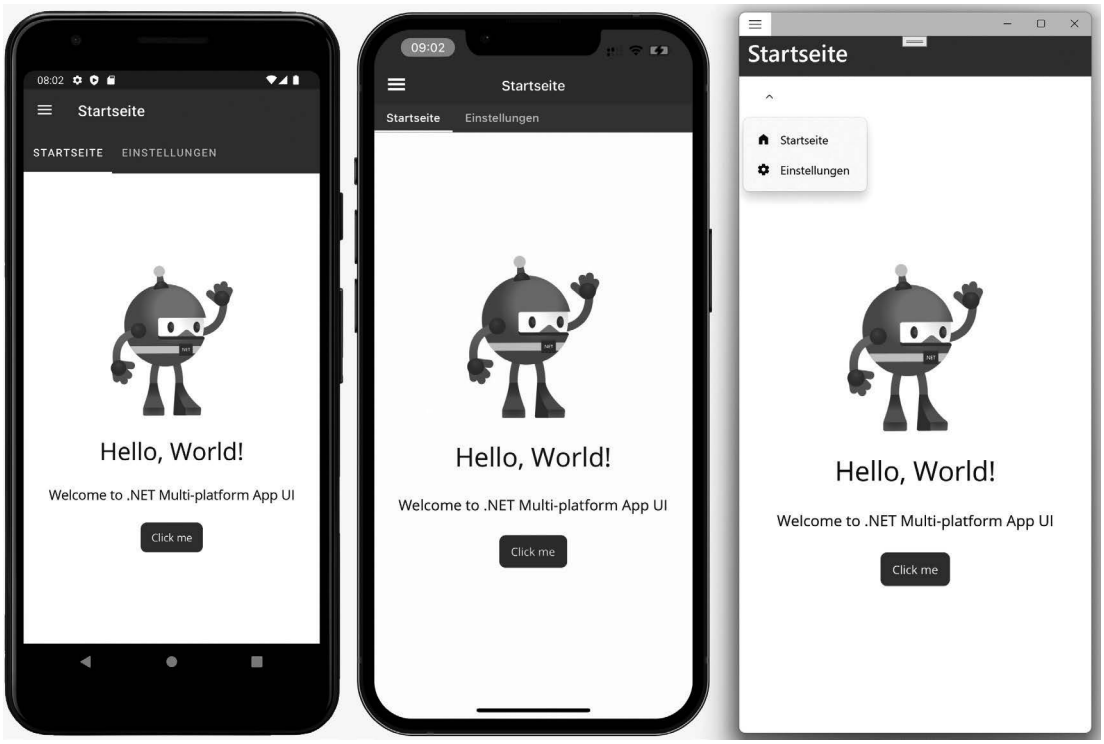


**Bild 10.9** Mithilfe der Shell können Registerkarten auch am oberen Bildschirmrand angezeigt werden.

Die angesprochene Änderung besteht darin, innerhalb des `FlyoutItem`-Elements noch ein `Tab`-Element einzusetzen:

```
<FlyoutItem Title="Gruppe 1" Icon="info.png">
  <Tab>
    <ShellContent Title="Startseite" Icon="home.png" Route="MainPage"
      ContentTemplate="{DataTemplate local:HomePage}" />
    <ShellContent Title="Einstellungen" Icon="settings.png" Route="SettingsPage"
      ContentTemplate="{DataTemplate local:SettingsPage}" />
  </Tab>
</FlyoutItem>
```

### 10.2.2.4 Menüeinträge als Aktionselemente nutzen

Bisher haben wir uns angesehen, wie wir die Shell nutzen können, um die Navigationshierarchie der Anwendung aufzubauen. Jeder Menüeintrag löste in den bisherigen Beispielen die Navigation zu einer Seite der App aus.

In einigen Fällen ist es nicht gewünscht, dass ein Menüeintrag eine Navigation auslöst. Stattdessen soll er eine Aktion auslösen. Zu diesem Zweck können Sie das Kindelement `MenuItem` der .NET-MAUI-Shell nutzen.

Ein `MenuItem` ist ein direktes Kindelement der Shell, das weder in einem `Tab`, noch in einem `FlyoutItem` oder einer `TabBar` geschachtelt wird. Über die Eigenschaft `Text` legen Sie die Beschriftung des Menüeintrags fest, über die Eigenschaft `IconImageSource` das Icon. Die auszuführende Aktion können Sie über zwei Wege angeben. Entweder registrieren Sie eine Ereignisbehandlungsroutine für das Ereignis `Clicked`, oder Sie binden ein Command an die Eigenschaft `Command`. Ein Beispiel für einen Aktionseintrag sehen Sie in Listing 10.12.

**Listing 10.12** Ein Aktionseintrag in der seitlichen Navigationsleiste

```
<ShellContent Icon="home.png" Title="Startseite"
    ContentTemplate="{DataTemplate local:MainPage}"
    Route="MainPage" />
<ShellContent Icon="settings.png" Title="Einstellungen"
    ContentTemplate="{DataTemplate local:SettingsPage}"
    Route="SettingsPage" />
<MenuItem IconImageSource="info.png" Text="Info"
    Clicked="MenuItem_Clicked"/>
```

Der Menüeintrag aus dem obigen Listing führt seine Aktion über die Ereignisbehandlungsroutine `MenuItem_OnClicked` aus. Diese befindet sich im Code-Behind, in der Datei *AppShell.xaml.cs*.

Wie Sie Listing 10.13 entnehmen können, handelt es sich bei der Methode um eine simple Ereignisbehandlungsroutine mit den Standardparametern vom Typ `object` bzw. `EventArgs`. Im vorliegenden Beispiel wird ein Pop-Up-Dialog angezeigt und das Menü anschließend wieder geschlossen. Letzteres geschieht durch die Zuweisung des Werts `false` an die Eigenschaft `FlyoutIsPresented`.

**Listing 10.13** Eine Ereignisbehandlungsroutine für einen Aktionseintrag der Shell

```
private async void MenuItem_Clicked(object sender, EventArgs e)
{
  await DisplayAlert("Versionsinfo", "1.0.0", "Ok");
  FlyoutIsPresented = false; // Flyout wieder schließen
}
```

Das Ergebnis des Listings zeigt Bild 10.10. Der Klick auf den Eintrag „Info“ des Menüs öffnet ein Pop-Up und navigiert nicht zu einer anderen Seite, wie es bei `ShellContent`-Elementen der Fall ist.

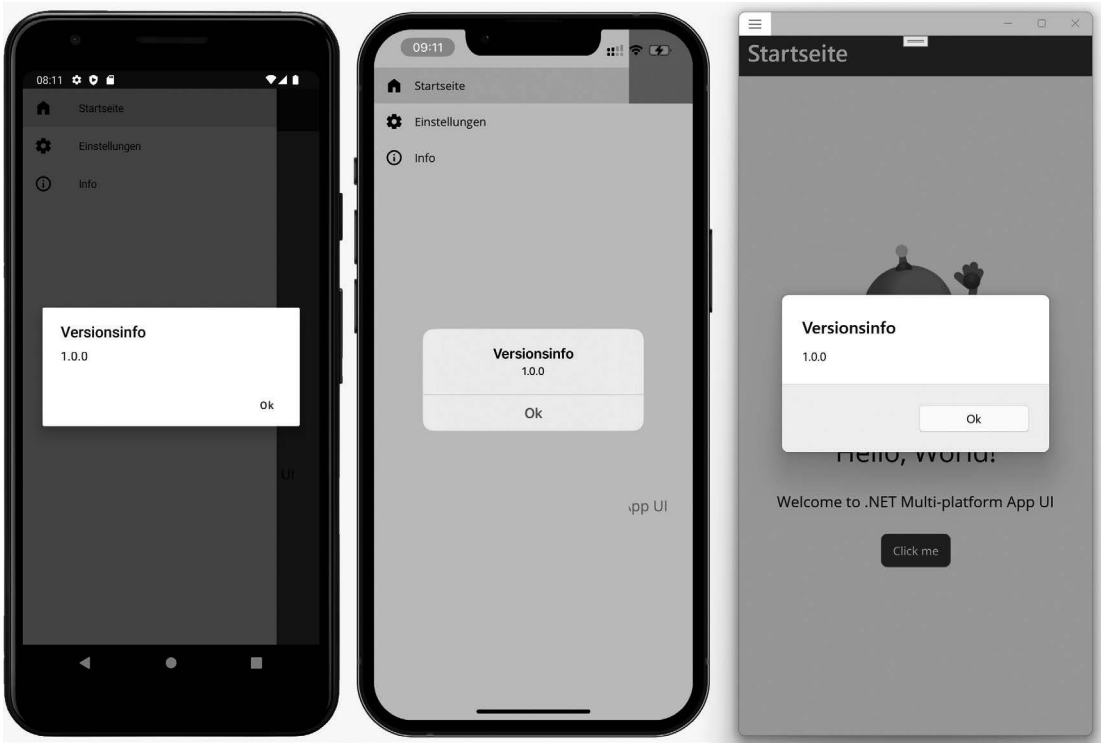


**Bild 10.10** Ein Aktionseintrag innerhalb einer Shell

### 10.2.3 Kopf- und Fußzeile der Shell definieren

Nachdem Sie in Abschnitt 10.2.2 gelernt haben, verschiedene Navigationsstrukturen mit der Shell aufzubauen, möchte ich Ihnen nun erklären, wie Sie die seitliche Navigationsleiste der Shell optisch durch eine Kopf- und Fußzeile aufwerten können, so wie in Bild 10.11 dargestellt.


**Bild 10.11** Eine Shell mit Kopf- und Fußzeile

Beide Elemente, sowohl die Kopf- als auch die Fußzeile, lassen sich sehr einfach anlegen. Die Shell stellt dazu die Eigenschaften `FlyoutHeaderTemplate` und `FlyoutFooterTemplate` bereit. Dem Kindelement `DataTemplate` beider Eigenschaften können Sie einen Layout-Container mit beliebigem XAML übergeben. Listing 10.14 zeigt ein einfaches Beispiel mit jeweils zwei `Label`-Elementen in Kopf- und Fußzeile.

**Listing 10.14** Kopf- und Fußzeile in einer Shell

```
<Shell.FlyoutHeaderTemplate>
<DataTemplate>
  <StackLayout BackgroundColor="#091F35" Padding="20">
    <Label Text="Shell Demo" FontSize="Large" TextColor="White"/>
    <Label Text=".NET MAUI Shell" TextColor="White"/>
  </StackLayout>
</DataTemplate>
</Shell.FlyoutHeaderTemplate>
<ShellContent Icon="home.png" Title="Startseite"
  ContentTemplate="{DataTemplate local:MainPage}"
  Route="MainPage" />
<ShellContent Icon="settings.png" Title="Einstellungen"
  ContentTemplate="{DataTemplate local:SettingsPage}"
  Route="SettingsPage" />
<MenuItem IconImageSource="info.png" Text="Info"
  Clicked="MenuItem_Clicked"/>
```

```
<Shell.FlyoutFooterTemplate>
  <DataTemplate>
    <StackLayout BackgroundColor="#2097C8" Padding="20">
      <Label Text="Footer der" FontSize="Large" TextColor="White"/>
      <Label Text=".NET MAUI Shell" TextColor="White"/>
    </StackLayout>
  </DataTemplate>
</Shell.FlyoutFooterTemplate>
```

### 10.2.4 Routenbasierte Navigation

Zu Beginn von Abschnitt 10.2 haben Sie erfahren, dass die Shell eine routenbasierte Navigation mitbringt, ähnlich wie man sie von Webanwendungen kennt.

Voraussetzung für die routenbasierte Navigation ist die vorherige Definition von Routen. Eine Route besteht aus drei Elementen:

- Dem Schlüssel bzw. Pfad. Hierbei handelt es sich um einen eindeutigen String.
- Der Seite, zu der navigiert werden soll.
- Optional ein oder mehrere Routenparameter.

Routen können wahlweise deklarativ im Markup der Shell für Seiten angegeben werden, die zur Shell-Navigationsstruktur gehören, oder imperativ im Code-Behind für Seiten, die nicht zur Shell-Navigationsstruktur gehören.

Im Fall der deklarativen Definition erzeugt das folgende Element `<ShellContent Title="Einstellungen" Icon="settings.png" Route="SettingsPage" ContentTemplate="{DataTemplate local:SettingsPage}" />` eine Route mit dem Pfad `SettingsPage` und der Zielseite `SettingsPage`.

Bei der imperativen Variante wird die Methode `RegisterRoute` der statischen Klasse `Routing` aufgerufen und als erstes Argument der Pfad und als zweites Argument der Typ der Zielseite übergeben. Dieser Aufruf findet in der Regel im Konstruktor der Shell statt. Um Tippfehler zu vermeiden, empfiehlt es sich, den Pfad gleich dem Typ der Zielseite zu halten und beides über die Operatoren `nameof` bzw. `typeof` anzugeben: `Routing.RegisterRoute(nameof(SettingsPage), typeof(SettingsPage));`. Dass der Schlüssel bzw. Pfad gleich dem Seitennamen sein muss, ist keine Pflicht. Sie können problemlos einen abweichenden Schlüssel wählen und diesen sogar über Schrägstriche hierarchisch gestalten:

```
Routing.RegisterRoute("settings/wifi", typeof(SettingsPage));
```

Nachdem eine Route registriert wurde, können Sie aus einer beliebigen Stelle der Anwendung über den Aufruf der Methode `GoToAsync` der Shell zu dieser Route navigieren:

```
Shell.Current.GoToAsync($"{nameof(SettingsPage)}");
```

Die Methode `GoToAsync` erwartet als Argument den Pfad der Route sowie optional ein oder mehrere Routenparameter. Für die sichere Angabe des Pfads können Sie wieder den `nameof`-Operator nutzen.

### Parametrisierte Routen

Während der Navigation ist es häufig erforderlich, Daten von der aufrufenden Seite an die Zielseite zu übergeben. Die .NET-MAUI-Shell löst diese Aufgabe über Routenparameter. Diese können Sie beim Aufruf der Methode `GoToAsync` mit einem vorangestellten `?` für den ersten Parameter bzw. `&` für alle weiteren Parameter ans Ende der Route anhängen. Das folgende Beispiel verdeutlicht dies:

```
Shell.Current.GoToAsync("settings/wifi?Param1=Wert1&Param2=Wert2&Param3=Wert3");
```

Im Code-Behind der Zielseite müssen Sie für jeden Parameter eine entsprechende Eigenschaft definieren und über das Attribut `QueryProperty` auf Klassenebene auszeichnen. Das Attribut erwartet zwei Argumente: den Namen der Eigenschaft und den Namen des Routenparameters. Im Idealfall halten Sie die Bezeichnung der Eigenschaft und des Routenparameters gleich und nutzen dazu auch hier wieder den `nameof`-Operator:

```
[QueryProperty(nameof(Param1), nameof(Param1))]
[QueryProperty(nameof(Param2), nameof(Param2))]
[QueryProperty(nameof(Param3), nameof(Param3))]
public partial class WifiSettingsPage : ContentPage
{
  public string Param1 { get; set; }
  public string Param2 { get; set; }
  public string Param3 { get; set; }
  public WifiSettingsPage()
  {
    InitializeComponent();
  }
}
```

Als Alternative zur Argumentenübergabe in Form eines Strings können Sie auch ein Dictionary vom Typ `<string, object>` an die Methode `GoToAsync` der Shell übergeben. Folgende Zeilen verdeutlichen dies.

```
// statt dieser Schreibweise
Shell.Current.GoToAsync("settings/wifi?Param1=Wert1&Param2=Wert2&Param3=Wert3");

// geht auch diese
var navigationArguments = new Dictionary<string, object>
{
  {"Param1", "Wert1"},
  {"Param2", "Wert2"},
  {"Param3", "Wert3"},
}
Shell.Current.GoToAsync("settings/wifi", navigationArguments);
```

Da der zweite Parameter des Dictionarys ein Objekt und kein einfacher String ist, können Sie über diesen Weg auch komplexe Objekte oder andere Datentypen wie zum Beispiel Zahlen übergeben.

```
var person = new Person { Firstname = "Wilhelm", Lastname = "Brause"};
var navigationArguments = new Dictionary<string, object>
{
  {"Person", person},
  {"Alter", 35}
}
Shell.Current.GoToAsync("person/details", navigationArguments);
```

Auf der Empfängerseite definieren Sie dann einfach die passenden Eigenschaften.

```
[QueryProperty(nameof(Person), nameof(Person))]
[QueryProperty(nameof(Age), nameof(Alter))]
public partial class PersonDetailsPage : ContentPage
{
  public Person Person { get; set; }
  public int Age { get; set; }
  public PersonDetailsPage ()
  {
    InitializeComponent();
  }
}
```

Falls Sie die Deserialisierung der Navigationsparameter lieber selbst übernehmen anstatt die automatische Zuweisung der Werte über das Attribut `QueryProperty` zu nutzen, dann müssen Sie in Ihrer Zielseite die Schnittstelle `IQueryAttributable` implementieren. Das Interface definiert die Methode `ApplyQueryAttributes`, die als Argument die Navigationsparameter eines Dictionarys vom Typ `<string, object>` übergeben bekommt.

```
public partial class PersonDetailsPage : ContentPage, IQueryAttributable
{
  public Person Person { get; set; }
  public int Age { get; set; }
  public PersonDetailsPage ()
  {
    InitializeComponent();
  }

  public void ApplyQueryAttributes(IDictionary<string, object> query)
  {
    Person = query["Person"] as Person;
    if(int.TryParse(query["Age"].ToString(), out int age))
    {
      Age = age;
    }
  }
}
```

### Praxisbeispiel zur routenbasierten Navigation

Nachdem wir uns mit der Theorie zur routenbasierten Navigation beschäftigt haben, ist es jetzt Zeit für ein kleines Praxisbeispiel.

In einer Shell-Anwendung soll der Anwender auf einer Seite seinen Vornamen, seinen Nachnamen und sein Alter eingeben und danach einen Button anklicken. Anschließend wird er auf eine neue Seite weitergeleitet, die seinen zuvor eingegebenen Namen ausgibt.

Bild 10.12 zeigt diesen Ablauf.

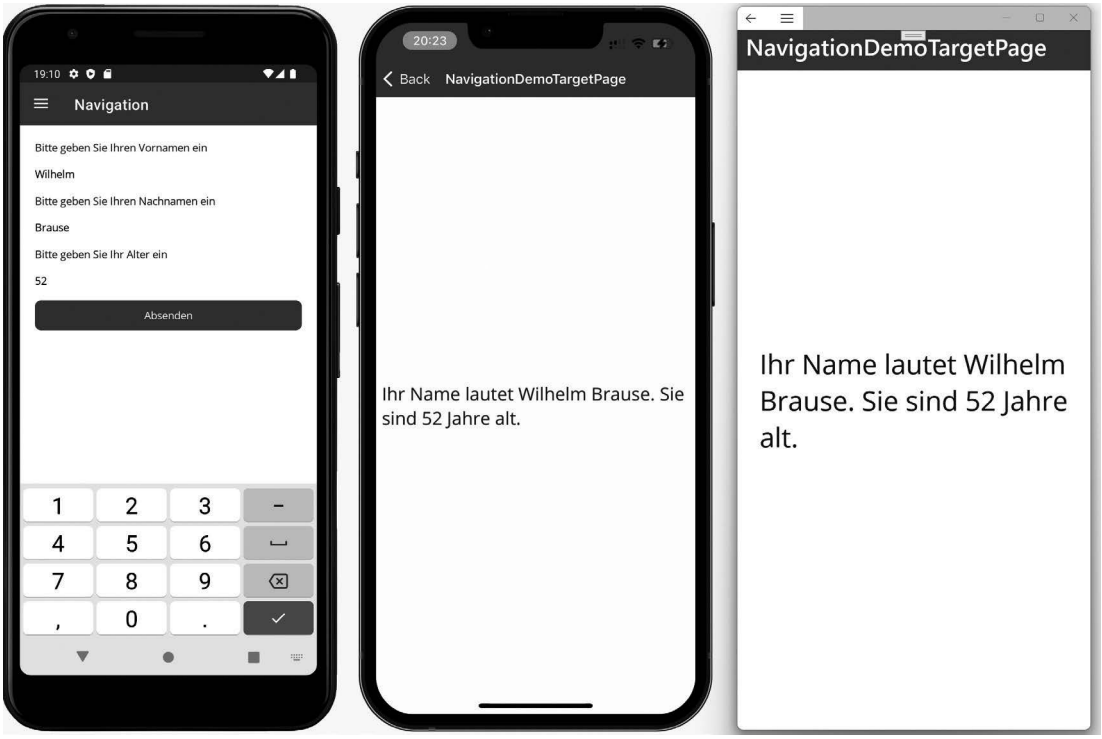


**Bild 10.12** Abbildung einer kleinen Beispielanwendung zur routenbasierten Navigation.

Der erste Schritt besteht aus der Anlage der Zielseite. Das Markup der Seite besteht aus einem Label mit dem Namen `MessageLabel`, in das später der Name und das Alter des Anwenders geschrieben wird:

```
<VerticalStackLayout VerticalOptions="Center">
  <Label x:Name="MessageLabel" FontSize="Large"
    HorizontalOptions="Center" />
</VerticalStackLayout>
```

Etwas spannender als das Markup ist die Code-Behind-Datei. Diese definiert die Eigenschaften `FirstName`, `LastName` und `Age`, die mithilfe des Attributs `QueryProperty` über die Shell beschrieben werden können. In der Methode `OnAppearing` wird der Inhalt der Eigenschaften dann gelesen und im Label `MessageLabel` ausgegeben:

```
namespace ShellSample;

[QueryProperty(nameof(FirstName), nameof(FirstName))]
[QueryProperty(nameof(LastName), nameof(LastName))]
[QueryProperty(nameof(Age), nameof(Age))]
public partial class NavigationDemoTargetPage : ContentPage
{

  public NavigationDemoTargetPage()
  {
    InitializeComponent();
```

```
  }

  public string FirstName { get; set; }

  public string LastName { get; set; }

  public int Age { get; set; }

  protected override void OnNavigatedTo(NavigatedToEventArgs args)
  {
    MessageLabel.Text =
      $"Ihr Name lautet {FirstName} {LastName}. Sie sind {Age} Jahre alt.";
    base.OnNavigatedTo(args);
  }
}
```

Passend zur Detailseite wird im Konstruktor der Klasse `AppShell` folgende Route definiert:

```
Routing.RegisterRoute(nameof(NavigationDemoTargetPage),
                      typeof(NavigationDemoTargetPage));
```

Im Code-Behind der Seite, die die Eingabefelder für Vornamen, Nachnamen und Alter anbietet, gibt es schließlich die folgende Ereignisbehandlungsroutine für den Klick auf den Button:

```
private async void Button_OnClicked(object sender, EventArgs e)
{
  var navigationParameters = new Dictionary<string, object>
  {
    {"FirstName", FirstNameEntry.Text },
    {"LastName", LastNameEntry.Text },
    {"Age", AgeEntry.Text },
  };
  await Shell.Current.GoToAsync($"{nameof(NavigationDemoTargetPage)}",
    navigationParameters);}
```

Mithilfe der vorgestellten Codezeilen ist die in Bild 10.12 gezeigte Aufgabenstellung nun erledigt. Wie Sie gesehen haben, kann die routenbasierte Navigation mit nur wenig Aufwand implementiert werden.

## ■ 10.3 Dependency Injection und die Shell

Wenn Sie die .NET-MAUI-Shell nutzen, dann erstellt die Shell beim Navigieren selbstständig Page-Objekte für Sie. Das gilt sowohl für die Navigation über das seitliche Navigationsmenü, über Registerkarten oder über die routenbasierte Navigation per Quellcode. In keinem der Fälle müssen Sie sich selbst darum kümmern, die Seite zu erzeugen, indem Sie zum Beispiel `var page1 = new Page1()` schreiben. Dies übernimmt stets die Shell für Sie.

Im Gegensatz zum Vorgänger Xamarin.Forms funktioniert dies unter .NET MAUI sogar im Zusammenspiel mit Dependency Injection auf den Zielseiten. Die Xamarin.Forms-Shell erzeugt Seiten per Reflection über die Methode `Activator.CreateInstance()`. Ein Zugriff auf den ServiceProvider oder zumindest den Xamarin.Forms DependencyService zum Auf-

lösen von Abhängigkeiten ist nicht angedacht. Aus diesem Grund wird nach einem parameterlosen Konstruktor für die Seite gesucht. Ist dieser nicht vorhanden, kommt es zu einem Laufzeitfehler.

Diese Einschränkung gibt es unter .NET MAUI nicht mehr. .NET MAUI versucht, Seiten bei der Navigation zunächst über den ServiceProvider zu erzeugen. Dies funktioniert, wenn sowohl die Abhängigkeiten der Seite (also die Konstruktorparametertypen) **als auch die Seiten selbst** bei der Initialisierung der App an der ServiceCollection registriert wurden. Wenn eine Seite über diesen Weg erzeugt werden kann, dann navigiert die Shell zu ihr. Nur wenn keine Seite erzeugt werden konnte, wird weiterhin die Reflection-Methode `Activator.CreateInstance()` zum Erzeugen der Seiten genutzt. In diesem Fall bleibt die Einschränkung, dass ein parameterloser Konstruktor vorhanden sein muss, auch weiter bestehen.

## ■ 10.4 Navigation in der Beispiel-App

In diesem Kapitel wollen wir, wie bereits im vorherigen Kapitel, an der Beispiel-App weiterarbeiten und diese um eine Navigation erweitern. Das zu erreichende Ziel sehen Sie in Bild 10.13. Es soll eine Shell mit einem seitlichen Navigationsmenü und vier Menüpunkten erzeugt werden. Der Menüpunkt *Speisekarte* soll wiederum eine Registerkartennavigation mit sechs Registerkarten öffnen.

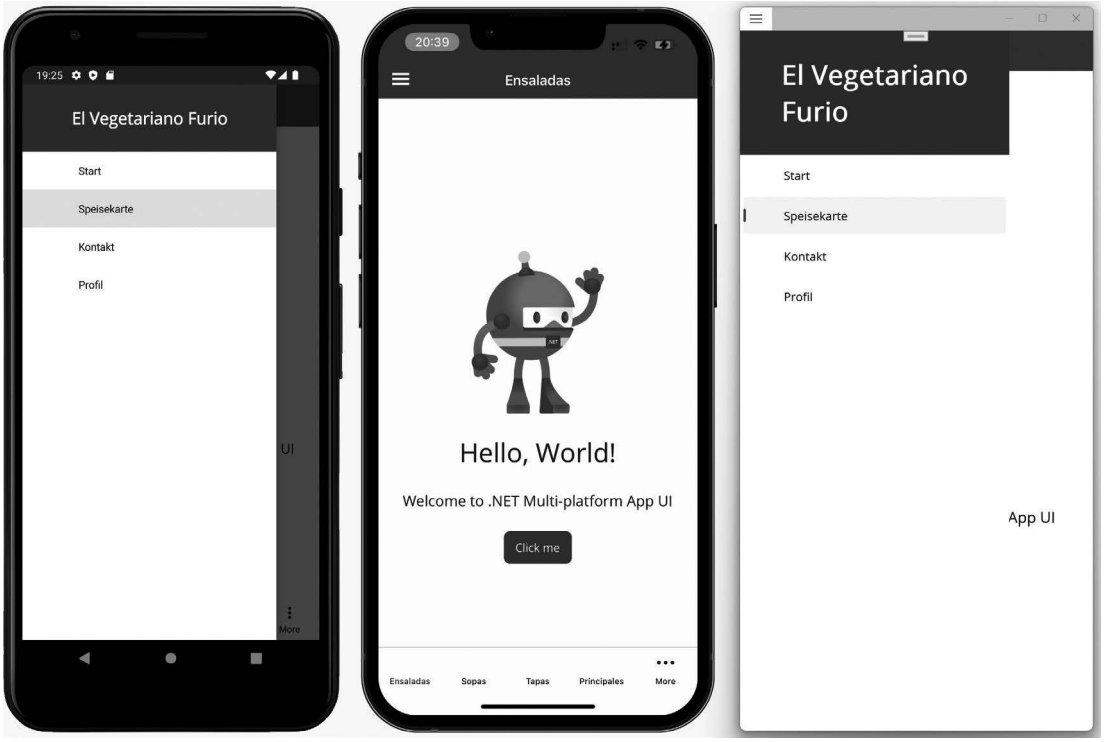


**Bild 10.13** Das Hauptmenü der App ist eine seitliche Navigation (links und rechts). Innerhalb der Speisekarte wird über Registerkarten navigiert (Mitte).

Da wir bisher erst eine Seite in der App angelegt haben, beschränken wir uns in diesem Kapitel auf die grundlegende Struktur der Shell und verlinken dort immer dieselbe Seite.

Den fertigen Quellcode finden Sie im Ordner *Kap10\ElVegetarianoFurio* der Beispielquellcodes zu diesem Buch. Der Quellcode setzt auf dem Beispielcode von Kapitel 9 *(Kap09\ElVegetarianoFurio)* auf.

### 10.4.1 Die Shell anlegen

Um die Aufgabenstellung zu lösen, öffnen wir den letzten Stand des Beispielprojekts aus Kapitel 9 und ersetzen das Markup der Datei *AppShell.xaml* durch den Code aus Listing 10.15.

**Listing 10.15** Die Definition der Shell für unsere Beispielanwendung

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  x:Class="ElVegetarianoFurio.AppShell"
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:ElVegetarianoFurio"
  xmlns:profile="clr-namespace:ElVegetarianoFurio.Profile">
  <Shell.FlyoutHeaderTemplate>
    <DataTemplate>
      <VerticalStackLayout Padding="30" BackgroundColor="DarkRed">
        <Label FontSize="Large"
       TextColor="White"
       FontAttributes="Bold"
       HorizontalOptions="Center">
          El Vegetariano Furio
        </Label>
      </VerticalStackLayout>
    </DataTemplate>
  </Shell.FlyoutHeaderTemplate>

  <ShellContent Title="Start"
  ContentTemplate="{DataTemplate local:MainPage}" Route="MainPage"/>

  <FlyoutItem Title="Speisekarte" Route="menu">
    <ShellContent Title="Ensaladas"
      ContentTemplate="{DataTemplate local:MainPage}" Route="ensaladas"/>
    <ShellContent Title="Sopas"
      ContentTemplate="{DataTemplate local:MainPage}" Route="sopas"/>
    <ShellContent Title="Tapas"
      ContentTemplate="{DataTemplate local:MainPage}" Route="tapas"/>
    <ShellContent Title="Principales"
      ContentTemplate="{DataTemplate local:MainPage}" Route="principales"/>
    <ShellContent Title="Postres"
      ContentTemplate="{DataTemplate local:MainPage}" Route="postres"/>
    <ShellContent Title="Bebidas"
      ContentTemplate="{DataTemplate local:MainPage}" Route="bebidas"/>
  </FlyoutItem>

  <ShellContent Title="Kontakt"
    ContentTemplate="{DataTemplate local:MainPage}" Route="ContactPage"/>
```

```
  <ShellContent Title="Profil"
    ContentTemplate="{DataTemplate profile:ProfilePage}" Route="ProfilePage" />

</Shell>
```

Im Code aus Listing 10.15 starten wir mit der Definition der Kopfzeile mithilfe des Elements `Shell.FlyoutHeaderTemplate`. Die Kopfzeile besteht aus einem einfachen VerticalStackLayout mit rotem Hintergrund und einem Label.

Anschließend legen wir einen Navigations-Menüpunkt und einen weiteren Menüpunkt, der die sechs Registerkarten für die Speisekarte beinhaltet, an. Es folgen zwei weitere Menüpunkte. Alle Menüpunkte mit Ausnahme des Menüpunkts *Profil* verweisen auf die Seite `MainPage`, die durch die Standardvorlage anlegt wurde. Der Grund hierfür ist, dass unsere App bisher nur eine andere Seite hat: die `ProfilePage`. Dies ist im Listing auch die einzige Seite, die korrekt verlinkt wurde. Die korrekte Verlinkung für die fehlenden Menüpunkte folgt in späteren Kapiteln.

### 10.4.2 Das Navigationsframework der Shell abstrahieren

Bis zu dieser Stelle haben wir die grundlegenden Anforderungen an die Shell für unsere Beispielanwendung zwar erfüllt, mit dem Blick in die Zukunft gerichtet bleibt jedoch eine wichtige Sache offen. An späteren Stellen der App möchten wir routenbasierte Navigation nutzen. Der Aufruf der entsprechenden Navigationsmethoden soll in den `ViewModel`-Klassen geschehen. Diese „wissen“ jedoch nichts von der .NET-MAUI-Shell und können somit auch die Methode `GoToAsync` der Shell nicht aufrufen.

Die Lösung des Problems liegt in einer kleinen Schnittstelle mit dem Namen `INavigationService`, die wir im Ordner *Core* des plattformübergreifend geteilten Projekts anlegen. Sie enthält folgenden Code:

```
namespace ElVegetarianoFurio.Core;

public interface INavigationService
{
  Task GoToAsync(string location);
  Task GoToAsync(string location, bool animate);

  Task GoToAsync(string location, Dictionary<string, object> paramters);

  Task GoToAsync(string location, bool animate, Dictionary<string, object>
paramters);
}
```

Es handelt sich bei der Schnittstelle lediglich um eine Abstraktion der vier Überladungen der Methode `GoToAsync` der .NET-MAUI-Shell.

Die Implementierung der Schnittstelle, die wir auch im Ordner *Core* ablegen, ist entsprechend simpel:

```
namespace ElVegetarianoFurio.Core;

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

  public async Task GoToAsync(string location, bool animate)
  {
    await Shell.Current.GoToAsync(location, animate);
  }

  public async Task GoToAsync(string location, Dictionary<string, object> paramters)
  {
    await Shell.Current.GoToAsync(location, paramters);
  }

  public async Task GoToAsync(string location, bool animate, Dictionary<string,
object> paramters)
  {
    await Shell.Current.GoToAsync(location, animate, paramters);
  }
}
```

Um den `NavigationService` über den ServiceProvider auflösen zu können, ergänzen wir außerdem noch die folgende Zeile in der Methode `CreateMauiApp` der Datei *MauiProgram.cs*:

```
builder.Services.AddSingleton<INavigationService, NavigationService>();
```

Somit sind sämtliche Vorbereitungen erledigt, um in den `ViewModel`-Klassen navigieren zu können. An dieser Stelle mag der konkrete Nutzen vielleicht noch nicht ersichtlich sein, spätestens am Ende des nächsten Kapitels werden Sie ihn jedoch erkennen.

## ■ 10.5 Was Sie in diesem Kapitel gelernt haben

In diesem Kapitel haben wir die gängigen Navigationsmuster mobiler Apps – Registerkarten, die seitliche Navigation und die hierarchische Navigation – besprochen. Sie haben gelernt, dass es für jede dieser Navigationsstrategien eine eigene „klassische“ .NET-MAUI-Implementierung gibt. Diese Implementierungen sind zwar noch alle funktionsfähig, sie haben seit der Einführung der .NET-MAUI-Shell jedoch an Bedeutung verloren.

In neuen Apps bietet es sich an, der Shell den Vorzug geben, da diese ein einheitliches Vorgehen ermöglicht. Selbstverständlich kann die Shell auch in Bestands-Apps genutzt werden, um bereits entstandene Komplexität zu verringern. Die notwendigen Schritte, um eine bestehende App um die Shell zu ergänzen, haben Sie in diesem Kapitel gelernt.