Mission 3: Making an interface to interact with the bot

Now that we've established how to connect to the alien MarsBot, we need to let our team members utilize this connection easily. By building a quick Xamarin application with a simple UI, our team members will be able to use all their different devices to talk to and get answers from the bot.

Introduction

In this mission, we will create a simple application page with these components:

1. Create a simple Xamarin UI using XAML
2. Make a ListView for displaying the messages between the user and the bot
3. Make a Entry for sending messages to the bot
4. Making the BotConnection work with the UI through data binding

Breaking it down

In the project, there should be a MainPage.xaml. This .xaml file represents the first page your application will default to when you start it.

Every .xaml file also has a "code-behind" file which can be used to contain the code and logic for that page. You can see this file by clicking the small arrow next to the MainPage.xaml file and clicking on MainPage.xaml.cs.

In this tutorial, we will be using both the .xaml and .xaml.cs files to design the UI and make it work.

Laying out the UI

XAML is used to define the visual contents of an application's page. It allows you to define the page's elements, including its position, color, text and many other properties.

For example, let's open and look at the default MainPage.xaml that we have:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MarsBuddy"
             x:Class="MarsBuddy.MainPage">

        <Label Text="Welcome to Xamarin Forms!"
           VerticalOptions="Center"
           HorizontalOptions="Center" />

</ContentPage>
```

This is a simple layout with 1 Label saying "Welcome to Xamarin Forms!" that is horizontally and vertically centered in the application.

Of course, we can also change many other properties of the Label in XAML, some examples being size and color. That would look something like this:

```
<Label Text="Welcome to Xamarin Forms!"
        TextColor="#77d065"
        FontSize = "20"
        VerticalOptions="Center"
        HorizontalOptions="Center" />
```

For our simple chat application, we want to have 2 things:

1. A ListView to show the messages in the conversation
2. A Entry (text box) which will use to enter text we want to send to the bot

In this scenario, a StackLayout works perfect. A StackLayout allows us to easily position elements in our XAML, setting horizontal and vertical properties.

Replace the Label with this StackLayout element:

```
<StackLayout Spacing="10" Padding="10" HorizontalOptions="Fill" VerticalOptions="Fill"
Orientation="Vertical">
</StackLayout>
```

This StackLayout fills the whole page, with some padding on the edges. It also defines a spacing of 10 between the elements in a vertical order.

A StackLayout is useless without elements in it. Let's add in ListView and Entry elements into the StackLayout.

```
<StackLayout Spacing="10" Padding="10" HorizontalOptions="Fill" VerticalOptions="Fill"
Orientation="Vertical">
        <ListView x:Name="MessageListView"
                        VerticalOptions="StartAndExpand"
                        HorizontalOptions="Fill"
                        >
        </ListView>

        <Entry Placeholder="Message"
                VerticalOptions="End"
                HorizontalOptions="Fill"
                HorizontalTextAlignment="End"
                />
</StackLayout>
```

In this StackLayout, we've added a ListView and Entry. It's important for us to also define the VerticalOptions and HorizontalOptions properties of the elements so that the elements know how to be positioned within the StackLayout.

The ListView in it's current state won't be able to work properly though. Remember that we define the visual properties of elements within our XAML, but a ListView is special in the sense that it has its own elements as well. When the list is populated, it will have its own rows/cells.

Currently, our ListView doesn't know how it should display these cells. Luckily, it's fairly simple to define how these cells should be displayed.

In this snippet, we use a simple TextCell, which is a pre-defined cell type provided by Xamarin. It's a simple cell that has Text and Detail properties.

```
<ListView x:Name="MessageListView"
                VerticalOptions="StartAndExpand"
                HorizontalOptions="Fill"
                >
        <ListView.ItemTemplate>
                <DataTemplate>
                <TextCell Text="{Binding Text}" Detail="{Binding Sender}" />
                </DataTemplate>
        </ListView.ItemTemplate>
</ListView>
```

Connecting the bot to this page

Since we've already abstracted all the connection logic into a different class in Mission 1, we can simply make a new object to connect to the bot in code.

Open the code-behind file MainPage.xaml.cs and simply make a new object with your name in the constructor parameter:

```
//Initialize a connection with ID and Name
```

```
BotConnection connection = new BotConnection("James");
```

Done! Now this page in your application will be able to access the bot through code.

Data binding

Introduction

Often times, UI isn't static. It is dynamic and needs to display data accordingly.

For example, if we wanted to populate a ListView, we can't just hardcode data in. That makes the list pretty useless. What you would do instead, is to retrieve data from a database or service and display it on the list.

But then comes another problem. Data changes all the time and updates can be made to the data. If the data is updated, we will need to manually update the displayed list as well, which means you have to write more code.

As developers, we want a easier way to do this and that's where data binding comes in. Data binding allows you to establish a contract bewteen the data and the display.

ListView Binding

In our case, the ListView will be used to display messages to and from the bot. We will use an ObservableCollection, which is a special collection type that can be binded to a ListView.

Previously in Mission 1, we created a MessageListItem class that we will be able to use here.

Let's open MainPage.xaml.cs and make a new ObservableCollection that takes in the type MessageListItem.

```
ObservableCollection<MessageListItem> messageList = new ObservableCollection<MessageListItem>();
```

The reason we created and used a custom MessageListItem class with the properties Text and Sender, is to bind the properties to the cells in the list.

You can see that binding in the XAML of the ListView.

```
<TextCell Text="{Binding Text}" Detail="{Binding Sender}" />
```

Previously, in our XAML, we gave the ListView a name of "MessageListView". Assigning a name to the element in XAML allows us to access it as a variable in the code-behind file of the XAML.

Let's use that to bind our newly created ObservableCollection to the ListView in the constructor of the MainPage.

```
//Initialize a connection with ID and Name
BotConnection connection = new BotConnection("James");

//ObservableCollection to store the messages to be displayed
ObservableCollection<MessageListItem> messageList = new ObservableCollection<MessageListItem>();

public MainPage()
{
    InitializeComponent();

    //Bind the ListView to the ObservableCollection
    MessageListView.ItemsSource = messageList;
}
```

Now that's done! Any new additions to the messageList collection will magically reflect on the UI in the MessageListView.

Connecting messages to the collection

Let's revise what we currently have. At the moment, we have a ListView UI element that has been binded to an ObservableCollection. This means any changes to the collection will reflect in the ListView.

Every MessageListItem object added to the collection will also get it's own cell with the message data in the ListView.

Now that the binding has been setup, we just have to let any new messages that come in be added to the ObservableCollection. This is where our Mission 1 method comes in handy. We can simply do this:

```
//Initialize a connection with ID and Name
BotConnection connection = new BotConnection("James");

//ObservableCollection to store the messages to be displayed
ObservableCollection<MessageListItem> messageList = new ObservableCollection<MessageListItem>();

public MainPage()
{
    InitializeComponent();

    //Bind the ListView to the ObservableCollection
    MessageListView.ItemsSource = messageList;

    //Start listening to messages and add any new ones to the collection
    var messageTask = connection.GetMessagesAsync(messageList);
}
```

Input events

We also have a Entry box that allows the user to enter messages. We need to make it so that when the user presses enter, it sends a message to the bot.

In order to do that, we have to link the UI up to the code. We want a method to execute when the user presses return.

We can do that by setting an event in the XAML of the Entry element.

```
<Entry Placeholder="Message"
        VerticalOptions="End"
        HorizontalOptions="Fill"
        HorizontalTextAlignment="End"
        Completed="Send"
        />
```

What this change does is that it tells the program: "Hey, when the user presses return, run the Send() method".

Now that we've declared this in the XAML, we also need to define the method in the code-behind file.

Open MainPage.xaml.cs and define a new method named Send():

```
public void Send(object sender, EventArgs args)
{
        //Get text in entry
        var message = ((Entry)sender).Text;

        if(message.Length > 0)
        {
            //Clear entry
            ((Entry)sender).Text = "";

            //Make object to be placed in ListView
```

```
            var messageListItem = new MessageListItem(message, connection.Account.Name);
            messageList.Add(messageListItem);

            //Send the message to the bot
            connection.SendMessage(message);
        }
}
```

Now this method will be executed whenever someone presses return on the Entry box:

1. Get the user's message
2. Clear the entry box
3. Add the message into the collection for display
4. Send the message to the bot

Finish Line

At the end, you should have an application that can talk to the bot.

Try asking it a few questions to test it!

1. How big is mars?
2. How far away is mars?
3. Is there life on mars?

You should get a reply from the bot.

You'll notice there's a an issue with the UI when you run it on Android however. The message cuts off if it's too long and the cells aren't suitable for messages. We'll look into this in Challenge 1 later.