**Team Connect Mapping functionality using Xamarin Forms Map Component.**

Xamarin Forms Map component has the following properties

1. ItemSource property of type IEnumerable, which specifies the collection of IEnumerable items to be displayed.
2. ItemTemplate, of type DataTemplate, which specifies the DataTemplate to apply to each item in the collection of displayed items.

Using these two properties we can bind the collection of EY Office locations to show on the map.

**How to recalculate the distance on Map visible region change**

The Map component as PropertyChanged event, which can be used to get the new visible region and recalculate the distances to EY office locations, as shown below

void BindableMap\_PropertyChanged(System.Object sender, PropertyChangedEventArgs e)

{

var m = (BindableMap)sender;

if (e.PropertyName.Equals ("VisibleRegion"))

{

if(m.VisibleRegion != null)

{

((MainViewModel)BindingContext).GetEYOfficeLocations(m.VisibleRegion.Center.Latitude, m.VisibleRegion.Center.Longitude);

}

}

}

This event can be mapped to Command using behaviors.

**Code Reusability**

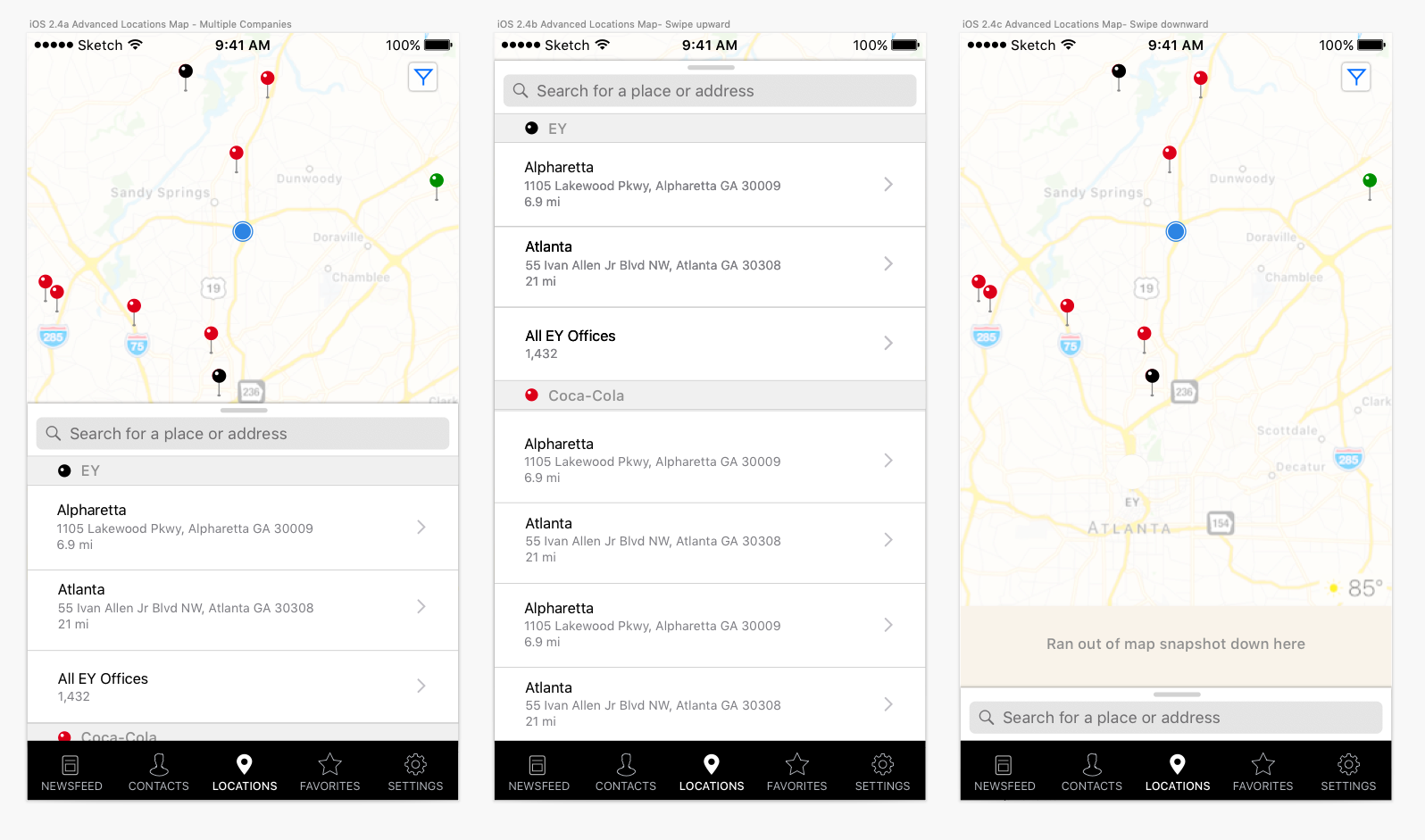
The Current Map functionality code of Team connect can be reused as is, thus reducing the development effort using Xamarin Forms Map component.

**Customizing the Pins**

If we need to use our own customized Pin shape, then we can write a CustomRenderer for Map and set the custom pin image for Pins on Map. Refer the below link for reference

<https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/custom-renderer/map/customized-pin>

**New UX Screen Designs realization for Team Connect**



The above UX screen designs can’t be realized with out of box controls available in Xamarin Forms.

we can realize the above UX screen design by implementing platform-specific customizations by creating a custom renderer for a [ContentPage](https://docs.microsoft.com/en-us/dotnet/api/xamarin.forms.contentpage) on each platform. The process for doing this is as follows:

1. [Create](https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/custom-renderer/contentpage#Creating_the_Xamarin.Forms_Page) a Xamarin.Forms page.
2. [Consume](https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/custom-renderer/contentpage#Consuming_the_Xamarin.Forms_Page) the page from Xamarin.Forms.
3. [Create](https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/custom-renderer/contentpage#Creating_the_Page_Renderer_on_each_Platform) the custom renderer for the page on each platform.

An unaltered [ContentPage](https://docs.microsoft.com/en-us/dotnet/api/xamarin.forms.contentpage) can be added to the shared Xamarin.Forms project, as shown in the following XAML code example:

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="CustomRenderer.MapPage">

<ContentPage.Content>

</ContentPage.Content>

</ContentPage>

Similarly, the code-behind file for the [ContentPage](https://docs.microsoft.com/en-us/dotnet/api/xamarin.forms.contentpage) should also remain unaltered, as shown in the following code example:

public partial class MapPage: ContentPage

{

public MapPage ()

{

// A custom renderer is used to display the map UI

InitializeComponent ();

}

}

**Page Renderer on iOS:**

On iOS platform an overridden version of the OnElementChanged method in the MapPageRenderer class is the place to perform the native page customization. A reference to the Xamarin.Forms page instance that's being rendered can be obtained through the Element property.

We can create the BottomSheetView, with list view and other required UI elements in the view and add MKMapView and the BottomSheetView as sub views to the page view.

We can define three anchor points based on the screen resolution, like 0% , 55% & 95% of the page view height.

We can add Up and Down swipe gesture for BottomSheetView as shown below

// Create a new swipe up gesture

UISwipeGestureRecognizer gestureUp = new UISwipeGestureRecognizer();

gestureUp.Direction = UISwipeGestureRecognizerDirection.Up;

// Wire up the event handler (have to use a selector)

gestureUp.AddTarget(() => HandleUpDrag(gestureUp)); // to be defined

// Add the gesture recognizer to the view

bottomSheetView.AddGestureRecognizer(gestureUp);

// Create a new swipe down gesture

UISwipeGestureRecognizer gestureDown = new UISwipeGestureRecognizer();

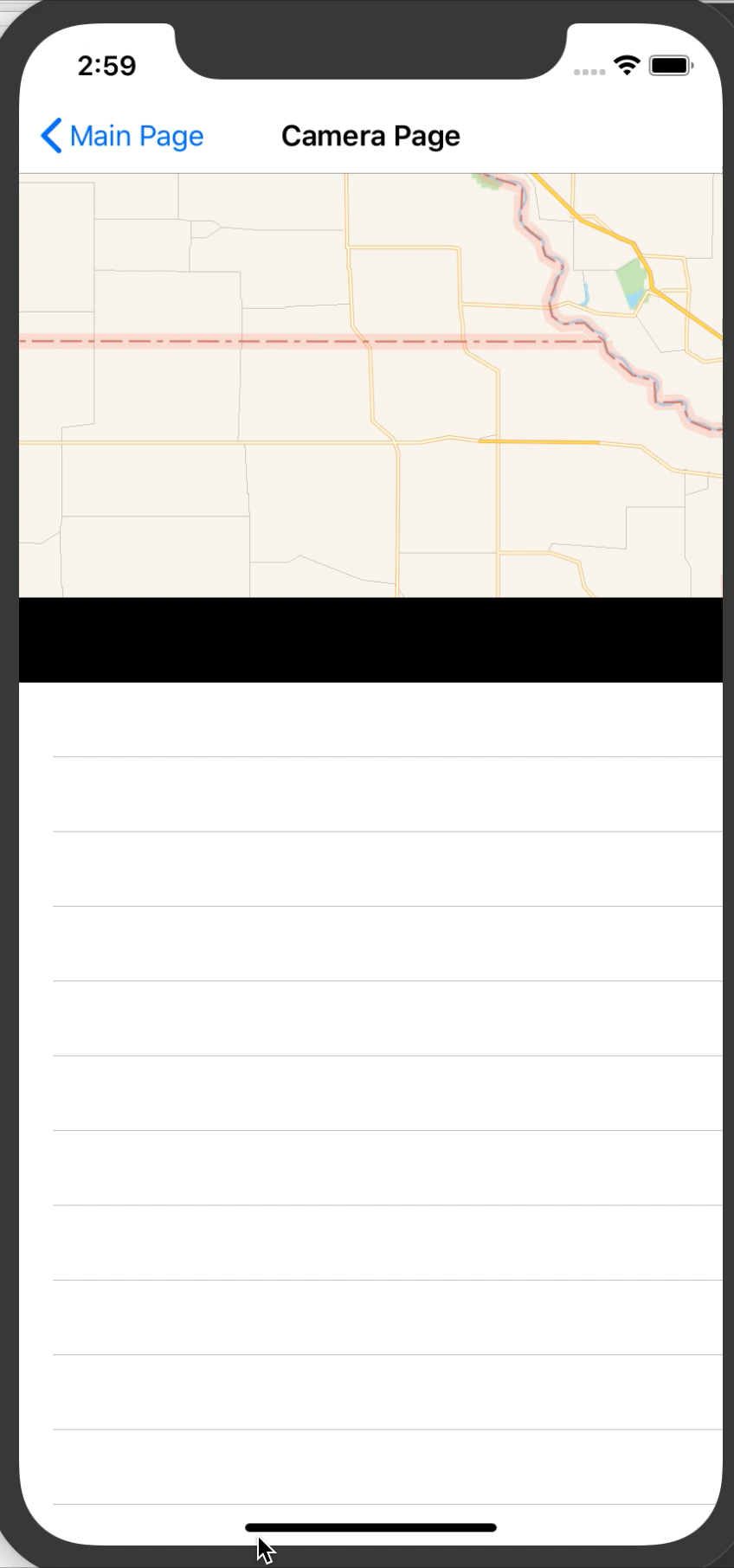
gestureDown.Direction = UISwipeGestureRecognizerDirection.Down;

// Wire up the event handler (have to use a selector)

gestureDown.AddTarget(() => HandleDownDrag(gestureDown));

Now, based on the swipe gesture and the current anchor position of the BottomSheetView, the BottoSheetView position can be changed to next anchor point with sliding animation using UIViewPropertyAnimator.

Enclosed the screen recording of the sample app depicting the above ux screen design



**Page Renderer on Android:**

Bottom Sheets are supported natively in AndroidX. There are two types of bottom sheets supported: [persistent](https://www.google.com/design/spec/components/bottom-sheets.html#bottom-sheets-persistent-bottom-sheets) and [modal](https://www.google.com/design/spec/components/bottom-sheets.html#bottom-sheets-modal-bottom-sheets). Persistent bottom sheets show in-app content, while modal sheets expose menus or simple dialogs.

The same screen layout can be realized in Android using Coordinator Layout with nested scroll view and bottom sheet view behavior.

The below links can be referred to realize this functionality for Android.

<https://guides.codepath.com/android/handling-scrolls-with-coordinatorlayout>

<https://stackoverflow.com/questions/34160423/how-to-mimic-google-maps-bottom-sheet-3-phases-behavior>

<https://github.com/miguelhincapie/CustomBottomSheetBehavior>

To realize this functionality we need to use the below pre-release version of AndroidX nuget packages.

Xamarin.Google.Android.Material

Xamarin.Android.Support.CoordinaterLayout

Xamarin.AndroidX.AppCompat

Xamarin.AndroidX.Browser

Xamarin.AndroidX.CoordinatorLayout

Xamarin.AndroidX.Core

Xamarin.AndroidX.Legacy.Support.Core.UI

Xamarin.AndroidX.Legacy.Support.Core.Utils

Xamarin.AndroidX.Legacy.Support.V4

Xamarin.AndroidX.Lifecycle.Common

Xamarin.AndroidX.Lifecycle.LiveData

Xamarin.AndroidX.Lifecycle.Runtime

Xamarin.AndroidX.Migration

In OnElementChanged of PageRenderer we can inflate this CoordinatorLayout along with BottmSheetViewBehavior to realize the UX screen.

**Pitfalls**

As we rely on native implementation of each platform to realize the UX design, it will be more easy to integrate the native implementation of Maps in each platform then using Xamarin Forms map component. As we will have two separate implementations for this page, and hence the maintenance cost in the long run.

**Effort**

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Effort in hours** | **Reason** | **Performance** |
| Effort required to realize current functionality and screen design for mapping functionality in Team connect using Xamarin Forms | 16 | Most of the code of existing implementation can be reused. Easy to integrate with Xamarin Forms Map component. One implementation for both platforms. | There is little performance hit using Xamarin Forms map component, but usable for team connect |
| Effort required to realize the Advanced screen designs in iOS | 24 - 30 |  |  |
| Effort required to realize the Advanced screen designs in Android | 40 - 45 | Implement BottomSheetViewBehavior and render the coordinator layout in Xamarin forms. |  |