Maps

Evolve Advanced Track, Chapter 7

Overview

Mapping technologies are a ubiquitous complement to mobile devices. Desktop computers and laptops don't tend to have location awareness built-in. On the other hand, mobile devices use such applications to display location information.

Both iOS and Android have powerful mapping capabilities that allow displaying and annotating maps. Both offer similar capabilities such as zooming and panning, displaying satellite imagery and annotating a map with graphics. One large difference between the APIs is on iOS mapping is built-in with the SDK, whereas on Android it is available from Google as a separate add-on library.

On iOS mapping is available via the *Map Kit* framework. Map Kit makes it easy to add interactive, customized maps to an application. On Android, maps are available from Google as part of the *Google Play Services*. As with iOS, Android maps are feature rich and support user touch interaction as well as map annotations.

In this chapter, we'll examine iOS Map Kit and show how to add mapping capability to the EvolveLite application. Then, we'll examine some of the features of Google Android maps and demonstrate using them from Xamarin.Android.

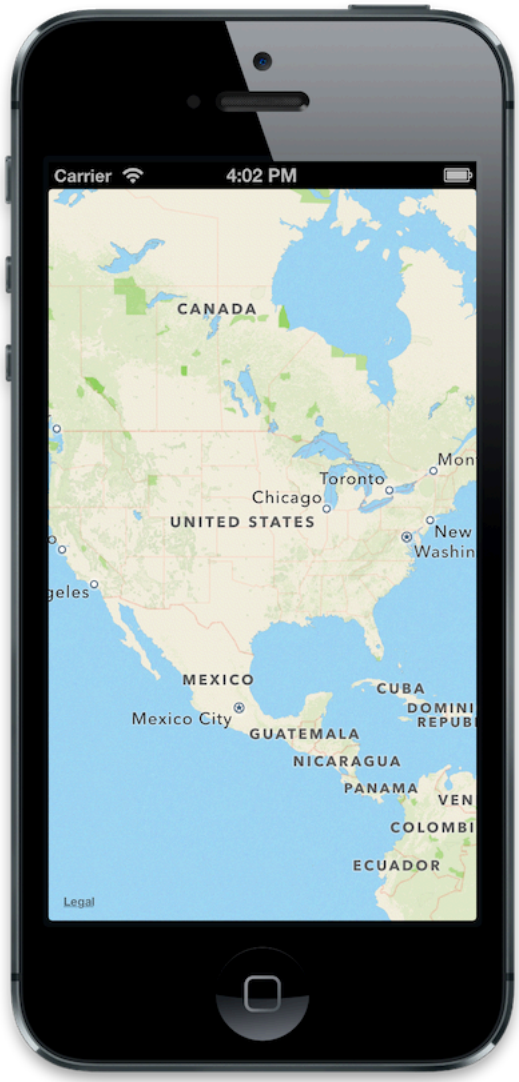
iOS Maps

Adding a Map

Adding a map to an application is accomplished by adding an `MKMapView` instance to the view hierarchy, as shown below:

```
// map is an MKMapView declared as a class variable
map = new MKMapView (UIScreen.MainScreen.Bounds);
View = map;
```

`MKMapView` is a `UIView` subclass that displays a map. Simply adding the map using the code above produces an interactive map:



Map Style

`MKMapView` supports 3 different styles of maps. To apply a map style, simply set the `MapType` property to a value from the `MapType` enumeration. The following screenshot show the different map styles that are available:

Standard Satellite Hybrid

Panning and Zooming

`MKMapView` includes support for map interactivity features such as:

- Zooming via a pinch gesture
- Panning via a pan gesture

These features can be enabled or disabled by simply setting the `ZoomEnabled` and `ScrollEnabled` properties of the `MKMapView` instance, where the default value is true for both. For example, to display a static map, simply set the appropriate properties to false:

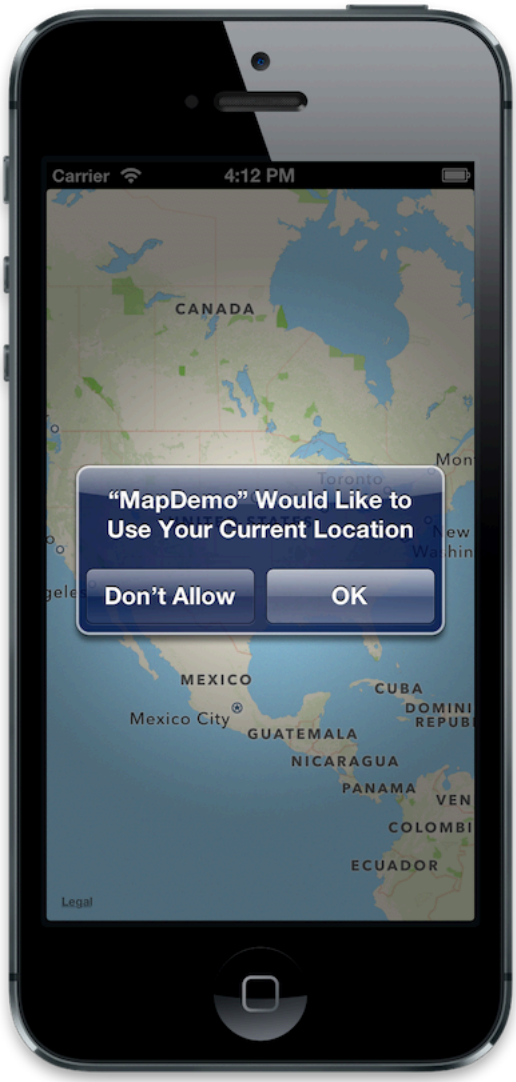
```
map.ZoomEnabled = false;
map.ScrollEnabled = false;
```

User Location

In addition to user interaction, `MKMapView` also has built-in support for displaying the location of the device. It does this internally by using the *Core Location* framework. However, enabling this is as simple as setting the `ShowsUserLocation` property to true:

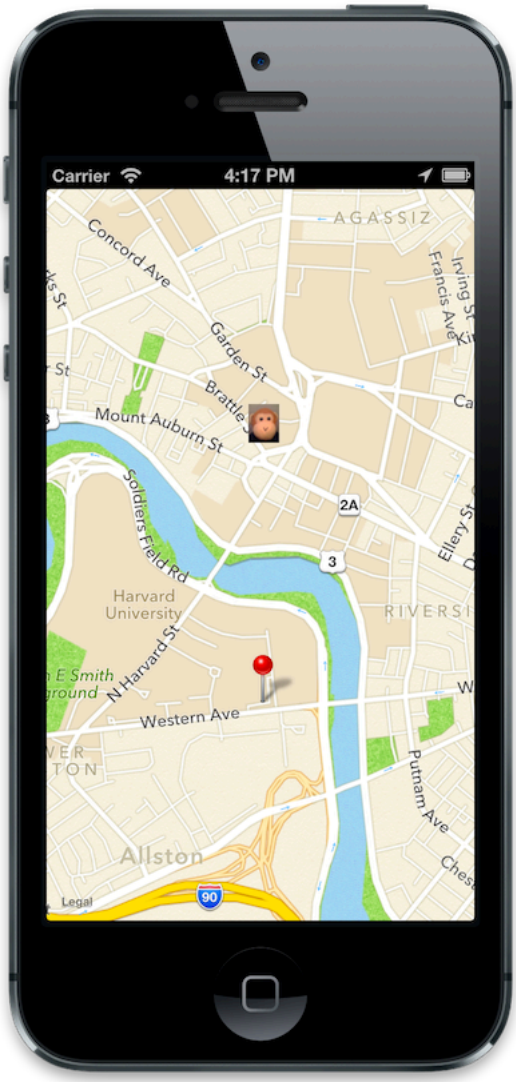
```
map.ShowsUserLocation = true;
```

When user location is enabled, the first time the application runs, the user will be prompted to allow location services for the application, as shown below:



Annotations

`MKMapView` also supports displaying images, known as annotations, on a map. These can be either custom images or system-defined pins of various colors. For example, the following screenshot shows a map with a both a pin and a custom image:



Adding an annotation

An annotation itself has two parts:

- The `MKAnnotation` object, which includes model data about the annotation, such as the title and location of the annotation.
- The `MKAnnotationView`, which contains the image to display and optionally a callout that is shown when the user taps the annotation.

Map Kit uses the iOS delegation pattern to add annotations to a map, where the `Delegate` property of the `MKMapView` is set to an instance of an `MKMapViewDelegate`. It is this delegate's implementation that is responsible for returning the `MKAnnotationView` for an annotation.

To add an annotation, first the annotation is added by calling `AddAnnotation` on the `MKMapView` instance:

```
// add an annotation
map.AddAnnotation (new MKPointAnnotation (){
```

```
        Title="MyAnnotation",
        Coordinate = new CLLocationCoordinate2D (42.364260, -71.120824)
});
```

When the location of the annotation becomes visible on the map, the `MKMapView` will call its delegate's `GetViewForAnnotation` method to get the `MKAnnotationView` to display.

For example, the following code returns a system-provided `MKPinAnnotationView`:

```
string pId = "PinAnnotation";

public override MKAnnotationView GetViewForAnnotation (MKMapView mapView,
NSObject annotation)
{
    MKAnnotationView pinView;

    if (annotation is MKUserLocation)
        return null;

    // create pin annotation view
    MKAnnotationView pinView =
(MKPinAnnotationView)mapView.DequeueReusableAnnotation (pId);

    if (pinView == null)
        pinView = new MKPinAnnotationView (annotation, pId);

    ((MKPinAnnotationView)anView).PinColor = MKPinAnnotationColor.Red;
    pinView.CanShowCallout = true;

    return pinView;
}
```

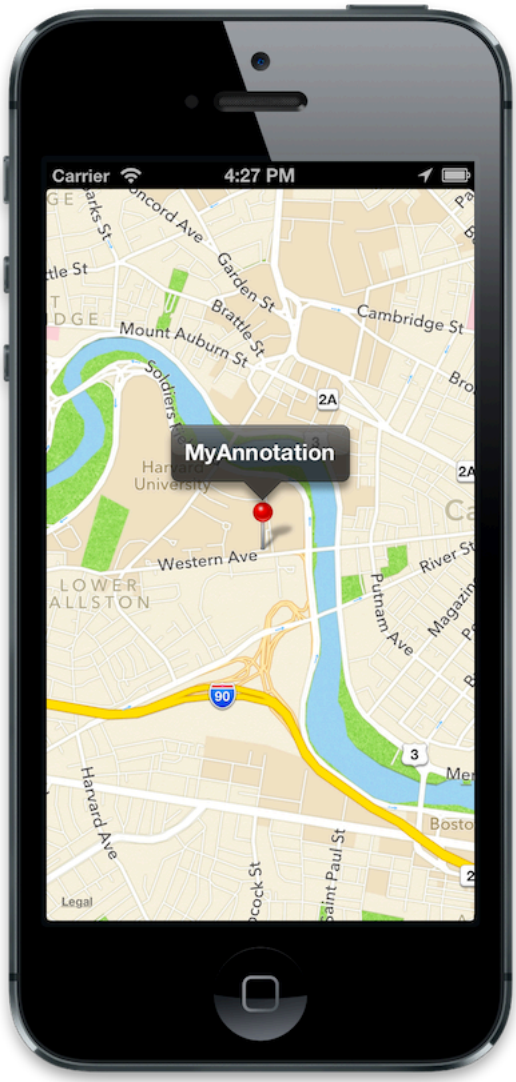
Reusing Annotations

To conserve memory, `MKMapView` allows annotation view's to be pooled for reuse, similar to the way table cells are reused. Obtaining an annotation view from the pool is done with a call to `DequeueReusableAnnotation`:

```
MKAnnotationView pinView =
(MKPinAnnotationView)mapView.DequeueReusableAnnotation (pId);
```

Showing Callouts

As mentioned earlier, an annotation can optionally show a callout. To show a callout simply set `CanShowCallout` to true on the `MKAnnotationView`. This results in the annotation's title being displayed when the annotation is tapped, as shown:

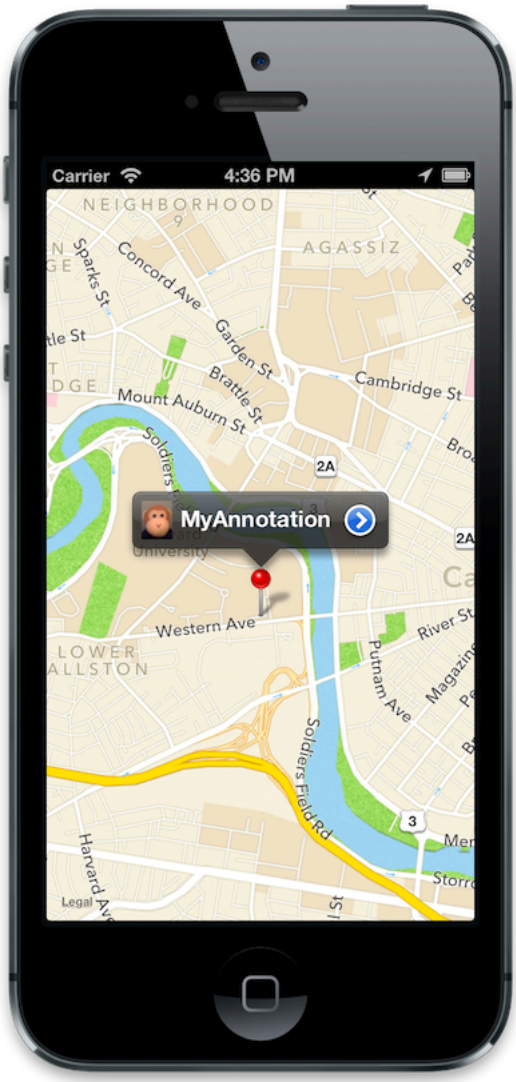


Customizing the Callout

The callout can also be customized to show left and right accessory views, as shown below:

```
pinView.RightCalloutAccessoryView = UIButton.FromType
(UIButtonType.DetailDisclosure);
pinView.LeftCalloutAccessoryView = new UIImageView(UIImage.FromFile
("monkey.png"));
```

This code results in the following callout:



To handle the user tapping the right accessory, simply implement the `CalloutAccessoryControlTapped` method in the `MKMapViewDelegate`:

```
public override void CalloutAccessoryControlTapped (MKMapView mapView,
MKAnnotationView view, UIControl control)
{
        ...
}
```

Overlays

Another way to layer graphics on a map is using overlays. Overlays support drawing graphical content that scales with the map as it is zoomed. iOS provides support for several types of overlays, including:

- Polygons - Commonly used to highlight some region on a map.
- Polylines - Often seen when showing a route.
- Circles - Used to highlight a circular area of a map.

Additionally, custom overlays can be created to show arbitrary geometries with granular, customized drawing code. For example, weather radar would be a good candidate for a custom overlay.

Adding an Overlay

Similar to annotations, adding an overlay involves 2 parts:

- Creating a model object for the overlay and adding it to the `MKMapView`.
- Creating a view for the overlay in the `MKMapViewDelagate`.

The model for the overlay can be any `MKShape` subclass. Xamarin.iOS includes `MKShape` subclasses for polygons, polylines and circles, via the `MKPolygon`, `MKPolyline` and `MKCircle` classes respectively.

For example, the following code is used to add an `MKCircle`:

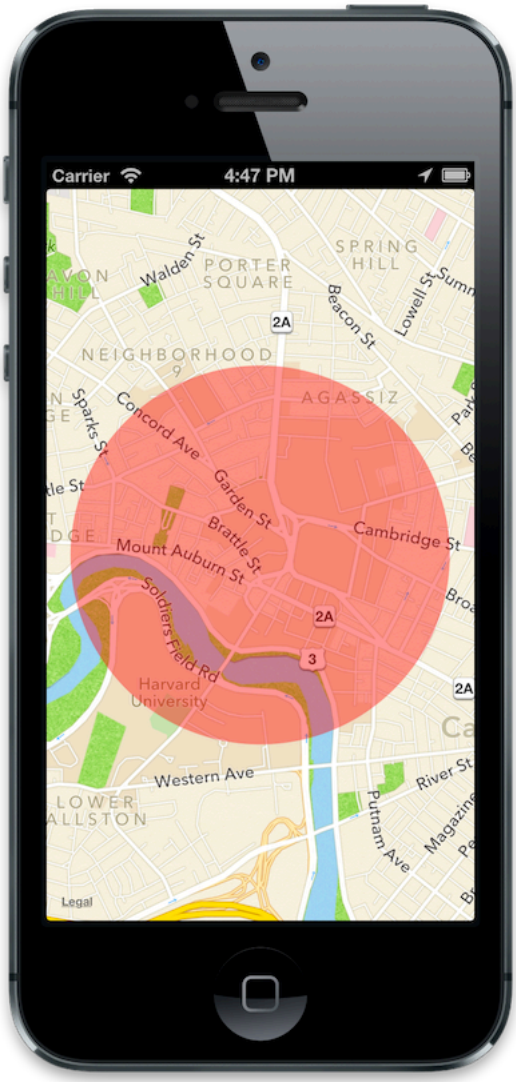
```
var circleOverlay = MKCircle.Circle (mapCenter, 1000);
map.AddOverlay (circleOverlay);
```

The view for an overlay is an `MKOverlayView` instance that is returned by the `GetViewForOverlay` in the `MKMapViewDelegate`. Each `MKShape` has a corresponding `MKOverlayView` that knows how to display the given shape. For `MKPolygon` there is `MKPolygonView`. Similarly, `MKPolyline` corresponds to `MKPolylineView`, and for `MKCircle` there is `MKCircleView`.

For example, the following code returns an `MKCircleView` for an `MKCircle`:

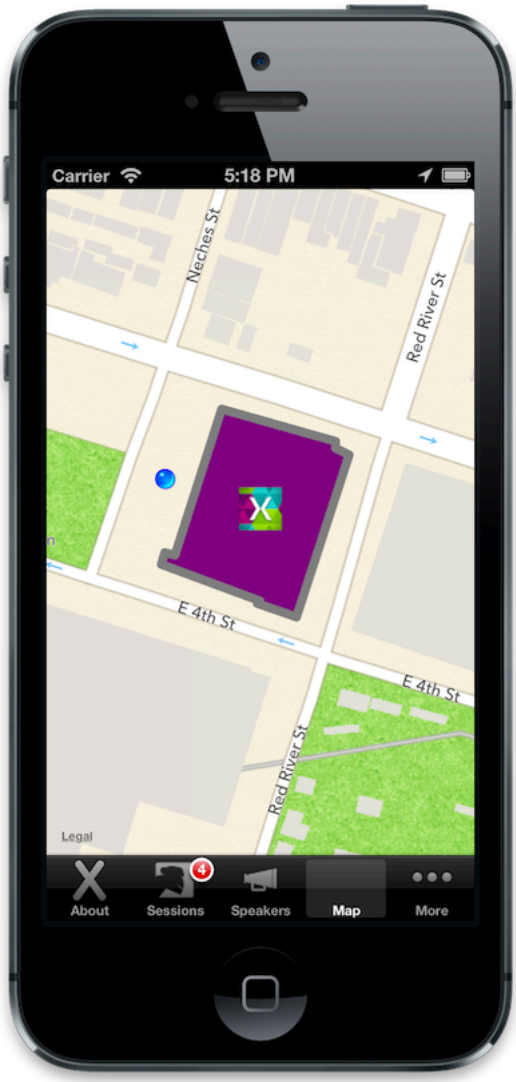
```
public override MKOverlayView GetViewForOverlay (MKMapView mapView,
NSObject overlay)
{
    var circleOverlay = overlay as MKCircle;
    var circleView = new MKCircleView (circleOverlay);
    circleView.FillColor = UIColor.Blue;
    return circleView;
}
```

This displays a circle on the map as shown:



EvolveLite Walkthrough

For this exercise we're going to add a map to the EvolveLite sample application that displays an annotation at the location of the conference, and an overlay of the hotel as shown below:



Open the EvolveLite sample. We'll be adding code to the `EvolveMapViewController` class contained in the EvolveMapViewController.cs file located in the **ViewControllers** folder.

1. Add the following namespaces to the `EvolveMapViewController`:

```
using MonoTouch.MapKit;
using MonoTouch.CoreLocation;
```

2. Add an `MKMapView` instance variable to the class, along with a `MapDelegate` instance. We'll create the `MapDelegate` shortly:

```
public partial class EvolveMapViewController : UIViewController
{
        MKMapView map;
        MapDelegate mapDelegate;
        …
```

3. In the controller's `LoadView` method, add an `MKMapView` and set it to the `View` property of the controller:

```
public override void LoadView ()
{
        map = new MKMapView (UIScreen.MainScreen.Bounds);
        View = map;
}
```

Next, we'll add code to initialize the map in the `ViewDidLoad` method.

4. In `ViewDidLoad` add code to set the map type, show the user location and allow zooming and panning:

```
// change map type, show user location and allow zooming and panning
map.MapType = MKMapType.Standard;
map.ShowsUserLocation = true;
map.ZoomEnabled = true;
map.ScrollEnabled = true;
```

5. Next add code to center the map and set it's region:

```
double lat = 30.2652233534254;
double lon = -97.73815460962083;
CLLocationCoordinate2D mapCenter = new CLLocationCoordinate2D (lat, lon);
MKCoordinateRegion mapRegion = MKCoordinateRegion.FromDistance (mapCenter,
200, 200);
map.CenterCoordinate = mapCenter;
map.Region = mapRegion;
```

6. Create an instance of the `MapDelegate` and assign it to the `Delegate` of the `MKMapView`. Again, we'll implement the `MapDelegate` shortly:

```
mapDelegate = new MapDelegate ();
map.Delegate = mapDelegate;
```

7. We're going to use a custom class for the annotation called `ConferenceAnnotation`. Add the following class to the project:

```
using System;
using MonoTouch.CoreLocation;
using MonoTouch.MapKit;

namespace MapDemo
{
    public class ConferenceAnnotation : MKAnnotation
    {
        string title;
        CLLocationCoordinate2D coord;

        public ConferenceAnnotation (string title,
               CLLocationCoordinate2D coord)
        {
            this.title = title;
            this.coord = coord;
        }

        public override string Title {
            get {
                return title;
            }
```

```
        }

        public override CLLocationCoordinate2D Coordinate {
            get {
                return coord;
            }
            set {
               // call WillChangeValue and DidChangeValue to use KVO with
               // an MKAnnotation so that setting the coordinate on the
               // annotation instance causes the associated annotation
               // view to move to the new location.

               WillChangeValue ("coordinate");
               coord = value;
               DidChangeValue ("coordinate");
            }
        }
    }
}
```

8. With the `ConferenceAnnotation` in place we can add it to the map. Back in the `ViewDidLoad` method of the `EvolveMapController`, add the annotation at the map's center coordinate:

```
map.AddAnnotation (new ConferenceAnnotation ("Evolve Conference",
mapCenter));
```

9. We also want to have an overlay of the hotel. Add the following code to create the `MKPolygon` using the coordinates for the hotel provided, and add it to the map by call `AddOverlay`:

```
// add an overlay of the hotel
MKPolygon hotelOverlay = MKPolygon.FromCoordinates(
        new CLLocationCoordinate2D[]{
        new CLLocationCoordinate2D(30.2649977168594, -97.73863627705),
        new CLLocationCoordinate2D(30.2648461170005, -97.7381627734755),
        new CLLocationCoordinate2D(30.2648355402574, -97.7381750192576),
        new CLLocationCoordinate2D(30.2647791309417, -97.7379872505988),
        new CLLocationCoordinate2D(30.2654525150319, -97.7377341711021),
        new CLLocationCoordinate2D(30.2654807195004, -97.7377994819399),
        new CLLocationCoordinate2D(30.2655089239607, -97.7377994819399),
        new CLLocationCoordinate2D(30.2656428950368, -97.738346460207),
        new CLLocationCoordinate2D(30.2650364981811, -97.7385709662122),
        new CLLocationCoordinate2D(30.2650470749025, -97.7386199493406)
});

map.AddOverlay (hotelOverlay);
```

This completes the code in `ViewDidLoad`. Now we need to implement our `MapDelegate` class to handle creating the annotation and overlay views respectively.

10. Create a class called `MapDelegate` that inherits from `MKMapViewDelegate` and include an `annotationId` variable to use as a reuse identifier for the annotation:

```
class MapDelegate : MKMapViewDelegate
{
```

```
    static string annotationId = "ConferenceAnnotation";
        ...
}
```

We only have one annotation here so the reuse code isn't strictly necessary, but it's a good practice to include it.

11. Implement the `GetViewForAnnotation` method to return a view for the `ConferenceAnnotation` using the conference.png image included in the project's **images** folder:

```
public override MKAnnotationView GetViewForAnnotation (MKMapView mapView,
NSObject annotation)
{
    MKAnnotationView annotationView = null;

    if (annotation is MKUserLocation)
        return null;

    if (annotation is ConferenceAnnotation) {

        // show conference annotation
               annotationView = mapView.DequeueReusableAnnotation
(annotationId);

        if (annotationView == null)
                    annotationView = new MKAnnotationView (annotation,
annotationId);

        annotationView.Image = UIImage.FromFile ("images/conference.png");
        annotationView.CanShowCallout = false;
    }

    return annotationView;
}
```

12. When the user taps on the annotation, we want to display an image showing the city of Austin. Add the following variables to the `MapDelegate` for the image and the view to display it:

```
UIImageView venueView;
UIImage venueImage;
```

13. Next, to show the image when the annotation is tapped, implement the `DidSelectAnnotation` method as follows:

```
public override void DidSelectAnnotationView (MKMapView mapView,
MKAnnotationView view)
{
    // show an image view when the conference annotation view is selected
    if (view.Annotation is ConferenceAnnotation) {

        venueView = new UIImageView ();
        venueView.ContentMode = UIViewContentMode.ScaleAspectFit;
        venueImage = UIImage.FromFile ("image/venue.png");
        venueView.Image = venueImage;
        view.AddSubview (venueView);
```

```
            UIView.Animate (0.4, () => {
                venueView.Frame = new RectangleF (-75, -75, 200, 200); });
        }
    }
```

14. To hide the image when the user deselects the annotation by tapping anywhere else on the map, implement the `DidSelectAnnotationView` method as follows:

```
public override void DidDeselectAnnotationView (MKMapView mapView,
MKAnnotationView view)
{
    // remove the image view when the conference annotation is deselected
    if (view.Annotation is ConferenceAnnotation) {

        venueView.RemoveFromSuperview ();
        venueView.Dispose ();
        venueView = null;
    }
}
```

We now have the code for the annotation in place. All that is left is to add code to the `MapDelegate` to create the view for the hotel overlay.

15. Add the following implementation of `GetViewForOverlay` to the `MapDelegate`:

```
public override MKOverlayView GetViewForOverlay (MKMapView mapView,
NSObject overlay)
{
    // return a view for the polygon
    MKPolygon polygon = overlay as MKPolygon;
    MKPolygonView polygonView = new MKPolygonView (polygon);
    polygonView.FillColor = UIColor.Purple;
    polygonView.StrokeColor = UIColor.Gray;
    return polygonView;
}
```

Run the application. We now have an interactive map with a custom annotation and an overlay! Tap on the annotation and the image of Austin is displayed.

Google Android Maps

Google Maps are available for Android via *Google Maps Android API V2* that is part of the *Google Play Services* library. In order to use Google Maps, a device must have Google Play Services installed. Additionally, the device must support *OpenGL ES 2* as Google Maps uses it for drawing map title vectors. Unfortunately, at the time of this writing, Google Maps is not supported in the emulator.

For more information on setting up Google Maps Android API V2, see the following:

https://github.com/xamarin/monodroid-samples/blob/master/MapsAndLocationDemo_v2/README.md

Adding a Map

The `GoogleMap` class encapsulates a map. It allows features such as:

- Controlling the map type
- Adding graphics and markers
- Zooming and panning the map
- Adding traffic data

A `GoogleMap` instance is added to a map using a fragment. Either the `MapFragment` class, for API Level 11 and above, or the `SupportMapFragment` class, for API Level less than 11, is used to encapsulate a `GoogleMap`.

Using XML

For example, adding the following XML to an Android layout file will include a `SupportMapFragment` that contains a `GoogleMap` instance:

```
<fragment
        android:id="@+id/map"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.google.android.gms.maps.SupportMapFragment" />
```

This results in the map shown below:

Using Code

A map can also be added in code using a fragment transaction. For example, say a layout contained the following XML:

```
<FrameLayout
        android:id="@+id/map"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
```

The following code creates a `SupportMapFragment` instance that encapsulates a `GoogleMap` and adds it to the `FrameLayout` above:

```
Android.Support.V4.App.FragmentTransaction ftrans =
SupportFragmentManager.BeginTransaction();
SupportMapFragment mapFragment = SupportMapFragment.NewInstance();
ftrans.Add(Resource.Id.map, mapFragment, "map");
ftrans.Commit();
```

This produces the same map shown earlier.

Changing the Map Type

The `GoogleMapOptions` class supports adding various features to a map. For example, the following code changes the map type to satellite, and adds built-in zoom controls:

```
GoogleMapOptions mapOptions = new GoogleMapOptions ()
        .InvokeMapType (GoogleMap.MapTypeSatellite)
        .InvokeZoomControlsEnabled (true);
```

The `GoogleMapOptions` instance is then added to the `mapFragment` as shown below:

```
Android.Support.V4.App.FragmentTransaction ftrans =
SupportFragmentManager.BeginTransaction();
SupportMapFragment mapFragment =
SupportMapFragment.NewInstance(mapOptions);
ftrans.Add(Resource.Id.map, mapFragment, "map");
ftrans.Commit();
```

This results in the map shown below, where clicking on the zoom buttons changes the map's zoom level:

Obtaining the GoogleMap Instance

Once a map has been added via a fragment, the `GoogleMap` instance can be obtained from the `Map` property of the map fragment. However, if the map hasn't been completely initialized, the `Map` property may return null. Therefore, it is a good practice to set the map to a class variable inside `OnResume`, where the map should be properly initialized, as shown below:

```
GoogleMap map;

protected override void OnResume()
{
    base.OnResume();
    SupportMapFragment mapFragment = (SupportMapFragment)
SupportFragmentManager.FindFragmentByTag("map");

    map = mapFragment.Map;
}
```

Changing the Camera

The `GoogleMap` instance includes a variety of methods of interacting with a map. For example, it supports moving the map to display a new location by moving the map's camera.

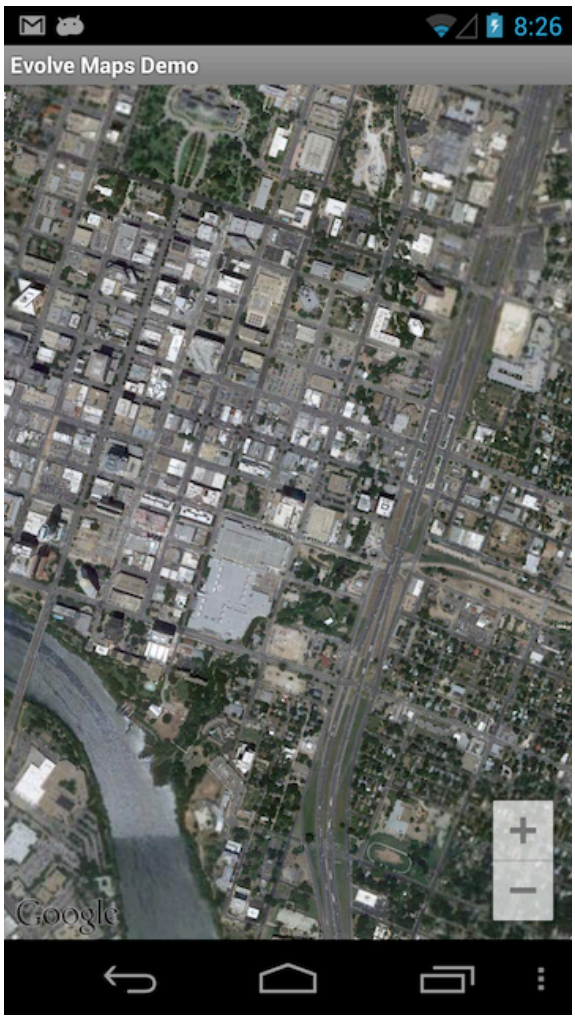
You can define a `CameraUpdate` instance to encapsulate some location and zoom level to display using the `CameraUpdateFactory`, as shown below for a location in Austin, TX:

```
// Create an instance of CameraUpdate
LatLng Austin = new LatLng(30.2652233534254, -97.73815460962083);
CameraUpdate cameraUpdate = CameraUpdateFactory.NewLatLngZoom(Austin, 15);
```

Then you can move the map to this location by calling the `MoveCamera` method on the `GoogleMap` instance:

```
// Move the map
map.MoveCamera(cameraUpdate);
```

The resulting map displays Austin,TX:



Adding a Marker

A `Marker` is an image corresponding to a location on a map. The `GoogleMap` class has an `AddMarker` method that makes it easy to add a marker. To create a marker, simply create a `MarkerOptions` instance and set the state of the marker. `MarkerOptions` uses a fluent API, so chaining method calls together initializes it. For example, the following code creates a `MarkerOptions` instance for the Evolve conference in Austin, TX:

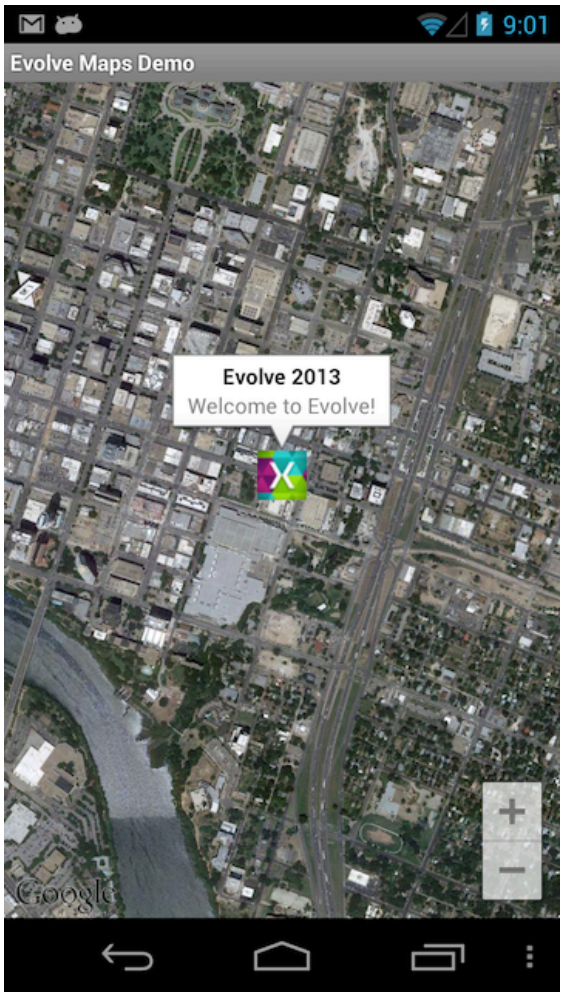
```
// image to use for the marker
BitmapDescriptor icon =
BitmapDescriptorFactory.FromResource(Resource.Drawable.conference);

// create a marker in Austin, TX
MarkerOptions markerOptions = new MarkerOptions ()
    .SetSnippet ("Welcome to Evolve!")
    .SetPosition (Austin)
    .SetTitle ("Evolve 2013")
    .InvokeIcon (icon);
```

Then, to add the marker, simply pass the `MarkerOptions` to the `AddMarker` method:

```
// add the marker to the map
Marker evolveMarker = map.AddMarker (markerOptions);
evolveMarker.ShowInfoWindow ();
```

The map with the marker is shown below:

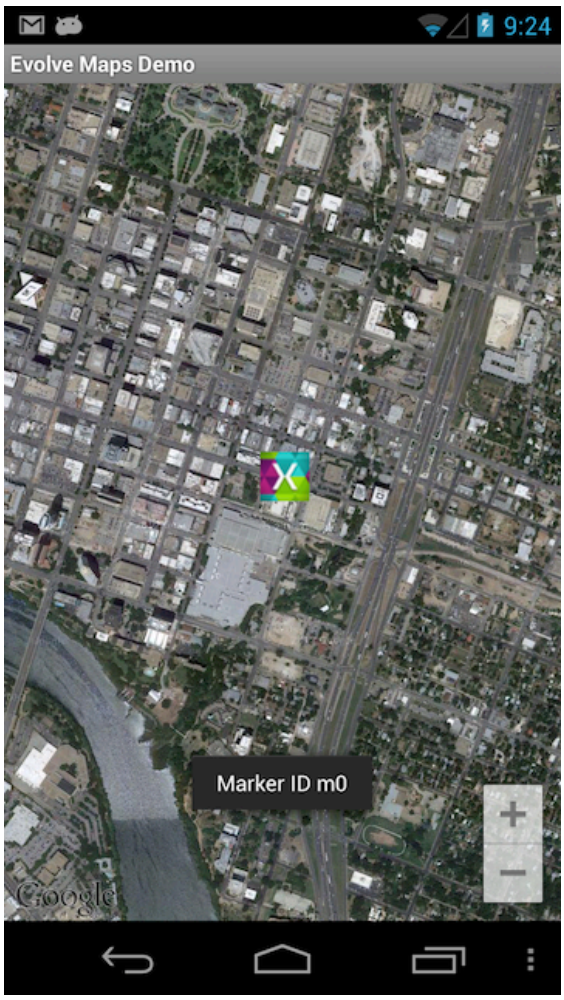


Handling Marker Interaction

The `GoogleMap` class has a `MarkerClick` event that is used to handle marker clicks on the map. The `MarkerClickEventArgs` class provides access to the marker instance. The `Marker` class has an `Id` property that can be used to determine which marker was clicked, as shown below:

```
// Handle marker click
map.MarkerClick += (object sender, GoogleMap.MarkerClickEventArgs e) => {
    Marker marker = e.P0;
    Toast.MakeText(this, String.Format("Marker ID {0}", marker.Id),
ToastLength.Short).Show();
};
```

Clicking on the marker closes the info window and executes the handler's code, which in this case displays a toast:



Summary

In this chapter we examined the *Map Kit* framework for iOS and the *Google Maps Android API V2* for Android. Starting with iOS, we looked at how the `MKMapView` class allows interactive maps to be included in an application. We then saw how to further customize maps using annotations and overlays. Next, we walked through an exercise that demonstrated adding an annotation and an overlay to a map in the EvolveLite application.

We then shifted gears to Android, examining several features of the Google Maps Android API V2. We saw how to add a map and change the map type, in addition to adding built-in controls for zoom and interactive markers.