Social + Components

Xamarin Evolve, Chapter 11

Overview

Mobile applications are frequently used to collect and share data, such as taking photos and then sharing them via simple email or social media sites like Twitter or Facebook. This chapter discusses how to add social media sharing to your cross-platform application, as well as showing the Xamarin Component Store and how it can help to speed application development.

iOS and Android include a variety of system provided capabilities that applications can take advantage of to deliver consistent experiences. Both platforms provide APIs to access the camera to take photos, and even though Xamarin gives you complete access to those native APIs it can be time-consuming to learn two different ways to accomplish the same task. This is where cross-platform components like *Xamarin.Mobile* can help – it runs on all mobile platforms and provides a single, easy to learn API to accomplish a wide variety of common tasks.

Components can help speed up development in other ways too, such as user interface controls that can quickly and easily give your app a professional makeover.

In this chapter we will see how to add the following features to iOS and Android apps using Xamarin:

- Taking a photo
- Image filtering
- Access to social networks
- Address book
- Email
- Adding User interface Enhancements

Some of the sample projects include components from the *Xamarin Component Store*. The components used in this chapter are:

- Xamarin.Mobile
- Xamarin.Social (includes Xamarin.Auth)
- Satellite Menu (iOS only)

By the end of the chapter you'll have added a lot of functionality to iOS and Android applications, and seen how Xamarin lets you use components and common code to work faster while at the same time still taking advantage of platform-specific features when that is appropriate.

The code samples are structured to incrementally add functionality as you progress through the chapter. There are five sample projects included in the solution:

- iOS_demo1 – Taking or picking photos and applying a filter
- iOS_demo2 – Email & Twitter integration
- iOS_demo3 – Adding a UI Component

- **Android_demo1** – Taking photos and applying a filter
- **Android_demo2** –Email and Twitter integration, and extending the UI

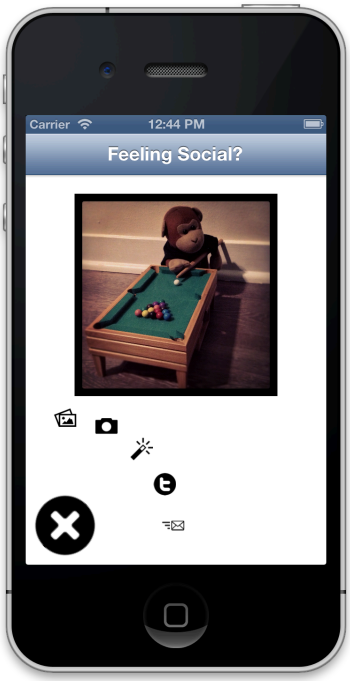
The code introduced in this chapter is already present in the samples, but commented out. Use the **Tasks** window in your IDE to quickly find the commented-out code that is marked with a `//TODO:` task identifier.

iOS

We're going to build an application that:

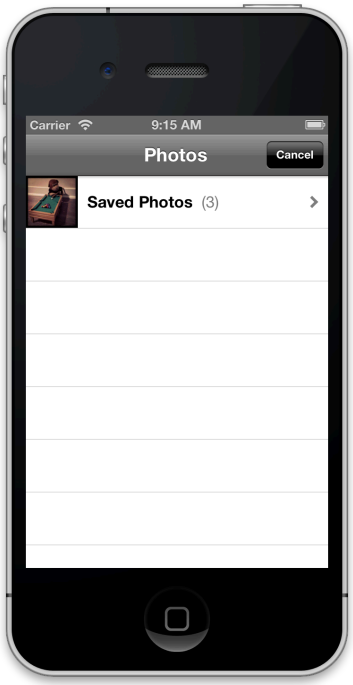
- Select a photo from the Camera Roll
- Take a photo
- Apply a 'sepia' image filter
- Post to Twitter
- Send an email attachment and access the address book
- Use the Satellite Menu component

The final app is shown below:



Selecting a Photo from the Photo Library

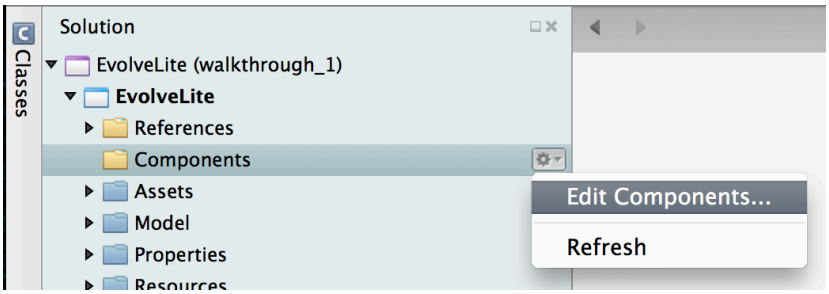
iOS provides support for selecting photos from the device photo library via a class called `UIImagePickerController`. This class provides a system view that any application can use to select photos, as shown below:



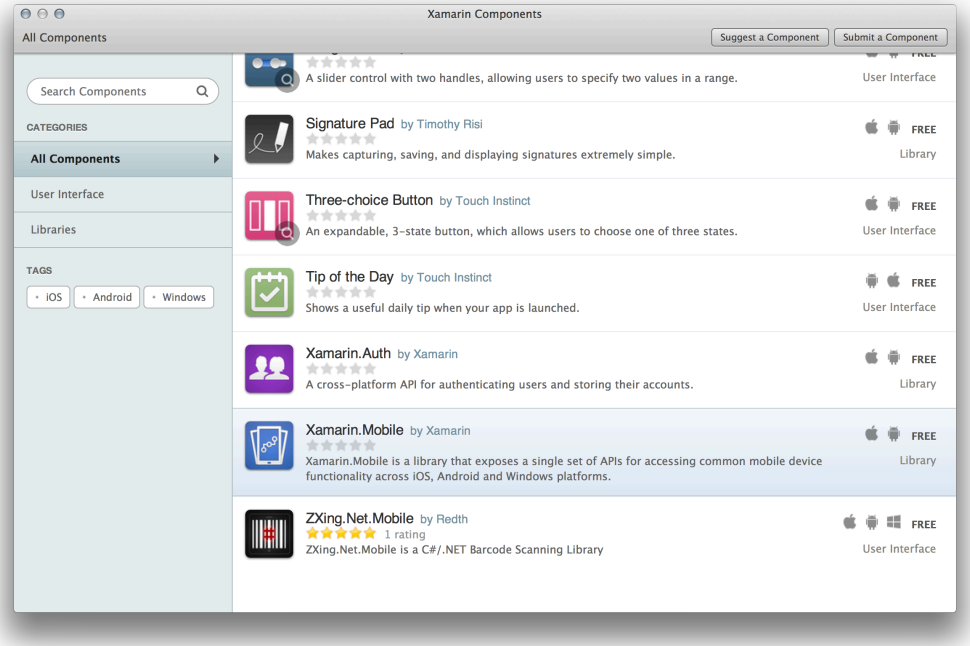
To make such code reusable across platforms, the Xamarin.Mobile API includes an abstraction of the `UIImagePickerController`, which we will use here to add a photo selection feature to the sample application.

Adding Xamarin.Mobile to the Project

Adding Xamarin.Mobile to our project is easy, thanks to the Xamarin Component Store. You will see the **Components** folder in your project's the solution view. You can right click (or click on the **gear) to show** a popup menu to access the component screen, as shown in this screenshot:



Select **Edit Components**, and the Xamarin Component Manager will appear. Scroll down the list until you see the **Xamarin.Mobile** component:



Clicking on Xamarin.Mobile will bring up some information on the component, as shown in the next screenshot:

Click on the green **Add to App** button, and the component will be downloaded and added to the project. We can now write the cross-platform code to choose or take a photo using **Xamarin.Mobile**.

Adding a UIImageView

First, we need a `UIImage` variable to store the selected photo in, as well as a `UIImageView` to display it. Start with the **iOS_demo1** project, and add the following variables to the `PhotosViewController` class included in the project:

```
public class SharedResourcesController : UIViewController
{
    UIImageView imageView;
    UIImage image;
```

To add the `imageView` to the view screen, we create and add it to the view hierarchy in the controller's `ViewDidLoad` method as follows:

```
imageView = new UIImageView {
    Frame = new RectangleF (40 , 20, 240, 240),
    ContentMode = UIViewContentMode.ScaleAspectFit
};

View.AddSubview (imageView);
```

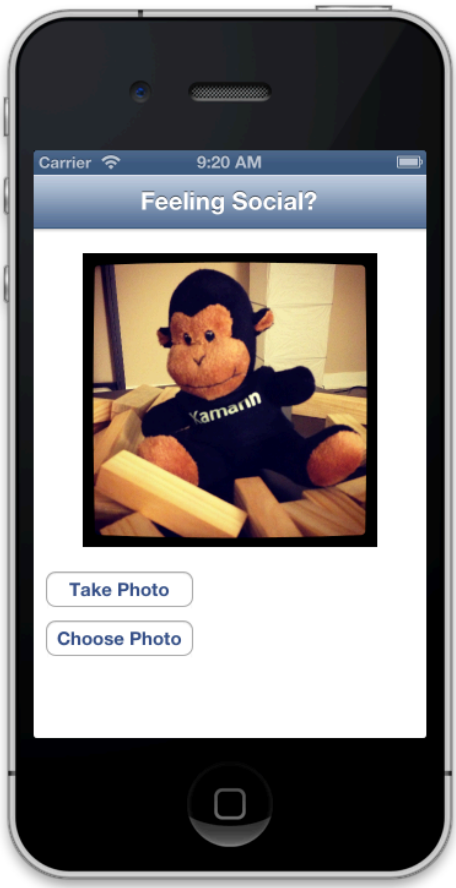
Using the MediaPicker Class

We can select a photo using Xamarin.Mobile with the `MediaPicker` class. In the `ChoosePhoto` method included in the starter project, add the following code:

```
var picker = new MediaPicker ();

picker.PickPhotoAsync ().ContinueWith (t => {
    if (t.IsCanceled)
        return;

    InvokeOnMainThread (delegate {
        image = UIImage.FromFile (t.Result.Path);
        imageView.Image = image;
    });
});
```

The `PickPhotoAsync` method opens the system media selection user interface, which on iOS is the `UIImagePickerController`. The result of the user's selection is returned in a `Task<MediaFile>` from which the image's path is obtained. Once we have the path, we use it to create a `UIImage` instance that is set to the `imageView.Image` property for display.

The result of selecting an image is shown below:



Taking a Photo with the Camera

In addition to selecting a photo, the `UIImagePickerController` also provides a user interface with access to the device cameras. Xamarin.Mobile's `MediaPicker` class abstracts this capability as well. Let's add support for taking a photo to our app.

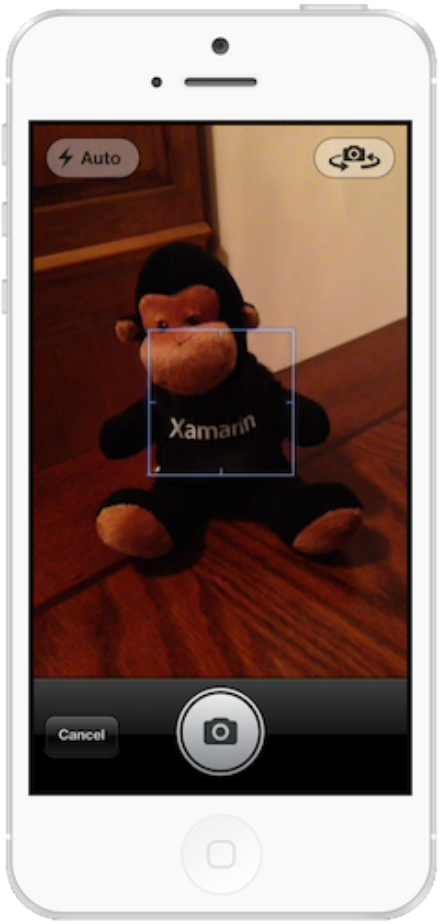
In the `TakePhoto` method, add the following code:

```
var picker = new MediaPicker ();

if (!picker.IsCameraAvailable || !picker.PhotosSupported) {
    new UIAlertView ("Camera Unavailable", "This device does not have a
camera", null, "OK").Show ();
    return;
}

picker.TakePhotoAsync (new StoreCameraMediaOptions{
    Name = "temp.png",
    Directory = "temp"
}).ContinueWith (t => {

if (t.IsCanceled)
    return;
```

```
            InvokeOnMainThread (delegate {
                image = UIImage.FromFile (t.Result.Path);
                imageView.Image = image;
            });
        });
```

After creating a `MediaPicker` instance we first check that a camera is available and that taking photos is supported. Then we call `TakePhotoAsync` to launch the camera. On the simulator, where no cameras are available, the following alert is displayed:

However, on a device, the camera interface is launched:



After taking a photo, it is saved to the file specified in `StoreCameraMediaOptions` passed to the `TakePhotoAsync` method. The code to then create the image and display it in the `UIImageView` is identical to the photo selection example shown earlier.

Filtering an Image

iOS includes support in the Core Image framework for filtering images. To demonstrate using Core Image, let's add a sepia filter to the image in our application. This is a good example of taking advantage of native platform features – later we'll see how to accomplish the same effect using Android-specific APIs.

In the `ApplyFilter` method, add the following code:

```
if (image != null) {
    var ciImage = new CIImage (image.CorrectOrientation ());

    var sepia = new CISepiaTone ();
    sepia.Image = ciImage;
    sepia.Intensity = 1.0f;

    var ctx = CIContext.FromOptions (null);
    var output = sepia.OutputImage;
    var cgImage = ctx.CreateCGImage (output, output.Extent);
```

```
        image = UIImage.FromImage (cgImage);
        imageView.Image = image;
    } else {
        new UIAlertView ("Image Unavailable", "Please select an image or take
    one with the camera", null, "OK").Show ();
    }
```

To use Core Image, we create a `CIImage` instance from the `UIImage` containing the photo. Photos contain orientation data in their metadata, which `UIImageView` uses to display the photo so that it appears upright. However, the actual image will not be upright when taken with the camera. In order for the image to not appear rotated after filtering, we apply the following extension method before creating a `CIImage`:

```
    public static UIImage CorrectOrientation (this UIImage image)
    {
        if (image.Orientation == UIImageOrientation.Up)
            return image;

        UIGraphics.BeginImageContextWithOptions (image.Size, false,
    image.CurrentScale);
        image.Draw (new RectangleF (0, 0, image.Size.Width,
    image.Size.Height));
        UIImage img = UIGraphics.GetImageFromCurrentImageContext ();
        UIGraphics.EndImageContext ();
        return img;
    }
```

Core Image filters provide a recipe of how to filter an image. In this example, we create a sepia filter instance, passing it the `CIImage` instance to filter. A `CIContext` instance orchestrates the actual image filtering process. We use the `CIContext` to create a `CGImage` from output of the filter. A `CGImage` is a Core Graphics image, with Core Graphics being a lower level drawing framework that UIKit classes such as `UIImage` abstract.

Finally, we call the `UIImage.FromImage` method to create a `UIImage` from the `CGImage`, which we then assign to the `imageView.Image` for display. This results in the image with a sepia tone applied, as shown below:

Sending a Tweet

iOS includes built-in support for connecting to *Twitter*, *Facebook* and *Sina Weibo*. The **iOS_demo2** sample project shows how to connect to Twitter to send a tweet with an image. Posting to Facebook or Sina Weibo uses the same basic principles, although Sina Weibo has the added restriction that a Chinese input method must be enabled on the user's device before a Sina Weibo account can be added in **Settings**.

Using SLComposeViewController

The Social Framework includes a controller called `SLComposeViewController` that presents a system provided view for editing and sending a tweet, as shown below:

To send a tweet including the image, add the following code to the `SendTweet` method provided in the sample:

```
if (SLComposeViewController.IsAvailable (SLServiceKind.Twitter)) {
    slComposer = SLComposeViewController.FromService
(SLServiceType.Twitter);
    slComposer.SetInitialText ("Hello from #Evolve2013");
    if (image != null)
        slComposer.AddImage (image);
        slComposer.CompletionHandler += (result) => {
            InvokeOnMainThread (() => {
            DismissViewController (true, null);
            new UIAlertView ("Tweet Result", result.ToString (), null,
"OK").Show ();
        });
    };
    PresentViewController (slComposer, true, null);
}
```

First we check if the Twitter service is available, after which we create an instance of the `SLComposeViewController` for the Twitter service. Once we have this instance, we set its text and image with the `SetInitialText` and `AddImage` methods respectively.

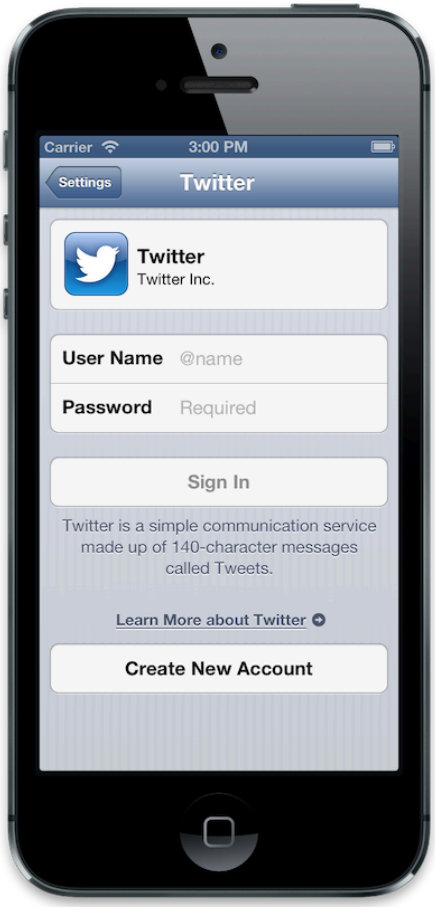
When the tweet operation is complete, we provide a `CompletionHandler` to close the controller and show the result in an alert.

Finally to actually display the controller's view, we call `PresentViewController`.

Adding a Twitter Account

In order to connect to Twitter, an account needs to be added to the device settings. If the `SLComposeViewController` is presented without an account being available, the controller will display an alert notifying the user as shown below:

Clicking on **Settings** takes the user to the Twitter account settings:



Once the account is in place using the `SLComposeViewController` will post to Twitter as expected.

Sending Email to a Contact

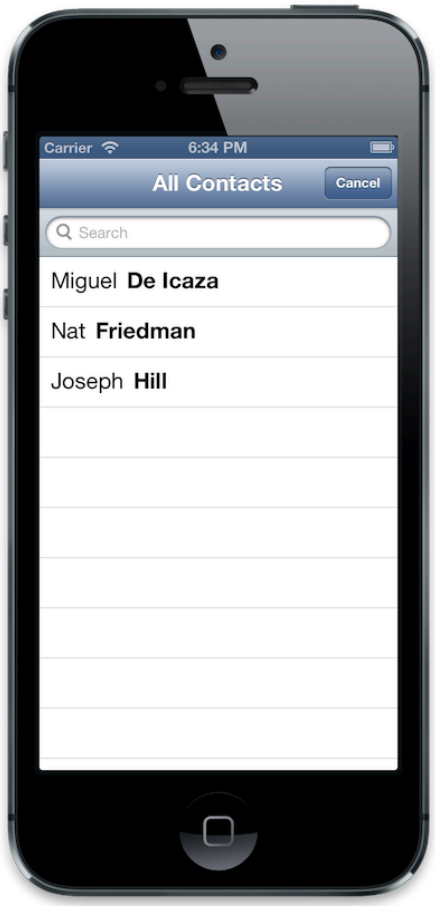
iOS includes systems provided controllers for accessing contacts and sending email through the `ABPeoplePickerNavigationController` and `MFMailComposeViewController` classes respectively. Let's add a feature to our application that allows the user to select a contact and then send an email with an image to the selected contact's address.

In the `PickContactSendEmail` method provided in the sample code, add the following implementation:

```
peoplePicker = new ABPeoplePickerNavigationController ();

peoplePicker.Cancelled += (sender, e) =>
{
    DismissViewController (true, null);
};

peoplePicker.SelectPerson += delegate(object sender,
ABPeoplePickerSelectPersonEventArgs e) {
```

```
            string emailAddr = "";

            var person = e.Person;
            var emails = person.GetEmails ();

            if (emails.Count > 0) {
                emailAddr = emails [0].Value;
            }

            DismissViewController (true, () => {
                SendEmail (emailAddr);
            });
        };

        PresentViewController (peoplePicker, true, null);
```

This code presents an `ABPeoplePickerNavigationController` from which the user can select a contact as shown below:



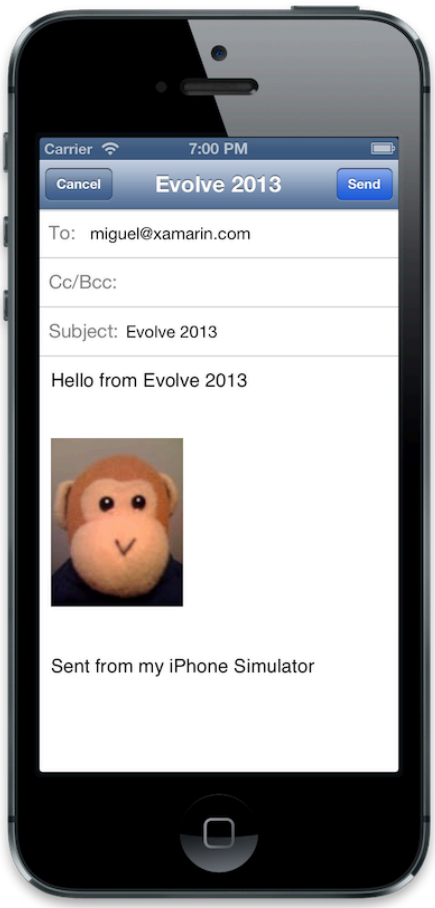
We handle the `SelectContact` event to retrieve the first email address of the selected contact, which we use to send an email in the `SendEmail` method. We call `SendEmail` to load the `MFMailComposeViewController` after the `ABPeoplePickerNavigationController` is dismissed.

Within the `SendEmail` method, add the following code:

```
mailComposer = new MFMailComposeViewController ();
mailComposer.SetToRecipients (new string[]{address});
mailComposer.SetSubject ("Evolve 2013");
mailComposer.SetMessageBody ("Hello from Evolve 2013", false);
mailComposer.AddAttachmentData (image.CorrectOrientation().AsPNG (),
"image/png", "PhotoFromEvolve");
mailComposer.Finished += (sender, e) => {
    DismissViewController (true, null);
};
PresentViewController (mailComposer, true, null);
```

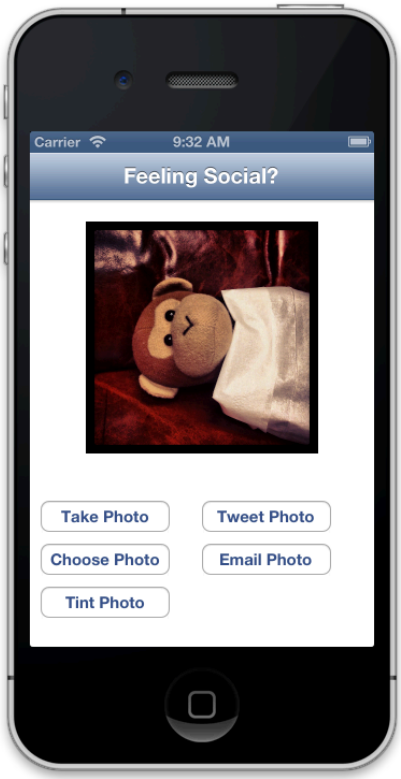
After creating the `MFMailComposeViewController` instance, we populate it with data needed to send an email, including the email address of the contact selected in the previous section, the subject, message body, and the image as an attachment.

We then show the `mailComposer` using the `PresentViewController` method and dismiss it when the `Finished` event is raised. The following screenshot shows the view of the populated `mailComposer`:

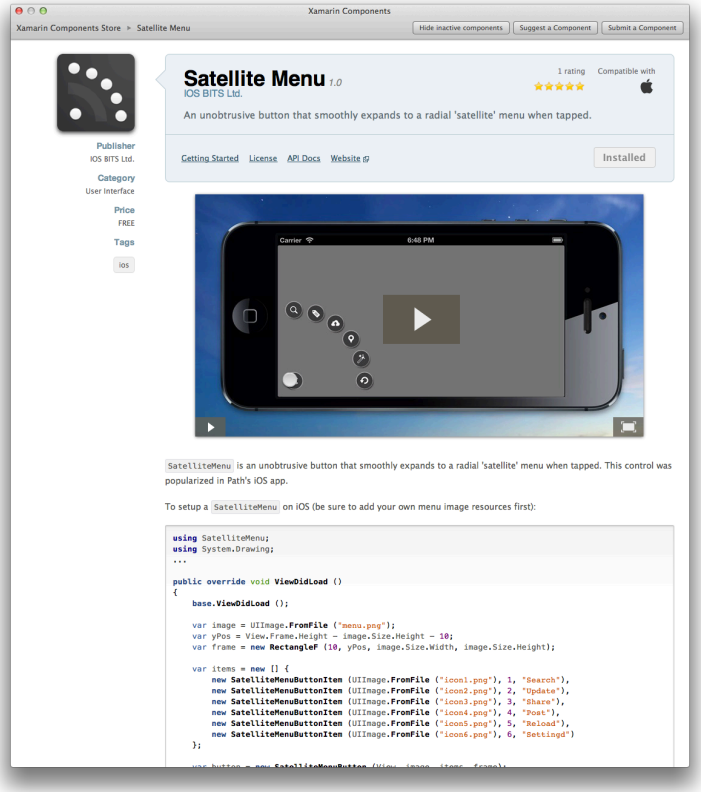


Adding a User Interface Component

Until now we have been adding simple buttons to trigger the features of our sample app, and the screen looks like this:



In the **iOS_demo3** project we are going to delete these buttons and add the Satellite Menu from the Xamarin Component Store. Double-click the **Components** folder in the project and choose **Get More Components** and find the **Satellite Menu**. Click the **Add to App** button and the component will automatically be added to the project.



The code to implement the menu is provided in its documentation – simply copy it and modify to match our application's features (set the images and the methods call for each menu item).

```
var menuImage = UIImage.FromFile ("images/menu.png");
var y = View.Frame.Height - menuImage.Size.Height - 10;
var frame = new RectangleF (10, y, menuImage.Size.Width,
menuImage.Size.Height);


var items = new [] {
    new SatelliteMenuButtonItem (UIImage.FromFile ("images/photos.png"), 1,
"Choose Photo"),
    new SatelliteMenuButtonItem (UIImage.FromFile ("images/camera.png"), 2,
"Take Photo"),
    new SatelliteMenuButtonItem (UIImage.FromFile ("images/filter.png"), 3,
"Filter"),
    new SatelliteMenuButtonItem (UIImage.FromFile ("images/twitter.png"),
4, "Twitter"),
    new SatelliteMenuButtonItem (UIImage.FromFile ("images/email.png"), 5,
"Email")
};


var menu = new SatelliteMenuButton (View, menuImage, items, frame){
    Radius = 115
};


menu.MenuItemClick += (_, args) => {
    Console.WriteLine ("{0} was clicked!", args.MenuItem.Name);
    switch (args.MenuItem.Tag) {
```

```
            case 1:
                ChoosePhoto ();
                break;
            case 2:
                TakePhoto ();
                break;
            case 3:
                ApplyFilter ();
                break;
            case 4:
                SendTweet ();
                break;
            case 5:
                PickContactSendEmail ();
                break;
            }
        };
        View.AddSubview (menu);
```

After adding the component the app interface looks like this – the buttons are now replaced by a cool user interface element with only a few lines of code:

Android

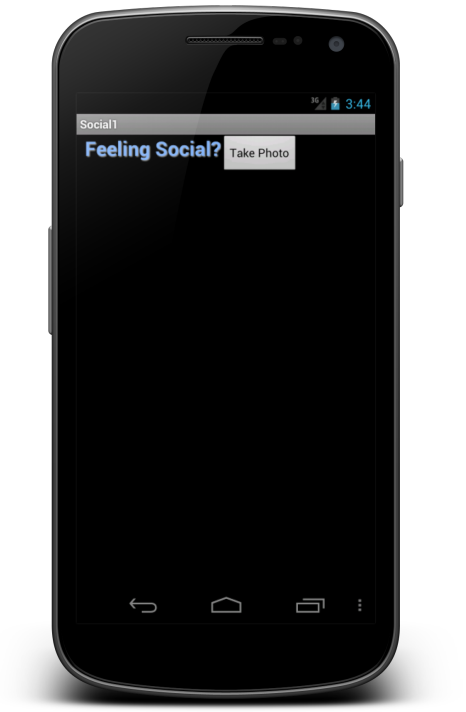
Android includes a variety of system provided capabilities that applications can take advantage of to enhance their functionality, and Xamarin provides both

complete access to the native APIs as well components that let you write cross platform code to save time and add functionality. In this chapter we will walk through how to include several of these in an application including:

- Take a picture with the camera.
- Applying a filter to an image.
- Creating Context and Options menus.
- Using Xamarin.Social to access social networks, such as Twitter.
- Sending Email with attachments.

In general the Android sample follows the same structure as the iOS sample, however there is a key difference in that it uses an Adapter to display images with a `GridView` widget. The `GridView` is a widget that shows items in a two-dimensional scrolling grid. Refer to the separate Advanced training chapter on the iOS Collection View for a similar control on that platform.

We're going to start with the Android_demo1 project; it should appear similar to the following screenshot before we start:



First we're going to look at making the **Take Photo** button work!

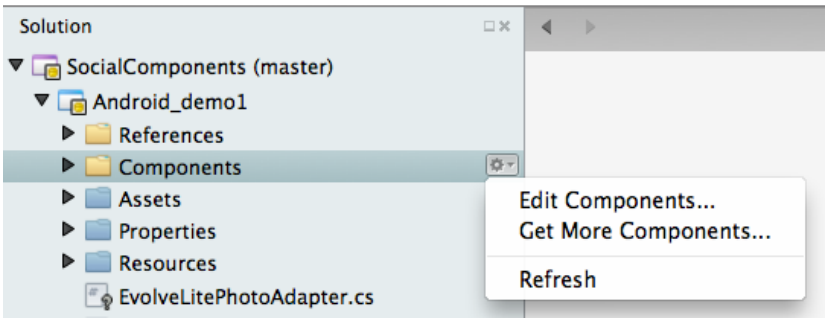
Photos Made Easy

It is easy to add picture-taking capabilities to an Android application using the existing Android API's inside the `Android.Hardware`. However, these API's are not

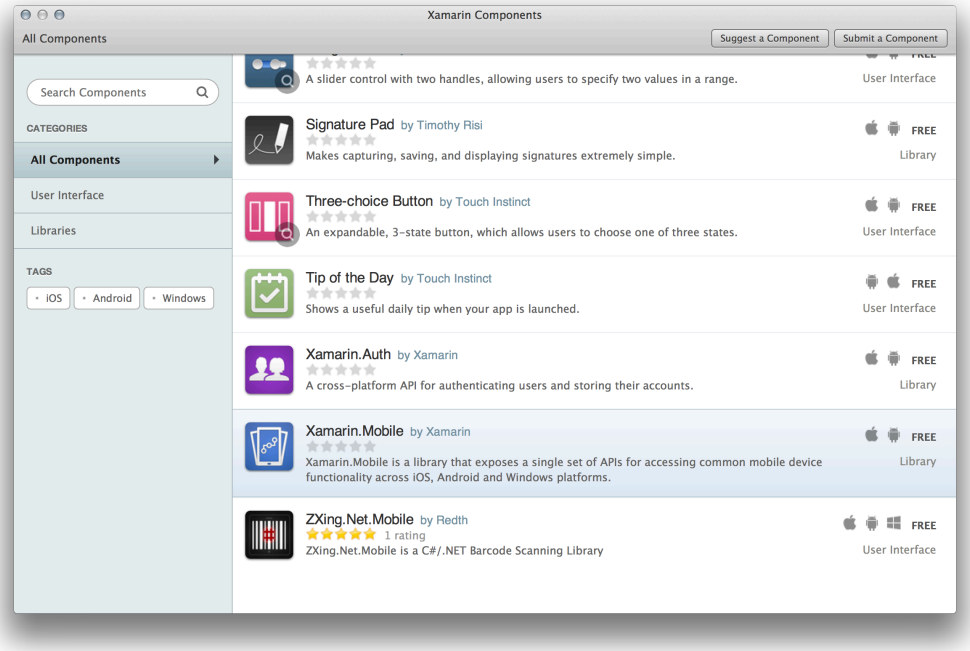
cross-platform. To address this issue, the *Xamarin.Mobile* component includes a cross-platform abstraction that makes is easy. Lets first see about adding *Xamarin.Mobile* to our application, and then go about using it to take a picture.

Adding Xamarin.Mobile to the Project

Adding Xamarin.Mobile to our project is easy, thanks to the Xamarin Components. If you look at the project in Xamarin.Studio, you will see the **Components** folder in the solution view. By clicking on the **gear**, a popup menu will appear, as show in the following screenshot:

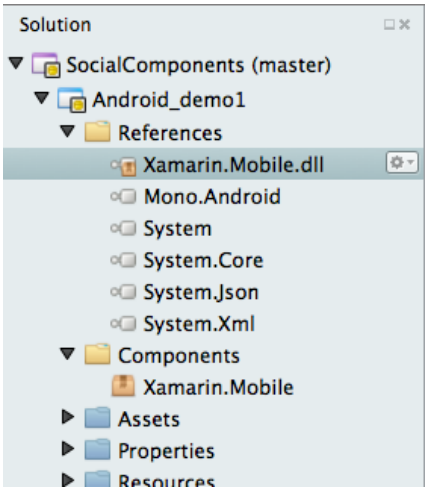


Select **Edit Components**, and the Xamarin Component Manager will appear. Scroll down the list until you see the **Xamarin.Mobile** component:



Clicking on **Xamarin.Mobile** will bring up some information on the component, as shown in the next screenshot:

Click on the green **Add to App** button, and the component will be downloaded and added to the project. The following screenshot shows how there is now a reference to the **Xamarin.Mobile.dll**:



Now that we have **Xamarin.Mobile** added to our project, lets modify the project so that it can take a picture.

Taking a Photo with the Camera

Xamarin.Mobile contains the `MediaPicker` class that handles the platform specific details of taking a picture. `MediaPicker` providers the following key methods:

- **IsCameraAvailable** – This property will tell us if the device in question has a camera or not.

- **TakePhotoAsync** – This method will call the system camera to take a picture. The return value from this method is a `Task<MediaFile>` instance. We were introduced to `Task<T>` from the Task Parallel Library in the Fundamentals chapter on Web Services.

Let's take care of the first thing – take a picture with our application.

1. Open the class `PhotoActivity`, and add in the following two methods:

```
private void InitiateTakeAPicture()
{
    var picker = new MediaPicker(this);
    if (!picker.IsCameraAvailable) {
        Toast.MakeText(this,
Resources.GetString(Resource.String.camera_not_available),
ToastLength.Short);
    }

    var fileName = string.Format("EvolveLite2013_{0:yyyyMMddHHmmss}.jpg",
DateTime.UtcNow);
    var mediaOptions = new StoreCameraMediaOptions {
        Directory = MediaDirectory,
        Name = fileName
    };

    picker.TakePhotoAsync(mediaOptions)
        .ContinueWith(PictureTaken,
TaskScheduler.FromCurrentSynchronizationContext());
}

private void PictureTaken(Task<MediaFile> task)
{
    if (task.IsCanceled || task.IsFaulted) {
        return;
    }
    var file = task.Result.Path;
    Log.Debug(GetType().FullName, "Picture saved to {0}.", file);
    adapter.AddFile(file);
}
```

The first method, `InitiateTakeAPicture` will create a new instance of `MediaPicker` and check that a camera is available. By default, `MediaPicker` will save pictures to the directory specified by `Android.OS.Enviroment.GetExternalFilesDir()`. It is not possible to specify a different directory with Xamarin.Mobile, but it is possible to specify a subdirectory for the images. In this example the application will provide a name of a subdirectory to store pictures in. The name of the picture and the directory to keep files stored in will be provided by an instance of `StoreCameraMediaOptions`.

To take a picture with the system camera we invoke `TakePhotoAsync` and pass it an instance of `StoreCameraMediaOptions`. `TakeAPhotoAsync` runs asynchronously so that it does not block the UI. `TakeAPictureAsync` returns a `Task<MediaFile>`, making it possible to exploit the benefits of the Task Parallel Library (we touched on the Task Parallel Library in the previous chapter on Web Services).

In this example we provide a continuation that will be executed when `MediaPicker` has finished. This continuation takes two parameters:

- An Action, which can be a lambda, delegate, or method that accepts a `Task<MediaFile>`.
- An optional `TaskScheduler` that tells the TPL on what thread to run this continuation.

The method `PictureTaken` will be called after `MediaPicker` is done with the system camera. The call to `TaskScheduler.FromCurrentSynchronizationContext()` will provide a synchronization context that ensures the continuation is executed on the UI thread. In this case our example will simply notify the Adapter that a new picture was taken so that it will be properly displayed in the UI.

2. At this point we have the code in place to take a picture, but no way for the user to actually initiate the process.

```
photoButton.Click += (sender, e) => {
    TakePhoto();
};
```

This wires up the **Take a Photo** button. If you run the application, it should now be possible to take a picture and have it show up in the **Photos** tab.

One of the limitations of mobile devices is limited memory available to each application. A common problem with display many images simultaneously is that the application will quickly exhaust the RAM available to it and crash. To get around this problem we shall have our application generate a thumbnail image that will be display instead of the full sized image. If you take a look in the solution, you will see that there is a class called `ImageResizer` that will do the resizing for us.

3. Let us update `PhotosActivity` so that a thumbnail will be created immediately after a photo is taken. Add the following instance variable to the class PhotosActivity:

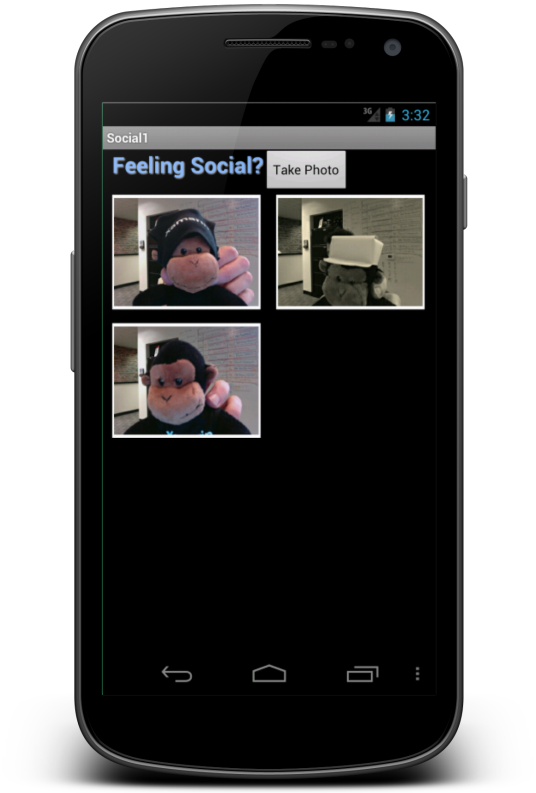
```
static readonly ImageResizer imageResizer = new ImageResizer();
```

4. Next, change the callback method `PictureTaken` so that it will create the thumbnail of the original image, and save it on the device for us:

```
private void PictureTaken(Task<MediaFile> task)
{
    if (task.IsCanceled || task.IsFaulted) {
        return;
    }
    var file = task.Result.Path;
    Log.Debug(GetType().FullName, "Picture saved to {0}.", file);
    var thumbnail = imageResizer.CreateThumbnailFile(file, 220, 220);
    photoAadapter.AddFile(thumbnail);
}
```

After a thumbnail is created, we add its filename to the adapter instead of the original filename. Now

5. If you now run the application and take a picture, it should look something like the following screenshot:



Congratulations! You've now integrated a camera within your application and have learned how to leverage ever-growing library of Xamarin Components to speed up development.

Filtering an Image

Now that the application is taking pictures and storing them for the user, let's get try something fancy with the pictures – applying a sepia filter to an individual picture. Our application will present a context menu for a photo. When the user selects the **Apply Sepia** option from the context menu, the application will manipulate the image (and it's thumbnail) and apply a sepia filter. The class `SepiaFilter` already contains the necessary code to apply a sepia filter to an image, so lets change the application to use this class.

1. The first thing we need is a new menu resource. Go ahead and add an XML file called `Resources/menu/photo_context_menu.xml`, which the contents shown below:

```
<?xml version="1.0" encoding="utf-8" ?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
  <item android:id="@+id/menu_apply_filter"
```

```
        android:title="@string/menu_apply_filter" />
</menu>
```

2. Associate the `GridView` with the context menu by calling `RegisterForContextMenu` in the `OnCreate` method of `PhotosActivity` as shown in the following code snippet:

```
gridView = FindViewById<GridView>(Resource.Id.gridView);
RegisterForContextMenu(_gridView);
```

3. Add the following method to `PhotosActivity` so that clicking on an image applies the sepia filter:

```
gridView.ItemClick += (sender, e) => {
    var thumbnailFile = photoAdapter.GetFileName(e.Position);
    var photoFile = thumbnailFile.Replace("_thumbnail", String.Empty);
    ApplySepiaFilterToPhoto(thumbnailFile, photoFile);
    Toast.MakeText(this,
Resources.GetString(Resource.String.apply_sepia_filter_done),
ToastLength.Short);
};
```

This method will be invoked when the Activity must create a context menu. It will create a floating context menu when the user long-clicks on a view that declares support for a context menu.

4. Next implement `OnContextItemSelected`. This method will be called when the user selects an item from the context menu. The following code snippet shows the code that should be in this method:

```
public override bool OnContextItemSelected(IMenuItem item)
{
    var info = (AdapterView.AdapterContextMenuInfo)item.MenuInfo;
    var thumbnailFile = _adapter.GetFileName(info.Position);
    var photoFile = thumbnailFile.Replace("_thumbnail", String.Empty);

    switch (item.ItemId) {
        case Resource.Id.menu_apply_filter:
            ApplySepiaFilterToPhoto(thumbnailFile, photoFile);
            return true;
        default:
            return base.OnContextItemSelected(item);
    }
}
```

This method should always return true once a context menu has been successfully handled. If the activity cannot handle the selected menu item, it should call `OnContextItemSelected` from the base class.

We get data about the selected image from the MenuInfo property – in this case we are interested in the position of the image. In this example we figure out the name of the original photo and then thumbnail and pass that on to the method `ApplySepiaFilterToPhoto`.

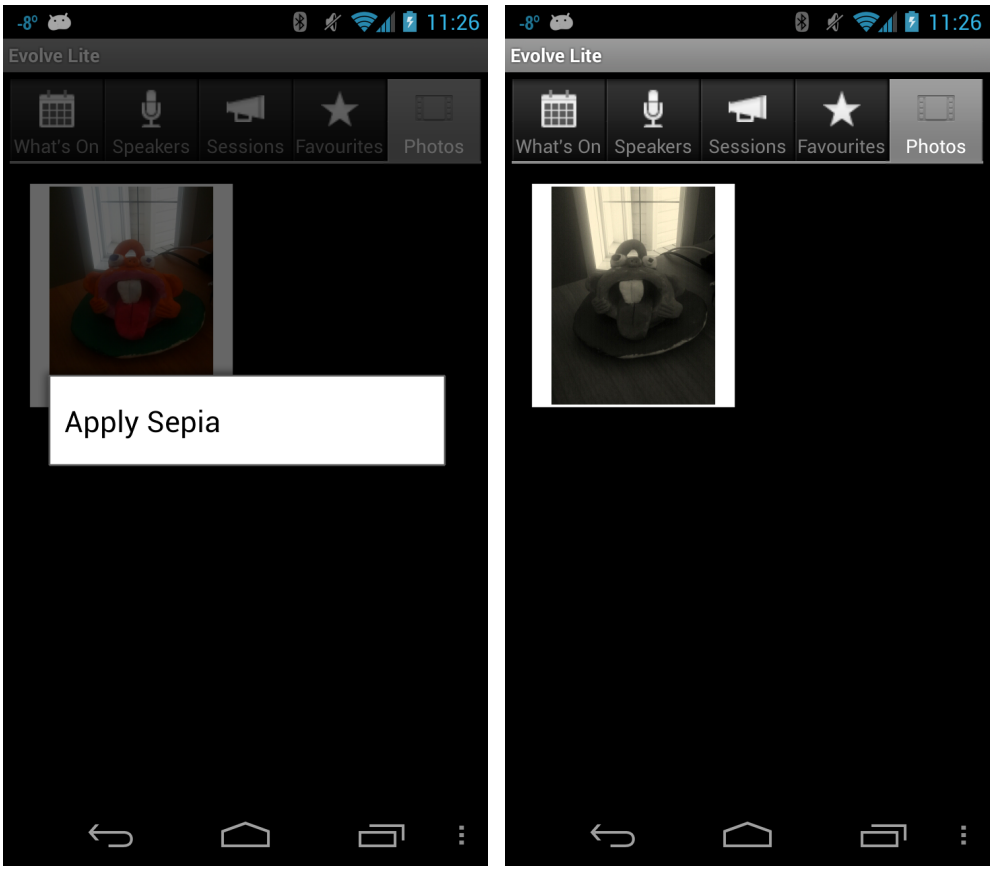
5. Next add the following method to `PhotosActivity`:

```
private void ApplySepiaFilterToPhoto
(string thumbnailFile, string photoFile)
{
```

```
        var tasks = new Task[2];
        tasks[0] = new Task(() =>
                            {
            var filter = new SepiaFilter();
            filter.ApplyToFile(thumbnailFile);

        });
        tasks[1] = new Task(() =>
                            {
            var filter = new SepiaFilter();
            filter.ApplyToFile(photoFile);

        });

        for (var i = 0; i < tasks.Length; i++)
        {
            tasks[i].Start();
        }
        Task.WaitAll(tasks);
        photoAdapter.NotifyDataSetChanged();
    }
```

In this method executes two instances `SepiaFilter` in parallel: one instance will deal with the thumbnail file, and the other instance deals with the original image. The method `NotifyDataSetChanged` is invoked after both tasks are finished to notify the GridView that it should refresh itself because it's underlying data source has been changed.

6. Run the application. Take a picture and select **Apply Sepia** from the context menu. Your application should look something like the following two screenshots:



Congratulations! At this point in our tutorials you have added picture-taking capabilities to the sample application and the ability to do some image manipulation. You have also gained some familiarity with the powerful Task Parallel Library and have seen how to execute tasks in parallel and in the background, keeping your application responsive to users.

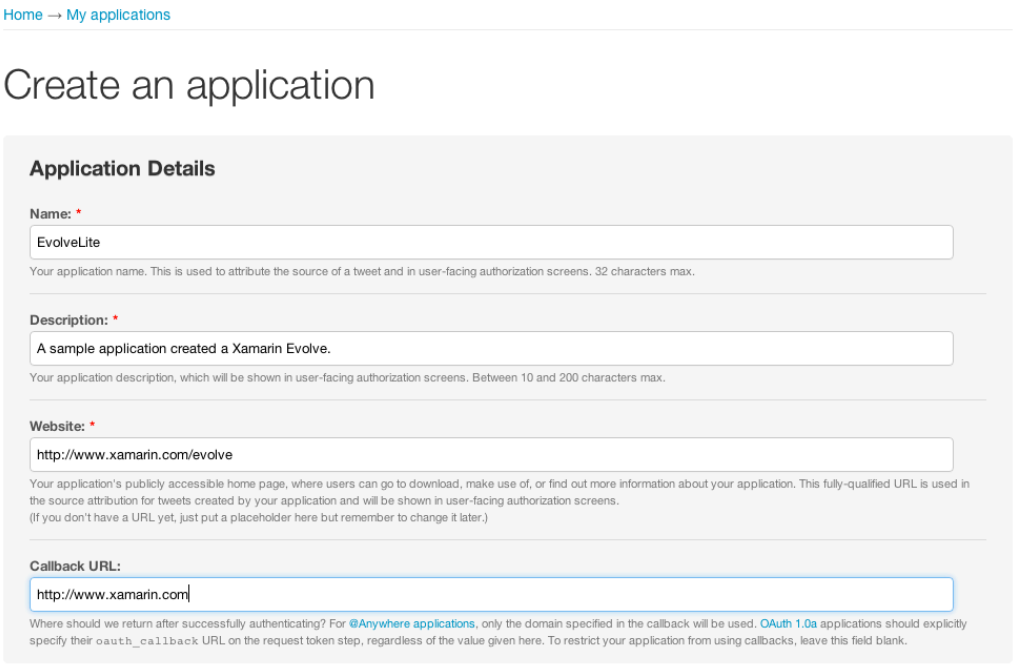
Integrating Social Networks

People love social apps on their mobile devices – sharing photos being one of the more common and popular social features. Let's see how Xamarin makes it easy to integrate social networks such as Twitter into a mobile application.

Xamarin.Social is a component available from the Xamarin Component Store that greatly simplifies using social services such as Twitter, Facebook, and Flickr. Rather than writing all the code to use Twitter ourselves, lets see how we can take advantage of Xamarin.Social.

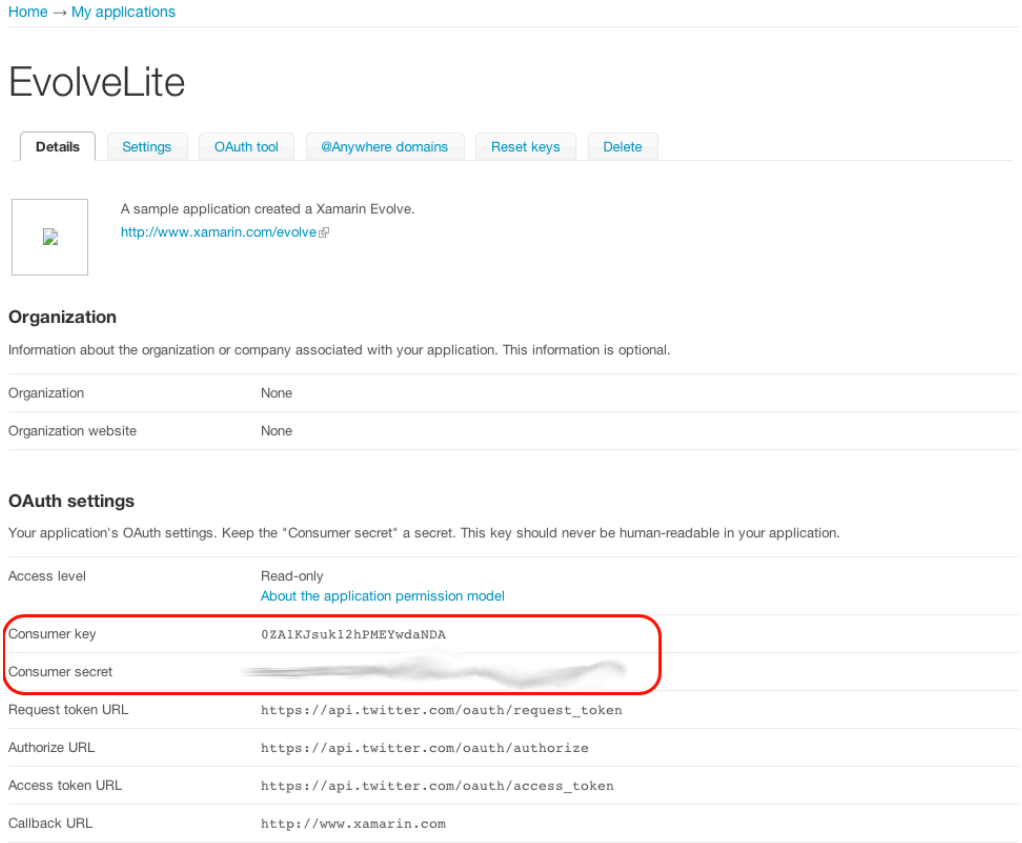
Xamarin.Social provides the class `Xamarin.Social.Services.TwitterService` performs much of the work required by a mobile application before it can interact with Twitter on behalf of a user. `TwitterService` provides a clean API that handles OAuth, posting tweets, and sending attachments.

1. The first thing we need to do is to register the application with Twitter. We do this by logging on to the Twitter Developers website with our Twitter account, and navigate to the My applications page. Create a new application by clicking on the Create a new application button in the upper left hand corner of the page, as shown in the following screenshot:



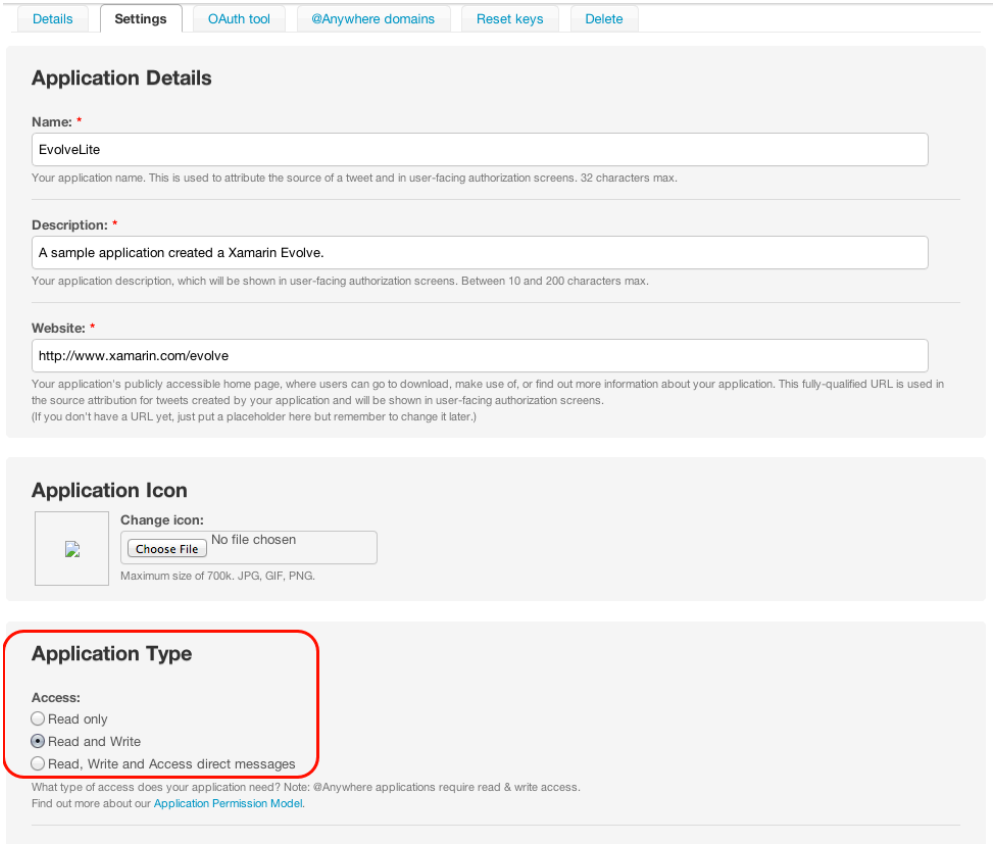
You need to provide a Callback URL. This can be any valid URL - in this case we used http://www.xamarin.com.

2. On this create an application page you must enter some metadata about the application, agree to the **Developer Rules of the Road**, and click **Okay**.
3. Once you have provided that information, scroll down to the OAuth settings section on the information page for your application. Notice in the screen shot below that Twitter generated a **Consumer key** and **Consumer secret** values.



Protect the **Consumer secret** value because it is unique to your application. Both of these values will be necessary when we want to interact with Twitter.

4. The final part of registration is changing the settings so that our application may post to Twitter on behalf of the user. Click on the settings tab, and change the **Access** on the **Application Type** section to Read and Write as shown in the following screenshot:



5. Next add the Xamarin.Social component to your application using the Component Manager.
6. Our activity needs an instance of Xamarin.Social.Services.TwitterService, so lets add this to the activity. Declare the following instance variable in `PhotosActivity`:

```
private readonly  Service _twitterService = new TwitterService
{
    ConsumerKey = "YOUR CONSUMER KEY",
    ConsumerSecret = "YOUR CONSUMER SECRET"
};
```

Notice that we do not have to specify the Callback URL when we instantiate the Twitter service.

7. In order to share a picture via Twitter, for now we'll change the `ItemClick` behavior so that it posts the photo to Twitter instead of applying a filter:

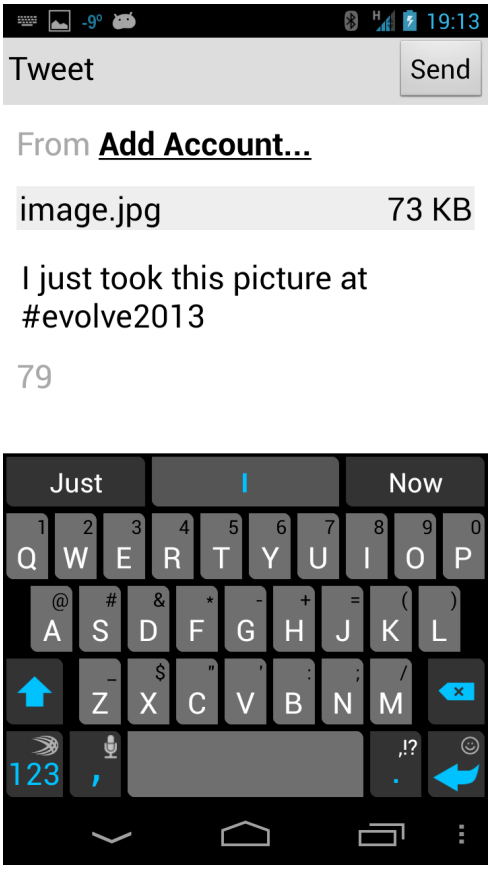
```
void SharePictureOnTwitter(String pathToPhoto)
{
    var photo = BitmapFactory.DecodeFile(pathToPhoto);
    var item = new Xamarin.Social.Item("Now I'm sharing things.
#testing");
    var imageData = new ImageData(photo);
    item.Images.Add(imageData);
    var intent = twitterService.GetShareUI(this, item, shareResult =>{});
    StartActivity(intent);
}
```

8. In order to share a picture via Twitter, for now we'll change the `ItemClick` behavior so that it posts the photo to Twitter instead of applying a filter:

```
gridView.ItemClick += (sender, e) => {
    var thumbnailFile = photoAdapter.GetFileName(e.Position);
    SharePictureOnTwitter(thumbnailFile);
    return true;
}
```

In the code above we retrieve the thumbnail file when the user clicks on a picture. The tweet itself is represented by a `Xamarin.Social.Item` instance, which has the thumbnail image attached as a `Xamarin.Social.ImageData` object. We use the thumbnail image because Twitter has a size limit for attachments.

Next notice how the method asks TwitterService to create an Intent for our tweet. We use this intent to start an Activity that is provided by Xamarin.Social that will allow the user to select their Twitter account and change their message before the tweet is sent. The Activity is shown in the following screenshot:



The user simply needs to click **Add Account...** and follow the instructions. Once that process is complete they will be able to take photos and tweet them from the sample app!

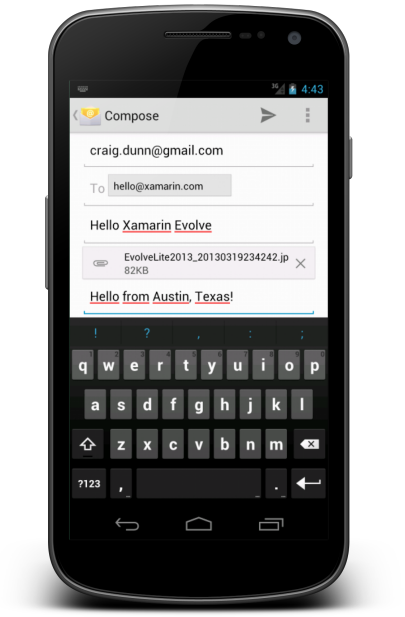
Sending an Email with Attachment

To send the photo by email instead of Twitter, add this method to the Activity:

```
void SharePictureViaEmail (String pathToPhoto)
{
    var email = new Intent (Android.Content.Intent.ActionSend);
    email.PutExtra (Android.Content.Intent.ExtraEmail, new
string[]{"hello@xamarin.com"} );
    email.PutExtra (Android.Content.Intent.ExtraSubject, "Hello Xamarin
Evolve");
    email.PutExtra (Android.Content.Intent.ExtraText, "Hello from Austin,
Texas!");
    email.PutExtra (Android.Content.Intent.ExtraStream,
Uri.Parse("file://"+pathToPhoto)); // attachment

    // Will let the user choose Messages or Email to handle intent
    //email.SetType("text/plain");
    // Will force Email to handle intent, if it has been configured
    email.SetType ("message/rfc822");
    StartActivity (email);
}
```

To wire up this method, simply use it in the `gridView.ItemClick` handler instead of the Twitter integration.

Sending an email in Android requires the user to have already set-up an email account. As long as they have already done so, they will be presented with an email input screen (with the photo already attached) like this:



Side Track: Adding Options and Context Menus

In this sample so far we have created methods to apply a sepia filter, post to twitter and send via email; but each time we have used clicking on the photo to trigger that action. To enable the user to choose any of these actions we need a

new user interface element – a context menu that presents all these options for each picture.

Let's change the user interface to use standard Android context menu to present these options.

Adding a Context Menu

1. The first thing we need is a new menu resource with entries for all the features we've built. Go ahead and add an XML file called `Resources/menu/photo_context_menu.xml`, with the contents shown below (don't forget to add entries to the `Strings.xml` file too):

```
<?xml version="1.0" encoding="utf-8" ?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
  <item android:id="@+id/menu_apply_filter"
        android:title="@string/menu_apply_filter" />
  <item android:id="@+id/menu_delete_picture"
        android:icon="@drawable/ic_menu_take_picture"
        android:title="@string/menu_delete_picture" />
  <item android:id="@+id/menu_tweet_picture"
      android:title="@string/menu_share_with_twitter" />
  <item android:id="@+id/menu_email_picture"
      android:title="@string/menu_share_with_email" />
</menu>
```

2. Associate the `GridView` with the context menu by calling `RegisterForContextMenu` in the `OnCreate` method of `PhotosActivity` as shown in the following code snippet:

```
gridView = FindViewById<GridView>(Resource.Id.gridView);
RegisterForContextMenu(gridView);
```

3. Override the OnCreateContextMenu item for the `PhotosActivity` class:

```
public override void OnCreateContextMenu(IContextMenu menu, View v,
IContextMenuContextMenuInfo menuInfo)
{
    base.OnCreateContextMenu(menu, v, menuInfo);
    MenuInflater.Inflate(Resource.Menu.photo_context_menu, menu);
}
```

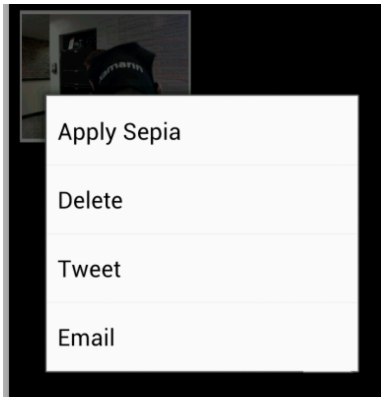
This method will be invoked when the Activity must create a context menu. It will create a floating context menu when the user long-clicks on a view that declares support for a context menu.

4. Next implement `OnContextItemSelected`. This method will be called when the user selects an item from the context menu. The following code snippet shows the code that should be in this method:

```
public override bool OnContextItemSelected(IMenuItem item)
{
    var info = (AdapterView.AdapterContextMenuInfo)item.MenuInfo;
    var thumbnailFile = photoAdapter.GetFileName(info.Position);
    var photoFile = thumbnailFile.Replace("_thumbnail", String.Empty);

    switch (item.ItemId)
    {
```

```
            case Resource.Id.menu_tweet_picture:
                SharePictureOnTwitter(thumbnailFile);
                return true;

            case Resource.Id.menu_email_picture:
                SharePictureViaEmail(photoFile);
                return true;

            case Resource.Id.menu_apply_filter:
                ApplySepiaFilterToPhoto(thumbnailFile, photoFile);
                return true;

            case Resource.Id.menu_delete_picture:
                Task.Factory.StartNew(() =>
                  {
                      // Delete the picture and the thumbnail, but
                      // don't do it on the UI thread.
                      File.Delete(thumbnailFile);
                      File.Delete(photoFile);
                      Log.Debug(GetType().FullName, "Deleted files {0} and
{1}.", thumbnailFile, photoFile);
                  });

                photoAdapter.RemoveFile(thumbnailFile);
                return true;

            default:
                return base.OnContextItemSelected(item);
        }
    }
```

This method should always return true once a context menu has been successfully handled. If the activity cannot handle the selected menu item, it should call `OnContextItemSelected` from the base class. The finished context menu will now appear when you perform a long-press on a photo, like this:



Options Menu

Another standard Android feature is the Options menu, which was previously triggered by a hardware key but is represented by three dots in newer versions of the operating system. We're going to add an options menu to take photos in addition to the button at the top of the screen.

1. The first thing we need is a new menu resource. Go ahead and add an XML file called `Resources/menu/photosactivity_menu.xml`, which is shown below:

```
<?xml version="1.0" encoding="utf-8" ?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
  <item android:id="@+id/menu_take_picture"
        android:icon="@drawable/ic_menu_take_picture"
        android:title="@string/menu_take_picture" />
</menu>
```

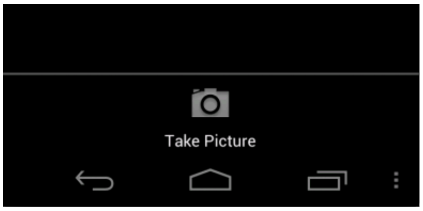
2. Let the activity know there is an options menu by overriding the `OnCreateOptionsMenu` method:

```
public override bool OnCreateOptionsMenu(IMenu menu)
{
    MenuInflater.Inflate(Resource.Menu.photosactivity_menu, menu);
    return true;
}
```

3. Add the following method to `PhotosActivity` so that the options menu item will start the process of taking a photo:

```
public override bool OnOptionsItemSelected(IMenuItem item)
{
    switch (item.ItemId) {
        case Resource.Id.menu_take_picture:
            TakePhoto();
            return true;
    }
    return base.OnOptionsItemSelected(item);
}
```

These methods will be invoked when the Activity must create an options menu. It will appear like this in the application:



Summary

In this chapter we looked at capturing an image in both iOS and Android, taking advantage of the Xamarin.Mobile component to create a simple, cross-platform way of taking a photo.

We then used platform-specific APIs – Core Image on iOS and ColorMatrix on Android – to create a similar sepia filter to apply to photos.

Two different approaches to social media were presented: on iOS using the platform-specific APIs and on Android using the Xamarin.Social component, which also works on iOS.

The last common piece of functionality we added was the ability to email photos as attachments on both platforms.

The iOS sample was further extended with the Satellite Menu component, and the Android sample was constructed to demonstrate the GridView control.