Web Services

Lab

Xamarin Inc.

## Overview

This tutorial introduces the process that integrates web service technologies into a Xamarin application. There are many choices when choosing this technology; Xamarin works with REST, WCF, and SOAP. Because of its simplicity and ubiquity, this tutorial will focus on how to use RESTful services with Xamarin.

REST stands for Representational State Transfer; it’s a style of architecture that is used for distributed systems. It is a stateless, cacheable protocol that was originally designed to work on simple HTTP calls. Many of the major service providers, such as Twitter, Flickr, and Facebook, use REST as the foundation of their public APIs. REST is also becoming increasingly popular because of its simplicity and platform independence. For example, REST makes it very easy for a PHP web application running on Linux to communicate with a Java desktop application running on Windows.

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

* **Android\_demo1** – Basic web services that use XML and JSON.
* **Android\_demo2** – Async web services.
* **iOS\_demo1** – Basic web services that use XML and JSON.
* **iOS\_demo1** – Async web services.

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

For more information about how to use web services in your mobile application, please consult Xamarin’s cross-platform [Introduction to Web Services](http://docs.xamarin.com/guides/cross-platform/application_fundamentals/introduction_to_web_services).

## REST Services

When you use Xamarin, consuming RESTful services is an architecturally simple and performance-conscious option. This section examines one technique you can use to interact with REST services from client applications, and introduces the various tools and patterns used to parse the most common message formats.

### Calling Web Services

The simplicity of RESTful web services has helped make them the primary option for mobile applications, since client interaction requires basic access to HTTP in lieu of a complete SOAP runtime.

System.Net.WebClient is one of the most common APIs used for retrieving data from a RESTful service. The following code snippet shows an example of how to call a web service to download a file from a (fictional) website:

var webClient = new WebClient();

var jsonString = webClient.DownloadString("http://someserver.com/svc/data.json");

The DownloadString method (as well as DownloadFile and DownloadData) perform an HTTP GET request, which means you can pass parameters in the query string and also set HTTP headers, but you cannot send additional data in the request body. REST services typically use other HTTP verbs to perform different actions on the server. Some examples are shown below that work against Microsoft’s Azure Web Service free-trial demo ToDo app.

HTTP headers are used to manage authentication and to negotiate the data format being used (in this case, JSON). The query string and request body are used to identify the operation and the specific object ID requested.

#### POST (Create a new item)

This code snippet creates a new item on the server. The UploadString method is used, and the object to be created is serialized as JSON and sent in the request body (payload).

var AddUrl = "https://" + subdomain + ".azure-mobile.net/tables/TodoItem";

client.Headers.Add (HttpRequestHeader.Accept, "application/json");

client.Headers.Add (HttpRequestHeader.ContentType, "application/json");

client.Headers.Add ("X-ZUMO-APPLICATION", MobileServiceAppId);

var payload = task.ToJson ();

var response = client.UploadString (AddUrl, "POST", payload);

// ...and wait...

var responseString = response;

The calling code can query the response body and the HTTP status code to confirm that the item was created and to determine its identifier (or primary key).

#### DELETE (Delete an item)

Unlike the POST request, the DELETE request includes the identifier (or primary key) of an object in the query string. This value, along with the DELETE HTTP verb, is how the REST server knows which operation to perform.

var DeleteUrl = "https://" + subdomain + ".azure-mobile.net/tables/TodoItem/**{0}**"; // includes object identifier

client.Headers.Add(HttpRequestHeader.Accept, "application/json");

client.Headers.Add(HttpRequestHeader.ContentType, "application/json");

client.Headers.Add("X-ZUMO-APPLICATION", MobileServiceAppId);

var payload = task.ToJson();

var response = client.UploadString(String.Format(DeleteUrl, task.Id), "DELETE", payload);

// ...and wait...

var responseString = response;

The calling code can query the response body and the HTTP status code to confirm that the delete operation was successful.

#### PATCH (Update existing item)

As with the DELETE request, the object being updated has its identifier included in the query string. The updated object is serialized as JSON in the request body. The server uses these pieces of information (plus the content type) to parse the request and perform the correct operation.

var UpdateUrl = "https://" + subdomain + ".azure-mobile.net/tables/TodoItem/**{0}**"; // includes object identifier

WebClient client = new WebClient();

client.Headers.Add (HttpRequestHeader.Accept, "application/json");

client.Headers.Add (HttpRequestHeader.ContentType, "application/json");

client.Headers.Add ("X-ZUMO-APPLICATION", MobileServiceAppId);

var payload = task.ToJson ();

var response = client.UploadString (String.Format (UpdateUrl,task.Id), "PATCH", payload);

// ...and wait...

var responseString = response;

The calling code can query the response body and the HTTP status code to confirm that the update succeeded.

### Alternatives for Web Services

The above examples using WebClient to perform REST operations are perfectly valid, but it’s clear that there is quite a bit of repetition setting up the HTTP headers and serializing the request body.

Rather than writing this code over and over again, there are a number of components and libraries that provide a higher-level API for accessing web services.

External libraries that you can use to access web services with less code include:

* **Parse** – <https://components.xamarin.com/view/parse/>
* **ServiceStack** – <https://github.com/ServiceStack/ServiceStack>
* **Azure Mobile Services** – <https://components.xamarin.com/view/azure-mobile-services/>
* **RestSharp** – <http://restsharp.org/>
* **Hammock** – <https://github.com/danielcrenna/hammock2>
* **Amazon Web Services**

### Blocking

When a web service is called synchronously, it blocks the main thread (also called the UI thread), preventing the user from interacting with the application.

In order to prevent this, an application should do as little work in the UI thread as possible. Long running or complex operations should be initiated and completed in a worker thread. When these operations are finished, their results are returned to the UI thread and the user interface is updated with these values.

### Calling Web Services Asynchronously

Writing safe, multi-threaded code can be hard, but the .NET framework provides the Task Parallel Library (TPL) to make things easier. TPL is a set of APIs that simplify the writing of multi-threaded code, allowing developers to concentrate on the functionality of their applications. This section briefly touches on how to use the TPL to perform asynchronous operations. The following code demonstrates how to use the TPL to download a file from the Internet and then update the UI thread:

private void MethodExecutingOnUIThread()

{

// This method must be called from the UI thread.

\_progressDialog = ProgressDialog.Show(this, “Downloading”,

“Please wait while the file downloads…”, true, false);

Task.Factory.StartNew(() =>

{

var webclient = new WebClient();

webclient.DownloadFile(“URI”, “LocalFile”);

// Do something with the file here – maybe load it into a database

})

.ContinueWith(task =>

{

// Do any updates to the UI here

\_progressDialog.Dismiss();

}, TaskScheduler.FromCurrentSynchronizationContext());

}

Task.Factory.StartNew creates and starts a new operation, represented by the class [Task<TResult>](http://msdn.microsoft.com/en-us/library/dd321424.aspx). The code that is inside the lambda expression will be executed asynchronously.

The call to ContinueWith creates a continuation that will execute after the task is complete. A continuation is a task that will execute after another task completes. The call to ContinueWith receives two parameters:

* A lambda with some code to execute.
* An instance of the [TaskScheduler](http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskscheduler.aspx) class. The TaskScheduler class is responsible for queuing work onto threads.

Normally, the continuation will execute asynchronously, but in this example we provide a TaskScheduler that tells the TPL how to queue the work. The call to TaskSchedule.FromCurrentSynchronizationContext()retrieves an instance of System.Threading.SynchronizationContext that is associated with the UI thread. The result of this call ensures that the second lambda that is provided will run on the UI thread, and not in a background thread.

For more in depth information about the Task Parallel Library, consult [Microsoft’s document](http://msdn.microsoft.com/en-us/library/dd460717.aspx).

Note: Only update the user interface from the UI thread. Do not perform any updates to the UI from a background thread.

## Options for Consuming Web Service Data

After returning the response from an HTTP request, we will be interested in consuming the data in a structured format. This can be accomplished by deserializing the response into defined model objects. Of course, one of the benefits associated with using the Xamarin mobile frameworks is the potential re-use of ubiquitous model objects employed throughout the overall application architecture.

The examples in this chapter use both XML and JSON data formats.

### XML

.NET has had great support for XML and serialization since version 1.0. Even with complex XML schemas, it is possible to use attributes to ensure that the XmlSerializer can successfully parse the data with only a few lines of code:

XmlSerializer serializerXml = new XmlSerializer(typeof(EventCollection));

object o;

using (System.IO.Stream stream = new FileStream(SessionsXmlFilePath, FileMode.Open)) {

o = serializerXml.Deserialize(stream);

stream.Close();

}

EventCollection events = (EventCollection)o;

### JSON

Xamarin ships with support for JSON out-of-the-box via classes in the [System.Json](http://msdn.microsoft.com/en-us/library/cc626400(v=vs.95).aspx) namespace. By using a [System.Json.JsonObject](http://msdn.microsoft.com/en-us/library/system.json.jsonobject(v=vs.95).aspx), results can be retrieved, as shown below:

var jarray = (JsonArray)JsonValue.Load(fileStream);

var json = jarray[0]; // first one

var speaker = new Speaker();

speaker.Name = json["name"];

speaker.Bio = json["about"];

speaker.Company = json["company"];

This is the familiar key-value coding style. However, it’s important to be aware that the System.Json tools load the entirety of the data into memory and may not be very efficient for large sets of data.

### Alternatives for JSON parsing

In addition to the built-in System.Json parsing support, there are a number of other options:

* **Json.NET** <https://components.xamarin.com/view/json.net/>
* **ServiceStack.Text** <https://github.com/ServiceStack/ServiceStack.Text>

## iOS Web Services

Now that we have an understanding of how to consume web services, let’s walk through updating an application that consumes web services. To get this example started, download the zip file WebServices.zip, and then unzip the solution. We must change the application so that it will call a RESTful web service, and then save the data from that web service to a file on the device.

These applications are cross-platform and share a common set of Core code. The web service access and parsing will be shared code that we’ll access from both iOS and Android samples.

### Consuming Web Services

Our application needs to be able to retrieve the conference information from the Internet. We will change our application so that it will retrieve updated conference information from the web, deserialize the data, and then update our user interface.

1. Let’s start by adding some code to call a web service to retrieve some data about the speakers and sessions. We’ll use XML for the session data and JSON for the speakers. Uncomment the code in the Core project’s Downloader.cs file:

private void DownloadSessionXml()

{

var file = SessionsXmlParser.Instance.SessionsXmlFilePath;

if (File.Exists(file)) {

File.Delete(file);

}

var webclient = new WebClient();

webclient.DownloadFile(sessionsXmlUrl, file);

}

public static void DownloadSpeakerJson()

{

var file = SpeakersJsonParser.Instance.SpeakersJsonFilePath;

if (File.Exists(file)) {

File.Delete(file);

}

var webclient = new WebClient();

webclient.DownloadFile(speakersJsonUrl, file);

}

1. Update the ViewDidLoad method of MyTabBarController so that it calls the download methods, as shown in the following code snippet:

public override void ViewDidLoad ()

{

base.ViewDidLoad ();

Downloader.DownloadSessionXml ();

Downloader.DownloadSpeakerJson ();

// rest of the code omitted for berivity...

}

Assuming there is an Internet connection, running the application now should cause the XML and JSON files to be downloaded. We still need to parse the data so that it can be displayed in the Sessions and Speakers tabs.

#### Parsing Sessions XML

1. First, examine the XML structure that needs to be parsed. An example of the Evolve schedule is shown here:

<?xml version="1.0" encoding="UTF-8"?>

<events>

<event>

<event\_key>4</event\_key>

<active>Y</active>

<title>Training Keynote/Introduction to Mobile Development</title>

<event\_start>2013-04-14 09:00</event\_start>

<event\_end>2013-04-14 10:00</event\_end>

<description>Welcome to Xamarin Evolve Training Days and Introduction to Mobile Development</description>

<venue>Dawkins Salon (4th Floor)</venue>

<speakers>

<person>

<name>Nat Friedman</name>

<bio></bio>

</person>

<person>

<name>Bryan Costanich</name>

<bio></bio>

</person>

</speakers>

<tags>Fundamentals</tags>

</event></events></xml>

The SessionXmlDTOs.cs file in the Core project contains class definitions that match this XML structure. Once you have defined the correct class structure, deserializing XML is easy!

[Serializable]

public class @event { // the @ symbol lets you use use keywords as names

[XmlElement("event\_key")]

public string EventKey {get;set;}

[XmlElement("title")]

public string title {get;set;}

[XmlElement("description")]

public string description {get;set;}

[XmlElement("venue")]

public string venue {get;set;}

[XmlElement("event\_type")]

public string EventType {get;set;}

[XmlElement("event\_start")]

public string EventStart {get;set;}

[XmlElement("event\_end")]

public string EventEnd {get;set;}

[XmlArray("speakers", IsNullable = true)]

public person[] speakers {get;set;}

[XmlElement("tags")]

public string Tags {get;set;}

}

public class person {

[XmlElement ("name", IsNullable = true)]

public string name {get;set;}

}

[System.Xml.Serialization.XmlRootAttribute("events", Namespace = "", IsNullable = false)]

public class EventCollection

{

[XmlElement("event")]

public @event[] @event { get; set; }

}

1. Now, let’s enable the XML parser in the Core project’s SessionsXmlParser.cs file. Uncomment the code in the GetSessions(). The first part of the method performs the deserialization by using the classes defined above:

XmlSerializer serializerXml = new XmlSerializer(typeof(EventCollection));

object o;

using (System.IO.Stream stream = new System.IO.FileStream(SessionsXmlFilePath, System.IO.FileMode.Open)) {

o = serializerXml.Deserialize(stream);

stream.Close();

}

EventCollection events = (EventCollection)o;

1. The second part of the GetSessions() method loops through the deserialized objects and copies them into our own Session model class. When accessing web services controlled by other parties, it’s usually a good idea not to rely too heavily on their object model, since it could change at any time. This also enables us to perform data conversions and validation before using the web service data in the rest of the application.

foreach (var evnt in events.@event) {

var session = new Session() {

Id = Convert.ToInt32 (evnt.EventKey),

Title = evnt.title,

Location = evnt.venue,

Abstract = evnt.description,

Begins = DateTime.Parse (evnt.EventStart),

Ends = DateTime.Parse (evnt.EventEnd),

};

if (evnt.speakers != null) { // which happens for LUNCH and TBA sessions without a Speaker

foreach (var sp in evnt.speakers) {

if (!String.IsNullOrWhiteSpace(sp.name))

session.Speaker = new Speaker() {Name = sp.name};

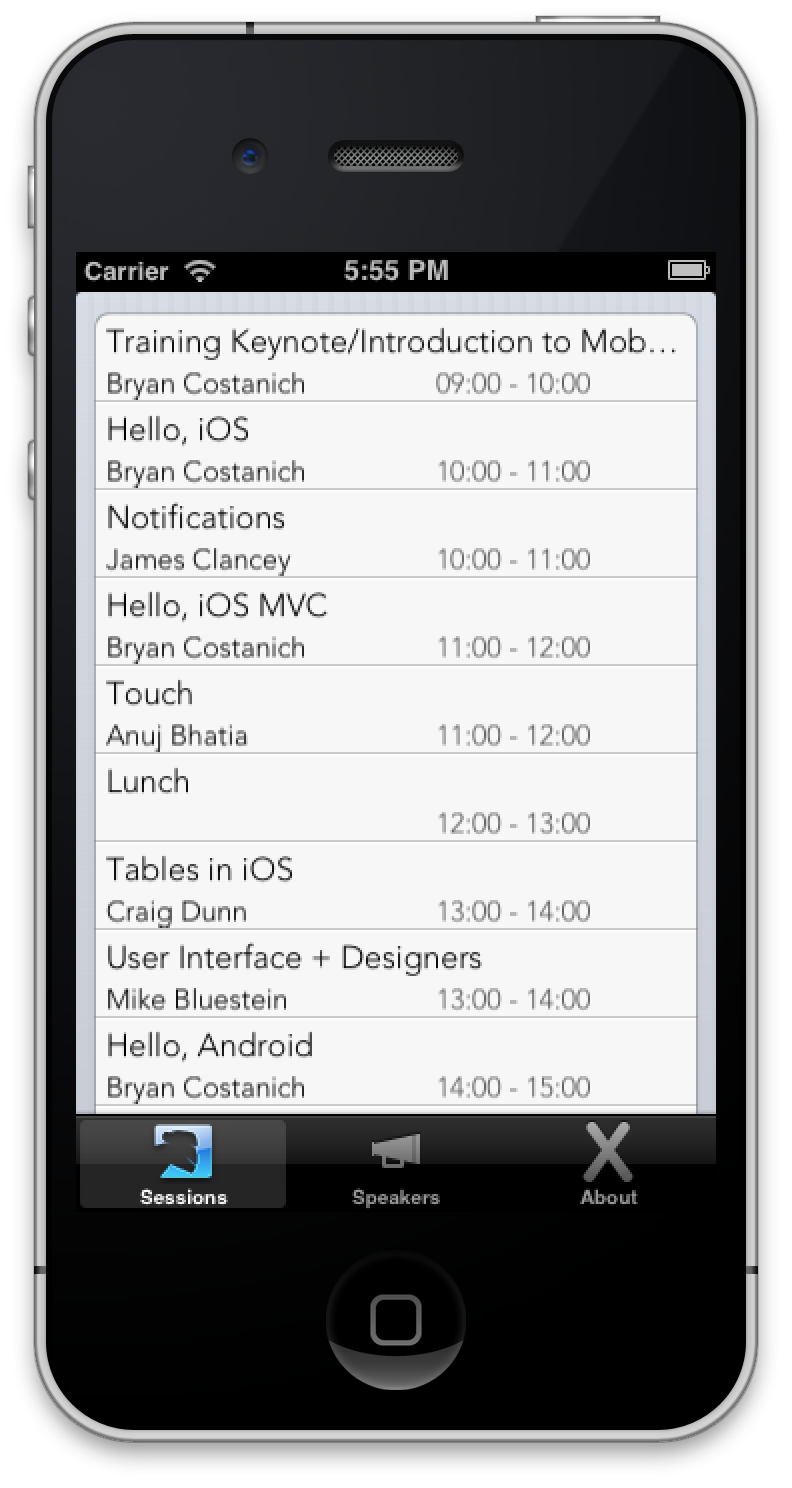
}

}

localSessions.Add(session);

}

If we click on the Sessions tab, we will see the data shown in the following screenshot:



#### Parsing Speakers JSON

1. To parse the Speakers JSON file, first examine the format:

[{"name":"Nat Friedman",

"company":"Xamarin",

"position":"CEO",

"about":"Nat Friedman, Xamarin&rsquo;s co-founder and CEO, has been writing software for 27 years and is passionate about entrepreneurship and building amazing products. Nat will cover key mobile trends and predictions, Xamarin&rsquo;s future direction, and ways for you to capitalize on mobile - the largest software market ever.",

"url":"@natfriedman",

"avatar":"http://static.sched.org/a2/547192/avatar.jpg.75x75px.jpg"},]

The code to parse JSON is in SpeakersJsonParser.cs in the Core project. The first step is to load the JSON file into a JsonArray, which is a key-value collection of the JSON data that we can then further process.

using (var r = new StreamReader(File.OpenRead(SpeakersJsonFilePath))) {

var ja = (JsonArray)JsonValue.Load(r);

foreach (var speaker in ja) {

var s = JsonToSpeaker (speaker);

if (s != null)

localSpeakers.Add (s);

}

}

The JsonToSpeaker method processes each element of the array and returns a Speaker object from our Model class definitions.

Speaker JsonToSpeaker(JsonValue json)

{

Speaker speaker = null;

try {

speaker = new Speaker();

speaker.Name = json["name"];

speaker.Bio = json["about"];

speaker.Company = json["company"];

} catch {

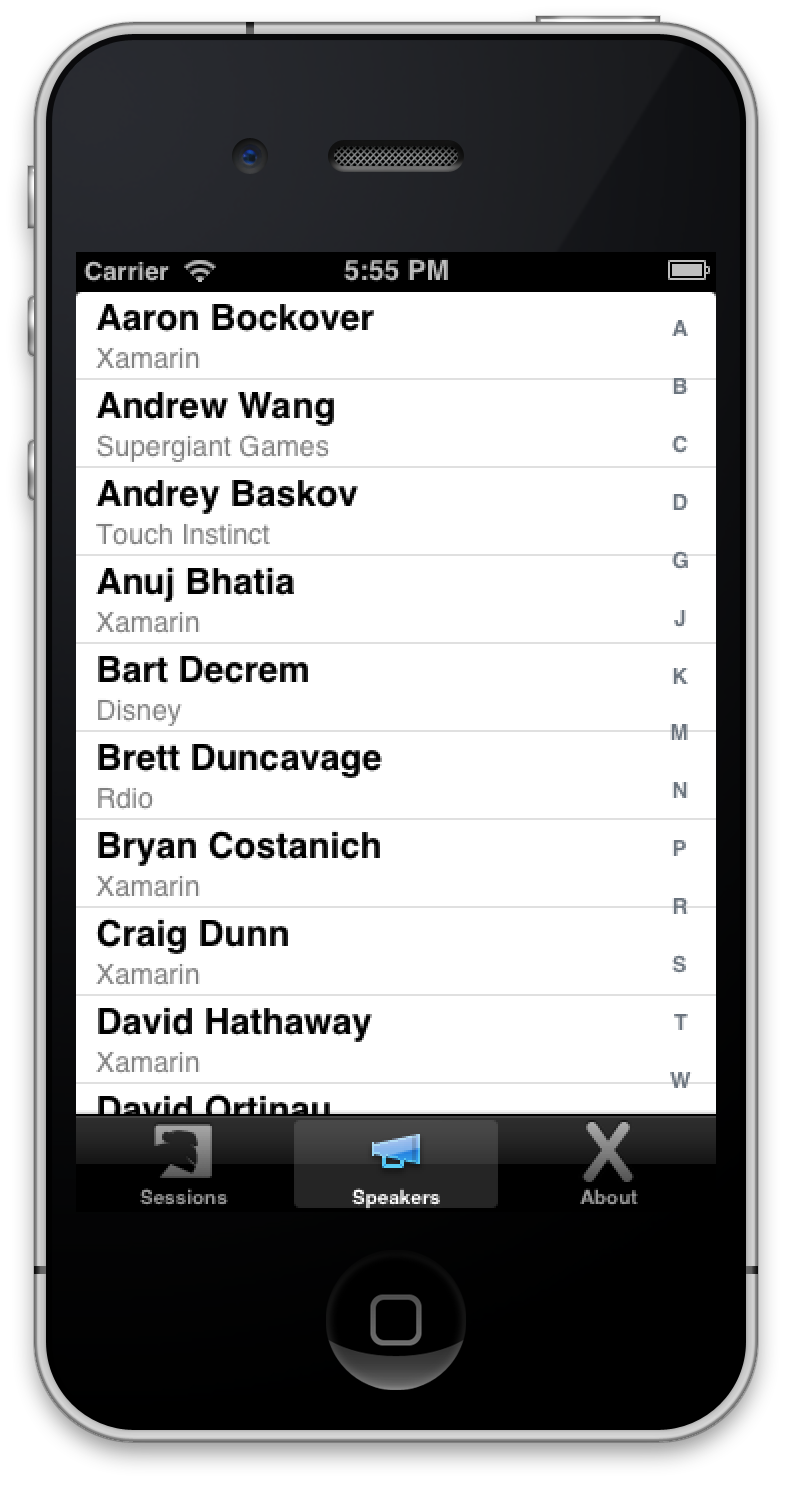
// lazy way to deal with missing json elements

}

return speaker;

}

The Speakers tab will now look like the following screenshot. The sample code includes all the code to enable the index that appears on the right of the screen.



### Using a Background Thread

This seems to work fine, but there is one problem—if the call to the web service takes a long time, say due to poor Internet connection or because there is a very large amount of data—our application will appear very unresponsive.

If the main UI thread is unresponsive for too long when the application starts up, iOS will intervene and terminate the application. Ideally, apps should be responsive within a few seconds; 10-15 seconds is probably too long!

Let’s use a background thread to address this potential problem. It’s easy to edit MyTabBarController’s ViewDidLoad to resemble the following code snippet:

public override void ViewDidLoad ()

{

base.ViewDidLoad ();

Task.Factory.StartNew(() => {

Downloader.DownloadSessionXml() ;

Downloader.DownloadSpeakerJson();

});

// rest of the code omitted for berivity...

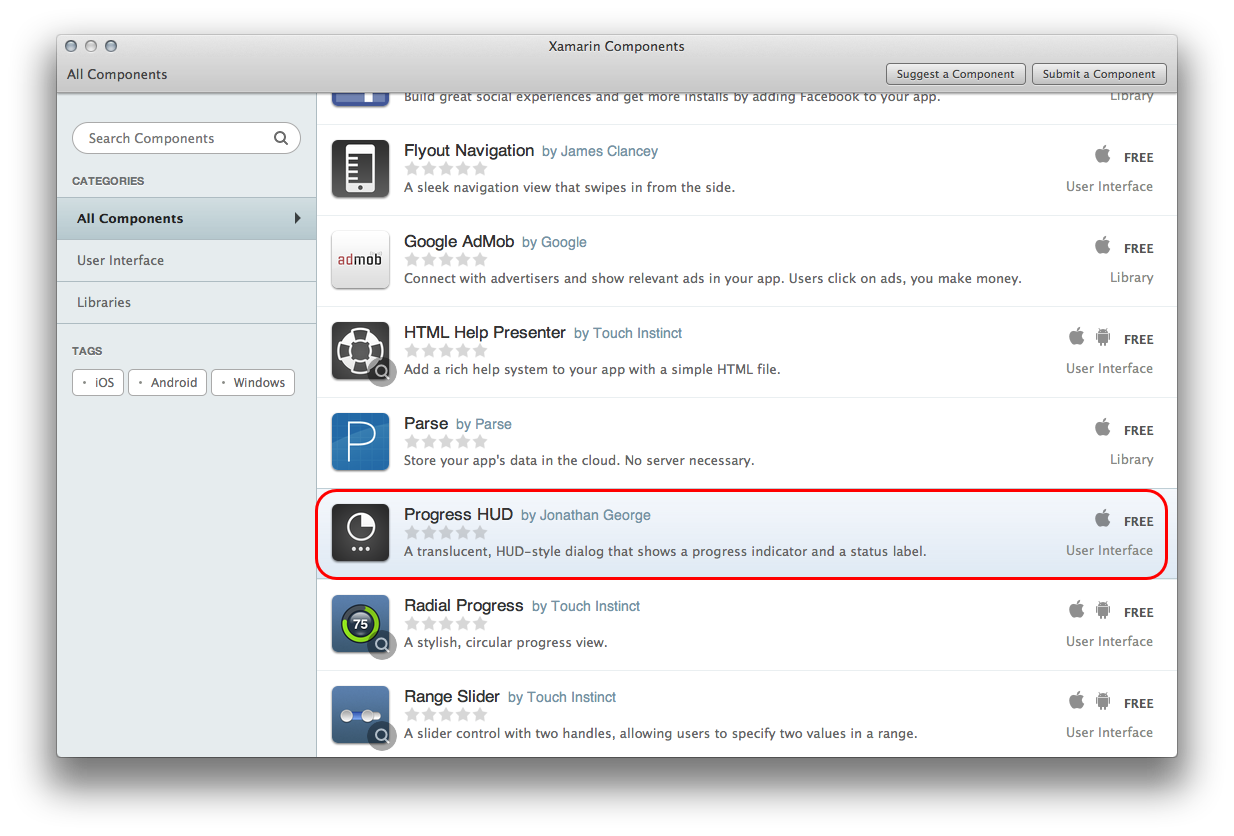
}

The change that we just made uses the Task Parallel Library to create a task that will run the web service in a background thread. There are two new problems with this:

* The user doesn’t know what is happening while the download occurs.
* When it has completed, the Sessions tab is already visible (and empty), and does not get updated with the downloaded data.

### Using the Component Manager

To fix the first issue, we’ll display some sort of visual indicator that the download is happening. We could code this all by hand, but there is already a third-party component, called Progress HUD, that will do this for us. Let’s walk through how to add and then use this component in our application.

1. Start up the Component Manager as shown in the following screen shot:
2. , 
3. Click on the Progress HUD row, which will display an information page for the Progress HUD. It should look something like the following screenshot:
4. Click Add to Project. At this point, the component has been added to your project.
5. Now, let’s edit MyTabBarController so that it will display a progress dialog while the web service is being called. Add the following instance variable to the class MyTabBarController:

private MTMBProgressHUD hud;

1. Now, update the ViewDidLoad method to resemble the following code snippet:

public override void ViewDidLoad ()

{

base.ViewDidLoad ();

hud = new MTMBProgressHUD(View)

{

LabelText = "Updating",

DetailsLabelText = "Downloading info",

RemoveFromSuperViewOnHide = true,

DimBackground = true

};

View.AddSubview(hud);

hud.Show(true);

Task.Factory.StartNew(() => {

Downloader.DownloadSessionXml() ;

Downloader.DownloadSpeakerJson();

}).ContinueWith(task1 => {

hud.Hide(true);

hud = null;

},

TaskScheduler.FromCurrentSynchronizationContext ());

// rest of the code omitted for berivity...

}

This code will display a progress indicator in the user interface and the run Download methods in a background thread. Once the download and parsing has finished executing, the continuation task will be executed on the UI thread and will then dismiss the component.

1. If we ran the application now, the Sessions tab would still be empty because it is already being displayed and it has no way to know that new data has been downloaded. To fix this, update the SessionsViewController ViewDidLoad method and add a Refresh method, as shown here:

public override void ViewDidLoad ()

{

base.ViewDidLoad ();

TableView = new UITableView (Rectangle.Empty, UITableViewStyle.Grouped);

Refresh ();

}

public void Refresh () {

TableView.Source = new SessionsTableSource ();

TableView.ReloadData ();

}

1. In MyTabBarController, add a call to the Refresh method, after the downloads are complete.

Task.Factory.StartNew(() => {

Downloader.DownloadSessionXml() ;

Downloader.DownloadSpeakerJson();

}).ContinueWith(task1 => {

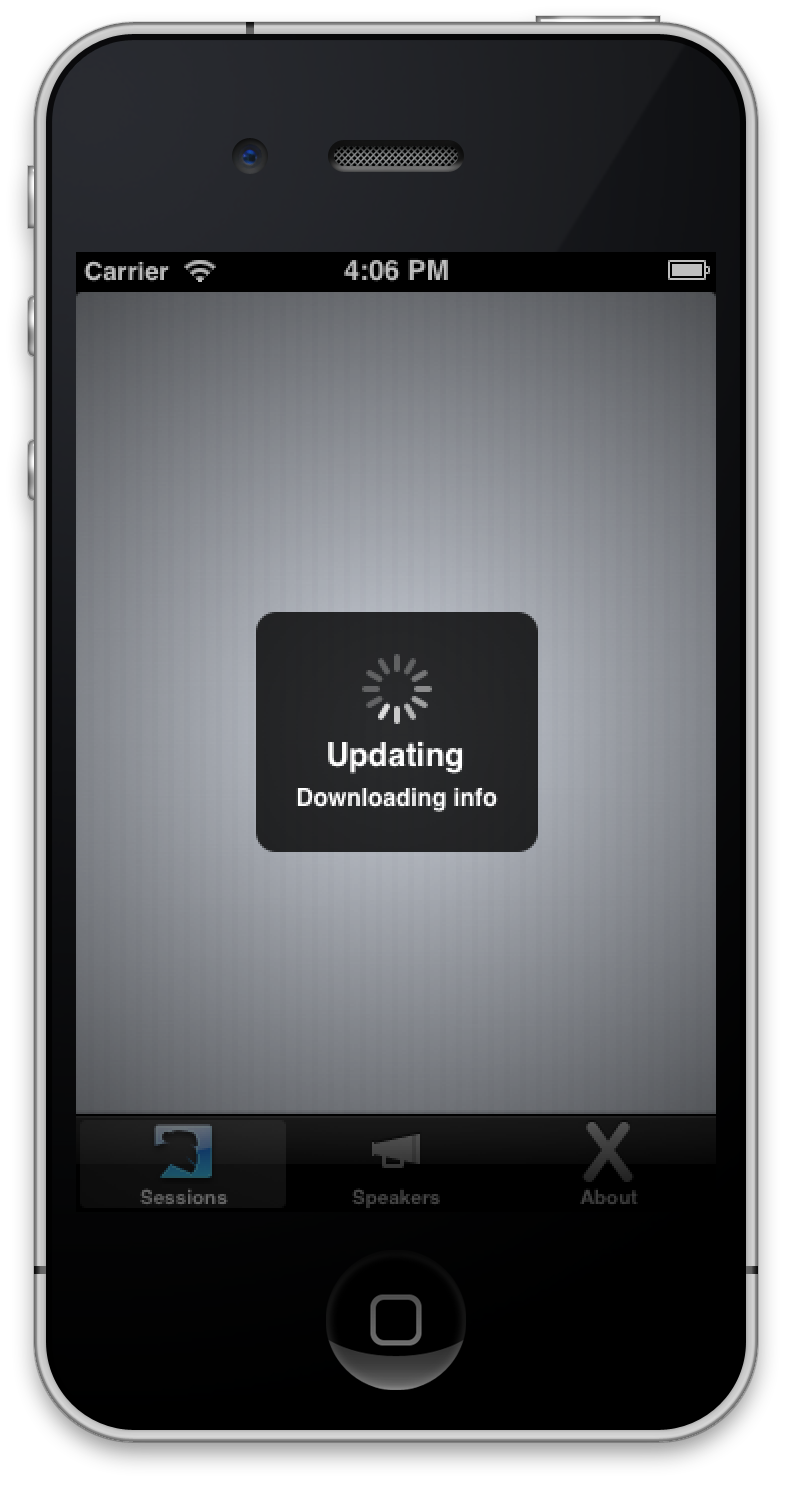
**vc1.Refresh();**

hud.Hide(true);

hud = null;

}, TaskScheduler.FromCurrentSynchronizationContext ());

1. After the download has completed, run the application. When the application starts, it will look something like the following screenshot:



Congratulations! At this point, you have successfully updated this application to consume a web service in the background.

## Android

Now that we have an understanding of how to consume web services in iOS, let’s apply the same concepts to Xamarin.Android.

### Getting Started

To get this example started, download the zip file WebServices.zip and unzip the solution.

Some parts of the sample applications are already set up:

* **Core project** – The XML and JSON downloading and parsing classes are shared with the iOS sample found at the start of this chapter. We don’t need to change that code at all for our Android samples.
* **Activities & Adapters** – There are two ListActivities and associated adapters already defined for the Speakers and Sessions tabs.
* **Resources/menu/mainactivity\_menu.xml** – This is a layout file for an options menu. As a part of this tutorial, we will add an options menu so that users may refresh the conference data with the push of a button.
* **Resources/values/Strings.xml** – This has been updated with some new string resources for the project.

You might want to take a quick minute to familiarize yourself with the changes that have been made.

### Consuming the Web Service

Our application needs to be able to retrieve the conference information from the Internet. We will change our application so that it will retrieve updated conference information from the web, deserialize the XML and JSON, and then update our screens. Because we have already done all the hard work earlier in the chapter—downloading and deserializing the data in the iOS example—we are going to reuse that code in Android.

1. Review the code in the Downloader.cs file in the Core project. This code was introduced in the iOS section above, and can be shared unchanged with Android.
2. Review the code in the SessionsXmlParser.cs and SpeakersJsonParser.cs files in the Core project. These classes were also introduced in the iOS section above, and can be shared with Android.
3. In the SessionsActivity class, uncomment the two lines that cause the XML to be downloaded and parsed, using the Core code that is shared with iOS.

protected override void OnCreate(Bundle bundle)

{

base.OnCreate(bundle);

**Downloader.DownloadSessionXml() ;**

DisplaySessions();

}

private void DisplaySessions()

{

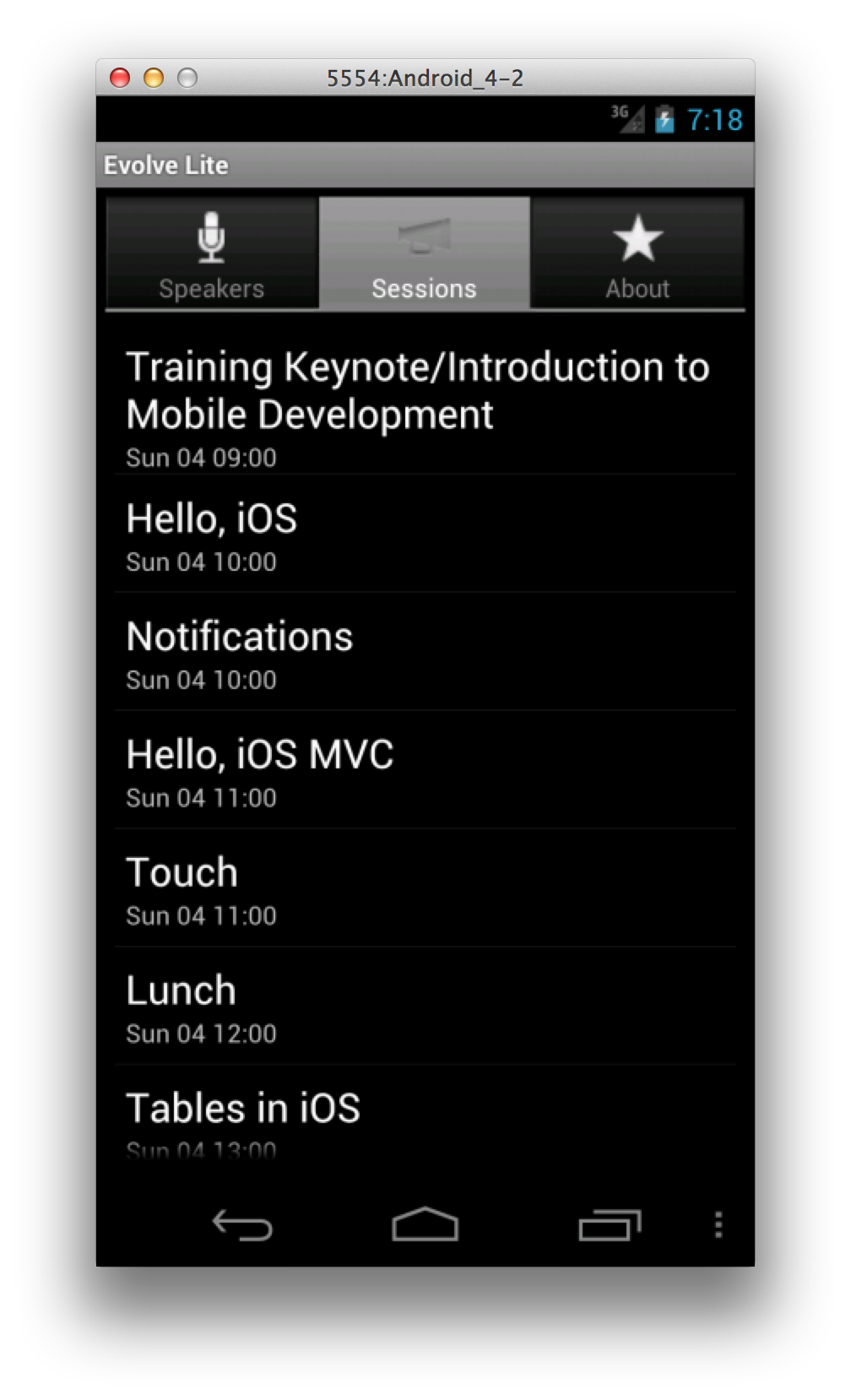
**var sessions = SessionsXmlParser.Instance.Sessions;**

adapter = new SessionsAdapter(this, sessions);

ListView.Adapter = adapter;

}

Because we have already written a SessionsAdapter class (in the ListViews chapter), that’s all we need to display the Session data retrieved from the web service. The Sessions tab looks like this:



1. It is equally simple to reuse the Core code to download and display the Speakers data. In the SpeakersActivity class, uncomment the two lines that cause the XML to be downloaded and parsed, using the Core code that is shared with iOS.

protected override void OnCreate(Bundle bundle)

{

base.OnCreate(bundle);

**Downloader.DownloadSpeakerJson();**

DisplaySpeakers();

}

private void DisplaySpeakers()

{

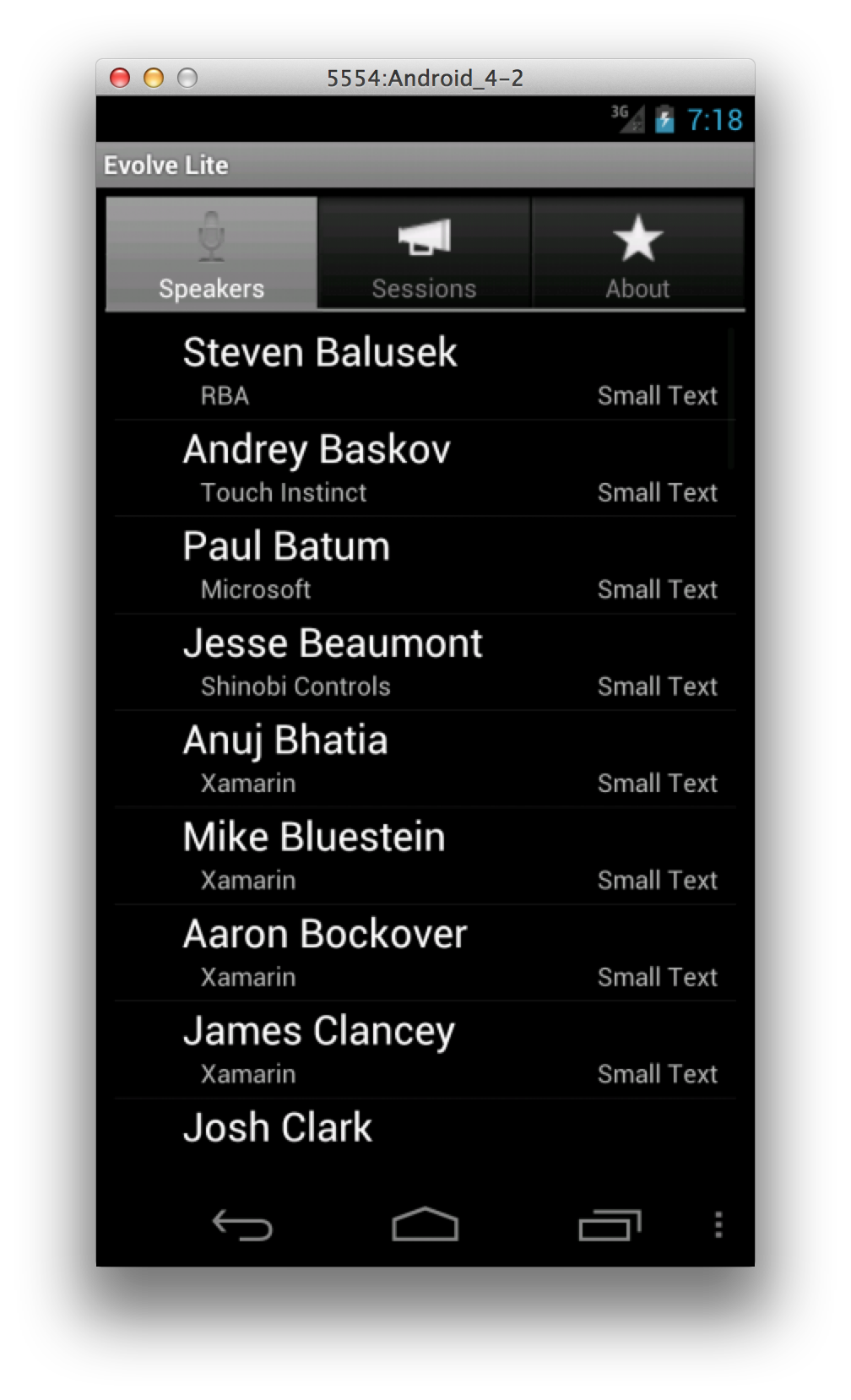
**var speakers = SpeakersJsonParser.Instance.Speakers;**

adapter = new SpeakersAdapter(this, speakers);

ListView.Adapter = adapter;

}

The Speakers tab looks like this when populated with data from the web service:



### Adding a Refresh Menu

We’ve just introduced a new concept in this step, so let’s take a minute and examine what is going on here. The first method, OnCreateOptionsMenu, is invoked by Android during the creation of an Activity. It is a callback that allows an Activity to create or contribute to a menu. In our example, we create the menu from an Android menu resource stored at Resources/menu/mainactivity\_menu.xml. The contents of this file are shown in the following snippet:

<?xml version="1.0" encoding="utf-8" ?>

<menu xmlns:android="http://schemas.android.com/apk/res/android">

<item android:id="@+id/menu\_refresh\_data"

android:icon="@drawable/ic\_menu\_refresh\_data"

android:title="@string/menu\_refresh\_data"

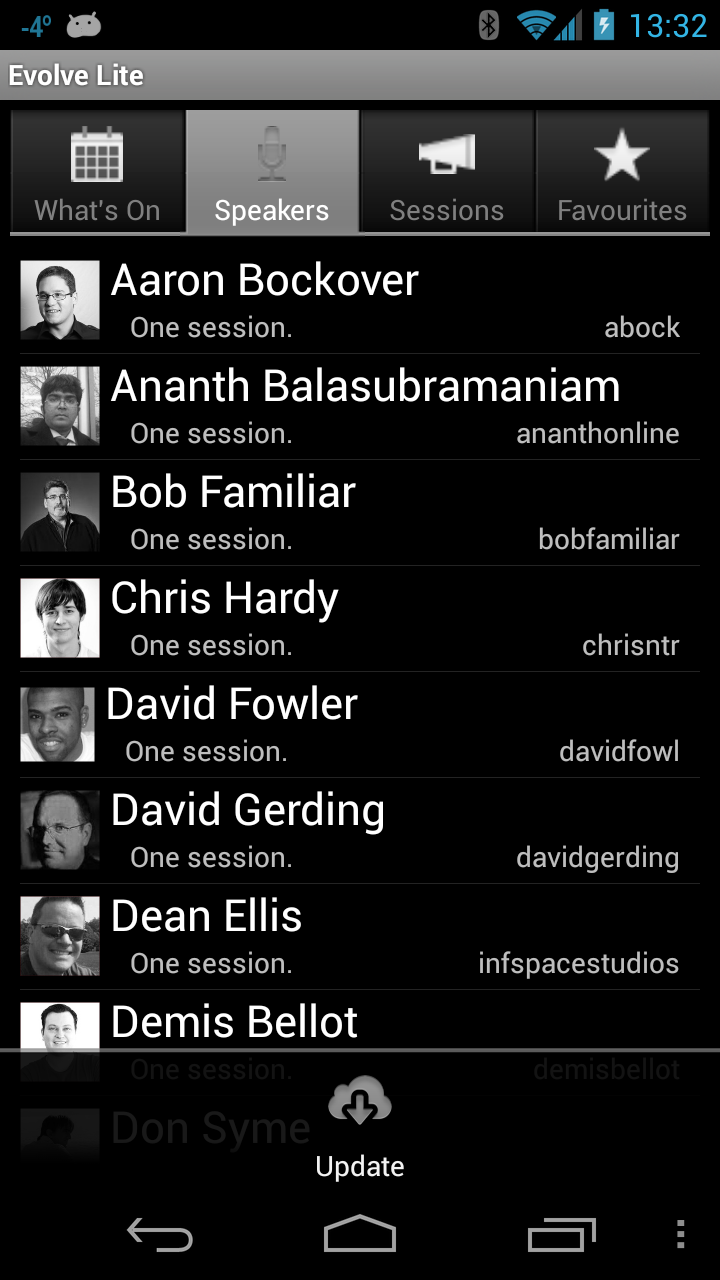
/>

</menu>

This file should seem pretty familiar to you by now. It holds the metadata that the MenuInflater helper will use to add to the options menu for this activity.

The other method that was added to the Activity, OnOptionsItemSelected, is called by the system when the user selects a menu item. It takes a MenuItem item parameter that is the item that the user selected. The method compares the ID of the selected menu item with a list of know menu IDs, and then performs the appropriate action. OnOptionsItemSelected will return true for each menu item that it successfully handled. Unknown menu IDs should always be handled by the superclass implementation. For more detailed information about how to create menus in Android, consult [Google’s documentation on Menus](http://developer.android.com/guide/topics/ui/menus.html).

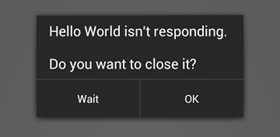
If you run the application, you should be able to see the options menu as shown in the following screenshot:



If you select Update, then the application will retrieve updated conference data from the web and update the speaker list. However, all this is happening on the UI thread, which is something our application should avoid doing. Let’s move on and address this issue by performing the update on a background thread and by displaying some sort of “in progress” indicator to the user so that they know what is happening.

### Using a Background Thread

When a web service is called synchronously, it blocks the main thread (also called the UI thread), preventing the user from interacting with the application. If the main UI thread is unresponsive for more than about five seconds, Android will intervene with the dreaded Application Not Responding (ANR) dialog, as shown in the following screenshot:



In order to prevent this, an application should do as little work in the UI thread as possible. Long running or complex operations should be run in a worker thread. When these operations are finished, their results are returned to the UI thread and then the user interface is updated with the values.

We can use the same Task.Factory.StartNew method that we did in iOS.

Edit the file SpeakersActivity.cs, updating the code to use a ProgressDialog and Task.Factory.StartNew:

SpeakersAdapter adapter;

ProgressDialog progressDialog;

List<Speaker> speakers;

protected override void OnCreate(Bundle bundle)

{

base.OnCreate(bundle);

progressDialog = ProgressDialog.Show(this, "Loading", "Speakers", true, false);

Task.Factory.StartNew(() => {

Downloader.DownloadSpeakerJson();

speakers = SpeakersJsonParser.Instance.Speakers;

}).ContinueWith(task => {

DisplaySpeakers();

}, TaskScheduler.FromCurrentSynchronizationContext());

}

private void DisplaySpeakers()

{

if (progressDialog != null) {

progressDialog.Dismiss();

progressDialog = null;

}

adapter = new SpeakersAdapter(this, speakers);

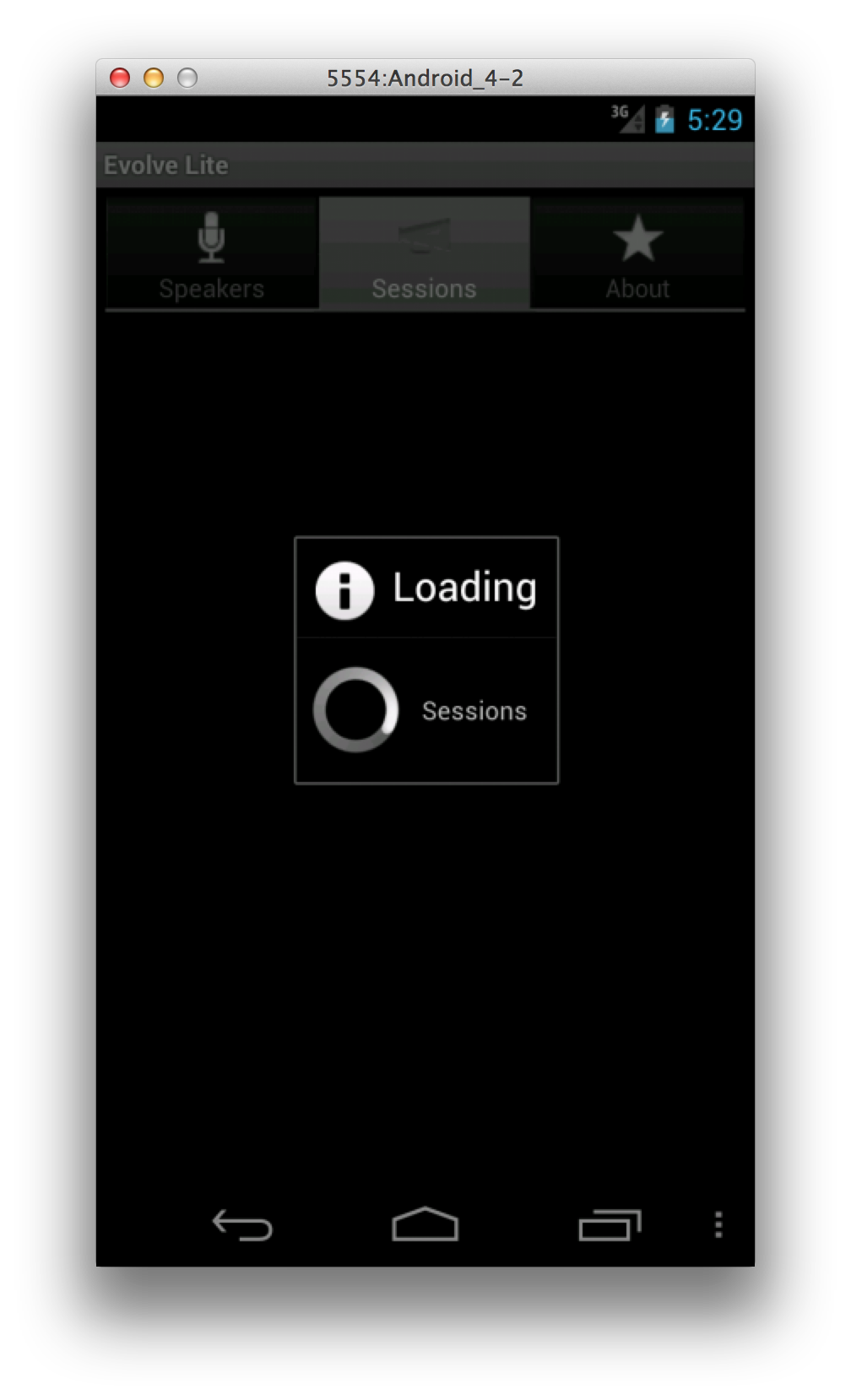
ListView.Adapter = adapter;

}

This method will be used to display a ProgressDialog on the Activity while the data is downloaded and parsed on a background thread.

Once the background thread is finished, the continuation that we have set up to run on the UI thread will display the data in the ListView and hide the progress indicator.

Run the application and select the Speakers tab. Open up the options menu, and then select Update. The screen of your application should change to resemble the following screenshot while the data is being updated:



Congratulations! At this point, you have a working application that will refresh itself from a web service and will avoid the dreaded ANR that would prompt users to force-close your application.

## Summary

This tutorial covered various options that showed how to consume web services seamlessly with Xamarin.iOS and with Xamarin.Android. It discussed consuming XML and JSON data that were provided by a RESTful web service.

To keep our application responsive, we saw how to perform busy or time-consuming applications in the background by using the Task Parallel Library. We also learned how to add a third-party component to our application, and how to use that component to display a progress dialog to the user while our application was busy.