Notifications

Evolve Advanced Track, Chapter 1

Overview

Mobile applications use notifications as a way of informing the user that some application specific event has happened while the application. Notifications are typically used to notify the user about the status of some background process, or to display information when then application is running in the background. An example of this might be downloading a large file. This file might take a long time to download, so this activity should occur in the background. When the download is complete, the user is informed of the fact by a notification.

Notifications can also be used to alert the user when certain events happen. For example, if you are building a calendar application, you can enable pop-up alerts when a notification arrives letting your users know of an event that is coming up.

There are two types of notifications – *local notifications* and *remote notifications* (also know as push notifications or messages). Local notifications are scheduled by applications running on the device that will notify the user that an event of interest has occurred. Local notifications are intended to efficiently notify the user that a local application has an event.

Remote notifications are lightweight messages that are send from another server or application to a mobile application over a network. They are can be a very important part of a mobile application as they are one of the more efficient and reliable ways to have a device communication with other applications on other devices.

For instance, consider an e-mail client. In order to make sure that the e-mail client is up to date with the latest e-mails, it has to poll a server every so often and check for new e-mail. Polling is a horribly inefficient technique for this type of communication. In order for the e-mail client to keep current, it must make frequent and expensive network calls that will drain the battery.

Push notifications are designed specifically for this type of scenario. Instead of having our application continuously poll a server for data, we can push a small message to the application that tells it that it needs to do something, such as contacting an e-mail server to retrieve the most recent e-mails.

However, remote notifications do add significant complexity, effort and infrastructure. They require a separate application (typically running on a server) that will send notifications to a notification service. The notification service, in turn, identifies devices that are to receive the notification and delivers them in an efficient manner.

Each mobile operating system has it's own infrastructure and API's for remote notifications. Apple has provided the *Apple Notification Push Service* to support remote notifications on iOS devices, while Google provides *Google Cloud Messaging* to enable messaging on Android devices.

To address the server side complexity of supporting two disparate notification services, Jonathan Dick created an open source library called PushSharp. PushSharp provides an abstraction that allows server applications to target one

coherent API that will communicate with either the Apple Notification Push Service or Google Cloud Messaging.

In this chapter we will look at local and remote notifications on Android and iOS. We will first look a how local and remote notifications work in iOS. We learn about the steps required to set up push notification in Application Notification Push Service and how iOS applications can respond to these notifications. After that we will move on to see how to do similar things in Android. We will learn about how local notifications work in Android, how to setup Google Cloud Messaging to publish messages, and how an Android application will respond to messages from Google Cloud Messaging.

Notifications in iOS

iOS has three ways to indicate to the user that a notification has been received:

- Sound or Vibration – iOS can play a sound to notify users. If the sound is turned off, the device can be vibrated as well.
- Alerts – it is possible to display a dialog on the screen with information about the notification.
- Badges – when a notification is published, it is possible to display an image or a number on the application icon.

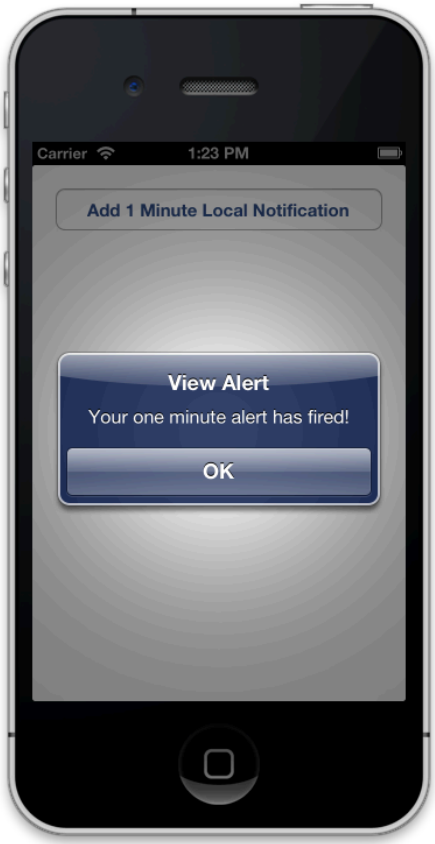
iOS also provides a *Notification Center* that will display all the notifications – local or remote – to the user. Users may access this by swiping down from the top of the screen. The following image shows the Notification Center:

iOS makes it fairly simple to create and handle local notifications. To schedule a local notification you create a `UILocalNotification` object, set the `FireDate`, and

schedule it via the `ScheduleLocalNotification` method on the `UIApplication.SharedApplication` object. The following code snippet will show how to schedule a notification that will fire one minute in the future, and display an alert with a message:

```
UILocationNotification notification = new UILocalNotification();
notification.FireDate = DateTime.Now.AddMinutes(1);
notification.AlertAction = "View Alert";
notification.AlertBody = "Your one minute alert has fired!";
UIApplication.SharedApplication.ScheduleLocalNotification(notification);
```

The following screenshot show what this alert might look like:



If you want to apply a badge to the application icon with a number, you can set it as shown in the following line code:

```
notification.ApplicationIconBadgeNumber = 1;
```

In order play a sound with the icon, set the `SoundName` property on the notification as shown in the following code snippet:

```
notification.SoundName = UILocationNotification.DefaultSoundName;
```

A notification should not consist of just a sound. There should also be a badge or an alert to help the user identify the application that raised the alert. Also, if the sound is longer than 30 seconds, iOS will play the default sound instead.

> **Note:** There is a bug in the iOS simulator that will fire the delegate notification twice. This issue should not occur when running the application on a device.

Handling Notifications

iOS applications handle remote and local notifications in almost exactly the same fashion. When an application is running, the `ReceivedLocalNotification` method or the `ReceivedRemoteNotification` method on the app delegate will be invoked with the notification information.

There are many ways an application can handle a notification. For instance, the application could just display an alert to remind users about some event. Or the notification might be used to modify the user interface by displaying a special image or changing some text. The following code shows how to handle a local notification and display an alert and reset the badge number to zero:

```
public override void ReceivedLocalNotification(UIApplication application,
UILocalNotification notification)
{
        // show an alert
        new UIAlertView(notification.AlertAction, notification.AlertBody,
null, "OK", null).Show();

        // reset our badge
        UIApplication.SharedApplication.ApplicationIconBadgeNumber = 0;
}
```

If the application is not running, iOS will play the sound and/or update the icon badge as applicable. When the user starts up the application associated with the alert, the application will launch and the `FinishedLaunching` method on the app delegate will be called and the notification information will be passed in via the `options` parameter. If the options dictionary contains the key `UIApplication.LaunchOptionsLocalNotificationKey`, then the app delegate knows that the application was launched from a local notification. The following code snippet demonstrates this process:

```
// check for a notification
if (options != null)
{
        // check for a local notification
        if
(options.ContainsKey(UIApplication.LaunchOptionsLocalNotificationKey))
        {
                var localNotification =
        options[UIApplication.LaunchOptionsLocalNotificationKey] as
        UILocalNotification;
                if (localNotification != null)
                {
                        new UIAlertView(localNotification.AlertAction,
                localNotification.AlertBody, null, "OK", null).Show();
                        // reset our badge
                        UIApplication.SharedApplication.ApplicationIconBadge
                Number = 0;
                }
        }
```

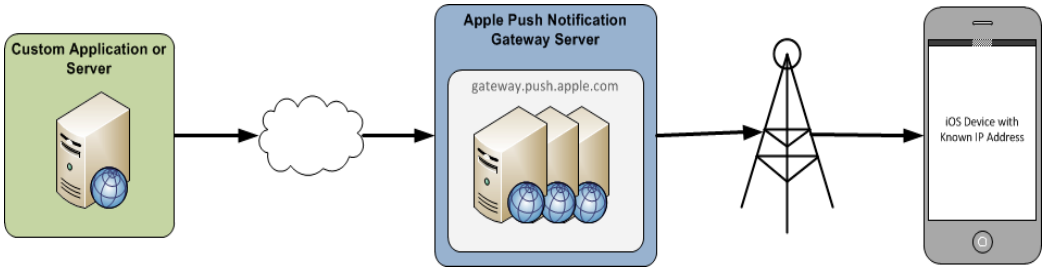
```
}
```

If it's a remote notification you still pull the object from the options dictionary; however, you use the `LaunchOptionsRemoteNotificationKey`, and the resulting object is an `NSDictionary` object with the remote notification payload. You can extract the notification payload via the `alert`, `badge`, and `sound` keys. The following code snippet shows how to get remote notifications:

```
NSDictionary remoteNotification =
options[UIApplication.LaunchOptionsRemoteNotificationKey];
if(remoteNotification != null)
{
        // code. To get the alert use remoteNotification["alert"], etc.
}
```

Implementing Push Notifications

At the center of push notifications in iOS is the *Apple Push Notification Gateway Service* (*APNS*). This is a service provided by Apple that is responsible for routing notifications from an application server to iOS devices. The following image shows an example of the push notification topology for iOS:



Remote notifications themselves are JSON formatted strings that adhere to the format and protocols specified in The Notification Payload section of the Local and Push Notification Programming Guide in the iOS developer documentation.

Restrictions and Limitations

Push notification must observe the following rules that are dictated by the architecture of APNS:

- **256kB message limit – The** entire message size of the notification must not exceed 256kB.
- **No confirmation of receipt –** APNS does not provide the sender with any notification that a message made it to the intended recipient. If the device is unreachable and multiple sequential notifications are sent, all notifications except the most recent will be lost. Only the most recent notification will be delivered to the device.
- **Each application requires a secure certificate –** Communication with APNS must be done over SSL.

Push notifications should be kept brief and only contain enough data to help the mobile application contact the server application for an update. For example, when new e-mail arrives, the server application would only notify the mobile application that new e-mail has arrived. The notification would not contain the new

e-mail itself. The mobile application would then retrieve the new e-mails from the server when it was appropriate.

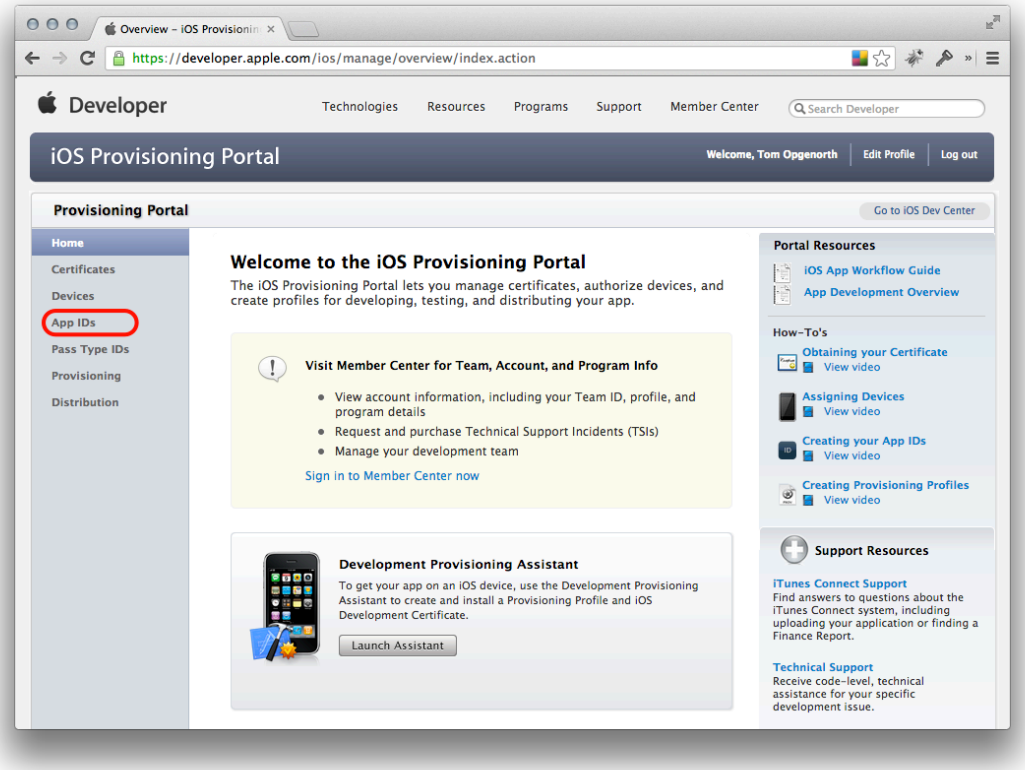
Sandbox and Production Environments

Apple maintains two environments of APNS: a *Sandbox* and a *Production* environment. The Sandbox environment is meant for testing during the development phase and can be found at `gateway.sandbox.push.apple.com` on TCP port 2195. The Production environment is meant to be used in applications that have been deployed and can be found at `gateway.push.apple.com` on TCP port 2195.

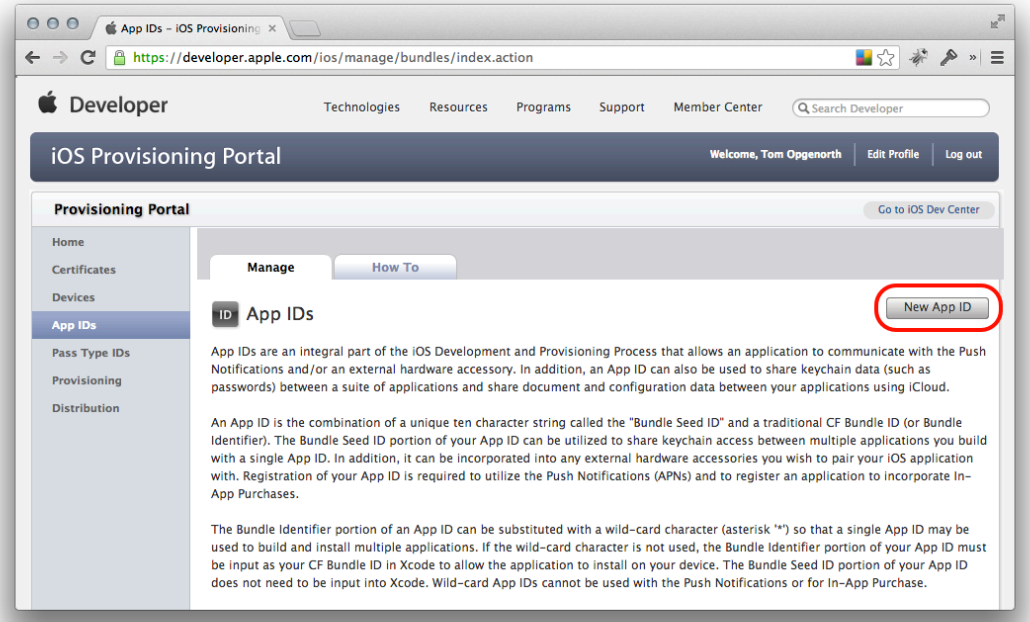
Creating and Using Certificates

Each of the environments mentioned in the previous section require their own certificate. This section will cover how to create a certificate, associate it with a provisioning profile, and then get a Personal Information Exchange certificate for use with PushSharp.

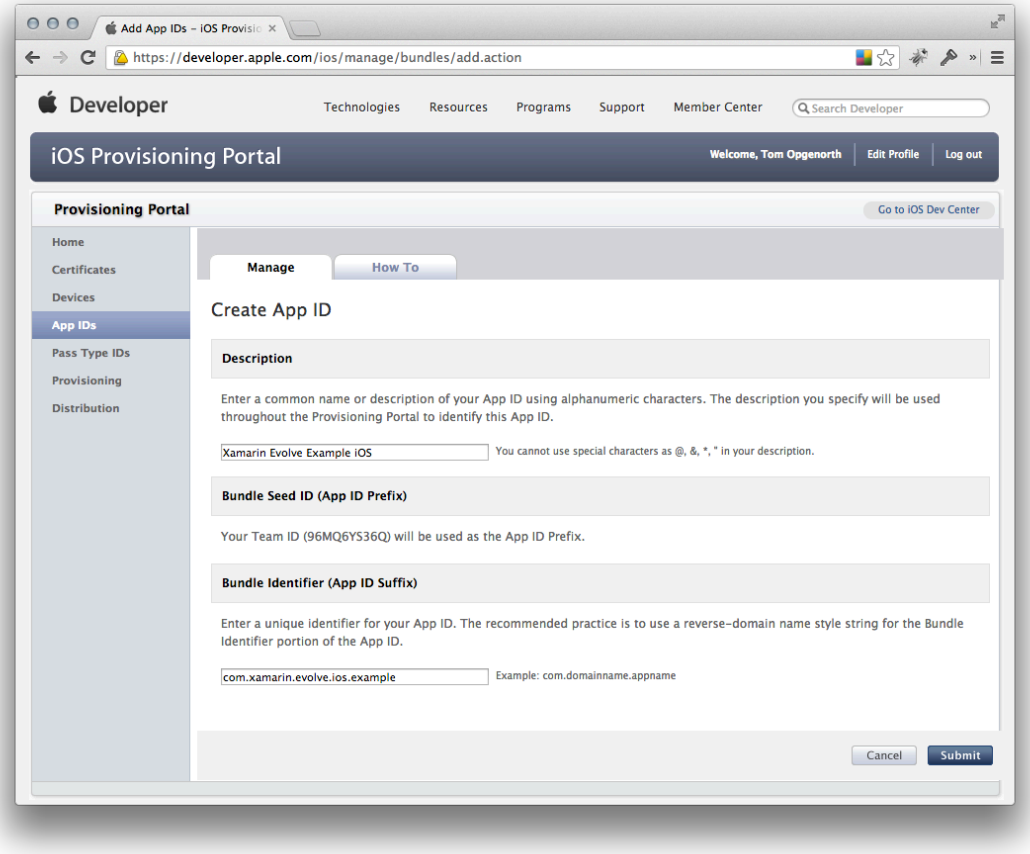
1. To create a certificates go to the **iOS Provisioning Portal** on Apple's website, as shown in the following screenshot (notice the App IDs menu item on the left):



2. Next navigate to the App ID's section and create a new app ID as shown in the following screenshot:

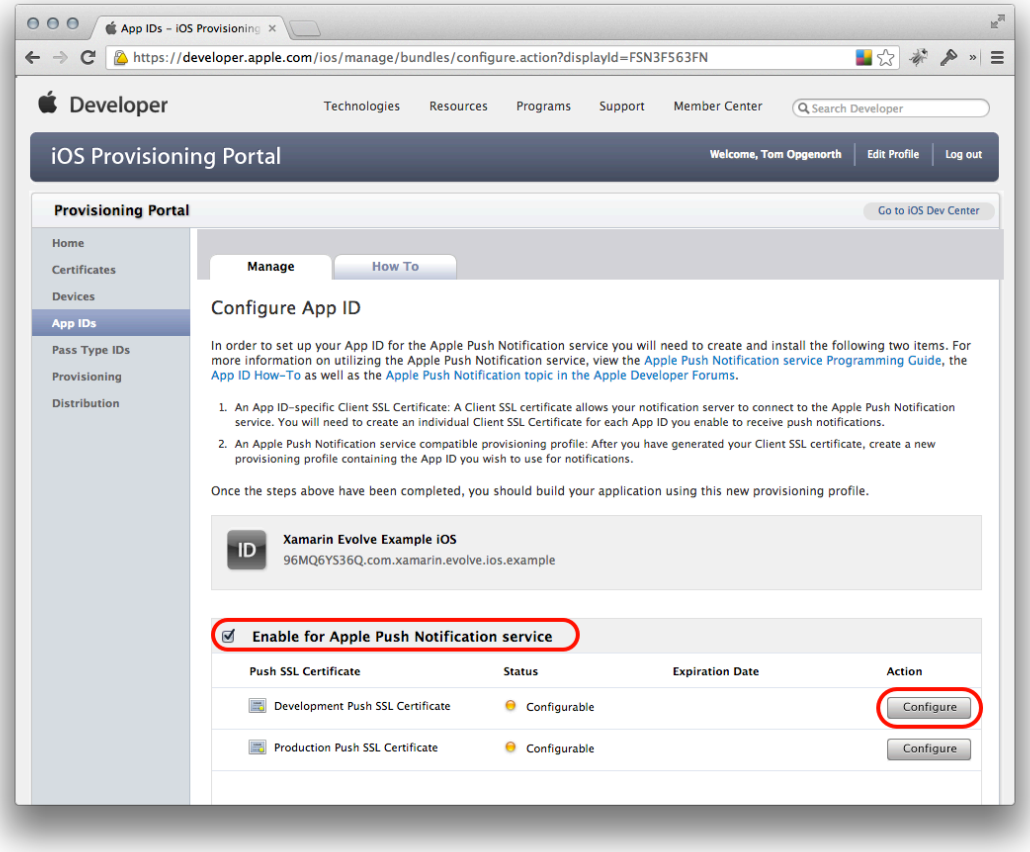


When you click on the **New App ID button**, you will be able to enter the description and a Bundle Identifier for the app ID, as shown in the next screenshot:



Make sure that you don't end the Bundle Identifier with a *. This will create an identifier that is good for multiple applications, and push notification certificates must be for a single application.

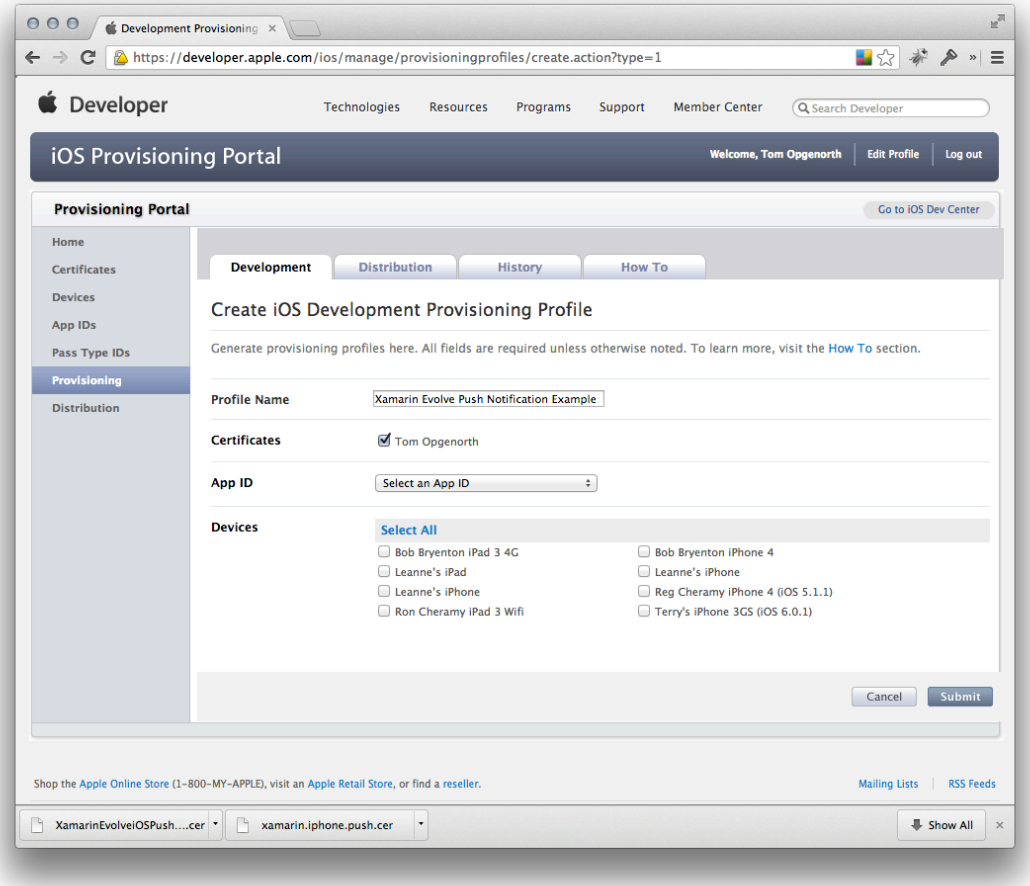
3. Next you must create the certificate for the app ID. Check the Enable for Apple Push Notification Service for the environment you want, as shown in the following screenshot:



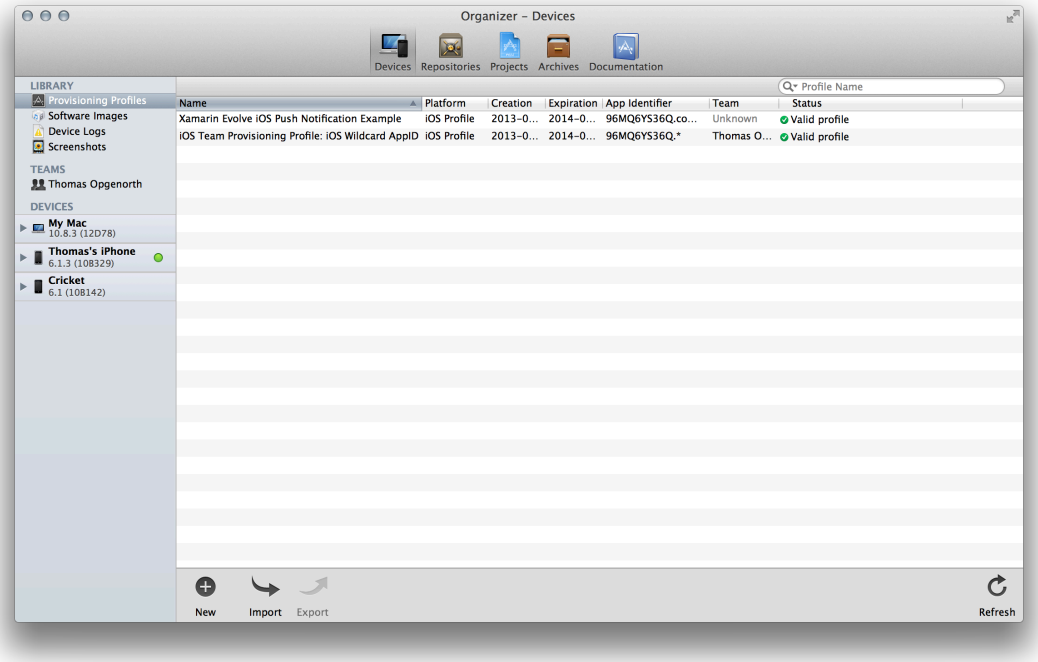
This will launch a wizard that will take you through the process of creating a *Certificate Signing Request* using the **Keychain Access** application on your Mac.

4. Now that the certificate has been created, it must be used as part of the build process to sign the application so that it may register with APNS. This requires creating and installing a provisioning profile that uses the certificate.

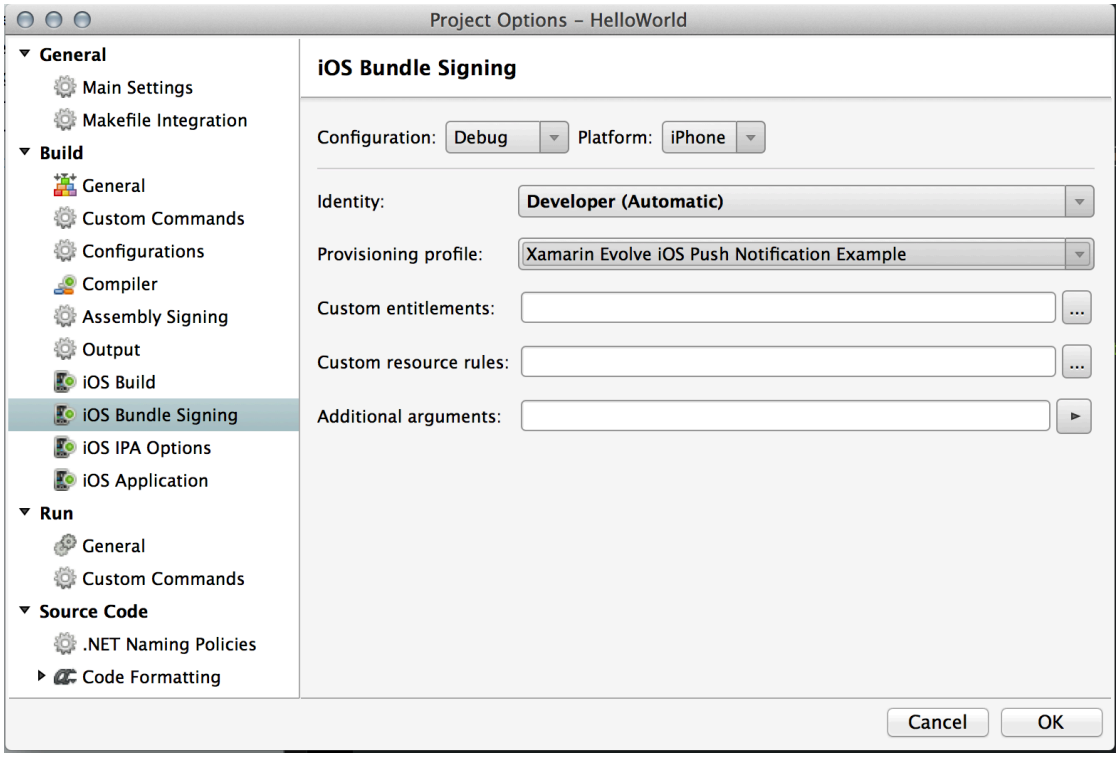
5. To create a provisioning profile, navigate to the iOS Provisioning Portal as shown in the following screenshot:



6. Once you've created the provisioning profile, open up **Xcode Organizer** and refresh it. If the provisioning profile you created does not appear it may be necessary to download the profile from the iOS Provisioning Portal and manually import it. The following screen shot shows an example of the Organizer with the provision profile added:



7. At this point we need to configure the Xamarin.iOS project to use this newly created provisioning profile. This is done from **Project Options** dialog, under **iOS Bundle Signing** tab, as showing in the following screenshot:



At this point the application is configured to work with push notifications. However, there are still a few more steps required with the certificate. This certificate is in DER format that is not compatible with PushSharp, which requires a Personal Information Exchange (PKCS12) certificate. To convert the certificate so that it is usable by PushSharp, perform these final steps:

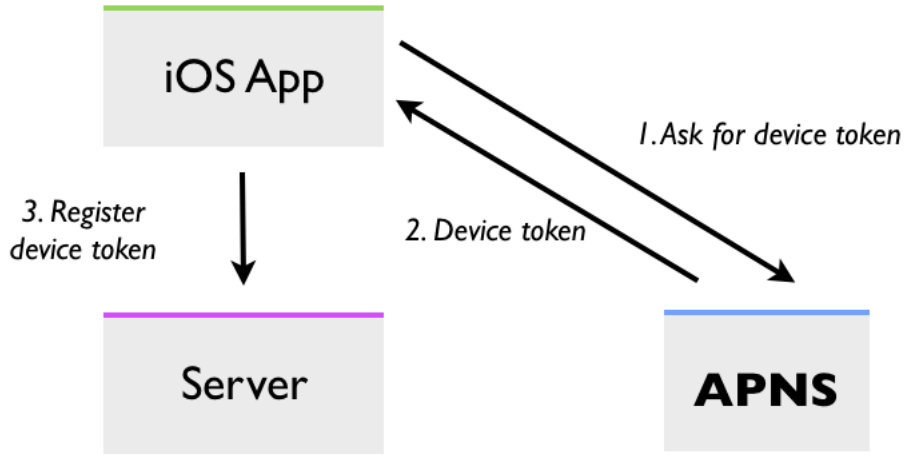
8. **Download the certificate file** – Login to the iOS Provisioning Portal, chose the Certificates tab, select the certificate associated with the correct provisioning profile and chose **Download**.
9. **Open Keychain Access**.
10. **Import the Certificate** – If the certificated isn't already installed, click the **+** button, navigate to the certificate, and select it.
11. **Export the Certificate** – Expand the certificate so the associated private key is visible, selected both and right-click on the selection and chose Export. You will be prompted for a filename and a password for the exported file.

At this point we are done with certificates. We have a certificate that will be used to sign iOS applications and another certificate that can by the application server to communicate with APNS. Next let's look at how iOS applications interact with APNS.

Registering with APNS

Before an iOS application can receive remote notification it must register with APNS. APNS will generate a unique device token and return that to the iOS application. The iOS application will then take the device token and then register itself with the application server.

In theory the device token may change each time an iOS application registers itself with APNS, however in practice this does not happen that often. As an optimization an application may cache the most recent device token and only update the application server when it does change. The following diagram illustrates the process of registration and obtaining a device token:



Registration with APNS is handled in the `FinishedLaunching` method of the application delegate class by calling `RegisterForRemoteNotificationTypes` on the current `UIApplication` object. When an iOS application registers with APNS, it must also specify what types of remote notifications it would like to receive. These remote notification types can be found in the enumeration `UIRemoteNotificationType`. The following code snippet is an example of how an iOS application can register to receive remote alert and badge notifications:

```
UIRemoteNotificationType notificationTypes =
UIRemoteNotificationType.Alert | UIRemoteNotificationType.Badge;
UIApplication.SharedApplication.RegisterForRemoteNotificationTypes
(notificationTypes);
```

The APNS registration request happens in the background – when the response is received, iOS will call the method `RegisteredForRemoteNotifications` in the application delegate class and pass the registered device token. The token will be contained in an `NSData` object. The following code snippet shows how to retrieve the device token that APNS provided:

```
public override void RegisteredForRemoteNotifications (UIApplication
application, NSData deviceToken)
{
        this._deviceToken = deviceToken.ToString();
        // code to register with your server application goes here
}
```

If the registration fails for some reason (such as the device is not connected to the Internet), iOS will call `FailedToRegisterForRemoteNotifications` on the application delegate class. The following code snippet shows how to display an alert to the user informing them that the registration failed:

```
public override void FailedToRegisterForRemoteNotifications (UIApplication
application , NSError error)
{
        new UIAlertView("Error registering push notifications",
        error.LocalizedDescription, null, "OK", null).Show();
}
```

Device Token Housekeeping

Device tokens will expire or change over time. Because of this fact, server applications will need to do some house cleaning and purge these expired or changed tokens. When an application sends as push notification to a device that has an expired token, APNS will record and save that expired token. Servers may then query APNS to find out what tokens have expired.

APNS provides a service known as the *Feedback Service* – an HTTPS endpoint that authenticates via the certificate that was created to send push notifications and sends back data about what tokens have expired. The Feedback Service section of the iOS Developer Guide contains detailed information about this service and its data.

Once again, PushSharp helps by simplifying interaction with the Feedback Service. It provides a `FeedbackService` class that makes easy for server applications to be informed about expired tokens. The following code snippet shows how to setup the a `FeedbackService` instance:

```
FeedbackService service = new FeedbackService(sandbox, p12Filename,
p12FilePassword); service.Feedback += new
FeedbackService.OnFeedback(service_Feedback);
service.Run();
```

The `FeedbackService` needs a Personal Information Exchange (PKCS12) certificate that was created in the previous section. The `OnFeedback` event handler is called for each expired device token and is passed a `FeedbackObject`. The token itself is available via the `DeviceToken` property of the `FeedbackObject`.

Walkthrough: Local Notifications in iOS

Let create a simple application that will show local notifications in action. This will be a simple application that has a single button on it. When we click on the button, it will create a local notification. After the specified time period has elapsed, we will see the notification appear.

1. In Xamarin Studio, create a new iPhone solution and call it `Notifications`.
2. Edit the class `MyViewController`, and change the event hander for `_button.TouchUpInside` to the following:

```
_button.TouchUpInside += (sender, e) =>
{
        //---- create the notification
         var notification = new UILocalNotification();

        //---- set the fire date (the date time in which it will fire)
        notification.FireDate = DateTime.Now.AddSeconds(60);

        //---- configure the alert stuff
        notification.AlertAction = "View Alert";
        notification.AlertBody = "Your one minute alert has fired!";

        //---- modify the badge
        notification.ApplicationIconBadgeNumber = 1;

        //---- set the sound to be the default sound
        notification.SoundName = UILocalNotification.DefaultSoundName;

        //---- schedule it
        UIApplication.SharedApplication.ScheduleLocalNotification(notificat
ion);
};
```

This code will create a notification that uses a sound, sets the value of the icon badge to 1, and displays an alert to the user.

3. Next edit the file `AppDelegate.cs`, and add override the method:

```
public override void ReceivedLocalNotification(UIApplication
application, UILocalNotification notification)
{
        // show an alert
        new UIAlertView(notification.AlertAction,
notification.AlertBody, null, "OK", null).Show();
```

```
        // reset our badge
        UIApplication.SharedApplication.ApplicationIconBadgeNumber
= 0;
}
```

4. Finally, we need to handle the case where the notification was launched because of a local notification. Edit the method `FinishedLaunching` in the `AppDelegate` to include the following snippet of code:

```
// check for a notification
if (options != null)
{
        // check for a local notification
        if
(options.ContainsKey(UIApplication.LaunchOptionsLocalNotificationKey))
        {
                var localNotification =
        options[UIApplication.LaunchOptionsLocalNotificationKey] as
        UILocalNotification;
                if (localNotification != null)
                {
                        new UIAlertView(localNotification.AlertAction,
                localNotification.AlertBody, null, "OK", null).Show();
                        // reset our badge
                        UIApplication.SharedApplication.ApplicationIconBadge
                Number = 0;
                }
        }
}
```

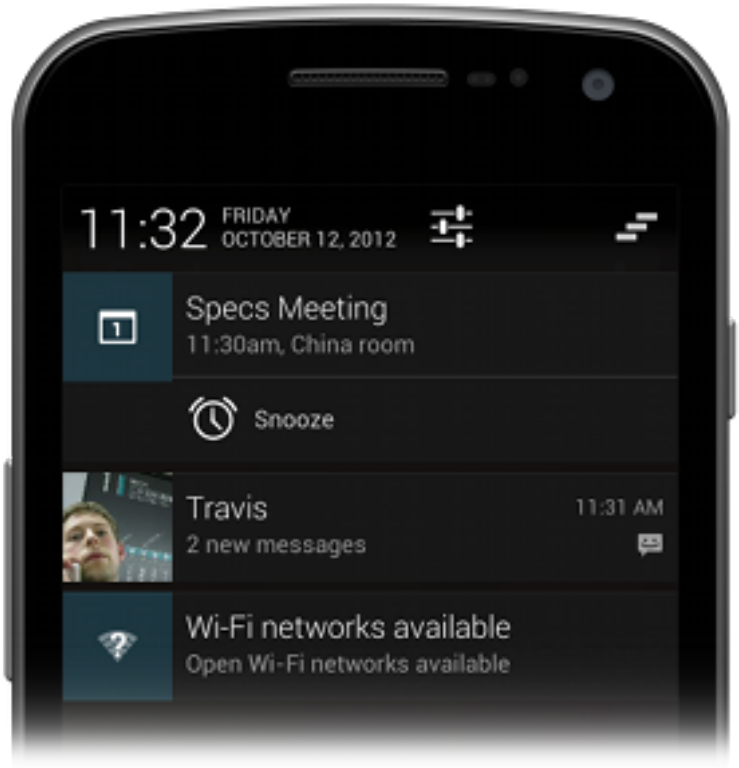
5. Next, run the application, and click the Add notification button. After a short pause you should see the alert dialog, as shown in the following screenshots:

At this point we've finished the section on local notifications in iOS. Lets move on to the next section on notifications in Android.

Notifications in Android

Android provides two system-controlled areas that will display notifications and notification information to the user. When a notification is first published, it will be displayed in the *notification area*, as show in the following screenshot:

To obtain details about the notification, the user will open the *notification drawer*, which will expand each notification icon with details and possibly actions that may be performed as a result of the notification. The following screenshot shows a notification drawer that corresponds to the notification area that was previously displayed:

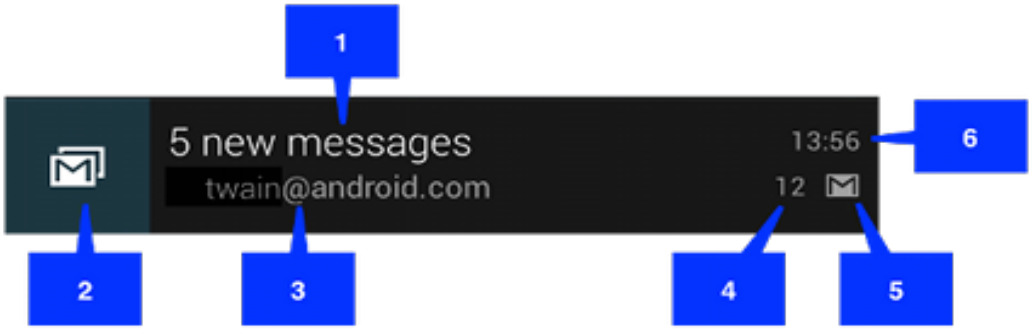


In addition to the container area and container drawer that were displayed above, Android can also vibrate the device or make an audible sound when a notification has been received.

Just like with Android, iOS has three ways to indicate to the user that a notification has been received:

- Sound or Vibration – iOS can play a sound to notify users. If the sound is turned off, the device can be vibrated as well.
- Alerts – it is possible to display a dialog on the screen with information about the notification.
- Badges – when a notification is published, it is possible to display an image or a number on the application icon.

While conceptual Android notifications are identical to iOS notifications, they are implemented with many more features. Let's examine some of the parts that a user may see when looking a notification in the notification drawer:



1. Content Title – This is the title of the notification. All notifications must set the title.
2. Large Icon – This is an optional icon that should represent the sending application or the sender's photo.
3. Content Text – This is some text that provides a summary about the notification. This is a mandatory property.
4. Content Info – This is a small piece of additional information about this notification. Optional
5. Small Icon – This is the small icon that will represent the notification in the notification area. Mandatory.
6. Timestamp – This is the time the notification was issued. Optional.

Starting in Android 4.1, notifications could display a *big view* when the notification is expanded in the notification drawer. The big view allows for more information to be displayed in the notification as shown in the following image:

1. Content Title – This is the title of the notification. All notifications must set the title.

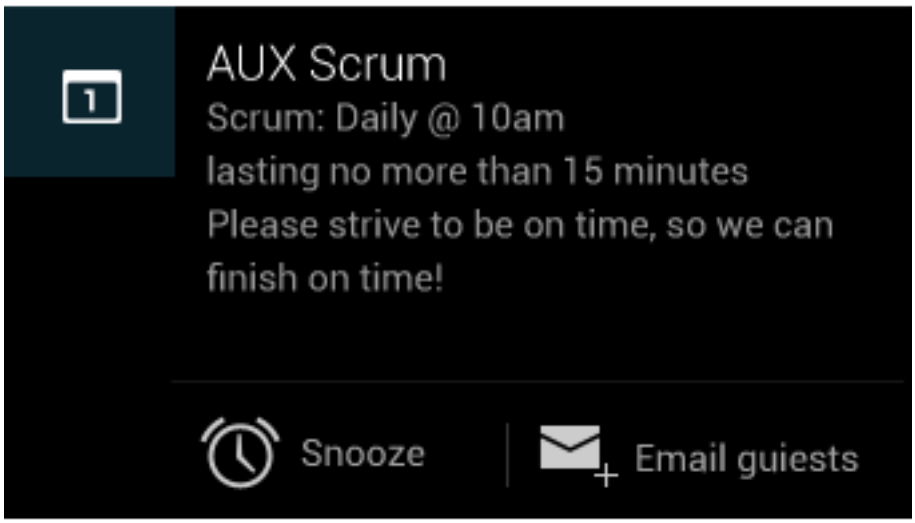
2. Large Icon – This is an optional icon that should represent the sending application or the sender's photo.
3. Content Text – This is some text that provides a summary about the notification. This is a mandatory property.
4. Content Info – This is a small piece of additional information about this notification. Optional
5. Small Icon – This is the small icon that will represent the notification in the notification area. Mandatory.
6. Timestamp – This is the time the notification was issued. Optional.
7. Details Area – What is displayed in the details area depends on the style of the big view.

The big view notification is visually very similar to the normal view with the exception of the details area. The details area has the following styles available:

- Big Picture Style – This style allows the notification to display a bitmap no taller than 256dp.
- Big Text Style – Displays a large text block in the details section
- Inbox Style – This style displays lines of text

Notification icons must be 24x24 density independent pixels in size.

In addition to the data provided above, a notification may optional be provided with an action to take when the user clicks the notification. Typically this action will open an Activity in the application. Android 4.1 expanded on this functionality by allowing applications to add action buttons to perform additional tasks such as creating e-mail or snoozing an alarm. The following image shows an example of one such notification:



Now that we are familiar with the anatomy of notifications in Android, lets move and learn how to create notifications in Android. Although we have mentioned some of the great new features available in Android 4.1, coverage of those API's

will be covered in a future document from Xamarin. This chapter will focus on those notification API's that are available in Android 4.0.3 or lower.

Creating Notifications

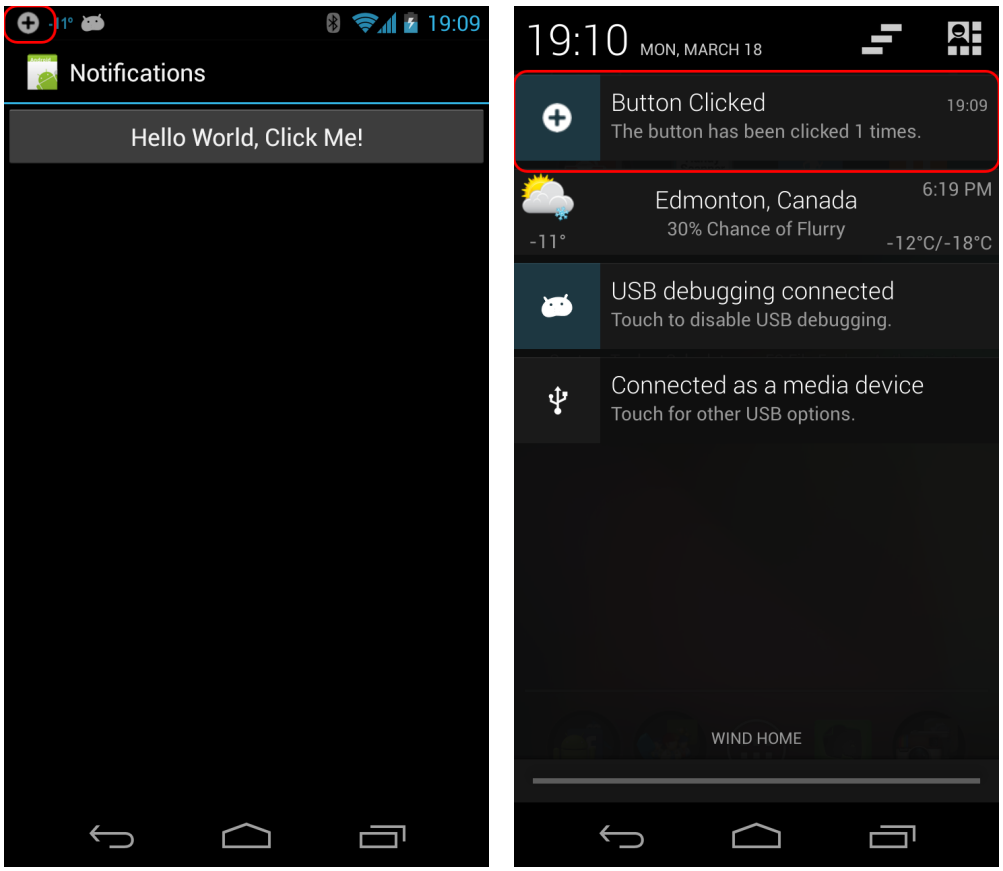
All notifications in Android are represented by an instance of the `Notification` object. To simplify the creation of `Notification` objects, Android 3.0 introduced the `Notification.Builder` class. Older versions of Android can use the `NotificationCompat.Builder` class that is provided by the Android Support Packages. To ensure the best compatibility, applications should use the `NotificationCompat.Builder` class.

Once a notification has been created, it is displayed by using the `NotificationManager.Notify` method. This method takes two parameters – the first is a unique integer that identifies the notification to your application while the second is a `Notification` object.

Notifications can be displayed by any context, such as an Activity or a Service. The following code snippet shows the minimum code required to create and publish a notification that will be immediately published:

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
        .SetContentTitle("Button Clicked")
        .SetSmallIcon(Resource.Drawable.ic_stat_button_click)
        .SetContentText(String.Format("The button has been clicked {0}
times.", _count));

NotificationManager notificationManager =
(NotificationManager)GetSystemService(NotificationService);
notificationManager.Notify(ButtonClickNotificationId, builder.Build());
```

The following two screenshots show how this notification will appear in the notification area and the notification drawer:



Now that we understand the basics of creating notifications, lets take a moment to see how an application may manage its notifications. To update a notification once it has been issued, create or update a `Notification` object as was shown in the previous section, and then call `NotificationManager.Notify` with the same unique notification id.

Notifications remain visible until one of the three things happen:

- The user dismisses the notification either individually or by using “Clear All”.
- The application makes a call to `NotificationManager.Cancel`, passing in the unique notification ID that was assigned when then notification was published.
- The application calls `NotificationManager.CancelAll`.

Starting an Activity from a Notification

It is also possible to start an Activity when the user selects a notification. This is done by creating a *PendingIntent* object and associating that with the notification. A `PendingIntent` is a special type of intent that allows the recipient application to run a predefined piece of code with the permissions of the sending application. When the user clicks on the notification, Android will start up the Activity specified by the `PendingIntent`. The following code snippet shows how to create the `PendingIntent` and add it to the notification:

```
Intent resultIntent = new Intent(this, typeof(SecondActivity));
```

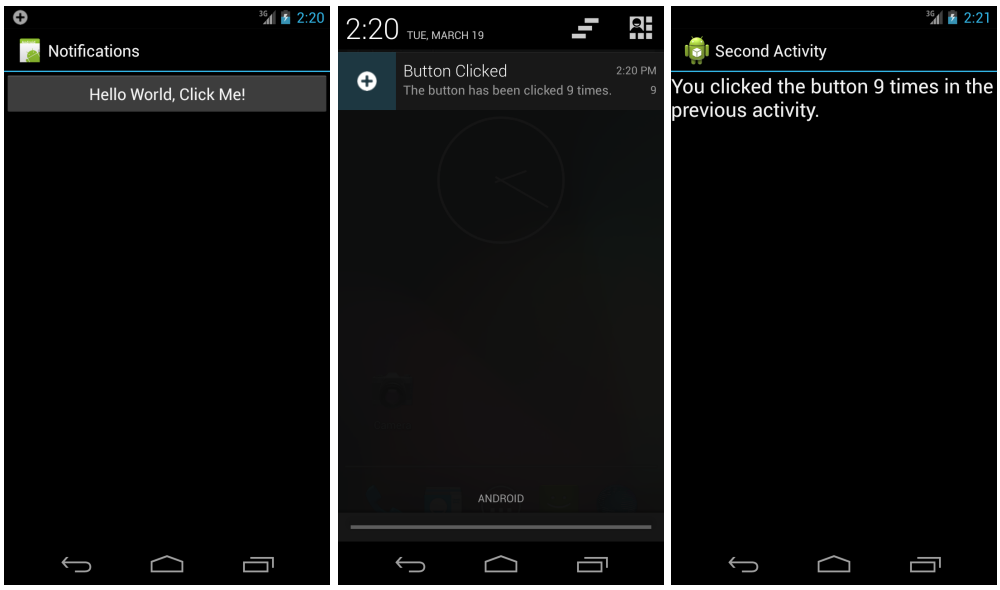
```
resultIntent.PutExtras(valuesForActivity); // Pass some values to
SecondActivity.

TaskStackBuilder stackBuilder = TaskStackBuilder.Create(this);
stackBuilder.AddParentStack(Class.FromType(typeof(SecondActivity)));
stackBuilder.AddNextIntent(resultIntent);

PendingIntent resultPendingIntent = stackBuilder.GetPendingIntent(0,
(int)PendingIntentFlags.UpdateCurrent);

// Build the notification
NotificationCompat.Builder builder = new NotificationCompat.Builder(this)
        .SetAutoCancel(true) // dismiss the notification from the
notification area when the user clicks on it
        .SetContentIntent(resultPendingIntent) // start up this activity
when the user clicks the intent.
        .SetContentTitle("Button Clicked") // Set the title
        .SetNumber(_count) // Display the count in the Content Info
        .SetSmallIcon(Resource.Drawable.ic_stat_button_click) // This is
the icon to display
        .SetContentText(String.Format("The button has been clicked {0}
times.", _count)); // the message to display.

// Finally publish the notification
NotificationManager notificationManager =
(NotificationManager)GetSystemService(NotificationService);
notificationManager.Notify(ButtonClickNotificationId, builder.Build());
```

This code is very similar to the code in the previous section, except that a `PendingIntent` is added to the `Notification` by first creating the `PendingIntent`, and then adding it to the notification by calling `NotificationCompat.Builder.SetContentIntent`.

Now that we understand the basics of notifications, lets see this working together in a sample application.

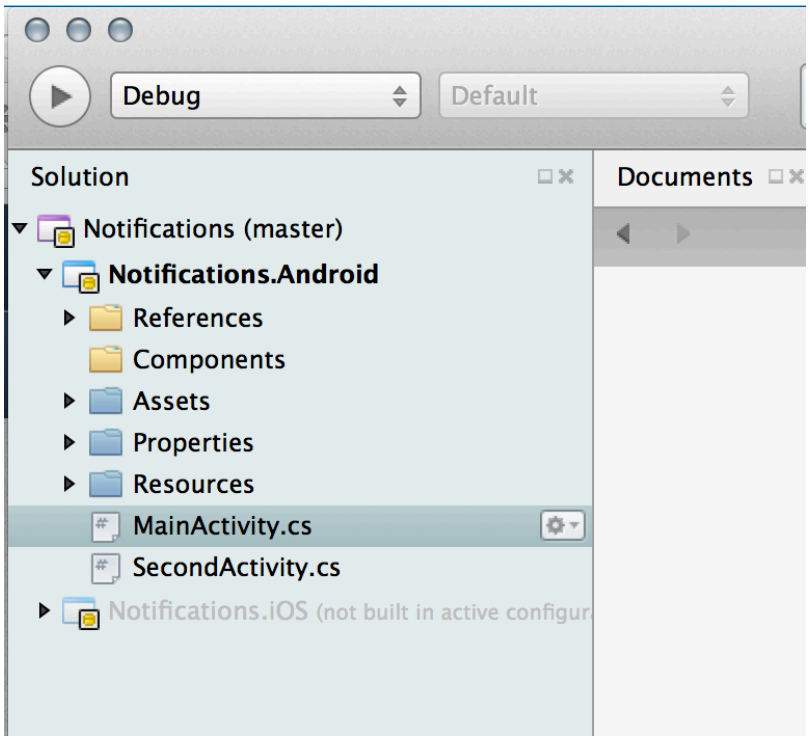
Walkthrough: Local Notifications in Android

Lets walk through a simple example in Android. This application will raise a notification when the user clicks on a button. When the user clicks on the notification it will start up a second activity in the application and display the number of times the user clicked the button in the first screen. The following screenshots show the application:



To help get you started, there already a project stubbed in with some drawable resources added, targeting Android 4.0.3 and already has a reference to the Android Support Packages added to it.

Open the `Notifications` solution in Xamarin Studio. Ensure that the startup project is set to **Notifications.Android** and that the build configuration is set to **Debug** as shown in the following screenshot:



With the solution now open, lets get started:

1. First we will need a unique ID for our notification. Edit the class `MainActivity`, and add the following static instance variable:

```
private static readonly int ButtonClickNotificationId = 1000;
```

2. Next we need an event handler for the Click event of the button that will handle creating and publishing the notification. Add the following method to `MainActivity`:

```
private void ButtonOnClick(object sender, EventArgs eventArgs)
{
        // These are the values that we want to pass to the next activity
        Bundle valuesForActivity = new Bundle();
        valuesForActivity.PutInt("count", _count);

        // Create the PendingIntent with the back stack
        // When the user clicks the notification, SecondActivity will start
up.
        Intent resultIntent = new Intent(this, typeof(SecondActivity));
        resultIntent.PutExtras(valuesForActivity); // Pass some values to
SecondActivity.

        TaskStackBuilder stackBuilder = TaskStackBuilder.Create(this);
        stackBuilder.AddParentStack(Class.FromType(typeof(SecondActivity)))
;
        stackBuilder.AddNextIntent(resultIntent);

        PendingIntent resultPendingIntent = stackBuilder.GetPendingIntent(0,
(int)PendingIntentFlags.UpdateCurrent);

        // Build the notification
        NotificationCompat.Builder builder = new
NotificationCompat.Builder(this)
                .SetAutoCancel(true) // dismiss the notification from the
        notification area when the user clicks on it
                .SetContentIntent(resultPendingIntent) // start up this
        activity when the user clicks the intent.
                .SetContentTitle("Button Clicked") // Set the title
                .SetNumber(_count) // Display the count in the Content Info
                .SetSmallIcon(Resource.Drawable.ic_stat_button_click) //
        This is the icon to display
                .SetContentText(String.Format("The button has been clicked
        {0} times.", _count)); // the message to display.

        // Finally publish the notification
        NotificationManager notificationManager =
(NotificationManager)GetSystemService(NotificationService);
        notificationManager.Notify(ButtonClickNotificationId,
builder.Build());

        _count++;
}
```

There is a lot going on with this method, so lets take a minute to examine what is going on here. First we will create a Bundle that we will use to store the number of times the button was clicked.

The next thing we do is to create the `PendingIntent`. We first create a regular intent that will tell Android which activity we want to start up. We haven't created

this activity yet, but we will very shortly. Then we use a `TaskStackBuilder` to create the `PendingIntent`.

The third step is to create the notification using a `NotificationCompat.Builder` instance.

The final step is to publish the event. We get a reference to the `NotificationManager`, and call `.Notify`. We don't have to worry about managing the `Notification`. The first time it is published, Android will add our `Notification` object to the notification area. The second and subsequent times, Android will update the existing notification.

3. Next, assign the method `ButtonOnClick` to the Click event of the button. Update `MainActivity.OnCreate` and add the following line of code:

```
button.Click += ButtonOnClick;
```

4. Now we need to create another activity that Android will display when the user clicks on our `Notification`. Add another activity named `SecondActivity`, and edit it so that it contains the following code:

```
 [Activity(Label = "Second Activity")]
public class SecondActivity : Activity
{
        protected override void OnCreate(Bundle bundle)
        {
                base.OnCreate(bundle);
                int count = Intent.Extras.GetInt("count", -1);
                if (count <= 0)
                {
                        return;
                }
                SetContentView(Resource.Layout.Second);
                TextView txtView =
        FindViewById<TextView>(Resource.Id.textView1);
                txtView.Text = String.Format("You clicked the button {0}
        times in the previous activity.", count);
        }
}
```

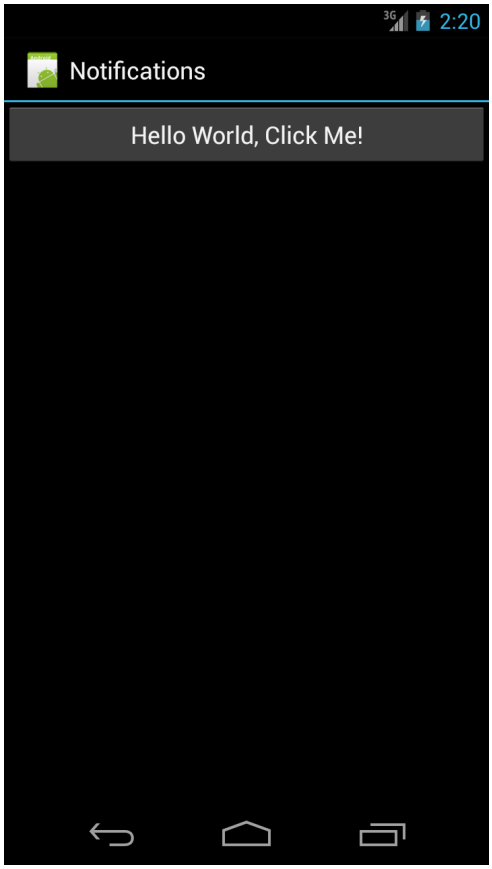
This activity is fairly simple. It will attempt to extra the count from the Intent that initiated it. If it can, then it will display the value of that count in a TextView.

5. Finally, create the file /Reosurces/Layout/Second.axml which will be inflated by SecondActivity. It should have the following contents:

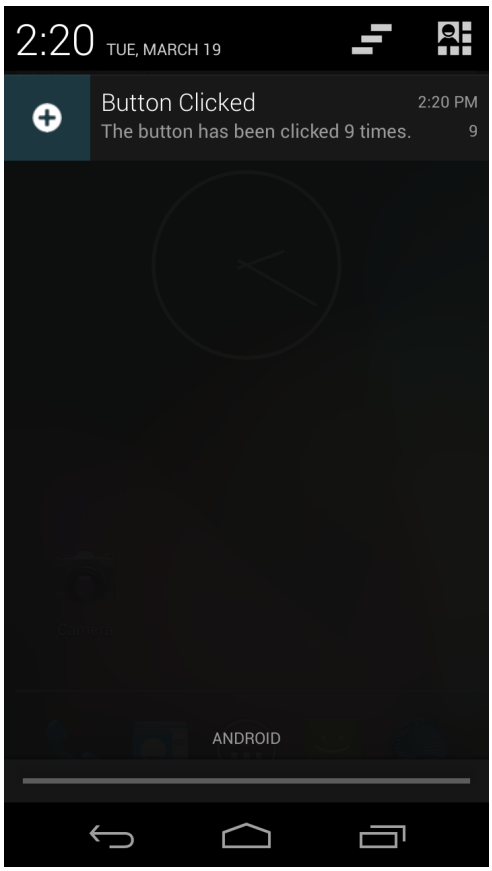
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:minWidth="25px"
        android:minHeight="25px">
        <TextView
                android:text=""
                android:textAppearance="?android:attr/textAppearanceLarge"
                android:layout_width="fill_parent"
```

```
                android:layout_height="wrap_content"
                android:id="@+id/textView1" />
</LinearLayout>
```

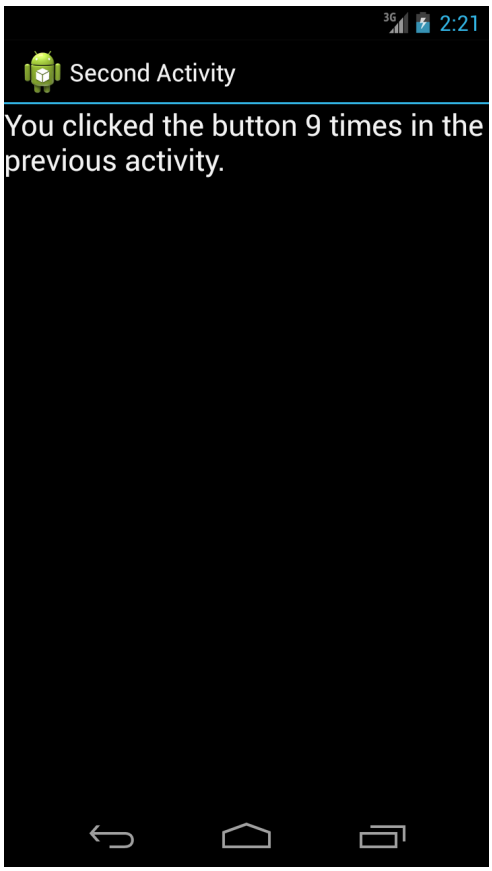
6. Run the application. You should be presented with the first activity, similar to the following screenshot:



As you click the button, you should notice the small icon for the notification appear in the notification area. If you swipe down and expose the notification drawer, then you should see our notification:



Once you click the notification, it should disappear, and our other activity should be launched, looking something like the following screenshot:



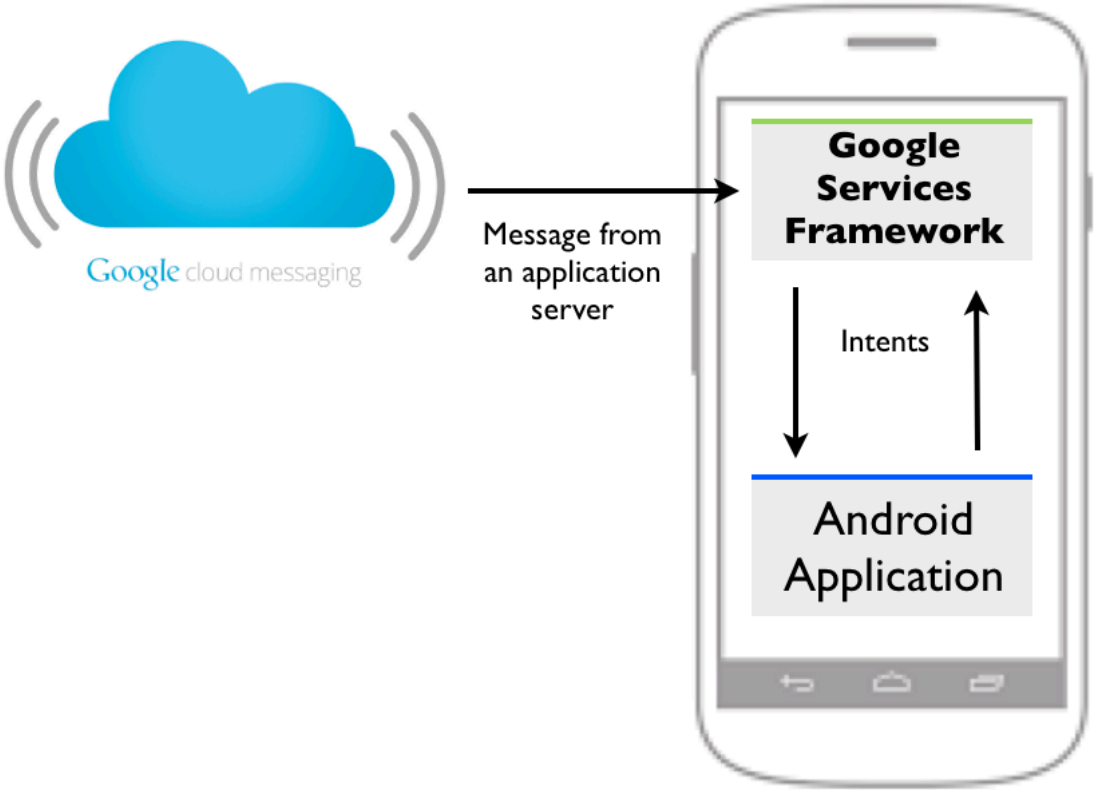
Congratulations! At this point you have completed the Android notification walkthrough and have a working sample you can refer to. There is a lot to notifications than what we have shown here, so if you want more information make sure you go and check out Google's documentation on notifications and the Android Design Guide Patterns on Notifications. Now lets move on to see how remote notifications work in Android.

Remote Notifications in Android

Remote notifications are handled in Android by a free service from Google called *Google Cloud Messaging* (*GCM*). This service handles the sending, routing and queuing of messages from application servers to Android applications. An Android application does receive the messages itself.

Instead, Android applications communicate with the Google Services framework that must be installed on each device. The Google Services framework runs in the background while the Android device is powered, quietly listening for messages from Google Cloud Messaging. When these messages arrive, Google Services framework will deserialize these messages into Intents and broadcast these Intents to application that have registered for them.

You can see the relationships between all of these components and systems in the following diagram:



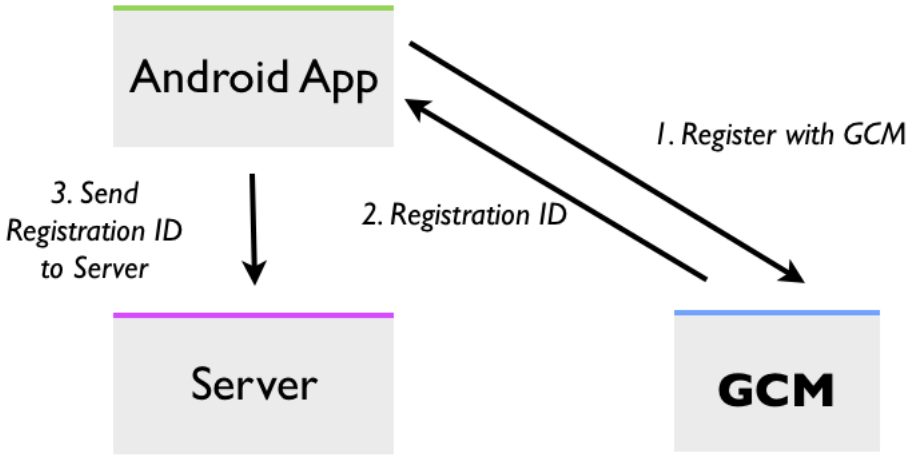
Restrictions and Limitations

In order to use GCM the following restrictions and limitations must be observed:

- 4kb message limit – the entire size of the message cannot exceed 4kb in size.
- Android 2.2 – Android devices must be running Android 2.2 (API level 8) or higher.
- Google Play Store – Android devices must have the Google Play Store installed on the device for GCM to work. You do not have to deploy your application through the Google Play Store for GCM to work. It is possible to test applications on an Android emulator by using an emulator image that has the Google API's installed.
- Google Account – A Google Account is required if the device is running a version of Android lower that 4.0.4.

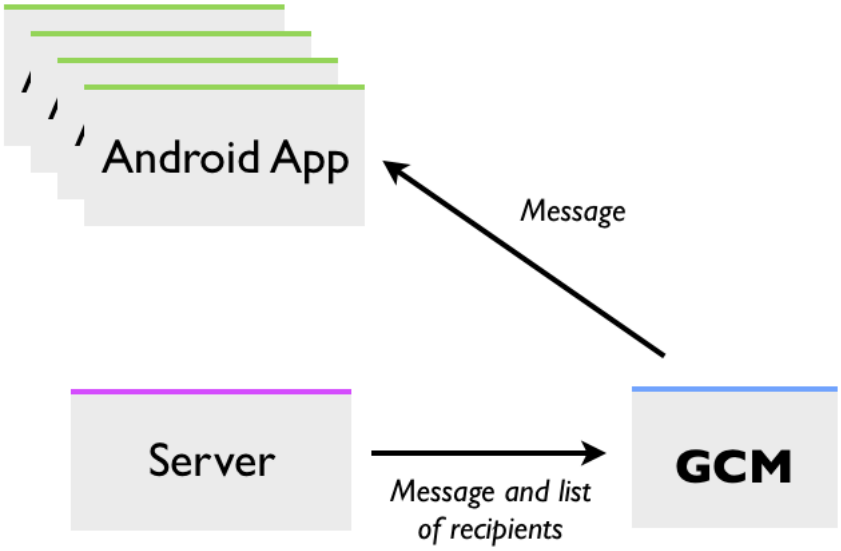
Google Cloud Messaging in Action

To begin with, an Android application must register with GCM. When it does, GCM will return a registration ID that uniquely identifies the Android application running on that specific Android device. It is the responsibility of the Android application to send this registration ID to the server application. Only when the server application receives this registration ID can the registration process be considered complete. The following diagram illustrates this concept:



The registration ID does not change that often, so ever time an application starts up it does not have to go through this process each time it is run. If Google Cloud Message does decided to change the registration ID, it will publish as message announcing that fact. It is the responsibility of the Android application to respond to this notification and to update the server application accordingly.

Now when a server application wishes to publish a notification, it will send a message to GCM, which in turn will send the message to the device. GCM will allow a server application to specify 1000 recipients on a single message. The following diagram shows this:

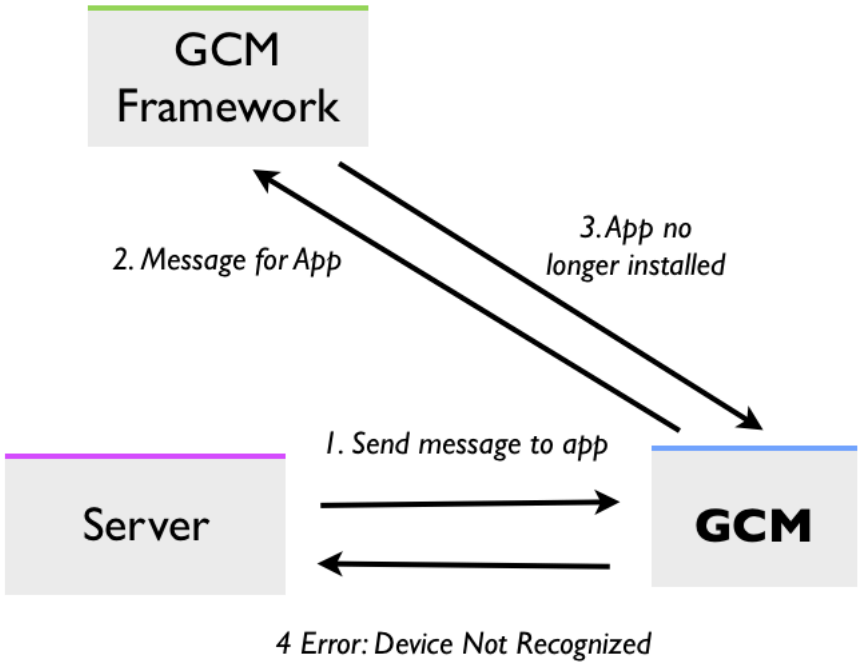


If the server tries to send a message to a device that is offline, GCM will queue the message for delivery when the device comes back online. When the device comes back online, Google Cloud Messaging will only send the most recent message. This prevents obsolete/redundant messages from being sent.

If an application is uninstalled from the application, neither the server application nor Google Cloud Messaging is aware of this fact. Messages will still be sent to the device, however the Google Services Framework that is installed on the device will receive the message and detect that the application is no longer installed.

Instead, Google Services framework message back to Google Cloud Messaging asking it to delete the registration ID.

From this point on, when the application server tries to publish a message for the now defunct registration ID, Google Cloud Messaging will reply back with an error message of "Device Not Registered". It is the responsibility of the server application to remove that registration ID from its list of registered devices and to stop trying to send messages that registration ID. The following diagram illustrates this in action:



There are three tasks that must be performed to use Google Cloud Messaging:

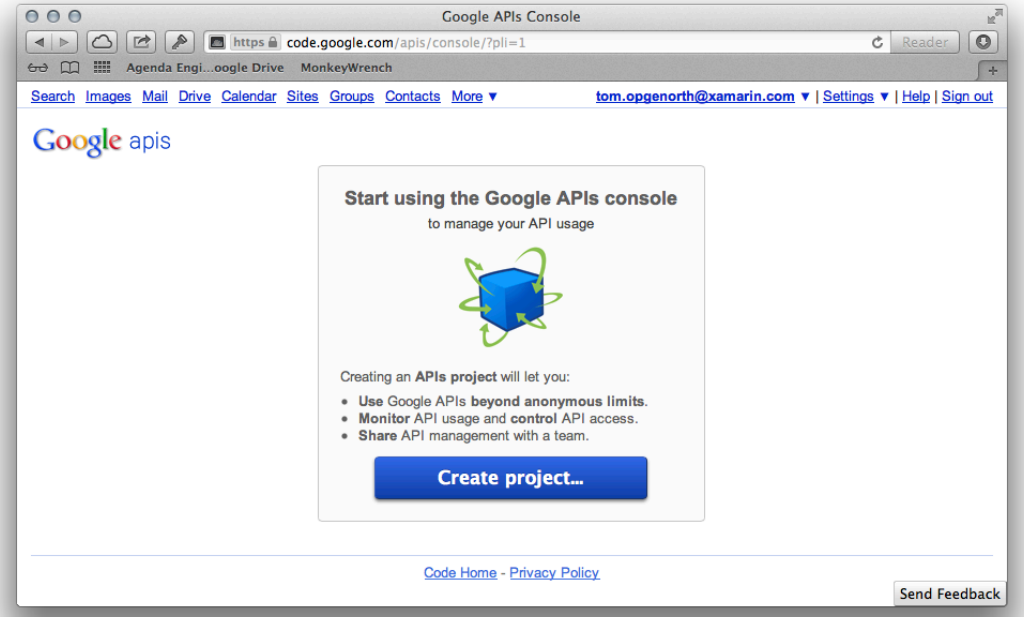
1. Create a Google API project – A Google API project must be created to setup Google Cloud Messaging. The project will contain an API key that server applications must use when publishing messages.
2. Write an Android application – it will be necessary to write an Android application (or modify an existing one) so that it contains the code necessary to communicate with GCM and process incoming messages.
3. Create a Server Application – this is the server application that will publish message to the mobile application. The server application can be on any platform, as long as it has a web API that can be used for tasks such as registering clients, unregistering clients, and possibly downloading data.

Lets look at each of these steps in more detail.

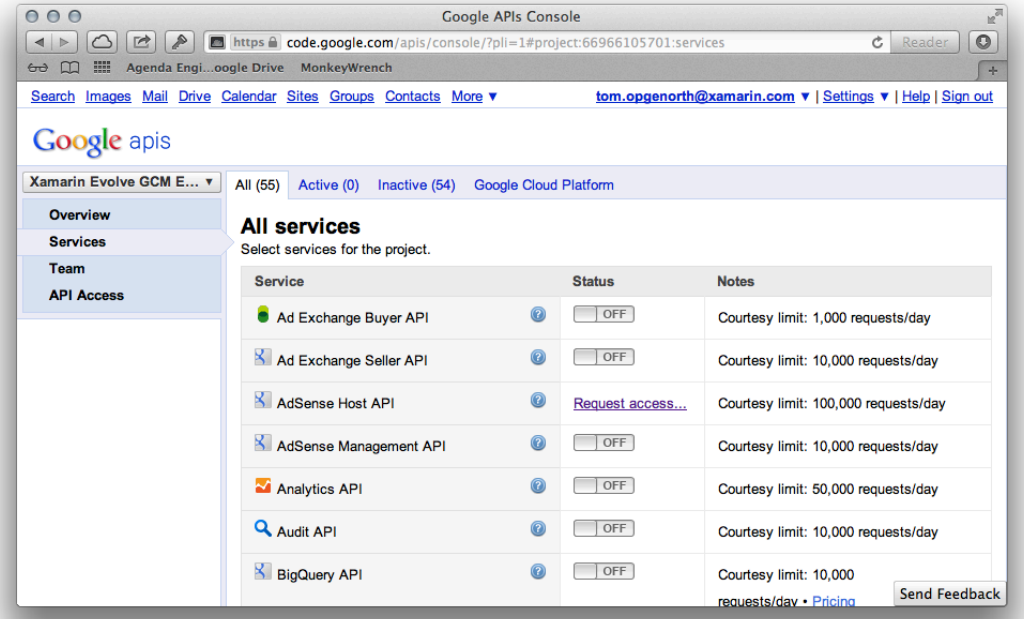
Creating a Google API Project

To get started with Google Cloud Messaging, it is necessary to setup a project with Google and enable the Google Cloud Messaging for Android. The following steps outline how to do so:

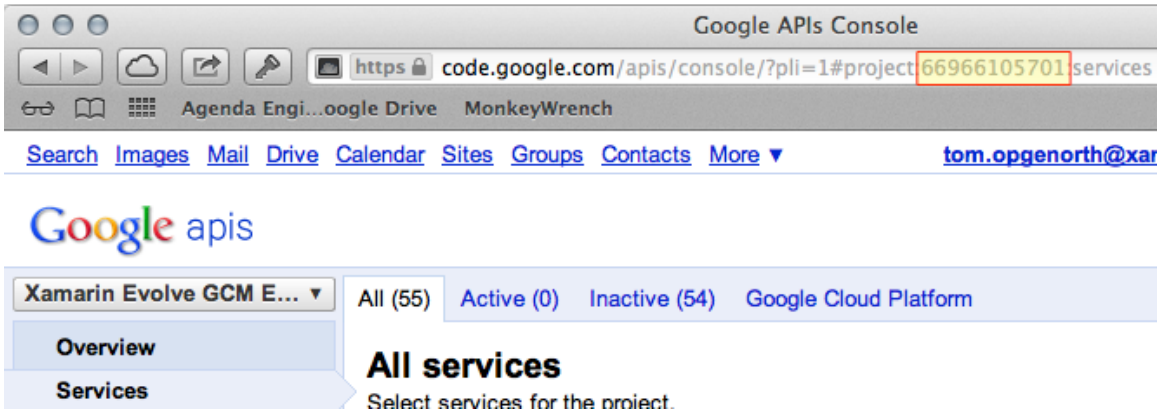
1. Navigate to the Google Developer API Console and create a project. The following screen is an example of what you will see when you visit this page for the first time:



When you create a project, you will be taken to a project page, which will look similar to the following screenshot:

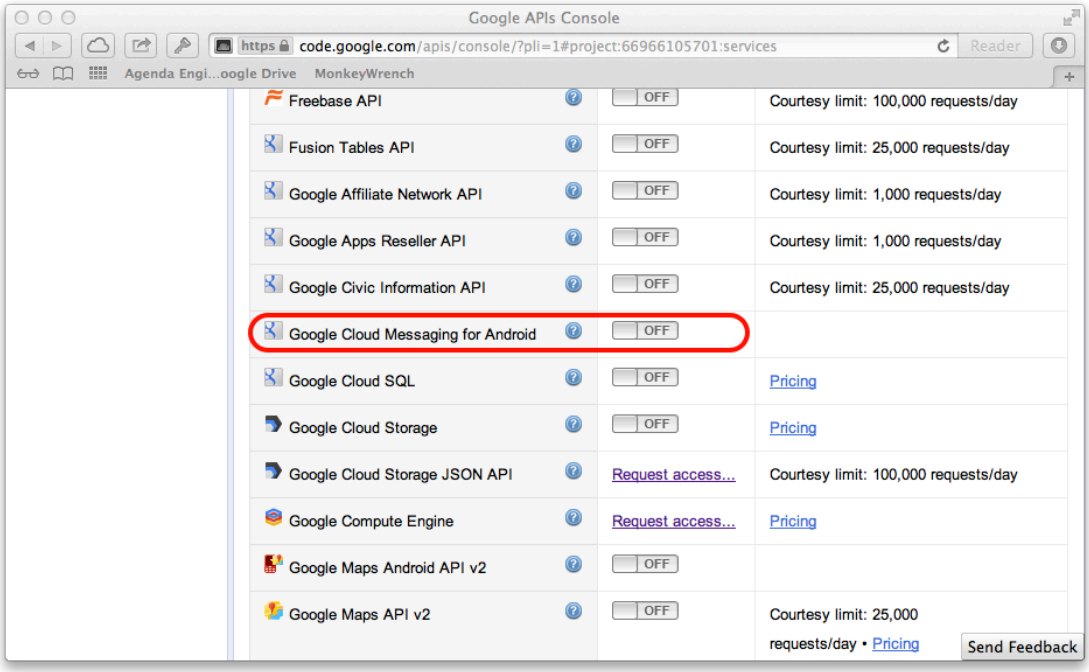


The previous screenshot shows a very long list of services available from Google. As well pay attention the URL for the project (as seen in the next screenshot):



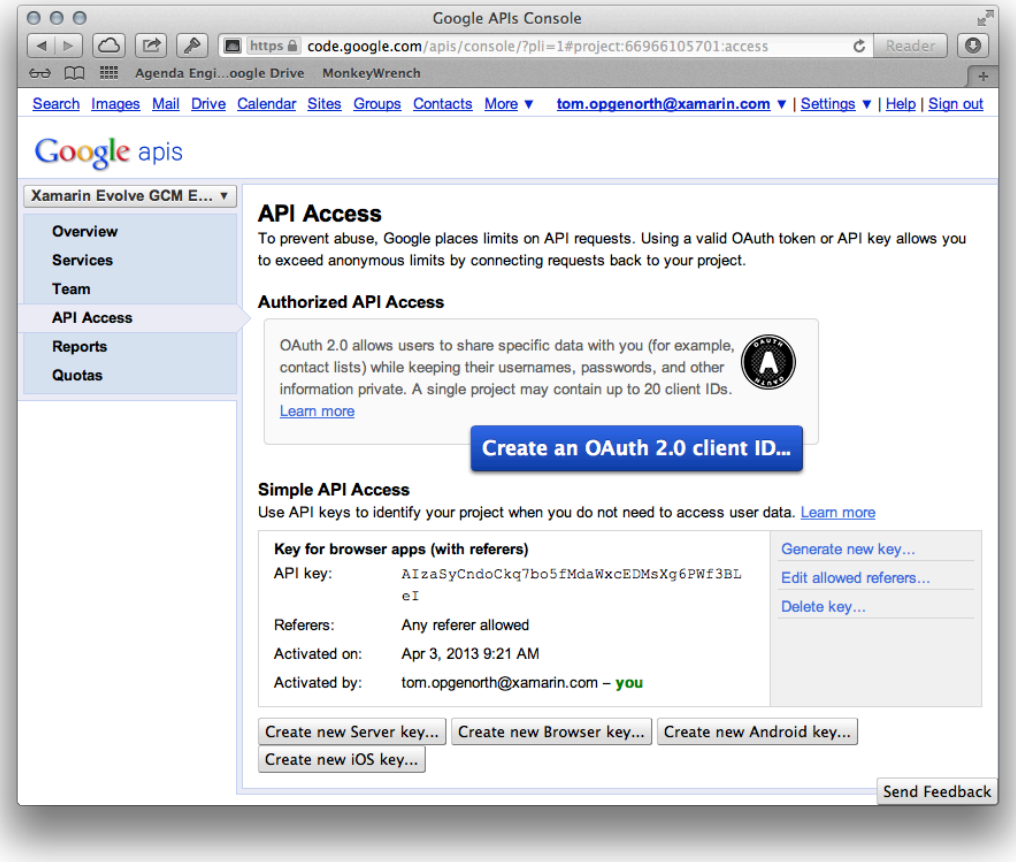
Notice the value on the `#project:` hash, `66966105701` in this case. This is the *project ID*. This is a unique number that identifies your project to the various Google APIs. GCM also refers to this value as the *sender ID*.

2. Next we will need to enable Google Cloud Messaging for the project. Scroll down from the list, until you see Google Cloud Messaging for Android, as shown in the following screenshot:

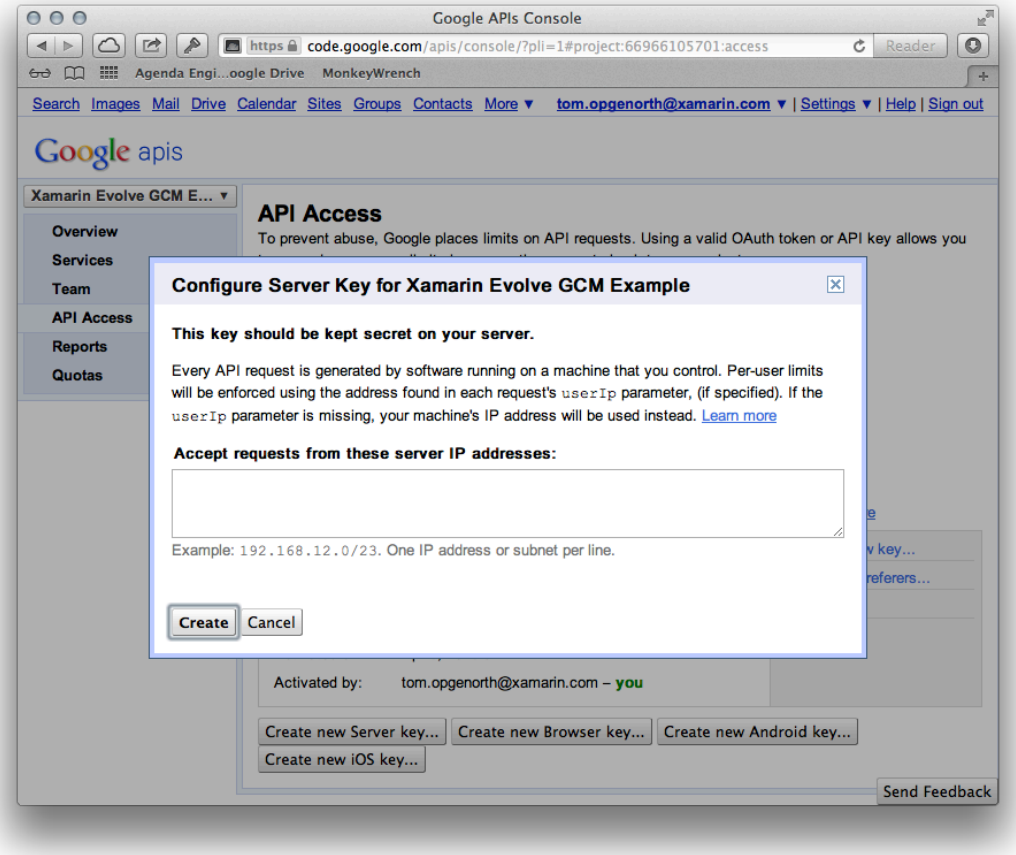


Turn the service on. You will have to consent to the terms of service and user agreements.

3. Now that the project has been created, we need an API key. Server applications will use this API key to identify and authenticate themselves to GCM. Select the API Access item from the menu on the left, and you should see a screen similar to the following:

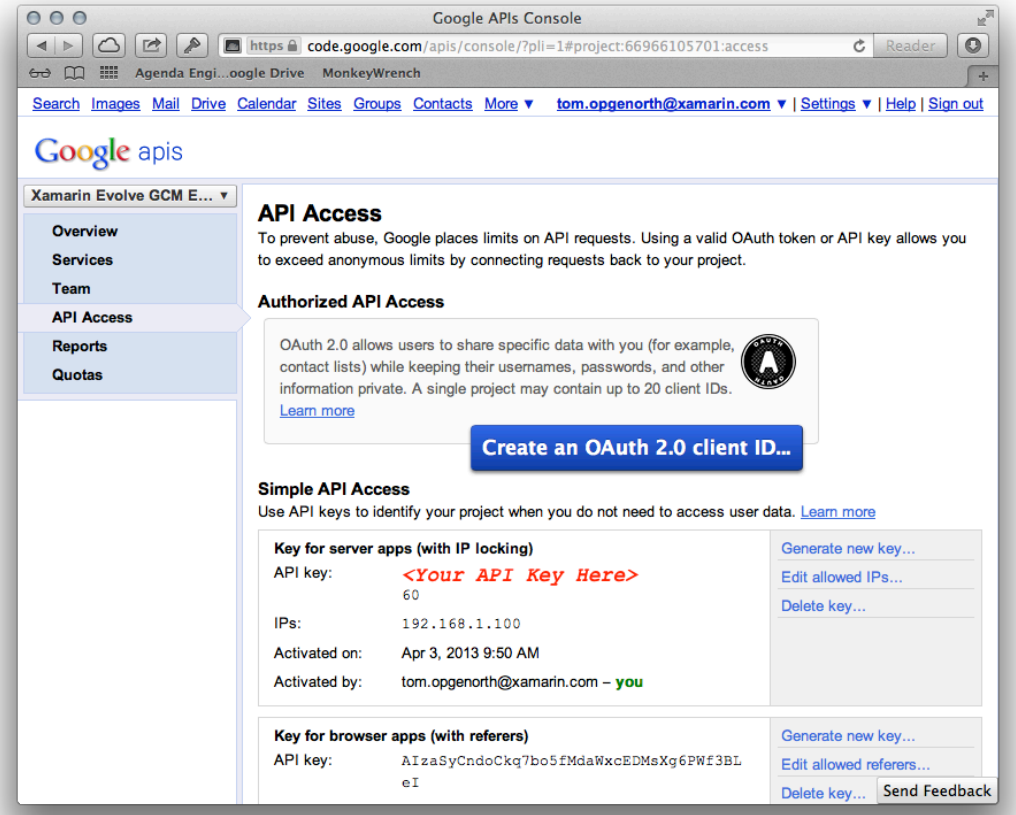


4. Click on the **Create new Server key...** button. A dialog should appear, similar to what you see in the following screenshot:



Enter the IP addresses or subnet of IP's of computer(s) that will be running the server application that will send messages.

5. When you click the **Create** button, you will be taken back to the **API Access** page, and it will look something similar to the following screenshot:



Notice in the above screenshot that an API key has been created for your server applications. Protect the API key – it is not meant for public use as it is part of the security surrounding the communication between your server application and Google Cloud Messaging.

At this point, armed with the sender id and the API key, Google Cloud Messaging has been setup.

Creating the Android Application

Before an Android application can communicate with Google Cloud Messaging, it requires the following changes and components:

1. It must have the appropriate permissions defined.
2. It must have a BroadcastReceiver configured to listen for the messages from Google Cloud Messaging.
3. An IntentService that will handle the messages from Google Cloud Messaging.

The following code snippet shows and explains the permissions that must be granted to an Android application in order to use Google Cloud Messaging:

```
// This will prevent other apps on the device from receiving GCM messages
for this app
```

```
// It is crucial that the package name does not start with an uppercase
letter - this is forbidden by Android.
[assembly: Permission(Name = "@PACKAGE_NAME@.permission.C2D_MESSAGE")]
[assembly: UsesPermission(Name = "@PACKAGE_NAME@.permission.C2D_MESSAGE")]


// Gives the app permission to register and receive messages.
[assembly: UsesPermission(Name = "com.google.android.c2dm.permission.RECEI
VE")]

// This permission is necessary only for Android 4.0.3 and below.
[assembly: UsesPermission(Name = "android.permission.GET_ACCOUNTS")]


// Need to access the internet for GCM
[assembly: UsesPermission(Name = "android.permission.INTERNET")]


// Needed to keep the processor from sleeping when a message arrives
[assembly: UsesPermission(Name = "android.permission.WAKE_LOCK")]
```

Next, the application requires a BroadcastReceiver that will be able to listen for the Intents from Google Cloud Message, and only the intents from Google Cloud Messaging. The BroadcastReceiver must listen for the following Intents:

- `com.google.android.c2dm.intent.RECEIVE` – This is to receive messages from Google Cloud Messaging
- `com.google.android.c2dm.intent.REGISTRATION` – This is to listen for registration related messages.
- `com.google.android.gcm.intent.RETRY` – This is to listen for registration retry messages.

The following code snippet is an example of how to subclass BroadcastReceiver to listen for these Intents:

```
[BroadcastReceiver(Permission= "com.google.android.c2dm.permission.SEND")]
[IntentFilter(new string[] { "com.google.android.c2dm.intent.RECEIVE" }, C
ategories = new string[] { "@PACKAGE_NAME@" })]
[IntentFilter(new string[] { "com.google.android.c2dm.intent.REGISTRATION"
 }, Categories = new string[] { "@PACKAGE_NAME@" })]
[IntentFilter(new string[] { "com.google.android.gcm.intent.RETRY" }, Cate
gories = new string[] { "@PACKAGE_NAME@" })]
public class MyGCMBroadcastReceiver : BroadcastReceiver
{
    const string TAG = "PushHandlerBroadcastReceiver";

    public override void OnReceive(Context context, Intent intent)
    {
        MyIntentService.RunIntentInService(context, intent);
        SetResult(Result.Ok, null, null);
    }
}
```

Handling some or all of these Intents will require I/O operations (network calls) that should not be performed in the `OnReceive` method of the BroadcastReceiver. Instead, Android applications should use an IntentService for handling the messages. The following code snippet is an example of how to implement an IntentService in Xamarin.Android:

```
[Service]
public class MyIntentService : IntentService
{
    static PowerManager.WakeLock sWakeLock;
```

```
    static object LOCK = new object();

    static void RunIntentInService(Context context, Intent intent)
    {
            lock (LOCK)
            {
               if (sWakeLock == null)
               {
                       // This is called from BroadcastReceiver, there is
no init.
                       var pm = PowerManager.FromContext(context);
                       sWakeLock = pm.NewWakeLock(WakeLockFlags.Partial,
WAKELOCK_KEY);
               }
            }

            sWakeLock.Acquire();
            intent.SetClass(context, classType);
            context.StartService(intent);
    }

    protected override void OnHandleIntent(Intent intent)
    {
        try
        {
            var context = this.ApplicationContext;
            var action = intent.Action;

            if
(action.Equals("com.google.android.c2dm.intent.REGISTRATION"))
            {
                HandleRegistration(context, intent);
            }
            else if
(action.Equals("com.google.android.c2dm.intent.RECEIVE"))
            {
                HandleMessage(context, intent);
            }
        }
        finally
        {
            lock (LOCK)
            {
                //Sanity check for null as this is a public method
                if (sWakeLock != null)
                {
                    sWakeLock.Release();
                }
            }
        }
    }
}
```

Registering with GCM

Now that we have an understanding of the components an Android application required by Google Cloud Messaging, lets take at some of then interactions that an Android application must be able to handle. The first thing that an Android application must do is to register with Google Cloud Messaging. It does this by sending off a `com.google.android.c2dm.intent.REGISTER` Intent to Google Cloud Messaging. The registration Intent requires two parameters:

- `app` – this is a PendingIntent that will allow the Google Services framework to interrogate the application to obtain information necessary to register with GCM
- `sender` – this is a comma separated string that holds a list of all the project IDs that the server applications that will be sending messages.

The following code snippet is an example of how this may be done:

```
string senders = "<Google API Console App Project ID>";
Intent intent = new Intent("com.google.android.c2dm.intent.REGISTER");
intent.SetPackage("com.google.android.gsf");
intent.PutExtra("app", PendingIntent.GetBroadcast(context, 0, new Intent(),
0));
intent.PutExtra("sender", senders);
context.StartService(intent);
```

If an application needs to unsubscribe, then it must send a `com.google.android.c2dm.intent.UNREGISTER` message to Google Cloud Messaging. This will take only one parameter, a PendingIntent that will allow the Googe Services framework to acquire information about the application. The following code snippet is an example of how to do this:

```
Intent intent = new Intent("com.google.android.c2dm.intent.REGISTER");
intent.PutExtra("app", PendingIntent.GetBroadcast(context, 0, new Intent(),
0));
context.StartService(intent)
```

Google Cloud Messaging will respond to registration and unregistration requests with a

`com.android.c2dm.intent.REGISTRATION` Intent that will contain one of the two possible values in the extras:

- `registration_id` – If registration is successful, this value will be set and the other two will not. It will hold the registration id that GCM created for this application on the device.
- `unregistered` – If unregistration succeeds, then only this extra will be set.
- `error` – if the registration fails, this will contain one of the strings as an error:

Error Code	Description
`SERVICE_NOT_AVAILABLE`	There was a server error returned, or the device cannot understand the response.

ACCOUNT_MISSING	There is no Google account on the device.
AUTHENTHICATION_FAILED	The password for the Google account is invalid.
INVALID_SENDER	The Android application is sending an invalid sender ID to GCM.
PHONE_REGISTRATION_ERROR	Incorrect phone registration with Google or this device does not support GCM.
INVALID_PARAMETERS	The request send by the Android application is invalid, or this device does not support GCM.

The following code snippet shows an example of how to access these values:

```
string registrationId = intent.GetStringExtra("registration_id");
string error = intent.GetStringExtra("error");
```

Once GCM has returned the registration ID, it is the responsibility of the Android application to contact the application server and provide this registration ID.

Handling Incoming Messages

When the application server sends a message, GCM will route that message to the appropriate applications. This will be contained in a `com.google.c2dm.intent.RECEIVE` Intent. The values from the server will be contained in key-value as part of the extras for the Intent. The following code snippet is an example of how to access some values from the intent:

```
private void HandleMessage(Intent intent)
{
        string score = intent.GetExtraString("score");
        // do something with the score
}
```

Roles of the Application Server

The application server is a crucial part of any system involving Google Cloud Message. There are no specific implementation requirements for an application server. The application server can be an ASP.NET web application, a web service written in Ruby and Sinatra running on Linux, or any possibility. Regardless of how the application server is implemented, it must meet the following set of criteria:

- The ability to communicate with the Android client application
- Store the API key necessary for communicating with Google Cloud Messaging
- Store, retrieve, and delete the registration IDs from the Android client application as appropriate.
- Sent requests to Google Cloud Messaging via HTTPS and resend them as necessary

For more details on the roles of the application server and how send messages via Google Cloud Messaging, consult Google's document on the Role of the 3rd-party Application Server.

Summary

[This chapter provided an introduction to local and remote notifications in both Android and iOS, with a focus on how to implement local notifications. First we looked at iOS, and saw to create a `UINotification` object and publish that notification by calling `UIApplication.SharedApplication.ScheduleLocalNotification`. We also learned how an application can respond to notifications by overriding the method `ReceivedLocalNotification`.]

[From there we moved on to explore local notifications in Android. We saw how to use the `NotificationCompat.Builder` to create notifications and how to use the `NotificationManager` to publish the notifications. The Android section also covered how users may start an Activity by clicking on a notification in the notification drawer.]