**Introduction to Common Push Notification Service**

Many mobile applications include a remote data provider web service that provides users with updated information or allows users to communicate with each other. In some cases mobile applications ‘pull’, i.e. request and receive, information. Often, the data provider service ‘pushes’ updated information to mobile applications in the form of push notifications. With push notifications, the device does not have to wake up periodically to check whether new data is available, thereby conserving battery life. For more details on push notifications, look up [chapter 8](http://windowsphone.interoperabilitybridges.com/articles/chapter-8-introduction-to-windows-phone-7-notifications-for-iphone-developers) in this series.

Mobile application developers who target iPhone, Android and Windows Phone need a separate data provider for each type of device since they use different protocols for push notifications. Developers need to use the [Apple Push Notification Service](http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/ApplePushService/ApplePushService.html) (APNS) for iOS devices, [Cloud to Device Messaging Framework](http://code.google.com/android/c2dm/) (C2DM) for Android devices and [Microsoft Push Notification Service](http://msdn.microsoft.com/en-us/library/ff402537(v=vs.92).aspx) (MPNS) for Windows Phone Devices. Mobile application developers need to manage different types of devices and they need to interface with each notification service in order to push messages.

## The Common Push Notification Service (CPNS)

[The Microsoft Interoperability Strategy Group](http://www.interoperabilitybridges.com/) is a Microsoft team focused on interoperability. It has developed a Common Push Notification Service sample and published it as an open source project, that makes it easy to send push messages to iPhone, Android and Windows Phone devices using the appropriate protocol. The Common push notification service has the following design goals for developers

1. Freedom from the burden of managing different types of devices
2. Freedom for the developers from implementing and operating separate data providers each using different push notification mechanisms.
3. Ease of extending the notification service to support additional device types in future.

The sample implementation uses Azure but the sample code can be modified to make it run on ASP.net. We have provided tips to do so below. Azure provides a compelling benefit that a short burst of notifications can be handled easily by dynamically provisioning additional resources.

This paper is divided into three parts:

1. Description of the Common Push Notification Service and its operation.
2. Developer documentation to integrate the CPNS with the mobile application and the data provider web service.
3. Developer documentation for customization of the CPNS including tips on implementing it as ASP.net.

## Design of a Sample Common Push Notification Service

A Common Push Notification Service sample has three primary entities, namely, subscriptions, devices and notifications.

1. **Subscriptions**: Subscriptions encapsulate a logical group of notifications on a particular topic. For example, a sports mobile application may create subscriptions such as “Baseball”, or “Oakland A’s.” Application Manager, a person responsible for the mobile application, can create the subscriptions in a CPNS.
2. **Devices**: Users sign up, explicitly or implicitly, for the subscriptions using mobile application. Users receive notifications via applications on their mobile device.
3. **Notifications**: Application managers use the CPNS to send notifications, to a subscription. CPNS sends notification messages to devices that are signed up for those subscriptions via the device specific notification service. For example, the latest Oakland A’s game score may be a notification sent to users that have signed up for Oakland A’s subscription.

## Architecture of a Common Push Notification Service

Common Push Notification Service sample is implemented as an Azure service. It consists of a web role that exposes REST end points for device, subscription and notification management and a worker role that sends push notifications.



### Service End Points

Common Push Notification Service has two end points:

1. An endpoint for devices to sign up for notification subscriptions.
2. An end point to be used by the data provider web service to manage subscriptions and send notifications to devices that are signed up for subscriptions.

### Common Notification Service engine.

The engine manages the list of subscriptions and the devices that have signed up for the subscriptions. It keeps the device information that is needed to send notification messages via each proprietary notification service. It also manages messages to be sent to devices.

### Message Pump.

Message pumps forward notification messages to iPhone, Android and Windows Phone devices via APNS, C2DM or MPNS respectively. The respective notification services forward the messages to the devices.

## Using the Common Push Notification Sample Service

This section guides you through the steps to run the sample service and explains the basics of the service. The sample includes three different solutions:

1. **AzurePushMessages**: This is a Common Push Notification Service sample.
2. **MessageReceiver**: A Windows Phone 7 sample application. This allows a user to sign up for subscriptions and to receive notification messages.
3. **PushMessageDataProvider**: This is a Windows application that is a sample of a notification data provider . A real data provider is likely to be a web service. This application allows the application manager to create subscriptions and send notifications to subscribing devices.

**Scenario:** In this section, we will look at the steps necessary to use **AzurePushMessages**. We will send push notifications to **MessageReceiver** using **PushMessageDataProvier**. Steps needed to send notifications to iOS and Android devices are similar; you can review those using included Android and iPhone sample applications.

### Before you Begin

Make sure you have installed Visual Studio 2010 or Visual Studio C# Express 2010, Visual Studio Web Developer Express 2010, and Windows Phone 7 tools for Visual Studio. We will use C# Express to run the data provider, Web Developer Express to run the Common Notification Service and Windows Phone 7 tools to run the sample WP7 application. Make sure you have installed [Microsoft WCF Rest Starter kit](http://www.asp.net/downloads/starter-kits/wcf-rest). You will need to add references to Microsoft.ServiceModel and Microsoft.ServiceModel.Web.

This chapter does not cover Azure development or Windows Phone development. If you are not familiar with Windows Phone 7 programming, review the material at [App Hub](http://create.msdn.com/en-US). To make yourself familiar with WP7 and Microsoft Push Notification Service, read [this MSDN article](http://msdn.microsoft.com/en-us/library/ff402558(v=vs.92).aspx). To learn about developing Azure applications, read [Introduction to the Windows Azure Platform](http://msdn.microsoft.com/en-us/library/ff803364.aspx), [Developing for Windows Azure](http://msdn.microsoft.com/en-us/windowsazure/default.aspx) and [Developing and Deploying Windows Azure Apps in Visual Studio 2010](http://msdn.microsoft.com/en-us/magazine/ee336122.aspx).

### Deploying the Sample Service Locally

In this step, we will run CPNS locally. Before we deploy the service to Azure, we will deploy it on the local compute and storage Azure fabric. Once we have tested it locally, we will deploy it to Azure cloud.

1. Start Visual Studio or Visual Studio Web Developer Express in an elevated (admin) mode.
2. Open AzurePushMessages solution in Visual Web Developer or Visual Studio.
3. Click **F5** to run the service in debug mode.

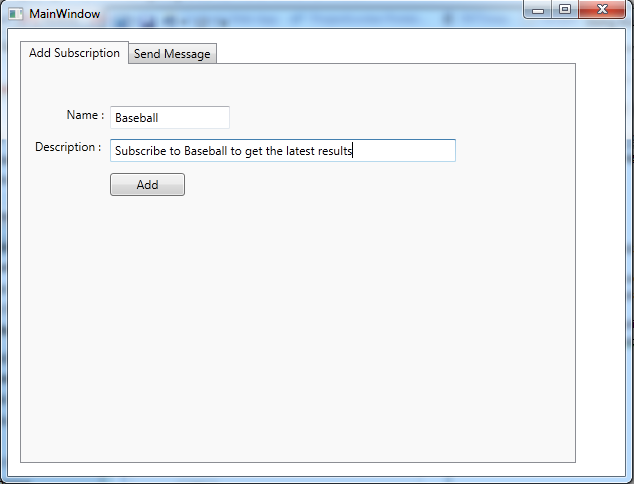
You will see IE start with <http://127.0.0.1/> address with “HTTP error 403” error which is expected since the web service end points are not at this address.

Normally, an Azure application starts locally at http://127.0.0.1 , but sometimes it may use a different port. If it uses a different port, you will need to modify the sources for both, the WP7 sample application and PushMessageDataProvider sample application, in order to use the new port. Look for http://127.0.0.1 in the other two projects and replace port number 81 with the new port number.

### Using Data Provider Application

In this step, we will use the sample data provider application to create subscriptions and send messages to devices.

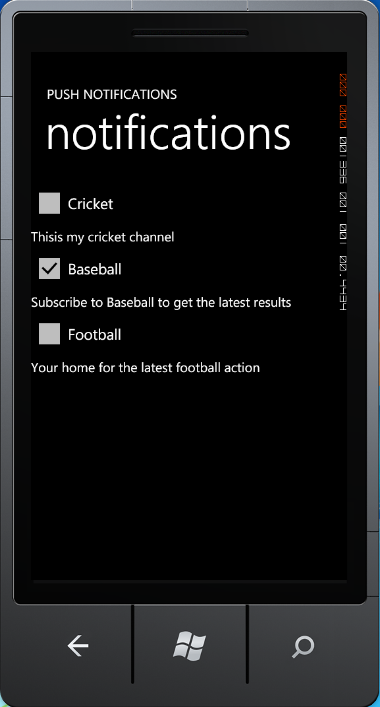
1. Start Visual Studio or Visual Studio C# Express
2. Open PushMessageDataProvider sample solution. If necessary, replace the port number as described above.
3. Click **F5** to run the PushMessageDataProvider.
4. Create a sample data provider subscription. Type “Baseball” in the **Name** field and “Subscribe to Baseball to get the latest results” in the **Description** field. Click **Add** to add the subscription to the service.
5. Similarly add two more subscriptions called “Football” and “Cricket” with appropriate descriptions.



### Windows Phone 7 Sample Application

In this step, we will run the sample Windows Phone 7 application to receive notifications sent by PushMessageDataProvider.

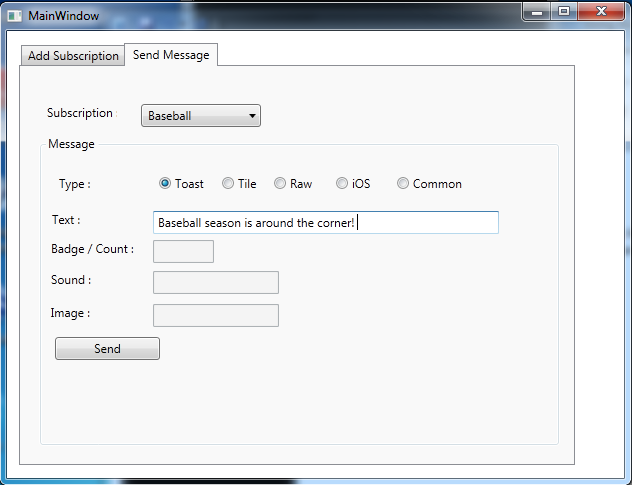
1. Start another instance of Visual Studio C# Express or Visual Studio and open **MessageReceiver** Windows Phone 7 solution. If necessary, replace the port number as described above.
2. Click **F5** to run the **MessageReceiver** sample Windows Phone 7 application. You should see screen as shown below. Select Baseball (and other subscriptions).



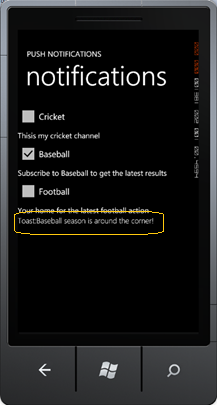
### Sending a Push Notifications

In this step, we will send notifications using PushMessageDataProvider.

1. Use the PushMessageDataProvider sample application and click the Send Message” tab.
2. In the **Subscription** dropdown, select Baseball. Select **Toast** and type a message in the **Text** field and click **Send.**



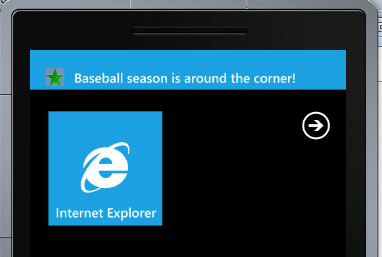
In a few seconds, you should receive a [toast message](http://msdn.microsoft.com/en-us/library/ff402558(v=vs.92).aspx) on the Windows Phone 7 application as shown below:



We can send a [raw notification](http://msdn.microsoft.com/en-us/library/ff402558(v=vs.92).aspx) message using **PushMessageProvider** in the same manner which will show up on the sample Phone application similar to above toast.

Let us look at how toast messages can be received even when the application is not running on the phone. Quit the MessageReceiver application running on Windows Phone 7 by clicking the back arrow button on the phone.

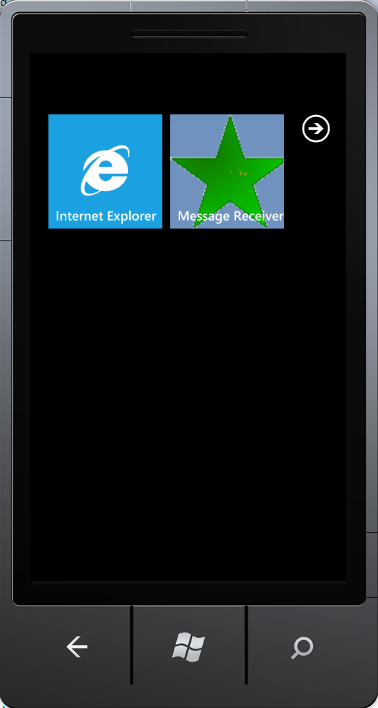
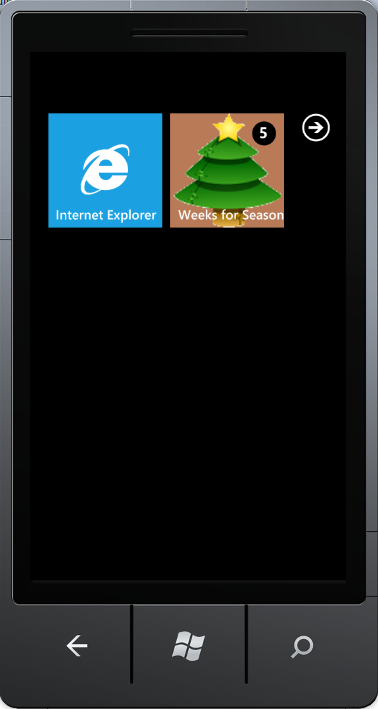
Use the PushMessageProvider application to send the another toast message. In a few seconds, you should see the toast appear on the phone emulator. While both raw and toast messages can be received by the application while the application is running, a toast message is received by the device and shown as a toast even when the application is not running.



#### Sending a Tile Notification

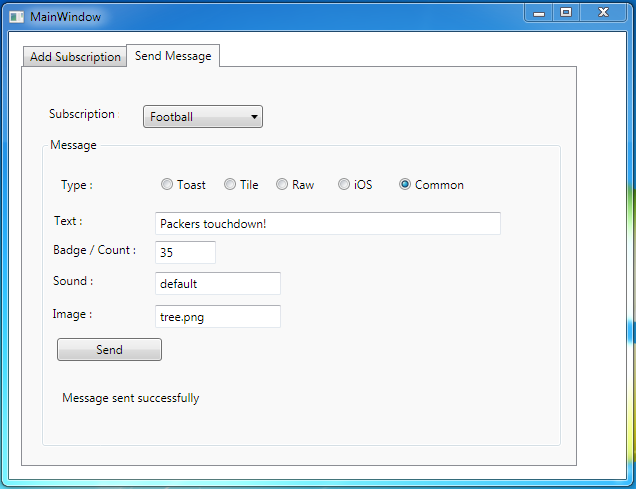
1. On the Windows Phone 7, press and hold the MessageReceiver application icon and select “Pin to Start” to pin the application tile.
2. In the PushMessageDataProvider select the **Tile** radio button. Type “Weeks for Season” in the **Text field,** 5 in the **Count** textbox, and “tree.png” in the **Image** field and click **Send.**

After a few seconds, you should see the updated tile as shown below:

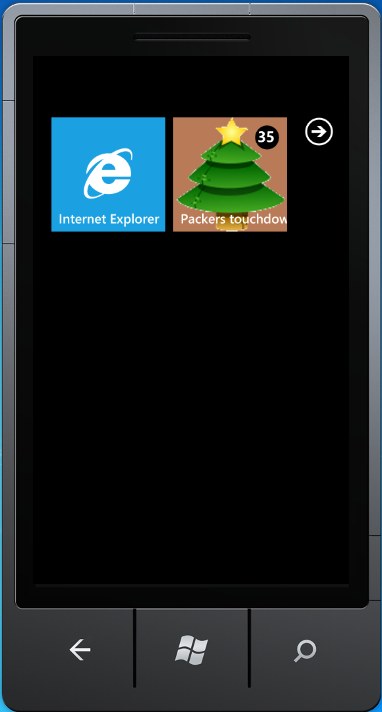
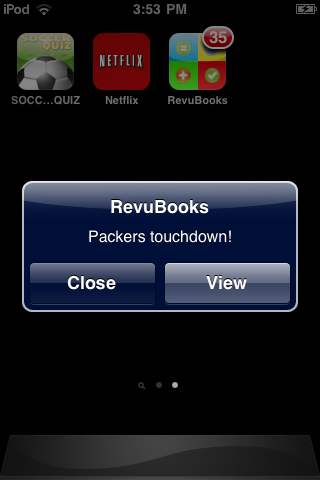
 

## Sending Push Notifications to iPhone, Android and Windows Phone 7

You can use the Common Push Notification Service to send a push message to iPhone, Android, and Windows Phone 7. The following images show an example where a message sent to Football subscription is forwarded to iPhone, Android, and Windows Phone 7 which are signed up for the Football subscription. As you see below, the same message manifests in a different way on the two platforms.



On iPhone, the common message is sent as a normal iPhone push notification whereas on Windows Phone 7 it is sent as a tile message. On Android, each application can choose how to handle the push notification.



### Configuration Changes To Support iPhone and Android

In order to send push messages to iPhone, the sender service must be registered with Apple Developer portal and use certificate authentication. You will need to provision to use APNS, download certificates, configure Azure or ASP.net service and deploy the certificates with your service. Read details about how to send Push Notifications using ASP.net, follow instructions listed [here.](http://arashnorouzi.wordpress.com/2011/04/01/sending-apple-push-notifications-in-asp-net-%E2%80%93-part-2-generating-apns-certificates/)

1. Add your apn\_developer\_identity.p12 file to your Azure Worker role project. In WorkerRole.cs file add the name of your certificate file and your cert password.

private const string ApnsP12File = "apn\_developer\_identity.p12";

// This is the password that you protected your p12File

// If you did not use a password, set it as null or an empty string

private const string ApnsP12FilePassword = "YourAPNCertPwd";

You will also need to uncomment a couple of lines in ApnsConnection.cs file. Look for “uncomment” in the comments.

For sending push notifications to Android devices, follow the instructions as [Android Cloud to Device Messaging Framework](http://code.google.com/android/c2dm/) and [signup to use C2DM](http://code.google.com/android/c2dm/signup.html).

1. In addition, add your registered Google username and password at C2DMConnection.cs file in CPNS:

private const string Username = "YourGoogleEmail@gmail.com";

private const string Password = "YourGooglePassword";

You will also need to uncomment a couple of lines in C2DMConnection.cs file. Look for “uncomment” in the comments.

## Deploying CPNS to Azure

Before you deploy the application to Windows Azure, you will need to create a Windows Live ID and sign up for Azure. You can start with a 30 day free trial to do so. For detailed instructions on how to deploy to Azure, review [Windows Azure How-to Topics](http://msdn.microsoft.com/library/gg432998.aspx) and follow the steps in “[How to: Deploy a Windows Azure Application](http://msdn.microsoft.com/en-us/library/gg432994.aspx)” or use the instructions in [Deploying Applications to Windows Azure](http://programming4.us/desktop/1220.aspx).

Once you signed up for your Azure subscription and created Azure storage, you can deploy the service to Azure using Visual Studio. After deploying it successfully, note the URL of the application in the staging environment**.**

#### Configuration change

You need to use your management certificate in Azure ServiceConfiguration.cscfg and ServiceDefinition.csdef file as shown below. For more information, read Visual Studio Azure documentation.

1. In Azure ServiceConfiguration.cscfg, modify certificates node

<Certificates>

<Certificate name="YourCert" thumbprint="CertThumbPrint" thumbprintAlgorithm="sha1" />

</Certificates>

1. Similarly, in your ServiceDefinition.csdef file, update the certificates node

<Certificates>

<Certificate name="YourCert" storeLocation="LocalMachine" storeName="My" />

</Certificates>

Update the URLs in the MainPage.xaml.cs file in MessageReceiver project and MainWindow.xaml.cs file in PushMessageDataProvider project with the staging URL using the instructions given above. Execute the same steps described above to send and receive push notifications.

# Integrating with the Common Push Notification Service

In this section, we will show you how to integrate with the Common Push Notification Service using a sample mobile application and its corresponding data provider web service. While your actual mobile application and the accompanying web service are likely to be very different and more complex, its interface with the Common Push Notification Service is likely to be very similar.

## Subscription Management End Point

The end point for managing subscriptions provides an API to add, delete or list subscriptions in the system. The [REST](http://en.wikipedia.org/wiki/Representational_state_transfer) APIs listed below are written using [WCF service contract definition format](http://msdn.microsoft.com/en-us/library/ms733070.aspx). WebInvoke refers to a POST message whereas WebGet refers to a GET message.

//list all subscriptions in the system. User may pick from these if needed

[WebGet(UriTemplate = "/subs")]

//list all subscriptions in the system. User may pick from these if needed

[WebInvoke(UriTemplate = "/sub/create/{subID}?desc={description}")]

//list all subscriptions that the device has signed up for

[WebGet(UriTemplate = "/subs/{deviceID}")]

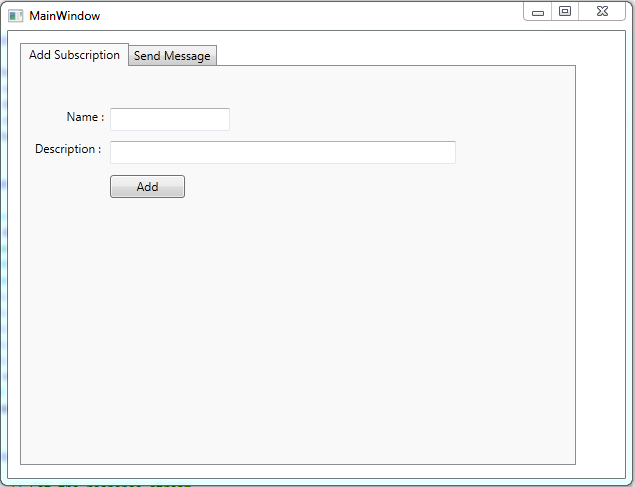
//sign up a device for a particular subscription

[WebInvoke(UriTemplate = "/sub/add/{subID}/{deviceID}")]

// remove a device subscription

[WebInvoke(UriTemplate = "/sub/delete/{subID}/{deviceID}")]

Let us look at how these APIs are used in our sample PushMessageDataProvider application shown above. The user enters the name and description of the subscription and clicks **Add** to create a new subscription. In response to **Add**, the subCreateBtn\_Click event handler, listed below,is invoked. The event handler sets up a URL for the “create” REST call and calls the SendPostRequest method. In the SendPostRequest method, we encode the POST data, if any, and create an HttpWebRequest that adds an authentication header with username and password. The HTTP Post request is sent to the subscription REST end point. subCreateBtn\_Click receives the server response synchronously and displays it to the user.



private void subCreateBtn\_Click(object sender, RoutedEventArgs e)

{

statusTextBlock.Text = "";

string subNameStr = subNameTextBox.Text;

string descStr = descTextBox.Text;

string status = SendPostRequest(BaseAddress + "/sub/create/"+subNameStr + "?desc="+descStr, "");

statusTextBlock.Text = status;

}

private static string SendPostRequest(string url, string postData)

{

ASCIIEncoding encoding = new ASCIIEncoding();

byte[] data = encoding.GetBytes(postData);

// Prepare web request...

HttpWebRequest request =

(HttpWebRequest)WebRequest.Create(url);

request.Credentials = new NetworkCredential(username, password);

request.PreAuthenticate = true;

request.Method = "POST";

request.ContentType = "application/x-www-form-urlencoded";

request.ContentLength = data.Length;

Stream newStream = request.GetRequestStream();

// Send the data.

newStream.Write(data, 0, data.Length);

try

{

using (HttpWebResponse response = request.GetResponse() as HttpWebResponse)

{

// Get the response stream

StreamReader reader = new StreamReader(response.GetResponseStream());

return reader.ReadToEnd();

}

}

catch (WebException e)

{

return ("Exception : " + e.Message);

}

}

In your data provider web service or application, you will need to use the above end point in a similar manner to create subscriptions.

## Device Management End Points

Common Push Notification Service includes three end points, one each for registering iOS, Android and Windows Phone devices respectively. In this design that uses separate endpoints for each device type, additional end points can be added without affecting existing code.

### API for Registering Windows Phone 7 devices

//Registers a device in the notification service

[WebInvoke(UriTemplate = "/register/{deviceID}?uri={uri}")]

//Delete a device from the notification service for which it is regisered. Removes all subscriptions associated with it

[WebInvoke(UriTemplate = "/unregister/{deviceID}")]

### APIs for Registering iOS and Android devices

//Registers a device in the notification service

[WebInvoke(UriTemplate = "/register/{deviceID}")]

//Delete a device from the notification service for which it is regisered. Removes all subscriptions associated with it

[WebInvoke(UriTemplate = "/unregister/{deviceID}")]

The code fragment below shows how the sample Windows Phone 7 application uses the API to register itself. For Windows Phone, the register API is called after a unique device ID and a notification channel have been created by the Windows Phone application. You can see the details in the Windows Phone sample application. To know more about how to use Microsoft Push Notification Service, refer to P[ush Notifications Overview for Windows Phone](http://msdn.microsoft.com/en-us/library/ff402558(v=vs.92).aspx). The code snippet uses the deviceID and channel URI tuple to register with the Common Push Notification Service. Common Push Notification Service stores the channel URI indexed by the device ID and later uses the URI to forward a notification to MPNS.

private void RegisterForNotifications()

{

string uriStr = \_channel.ChannelUri.ToString();

encodedUriStr = EncodeTo64(uriStr);

string deviceAddUrl = "/register/" + deviceIDStr + "?type=WP7&uri=" + encodedUriStr;

deviceRestConn.SendPostRequest(deviceAddUrl, "");

…

}

The APNS use the unique device ID for sending notifications. Consequently, the iOS device registration API only requires a device ID. Similarly, Google C2DM uses device registration ID for sending notifications and the API uses the unique registration ID as a parameter in the register API. While we have not shown the Objective-C code to register the iOS device, or the Java code for Android, the code that makes an HTTP POST request is similar in both cases.

## Signing up for Notification Subscription

The end point used by devices to sign up for notification subscriptions consists of methods to sign up, list or delete subscriptions.

//list all subscriptions for which the device has signed up

[WebGet(UriTemplate = "/subs/{deviceID}")]

//sign up a device for a particular subscription

[WebInvoke(UriTemplate = "/sub/add/{subID}/{deviceID}")]

// remove a device from a subscription

[WebInvoke(UriTemplate = "/sub/delete/{subID}/{deviceID}")]

In PushMessageReceiver, our sample Windows Phone 7 application, the user needs to explicitly sign up for notification subscriptions. However, you may choose to signup the user for a subscription implicitly in your application. For example, based on the user’s location, you may choose to enlist the user in a specific traffic subscription. As an application developer, you can decide the most appropriate model for notification subscriptions.

### Displaying Available Device Subscriptions

In our sample application the device subscriptions are shown by retrieving the subscription information from the Common Notification Service using the “/subs/{deviceID}” API.. In our WP7 sample application, we have created a helper class called RESTConnection in order to send GET or POST requests. SendGetRequest is a simple method which makes an HTTP GET request.

public class RESTConnection

{

…

public void SendGetRequest(string url, string token)

{

\_url = url;

WebClient pushMsgService = new WebClient();

pushMsgService.Credentials = new NetworkCredential(\_username, \_password);

pushMsgService.DownloadStringCompleted += new System.Net.DownloadStringCompletedEventHandler(pushMsgService\_DownloadStringCompleted);

pushMsgService.DownloadStringAsync(new Uri(\_baseUrl + url), token);

}

private void pushMsgService\_DownloadStringCompleted(object sender, System.Net.DownloadStringCompletedEventArgs e)

{

if (DownloadHandler != null)

DownloadHandler(this, e);

}

public void SendPostRequest(string url, string postData, string token)

{

\_url = url;

WebClient pushMsgService = new WebClient();

pushMsgService.Credentials = new NetworkCredential(\_username, \_password);

pushMsgService.UploadStringCompleted += new System.Net.UploadStringCompletedEventHandler(pushMsgService\_UploadStringCompleted);

pushMsgService.UploadStringAsync(new Uri(\_baseUrl + url), "POST", "", token);

}

private void pushMsgService\_UploadStringCompleted(object sender, System.Net.UploadStringCompletedEventArgs e)

{

if (UploadHandler != null)

UploadHandler(this, e);

}

…

}

Once the asynchronous GET request is completed, RestConnection class invokes the DownloadHandler event handler which in turn invokes pushRestConn\_DownloadHandler. The response, serialized using XML, is parsed and queried using LINQ. We retrieve all the SubscriptionInfo objects that include name, description and the signup status of the device. We bind the list of subscriptions with the subscriptions listbox.

private void SetupSubscriptionOptions()

{

// set up notification channel and subscribe to notifications

pushRestConn.SendGetRequest("/subs/" + deviceIDStr, "subs");

pushRestConn.DownloadHandler += new DownloadEventHandler(pushRestConn\_DownloadHandler);

}

void pushRestConn\_DownloadHandler(object sender, DownloadStringCompletedEventArgs e)

{

pushRestConn.DownloadHandler -= new DownloadEventHandler(pushRestConn\_DownloadHandler);

string userToken = (e as DownloadStringCompletedEventArgs).UserState as string;

if (userToken == "subs") {

XDocument xdoc = XDocument.Parse(e.Result);

IEnumerable<XElement> xelements = xdoc.Descendants();

var subscriptions = from element in xdoc.Descendants("{http://schemas.datacontract.org/2004/07/MsgHelperLib.Common}DeviceSubscriptionInfo")

select new SubscriptionInfo

{

Name = element.Element("{http://schemas.datacontract.org/2004/07/MsgHelperLib.Common}Name").Value,

Description = element.Element("{http://schemas.datacontract.org/2004/07/MsgHelperLib.Common}Description").Value,

IsSubscribed = element.Element("{http://schemas.datacontract.org/2004/07/MsgHelperLib.Common}IsSubscribed").Value == "true" ? true : false

};

subscriptionsList.ItemsSource = subscriptions;

}

}

### Add/Remove Subscriptions

In the above UI, the user opts in or out of a particular subscription using checkboxes on the screen. In response to the user action, we add or remove the device from that notification subscription using /sub/add/{subID}/{deviceID} and /sub/delete/{subID}/{deviceID} APIs resp. An application may also choose to add or remove the subscription implicitly by using these APIs. The SubscribeToNotification and UnSubscribeToNotification methods use SendPostRequest method shown above.

void cb\_Click(object sender, RoutedEventArgs e)

{

CheckBox cb = sender as CheckBox;

if ((bool) cb.IsChecked)

{

SubscribeToNotification((string)cb.Tag);

}

else

{

UnSubscribeToNotification((string)cb.Tag);

}

}

private void SubscribeToNotification(string p)

{

string deviceSubUrl = "/sub/add/" + p + "/" + deviceIDStr;

pushRestConn.SendPostRequest(deviceSubUrl, "", "subscribe");

…

}

private void UnSubscribeToNotification(string p)

{

string deviceSubUrl = "/sub/delete/" + p + "/" + deviceIDStr;

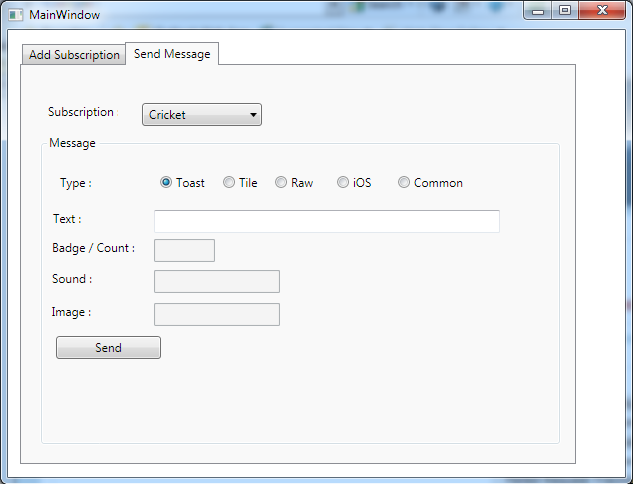
pushRestConn.SendPostRequest(deviceSubUrl, "", "unsubscribe");

…

}

## Sending Push Notification Messages

The PushMessageDataProvider application panel, shown below, is used to send a notification message to specific subscriptions. The mobile application application manager selects the type of message to send, types the message text and other parameters and clicks **Send**. When the application manager clicks **Send**, the data provider application collects the message text, the subscription name, and create a REST message for sending a notification. The REST APIs for sending for all types of messages are invoked in the same way.



private void SendToastMessage()

{

msgStatusLbl.Content = "";

try

{

string msg = msgTxtBox.Text;

string sub = GetSelectedSubscription();

if (!string.IsNullOrEmpty(msg))

{

SendPostRequest(BaseAddress + "/message/toast/" + sub + "?mesg=" + msg, "");

msgStatusLbl.Content = "Message sent successfully";

}

else

msgStatusLbl.Content = "Message should not be blank";

}

catch (Exception e)

{

msgStatusLbl.Content = "Exception: " + e.Message;

}

}

The above method uses the SendPostRequest method shown above.

## Design of a Common Push Notification Service

As described above, the service REST end points are implemented by the Azure web role, whereas the message pump is implemented by the Azure worker role.

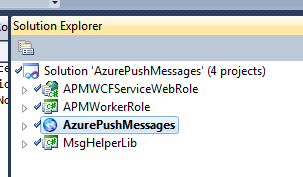
Our service sample uses Azure table storage to store the notification subscriptions, devices IDsn and device sign up data. Azure web role uses LINQ to store data received via REST API in Azure tables. On the other hand, notification messages received by the Azure web role are added to an Azure Queue. The worker role fetches each message from the notification queue, processes the message, and forwards it. The worker role looks up the list of devices for each subscription and sends the message to each device via APNS, C2DM or MPNS, depending upon the type of the device.



The AzurePushMessages solution has three different projects:

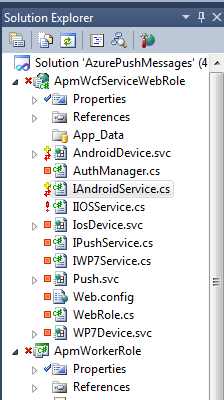
1. APMWCFServiceRole - a web role or front end that implements REST endpoints.
2. APMWorkerRole – a worker role or a service that implements the message pump.
3. MsgHelperLib – a library of classes used by both roles to use data stored in Azure storage.

The AzurePushMessages solution also includes a fourth project of the same name which contains Azure roles.



### The Web Role

The web role implements four different end points, an iOS registration endpoint (IIOSService.cs), an Android registration endpoint (IAndroidService.cs), a WP7 registration endpoint (IWP7Service.cs) and the subscription and notification endpoint (IPushService.cs). These interfaces are implemented in iOSDevice.svc.cs, AndroidDevice.svc.cs, WP7Device.svc.cs and Push.svc.cs files respectively.



## Device Registration

The RegisterDevice method in WP7Device.svc.cs file registers WP7 devices. This method first checks the validity of the WP7 device ID and the URI and then saves or updates the device details to the Azure table. It may be necessary to update the device details since Windows Phone 7, device notification URI may change over time.

public string RegisterDevice(string deviceID, string uri)

{

if (IsDeviceIDValid(deviceID) && IsUriValid(ref uri, true))

return push.RegisterDevice(deviceID, DEVICE\_TYPE\_NAME, uri);

else

throw new WebFaultException<string>(Util.GetExceptionString(PushMessageErrors.Err\_IllLegalDeviceID), HttpStatusCode.BadRequest);

}

public string RegisterDevice(string deviceID, string type, string uri)

{

PushMessageErrors err;

// the device endpoints have made sure that device IDs are valid. This method is not visible on the outside.

try

{

err = devMgr.AddOrUpdateDevice(type, deviceID, uri);

}

catch (Exception e)

{

Trace.TraceError(string.Format("Internal Error: RegisterDevice device: {0}, type:{1}, uri:{2}, Error: {3}", deviceID, type, uri, e.Message));

throw new WebFaultException<string>(e.Message, System.Net.HttpStatusCode.InternalServerError);

}

//if there was error in adding the device, return error.

if (err != PushMessageErrors.Success)

throw new WebFaultException<string>(Util.GetExceptionString(err), HttpStatusCode.BadRequest);

else

return "Success";

}

We have implemented a helper class called DeviceManager that encapsulates all Azure tables operations making it easy to manage devices. . The device data is encapsulated using DeviceDataModel class and persisted in table.

public PushMessageErrors AddOrUpdateDevice(string devTypeName, string deviceID, string uri)

{

DeviceDataModel ddm = dds.SelectByDeviceIDAndType(deviceID, devTypeName);

try

{

if (ddm == null)

{

//no such device, create it

ddm = new DeviceDataModel(devTypeName, deviceID);

ddm.Address = uri;

dds.Insert(ddm);

return PushMessageErrors.Success;

}

else

{

//for WP7, we will update the URI with the new value

ddm.Address = uri;

dds.Update(ddm);

return PushMessageErrors.Success;

}

}

catch (Exception e)

{

throw (new ApplicationException(e.Message));

}

}

### Adding a new Device Type

Now that we have seen how devices are registered, let us see what it takes to add a new device type. To add a new device type, here are the steps :

1. Create a new device endpoint using code that is simlar to IIOSDevice.cs and iOSDevice.svc.cs or IWP7Device.cs and WP7Device.svc.cs. You will need to implement methods to register and unregister the deviceDecide which unique device data will serve as identification data for the API.
2. Implement methods to validate unique devicedata. Each device specific push notification service has its own validation rules.
3. Save the device details in the Azure table using the DeviceDataModel class.

## Notification Subscriptions

Let us look at CreateSubscription method which may be invoked only by authorized users. The method authenticates the user using username/passwordvia AuthManager class. Since we cannot configure IIS for basic authentication on Azure, we have implemented own authentication manager.

The method uses the SubscriptionInfoManager , a class that encapsulates subscriptions, to save the subscriptions. The code for this class to save the subscription information in Azure tables is similar to the code that is used for saving the device information.

public string CreateSubscription(string subName, string description)

{

//user must be authenticated

if (!AuthManager.AuthenticateUser())

{

//if not, return 401

AuthManager.ConstructAuthResponse();

return null;

}

PushMessageErrors err;

try

{

//ask the subscription manager to add one more subscription name

err = subscriptionInfoMgr.AddSubscriptionInfo(subName, description);

}

catch (Exception e)

{

Trace.TraceError(string.Format("Internal Error: CreateSubscription subName: {0}, Error: {1}", subName, e.Message));

throw new WebFaultException<string>(e.Message, System.Net.HttpStatusCode.InternalServerError);

}

if (err != PushMessageErrors.Success)

throw new WebFaultException<string>(Util.GetExceptionString(err), System.Net.HttpStatusCode.BadRequest);

else

return "success";

}

## Notification Messages

Notification messages may be sent only by authorized users. After authenticating users SendToastNotification method checks the validity of the subscription. It then creates a new notification message, such as a toast shown below, using message text received in the REST call. This message is then enqued in the Azure queue.

[OperationContract]

[WebInvoke(UriTemplate = "/message/toast/{subID}?mesg={message}")]

string SendToastNotification(string subID, string message);

…

public string SendToastNotification(string subscriptionName, string toast)

{

if (!AuthManager.AuthenticateUser())

{

AuthManager.ConstructAuthResponse();

return null;

}

//make sure subscription name is created

bool subExists = subscriptionInfoMgr.IsSubscriptionRegistered(subscriptionName);

if (!subExists)

{

throw new WebFaultException<string>(Util.GetExceptionString(PushMessageErrors.Err\_SubscriptionNameNotFound), System.Net.HttpStatusCode.BadRequest);

}

try

{

ToastMessage toastMsg = new ToastMessage(subscriptionName, toast);

msgQueue.Enque(toastMsg);

return "success";

}

catch (Exception e)

{

Trace.TraceError(string.Format("Internal Error: SendToast subscription: {0} toast: {1}, Error: {2}", subscriptionName, toast, e.Message));

throw new WebFaultException<string>(e.Message, System.Net.HttpStatusCode.InternalServerError);

}

}

Let us see how ToastMessage class is implemented. ToastMessage is subclass of PushMessage which is uses a dictionary of key-value pairs in addition to message type and the subscription.

public class PushMessage

{

public short Type { get; set; }

public string SubscriptionName { get; set; }

public Dictionary<string, string> message { get; set;}

public PushMessage(string subscriptionName, short type)

{

SubscriptionName = subscriptionName;

Type = type;

message = new Dictionary<string, string>();

}

public PushMessage()

{

}

}

public class ToastMessage: PushMessage

{

public ToastMessage(string subscriptionName, string toast)

: base(subscriptionName, (short) PushMessageType.Toast)

{

message.Add("toast", toast);

}

public String Toast { get { return message["toast"]; } }

public ToastMessage()

: base()

{

}

}

All other message types, namely, raw message, tile message, iOS notification message as well as common message which can be sent to iPhone, Android, and Windows Phone, are defined in a similar manner. For all message types, the message specific data is stored in the dictionary. This makes it extensible for other message types.

### Adding a new Message Type

To add a new message type

1. Create a new message class that is a subclass of PushMessage. Map the message components in the dictionary.
2. Add a new API to send a messagetothe iPushService.cs file and add its implementation to the PushService.svc.cs file. In the method, authenticate the user, validate the message data, construct an object of the new message type class and queue it for sending it to the worker role.

## Message Pump

We use the Azure worker role a message pump to forward notifications In the Azure worker role run method, we instantiate the connection classes, namely, APNSConnection, C2DMConnection, and MPNSConection which handle the MPNS, C2DM or to APNS respectively. The worker role polls the message queue and waits for a notification message and forwards it using the appropriate connection class. It checks each connection class to see if it supports the message type and forwards it to each device that has signed for the subscription of that message.

mpnsConnection = new MPNSConnection(WP7BatchingPolicy.Immediately, 3, MPNSCertificate);

//get the devicetype MPNS supports - only WP7

mpnsDevType = mpnsConnection.SupportedDeviceType;

//Get the type of messages it handles, toast, raw, tile and common

mpnsSupportedMessages = mpnsConnection.HandlesMessageTypes(Enum.GetValues(enumType));

//create APNSConnection for Apple Push notification. Use Sandbox using certs and use 3 retries before lo

apnsConnection = new APNSConnection(true, APNSp12File, APNSp12FilePassword, 3);

//get the type of device it support

apnsDevType = apnsConnection.SupportedDeviceType;

//find the types of messages it supports. iPhone and Common types

apnsSupportedMessages = apnsConnection.HandlesMessageTypes(Enum.GetValues(enumType));

…

while (true)

{

//sleep for the leftover time and dequeue a message and then process it. We sleep for at most "WorkerRoleLoopTime" in every cycle

currentTime = DateTime.Now;

int millSecForNextCheck = WorkerRoleLoopTime - currentTime.Subtract(lastMsgCheckTime).Milliseconds;

Thread.Sleep(millSecForNextCheck > 0 ? millSecForNextCheck : 0);

lastMsgCheckTime = DateTime.Now;

if ((pushMsg = msgQueue.Deque()) != null)

{

//process will look at the message type and send it to appropriate connection

processMessage(pushMsg);

Trace.WriteLine("APMWorkerRole Received message", "Information");

}

}

//based on the type of message, it is sent to appropriate connection

private void processMessage(PushMessage pushMsg)

{

//check if APNS and MPNS support a given type

bool ifAPNSSupportsType = apnsSupportedMessages[(PushMessageType)pushMsg.Type];

bool ifMPNSSupportsType = mpnsSupportedMessages[(PushMessageType)pushMsg.Type];

if (ifAPNSSupportsType)

{

//if APNS supports it, get the list of devices of the type APNS supports.

IEnumerable<DeviceDataModel> ddmList = sds.SelectByDeviceTypeandSubscription(pushMsg.SubscriptionName, apnsDevType);

apnsConnection.EnqueueMessage(ddmList, pushMsg);

}

if (ifMPNSSupportsType)

{

//if MPNS supports the message type, get the list of devices of MPNS type and subscribed to a given subscription

IEnumerable<DeviceDataModel> ddmList = sds.SelectByDeviceTypeandSubscription(pushMsg.SubscriptionName, mpnsDevType);

mpnsConnection.EnqueueMessage(ddmList, pushMsg);

}

}

The MPNSConnection class implements the EnqueueMessage wherein it extracts the data specific to the message type and sends the message to Microsoft Push Notification Service. The code used to send the toast notification is based on this sample: [How to: Send a Push Notification for Windows Phone](http://msdn.microsoft.com/en-us/library/ff402545(v=vs.92).aspx) .

/\* MPNSConnection Class \*/

public override void EnqueueMessage(IEnumerable<DeviceDataModel> devices, PushMessage pm)

{

try

{

if (pm.Type == (short)PushMessageType.Toast)

EnqueWP7ToastNotification(devices, pm.message["toast"]);

else if (pm.Type == (short)PushMessageType.Raw)

EnqueWP7RawNotification(devices, pm.message["raw"]);

else if (pm.Type == (short)PushMessageType.Tile || pm.Type == (short)PushMessageType.common)

EnqueWP7TileNotification(devices, pm.message["title"], int.Parse(pm.message["count"]), pm.message["url"]);

else

return;

}

catch (Exception e)

{

if (this.NotificationError != null)

this.NotificationError(this, new NotificationFormatException(pm));

}

}

### Adding Support for Other Notification Services

If you want to extend the Common Push Notification Service to send notifications to other notification services, you will need to make the following changes:

1. Create a new subclass of the Connection class.
2. Define capabilities of the new connection class in terms of the types of messages and types of devices it supports. Implement the SupportedDeviceType property and theHandlesMessageTypes method.
3. Override one or both EnqueueMessage virtual methods, one to send notification message to one device and the other to send to a list of devices.
4. Raise events to notify error conditions.

## Implementing CPNS using ASP.net

Common Push Notification Service can be implemented using ASP.net by modifying this sample. The overall structure of the service can remain the same using the same three main components. The key changes needed are to

1. Implement the functionality of the Azure web role using ASP.net web service.
2. Implement device, subscription, and notifications using database such as SQL server or SQL CE.
3. Implement message pump using Windows Service.

The CPNS REST endpoints for device registration, subscription management and notifications will remain unchanged. However, instead of implementing the in an Azure web role, use ASP.net REST web service. Instead of Azure diagnostics traces, use ASP.net trace logs.

Various classes used to encapsulate device information, subscription data and notification messages, can remain unchanged. However, the device manager, device subscription manager and subscription information manager will need to be changed to persist the information using database. Functionality provided by Azure tables will have to be implemented using SQL tables. The notification message queue will also have to be implemented using SQL tables.

The functionality provided by the Azure worker role will need to be implemented using Windows service. Windows service will need to periodically query the SQL table for new and unprocessed messages and forward the message just like the Azure worker role does. The actual processing of messages will remain the same. Instead of using Azure diagnostics, the code will need to use Windows eventing to log errors and trace data.

## Scalability of the Service

The Common Push Notification Service can scale to handle large number of notification messages using Azure . Mobile application data provider service needs to send only one message for the specific subscription to the Azure notification service. The Common Push Notification Service, in turn, sends notification messages for each device that is signed up for the subscription. The notifications to a large number of devices can be handled by the CPNS, freeing the application data provider from sending messages to each device. Another advantage is that the burst processing and network traffic can be handled by the Common Push Notification Service by deploying additional worker roles, which can work in parallel to handle the extra workload.

# Conclusions

The Common Push Notification Service provides an easy way to manage push notifications to iPhone, Android and Windows Phone 7 devices It frees up mobile application developers from the need to manage devices or to implement separate data provide services for each type of devices. Using the Common Push Notification Service, they can send notifications to subscribing devices.

It is easy to integrate Common Push Notification Service with iPhone, Android and Windows Phone 7 applications. The REST APIs used by the Common Push Notification Service are very simple and can be integrated with a few lines of code.

Common Push Notification service provides a modular architecture making it easy to add support for additional device types, message types, and push notification protocols. CPNS sample can be modified to implement it using ASP.net.

## Resources

1. [Introduction to the Windows Azure Platform](http://msdn.microsoft.com/en-us/library/ff803364.aspx)
2. [Developing for Windows Azure](http://msdn.microsoft.com/en-us/windowsazure/default.aspx)
3. [Developing and Deploying Windows Azure Apps in Visual Studio 2010](http://msdn.microsoft.com/en-us/magazine/ee336122.aspx)
4. [Apple Push Notification Service](http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/ApplePushService/ApplePushService.html)
5. [Push Notifications Overview for the Windows Phone](http://msdn.microsoft.com/en-us/library/ff402558(v=vs.92).aspx)
6. [Introduction to Windows Phone 7 Notifications for iPhone Developers](http://windowsphone.interoperabilitybridges.com/articles/chapter-8-introduction-to-windows-phone-7-notifications-for-iphone-developers)
7. [Cloud to Device Messaging Framework](http://code.google.com/android/c2dm/)
8. [App Hub](http://create.msdn.com/en-US)
9. [Windows Azure How-to Topics](http://msdn.microsoft.com/library/gg432998.aspx)
10. [How to: Deploy a Windows Azure Application](http://msdn.microsoft.com/en-us/library/gg432994.aspx)”
11. [Deploying Applications to Windows Azure](http://programming4.us/desktop/1220.aspx)