Alive with Activity, Part 3: Push Notifications and Windows Azure Mobile Services

Kraig Brockschmidt

In Part 1 of this series we explored what “aliveness” means to a user and how apps participate in creating that experience. In Part 2 we looked at how to write and debug web services to support periodic updates for live tiles. Here in Part 3 we’ll complete the story by understanding how to deliver tile updates, toasts, and raw notifications to specific client devices on demand through the Windows Push Notification Service (WNS), and how Windows Azure Mobile Services simplifies the whole process.

# Push Notifications

Periodic updates, as we saw in Part 2, are initiated from the client side and provide a pollingor “pull” method of updating a tile or badge. “Push” notifications happen when a service directly sends an update to a device, where that update can be specific to a user, an app, and even a secondary tile.

Unlike polling, push notifications can happen at any time with much greater frequency, though be aware that Windows will throttle the amount of push notification traffic on a device if it’s on battery power, in connected standby mode, or if notification traffic becomes excessive. This means there is no guarantee that all notifications will be delivered (especially if the device is turned off).

So avoid thinking that you can use push notifications to implement a clock tile or other tile gadgets with a similar kind of frequency or time resolution. Instead, think about how you can use push notifications to connect tiles and notifications to a back-end service that has interesting and meaningful information to convey to the user, which invites them to re-engage with your app.

Before we dive into the details, understand that there are two distinct kinds of push notifications:

1. XML updates that contain tile or badge updates, or toast notification payloads: Windows can process such push notifications and issue the update or toast on behalf of an app. An app can also handle these notifications directly, if desired.
2. Binary or *raw* notifications that contain whatever data the service wants to send: these *must* be handled by app-specific code because Windows won’t know what to do with the data otherwise. See [Guidelines and checklist for raw notifications](Guidelines%20and%20checklist%20for%20raw%20notifications) for details such as size limits (5KB) and encoding (base64).

In both cases, a running (foreground) app can handle push notifications directly through the [PushNotificationChannel](http://msdn.microsoft.com/en-us/library/windows/apps/br241283.aspx) class and its [PushNotificationReceived](http://msdn.microsoft.com/en-us/library/windows/apps/windows.networking.pushnotifications.pushnotificationchannel.pushnotificationreceived.aspx) event. For XML payloads, the app could modify the contents, change tags, and so forth, before issuing it locally (or choosing to ignore it). For raw notifications, the app will process the contents and then decide which of notifications to issue, if any.

If the app is suspended or not running, it can also provide a background task for this same purpose. This requires that the app request and be granted lock screen access, which then allows such app-specific code to run when a notification is received.

A background task typically does one or two things when the notification arrives. First, it might save some relevant information from the notification into local app data, where the app can retrieve it when it’s next activated or resumed. Second, the background task can issue local tile and badge updates and/or toast notifications.

To understand raw notifications better, consider a typical email app. When its back-end service detects new messages for a user, it sends a raw push notification to WNS that contains a number of email message headers. The service does so using a *channel URI* that’s connected to the specific app on that user’s specific device.

WNS then attempts to push that notification to the client. Assuming it succeeds, Windows receives that notification and looks for the app that’s associated with the channel URI in question. If it cannot find a suitable app, the notification is ignored. If an app exists and it’s running, Windows fires the *PushNotificationReceived* event, otherwise Windows looks for and invokes an available background task for that app.

Either way the raw notification ends up in the hands of some app code, which then processes the data, issues a badge update to the app tile to indicate the number of new messages, and issues up to five cycling tile updates with message headers. It can also issue toast notifications for each new message that’s arrived, or at least one that indicates there’s new mail.

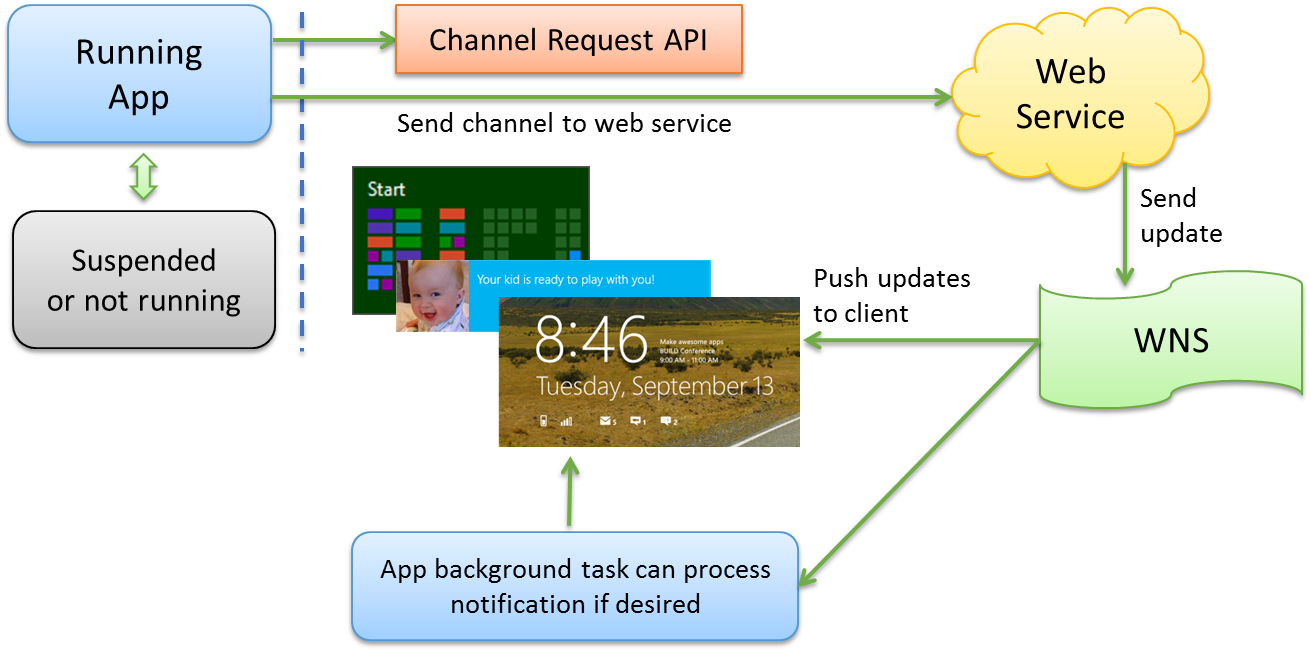
As a result, toasts tell the user that new email has arrived, and the app’s tile on the Start screen provides a quick and immediate view of new mail activity.

For more information on these client-side event handlers and background tasks, refer to the [Raw notifications sample](http://code.msdn.microsoft.com/windowsapps/Raw-notifications-sample-3bc28c5d), the [Being productive in the background—background tasks](http://blogs.msdn.com/b/windowsappdev/archive/2012/05/24/being-productive-in-the-background-background-tasks.aspx) post on this blog, and the [Background networking whitepaper](http://www.microsoft.com/en-us/download/details.aspx?id=28999). For our purposes here, let’s turn now to the service side of the story.

# Working with Windows Push Notification Service (WNS)

Push notifications are a very sophisticated capability. Through the cooperation of Windows, apps, services, and WNS, it’s possible to deliver user-specific data to a specific app tile (or toast or raw notification handler) on a specific device for a specific user.

The relationships between all these pieces is illustrated below.



Clearly, some wiring is necessary to make all this work harmoniously:

1. You (the developer) register the app in the Windows Store to use push notifications. This provides an SID and client secret that the service uses to authenticate with WNS (these bits should never be stored on client devices for security purposes).
2. At runtime, the app requests a WNS channel URI from Windows for each of its live tiles (primary and secondary), or one for raw notifications. An app must also refresh these channel URIs every 30 days, for which you can use another background task.
3. The app’s service provides a URI through which the app can upload those channel URIs along with any data that describes its use (such as location for weather updates or a specific user account and activity). Upon receipt, the service stores those channel URIs and associated data for later use.
4. The service monitors its backend for changes that apply to each particular user/device/app/tile combination. When it detects a condition that triggers a notification for a particular channel, it builds the content of that notification (XML or raw), authenticates itself with WNS using the SID and client secret, and then sends the notification to WNS along with the channel URI.

Let’s look at each step in detail. (And just as a preview, if dealing with HTTP requests makes you nervous, know that Windows Azure Mobile Services will relieve you from many of the details, as we’ll see.)

## App Registration with the Windows Store

For exact instructions on obtaining the SID and client secret for your service, refer to [How to authenticate with the Windows Push Notification Service (WNS)](http://msdn.microsoft.com/library/windows/apps/hh465407.aspx) on the Windows Developer Center. The SID is what identifies your app with WNS, and the client secret is what your service uses to tell WNS that it’s allowed to send notifications for your app. Again, these should only be stored in the service.

Note that Step 4 in the topic linked above, “Send the cloud server's credentials to WNS,” is something you do only when your service sends a push notification. We’ll come back to that shortly, because at this stage your service lacks the key piece that it needs to send a notification, namely a channel URI.

## Obtaining and Refreshing Channel URIs

The client app obtains channel URIs at runtime through the [Windows.Networking.PushNotifications.PushNotificationChannelManager](http://msdn.microsoft.com/en-us/library/windows/apps/windows.networking.pushnotifications.pushnotificationchannelmanager.aspx) object. This object has only two methods:

* *createPushNotificationChannelForApplicationAsync*: creates a channel URI for the app’s primary tile as well as toasts and raw notifications.
* *createPushNotificationChannelForSecondaryTileAsync:* creates a channel URI for a specific secondary tile identified by a *tileId* argument.

The result of both async operations is a [PushNotificationChannel](http://msdn.microsoft.com/en-us/library/windows/apps/windows.networking.pushnotifications.pushnotificationchannel.aspx) object. This object contains the channel URI in its *Uri* property, along with an *ExpirationTime* to indicate the deadline for refreshing that channel. A *Close* method specifically terminates the channel if needed, and most importantly is the *PushNotificationReceived* event, which is again what’s fired when the app is in the foreground and a push notification is received through this channel.

The lifetime of a channel URI is 30 days, after which WNS will reject any requests made for that channel. App code thus needs to refresh those URIs with the *create* methods above at least once every 30 days and send those URIs to its service. A good strategy for this is as follows:

* On first launch, request a channel URI and save the string in the *Uri* property in your local app data. Because channel URIs are specific to a device, they should not be stored in your roaming app data.
* On subsequent launches, request a channel URI again and compare it to the one previously saved. If it’s different, send it to the service, or send it and let the service replace an older one if necessary.
* Also perform the previous step in your app’s *Resuming* handler (see [Launching, resuming, and multitasking](http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh770837.aspx) in the docs), because it’s possible that the app might have been suspended for more than 30 days.
* If you’re concerned the app won’t be run within 30 days, implement a background task with a maintenance trigger to run every few days or once a week. For details, refer again to [Being productive in the background—background tasks](http://blogs.msdn.com/b/windowsappdev/archive/2012/05/24/being-productive-in-the-background-background-tasks.aspx); the background task in this case will just execute the same code as the app to request a channel and send it to your service.

## Sending Channel URIs to the Service

Typically, push notification channels work with user-specific updates like email status, instant messages, and other personalized information. It’s generally unlikely that your service will need to broadcast the same notification to every user and/or every tile. For this reason, the service needs to associate each channel URI with more specific information. For an email app, the user’s id is paramount, as that would specify the account to check. A weather app, on the other hand, would likely associate each channel URI with a specific latitude and longitude, such that each tile (primary and secondary) would reflect a distinct location.

The app, then, must include these details when it sends a channel URI to its service, and the service must store them for later use.

Where user identity is concerned, it’s best for the app to authenticate the user with the service separately, using service-specific credentials or through an OAuth provider such as Facebook, Twitter, Google, or the user’s Microsoft Account (using OAuth is helpful with Windows Azure Mobile Services, as we’ll see later). If for some reason that’s not possible, be sure to encrypt any user id you send to the service or make sure to send it over HTTPS.

Whatever the case, how you send all this information to your service (in headers, through data in the message body, or as parameters on the service URI) is up to you. This part of the communication is strictly between the app and its service.

As a simple example, let’s say we have a service with a page called *receiveuri.aspx* (as we’ll see in the next section), the full address of which is in a variable called *url*. The following code requests a primary channel URI from Windows for the app and posts it to that page via HTTP. (This code derived and simplified from the [Push and periodic notifications client side sample](http://code.msdn.microsoft.com/windowsapps/Push-and-periodic-de225603) where the *itemId* variable, defined elsewhere, is used to identify secondary tiles; the sample also has a C++ variant, not shown here):

JavaScript:

Windows.Networking.PushNotifications.PushNotificationChannelManager

.createPushNotificationChannelForApplicationAsync()

.done(function (channel) {

//Typically save the channel URI to appdata here.

WinJS.xhr({ type: "POST", url:url,

headers: { "Content-Type": "application/x-www-form-urlencoded" },

data: "channelUri=" + encodeURIComponent(channel.uri)   
 + "&itemId=" + encodeURIComponent(itemId)

}).done(function (request) {

//Typically update the channel URI in app data here.

}, function (e) {

//Error handler

});

});

C#:

using Windows.Networking.PushNotifications;

PushNotificationChannel channel = await PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsync();

HttpWebRequest webRequest = (HttpWebRequest)HttpWebRequest.Create(url);

webRequest.Method = "POST";

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

byte[] channelUriInBytes = Encoding.UTF8.GetBytes("ChannelUri=" + WebUtility.UrlEncode(newChannel.Uri) + "&ItemId=" + WebUtility.UrlEncode(itemId));

Task<Stream> requestTask = webRequest.GetRequestStreamAsync();

using (Stream requestStream = requestTask.Result)

{

requestStream.Write(channelUriInBytes, 0, channelUriInBytes.Length);

}

The following ASP.NET code is a basic implementation of a *receiveuri.aspx* page that processes this HTTP POST, making sure it received a valid channel URI, a user, and some identifier for the item.

I emphasize the “basic” nature of this code because, as you can see, the *SaveChannel* function simply writes the contents of the request into a fixed text file, which clearly won’t scale beyond a single user! A real service, of course, would employ a database for this purpose, but the structure here will be similar.

ASP.NET (C#)

<%@ Page Language="C#" AutoEventWireup="true" %>

<script runat="server">

protected void Page\_Load(object sender, EventArgs e)

{

//Output page header

Response.Write("<!DOCTYPE html>\n<head>\n<title>Register Channel URI</title>\n</head>\n<html>\n<body>");

//If called with HTTP GET (as from a browser), just show a message.

if (Request.HttpMethod == "GET")

{

Response.StatusCode = 400;

Response.Write("<p>This page is set up to receive channel URIs from a push notification client app.</p>");

Response.Write("</body></html>");

return;

}

if (Request.HttpMethod != "POST") {

Response.StatusCode = 400;

Response.Status = "400 This page only accepts POSTs.";

Response.Write("<p>This page only accepts POSTs.</p>");

Response.Write("</body></html>");

return;

}

//Otherwise assume a POST and check for parameters

try

{

//channelUri and itemId are the values posted from the Push and Periodic Notifications Sample in the Windows 8 SDK

if (Request.Params["channelUri"] != null && Request.Params["itemId"] != null)

{

// Obtain the values, along with the user string

string uri = Request.Params["channelUri"];

string itemId = Request.Params["itemId"];

string user = Request.Params["LOGON\_USER"];

//TODO: validate the parameters and return 400 if not.

//Output in response

Response.Write("<p>Saved channel data:</p><p>channelUri = " + uri + "<br/>" + "itemId = " + itemId + "user = " + user + "</p>");

//The service should save the URI and itemId here, along with any other unique data from the app such as the user;

SaveChannel(uri, itemId, user);

Response.Write("</body></html>");

}

}

catch (Exception ex)

{

Trace.Write(ex.Message);

Response.StatusCode = 500;

Response.StatusDescription = ex.Message;

Response.End();

}

}

</script>

protected void SaveChannel(String uri, String itemId, String user)

{

//Typically this would be saved to a database of some kind; to keep this demonstration very simple, we'll just use

//the complete hack of writing the data to a file, paying no heed to overwriting previous data.

//If running in the debugger on localhost, this will save to the project folder

string saveLocation = Server.MapPath(".") + "\\" + "channeldata\_aspx.txt";

string data = uri + "~" + itemId + "~" + user;

System.Text.ASCIIEncoding encoding = new System.Text.ASCIIEncoding();

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

using (System.IO.FileStream fs = new System.IO.FileStream(saveLocation, System.IO.FileMode.Create))

{

fs.Write(dataBytes, 0, data.Length);

}

return;

}

The code for this service can be found in Chapter 13 of my free ebook, [*Programming Windows 8 Apps in HTML, CSS, and JavaScript*](http://go.microsoft.com/fwlink/?LinkID=269853)*,* specifically in the HelloTiles sample of the companion content. It is designed to work with the client-side SDK sample referenced earlier. If you run HelloTiles in the debugger (Visual Studio 2012 Ultimate or Visual Studio Express 2012 for Web) with the localhost enabled, you’ll have URL like *http://localhost:52568/HelloTiles/receiveuri.aspx* that you can paste into the client-side SDK sample. When the sample then makes a request to that URL, you’ll stop on any breakpoints in the service and can step through the code.

## Sending the Notification

In a real service, you’ll have some kind of ongoing process that monitors its data sources and sends push notifications to specific users when appropriate. This might happen in a number of ways:

* A scheduled job can check for weather alerts for a specific location perhaps once every 15-30 minutes, depending on how often those alerts are issued, and issue push notifications in response.
* Another service, like a messaging back-end, might make a request to a page on your server when a new message is available. That page can then issue appropriate notification.
* Users might be interacting with web pages on your server, and their activities trigger push notifications to specific channels.

In short, there are any number of triggers for push notifications, depending entirely on the nature of your back-end service or website. For the purposes of this article, the same HelloTiles sample service from Chapter 13 of [*Programming Windows 8 Apps in HTML, CSS, and JavaScript*](http://go.microsoft.com/fwlink/?LinkID=269853) has a page called *sendBadgeToWNS.aspx* that sends a push notification whenever you visit that page in a browser, using whatever channel URI was saved in the text file earlier. A real service will perform some kind of database lookup to obtain channel URIs instead of reading the contents of a file, but again the overall structure is very similar.

ASP.NET Page:

<%@ Page Language="C#" AutoEventWireup="true" CodeFile="sendBadgeToWNS.aspx.cs" Inherits="sendBadgeToWNS" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">

<title>Send WNS Update</title>

</head>

<body>

<p>Sending badge update to WNS</p>

</body>

</html>

C# code-behind:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Net;

using System.IO;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Json;

using System.Text;

public partial class sendBadgeToWNS : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

//Load our data that was previously saved. A real service would do a database lookup here

//with user- or tile-specific criteria.

string loadLocation = Server.MapPath(".") + "\\" + "channeldata\_aspx.txt";

byte[] dataBytes;

using (System.IO.FileStream fs = new System.IO.FileStream(loadLocation, System.IO.FileMode.Open))

{

dataBytes = new byte[fs.Length];

fs.Read(dataBytes, 0, dataBytes.Length);

}

if (dataBytes.Length == 0)

{

return;

}

System.Text.ASCIIEncoding encoding = new System.Text.ASCIIEncoding();

string data = encoding.GetString(dataBytes);

string[] values = data.Split(new Char[] { '~' });

string uri = values[0]; //Channel URI

string secret = "9ttsZT0JgHAFveYahK1B6jQbvMOIWYbm";

string sid = "ms-app://s-1-15-2-2676450768-845737348-110814325-22306146-1119600341-293560589-2707026538";

//Create some simple XML for a badge update

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

xml += "<badge value='alert'/>";

PostToWns(secret, sid, uri, xml, "wns/badge");

}

}

All we’re doing here is retrieving the channel URI, building the XML payload, and then authenticating with WNS and doing an HTTP POST to that channel URI. The last two steps are done in the *PostToWns* function that comes from the [QuickStart: Sending a push notification](http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh868252.aspx) topic on the Windows Developer Center. I’m omitting a full listing because the bulk of it is just authenticating with WNS via OAuth (at <https://login.live.com/accesstoken.srf>) using you client secret and SID from the Windows Store. The result of this authentication is an access token that’s then included in the HTTP POST we send to the channel URI:

C#:

public string PostToWns(string secret, string sid, string uri, string xml, string type = "wns/badge")

{

try

{

// You should cache this access token

var accessToken = GetAccessToken(secret, sid);

byte[] contentInBytes = Encoding.UTF8.GetBytes(xml);

// uri is the channel URI

HttpWebRequest request = HttpWebRequest.Create(uri) as HttpWebRequest;

request.Method = "POST";

request.Headers.Add("X-WNS-Type", type);

request.Headers.Add("Authorization", String.Format("Bearer {0}", accessToken.AccessToken));

using (Stream requestStream = request.GetRequestStream())

requestStream.Write(contentInBytes, 0, contentInBytes.Length);

using (HttpWebResponse webResponse = (HttpWebResponse)request.GetResponse())

return webResponse.StatusCode.ToString();

}

catch (WebException webException)

{

// Implements a maximum retry policy (omitted)

}

}

In this example, note that the *X-WNS-Type* header in the HTTP request is set to *wns/badge* and the *Content-Type* header defaults to *text/xml*. For tiles, the type should be *wns/tile*; toasts use *wns/toast*. With raw notifications, use the type *wns/raw* and set *Content-Type* to *application/octet-stream*. For full details on headers, refer to the [Push notification service request and response headers](http://msdn.microsoft.com/en-us/library/windows/apps/hh465435.aspx) page in the documentation.

## Push Notification Failures

Of course, sending an HTTP request like this might not always work, and there are a number of reasons why WNS will respond with something other than a 200 code (success). For specific details, see the “Response code” section of the [Push notification service request and response headers](http://msdn.microsoft.com/en-us/library/windows/apps/hh465435.aspx); here are the more common errors and causes:

* The channel URI is invalid (404 Not Found) or expired (410 Gone). In these cases the service should remove the channel URI from its database and not make any further requests to it.
* The client secret and SID might be invalid (401 Unauthorized) or there’s a mismatch between the app’s package ID in its manifest and the one in the Windows Store (403 Forbidden). The best way to assure that these match is to use the *Store > Associate App with Store* menu command in Visual Studio (this command is found on the *Project* menu in Visual Studio 2012 Ultimate).
* The raw notification payload is over 5KB (413 Request Entity Too Large)
* The client could be offline, in which case WNS will automatically try again, but eventually report a failure. For XML notifications, the default behavior is that push notifications are cached and delivered when the client comes back online. For raw notifications, caching is disabled by default; you can change this by setting the *X-WNS-Cache-Policy* header to *cache* in the request to the channel URI.

For other errors (400 Bad Request), make sure that XML payloads contain UTF-8 encoded text, and that raw notifications are in base64 with the *Content-Type* header set to *application/octet-stream*. It’s also possible that WNS is throttling delivery because you’re simply trying to send too many push notifications within a specific time period.

A raw push notification might also be rejected if the app isn’t present on the lock screen and the device is in connected standby. Because Windows blocks raw notifications to non-lock screen apps in this state (and anytime the app isn’t in the foreground), WNS has optimizations to drop notifications that it knows won’t be delivered.

# Windows Azure Mobile Services

Now that we’ve gone through the intricate details of working with push notifications, even omitting the question of storage, you’re probably wondering, “Is there any way to make all this simpler?” Just imagine what it would take to manage potentially thousands or millions of channel URIs for a hopefully large and expanding customer base!

Fortunately, you’re not the first to ask such questions. Besides third party solutions such as those offered by Urban Airship, Windows Azure Mobile Services can make your life a whole lot easier.

Windows Azure Mobile Services, which I’ll just abbreviate as AMS, provides a pre-built solution (basically a number of REST endpoints) for most of the service details we’ve been discussing. A “mobile service” is basically one that manages a database on your behalf and provides library functions to easily send payloads to WNS. An introduction to AMS can be found on [Add cloud to your app with Windows Azure Mobile Services](http://blogs.msdn.com/b/windowsappdev/archive/2012/08/28/add-cloud-to-your-app-with-windows-azure-mobile-services.aspx).

For push notifications in particular, first obtain the client-side library in the [Windows Azure Mobile Services SDK for Windows 8](http://go.microsoft.com/fwlink/?LinkId=257545&clcid=0x409). Then refer to [Get started with push notifications in Mobile Services](http://www.windowsazure.com/en-us/develop/mobile/tutorials/get-started-with-push-dotnet/) (note that there’s a [JavaScript version of the topic](http://www.windowsazure.com/en-us/develop/mobile/tutorials/get-started-with-push-js/) too), which describes how AMS helps you fulfill all the wiring requirements we saw earlier:

* App Registration with the Windows Store: Once you obtain the client secret and SID for your app from the Windows Store, save those values in the mobile service configuration. See the “Register your app for the Windows Store” section in the [Get started](http://www.windowsazure.com/en-us/develop/mobile/tutorials/get-started-with-push-dotnet/) topic just mentioned.
* Obtaining and Refreshing Channel URIs: requesting and managing channel URIs in the app is purely a client-side concern, and is the same as before.
* Sending Channel URIs to the Service: this step becomes far easier with AMS. First you create a database table in the mobile service (through the Windows Azure portal). Then the app can simply insert records into that table with channel URIs and any other key information you need to attach. The AMS client library takes care of the HTTP request behind the scenes, and even updates the client-side record with any changes made on the server. Furthermore, AMS can automatically handle authentication through the user’s Microsoft Account or through three other OAuth providers—Facebook, Twitter, or Google—if you’ve registered your app with one of them. See [Get started with authentication in Mobile Services](http://www.windowsazure.com/en-us/develop/mobile/tutorials/get-started-with-users-js/).
* Sending the Notification: within the mobile service, you can attach scripts (written in the variant of JavaScript known as Node.js) to database operations, and also created scheduled jobs in JavaScript. In these scripts, a simple call to the *push.wns* object with a channel URI and a payload generates the necessary HTTP request to the channel. It’s also a simple matter to capture push failures and record the response with *console.log*. Those logs are easily reviewed on the Windows Azure portal.

For extensive details, see these two sample tutorials: [Tile, Toast, and Badge Push Notifications using Windows Azure Mobile Services](http://code.msdn.microsoft.com/windowsapps/Tile-Toast-and-Badge-Push-90ee6ff1) and [Raw Notifications using Windows Azure Mobile Services](http://code.msdn.microsoft.com/windowsapps/Raw-Notifications-using-adb63a01). Here, instead of repeating all those instructions, let’s review a few of the highlights.

When you set up a mobile service, it will have a specific service URL. This is what you use when creating an instance of the *MobileServiceClient* object in the AMS SDK:

JavaScript:

var mobileService = new Microsoft.WindowsAzure.MobileServices.MobileServiceClient(

"https://{mobile-service-url}.azure-mobile.net/",

"{mobile-service-key}");

C#:

using Microsoft.WindowsAzure.MobileServices;

MobileServiceClient MobileService = new MobileServiceClient(

"https://{mobile-service-url}.azure-mobile.net/",

"{mobile-service-key}");

C++ (taken from [an additional sample](http://code.msdn.microsoft.com/windowsapps/C-Windows-Store-apps-and-ab1ea114))

using namespace Microsoft::WindowsAzure::MobileServices;

auto MobileService = ref new MobileServiceClient(  
 ref new Uri(L" https://{mobile-service-url}.azure-mobile.net/"),

ref new String(L"{mobile-service-key}"));

This class encapsulates all the HTTP communication with the service, relieving you from all that low-level plumbing that you probably don’t want to think about.

To authenticate with a particular OAuth provider, use the [*login*](http://msdn.microsoft.com/en-us/library/windowsazure/jj554236.aspx) or [*LoginAsync*](http://msdn.microsoft.com/en-us/library/windowsazure/jj553650.aspx) method, the result of which is a [*User*](http://msdn.microsoft.com/en-us/library/windowsazure/jj554220.aspx) object that provides that information to the app. (Once authenticated, *CurrentUser* property of the client object will also contain the user id.) When you authenticate the mobile service directly like this, the service will have access to the user id and there’s no need for the client to send it explicitly:

JavaScript:

mobileService.login("facebook").done(function (user) { /\* ... \*/ });

mobileService.login("twitter").done(function (user) { /\* ... \*/ });

mobileService.login("google").done(function (user) { /\* ... \*/ });

mobileService.login("microsoftaccount").done(function (user) { /\* ... \*/ });

C#:

MobileServiceUser user = await MobileService.LoginAsync(MobileServiceAuthenticationProvider.Facebook);

MobileServiceUser user = await MobileService.LoginAsync(MobileServiceAuthenticationProvider.Twitter);

MobileServiceUser user = await MobileService.LoginAsync(MobileServiceAuthenticationProvider.Google);

MobileServiceUser user = await MobileService.LoginAsync(MobileServiceAuthenticationProvider.MicrosoftAccount);

C++

task<MobileServiceUser^> (MobileService->LoginAsync(MobileServiceAuthenticationProvider::Facebook))

.then([this](MobileServiceUser^ user) { /\* \*/ } );

task<MobileServiceUser^> (MobileService->LoginAsync(MobileServiceAuthenticationProvider::Twitter))

.then([this](MobileServiceUser^ user) { /\* \*/ } );

task<MobileServiceUser^> (MobileService->LoginAsync(MobileServiceAuthenticationProvider::Google))

.then([this](MobileServiceUser^ user) { /\* \*/ } );

task<MobileServiceUser^> (MobileService->LoginAsync(MobileServiceAuthenticationProvider::MicrosoftAccount))

.then([this](MobileServiceUser^ user) { /\* \*/ } );

Sending a channel URI to the service is again a simple matter of storing a record in the service’s database, and the client object makes the HTTP requests. To do this, just request the database object and insert the record as shown in the examples below from the push notification tutorial sample linked earlier. In each code snippet, assume that *ch* contains the *PushNotificationChannel* object from the WinRT APIs. You can also include any other properties in the *channel* object that’s built up, such as a secondary tile id or other data that identifies the channel’s purpose.

JavaScript:

var channelTable = MobileServicesSample.mobileService.getTable('Channels');

var channel = {

uri: ch.uri,

expirationTime: ch.expirationTime.

};

channelTable.insert(channel).done(function (item) {

},

function () {

// Error on the insertion.

});

}

C#:

var channel = new Channel { Uri = ch.Uri, ExpirationTime = ch.ExpirationTime };

var channelTable = privateClient.GetTable<Channel>();

if (ApplicationData.Current.LocalSettings.Values["ChannelId"] == null)

{

// Use try/catch block here to handle exceptions

await channelTable.InsertAsync(channel);

}

C++:

auto channel = ref new JsonObject();

channel->Insert(L"Uri", JsonValue::CreateStringValue(ch->Uri));

channel->Insert(L"ExpirationTime", JsonValue::CreateBooleanValue(ch->ExpirationTime));

auto table = MobileService->GetTable("Channel");

task<IJsonValue^> (table->InsertAsync(channel))

.then([this, item](IJsonValue^ V) { /\* ... \*/ });

Note that once the channel record has been inserted, any changes or additions that the service might have made to that record will be reflected in the client.

In addition, if you misspell the name of the database in the *GetTable*/*getTable* call, you won’t see any exceptions until you try to insert a record. This can be a hard bug to track down, so if you’re convinced everything should be working, but it isn’t, check that database name.

Now again, these insertions on the client side are translated into HTTP requests to the service, but even on the service side this is also hidden from you. Instead of receiving and processing the requests, you attach custom scripts to each database operation (insert, read, update, and delete).

These scripts are written as JavaScript functions using the same intrinsic objects and methods that are available in Node.js (all of which has nothing whatsoever to do with any client-side JavaScript in the app). Each function receives appropriate parameters: *insert* and *update* receive the new record, *delete* receives the id of the item, and *read* receives a query. All of the functions also receive a *user* object, if the app authenticated the user with the mobile service, and a *request* object that lets you execute the operation and generate what becomes the HTTP response.

The simplest (and default) *insert* script, just executes the request and inserts the record, *item*, as-is:

function insert(item, user, request) {

request.execute();

}

If you want to attach a timestamp and the user id to the record, it’s as easy as adding those properties to the *item* parameter prior to executing the request:

function insert(item, user, request) {

item.user = user.userId;

item.createdAt = new Date();

request.execute();

}

Note that the changes made to *item* in these scripts before insertion into the database are automatically propagated back to the client. In the code above, the *channel* object in the client would contain *user* and *createdAt* properties once the insertion succeeds. Very convenient!

The service scripts can also perform additional actions after *request.execute*, especially in response to success or failure, but I’ll leave such details to the [Server script example how tos documentation](http://msdn.microsoft.com/en-us/library/windowsazure/jj591477.aspx).

To return now to push notifications, saving channel URIs into a table is just one part of the equation, and the service may or may not send notifications in response to this particular event. It’s more likely that the service will have other tables with additional information, where operations performed on those tables trigger notifications to some subset of channel URIs. We’ll discuss a few examples in the next section. Whatever the case may be, you send a push notification from a script using the [*push.wns*](https://github.com/tjanczuk/wns/blob/master/README.md)object. This again has many methods to send specific types of updates (including raw), where tiles, toasts, and badges work directly through names methods that match the available templates. For example:

push.wns.sendTileSquarePeekImageAndText02(channel.uri, {

image1src: baseImageUrl + "image1.png",

text1: "Notification Received",

text2: item.text

}, {

success: function (pushResponse) {

console.log("Sent Tile Square:", pushResponse);

},

error: function (err) {

console.log("Error sending tile:", err);

}

});

push.wns.sendToastImageAndText02(channel.uri, {

image1src: baseImageUrl + "image2.png",

text1: "Notification Received",

text2: item.text

}, {

success: function (pushResponse) {

console.log("Sent Toast:", pushResponse);

}

});

push.wns.sendBadge(channel.uri, {

value: value,

text1: "Notification Received"

}, {

success: function (pushResponse) {

console.log("Sent Badge:", pushResponse);

}

});

Again, the *console.log* function will create an entry in the logs that you can view on the Azure Mobile Services portal, and you’ll typically want to include log calls in error handlers as shown with the tile notification above.

You might have noticed that the *send\** methods are each tied to a specific template, and for tiles it means that wide and square payloads must be sent separately as two notifications. Remember that you almost always want to send both sizes together because the user controls how that tile appears on their Start screen. With the template specific *send* functions of *push.wns*, then, it means making two sequential calls which each generate a separate push notification.

To combine multiple updates, which includes sending multiple tile updates with different tags at one time or sending multiple toasts, use the *push.wns.sendTile* and *push.wns.sendToast* methods. For example:

var channel = '{channel\_url}';

push.wns.sendTile(channel,

{ type: 'TileSquareText04', text1: 'Hello' },

{ type: 'TileWideText09', text1: 'Hello', text2: 'How are you?' },

{ client\_id: '{your Package Security Identifier}', client\_secret: '{your Client Secret}' },

function (error, result) {

// ...

});

On an even lower level, *push.wns.send* lets you be very exact with the notification contents; *push.wns.sendRaw* is also there for raw notifications. For all the details, refer again to the [*push.wns*](https://github.com/tjanczuk/wns/blob/master/README.md) object documentation.

# Real-World Scenarios with Windows Azure Mobile Services

The sample app in the [Tile, Toast, and Badge Push Notifications using Windows Azure Mobile Services](http://code.msdn.microsoft.com/windowsapps/Tile-Toast-and-Badge-Push-90ee6ff1) tutorial is instructive, as it shows how to send push notifications in response to a new message being inserted into a database table. However, this means that the app ends up sending a push notification to itself, which it wouldn’t typically need to do (except perhaps to send notifications to the same app on the user’s other devices).

What’s more likely to happen is that the service will send push notifications in response to events that occur outside the app/tile that ultimately receives those notifications. Consider a few scenarios:

**Using Social Networks**: Apps can employ a user’s social network to implement features like issuing challenges to one’s friends. For example, when one user sets a new high score in a game, you might want to issue challenges via tile updates or toasts to the user’s friends who also have that game installed. The same thing can happen in fitness apps where you might be posting a new best time for a certain kind of activity.

To do this, the app can insert a new record into an appropriate mobile service table (Scores, BestTimes, etc.). Within the insert script, the service queries its database for appropriate friends of the current user, then sends notifications to those channel URIs. Additional query criteria describe the exact aspect of a game, the particular kind of exercise (for secondary tiles), and so on.

**Weather updates and alerts**: Weather apps typically allow you to assign a location to the app’s primary tile and to create secondary tiles for additional locations. The important information with each tile is the latitude and longitude of the location, which the app sends to the service along with each channel URI (by inserting into a table). To trigger an update to that channel, the service might have another process (such as a scheduled job described below) that periodically queries a central weather service for updates and alerts, and then processes those responses and inserts the appropriate messages into an alerts table in the mobile service. The insert script would then retrieve appropriate channel URIs and send the updates. Alternately, if a weather service itself allows you to register for alerts or periodic updates, then another page in the service would receive those requests (HTTP PUTs, most likely), process them, and invoke the mobile service to insert a record, thus triggering an update.

**Messaging**: Handling instant messaging would happen very much in the same way as receiving weather updates. Again, another process would monitor the receipt of new messages, such as one that checks the incoming messages periodically, or registers with the message source for alerts when new messages arrive. In either case, the arrival of a new message would trigger push notifications to the appropriate channels. In this case the channel URIs would be associated with a user’s tile for a particular kind of messaging. You’d likely use raw notifications in this case, as it’s similar to the email scenario described at the beginning of this post.

In all of these scenarios, note that it’s not actually necessary to insert anything into a database table at all. If you don’t call *request.execute* in the insertion script, then nothing ends up in the database, yet you can still perform other tasks within that script such as sending notifications. In other words, there’s no need to fill up a database with records that you aren’t going to use later on, especially because the storage will incur costs.

Note too that Azure Mobile Services has a facility for scheduling jobs, which you can read about on [Schedule recurring jobs in Mobile Services](http://www.windowsazure.com/en-us/develop/mobile/tutorials/schedule-backend-tasks/). Such jobs could routinely retrieve data from other services, process it, and insert records into the service’s database, again triggering push notifications. Likewise, as we pointed out in Part 2 of this series, other web pages and mobile apps can also make changes to that database and trigger push notifications. The database scripts in the mobile service will run in all of these cases.

# In Closing

To wrap up this series, then, we’ve very much explored the whole scope of what “alive with activity” means-for users, for developers, for apps, and for services. We’ve seen the capabilities of tile, badge, and toast updates, how to set up periodic notifications, how to use background tasks as part of this process, and the structure of services to handle periodic and push notifications. And clearly, Windows Azure Mobile Services provides a much higher-level view of the whole process of working with push notifications. It will certainly help you be much more productive—and save you many headaches, most likely!—than writing new services from scratch.

Kraig Brockschmidt  
Program Manager, Windows Ecosystem Team  
Author, [*Programming Windows 8 Apps in HTML, CSS, and JavaScript*](http://blogs.msdn.com/b/microsoft_press/archive/2012/10/29/free-ebook-programming-windows-8-apps-with-html-css-and-javascript.aspx)