Alive with Activity, Part 2: Writing and Debugging Services for Live Tiles

Kraig Brockschmidt

In Part 1 of this series <LINK> we explored the nature of tile updates, badge updates, and toast notifications that bring a sense of “aliveness” to the overall user experience of Windows 8. In particular, we saw how these notifications are composed with XML payloads that are then issued locally, from a running app or a background task, or can be supplied upon request by an online service.

Generating an XML payload and issuing it from a running app is something you can easily develop and debug within an app using Visual Studio Ultimate 2012 or Visual Studio Express 2012 for Windows 8. The [App tiles and badges sample](http://code.msdn.microsoft.com/windowsapps/App-tiles-and-badges-sample-5fc49148), [Secondary tiles sample](http://code.msdn.microsoft.com/windowsapps/Secondary-Tiles-Sample-edf2a178), and the [Scheduled notifications sample](http://code.msdn.microsoft.com/windowsapps/Scheduled-notifications-da477093), give you everything you need here.

What takes a little more effort is developing and debugging a web service to support periodic updates as well as push notifications. The client side of these stories is well-demonstrated in the [Push and periodic notifications client-side sample](http://code.msdn.microsoft.com/windowsapps/Push-and-periodic-de225603), but in order to make use of that sample at all you need some services to work with! In this post, then, we’ll specifically explore how to develop services that support periodic notifications for tile and badge updates, with a focus on using Visual Studio tools and the localhost to debug your services before deploying them to a production environment. We’ll also begin to explore how you can use Windows Azure Mobile Services for this purpose, which are also very helpful in supporting push notifications as we’ll see in Part 3.

# The Basic Nature of Services

What does a service that supports periodic tile and badge updates actually do? Let’s begin by reviewing what we mean by a “service,” because that word is often rather intimidating for developers who have primarily been focused on client apps.

In the simplest terms, a service is some piece of code sitting on some web server that runs *on that server* whenever an HTTP request is made to it. HTML pages (.htm or .html) are not like this: as there is no server-side code involved, the server simply returns that page’s text and all processing is done on the client (including running any client-side script contained in that page). However, if you have URIs to pages that end with php, asp, aspx, cgi, cshtml, or any number of other server-side extensions, then you’re effectively talking to a “service” in this most general sense.

A service is responsible to receive a client HTTP request, process whatever arguments might be included with the URI, and return an appropriate text response. For *web pages* written with technologies like PHP or ASP.NET, the response should be in the form of HTML. Services that implement web APIs, such as those of Facebook and Twitter (and thousands of others!), typically accept any number of parameters in URI query strings (or in the request header), and return either XML or JSON data in response. (And just to note, we’re talking here of services built on [representational state transfer](http://en.wikipedia.org/wiki/Representational_state_transfer), or REST, rather than those build on other protocols like [SOAP](http://en.wikipedia.org/wiki/SOAP), as REST the most common among services today.)

Thus a service that provides periodic tile and/or badge updates is simply one that exists at whatever URI an app gives to Windows for this purpose, and which responds to HTTP requests with an appropriate XML payload. That payload contains elements that correspond to any supported template, where images are referenced with other URIs (inline encoding is not supported). The specific information included in the payload can also come from any source, as we’ll talk about in the next section.

Regardless of such details, however, all such services will share a common structure that receives and processes the request, and constructs the XML response. Let’s now look at how to create that basic structure.

# Writing Services

To write and debug services you can really use whatever tools you like that support your chosen server-side language. Here we’ll focus on Visual Studio, specifically Visual Studio Ultimate 2012 and [Visual Studio Express 2012 **for Web**](http://msdn.microsoft.com/en-us/subscriptions/dd537667(v=vs.110).aspx), the latter of which is available for free alongside its Windows partner that you already know. To install it, run the [Web Platform Installer](http://www.microsoft.com/web/downloads/platform.aspx) through which you can also install a variety of other related technologies such as PHP and WebMatrix. This will allow you to create services in a variety of languages.

As a very simple example, the following code comprises a complete PHP service that responds with a badge update XML payload in which the badge value is set to the current day of the month (according to the server, mind you!). It comes from the HelloTiles sample site that’s included in Chapter 13 of my free ebook, [*Programming Windows 8 Apps with HTML, CSS, and JavaScript*](Programming%20Windows%208%20Apps%20in%20HTML,%20CSS,%20and%20JavaScript):

<?php

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

echo "<badge value='".date("j")."'/>";

?>

You can try this service directly on the Windows Azure site I set up for this purpose: <http://programmingwin8-js-ch13-hellotiles.azurewebsites.net/dayofmonthservice.php>. Click on this link, and you’ll see that the XML that comes back is something like this:

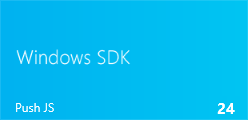
<?xml version="1.0" encoding="UTF-8"?>

<badge value="24"/>

Try this same URI in the [Push and periodic notifications client-side sample](http://code.msdn.microsoft.com/windowsapps/Push-and-periodic-de225603), scenario 5, “Polling for badge updates.” When you first run the app (within Visual Studio Express for Windows), its tile appears on the start screen as follows:



Now enter the URI above into the text box in scenario 5, press the Start periodic updates button, and, assuming you have network connectivity, you’ll soon see a numerical badge appear on the sample’s tile:



Note that Windows will attempt to poll for an update as soon as the app starts periodic updates, then continues to poll at the specified interval thereafter.

For a fuller PHP example, see [Creating a great tile experience, part 2,](http://blogs.msdn.com/b/windowsappdev/archive/2012/04/18/creating-a-great-tile-experience-part-2.aspx) which shows more customization of a tile update. In that example, the hypothetical *get\_trucks\_from\_database* function queries a database using a zip code that’s included in the URI’s query string parameters, then builds the XML response with the results of that query.

There’s much more the service could do along these lines. For example:

* With the day-of-month PHP service above, the app could indicate its local time zone in the query string to get a more accurate date, since the server can easily be located in another time zone.
* A weather service could use latitude and longitude values in the URI to retrieve current conditions for that location.
* The service could generate images on the fly and store them on a web server, then insert appropriate URIs into the XML payload.
* The service would send its own requests to other services to obtain additional data, customized with the query string parameters (more on this later).
* If an app enables the queue for tile updates (see the [*EnableNotificationQueue*](http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.notifications.tileupdater.enablenotificationqueue.aspx) method), it can specify up to five separate URIs to poll for periodic updates, as demonstrated in scenario 4 of the Push and periodic notifications client-side sample. The tile update queue will be populated with an update from each URI. Each one of those URIs, of course, can have its own query string for further customization such that the same service could accommodate all the requests itself.
* An app could include a user id in the query string such that the service could query its data stores for the user’s workout history, their high scores in a game, news items from the feeds they’re registered, and so forth. In such cases, a user id could be personally-identifiable information (PII), so the app must respect privacy concerns. This means that the app should either encrypt the username in the query string or use https:// URIs.

**Note**: Windows does not provide a means within the periodic update mechanism to authenticate the user with the service. That level of support can only be accomplished with push notifications (or in enterprise scenarios with the Enterprise Authentication capability declared in the manifest).

Services can be written with other technologies, of course. ASP.NET is a good choice because you can then employ the same [Notifications Extensions Library](http://msdn.microsoft.com/library/windows/apps/hh969156.aspx) (written in C#) that you can use in an app to easily generate well-structured XML payloads.

To walk through a quick example, consider the very basic WebMatrix service I created for the HelloTiles sample service in Chapter 13 of my book. This particular service just returns a fixed XML payload (with bindings for both square and wide tiles) and is structurally similar to the first ASP.NET example given in [Creating a great tile experience, part 2](http://blogs.msdn.com/b/windowsappdev/archive/2012/04/18/creating-a-great-tile-experience-part-2.aspx):

@{

//

// This is where any other code would be placed to acquire the dynamic content

// needed for the tile update. In this case we'll just return static XML to show

// the structure of the service itself.

//

var weekDay = DateTime.Now.DayOfWeek;

}

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

<tile>

<visual lang="en-US">

<binding template="TileSquarePeekImageAndText02" branding="none">

<image id="1" src="http://www.kraigbrockschmidt.com/images/Liam07.png"/>

<text id="1">Liam--</text>

<text id="2">Giddy on the day he learned to sit up!</text>

</binding>

<binding template="TileWideSmallImageAndText04" branding="none">

<image id="1" src="http://www.kraigbrockschmidt.com/images/Liam08.png"/>

<text id="1">This is Liam</text>

<text id="2">Exploring the great outdoors!</text>

</binding>

</visual>

</tile>

This particular service is deployed to <http://programmingwin8-js-ch13-hellotiles.azurewebsites.net/Default.cshtml> if you want to give it a try in scenario 4 of the Push and periodic notifications client-side sample. In doing so, after a few seconds you should see the following tile updates (wide on the left, and the two parts of the square peek tile on the right):

|  |  |
| --- | --- |
|  |  |

Let’s now write the same thing using the Notification Extensions Library. The first thing you’ll need to do is build a version of the library for your web site as follows:

1. Go to the [App tiles and badges sample](http://code.msdn.microsoft.com/windowsapps/App-tiles-and-badges-sample-5fc49148) and copy the Notifications Extensions folder from that project into a folder of your own. (You can also install the library from Visual Studio directly by right-clicking on a project, selecting Manage NuGet Packages… and searching for *NotificationsExtensions.WinRT*. However, this is meant to bring the library into an existing app project whereas we need to build a standalone DLL here.)
2. In Visual Studio Express for Windows, open the NotificationsExtensions.csproj file.
3. In Solution Explorer, right click the NotificationExtensions project, select Properties, and make the following changes:
   1. In the Application settings, change the Output type to *Class Library* (a .dll). This is necessary to use the library with an ASP.NET site.
   2. In the Build settings, change the Configuration to *All Configurations*, change the conditional compilation symbols to read *NETFX\_CORE; WINRT\_NOT\_PRESENT*, and make sure XML documentation file is checked near the bottom of the page. The *WINRT\_NOT\_PRESENT* flag enables the library to compile without WinRT.
4. Select a Debug or Release target, then right click the Notifications Extensions project and select Build.

This should generate a DLL and associated files in that project folder. Now we need to pull it into a web site project.

1. In Visual Studio Express for Web, right click your web site project and select Add > Add ASP.NET folder > Bin if your site doesn’t have a Bin folder already.
2. Right click that Bin folder and select Add Reference…. In that dialog, navigate to the bin\Debug or bin\Release folder of the Notifications Extensions project and select the DLL found there.

If you’re using Visual Studio Ultimate, you can add the Notification Extensions project to your web site solution, if desired, as the tool can handle both project types. Just be sure you don’t deploy the source code for that project to your web server!

Also note that if you build your site to run it locally in a browser (as we’ll see in the Debugging section below), you might get an error about adding a reference to *System.Runtime*. To correct this, open the web.config file and change the *compilation* element to read as follows:

<compilation debug="true" targetFramework="4.0">

<assemblies>

<add assembly="System.Runtime, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />

</assemblies>

</compilation>

With all this in place, here’s a page (called DefaultNE.aspx, without using a separate code-behind file) that produces the same output as the earlier hard-coded example:

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

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

<script runat="server">

public string GenerateTileXML()

{

// Construct the square template

NotificationsExtensions.TileContent.ITileSquarePeekImageAndText02 squareTile =

NotificationsExtensions.TileContent.TileContentFactory.CreateTileSquarePeekImageAndText02();

squareTile.Branding = NotificationsExtensions.TileContent.TileBranding.None;

squareTile.Image.Src = "http://www.kraigbrockschmidt.com/images/Liam07.png";

squareTile.TextHeading.Text = "Liam--";

squareTile.TextBodyWrap.Text = "Giddy on the day he learned to sit up!";

// Construct the wide template

NotificationsExtensions.TileContent.ITileWideSmallImageAndText04 wideTile =

NotificationsExtensions.TileContent.TileContentFactory.CreateTileWideSmallImageAndText04();

wideTile.Branding = NotificationsExtensions.TileContent.TileBranding.None;

wideTile.Image.Src = "http://www.kraigbrockschmidt.com/images/Liam08.png";

wideTile.TextHeading.Text = "This is Liam";

wideTile.TextBodyWrap.Text = "Exploring the great outdoors!";

// Attach the square template to the notification

wideTile.SquareContent = squareTile;

wideTile.Lang = "en-US";

// The wideTile object is an XMLDOM object, suitable for issuing tile updates

// directly. In this case we just want the XML text.

return wideTile.ToString();

}

</script>

<% = GenerateTileXML() %>

You can visit this service at <http://programmingwin8-js-ch13-hellotiles.azurewebsites.net/DefaultNE.aspx>, from which you’ll get essentially the same XML back as before, with very minor differences. Pasting the URI into scenario 4 of the Push and periodic notifications client-side sample will also generate the same tile updates as before.

# Debugging Services

Generating a successful tile or badge update assumes, of course, that the XML in the service’s response is properly formed: Windows will reject any XML that isn’t. This is a good reason to use the Notifications Extensions Library as it will greatly reduce the errors you might otherwise make in the process.

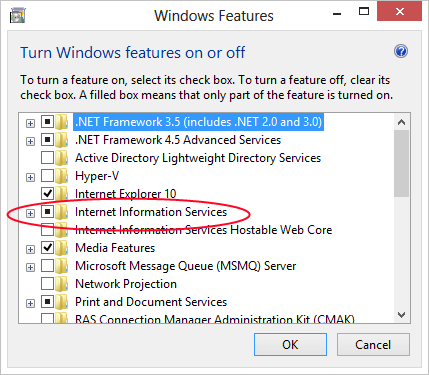
But what if your service isn’t working well at all? How you can diagnose and debug how it’s handling requests and generating its response?

In fact, when I first tried the ASP.NET service code above, the updates would not appear because there was a leading newline at the top of the XML response. This is specifically why the *<?xml ?>* header appears as the first line in the file rather than after the *<%@ Page %>* directive, and why there aren’t any extra line feeds.

Clearly, there are many reasons why you’ll want to be able to step through your service code and debug it line by line, especially if you’re querying a database and processing those query results.

The trick is to doing this is to utilize the localhost on your development machine, which allows you to run and debug your service locally at the same time you’re running client test code such as the SDK samples.

Having a localhost means you’re running a local web server like Internet Information Services or Apache. To turn on IIS in Windows (its built-in), go to Control Panel > Turn Windows features on or off. Check the Internet Information Services box at the top level, as shown below, to install the core features:



Once IIS is installed, the local site addressed by *http://localhost/* is found in the folder *c:\inetpub\wwwroot*. That’s where you drop something like the PHP page described in the last section so that you can use a URI like *http://localhost/dayofmonthservice.php* in the client samples.

To use PHP with IIS, you might need to install it through Microsoft’s [Web platform installer](http://www.microsoft.com/web/downloads/platform.aspx) or the server-side code won’t execute properly. After PHP installation, try entering a URI for a local PHP page in your browser. If you get an error message that says “Handler PHP53\_via\_FastCGI has a bad module” (yeah, that’s really helpful!), return to the Turn Windows features on or off dialog shown earlier, navigate to Internet Information Services > World Wide Web Services > Application Development Features, check the box for CGI, and press OK. Once the CGI engine is installed, your PHP page should work.

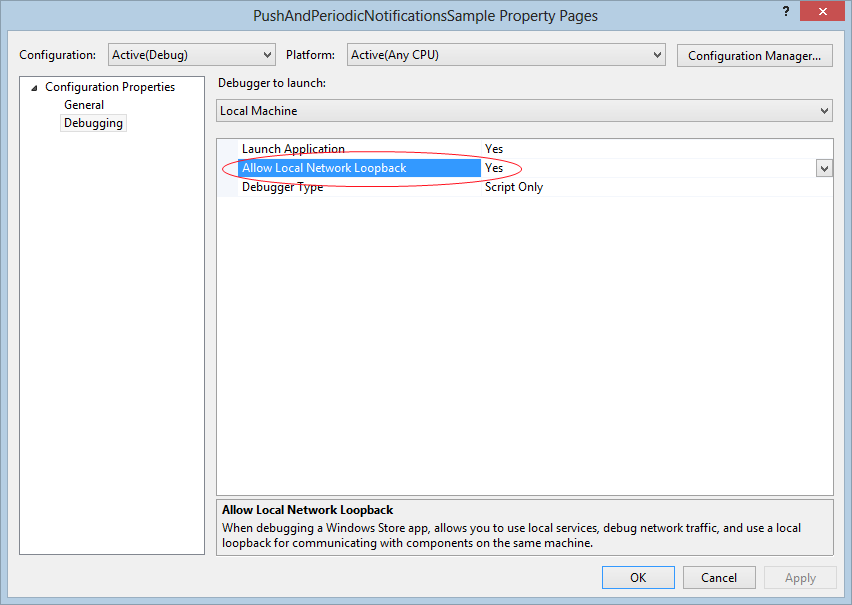
With your localhost in place, you can then debug services on your own machine using Visual Studio Express for Web or Visual Studio Ultimate. You can also use Visual Studio Express for Web alongside Visual Studio Express for Windows to debug client and server code together.

When you run a service or website in the Visual Studio (for Web) debugger, it will run in the browser at a URL like *http://localhost:<port>/* where <port> is randomly assigned for that project. For example, when I run the DefaultNE.aspx page from the previous section in Visual Studio Express for Web, that page opens in Internet Explorer with the URI of *http://localhost:52568/HelloTiles/DefaultNE.aspx.* If I’ve set a breakpoint in that page’s code, the debugger will immediately stop at that point.

Breakpoints are also hit if you use that same localhost URI in client code to initiate requests. So, for example, if I’m running the Push and periodic notifications client side sample in Visual Studio Express for Windows and paste the URI into scenario 4, Visual Studio Express for Web will stop my service in the debugger as soon as Windows makes a request. I can then step through that code (Windows is fortunately patient) and make sure the right response is being generated. If not, I can fix the code and restart the service on the localhost.

When you’re confident that your service is operating as intended, you can then upload it to your live web host (or staging environment) and do your final production testing.

Note that it’s not necessary to run client code in a debugger to use the localhost in this way. If you do, however, there’s an option in Visual Studio that must be turned on for localhost to work. It’s checked by default, but if you need to change it, you’ll find it in you project properties under Debugging > Allow local network loopback:



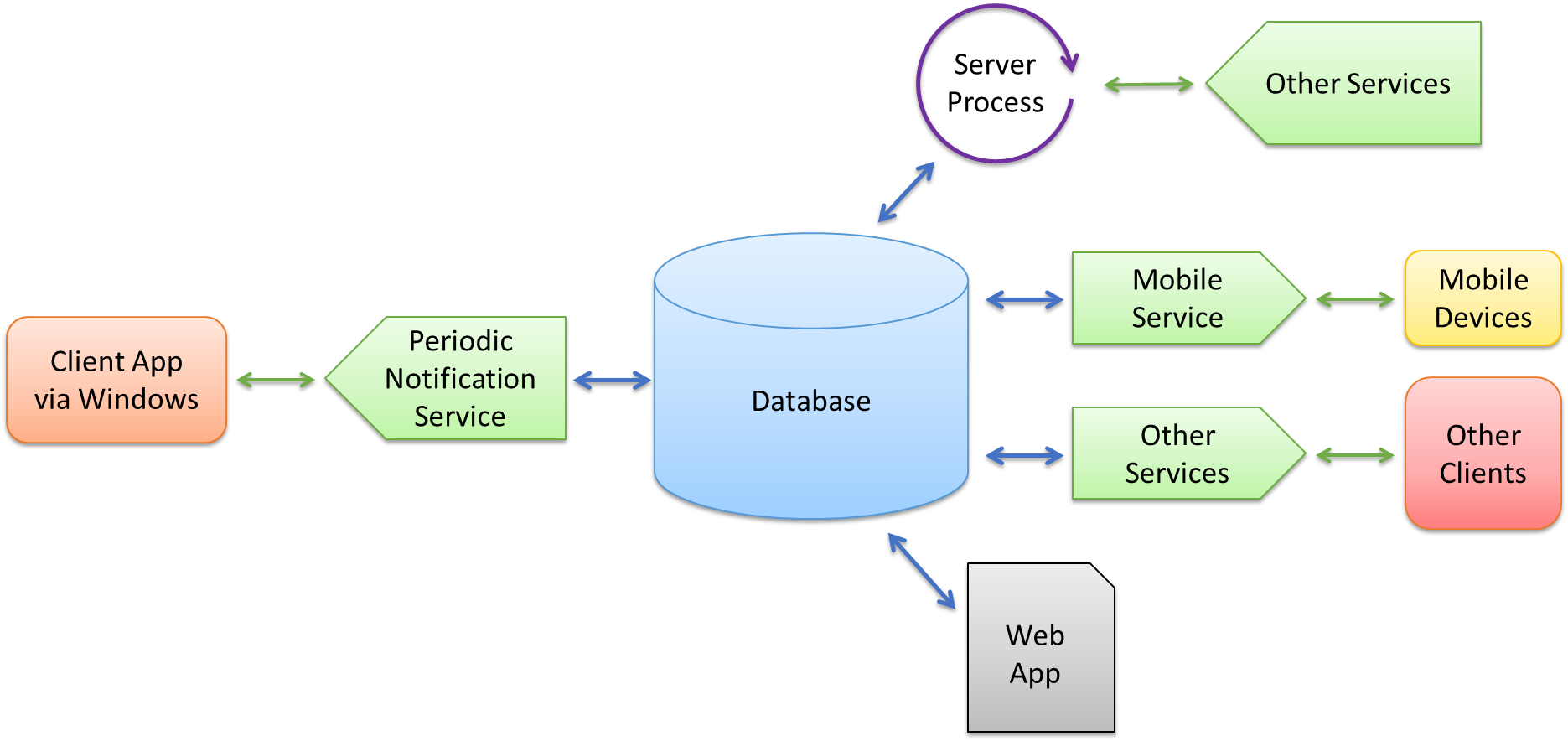
# Data from External Sources

In addition to querying its own database, it’s entirely possible for a periodic notification service to send requests to other services to obtain data for its response. The caveat, however, is that such requests are asynchronous in nature and are prone to a variety of failure conditions. As such, using them can greatly complicate the implementation of your service.

To simplify matters, we can take advantage of the fact that Windows will only make requests to your service at 30-minute or longer intervals, as enforced by the client API. This means you have lots of time during which other server-side processes can make external requests to monitor weather alerts, leaderboards, RSS feeds, and just about anything else for which there’s a web API. Those processes store results in your database, which are then ready for your periodic notification service to query (synchronously) when it receives its next request.

Indeed, any number of agents can update the same database. For example, users might be entering data through your website. They might be using mobile phone apps to track their activities, with results being automatically uploaded to a database. The database might also be getting updated by friends who are using the same app on their individual devices.

Such an arrangement is illustrated below, showing how the database serves as the central store for your back-end state, with the periodic notification service acting only as a simple consumer of that state.



# Using Windows Azure Mobile Services with Periodic Update Services

As you begin to understand and build out back-end services to support live tiles and other notifications, you really owe it to yourself to explore [Windows Azure Mobile Services](http://www.windowsazure.com/en-us/develop/mobile/tutorials/get-started/), which I’ll refer to as AMS for short. In addition to greatly simplifying push notifications, as we’ll see in Part 3 of this series, AMS can be used to support periodic updates services in several ways:

* In a mobile service you can create scheduled background jobs that can make requests to other services and store the results in your database. An example of this, to obtain tweets from Twitter, can be found on the [Schedule recurring jobs in Mobile Services](http://www.windowsazure.com/en-us/develop/mobile/tutorials/schedule-backend-tasks/) topic.
* When you create a SQL Server database in AMS (or elsewhere in Windows Azure), that database is accessible like any other web-hosted SQL Server database, so you can use it from web sites and other services, including those written in PHP.
* AMS makes it very easy to insert records into a database from a client app, using the [Mobile Services SDK](http://www.windowsazure.com/en-us/develop/mobile/developer-tools/).
* Separate from mobile services, Windows Azure can host server-side processes written in a variety of languages, including Node.js, Python, Java, PHP, and .NET.

In the future, also watch for a new feature called “service operations” In Azure Mobile Services, through which you will be able to create arbitrary http endpoints, including a periodic notification service.

For more information about Windows Azure, visit <http://www.windowsazure.com>. For introductory videos, also visit [Windows Azure Mobile Services Tutorials](http://channel9.msdn.com/Series/Windows-Azure-Mobile-Services) on Channel 9.

Having now explored how to create periodic notification services, we’re ready to take the next step into push notifications. Push notifications are necessary when you need to issue updates more frequently than periodic notifications allow, essentially on demand (hence the name, “push”). This is the topic that we’ll return to in Part 3 of this series, where we’ll see much more of Azure Mobile Services.

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)