Windows Phone 7 Push Notification Server Side Helper Library

A Windows Phone Recipe

# Document Purpose

This document introduces the Windows Phone Push Notification Server Side Helper, explains how to use it when a third-party web server application needs to send Push Notification (PN) messages to a Windows Phone application, and describes few common Push Notification patterns.

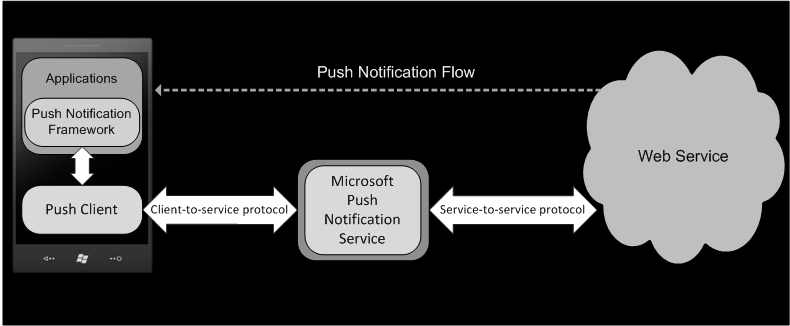
# Windows Phone Push Notification Overview

The Microsoft Push Notification Service (MPNS) in Windows Phone offers third-party developers a resilient, dedicated, and persistent channel to send information and updates to a mobile application from a web service.

In the past, a mobile application would need to poll its corresponding web service frequently to know if there were any pending notifications. While effective, each time the phone is polled, the device radio is turned on, drawing down the battery in the process. By using push notifications instead of polling, a web service can notify an application of important updates on an as-needed basis.

When a web service has information to send to an application, it sends a push notification to the Push Notification Service, which in turn routes the push notification to the application. Depending on the format of the push notification and the payload attached to it, the information is delivered in one of three ways: raw data is delivered to the application, the application's Tile is visually updated, or a toast notification is displayed. The application can then contact the web service using its own protocol, if needed.

The Push Notification Service sends a response code to your web service after a push notification is sent. However, the Push Notification Service does not provide an end-to-end confirmation that your push notification was delivered from your web service to your application.



For more information, see [Push Notification Service Response Codes for Windows Phone](http://msdn.microsoft.com/en-us/library/ff941100(v=VS.92).aspx). If you are not familiar with Windows Phone Push Notification please review [Push Notifications Overview for Windows Phone](http://msdn.microsoft.com/en-us/library/ff402558(v=VS.92).aspx) on MSDN, and read [Using Push Notification from Your Windows Phone Application](http://windowsteamblog.com/windows_phone/b/wpdev/archive/2010/05/06/using-push-notification-from-your-windows-phone-application.aspx)

# Push Notification Server Side Helper Library

* 1. The Push Notification Server Side Helper Library, a part of the Windows Phone 7 Push Recipe, provides an easy way for sending all three kinds of push notification messages currently supported by MPNS: **Tile**, **Toast** and **Raw**. These are described in more detail in [Windows Phone Push Notification](#_Windows_Phone_Push). Our main goal here is to extract any complexity for sending PN messages from your website to the Windows Phone client. This library helps developers to send push messages using only a few lines of code.

## PushNotificationMessage Base Class and Sub-Classes

* 1. In order to send a PN message to your Windows Phone application, you need to post that message to MPNS. Therefore, it makes sense to abstract the sending functionality and create a **PushNotificationMessage** base class. This base class handles the majority of the logic and heavy lifting of sending the message to MPNS.
  2. The **PushNotificationMessage** base class includes properties such as **Unique ID**, **Class ID**, **Priority** and **Payload**. These properties are required for each PN message, and differ for each sub-class.
  3. The main public functions that the **PushNotificationMessage** base class exposes are two send methods: **Send,** for sending the PN message synchronously – blocking the current thread, and **SendAsync**, for sending the PN message asynchronously using the .NET thread pool. Both methods have a callback for returning the MPNS response codes, or can throw an exception if something goes wrong.
  4. A separate sub-class represents each the different PN messages types, as illustrated in the following class diagram.
  5. 
  6. *Push Messages Class Diagram*
  7. The three supported PN message sub-classes are:
* **RawPushNotificationMessage** – This sub-class represents sending raw bytes to the Windows Phone application while it is running
* **TilePushNotificationMessage** – This sub-class represents sending tile updates to the Windows Phone when your application tile is pinned to the phone’s Start screen
* **ToastPushNotificationMessage** – This sub-class represents sending toast “alert” messages to the Windows Phone
  1. The difference between the three types of messages is the way each one encapsulates the push message format and exposes it with regular .NET properties.

## Examples of Sending Push Notification Messages

Below are examples that use the different Push Notification classes to send PN messages.

### Sending a Tile Message

* 1. The code snippet below demonstrates how to use the Push Notification Library to send a Tile notification message both synchronously and asynchronously.
  2. C#
  3. // Prepare a tile push notification message.
  4. var tile = new TilePushNotificationMessage
  5. {
  6. BackgroundImageUri = tileImageUri, // Remote or phone-local tile image uri.
  7. Count = tileCount, // Counter between 1 to 99 should be displayed on the tile.
  8. Title = “Tile Title” // Title to be displayed on the tile.
  9. };
  10. // Send the message synchronously.
  11. try
  12. {
  13. var sendResult = tile.Send(phoneChannelUri);
  14. // Check the send result.
  15. }
  16. catch (Exception ex)  
      {
  17. // Log the error.  
      }
  18. // Send the message asynchronously.
  19. tile.SendAsync(
  20. phoneChannelUri,
  21. result => {/\* Check the send result \*/},
  22. exception => {/\* Log the error \*/});

In this code you can see how easy it is to send a tile update to your Windows Phone application. Simply create a new TilePushNotificationMessage providing the relevant properties. Next call the Send or SendAsync method to send the messages.

### Sending a Toast Message

* 1. The code snippet below demonstrates how to use the Push Notification Library to send a Toast notification message both synchronously and asynchronously.
  2. C#
  3. // Prepare a toast push notification message.
  4. var toast = new ToastPushNotificationMessage
  5. {
  6. Title = “Title”, // Title to be displayed as the toast header.
  7. Subtitle = “Sub Title” // Message to be displayed next to the toast header.
  8. };
  9. // Send the message synchronously.
  10. try
  11. {
  12. var sendResult = toast.Send(phoneChannelUri);
  13. // Check the send result.
  14. }
  15. catch (Exception ex)  
      {
  16. // Log the error.  
      }
  17. // Send the message asynchronously.
  18. toast.Send(
  19. phoneChannelUri,
  20. result => { /\* Check the send result \*/ },
  21. exception => { /\* Log the error \*/ });

### Sending a Raw Message

* 1. The code snippet below demonstrates how to use the Push Notification Library to send a Raw notification message both synchronously and asynchronously.
  2. C#
  3. // Prepare a raw push notification message.
  4. byte[] rawData = {};
  5. var raw = new RawPushNotificationMessage
  6. {
  7. RawData = rawData, // Raw data to be sent with the message.
  8. };
  9. // Send the message synchronously.
  10. try
  11. {
  12. var sendResult = raw.Send(phoneChannelUri);
  13. // Check the send result.
  14. }
  15. catch (Exception ex)  
      {
  16. // Log the error.  
      }
  17. // Send the message asynchronously.
  18. raw.Send(
  19. phoneChannelUri,
  20. result => { /\* Check the send result \*/ },
  21. exception => { /\* Log the error \*/ });

## Understanding the Microsoft Push Notification Server HTTP Header Response

After your web service sends a push notification to the Microsoft Push Notification Service, your web service will receive one of many response codes with possible custom headers. All responses are comprised of three statuses that when combined give you a clear indication of what happened with your message:

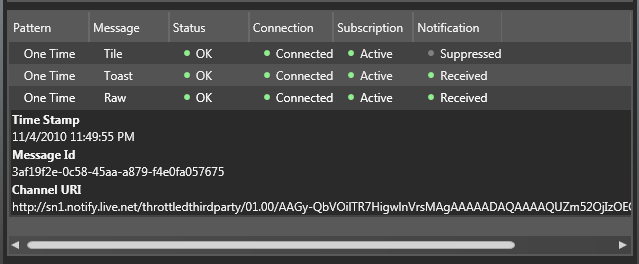
* + **Notification Status** - The status of the notification received by the Microsoft Push Notification Service
  + **Device Connection Status** – The connection status of the device
  + **Subscription Status** – The subscription status of the PN channel that is being used

We are not going to cover all the possible different status values and their combination. This information is found in the [Push Notification Service Response Codes for Windows Phone](http://msdn.microsoft.com/en-us/library/ff941100(v=VS.92).aspx) topic on MSDN. However there are a couple of status combinations worth mentioning. For example:

* + **Notification Status is suppressed, Device Connection Status is connected, and the Subscription is active** – The push notification was received by MPNS and dropped by the MPNS. The Suppressed status can occur if the notification channel was configured to suppress push notifications for a particular push notification class, for example the application tile was not pined to the Start menu.
  + **Notification Status is dropped, Device Connection Status is connected, and the Subscription has expired** – The subscription is invalid and is not present on the Push Notification Service. The web service should stop sending new notifications to this subscription, and drop the subscription state for its corresponding application session.

Again, for the complete list, refer to the [Push Notification Service Response Codes for Windows Phone](http://msdn.microsoft.com/en-us/library/ff941100(v=VS.92).aspx) topic on MSDN.

The WPF application acts as your web service, records the return values, and displays them together with some information about the message that was sent, as shown by this image:



# Windows Phone Test Client Application

* 1. The Windows Phone Push Notification Test client is a basic (minimal UI) Silverlight application that we use to demonstrate how push messages are sent to the phone.

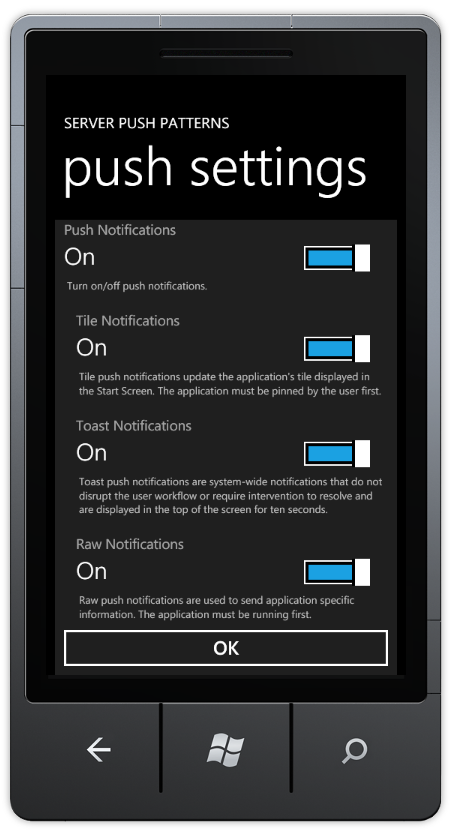
## Registering for Push Notification

In order to register to receive PNs, the Windows Phone application needs to initiate a request to the MPNS, which in response returns a URI (uniform resource identifier). This URI represents the application end point on the MPNS to which your web server will post the PN messages.

In our sample Windows Phone application, the PN functionality is encapsulated in the **PushContext** class. We highly recommend that you open (or create) the PN channel and send the push channel URI to your web service each time you initiate the application. Sending the URI every time your application starts, ensures that your web service has the latest URI for your phone. The **Connect** method does just that, as it tries to open an existing channel. If such channel doesn’t exist, it tries to create a new channel. Once a channel is established, the application registers the PN channel URI with your web service – in our case, the Windows-based WPF application that acts as the web application (hosting the web service). Once the URI is registered, your web service can use that URI to send PN messages to the Windows Phone application. While this document doesn’t cover all aspects of Windows Phone client programming, you can read about it at [Using Push Notification from Your Windows Phone Application](http://windowsteamblog.com/windows_phone/b/wpdev/archive/2010/05/06/using-push-notification-from-your-windows-phone-application.aspx).

## The Push Settings Page

Each Windows Phone application that uses Push Notifications must have a Settings page to allow the user to turn the Push Notification functionality on and off in the application. According to Windows Phone Marketplace guidance, the application needs to ask the user for permission to use PN and offer a way to turn off PN. The sample application has a Settings page (found in the PushSettingControl.XAML file) that enables the user to turn PN on and off, as well as control settings for each PN message type. While it is not required to have such granular control over each message type, for this sample application we want to use this functionality to illustrate what happens on the server side when you send PN messages and when the phone blocks some PN message types. The following image shows the PN setting page in the sample application.



The above screen is implemented as a user control, and we bind the control **DataContext** to the **PushContext** we mentioned above. The **ToggleSwitch** buttons are bound (TwoWay binding) to the respective properties in the **PushContext**, so the **PushContext** is always up to date.

# Push Notification Server-Side Common Patterns

To illustrate the use of the Push Notification Server Side Helper, we’ve create a WPF application that acts like a web service, mimicking a third-party server. To enable this, the WPF application has a WCF service that lets the Windows Phone application register and communicate with MPNS. The WPF has tabs, each of which represents a specific Push Pattern that we wish to illustrate.

Push Patterns are a generalization of a common use of PN in an end–to-end solution including specific Windows Phone and server-side logic. The current application has five patterns (One-time Push, Push Counter Resets upon Logon, Ask to Pin Application Tile, Create Custom Server-side Image, and Scheduled Tile Updates), which are described in the following sections.

**Please note:** The server-side patterns are provided as a **samples** and only to articulate the specific pattern usage; they are NOT best practices for server-side or web programming. The goal is to explain the patterns and server-side requirements. To keep the examples simple, we **DO NOT** use server-side authentication, a database, or any complex setups that would be included in actual scenarios.

## One-Time Push

### Description

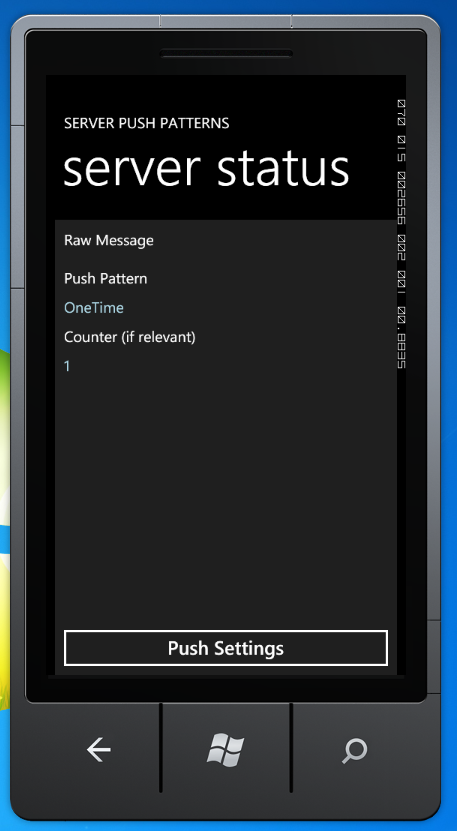
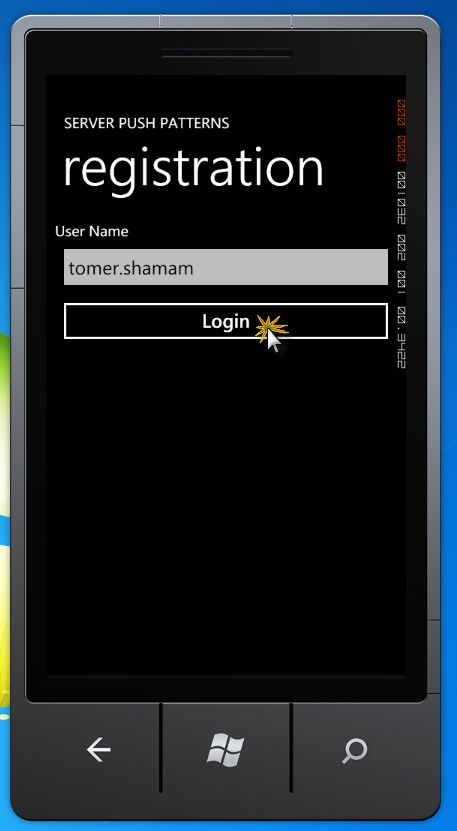
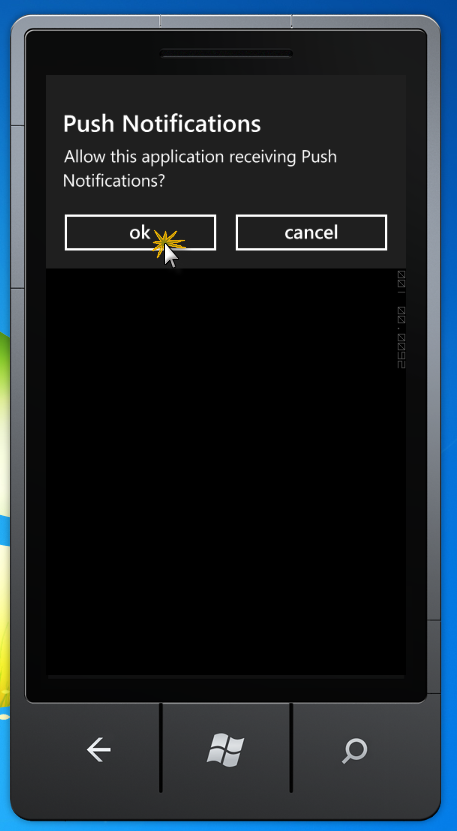
Pushing a one-time message to a registered client is the simplest PN pattern. One-time push notifications can be any of the three types, Tile, Toast, or Raw, or any combination of them. Our goal is to show simple usage of the Push Notification Server Side Helper Library as well as the different returned values that are sent back from MPNS.

### Implementation

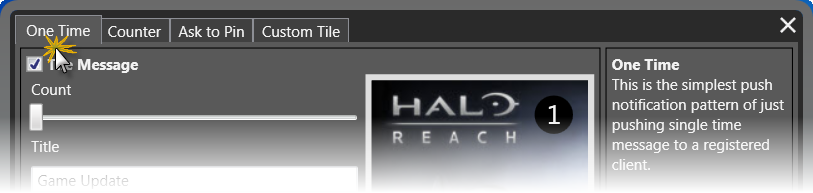
1. Select the desired message type checkbox in the **Send** method found in the **OneTimePushPatternViewModel** class to define the PN message type to send.
2. Use the Push Notification Server Side Helper Library to asynchronously send the message to the Windows Phone client application.
3. Get the return values from MPNS, and log the PN message.

### Demonstration

1. Run both the WindowsPhone.Recipes.Push.Server and WindowsPhone.Recipes.Push.Client projects (Note: You need to run the WPF application - WindowsPhone.Recipes.Push.Server as admin).
2. On the phone emulator, accept any existing push messages and log on with any user name.

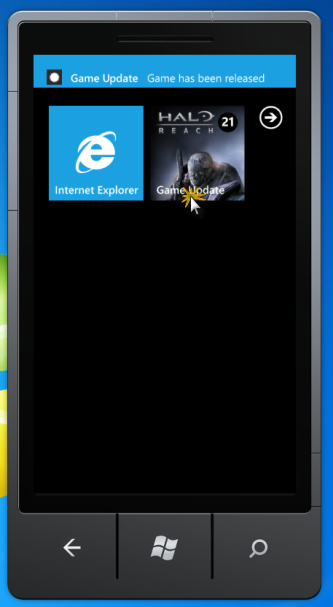
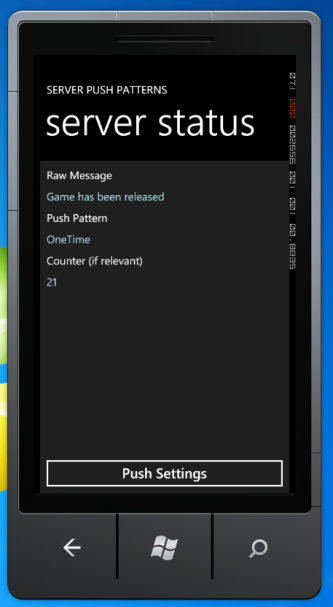
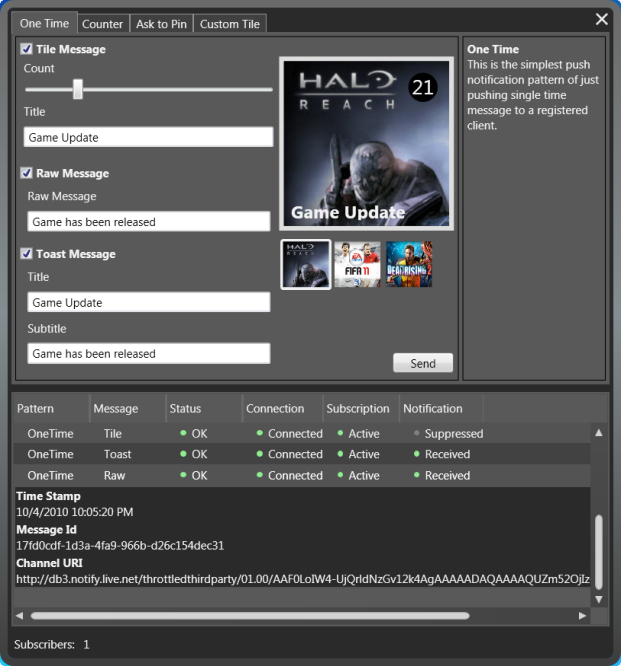


1. On the server UI, make sure that the ‘One Time’ tab is selected.



1. Select one or more types of messages to send, fill in the relevant data per PN message type, such as title, Raw message, Toast, etc., and click **Send**.

The phone client should display the raw message if the application is running and Raw was selected, the toast message if the application is not running, and the tile if the application tile was pinned to the phone Start screen.



### Code

[Implemented as part of the *OneTimePushPatternViewModel.cs* file]

The PN pattern starts when sending a push notification to one or more clients is relevant. In our case, the code below picks the type of messages to send based on current server logic and then asynchronously sends each message to the relevant client.

* 1. C#
  2. protected override void OnSend()
  3. {
  4. var messages = new List<PushNotificationMessage>();
  5. if (IsTileEnabled)
  6. {
  7. // Prepare a tile push notification message.
  8. messages.Add(new TilePushNotificationMessage
  9. {
  10. BackgroundImageUri = BackgroundImageUri,
  11. Count = Count,
  12. Title = Title
  13. });
  14. }
  15. if (IsToastEnabled)
  16. {
  17. // Prepare a toast push notification message.
  18. messages.Add(new ToastPushNotificationMessage
  19. {
  20. Title = ToastTitle,
  21. SubTitle = ToastSubTitle
  22. });
  23. }
  24. if (IsRawEnabled)
  25. {
  26. // Prepare a raw push notification message.
  27. messages.Add(new RawPushNotificationMessage
  28. {
  29. RawData = Encoding.ASCII.GetBytes(RawMessage)
  30. });
  31. }
  32. foreach (var subscriber in PushService.Subscribers)
  33. {
  34. messages.ForEach(m => m.SendAsync(subscriber.ChannelUri, Log, Log));
  35. }
  36. }

## Push Counter Resets when User Logs on

### Description

Tiles on the Windows Phone Start screen can present information such as title, short message, and a number. Some phone applications such as email and SMS messages use the number on the tile to indicate the number of unread messages. The Push Counter Pattern mimics that behavior by sending a tile PN message with a counter value. Each time a push notification message is sent, the counter value increases by one. The next time the user logs on to the server side application, the counter resets.

### Implementation

1. Maintain a list of tile counters, one per subscription.

* Each counter should be kept on the server side until the client unregisters or the application is activated.

1. When a Windows Phone application connects to the server, a user is registered.

* Upon registration, delete the relevant subscription counter.

1. To save resources, before sending a tile message, send a Raw PN message to the client.

* This tests if the application is currently running.

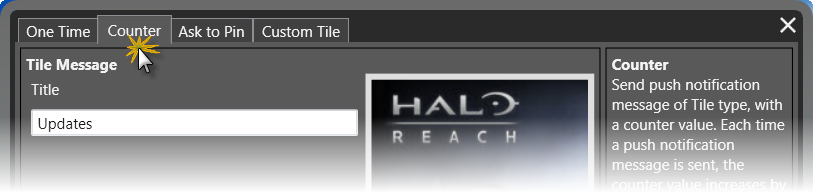
1. If the send completes successfully, check the return values from the MPNS, specifically the subscription, notification, and device connection status as follows:

* If the device connection is temp-disconnected, do nothing.
* If device is connected, subscription is active, and the notification is received, either create a new counter for the current subscription starting at one, or increase the current subscription counter by one.
* Update the phone tile counter by sending the current subscription tile counter.

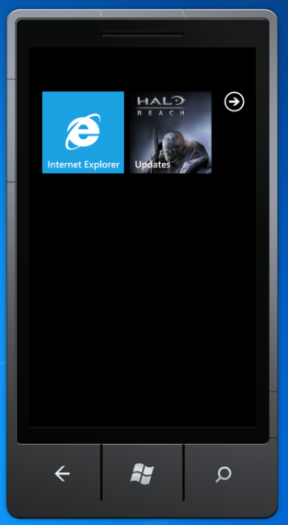
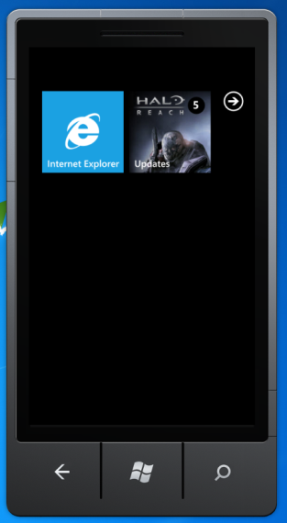
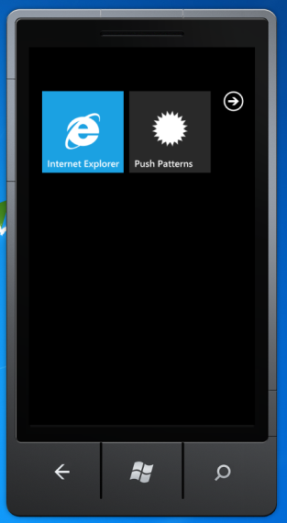
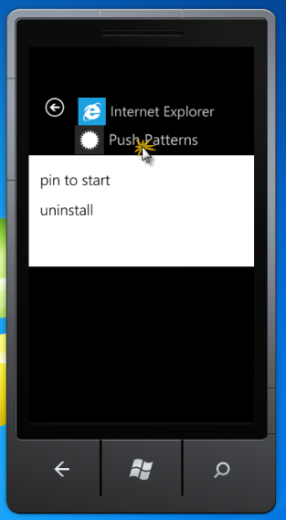
1. Log message send error or message send success.

### Demonstration

1. Run both the WindowsPhone.Recipes.Push.Server and WindowsPhone.Recipes.Push.Client projects. (Note: You need to run the WPF application - WindowsPhone.Recipes.Push.Server as admin.)
2. On the phone emulator, accept any existing push messages and log on with any user name.
3. On the server UI, make sure that the ‘Counter’ tab is selected.



1. In the phone emulator, pin the application to the start menu.
2. In the server UI, click **Send** several times; you should notice that the application tile counter on the phone increases by one on each click.
3. Click the application’s tile to run the phone application. This will reset the tile counter.
4. Exit the phone application; notice that the tile counter resets.



### Code

[Implemented as part of the *CounterPushPatternViewModel.cs* file]

The pattern starts when the server prepares to send a tile notification message to one or more clients. In our case, the OnSend method is called by clicking **Send**. Here the application creates a raw message and sends it to all subscribers. Upon send complete, it calls OnRawSent with the relevant subscription.

* 1. C#
  2. protected override void OnSend()
  3. {
  4. // Notify phone for having waiting messages.
  5. var rawMsg = new RawPushNotificationMessage
  6. {
  7. RawData = Encoding.ASCII.GetBytes(RawMessage)
  8. };
  9. foreach (var subscriber in PushService.Subscribers)
  10. {
  11. rawMsg.SendAsync(
  12. subscriber.ChannelUri,
  13. result =>
  14. {
  15. Log(result);
  16. OnRawSent(subscriber.UserName, result);
  17. },
  18. Log);
  19. }
  20. }

When a Raw message is sent, the application gets the callback **OnRawSent**. Using the returned value from MPNS, it checks whether the device is connected. If the phone is not connected, sending a tile message is meaningless. If the device is connected, it checks the three flags described above and sends a tile message with the counter increased by one.

* 1. C#
  2. private void OnRawSent(string userName, MessageSendResult result)
  3. {
  4. // In case that the device is disconnected, no need to send a tile message.
  5. if (result.DeviceConnectionStatus == DeviceConnectionStatus.TempDisconnected)
  6. {
  7. return;
  8. }
  9. // Checking these three flags we can know what's the state of both the device and application.
  10. bool isApplicationRunning =
  11. result.SubscriptionStatus == SubscriptionStatus.Active &&
  12. result.NotificationStatus == NotificationStatus.Received &&
  13. result.DeviceConnectionStatus == DeviceConnectionStatus.Connected;
  14. // In case that the application is not running, send a tile update with counter increase.
  15. if (!isApplicationRunning)
  16. {
  17. var tileMsg = new TilePushNotificationMessage
  18. {
  19. Count = IncreaseCounter(userName),
  20. BackgroundImageUri = BackgroundImageUri,
  21. Title = Title
  22. };
  23. tileMsg.SendAsync(result.ChannelUri, Log, Log);
  24. }
  25. }

Intercepting the subscription event, the application creates a tile message for resetting the tile counter:

* 1. C#
  2. protected override void OnSubscribed(SubscriptionEventArgs e)
  3. {
  4. // Create a tile message to reset tile count.
  5. var tileMsg = new TilePushNotificationMessage
  6. {
  7. Count = 0,
  8. BackgroundImageUri = BackgroundImageUri,
  9. Title = Title
  10. };
  11. tileMsg.SendAsync(e.Subscription.ChannelUri, Log, Log);
  12. ResetCounter(e.Subscription.UserName);
  13. }

## Ask to Pin Application Tile

### Description

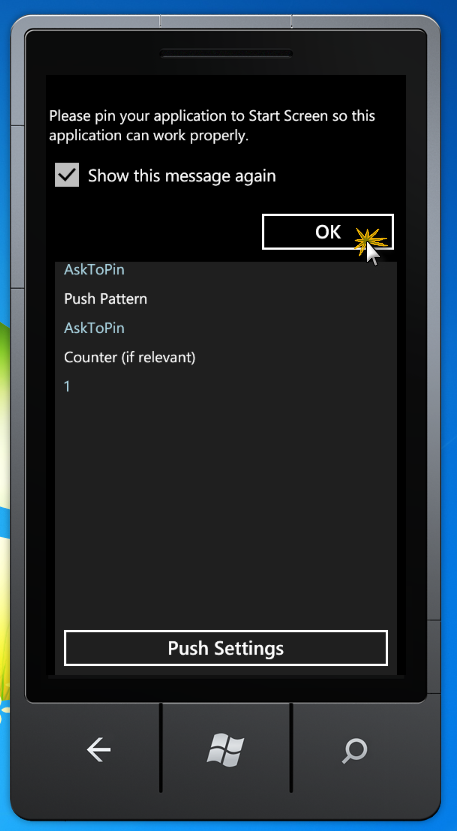
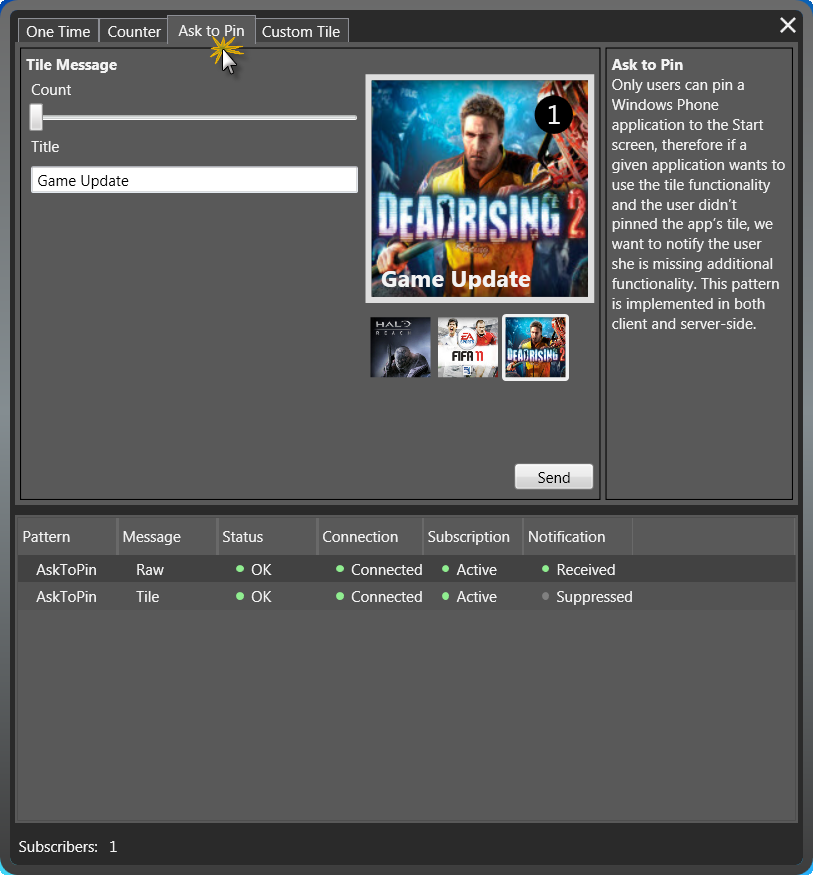
If a given application sends a tile PN message and that application’s tile is not pinned to the phone Start screen, the PN message will not be delivered to the phone, and the third-party server side will be notified that the application tile is not pinned. Since in Windows Phone, only end users can pin an application’s tile to the phone Start Screen, applications need a way to “ask” the user to pin the application’s tile. This pattern illustrates how an application can do that by sending a specific message to the phone. The pattern implementation includes both server-side and Windows Phone client-side code.

### Implementation

1. The server side maintains a list of the last tile message data sent per registered user.
2. Upon client registration the server side application knows that the Windows Phone client application is running.
3. The server side sends a tile message with the most updated tile data.
4. Based on the response to this message from MPNS, the application knows if the Windows Phone application’s tile is pinned to the Start screen or not:
   1. If the device is connected, the subscription is active, and the notification is received, we know that the application’s tile is pinned to the Start screen.
   2. If not, the server side sends a custom raw PN message to the Windows Phone client application asking it to display the “Ask to Pin” message to the user.
5. Log message send error or message send success.

### Demonstration

1. Run only the WindowsPhone.Recipes.Push.Server project and make sure that the ‘Ask to Pin’ tab is selected.
2. Run the WindowsPhone.Recipes.Push.Client (Windows Phone application), and make sure that the application’s tile is not pinned to the Start Menu.
3. Accept existing push messages and log on with any user name.
4. A message box should pop up asking to pin the application. Click **OK**.
5. On the server UI, click **Send** to update the tile.
6. A message box should pop up asking to pin the application. Click **OK**.



1. Close the application, pin it to the Start Page, and run the application again. The previous ‘Ask to Pin’ message should be gone.
2. On the server UI, click **Send** to update the tile. The previous ‘Ask to Pin’ message should be gone and the tile number should be updated.

Note –This pattern can be adjusted to make the Windows Phone client application display the “Ask to Pin” message upon logon.

### Code

[Implemented as part of the *AskToPinPushPatternViewModel.cs* file]

The pattern starts when a client first registers to the server. In order to check if the application is pinned or not, the OnSubscribed method below sends a tile notification message to the relevant subscriber (based on the Windows Phone client application URI and username). The data used with the message is new for first time subscriptions or from previously stored data for re-subscriptions. Using stored data prevents updating the tile for no reason.

* 1. C#
  2. protected override void OnSubscribed(SubscriptionEventArgs args)
  3. {
  4. // Asynchronously try to send Tile message to the relevant subscriber
  5. // with data already sent before so the tile won't change.
  6. var tileMsg = GetOrCreateMessage(args.Subscription.UserName, false);
  7. tileMsg.SendAsync(
  8. args.Subscription.ChannelUri,
  9. result =>
  10. {
  11. Log(result);
  12. OnMessageSent(args.Subscription.UserName, result);
  13. },
  14. Log);
  15. }

The PN message is sent asynchronously, and the OnMessageSent callback is called when the tile PN message is sent. At that point, the server side can check if the application is pinned to the Start screen by checking the response returned by MPNS. If the tile is pinned, do nothing. However, if the application’s tile is not pinned, the Windows Phone client application will “ask” the user to pin the application’s tile by sending a raw message with custom data, in our case the “Ask to Pin” message. This raw message is received by the phone client and displays a message to the user.

* 1. C#
  2. /// <summary>
  3. /// Once tile update sent, check if handled by the phone.
  4. /// In case that the application is not pinned, ask to pin.
  5. /// </summary>
  6. private void OnMessageSent(string userName, MessageSendResult result)
  7. {
  8. if (!CheckIfPinned(result))
  9. {
  10. AskUserToPin(result.ChannelUri);
  11. }
  12. }

The same logic repeats in cases where the server wants to send a tile update to one or more clients. This time the message is newly created.

* 1. C#
  2. /// <summary>
  3. /// Send a tile notification to all relevant subscribers.
  4. /// </summary>
  5. protected override void OnSend()
  6. {
  7. // Asynchronously try to send this message to all relevant subscribers.
  8. foreach (var subscriber in PushService.Subscribers)
  9. {
  10. // Create a tile message to send an update.
  11. var tileMsg = GetOrCreateMessage(subscriber.UserName);
  12. tileMsg.SendAsync(
  13. subscriber.ChannelUri,
  14. result =>
  15. {
  16. Log(result);
  17. OnMessageSent(subscriber.UserName, result);
  18. },
  19. Log);
  20. }
  21. }

The CheckIfPinned method below checks if the application is pinned to the Start screen by observing the three previously described flags.

* 1. C#
  2. private bool CheckIfPinned(MessageSendResult result)
  3. {
  4. // We known if the application is pinned by checking the following send result flags:
  5. return result.DeviceConnectionStatus == DeviceConnectionStatus.Connected &&
  6. result.SubscriptionStatus == SubscriptionStatus.Active &&
  7. result.NotificationStatus == NotificationStatus.Received;
  8. }

The AskUserToPin method below sends a raw message with the custom “AskToPin” data, which is intercepted by the phone.

* 1. C#
  2. /// <summary>
  3. /// Just in case that the application is running, send a raw message, asking
  4. /// the user to pin the application. This raw message has to be handled in client side.
  5. /// </summary>
  6. private void AskUserToPin(Uri uri)
  7. {
  8. new RawPushNotificationMessage
  9. {
  10. RawData = Encoding.ASCII.GetBytes(RawMessage)
  11. }.SendAsync(uri);
  12. }

## Create a Fully Customized Image on the Server-Side

### Description

The application’s tile on the Start screen is dynamic. It can be any image that you want, and the source of the image doesn’t have to be the phone’s local store; it can also be any valid URI (of any remote server on the Internet). This pattern shows how to send a customized tile image to the application. Basically this is an exercise in image manipulation on the server side, which is very easy using WPF (which is not a true server-side technology). The image is generated upon request.

#### Limitations

Please note the following limitations when the tile URI points to a remote server image:

* The URI must be accessible to the phone
* The image size must be less than 80KB
* The download time cannot exceed 60 seconds

### Implementation

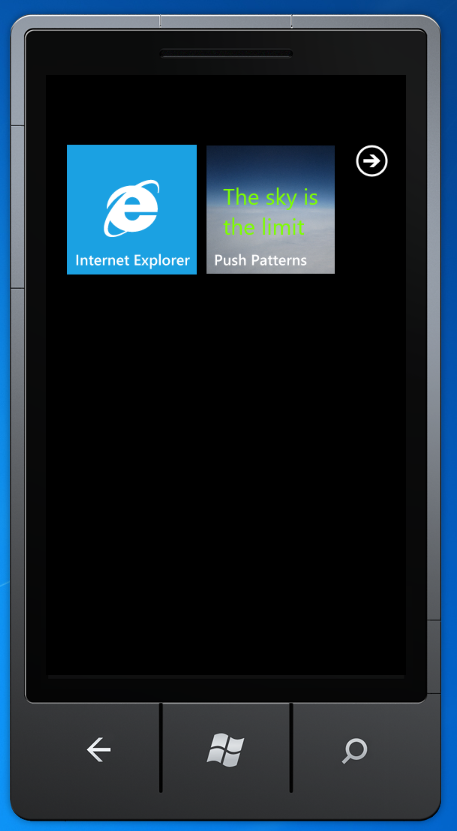
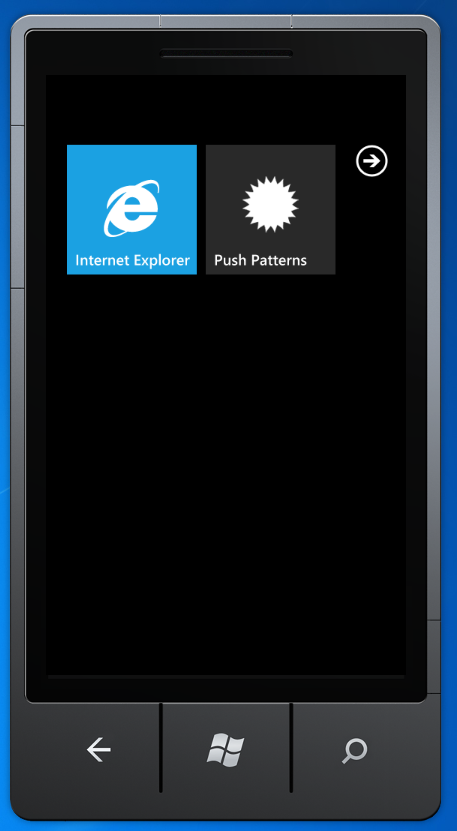
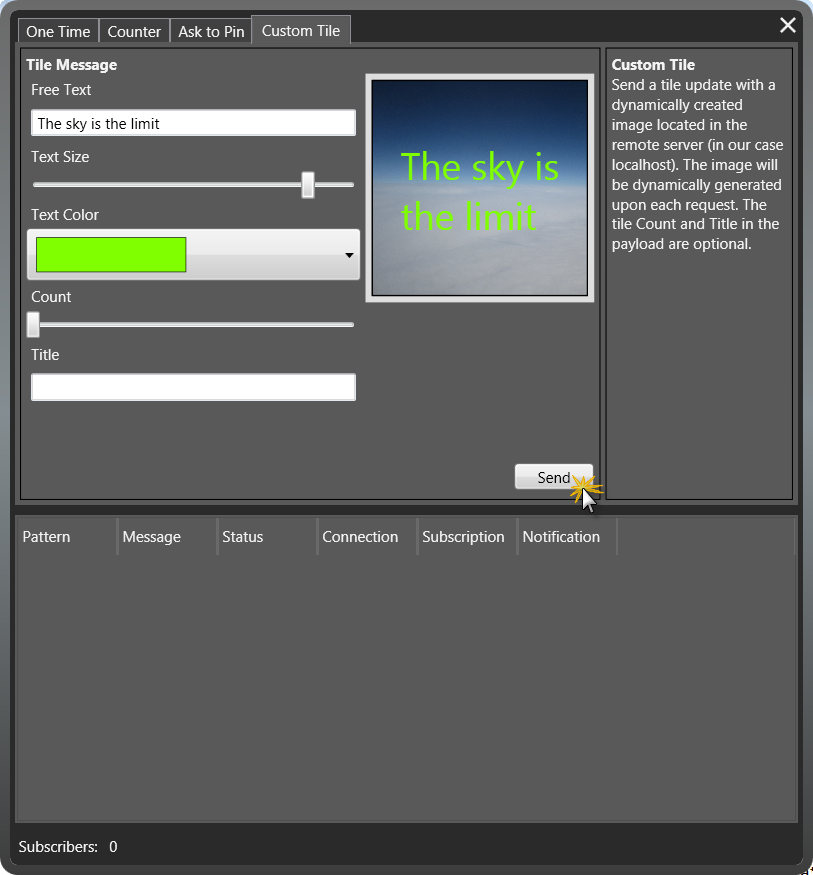
1. Create a tile notification message initialized with a remote URL to the desired image.
2. Asynchronously send the message to the relevant client.

* Note: The phone client tile binding should add the remote address to the allowed addresses/domains collection.

1. Log message send error or message send success.

### Demonstration

1. Run both the WindowsPhone.Recipes.Push.Server and WindowsPhone.Recipes.Push.Client projects.
2. On the phone emulator, accept existing push messages and log on with any user name.
3. Close the phone application and pin the application’s tile to the Start Page.
4. On the server UI, make sure that the ‘Custom Tile’ tab is selected.
5. On the server UI, click the image frame and pick your tile background image from your local machine.
6. Enter the custom text to be displayed on the tile and choose the font size and color. Tile counter and title are optional.
7. Click **Send**. You should notice the tile background update with the custom details.



### Code

* 1. [Implemented as part of the *CustomTileImagePushPatternViewModel.cs* file]

The pattern begins when the server is instructed to generate and update a phone application tile, in our case, by clicking **Send**. This activates the OnSend method. This method creates a tile message with the following URI: <http://localhost:8000/ImageService/GetTileImage?uri=channel_uri> instead of a URL to an image located on a web server.

Since the tile image URI must point to a remote address, such as an image resource on a web server, and since we are not using a web server and the deployment of such a configuration is complicated, for this example we’re using a little trick. We provide a URL to a REST service exposed by our WCF ImageService class. This REST service is called GetTileImage and it returns a Stream object to the relevant image.

* 1. C#
  2. protected override void OnSend()
  3. {
  4. // Starts by sending a tile notification to all relevant subscribers.
  5. // This tile notification updates the tile with custom image.
  6. var tileMsg = new TilePushNotificationMessage
  7. {
  8. Count = Count,
  9. Title = Title
  10. };
  11. foreach (var subscriber in PushService.Subscribers)
  12. {
  13. // Set the tile background image uri with the address of the ImageService.GetTileImage,
  14. // REST service, using current subscriber channel uri as a parameter to besent to the service.
  15. tileMsg.BackgroundImageUri = new Uri(string.Format(GetTileImageService, subscriber));
  16. tileMsg.SendAsync(subscriber.ChannelUri, Log, Log);
  17. }
  18. }

After sending a tile message with the above URI, the phone returns back to the address that activates the REST service: ImageService.GetTileImage. This method raises an event requesting the rendered image stream that was rendered in our demo by WPF. In a real scenario you may use an image processing library for creating a custom image efficiently and without the need for UI technology.

* 1. C#
  2. public Stream GetTileImage(string uri)
  3. {
  4. if (ImageRequest != null)
  5. {
  6. var args = new ImageRequestEventArgs();
  7. ImageRequest(this, args);
  8. // Seek the stream back to the beginning just in case.
  9. args.ImageStream.Seek(0, SeekOrigin.Begin);
  10. return args.ImageStream;
  11. }
  12. return Stream.Null;
  13. }

## Scheduled Tile Update

### Description

While the server side can initiate Windows Phone application tile updates by sending a tile PN message, the phone can schedule a periodic tile update using the **ShellTileSchedule** class. This enables the Windows Phone client application to register for periodic tile updates from a specific URI. Per the schedule, the phone will automatically initiate a request to the registered URI. If the URI returns an image, the tile on the phone will be updated.

### Implementation

From the server side there is very little that must be performed, as the work is done by Windows Phone. In our example, the **Create Fully Customized Tile** already provides a URI that allows Windows Phone to download images; therefore we can use that and the phone will do all the work.

### Demonstration

To test the Tile Schedule:

1. Run both the WindowsPhone.Recipes.Push.Server and WindowsPhone.Recipes.Push.Client projects.
2. On the phone emulator, accept existing push messages and log on with any user name.
3. In the server side application, go to the Tile Schedule tab.
4. On the phone emulator, select the image you wish to schedule from the list.
5. Click **Test Now** to test the server.

* This will result in the server side sending that image as tile PN message (testing whether the URI is working)

1. On the emulator, select a different image and click **Schedule.**

* Note that in the current Windows Phone version, the most frequently updates can be scheduled is hourly. Therefore testing this can take some time.

1. Now you need to wait for the internal phone scheduler to start the download process of the image

# Summary

* 1. The Windows Phone 7 Push Notification Server Side Helper Library is a simple class to use that extracts all the complexity of sending Push Notification messages. This library is ideal for server side programming as it built for scale.