While working with Azure Service Bus, either with the queue or with the topic/subscription, you need to explicitly write code to pull messages from the queue or from the subscription. A typical pattern looks like the following:

**Code Snippet**

* 1. // Create a SB queue
  2. QueueDescription queue = CreateQueue();
  3. // Create a factory and client to talk to this queue
  4. MessagingFactory factory = MessagingFactory.Create(ServiceUri, TokenProvider);
  5. QueueClient client = factory.CreateQueueClient(queue.Path);
  6. // Send a message to the queue
  7. string requestMessage = "Hello World!";
  8. client.Send(new BrokeredMessage(requestMessage));
  9. // Receive the message from the queue
  10. BrokeredMessage responseQueueMessage = client.Receive();
  11. string responseMessage = responseQueueMessage.GetBody<string>();

Note how you have to write code in line 13,14 to receive the messages from the queue. Programing with topic/subscription follow similar pattern in how you need to pull messages from the subscription client.

Now what if the queue is constantly receiving messages that you need to process or if you want to use a session full communication with the queue or if you want to do custom processing of messages based on some message content or if you want to carry out an async operation when you receive the message? You may be able to program these scenarios but the number of lines and the complexity of the code increases.

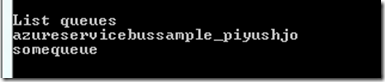
If you think about it – receiving a message and doing “something” with it is exactly what a WCF service does so wouldn’t it be cool if there is an ability to start up a WCF service which will listen for messages received by the queue/subscription and do whatever that you want to do with this message? This is essentially what I meant by the title – Push based programing approach. This means that you don’t have to worry about constantly pulling the messages from the queue – you define a service method (or methods as we will see) to do whatever you want to do when the message is received and the WCF service, like a humble servant, will do it for you.

Today we uploaded a Nuget package which introduce a set of extension methods on the QueueClient and SubscriptionClient which take care of starting up a WCF service instance and then we follow a convention based approach to route the queued messages to the appropriate service method which takes care of handling its processing.

The package walks you through a set of ten scenarios – first five of which explore Service Bus as it is today and the remaining five build on the push based programming model to show how easy it has become now to program against Service Bus.

**Scenario 1** – We enumerate all the **queues** in your namesapce using NamespaceManager.GetQueues() method:

A typical output will look like the following:

[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image53.png)

**Scenario 2** - We enumerate all the **topics** in your namespace using NamespaceManager.GetTopics() method:

A typical output will look like the following:

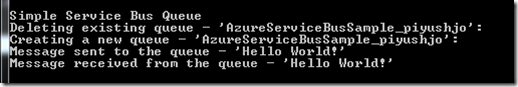
[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image52.png)

**Scenario 3** - We show how you can send and receive messages to a queue using a QueueClient formed from a MessagingFactory:

**Code Snippet**

* 1. // Create a SB queue
  2. QueueDescription queue = CreateQueue();
  3. // Create a factory and client to talk to this queue
  4. MessagingFactory factory = MessagingFactory.Create(ServiceUri, TokenProvider);
  5. QueueClient client = factory.CreateQueueClient(queue.Path);
  6. // Send a message to the queue
  7. string requestMessage = "Hello World!";
  8. client.Send(new BrokeredMessage(requestMessage));
  9. Console.WriteLine("Message sent to the queue - '{0}' ", requestMessage);
  10. // Receive the message from the queue
  11. BrokeredMessage responseQueueMessage = client.Receive();
  12. string responseMessage = responseQueueMessage.GetBody<string>();
  13. Console.WriteLine("Message received from the queue - '{0}' ", responseMessage);
  14. // Cleanup
  15. client.Close();
  16. factory.Close();

This example shows how you program against Service Bus queue right now. A typical output will look like the following:

[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image41.png)

**Scenario 4** - We show how you can send messages to a topic and receive messages via a subscription client:

**Code Snippet**

* 1. string topicName = "news";
  2. string subscriptionName = "mySubscription";
  3. // Delete the topic if it already exists
  4. if (NamespaceManager.TopicExists(topicName))
  5. {
  6. NamespaceManager.DeleteTopic(topicName);
  7. }
  8. // Create the topic
  9. Console.WriteLine("Creating new topic - '{0}'", topicName);
  10. TopicDescription topic = NamespaceManager.CreateTopic(topicName);
  11. // Create a subscription
  12. Console.WriteLine("Creating new subscription - '{0}'", subscriptionName);
  13. SubscriptionDescription subs = NamespaceManager.CreateSubscription(topic.Path, subscriptionName);
  14. // Create a factory to talk to the topic & subscription
  15. MessagingFactory factory = MessagingFactory.Create(ServiceUri, TokenProvider);
  16. // Create a topic client to talk to this topic
  17. TopicClient topicClient = factory.CreateTopicClient(topic.Path);
  18. // Send a message to the topic
  19. string requestMessage = "Hello World!";
  20. Console.WriteLine("Message sent to the topic - '{0}' ", requestMessage);
  21. topicClient.Send(new BrokeredMessage(requestMessage));
  22. // Create a subscription client to receive messages
  23. SubscriptionClient subscriptionClient = factory.CreateSubscriptionClient(topic.Path, subs.Name);
  24. // Receive the messages from the subscription
  25. BrokeredMessage responseQueueMessage = subscriptionClient.Receive();
  26. string responseMessage = responseQueueMessage.GetBody<string>();
  27. Console.WriteLine("Message received from the subscription - '{0}' ", responseMessage);
  28. // Cleanup
  29. subscriptionClient.Close();
  30. topicClient.Close();
  31. factory.Close();

Again – this example shows how you program against Service Bus topic/subscription currently. A typical output will look like the following:

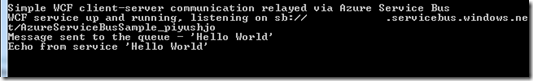
[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image42.png)

**Scenario 5** - We show the WCF integration – how you can create a WCF service which will listen on a Service Bus endpoint and how you can relay messages using a WCF client to this WCF service via the Azure Service Bus.

**Code Snippet**

* 1. NetMessagingBinding binding = new NetMessagingBinding();
  2. // Start up the Echo WCF service listening via Azure Service Bus
  3. ServiceHost host = new ServiceHost(typeof(EchoService));
  4. ServiceEndpoint echoServiceEndpoint
  5. = host.AddServiceEndpoint(typeof(IEchoService), binding, QueueAddress);
  6. echoServiceEndpoint.Behaviors.Add(new TransportClientEndpointBehavior(TokenProvider));
  7. host.Open();
  8. Console.WriteLine("WCF service up and running, listening on {0}", QueueAddress);
  9. // Create a WCF channel factory and client to send message to the Azure Service Bus
  10. ChannelFactory<IEchoService> factory = new ChannelFactory<IEchoService>
  11. (binding, new EndpointAddress(QueueAddress));
  12. factory.Endpoint.Behaviors.Add(new TransportClientEndpointBehavior(TokenProvider));
  13. IEchoService proxy = factory.CreateChannel();
  14. // Send messages to the service
  15. string requestMessage = "Hello World";
  16. proxy.Echo(requestMessage);
  17. Console.WriteLine("Message sent to the queue - '{0}' ", requestMessage);
  18. // clean up the client and service
  19. factory.Close();
  20. host.Close();

Nothing new here - a typical output will look like the following:

[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image43.png)

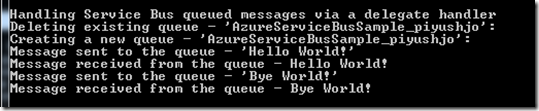
**Scenario 6** – With scenario 6 onwards we start harnessing the power of the Service Bus push based programming library.

We start with a simple scenario applying this new approach to receive messages.

**Code Snippet**

* 1. // Create a ServiceBus queue.
  2. QueueDescription queue = CreateQueue();
  3. // Create a factory and client to talk to this queue
  4. MessagingFactory factory = MessagingFactory.Create(ServiceUri, TokenProvider);
  5. QueueClient client = factory.CreateQueueClient(queue.Path);
  6. // Use the extension method to pass in a delegate
  7. // which will handle all messages received by the queue
  8. client.StartListener(message =>
  9. Console.WriteLine("Message received from the queue - {0}", message.GetBody<string>()));
  10. // Send a message to the queue
  11. string helloMessage = "Hello World!";
  12. client.Send(new BrokeredMessage(helloMessage));
  13. Console.WriteLine("Message sent to the queue - '{0}' ", helloMessage);
  14. // Send another message to the queue
  15. string byeMessage = "Bye World!";
  16. client.Send(new BrokeredMessage(byeMessage));
  17. Console.WriteLine("Message sent to the queue - '{0}' ", byeMessage);
  18. // clean up the client and service
  19. client.StopListener();
  20. client.Close();
  21. factory.Close();

Note line 10 – here we are using the StartListener extension method on QueueClient which takes in a delegate which will handle the messages received by the queue henceforth. A typical output will look like the following:

[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image44.png)

**Scenario 7** – We expand on the previous scenario by explicitly providing a service type which will be responsible for handling the incoming messages to the queue. We also make the scenario interesting by attaching an Action property to each BrokeredMessage we are sending to the queue. The idea is to route messages to different service methods based on this Action property of the incoming message.

**Code Snippet**

* 1. class MyActionOrientedEchoService
  2. {
  3. [OnMessage(Action = "EchoOnce")]
  4. public void EchoOnce(BrokeredMessage msg)
  5. {
  6. Console.WriteLine("Message received by EchoOnce method");
  7. string messageBody = msg.GetBody<string>();
  8. Console.WriteLine(messageBody);
  9. }
  10. [OnMessage(Action = "EchoTwice")]
  11. public void EchoTwice(BrokeredMessage msg)
  12. {
  13. Console.WriteLine("Message received by EchoTwice method");
  14. string messageBody = msg.GetBody<string>();
  15. Console.WriteLine(messageBody);
  16. Console.WriteLine(messageBody);
  17. }
  18. }

**Code Snippet**

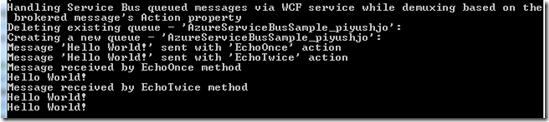
* 1. // Create a ServiceBus queue.
  2. QueueDescription queue = CreateQueue();
  3. // Create a factory and client to talk to this queue
  4. MessagingFactory factory = MessagingFactory.Create(ServiceUri, TokenProvider);
  5. QueueClient client = factory.CreateQueueClient(queue.Path);
  6. // Use the extension method to start the WCF service
  7. // which will handle the queued messages based on its Action property
  8. client.StartListener(typeof(MyActionOrientedEchoService));
  9. string requestMessage = "Hello World!";
  10. // Send message to the queue - to the EchoOnce operation
  11. BrokeredMessage message = new BrokeredMessage(requestMessage);
  12. message.SetAction("EchoOnce");
  13. Console.WriteLine("Message '{0}' sent with '{1}' action", requestMessage, "EchoOnce");
  14. client.Send(message);
  15. // Send message to the queue - to the EchoTwice operation
  16. BrokeredMessage anotherMessage = new BrokeredMessage(requestMessage);
  17. anotherMessage.SetAction("EchoTwice");
  18. Console.WriteLine("Message '{0}' sent with '{1}' action", requestMessage, "EchoTwice");
  19. client.Send(anotherMessage);
  20. // Cleanup
  21. client.StopListener();
  22. client.Close();
  23. factory.Close();

In line 10 – we are again using the StartListener extension method to start up the service of type ‘MyActionOrientedEchoService’. Note the following points about the service:

* 1. the service doesn’t need to implement any interface or be marked with a ServiceContract attribute
  2. the service has two methods marked with the OnMessage attribute. This is a new attribute defined in the sample. All queued messages will be directed to a method decorated with this attribute.
  3. The regular convention though, which is automatically understood, is to use a public method named OnReceivedMessage returning void and accepting a single parameter of type BrokeredMessage which will be used to dispatch the incoming messages. In the absence of such a method, the dispatching is done to a public method with return type void and which accepts a parameter of type BrokeredMessage only if there is exactly one such method. Of course – OnMessage attribute is used to override such conventions.
  4. In this scenario – we want to do custom processing based on received messages and hence we define two service methods each with OnMessage attribute and using the Action parameter which will help in directing the message to the correct operation.

In line 15 and 21, we are using the SetAction extension method to add an ‘action’ property to the BrokeredMessage and this will be used to direct the message to the suitable service method.

A typical output will look like the following:

[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image45.png)

**Scenario 8** – We integrate sessions in this scenario. We see how we can easily spawn up different service instances which will handle the incoming queued messages based on the SessionId we attach to the brokered message.

**Code Snippet**

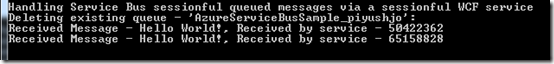
* 1. class MySessionfulService
  2. {
  3. public void OnReceiveMessage(BrokeredMessage msg)
  4. {
  5. Console.WriteLine("Received Message - {0}, Received by service - {1}",
  6. msg.GetBody<string>(), this.GetHashCode());
  7. }
  8. public void OnError(Exception e)
  9. {
  10. Console.WriteLine("Error occurred");
  11. Console.WriteLine(e.Message);
  12. }
  13. }

**Code Snippet**

* 1. // Create a sessionful queue
  2. QueueDescription queue = NamespaceManager.CreateQueue
  3. (new QueueDescription(QueueName) { RequiresSession = true });
  4. // Create a factory and client to talk to this queue
  5. MessagingFactory factory = MessagingFactory.Create(ServiceUri, TokenProvider);
  6. QueueClient client = factory.CreateQueueClient(queue.Path);
  7. // Use the extension method to start up the WCF service
  8. // which will process the sessionful queued messages received by the queue
  9. client.StartListener(typeof(MySessionfulService), requiresSession: true);
  10. string helloMessage = "Hello World!";
  11. string byeMessage = "Bye World!";
  12. // Start sending messages to the queue
  13. // Session 1
  14. BrokeredMessage message1 = new BrokeredMessage(helloMessage);
  15. message1.SessionId = "1";
  16. client.Send(message1);
  17. BrokeredMessage message2 = new BrokeredMessage(byeMessage);
  18. message2.SessionId = "1";
  19. client.Send(message2);
  20. // Session 2
  21. BrokeredMessage message3 = new BrokeredMessage(helloMessage);
  22. message3.SessionId = "2";
  23. client.Send(message3);
  24. BrokeredMessage message4 = new BrokeredMessage(byeMessage);
  25. message4.SessionId = "2";
  26. client.Send(message4);
  27. // Cleanup
  28. client.Close();
  29. factory.Close();

In line 2, we are ensuring that the ServiceBus queue enables sessions and correspondingly in line 11, we are starting up the WCF service listener enabling sessions. Note that in the MySessionfulService implementation we are sticking to the convention of using the method with the signature – void OnReceivedMessage (BrokeredMessage msg) which will handle all incoming messages. Also note another convention based method void OnError(Exception ex) – this is used to handle all the errors arising out of handling the queued messages. Again – for handling errors - we will look for a method which conforms to this signature if an OnError method does not exist. Internally, we are simply registering an IErrorHandler that redirects Exceptions to the error handler. This means that anything that is bubbled up to the IErrorHandler by ServiceModel will make its way to the error handler

A typical output will look like the following:

[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image46.png)

Note that I can also add a service behavior specifying the InstanceContextMode of my WCF service if I so desire – by default it is PerSession but I can change it to PerCall or Single to influence how my service method processes the incoming messages.

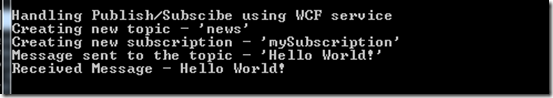
**Scenario 9** – We extend the example to show how you can use the same WCF based push approach to receive messages for a subscription from a topic.

**Code Snippet**

* 1. // Create the topic
  2. Console.WriteLine("Creating new topic - '{0}'", topicName);
  3. TopicDescription topic = NamespaceManager.CreateTopic(topicName);
  4. // Create a subscription
  5. Console.WriteLine("Creating new subscription - '{0}'", subscriptionName);
  6. SubscriptionDescription subs = NamespaceManager.CreateSubscription(topic.Path, subscriptionName);
  7. // Create a factory to talk to the topic & subscription
  8. MessagingFactory factory = MessagingFactory.Create(ServiceUri, TokenProvider);
  9. // Create a topic client to talk to this topic
  10. TopicClient topicClient = factory.CreateTopicClient(topic.Path);
  11. // Create a subscription client to retrive messages
  12. SubscriptionClient subscriptionClient = factory.CreateSubscriptionClient(topic.Path, subs.Name);
  13. // Start the WCF service which will receive the messages sent to the topic
  14. subscriptionClient.StartListener(typeof(MyQueuedMessageHandlerService));
  15. // Send a message to the topic
  16. string requestMessage = "Hello World!";
  17. Console.WriteLine("Message sent to the topic - '{0}' ", requestMessage);
  18. topicClient.Send(new BrokeredMessage(requestMessage));
  19. // Cleanup
  20. subscriptionClient.StopListener();
  21. subscriptionClient.Close();
  22. topicClient.Close();
  23. factory.Close();

All the overloads which exist on the QueueClient, as we saw above, are also available on the SubscriptionClient. A WCF service will be started up in a similar way to handle all messages received by this subscription.

A typical output will look like the following:

[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image49.png)

**Scenario 10** – We show how you can use the same approach to carry out an async operation at the receipt of the queued message.

**Code Snippet**

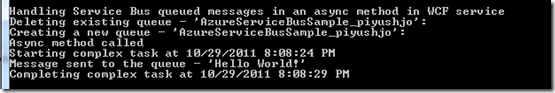
* 1. class MyAsyncEchoService
  2. {
  3. public Task DoComplexTaskAsync(BrokeredMessage message)
  4. {
  5. Console.WriteLine("Async method called");
  6. return Task.Factory.StartNew(() =>
  7. {
  8. // Do some complex task
  9. Console.WriteLine("Starting complex task at {0}", DateTime.Now.ToString());
  10. Thread.Sleep(5000);
  11. Console.WriteLine("Completing complex task at {0}", DateTime.Now.ToString());
  12. });
  13. }
  14. }

**Code Snippet**

* 1. // Create a ServiceBus queue.
  2. QueueDescription queue = CreateQueue();
  3. // Create a factory and client to talk to this queue
  4. MessagingFactory factory = MessagingFactory.Create(ServiceUri, TokenProvider);
  5. QueueClient client = factory.CreateQueueClient(queue.Path);
  6. // Use the extension method to start the WCF service
  7. // which will handle the queued messages based on its Action property
  8. client.StartListener(typeof(MyAsyncEchoService));
  9. // Send message to the queue
  10. string requestMessage = "Hello World!";
  11. BrokeredMessage message = new BrokeredMessage("Hello World!");
  12. client.Send(message);
  13. Console.WriteLine("Message sent to the queue - '{0}' ", requestMessage);
  14. // Cleanup
  15. client.StopListener();
  16. client.Close();
  17. factory.Close();

The service implementation is the only interesting part otherwise the sample is identical to what we have seen in earlier scenarios. Note how the method takes in a BrokeredMessage and returns a Task.

A typical output will look like the following:

[](file:///C:\Users\piyushjo\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles4E50D13\image51.png)