Agenda: Azure Service Bus

- Service Bus Basics.
- Relayed Messaging.
- Service Bus Queues.
- Topics and Subscriptions.
- Handling Transactions.

Azure Service Bus

- Whether an application or service runs in the cloud or on-premises, it often needs to interact with other applications or services. To provide a broad communication channel between different applications, Azure provides a **secure infrastructure** called as Service Bus.
- Service Bus allows communication between **on-premises solutions to Microsoft Azure solutions**, and even Microsoft Azure solutions to other solutions within the cloud.
- It provides connectivity options for Windows Communication Foundation (WCF), which includes REST endpoints.
- It provides "Relayed" and "Brokered" messaging capabilities and uses ACS (Access Control Service) to grant or deny access to the services it exposes.
    - ***Relayed Messaging**:* It helps disparate applications and services communicate through firewalls, NAT gateways and other network boundaries. It supports direct one-way messaging, request/response messaging, and peer-to-peer messaging. It is only possible when the message sender and receiver **are both online at the same time**.
    - **Brokered Messaging**: It is asynchronous communication in which sender and receiver **do not have to be online at the same time**. Messaging infrastructure stores messages in a "broker" (such as queue) until receiver application is ready to receive them. This allows the various components of application to be disconnected.
- Service Bus is a multi-tenant cloud service, which means that the service is shared by multiple users. Each user, such as an application developer, creates a ***namespace***, then defines the communication mechanisms she needs within that namespace.
- Within a namespace, you can use one or more instances of three different communication mechanisms, each of which connects applications in a different way.

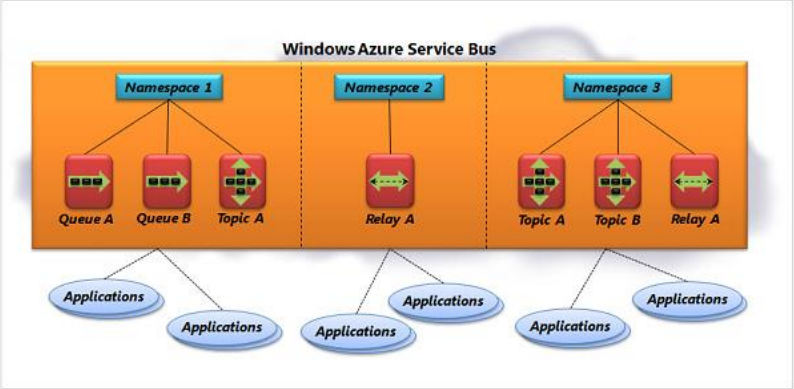
The choices are:

1. ***Relays***, which provide bi-directional communication. Unlike queues and topics, a relay doesn't store in-flight messages-it's not a broker. Instead, it just passes them on to the destination application.
2. ***Queues***, which allow one-directional communication. Each queue acts as an intermediary (sometimes called a *broker*) that stores sent messages until they are received. Each message is received by a single recipient.

3. ***Topics***, which provide one-directional communication using *subscriptions*-a single topic can have multiple subscriptions. Like a queue, a topic acts as a broker, but each subscription can optionally use a filter to receive only messages that match specific criteria.

Namespaces

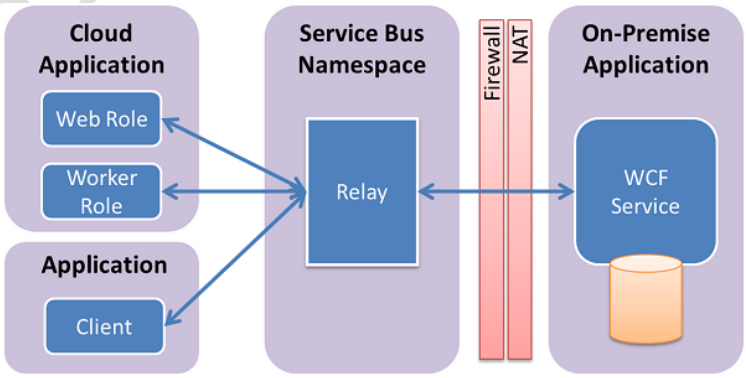
- Namespaces serve as a basic **logical grouping** of Service Bus service instances.
- When you create a queue, topic, or relay, you give it a name. The instance name is then combined the name of your namespace to create a **unique identifier** for the object. Applications can provide this name to Service Bus, and then use that queue, topic, or relay to communicate with one another.
- Service Bus namespaces can also contain **management credentials**, or shared keys, that your client applications can use to connect to Service Bus.



Relayed Messaging

Suppose your applications need to both send and receive messages, or perhaps you want a direct link between them and you don't need a broker to store messages. To address scenarios such as this, Service Bus provides relays.

Relay component can connect existing services to new client applications without exposing the true location or address of the service.



- Relays are the right solution when you need direct communication between applications. For example, consider an airline reservation system running in an on-premises datacenter that must be accessed from

check-in kiosks, mobile devices, and other computers. Applications running on all of these systems could rely on Service Bus relays in the cloud to communicate, wherever they might be running.

- To communicate bi-directionally through a relay, each application establishes an outbound TCP connection with Service Bus, then keeps it open. All communication between the two applications will travel over these connections. Because each connection was established from inside the datacenter, the firewall will allow incoming traffic to each application without opening new ports.
- When an application that wishes to receive messages establishes a TCP connection with Service Bus, **a relay is created automatically**. When the connection is dropped, the relay is deleted.
- To use Service Bus relays, applications rely on the Windows Communication Foundation (WCF). Service Bus provides WCF bindings that make it straightforward for Windows applications to interact via relays.

***Service Bus Bindings**:* The Service Bus uses numerous bindings between senders and receivers that determine how the connection to the Service Bus is made.

WCF Binding	**Relay Binding**
BasicHttpBinding	BasicHttpRelayBinding
WebHttpBinding	WebHttpRelayBinding
WS2007HttpBinding	WS2007HttpRelayBinding
NetTcpBinding	NetTcpRelayBinding
N/A	NetOnewayRelayBinding
N/A	NetEventRelayBinding

Steps for the exercise

1. Azure Portal → Service Bus → +Add
   a. Service Bus → Create, Provide Namespace="dssdemoservicebus" → OK
   b. Select Sevice Bus → Service Bus Essentials → Connection Strings → Click on RootManagerSharedAccessKey → Copy Primary Key and ConnectionString for further use.
2. Visual Studio → File → New Project → Visual C# → Console Application → Name=AddService → OK.
3. Right click References→Manage Nuget Packages→Install **Windows Azure Service Bus**.
   Note that the NuGet package has already added a range of definitions to the App.config file, which are the required configuration extensions for Service Bus.
4. Edit Program.cs

```
using System.ServiceModel;
using Microsoft.ServiceBus;
using System;

namespace DemoService
{
```

```
[ServiceContract()]
interface IMathService
{
    [OperationContract]
    int Add(int a, int b);
}
class MathService : IMathService
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
}
```

5. Providing EndPoint and its Behavior in code:

```
class Program
{
    static void Main(string[] args)
    {
        ServiceHost sh = new ServiceHost(typeof(MathService));
        var endpoint = sh.AddServiceEndpoint(
          typeof(IMathService),  //Contract
          new NetTcpRelayBinding(), //Binding
          ServiceBusEnvironment.CreateServiceUri("sb", "<servicebusNamespace>", "math")); //Address
        endpoint.Behaviors.Add(new TransportClientEndpointBehavior
        {
            TokenProvider =
TokenProvider.CreateSharedAccessSignatureTokenProvider("RootManageSharedAccessKey", "<ServiceBusKey>")
        });
//Endpoint=sb://servicebusdemo.servicebus.windows.net/math;SharedAccessKeyName=RootManageSharedAcces
sKey;SharedAccessKey=xMVCzztGHEbH641RosoBeYcRXLiGb51gUa1Q9bPibEE=
        sh.Open();
        Console.WriteLine("Press ENTER to close");
        Console.ReadLine();
        sh.Close();
    }
```

```
}
```

OR

Add the following to <system.serviceModel> in the App.Config file and **delete** call to AddServerEndPoint from Main method.

```
<system.ServiceModel>
<services>
 <service name="DemoService.MathService">
  <endpoint contract="DemoService.IMathService"
        binding="netTcpRelayBinding"
        address="sb://dssrelayservicebus.servicebus.windows.net/math"
        behaviorConfiguration="sbTokenProvider"/>
 </service>
</services>
<behaviors>
 <endpointBehaviors>
  <behavior name="sbTokenProvider">
   <transportClientEndpointBehavior>
    <tokenProvider>
     <sharedAccessSignature keyName="RootManageSharedAccessKey"
key="f24VIbPGurwIBKZN/IWa/MIF/Ly0GF8PIDkXsJ/TZEM=" />
    </tokenProvider>
   </transportClientEndpointBehavior>
  </behavior>
 </endpointBehaviors>
</behaviors>
</system.ServiceModel>
```

6. Run a Console Application to Register with Service Bus

Add another Console Application Project (AddClient) to the Solution.

7. Right click References→Manage NuGet Packages→Install Windows Azure Service Bus.
8. Edit Program.cs

```
using Microsoft.ServiceBus;
using System;
using System.ServiceModel;
namespace DemoClient
{
```

```
    [ServiceContract()]
    interface IMathService
    {
        [OperationContract]
        int Add(int a, int b);
    }
    interface IMathServiceChannel : IMathService, IClientChannel { }
}
```

9. Edit Program.cs in client application

```
    class Program
    {
        static void Main(string[] args)
        {
            var factory = new ChannelFactory<IMathServiceChannel>(
                                new NetTcpRelayBinding(),
                                new EndpointAddress(ServiceBusEnvironment.CreateServiceUri("sb",
"dssdemoservicebus", "math")));
            factory.Endpoint.Behaviors.Add(new TransportClientEndpointBehavior
                        { TokenProvider =
TokenProvider.CreateSharedAccessSignatureTokenProvider("RootManageSharedAccessKey",
"f24VlbPGurwIBKZN/lWa/MIF/Ly0GF8PIDkXsJ/TZEM=") });
            using (var ch = factory.CreateChannel())
            {
                Console.WriteLine(ch.Add(4, 5));
            }
        }
    }
}
```

OR

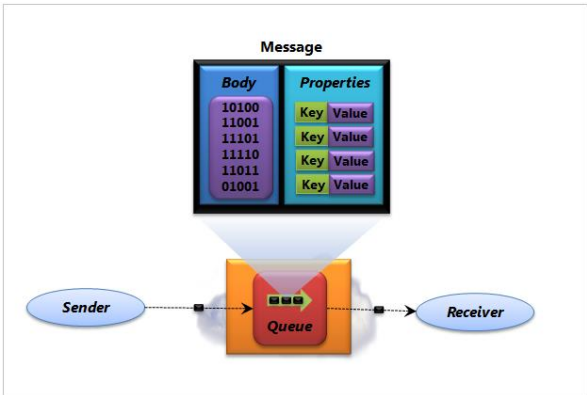
Add the following to <system.ServiceModel> in App.Config and update Main method as provided below.

```
<client>
  <endpoint name="MathEndPoint" contract="DemoClient.IMathService"
            binding="netTcpRelayBinding"
            address="sb://dssdemoservicebus.servicebus.windows.net/math"
            behaviorConfiguration="sbTokenProvider"/>
```

```
</client>
<behaviors>
 <endpointBehaviors>
  <behavior name="sbTokenProvider">
   <transportClientEndpointBehavior>
    <tokenProvider>
     <sharedAccessSignature keyName="RootManageSharedAccessKey"
key="f24VlbPGurwIBKZN/IWa/MIF/Ly0GF8PIDkXsJ/TZEM=" />
    </tokenProvider>
   </transportClientEndpointBehavior>
  </behavior>
 </endpointBehaviors>
</behaviors>
class Program
{
  static void Main(string[] args)
  {
    var factory = new ChannelFactory<IMathServiceChannel>("MathEndPoint");
    using (var ch = factory.CreateChannel())
    {
      Console.WriteLine(ch.Add(4, 5));
    }
  }
}
```

Service Bus Queues

Queues represent a persistent sequenced buffer into which **one or more senders** / **producer** send messages to **one or more receivers / consumer.** They are based on the first in first out (FIFO) model. They provide various methods to indicate for example time visibility of messages on the queue and the ability of messages to reappear on the queue, duplicate detection, deferred messaging etc.



Load Balancing: If the load in queue increases, more worker process can be added to read messages from queue. Each message will be processed by **only one** of the consumer. Consumer computer may differ in processing power, as they will pull messages from queue depending on their own capability/processing power, this pattern is often termed as "**competing consumer**" pattern.

Azure supports two types of queue mechanisms: Storage queues and Service Bus queues. Now that you have learned about Service Bus queues you may be wondering how they are different from Storage queues. This table provides a summary.

Comparison Criteria	**Storage Queues**	**Service Bus Queues**
Ordering guarantee	No	Yes – FIFO (Sessions)
Delivery guarantee	At-Least-Once	At-Least-Once (PeekAndLook) At-Most-Once (ReceiveAndDelete)
Lease/lock level	Message level	Queue level
Batch receive	Yes	Yes
Batch send	No	Yes
Scheduled delivery	Yes	Yes
Automatic dead lettering	No	Yes
Message auto-forwarding	No	Yes
Message groups	No	Yes
Duplicate detection	No	Yes
TTL	Max 7 days	---

Notice if your queue size will exceed **80 GB**, then you must use storage queues (**500TB**).

Walkthrough:

1. Please ensure that you have already created a Service Bus namespace as shown in the previous example
2. Azure Portal → Service Bus → Select Namespace → Queue Tab → Create a New Queue → Quick Create
3. Queue Name = DemoQueue→. . . → OK

- **Message time to live**. Determines how long a message will stay in the queue before it expires and is removed or dead lettered. This default will be used for all messages in the queue which do not specify a time to live for themselves.
- **Lock duration**. Sets the amount of time a message is locked from other receivers. After its lock expires, a message is pulled by one receiver before being available to be pulled by other receivers. The default is 30 seconds, with a maximum of 5 minutes.
- **Enable duplicate detection**. Configures your queue to keep a history of all messages sent to the queue during a configurable amount of time. During that interval, your queue will not accept any duplicate messages.
- **Enable dead lettering**. Enables holding messages that cannot be successfully delivered to any receiver. The messages are held in a separate queue after they expire. You can inspect this queue.
- **Enable sessions**. Allows ordered handling of unbound sequences of related messages. This guarantees first-in-first-out delivery of messages.
- **Enable Partitions**. Partitions a queue across multiple message brokers and message stores. Partitioning means that the overall throughput of a partitioned entity is no longer limited by the performance of a single message broker or messaging store. In addition, a temporary outage of a messaging store does not render a partitioned queue or topic unavailable.

Programming in C#

4. Create a Console Application (Sender)
5. Manage NuGet Package → Search **Microsoft Azure Service Bus and Reference**
6. Add Reference→ System.Configuration
7. Edit the following in App.Config

```
<appSettings>
 <add key="Microsoft.ServiceBus.ConnectionString"
 Value =
"Endpoint=sb://<namespace>.servicebus.windows.net;SharedAccessKeyName=RootManageSharedAccessKey;Shar
edAccessKey=<key>"/>
</appSettings>
```

8. Edit Program.cs

```
class Program
{
   static void Main(string[] args)
   {
      var connectionString = ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];
      var queueName = "DemoQueue";
```

```
        var client = QueueClient.CreateFromConnectionString(connectionString, queueName);

        var message = new BrokeredMessage("This is a test message sent at " + DateTime.Now.ToLongTimeString());
        client.Send(message);
        Console.WriteLine(message.MessageId);
    }
}
```

To send a **group of messages**, you can use the **SendBatchAsync** method. Both methods require that you encapsulate the queue message in an object of type Message:

```
var messages = new List<BrokeredMessage>();
for (int i = 0; i < 10; i++)
{
    message = new BrokeredMessage($"Message {i}");
    messages.Add(message);
}
client.SendBatch(messages);
```

9. Run the Server Console Application for posting messages into the queue.
10. Repeat steps 4,5,6 and 7 for Client Application
11. Edit Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        var connectionString = ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];
        var queueName = "DemoQueue";
        var client = QueueClient.CreateFromConnectionString(connectionString, queueName,
ReceiveMode.PeekLock);

        BrokeredMessage msg = client.Receive();
//Receive will put the message in Lock state until either Complete() or Abandon() method is called. After 60 secs
it's automatically abandoned.
        Console.WriteLine("First Queue Message: " + msg.GetBody<string>());
        msg.Complete(); //the queue deletes the message.
    }
```

```
}
```

12. Run the Client Application and note the string messages received.

Note: Service Bus sessions enable joint and ordered handling of unbounded sequences of related messages. To realize a **FIFO guarantee** in Service Bus you need to use **Sessions**. Any sender can create a session when submitting messages into a topic or queue by setting the SessionId broker property to some application-defined identifier that is unique to the session.

Custom (Complex) Objects:

If the Message is posted as below:

```
[Serializable]
class Person
{
   public string Name { get; set; }
   public string Location { get; set; }
   public int Age { get; set; }
}
Person p = new Person() { Name = "P1", Age = 20, Location = "USA" };
var message = new BrokeredMessage(p);
client.Send(message);
```

It should be retrieved as below:

```
BrokeredMessage msg = client.Receive();
Person p = msg.GetBody<Person>();
Console.WriteLine("{0} {1} {2}", p.Name, p.Age, p.Location);
msg.Complete();
```

Message Properties:

If the Message is posted as below:

```
var message = new BrokeredMessage("These are details of Person");
message.Properties["Name"] = "P1";
message.Properties["Age"] = 20;
message.Properties["Location"] = "USA";
client.Send(message);
```

It should be retrieved as below:

```
BrokeredMessage msg = client.Receive();
Console.WriteLine(msg.GetBody<string>());
string name = msg.Properties["Name"].ToString();
int age = Convert.ToInt32(msg.Properties["Age"]);
string location = msg.Properties["Location"].ToString();
Console.WriteLine("{0} {1} {2}", name, age, location);
msg.Complete();
```

In a receiver application we can use the following **To Process a message in an event driven event pump**:

```
client.OnMessage(message =>
{
   Person p = message.GetBody<Person>();
   Console.WriteLine(String.Format("Message body: {0} {1} {2}", p.Name, p.Age, p.Location));
});
```

Receiving messages from a queue:

Namespace: **Microsoft.Azure.ServiceBus**

Assembly: Microsoft.Azure.ServiceBus.dll (.NET Standard Library)

Using the same queue client, you can register a method that will handle any new messages that arrive asynchronously:

```
queueClient.RegisterMessageHandler(MethodToHandleNewMessage, new RegisterHandlerOptions());
```

Now, you will need to implement a new method to handle each message that may come in. Within this method, you can implement any business logic you like, but it must conform to a specific method signature—Handler(Message, CancellationToken). As an example, this method will write each message to the console:

```
static async Task MethodToHandleNewMessage(BrokeredMessage message, CancellationToken token)
{
   var connectionString = ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];
   var queueName = "DemoQueue";
   var queueClient = QueueClient.CreateFromConnectionString(connectionString, queueName,
ReceiveMode.PeekLock);
   string messageString = message.GetBody<string>();
   Console.WriteLine($"Received message: {messageString}");
   await queueClient.CompleteAsync(message.LockToken);
}
```

You will notice that we need to call the CompleteAsync method of the queue client at the end of the message handler method. This ensures that the message is not received again. Alternatively, you can call AbandonAsync if you wish to stop handling the message and receive it again.

Dead Letter Queue Messages

- Messages that expire before being received are called as Dead Letter Messages.
- Expiring a message can be useful in scenarios where the message has no meaning after certain period of time. For example Weather forecasting website may not be interested in updating yesterday's weather forecast.
- Setting expiration on messages that are not relevant to consumer will reduce the size of queue and will prevent the application with additional burdon of receiving and discarding those messages. Thus overall improving the performance of the system.
- While creating a queue it is possible to specify **default message time to live**, also we can specify that the expired messages will be dead-lettered instead of getting ignored.

1. Write the following in Sender

```
class Program
{
   static void Main(string[] args)
   {
      var connectionString = ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];

//To Create the Queue Programmatically
      string queueName = "MyDemoQueue";
      QueueDescription myQueue = new QueueDescription(queueName);
      myQueue.DefaultMessageTimeToLive = new TimeSpan(0, 100, 0);
      myQueue.EnableDeadLetteringOnMessageExpiration = true;
      myQueue.LockDuration = new TimeSpan(0, 0, 15);

      var namespaceClient = NamespaceManager.CreateFromConnectionString(connectionString);
      if (!namespaceClient.QueueExists(queueName))
         namespaceClient.CreateQueue(myQueue);

      QueueClient myQueueClient = QueueClient.CreateFromConnectionString(connectionString, queueName);
      var message = new BrokeredMessage("This msg will expire after 10 secs");
      message.TimeToLive = new TimeSpan(0, 0, 10);
      myQueueClient.Send(message);
```

```
    }
}
```

Alternative API to create a Queue:

Example: To create a Queue to store messages for 10 mins before the application uses the message:

```
var address = ServiceBusEnvironment.CreateServiceUri("sb",
"mynamespace.servicebus.windows.net/applicationName", string.Empty)
var namespaceClient = new NamespaceManager(address, new NamespaceManagerSettings()
                                                    { OperationTimeout = new TimeSpan(0,10,0)} );
ns.CreateQueue();
```

2. Write the following in Receiver:

```
class Program
{
   static void Main(string[] args)
   {
      var connectionString = ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];
      var queueName = "MyDemoQueue";
      QueueClient myQueueClient = QueueClient.CreateFromConnectionString(connectionString, queueName);

//To get the reference to Dead Letter Queue
      string deadQueueName = QueueClient.FormatDeadLetterPath(myQueueClient.Path);
//MyDemoQueue/$DeadLetterQueue
      QueueClient deadQueueClient = QueueClient.CreateFromConnectionString(connectionString,
deadQueueName, ReceiveMode.PeekLock);

      BrokeredMessage bm = deadQueueClient.Receive(new TimeSpan(0,0,0)); //Default wait time is 30 secs
      if (bm == null)
         Console.WriteLine("No messages in Dead Letter Queue");
      else {
         Console.WriteLine(bm.GetBody<string>());
         bm.Complete();
      }
   }
}
```

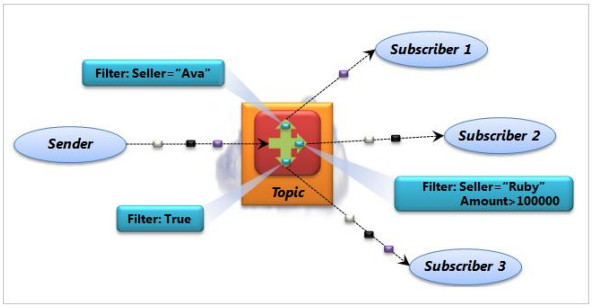
Topics and Subscriptions

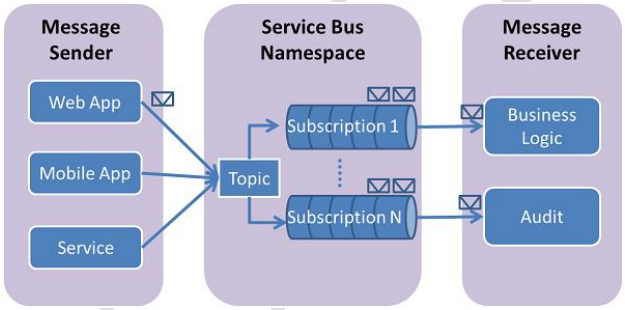
Topics extend the messaging features provided by Queues with the addition of **Publish-Subscribe** Mechanism.

Senders submit messages to a topic in the same way that they submit messages to a queue, and those messages look the same as with queues.

Based on the filter a subscribing application specifies, it can receive some or all of the messages sent to a Service Bus topic.

Unlike queues, however, a single message sent to a topic can be received by multiple subscriptions. This approach, commonly called *publish and subscribe* (or *pub/sub*), is useful whenever multiple applications are interested in the same messages





Very Important

A message is submitted from topic into all the subscription **queues** based on filter condition.

Message is received from Subscription

From every subscription (**queue)**, the message will be received by an application only once (even if multiple application instances are running).

Sender Console Application:

```
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;
using System;
using System.Configuration;

//Please ensure that the Course class here and in Receiver Application has SAME/NO NAMESPACE
[Serializable()]
public class Course
{
   public string CourseName { get; set; }
   public string Prerequisite { get; set; }
   public decimal Fees { get; set; }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      var connectionString = ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];

      string topicName = "Courses";
      // Configure Topic Settings.
      TopicDescription td = new TopicDescription(topicName);
      td.MaxSizeInMegabytes = 5120;
      td.DefaultMessageTimeToLive = new TimeSpan(0, 1, 0);

      var namespaceManager = NamespaceManager.CreateFromConnectionString(connectionString);
      if (!namespaceManager.TopicExists(topicName))
      {
         namespaceManager.CreateTopic(td);
      }

      TopicClient client = TopicClient.CreateFromConnectionString(connectionString, topicName);

      Course c1 = new Course() { CourseName = "C#", Fees = 1000 };
      BrokeredMessage bm1 = new BrokeredMessage(c1);
      bm1.Properties["category"] = "basic";
      client.Send(bm1);

      Course c2 = new Course() { CourseName = "Java", Fees = 1000 };
      BrokeredMessage bm2 = new BrokeredMessage(c2);
      bm2.Properties["category"] = "basic";
      client.Send(bm2);

      Course c3 = new Course() { CourseName = "Azure", Fees = 3000 };
      BrokeredMessage bm3 = new BrokeredMessage(c3);
      bm3.Properties["category"] = "advanced";
      client.Send(bm3);

      Course c4 = new Course() { CourseName = "SharePoint", Fees = 2000 };
```

```
        BrokeredMessage bm4 = new BrokeredMessage(c4);
        bm4.Properties["category"] = "advanced";
        client.Send(bm4);

        client.Close();
    }
}
```

Subscriber Console Application:

```
//Please ensure that the Course class here and in Receiver Application has SAME/NO NAMESPACE
[Serializable()]
public class Course
{
    public string CourseName { get; set; }
    public string Prerequisite { get; set; }
    public decimal Fees { get; set; }
}
class Program
{
    static void Main(string[] args)
    {
        var connectionString = ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];
        string topicName = "courses";
        var namespaceManager = NamespaceManager.CreateFromConnectionString(connectionString);

        if (!namespaceManager.SubscriptionExists(topicName, "BasicCourses"))
        {
            SqlFilter filter = new SqlFilter("category='basic'");
            namespaceManager.CreateSubscription(topicName, "BasicCourses", filter);
        }

        if (!namespaceManager.SubscriptionExists(topicName, "AdvancedCourses"))
        {
            SqlFilter filter = new SqlFilter("category='advanced'");
             namespaceManager.CreateSubscription(topicName, "AdvancedCourses", filter);
        }
```

```
    if (!namespaceManager.SubscriptionExists(topicName, "AllCourses"))
    {
        namespaceManager.CreateSubscription(topicName, "AllCourses");
    }


    SubscriptionClient basicClient = SubscriptionClient.CreateFromConnectionString(connectionString, topicName,
"BasicCourses", ReceiveMode.PeekLock);
    SubscriptionClient advancedClient = SubscriptionClient.CreateFromConnectionString(connectionString,
topicName, "AdvancedCourses", ReceiveMode.PeekLock);
    SubscriptionClient allClient = SubscriptionClient.CreateFromConnectionString(connectionString, topicName,
"AllCourses", ReceiveMode.PeekLock);

    basicClient.OnMessage((message) =>
    {
        Course course = message.GetBody<Course>();
        Console.WriteLine("BasicCourse: " + course.CourseName);
        message.Complete();
    });

    advancedClient.OnMessage((message) =>
    {
        Course course = message.GetBody<Course>();
        Console.WriteLine("AdvancedCourse: " + course.CourseName);
        message.Complete();
    });

    allClient.OnMessage((message) =>
    {
        Course course = message.GetBody<Course>();
        Console.WriteLine("AllCourse: " + course.CourseName);
        message.Complete();
    });
    Console.ReadLine();
  }
```

```
}
```

Note:

1. Run the Sender for creation of Topic. End the Sender application.
2. Run Multiple instances of Receiver Application. The first instance creates Subscribers (Queues).
3. Run the Sender multiple times and note that every message is processed by only by any one subscriber client based on their respective filter.

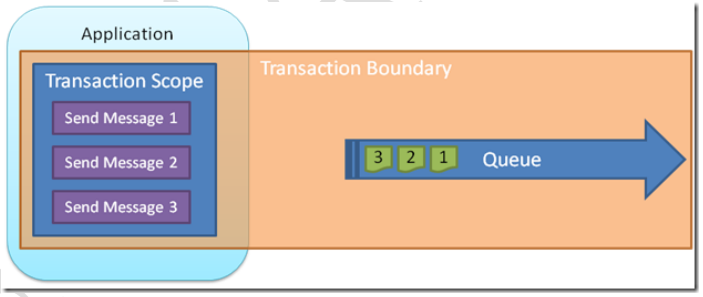
Transactions in Brokered Messaging

Transaction support in Service Bus Brokered Messaging allows message operations to be performed within a transactional scope; however there are some limitations around what operations can be performed within the transaction.

In the current release, only one top level messaging entity, such as a **only one queue** or or **only one topic** can participate in a transaction, and the transaction cannot include any other transaction resource managers, **making transactions spanning a messaging entity (multiple queues or topic) and a database not possible.**

Sending Transactional Messages:

When sending messages, the send operations can participate in a transaction allowing multiple messages to be sent within a transactional scope. This allows for "**all or nothing**" delivery of a series of messages to a single queue or topic.



An example of the code used to send 10 messages to a queue as a single transaction from a console application is shown below.

To the project add reference to System.Transactions.

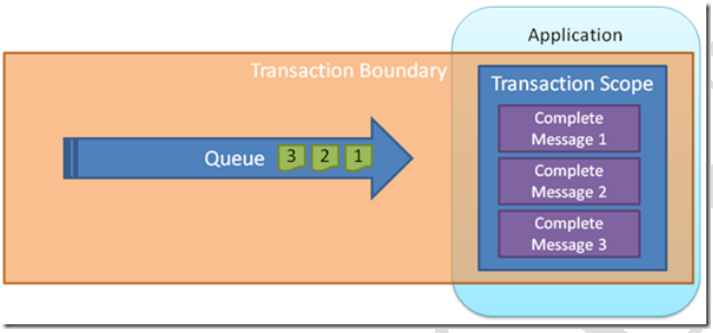
```
using Microsoft.ServiceBus.Messaging;
using System;
using System.Configuration;
using System.Text;
using System.Transactions;

namespace TransactionQueueSenderDemoApp
```

```
{
  class Program
  {
    static void Main(string[] args)
    {
      var connectionString = ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];
      var queueName = "DemoQueue";
      var queueClient = QueueClient.CreateFromConnectionString(connectionString, queueName);
      Console.Write("Sending");

      // Create a transaction scope.
      using (TransactionScope scope = new TransactionScope())
      {
        for (int i = 0; i < 10; i++)
        {
          // Send a message
          BrokeredMessage msg = new BrokeredMessage("Message: " + i);
          msg.PartitionKey = "demopartition";
          queueClient.Send(msg);
          Console.Write(".");
        }
        Console.WriteLine("Done!");
        Console.WriteLine();

        // Should we commit the transaction?
        Console.WriteLine("Commit send 10 messages? (yes or no)");
        string reply = Console.ReadLine();
        if (reply.ToLower().Equals("yes"))
        {
          // Commit the transaction.
          scope.Complete();
        }
      }
      Console.WriteLine();
    }
  }
```

```
}
```

Receiving Multiple Messages in a Transaction

The receiving of multiple messages is another scenario where the use of transactions can improve reliability. When receiving a group of messages that are related together, maybe in the same message session, it is possible to receive the messages in the peek-lock receive mode, and then complete, defer, or deadletter the messages in one transaction. (In the current version of Service Bus, abandon is not transactional.)



The following code shows how this can be achieved.

```
using Microsoft.ServiceBus.Messaging;
using System;
using System.Configuration;
using System.Transactions;

namespace TransactionQueueReceiverDemoApp
{
  class Program
  {
    static void Main(string[] args)
    {
      var connectionString = ConfigurationManager.AppSettings["Microsoft.ServiceBus.ConnectionString"];
      var queueName = "DemoQueue";
      var queueClient = QueueClient.CreateFromConnectionString(connectionString, queueName);
      using (TransactionScope scope = new TransactionScope())
      {
        while (true)
        {
          // Receive a message.
          BrokeredMessage msg = queueClient.Receive(TimeSpan.FromSeconds(1));
          if (msg != null)
```

```
            {
                // Wrote message body and complete message.
                string text = msg.GetBody<string>();
                Console.WriteLine("Received: " + text);
                msg.Complete();
            }
            else
            {
                break;
            }
        }
        Console.WriteLine();
        // Should we commit?
        Console.WriteLine("Commit receive? (yes or no)");
        string reply = Console.ReadLine();
        if (reply.ToLower().Equals("yes"))
        {
            // Commit the transaction.
            scope.Complete();
        }
        Console.WriteLine();
    }
  }
 }
}
```

Note that if there are a large number of messages to be received, there will be a chance that the transaction may time out before it can be committed. It is possible to specify a longer timeout when the transaction is created, but It may be better to receive and commit smaller amounts of messages within the transaction

Session Service Bus:

http://social.technet.microsoft.com/wiki/contents/articles/32561.azure-service-bus-messaging-with-queues-using-sessions.aspx