Azure Service Bus

- Service Bus Basics Introduction
- Relayed Messaging.
- Service Bus Queues.
- Topics and Subscriptions.
- Handling Transactions.
- FIFO using Sessions

Azure Service Bus Introduction

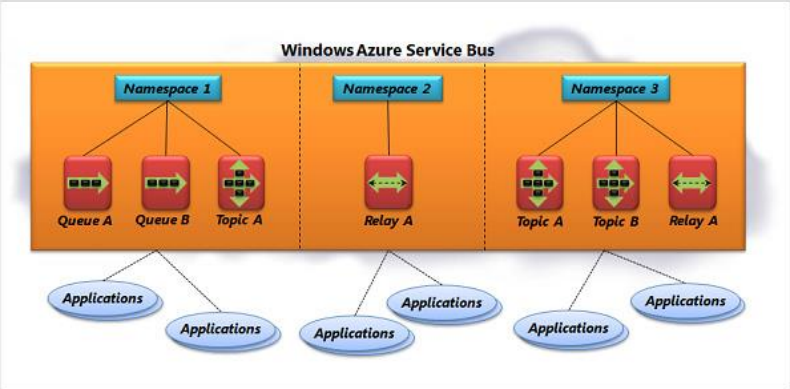
- Whether an application or service runs in the cloud or on-premises, it often needs to interact with other applications or services. To provide a broad communication channel between different applications, Azure provides a **secure infrastructure** called as Service Bus.
- Service Bus allows communication 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.

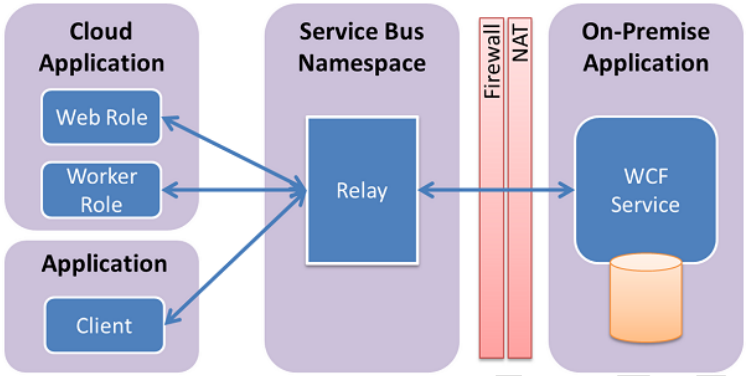


Relayed Messaging

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.

- 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.

- The **Azure Relay service** enables you to securely expose services that run in your corporate network to the public cloud. You can do so without opening a port on your firewall, or making intrusive changes to your corporate network infrastructure.



- 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.

Azure Relay has two features:

- **Hybrid Connections** - Uses the open standard web sockets enabling multi-platform scenarios.
- **WCF Relays** - Uses Windows Communication Foundation (WCF) to enable remote procedure calls. WCF Relay is the **legacy** relay offering that many customers already use with their WCF programming models.

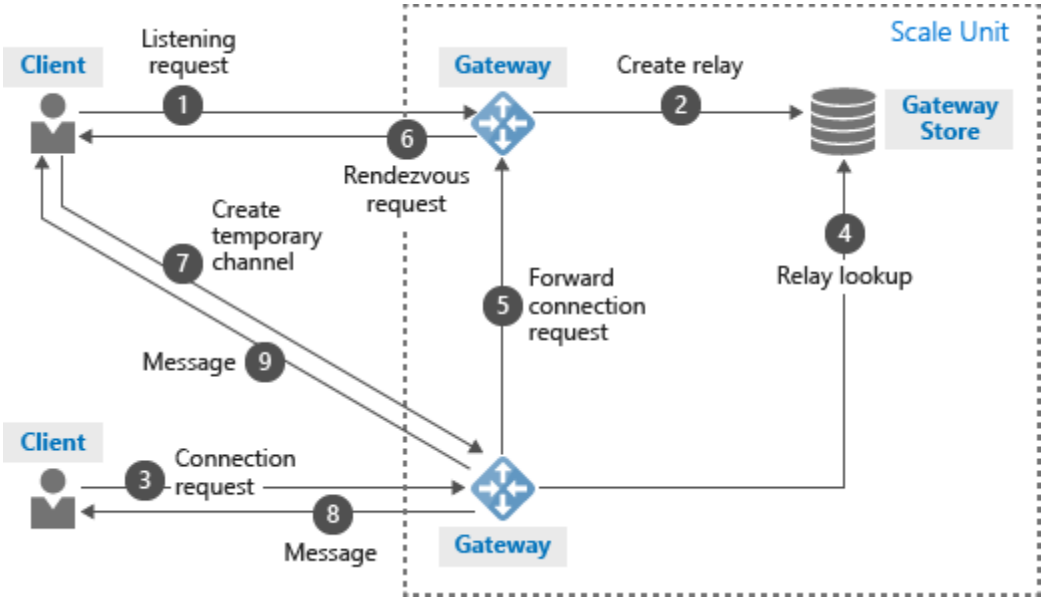
These two service capabilities (WCF Relay and Hybrid Connections) exist side-by-side in the Azure Relay service. They share a common gateway, but are otherwise different implementations.

Hybrid Connections and WCF Relay both enable secure connection to assets that exist within a corporate network. Use of one over the other depends on your particular needs, as described in the following table:

	WCF Relay	**Hybrid Connections**
WCF	x	
.NET Core		x
.NET Framework	x	x
Java script/Node.JS		x
Standards-Based open protocol		x
RPC programming models		x

The following diagram shows you how incoming relay requests are handled by the Azure Relay service:

Deccansoft Software Services H.No: 153, A/4, Balamrai, Rasoolpura, Secunderabad-500003 TELANGANA, NDIA. http://www.deccansoft.com | http://www.bestdotnettraining.com
Phone: +91 40 2784 1517 OR +91 8008327000 (INDIA)



1. Listening client sends a listening request to the Azure Relay service. The Azure load balancer routes the request to one of the gateway nodes.
2. The Azure Relay service creates a relay in the gateway store.
3. Sending client sends a request to connect to the listening service.
4. The gateway that receives the request looks up for the relay in the gateway store.
5. The gateway forwards the connection request to the right gateway mentioned in the gateway store.
6. The gateway sends a request to the listening client for it to create a temporary channel to the gateway node that's closest to the sending client.
7. The listening client creates a temporary channel to the gateway that's closest to the sending client. Now that the connection is established between clients via a gateway, the clients can exchange messages with each other.
8. The gateway forwards any messages from the listening client to the sending client.
9. The gateway forwards any messages from the sending client to the listening client.

- 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.

WCF Relays:

- 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 → **Relays** → +Add
   a. Service Bus → Create, Provide Namespace="dssdemoservicebus" → OK
   b. Select Azure Portal → Relay → Shared access policies→ 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
```


Deccansoft Software Services H.No: 153, A/4, Balamrai, Rasoolpura, Secunderabad-500003 TELANGANA, NDIA. http://www.deccansoft.com | http://www.bestdotnettraining.com
Phone: +91 40 2784 1517 OR +91 8008327000 (INDIA)

```
  {
    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="f24VlbPGurwIBKZN/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.

Deccansoft Software Services H.No: 153, A/4, Balamrai, Rasoolpura, Secunderabad-500003 TELANGANA, NDIA. http://www.deccansoft.com | http://www.bestdotnettraining.com
Phone: +91 40 2784 1517 OR +91 8008327000 (INDIA)

```
<system.ServiceModel>
<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>
</system.ServiceModel>
```

```
class Program
{
  static void Main(string[] args)
  {
    var factory = new ChannelFactory<IMathServiceChannel>("MathEndPoint");
    using (var ch = factory.CreateChannel())
    {
      Console.WriteLine(ch.Add(4, 5));
    }
  }
}
```


Deccansoft Software Services H.No: 153, A/4, Balamrai, Rasoolpura, Secunderabad-500003 TELANGANA, NDIA. http://www.deccansoft.com | http://www.bestdotnettraining.com
Phone: +91 40 2784 1517 OR +91 8008327000 (INDIA)

https://docs.microsoft.com/en-us/azure/azure-relay/relay-hybrid-connections-http-requests-dotnet-get-started

Hybid Connections

- It allows you to send requests and receive responses over **web sockets or HTTP(S).**
- You can use Hybrid Connections with any web sockets library for any runtime/language.

Unlike Service Bus messaging, Relay Hybrid Connections supports unauthorized or anonymous senders.

1. Create a namespace using Azure Portal
2. Get Management credentials to be used in Code: **Select Relay Namespace → Shared access policies**
3. Create a Hybrid Connection ("demo"): **Replay Namespace → Hybrid Connections**

4. Create Server Application(Listener)

Add Reference to NuGet Package **Microsoft.Azure.Relay**

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
using System.Net;
using Microsoft.Azure.Relay;

class Program
{
   private const string RelayNamespace = "dsssbdemo.servicebus.windows.net";
   private const string ConnectionName = "demo";
   private const string KeyName = "RootManageSharedAccessKey";
   private const string Key = "vYGVn35MgALoCmTXPgkDhQLhdk/wpu/xHYKcLiiyCnM=";
   static void Main(string[] args)
   {
      RunAsync().GetAwaiter().GetResult();
   }
   // The method initiates the connection.
   private static async Task RunAsync()
   {
      var cts = new CancellationTokenSource();
```

```
    var tokenProvider = TokenProvider.CreateSharedAccessSignatureTokenProvider(KeyName, Key);
    var listener = new HybridConnectionListener(new Uri(string.Format("sb://{0}/{1}", RelayNamespace,
ConnectionName)), tokenProvider);

    // Subscribe to the status events.
    listener.Connecting += (o, e) =>  { Console.WriteLine("Connecting"); };
    listener.Offline += (o, e) => { Console.WriteLine("Offline"); };
    listener.Online += (o, e) => { Console.WriteLine("Online"); };

    // Provide an HTTP request handler
    listener.RequestHandler = (context) =>
    {
       context.Response.StatusCode = HttpStatusCode.OK;
       context.Response.StatusDescription = "OK, This is pretty neat";
       StreamReader sr = new StreamReader(context.Request.InputStream);
       Console.WriteLine("Input: " + sr.ReadToEnd());
       using (var sw = new StreamWriter(context.Response.OutputStream))
       {
          sw.WriteLine("hello!");
       }
       context.Response.Close();
    };

    // Opening the listener establishes the control channel to
    // the Azure Relay service. The control channel is continuously
    // maintained, and is reestablished when connectivity is disrupted.
    await listener.OpenAsync();
    Console.WriteLine("Server listening");

    // Start a new thread that will continuously read the console.
    await Console.In.ReadLineAsync();
```

```
        // Close the listener after you exit the processing loop.
        await listener.CloseAsync();
    }
}
```

5. Create Client Application (sender):

```
using Microsoft.Azure.Relay;
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    private const string RelayNamespace = "dsssbdemo.servicebus.windows.net";
    private const string ConnectionName = "demo";
    private const string KeyName = "RootManageSharedAccessKey";
    private const string Key = "vYGVn35MgALoCmTXPgkDhQLhdk/wpu/xHYKcLiiyCnM=";

    static void Main(string[] args)
    {
        RunAsync().GetAwaiter().GetResult();
    }

    private static async Task RunAsync()
    {
        var tokenProvider = TokenProvider.CreateSharedAccessSignatureTokenProvider(
         KeyName, Key);
        var uri = new Uri(string.Format("https://{0}/{1}", RelayNamespace, ConnectionName));
        var token = (await tokenProvider.GetTokenAsync(uri.AbsoluteUri, TimeSpan.FromHours(1))).TokenString;
        var client = new HttpClient();
        var request = new HttpRequestMessage()
        {
            Content = new StringContent("This is demo", System.Text.Encoding.UTF8, "application/json"),
```

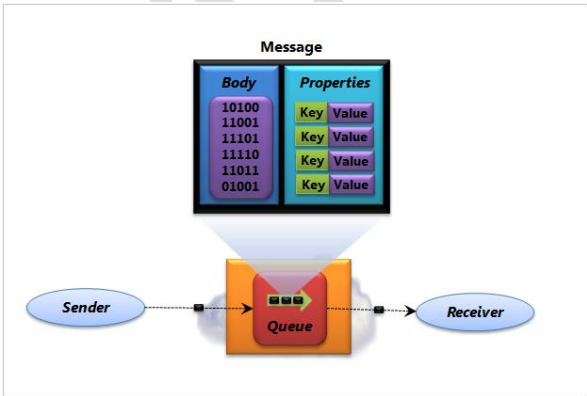
```
        RequestUri = uri,
        Method = HttpMethod.Get,
      };
      request.Headers.Add("ServiceBusAuthorization", token);
      var response = await client.SendAsync(request);
      Console.WriteLine(await response.Content.ReadAsStringAsync());
    }
}
```

Using WebSockets:

https://docs.microsoft.com/en-us/azure/azure-relay/relay-hybrid-connections-dotnet-get-started

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**.

This table provides a summary.

Comparison Criteria	Storage Queues	Service Bus Queues
Ordering (FIFO) guarantee	No	Yes (Sessions)
Delivery guarantee	At-Least-Once	At-Least-Once (PeekLock) At-Most-Once (ReceiveAndDelete)
Batch receive	Yes	Yes
Batch send	No	Yes
Scheduled delivery	Yes	Yes
Max TTL	7 days	1,06,75,199 days
Automatic dead lettering	No	Yes
Message auto-forwarding	No	Yes
Message groups	No	Yes
Duplicate detection	No	Yes
Transaction	No	Yes
Max Size of Message	64 KB	256 KB (Standard Pricing) 1024 KB (Premium Pricing)
Queue Size	500TB	80 GB (Premium Pricing) 5GB (Standard Pricing),

Walkthrough:

1. Create Service Bus namespace:
    a. Topics/subscriptions are not supported in the **Basic pricing** tier.
    b. The **Premium pricing tier** provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit.

Premium	Standard
High throughput	Variable throughput
Predictable performance	Variable latency
Fixed pricing	Pay as you go variable pricing
Ability to scale workload up and down	N/A
Message size up to 1 MB.	Message size up to 256 KB

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.

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.

Get Message Counters:

Counter	Description
ActiveMessageCount	Number of messages in the queue or subscription that are in the active state and ready for delivery.
ScheduledMessageCount	Number of messages in the scheduled state.
DeadLetterMessageCount	Number of messages in the dead-letter queue.
TransferMessageCount	Number of messages pending transfer into another queue or topic.
TransferDeadLetterMessageCount	Number of messages that failed to transfer into another queue or topic and have been moved into the transfer dead-letter queue.

Programming in C#

4. Create a two Console Application (Sender and Receiver)
5. Manage NuGet Package → Search **Azure.Messaging.ServiceBus**
6. Edit Program.cs

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
class Program
{
   static string connectionString = "<NAMESPACE CONNECTION STRING>";
   static string queueName = "<QUEUE NAME>";

   static async Task Main()
   {
      ServiceBusClient client = new ServiceBusClient(connectionString);
      ServiceBusSender sender = client.CreateSender(queueName);
      using ServiceBusMessageBatch messageBatch = await sender.CreateMessageBatchAsync();
      while (true)
      {
         Console.WriteLine("Enter Message (exit to terminate): ");
         string m = Console.ReadLine();
         if (m == "exit")
```

```
            break;
        var msg = new ServiceBusMessage(m);
        msg.ApplicationProperties.Add("Author", "sandeep");
        msg.ApplicationProperties.Add("CreatedAt", DateTime.Now);
        msg.ApplicationProperties.Add("Source", "DemoApp");

        msg.TimeToLive = new TimeSpan(0, 0, 5);
        msg.MessageId = msg.GetHashCode().ToString();
        await sender.SendMessageAsync(msg);
        Console.WriteLine("Sent...");
      }
      await sender.DisposeAsync();
      await client.DisposeAsync();
   }
}
```

//To send a group of messages, you can use the **sender.SendMessagesAsync** method with list of Message object

Receiver Application:

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;

namespace QueueReceiver
{
   class Program
   {
      static string connectionString =
"Endpoint=sb://dssdemo.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedA
ccessKey=51k0iCoL2iUpiDIFATNdhF0tKCNEcY1VkiGT0t3H6As=";
      static string queueName = "demo-queue";
      static ServiceBusClient client;
      static ServiceBusProcessor processor;
      static async Task MessageHandler(ProcessMessageEventArgs args)
```

```
    {
        string body = args.Message.Body.ToString();
        Console.WriteLine($"Received: {body}, Count: {args.Message.DeliveryCount}");
        await args.CompleteMessageAsync(args.Message);
    }
    static Task ErrorHandler(ProcessErrorEventArgs args)
    {
        Console.WriteLine(args.Exception.ToString());
        return Task.CompletedTask;
    }

    static async Task Main()
    {
        client = new ServiceBusClient(connectionString);
        processor = client.CreateProcessor(queueName, new ServiceBusProcessorOptions() {
ReceiveMode=ServiceBusReceiveMode.PeekLock, AutoCompleteMessages=false});

        processor.ProcessMessageAsync += MessageHandler;
        processor.ProcessErrorAsync += ErrorHandler;
        await processor.StartProcessingAsync();

        Console.WriteLine("Wait for a minute and then press any key to end the processing");
        Console.ReadKey();

        // stop processing
        Console.WriteLine("\nStopping the receiver...");
        await processor.StopProcessingAsync();
        Console.WriteLine("Stopped receiving messages");

        await processor.DisposeAsync();
        await client.DisposeAsync();
    }
}
```

```
}
```

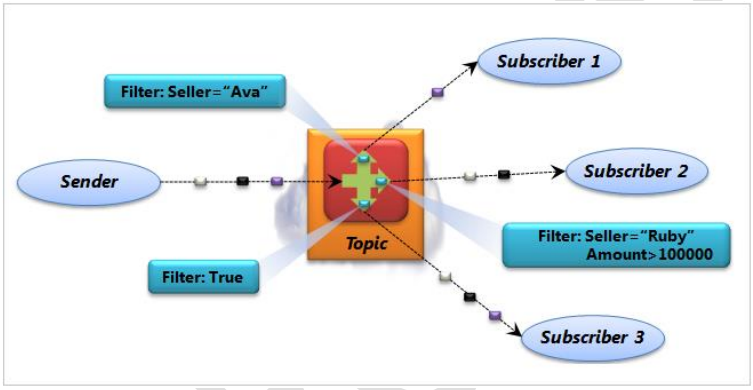
Note: You will notice that we need to call the **CompleteMessageAsync** method of the queue client at the end of the message handler method. This ensures that the message is not received again. Alternatively, you can call **AbandonAsync** if you wish to stop handling the message and receive it again.

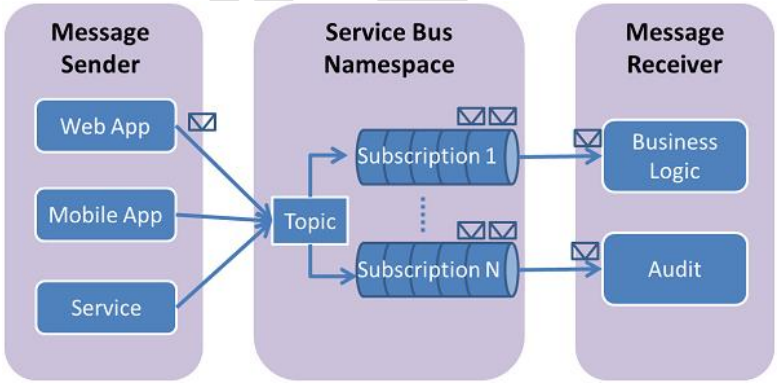
Topics and Subscriptions

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.

Receiver App will receive messages from Subscription and not directly from Topic.

From every subscription (**queue)**, the message will be received by only one application at any point of time (even if multiple application instances are running).

For message to be handled by multiple applications, create a separate subbsciber for each application.

Sample to Create Queue, Topic and Subscriptions Programmatically:

https://github.com/Azure-Samples/service-bus-dotnet-management/blob/a55185bef30d1763c1a8182a3361dbb548bad436/src/service-bus-dotnet-management/ServiceBusManagementSample.cs

About Filters:

https://docs.microsoft.com/en-us/azure/service-bus-messaging/topic-filters

Publisher Console Application: (.NET Core Console Application)

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;

class ProgramToSentInTopic
{
   static string connectionString =
"Endpoint=sb://dssdemo.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedA
ccessKey=t0P0CtA3PhEZLmzsEHSFijxFmXH4Uk8JYXhbMqoK+JM=";
   static string topicName = "courses";
   static ServiceBusClient client;
   static ServiceBusSender sender;
   static async Task Main()
   {
      client = new ServiceBusClient(connectionString);
      sender = client.CreateSender(topicName);
      while (true)
      {
         Console.WriteLine("Enter Message (exit to terminate): ");
```

```
        string m = Console.ReadLine();
        if (m == "exit")
            break;
        var msg = new ServiceBusMessage(m);
        Console.WriteLine("Enter Author of message: ");
        string author = Console.ReadLine();
        msg.ApplicationProperties.Add("Author", author);
        msg.ApplicationProperties.Add("CreatedAt", DateTime.Now);
        msg.ApplicationProperties.Add("Source", "DemoApp");

        msg.TimeToLive = new TimeSpan(0, 0, 5);
        msg.MessageId = msg.GetHashCode().ToString();
        await sender.SendMessageAsync(msg);
        Console.WriteLine("Sent...");
      }
      await sender.DisposeAsync();
      await client.DisposeAsync();
   }
}
```

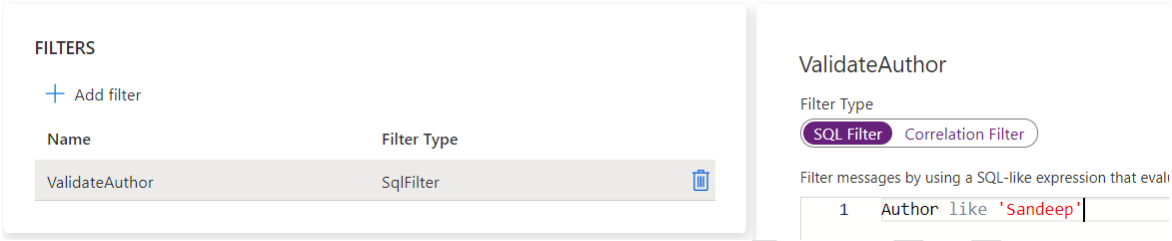
All Author Subscriber Console Application (.NET Core)

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;

class Program
{
   static string connectionString = "<NAMESPACE CONNECTION STRING>";
   static string topicName = "<SERVICE BUS TOPIC NAME>";
   static string subscriptionName = "<SERVICE BUS - TOPIC SUBSCRIPTION NAME>"; //AllSubscription or
SandeepSubscription
   static ServiceBusClient client;
   static ServiceBusProcessor processor;
```

```
static async Task MessageHandler(ProcessMessageEventArgs args)
{
   string body = args.Message.Body.ToString();
   Console.WriteLine($"Received: {body} from subscription: {subscriptionName}");
   await args.CompleteMessageAsync(args.Message);
}
static Task ErrorHandler(ProcessErrorEventArgs args)
{
   Console.WriteLine(args.Exception.ToString());
   return Task.CompletedTask;
}

static async Task Main()
{
   client = new ServiceBusClient(connectionString);
   var options = new ServiceBusProcessorOptions();
   options.ReceiveMode = ServiceBusReceiveMode.PeekLock;
   processor = client.CreateProcessor(topicName, subscriptionName, options);
   try
   {
      processor.ProcessMessageAsync += MessageHandler;
      processor.ProcessErrorAsync += ErrorHandler;
      await processor.StartProcessingAsync();
      Console.WriteLine("Wait for a minute and then press any key to end the processing");
      Console.ReadKey();
      Console.WriteLine("\nStopping the receiver...");
      await processor.StopProcessingAsync();
      Console.WriteLine("Stopped receiving messages");
   }
   finally
   {
      await processor.DisposeAsync();
```

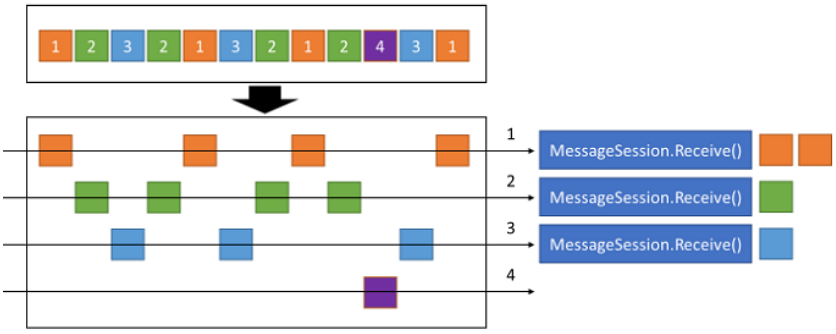
```
        await client.DisposeAsync();
    }
  }
}
```

AuthorSandeep Subscription (Author = Sandeep)



Session Service Bus

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.



To send messages in a session.

```
message.SessionId = "abcd";
```

To Receive Messages:

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
```

```
class Program
{
   static string connectionString = "<NAMESPACE CONNECTION STRING>";
   static string queueName = "<QUEUE NAME>";
   static ServiceBusClient client;
   static ServiceBusSessionProcessor processor;
   static async Task MessageHandler(ProcessSessionMessageEventArgs args)
   {
      string body = args.Message.Body.ToString();
      Console.WriteLine($"Received: {body}");
      await args.CompleteMessageAsync(args.Message);
   }
   static Task ErrorHandler(ProcessErrorEventArgs args)
   {
      Console.WriteLine(args.Exception.ToString());
      return Task.CompletedTask;
   }

   static async Task Main()
   {
      client = new ServiceBusClient(connectionString);
      var options = new ServiceBusSessionProcessorOptions()
      {
         SessionIds = { "abcd" }
      };

      processor = client.CreateSessionProcessor(queueName, options);

      processor.ProcessMessageAsync += MessageHandler;
      processor.ProcessErrorAsync += ErrorHandler;
      await processor.StartProcessingAsync();

      Console.WriteLine("Wait for a minute and then press any key to end the processing");
```

```
    Console.ReadKey();

    // stop processing
    Console.WriteLine("\nStopping the receiver...");
    await processor.StopProcessingAsync();
    Console.WriteLine("Stopped receiving messages");

    await processor.DisposeAsync();
    await client.DisposeAsync();
  }
}
```

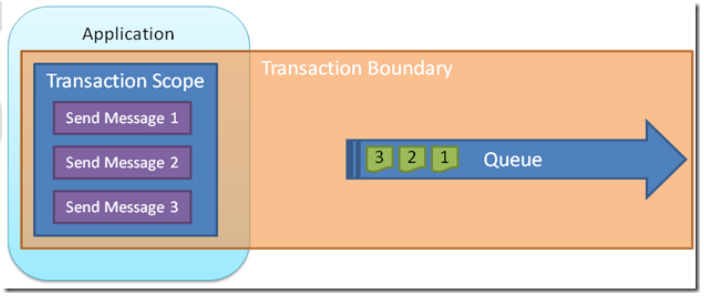
Transactions in Brokered Messaging

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.

Messages that are sent as part of a transaction must specify a partition key. The key can be one of the following properties: session ID, partition key, or message ID. **All messages in a given transaction must be in same partition key.**

Note: Partition Key in order of Priority

1. SessionId
2. PartitionKey
3. MessageId

To the project add reference to System.Transactions.

```
using Microsoft.Azure.ServiceBus;
using Microsoft.Azure.ServiceBus.Management;
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Transactions;

class Program
{
   static void Main(string[] args)
   {
      var queueName = "tranqueue";
      var connectionString = "Endpoint=sb://dssdemo.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=fXbUbJTu7hhrQO93ZslGSgV+L0l9PA3bxBV4ZOtbIpg=";
       var queueClient = new QueueClient(connectionString, queueName);
      using (TransactionScope scope = new TransactionScope())
      {
         for (int i = 0; i < 10; i++)
         {
            // Send a message
            Message msg = new Message(System.Text.Encoding.UTF8.GetBytes("Message" + i));
            msg.PartitionKey = "demopartition";
            queueClient.SendAsync(msg).Wait();
            Console.Write(".");
```

```
        }
        Console.WriteLine("Done!");
        Console.WriteLine();

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

Receive is not part of transaction. Only operations which actually do something with the message on the broker are part of the transaction.

These are: Send, Complete, Deadletter, Defer. Receive itself already utilizes the peeklock concept on the broker.

Note: Partitioned queues and topics aren't supported in the Premium messaging tier. Sessions are supported in the premier tier by using SessionId.

Service Bus client libraries that are used for operations like send and receive messages can also be used to manage queues, topics, subscriptions, and rules in an existing Service Bus namespace. Use **ServiceBusAdministrationClient** class.