What is Azure Service Bus?

Azure Service Bus is a fully managed enterprise message broker with message queues and publish-subscribe topics.
Service Bus is used to decouple applications and services from each other, providing the following benefits:

- Load-balancing work across competing workers
- Safely routing and transferring data and control across service and application boundaries
- Coordinating transactional work that requires a high degree of reliability

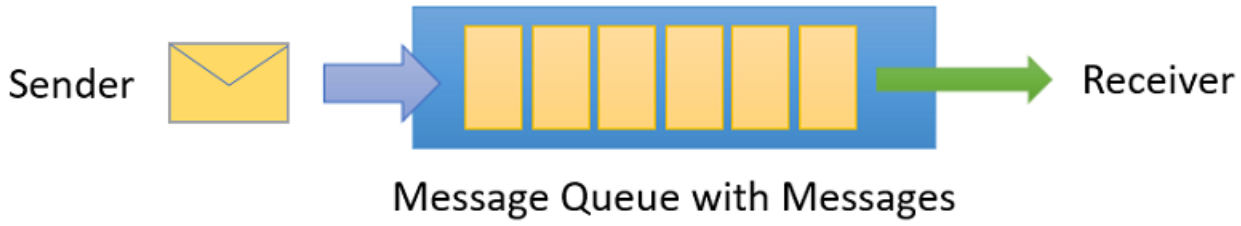
Service Bus queues, topics, and subscriptions

Azure Service Bus supports reliable message queuing and durable publish/subscribe messaging. The messaging entities that form the core of the messaging capabilities in Service Bus are queues, topics and subscriptions.

Queues

Queues offer **First In, First Out** (FIFO) message delivery to one or more competing consumers.

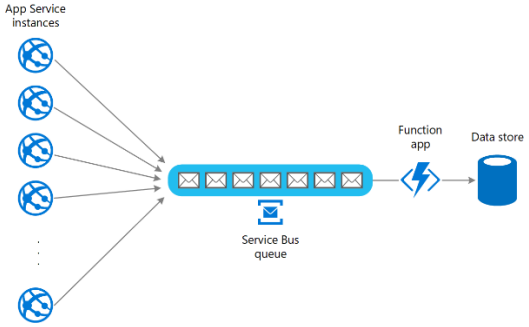
That is, receivers typically receive and process messages in the order in which they were added to the queue. And, only one message consumer receives and processes each message.



A key benefit of using queues is to achieve **temporal decoupling of application components**.

In other words, the producers (senders) and consumers (receivers) don't have to send and receive messages at the same time, because messages are stored durably in the queue. Furthermore, the producer doesn't have to wait for a reply from the consumer to continue to process and send messages.

A related benefit is **load-leveling**, which enables producers and consumers to send and receive messages at different rates. In many applications, the system load varies over time.



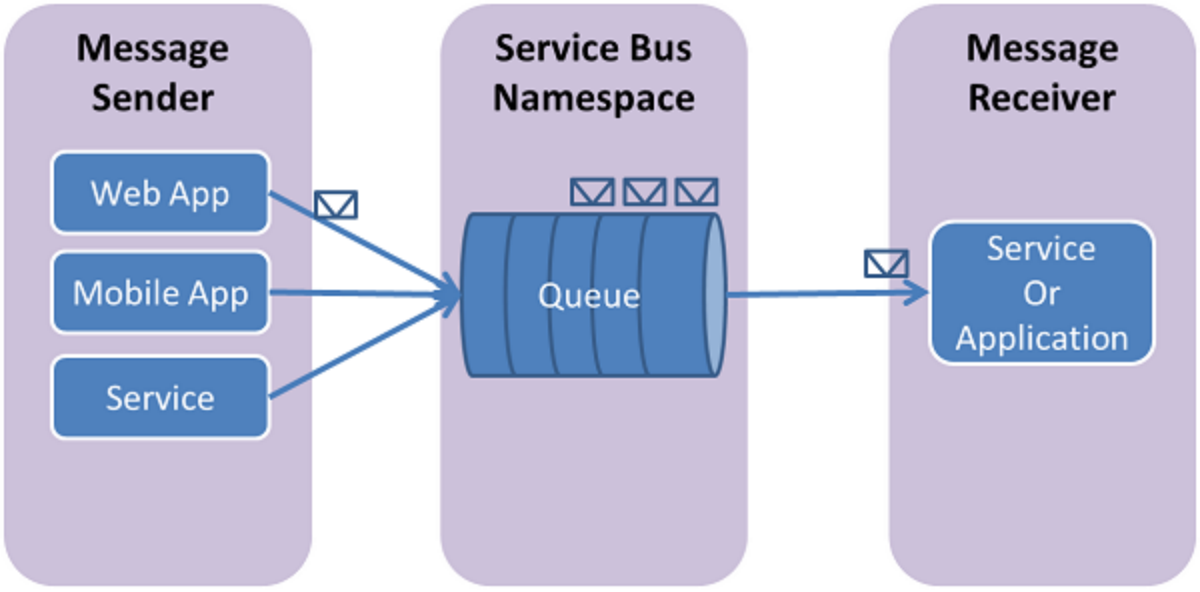
However, the processing time required for each unit of work is typically constant. Intermediating message producers and consumers with a queue means that the consuming application only must be able to handle average load instead of peak load.

The depth of the queue grows and contracts as the incoming load varies. This capability directly saves money regarding the amount of infrastructure required to service the application load. As the load increases, more worker processes can be added to read from the queue. Each message is processed by only one of the worker processes.

Furthermore, this pull-based load balancing allows for best use of the worker computers even if the worker computers with processing power pull messages at their own maximum rate. This pattern is often termed the **competing consumer** pattern.

Using queues to intermediate between message producers and consumers provides an inherent loose coupling between the components. Because producers and consumers aren't aware of each other, a consumer can be **upgraded** without having any effect on the producer.

Create queues



Service Bus queues are a general-purpose technology that can be used for a wide variety of scenarios:

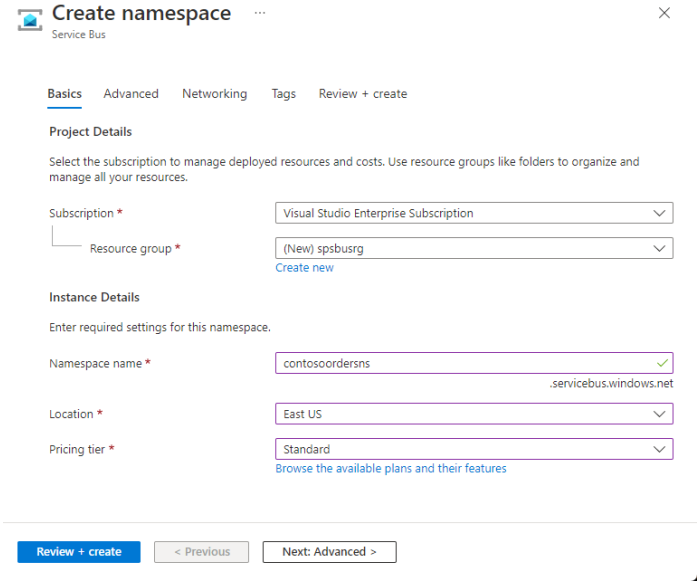
- Communication between web and worker roles in a multi-tier Azure application.
- Communication between on-premises apps and Azure-hosted apps in a hybrid solution.
- Communication between components of a distributed application running on-premises in different organizations or departments of an organization.

Create a namespace in the Azure portal

A namespace is a container for all messaging components (queues and topics). A namespace can have one or more queues and topics and it often serves as an application container.

A namespace can be compared to a server in the terminology of other brokers, but the concepts aren't directly equivalent. A Service Bus namespace is your own capacity slice of a large cluster made up of dozens of all-active virtual machines. It optionally spans three Azure availability zones. So, you get all the availability and robustness benefits of running the message broker at enormous scale. And, you don't need to worry about underlying complexities. Service Bus is serverless messaging.

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for Service Bus resources (queues, topics, etc.) within your application.



Create a Service Bus messaging namespace.

```
az servicebus namespace create --resource-group ContosoRG --name ContosoSBusNS --location eastus
```

create a queue in the namespace you created in the previous step.

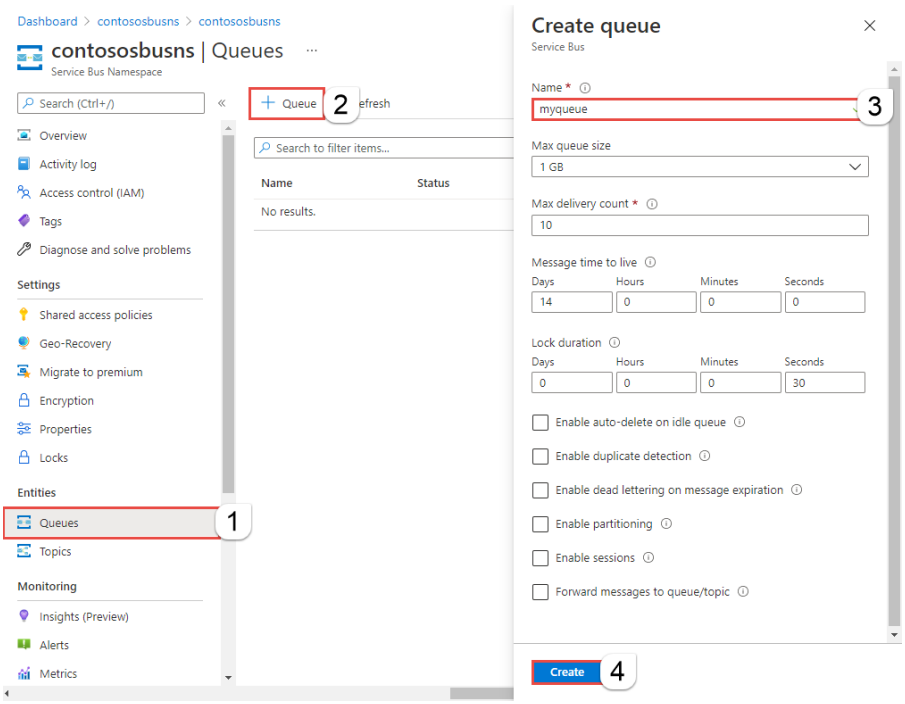
```
az servicebus queue create --resource-group ContosoRG --namespace-name ContosoSBusNS --name ContosoOrdersQueue
```

To get the primary connection string for the namespace.

```
az servicebus namespace authorization-rule keys list --resource-group ContosoRG --namespace-name ContosoSBusNS --name RootManageSharedAccessKey --query primaryConnectionString --output tsv
```

Create a queue in the Azure portal

1. On the Service Bus Namespace page, select Queues in the left navigational menu.
2. On the Queues page, select + Queue on the toolbar.
3. Enter a name for the queue and leave the other values with their defaults.
4. Now, select Create.



Then, send and receive messages using clients written in programming languages.

Authenticate the app to Azure

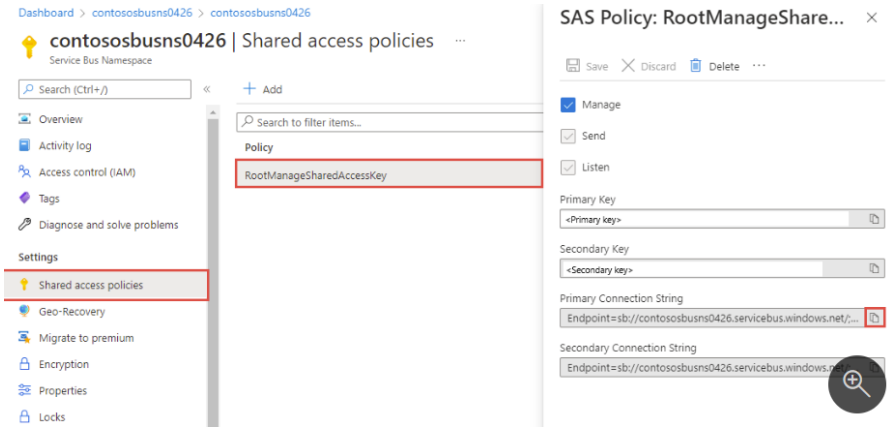
Two ways of connecting to Azure Service Bus: passwordless and connection string.

The first option shows you how to use your security principal in Microsoft Entra ID and role-based access control (RBAC) to connect to a Service Bus namespace.

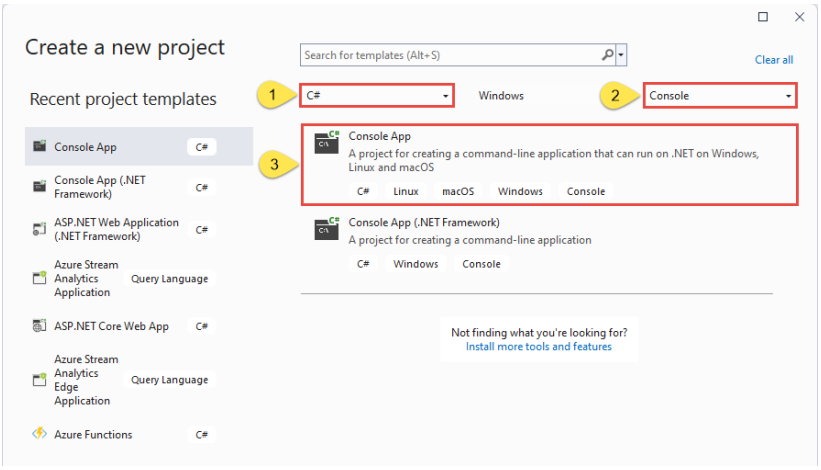
1. Assign roles to your Microsoft Entra user
2. Azure built-in roles for Azure Service Bus
3. Add Microsoft Entra user to Azure Service Bus Owner role

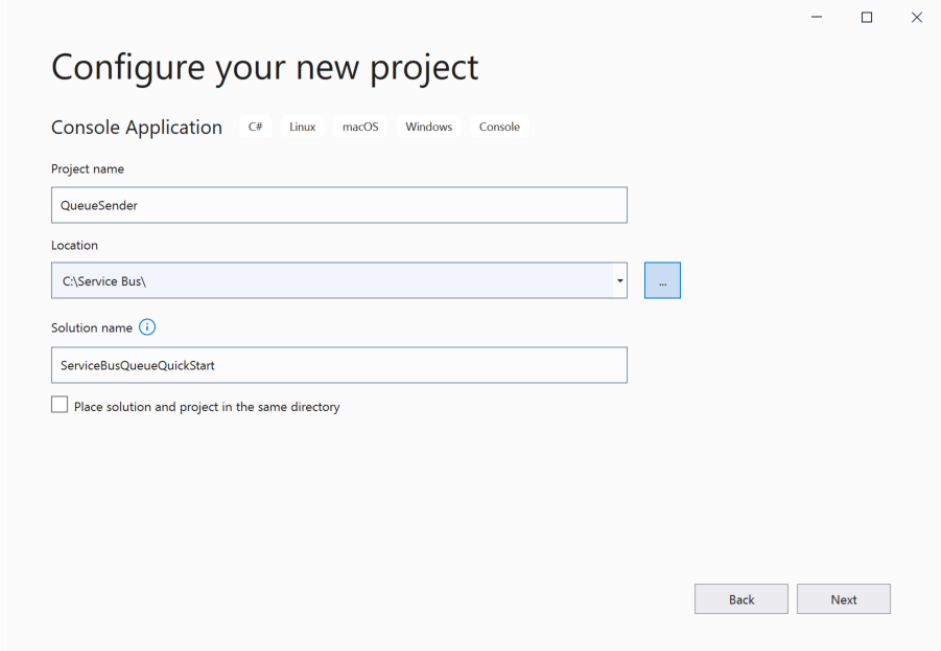
The second option shows you how to use a connection string to connect to a Service Bus namespace.

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) policy with primary and secondary keys, and primary and secondary connection strings that each grant full control over all aspects of the namespace.



Send messages to the queue





Add the NuGet packages to the project

1. Passwordless
   Install-Package Azure.Messaging.ServiceBus
   Install-Package Azure.Identity
2. Connection String
   Install-Package Azure.Identity

Add code to send messages to the queue

- Creates a ServiceBusClient object using the DefaultAzureCredential object. DefaultAzureCredential automatically discovers and uses the credentials of your Visual Studio sign-in to authenticate to Azure Service Bus.

- Invokes the CreateSender method on the ServiceBusClient object to create a ServiceBusSender object for the specific Service Bus queue.

- Creates a ServiceBusMessageBatch object by using the ServiceBusSender.CreateMessageBatchAsync method.

- Add messages to the batch using the ServiceBusMessageBatch.TryAddMessage.

- Sends the batch of messages to the Service Bus queue using the ServiceBusSender.SendMessagesAsync method.

Receive messages from the queue

- Creates a ServiceBusClient object using the DefaultAzureCredential object. DefaultAzureCredential automatically discovers and uses the credentials of your Visual Studio sign in to authenticate to Azure Service Bus.
- Invokes the CreateProcessor method on the ServiceBusClient object to create a ServiceBusProcessor object for the specified Service Bus queue.
- Specifies handlers for the ProcessMessageAsync and ProcessErrorAsync events of the ServiceBusProcessor object.
- Starts processing messages by invoking the StartProcessingAsync on the ServiceBusProcessor object.
- When user presses a key to end the processing, invokes the StopProcessingAsync on the ServiceBusProcessor object.

Receive modes

You can specify two different modes in which consumers can receive messages from Service Bus.

Receive and delete. In this mode, when Service Bus receives the request from the consumer, it marks the message as being consumed and returns it to the consumer application. This mode is the simplest model. It works best for scenarios in which the application can tolerate not processing a message if a failure occurs. To understand this scenario, consider a scenario in which the consumer

issues the receive request and then crashes before processing it. As Service Bus marks the message as consumed, the application begins consuming messages upon restart. It will miss the message that it consumed before the crash. This process is often called at-most once processing.

Peek lock. In this mode, the receive operation becomes two-stage, which makes it possible to support applications that can't tolerate missing messages.
Finds the next message to be consumed, locks it to prevent other consumers from receiving it, and then, return the message to the application.

After the application finishes processing the message, it requests the Service Bus service to complete the second stage of the receive process. Then, the service marks the message as consumed.

If the application is unable to process the message for some reason, it can request the Service Bus service to abandon the message. Service Bus unlocks the message and makes it available to be received again, either by the same consumer or by another competing consumer. Secondly, there's a timeout associated with the lock. If the application fails to process the message before the lock timeout expires, Service Bus unlocks the message and makes it available to be received again.

If the application crashes after it processes the message, but before it requests the Service Bus service to complete the message, Service Bus redelivers the message to the application when it restarts. This process is often called at-least once processing. That is, each message is processed at least once. However, in certain situations the same message might be redelivered. If your scenario can't tolerate duplicate processing, add extra logic in your application to detect duplicates. For more information, see Duplicate detection, which is known as exactly once processing.

Consider using Service Bus queues

As a solution architect/developer, you should consider using Service Bus queues when:

- Your solution needs to receive messages without having to poll the queue. With Service Bus, you can achieve it by using a long-polling receive operation using the TCP-based protocols that Service Bus supports.
- Your solution requires the queue to provide a guaranteed first-in-first-out (FIFO) ordered delivery.
- Your solution needs to support automatic duplicate detection.
- You want your application to process messages as parallel long-running streams (messages are associated with a stream using the **session ID** property on the message). In this model, each node in the consuming application competes for streams, as opposed to messages. When a stream is given to a consuming node, the node can examine the state of the application stream state using transactions.
- Your solution requires transactional behavior and atomicity when sending or receiving multiple messages from a queue.
- Your application handles messages that can exceed 64 KB but won't likely approach the 256 KB or 1 MB limit, depending on the chosen service tier (although Service Bus queues can handle messages up to 100 MB).

- You deal with a requirement to provide a role-based access model to the queues, and different rights/permissions for senders and receivers. For more information, see the following articles:
    - Authenticate with managed identities
    - Authenticate from an application
- Your queue size won't grow larger than 80 GB.
- You want to use the AMQP 1.0 standards-based messaging protocol. For more information about AMQP, see Service Bus AMQP Overview.
- You envision an eventual migration from queue-based point-to-point communication to a publish-subscribe messaging pattern. This pattern enables integration of additional receivers (subscribers). Each receiver receives independent copies of either some or all messages sent to the queue.
- Your messaging solution needs to support the "At-Most-Once" and the "At-Least-Once" delivery guarantees without the need for you to build the additional infrastructure components.
- Your solution needs to publish and consume batches of messages.

Compare Storage queues and Service Bus queues

The tables in the following sections provide a logical grouping of queue features. They let you compare, at a glance, the capabilities available in both Azure Storage queues and Service Bus queues.

Foundational capabilities

This section compares some of the fundamental queuing capabilities provided by Storage queues and Service Bus queues.

Comparison Criteria	Storage queues	Service Bus queues
Ordering guarantee	No For more information, see the first note in the Additional Information section.	Yes - First-In-First-Out (FIFO) (by using message sessions)
Delivery guarantee	At-Least-Once	At-Least-Once (using PeekLock receive mode. It's the default) At-Most-Once (using ReceiveAndDelete receive mode) Learn more about various Receive modes
Atomic operation support	No	Yes
Receive behavior	Non-blocking (completes immediately if no new message is found)	Blocking with or without a timeout (offers long polling, or the "Comet technique") Non-blocking (using .NET managed API only)

Push-style API	No	Yes Our .NET, Java, JavaScript, and Go SDKs provide push-style API.
Receive mode	Peek & Lease	Peek & Lock Receive & Delete
Exclusive access mode	Lease-based	Lock-based
Lease/Lock duration	30 seconds (default) 7 days (maximum) (You can renew or release a message lease using the UpdateMessage API.)	30 seconds (default) You can renew the message lock for the same lock duration each time manually or use the automatic lock renewal feature where the client manages lock renewal for you.
Lease/Lock precision	Message level Each message can have a different timeout value, which you can then update as needed while processing the message, by using the UpdateMessage API.	Queue level (each queue has a lock precision applied to all of its messages, but the lock can be renewed as described in the previous row)
Batched receive	Yes (explicitly specifying message count when retrieving messages, up to a maximum of 32 messages)	Yes (implicitly enabling a pre-fetch property or explicitly by using transactions)
Batched send	No	Yes (by using transactions or client-side batching)

Messages, payloads, and serialization

The equivalent names used at the Advanced Message Queuing Protocol (AMQP) protocol level are listed in parentheses. While the following names use pascal casing, JavaScript and Python clients would use camel and snake casing respectively.

Property Name	Description
ContentType (content-type)	Optionally describes the payload of the message, with a descriptor following the format of RFC2045, Section 5; for example, application/json.
CorrelationId (correlation-id)	Enables an application to specify a context for the message for the purposes of correlation; for example, reflecting the **MessageId** of a message that is being replied to.

DeadLetterSource	Only set in messages that have been dead-lettered and later autoforwarded from the dead-letter queue to another entity. Indicates the entity in which the message was dead-lettered. This property is read-only.
DeliveryCount	Number of deliveries that have been attempted for this message. The count is incremented when a message lock expires, or the message is explicitly abandoned by the receiver. This property is read-only. The delivery count isn't incremented when the underlying AMQP connection is closed.
EnqueuedSequenceNumber	For messages that have been autoforwarded, this property reflects the sequence number that had first been assigned to the message at its original point of submission. This property is read-only.
EnqueuedTimeUtc	The UTC instant at which the message has been accepted and stored in the entity. This value can be used as an authoritative and neutral arrival time indicator when the receiver doesn't want to trust the sender's clock. This property is read-only.
ExpiresAtUtc (absolute-expiry-time)	The UTC instant at which the message is marked for removal and no longer available for retrieval from the entity because of its expiration. Expiry is controlled by the **TimeToLive** property and this property is computed from EnqueuedTimeUtc+TimeToLive. This property is read-only.
Label or Subject (subject)	This property enables the application to indicate the purpose of the message to the receiver in a standardized fashion, similar to an email subject line.
LockedUntilUtc	For messages retrieved under a lock (peek-lock receive mode, not presettled) this property reflects the UTC instant until which the message is held locked in the queue/subscription. When the lock expires, the DeliveryCount is incremented and the message is again available for retrieval. This property is read-only.
LockToken	The lock token is a reference to the lock that is being held by the broker in *peek-lock* receive mode. The token can be used to pin the lock permanently through the Deferral API and, with that, take the message out of the regular delivery state flow. This property is read-only.
MessageId (message-id)	The message identifier is an application-defined value that uniquely identifies the message and its payload. The identifier is a free-form string and can reflect a GUID or an identifier derived from the application context. If enabled, the duplicate detection feature identifies and removes second and further submissions of messages with the same **MessageId**.
PartitionKey	For partitioned entities, setting this value enables assigning related messages to the same internal partition, so that submission sequence order is correctly recorded. The partition is chosen by a hash function over this value and can't be chosen directly. For session-aware entities, the **SessionId** property overrides this value.
ReplyTo (reply-to)	This optional and application-defined value is a standard way to express a reply path to the receiver of the message. When a sender expects a

	reply, it sets the value to the absolute or relative path of the queue or topic it expects the reply to be sent to.
ReplyToSessionId (reply-to-group-id)	This value augments the **ReplyTo** information and specifies which **SessionId** should be set for the reply when sent to the reply entity.
ScheduledEnqueueTimeUtc	For messages that are only made available for retrieval after a delay, this property defines the UTC instant at which the message will be logically enqueued, sequenced, and therefore made available for retrieval.
SequenceNumber	The sequence number is a unique 64-bit integer assigned to a message as it is accepted and stored by the broker and functions as its true identifier. For partitioned entities, the topmost 16 bits reflect the partition identifier. Sequence numbers monotonically increase and are gapless. They roll over to 0 when the 48-64 bit range is exhausted. This property is read-only.
SessionId (group-id)	For session-aware entities, this application-defined value specifies the session affiliation of the message. Messages with the same session identifier are subject to summary locking and enable exact in-order processing and demultiplexing. For entities that aren't session-aware, this value is ignored.
TimeToLive	This value is the relative duration after which the message expires, starting from the instant it has been accepted and stored by the broker, as captured in **EnqueueTimeUtc**. When not set explicitly, the assumed value is the **DefaultTimeToLive** for the respective queue or topic. A message-level **TimeToLive** value can't be longer than the entity's **DefaultTimeToLive** setting. If it's longer, it's silently adjusted.
To (to)	This property is reserved for future use in routing scenarios and currently ignored by the broker itself. Applications can use this value in rule-driven autoforward chaining scenarios to indicate the intended logical destination of the message.
ViaPartitionKey	If a message is sent via a transfer queue in the scope of a transaction, this value selects the transfer queue partition.

The abstract message model enables a message to be posted to a queue via HTTPS and can be retrieved via AMQP. In either case, the message looks normal in the context of the respective protocol. The broker properties are translated as needed, and the user properties are mapped to the most appropriate location on the respective protocol message model. In HTTP, user properties map directly to and from HTTP headers; in AMQP they map to and from the application-properties map.

Topic filters and actions

Subscribers can define which messages they want to receive from a topic. These messages are specified in the form of one or more named subscription rules. Each rule consists of a **filter condition** that selects particular messages, and **optionally** contain an **action** that annotates the selected message.
All rules **without actions** are combined using an OR condition and result in a **single message** on the subscription even if you have multiple matching rules.

Each rule **with an action** produces a copy of the message. This message will have a property called RuleName where the value is the name of the matching rule. The action can add or update properties, or delete properties from the original message to produce a message on the subscription. Consider the following scenario where a subscription has five rules: two rules with actions and the other three without actions. In this example, if you send one message that matches all five rules, you get three messages on the subscription. That's two messages for two rules with actions and one message for three rules without actions.

Each newly created topic subscription has an initial default subscription rule. If you don't explicitly specify a filter condition for the rule, the applied filter is the **true** filter that enables all messages to be selected into the subscription. The default rule has no associated annotation action.

Note

This article applies to non-JMS scenarios. For JMS scenarios, use **message selectors**.

Filters

Service Bus supports three types of filters:

- SQL filters
- Boolean filters
- Correlation filters

The following sections provide details about these filters.

SQL filters

A **SqlFilter** holds a SQL-like conditional expression that is evaluated in the broker against the arriving messages' user-defined properties and system properties. All system properties must be prefixed with sys. in the conditional expression. The SQL-language subset for filter conditions tests for the existence of properties (EXISTS), null-values (IS NULL), logical NOT/AND/OR, relational operators, simple numeric arithmetic, and simple text pattern matching with LIKE.

Here's a .NET example for defining a SQL filter:

Boolean filters

The **TrueFilter** and **FalseFilter** either cause all arriving messages (**true**) or none of the arriving messages (**false**) to be selected for the subscription. These two filters derive from the SQL filter.

Here's a .NET example for defining a boolean filter:

Correlation filters

A **CorrelationFilter** holds a set of conditions that are matched against one or more of an arriving message's user and system properties. A common use is to match against the **CorrelationId** property, but the application can also choose to match against the following properties:

- ContentType
- Label
- MessageId
- ReplyTo
- ReplyToSessionId
- SessionId
- To
- any user-defined properties.

A match exists when an arriving message's value for a property is equal to the value specified in the correlation filter. For string expressions, the comparison is case-sensitive. If you specify multiple match

properties, the filter combines them as a logical AND condition, meaning for the filter to match, all conditions must match.