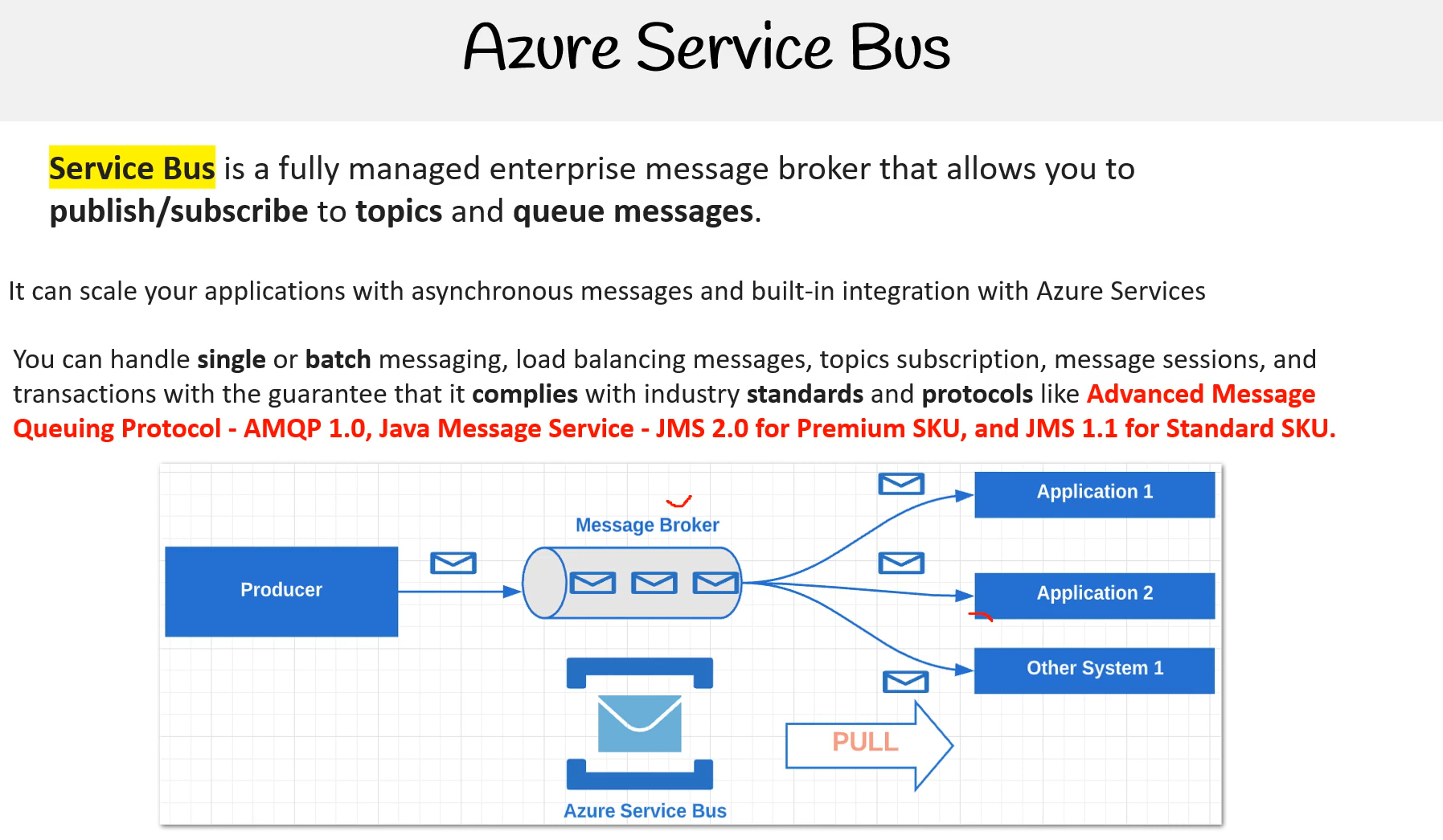
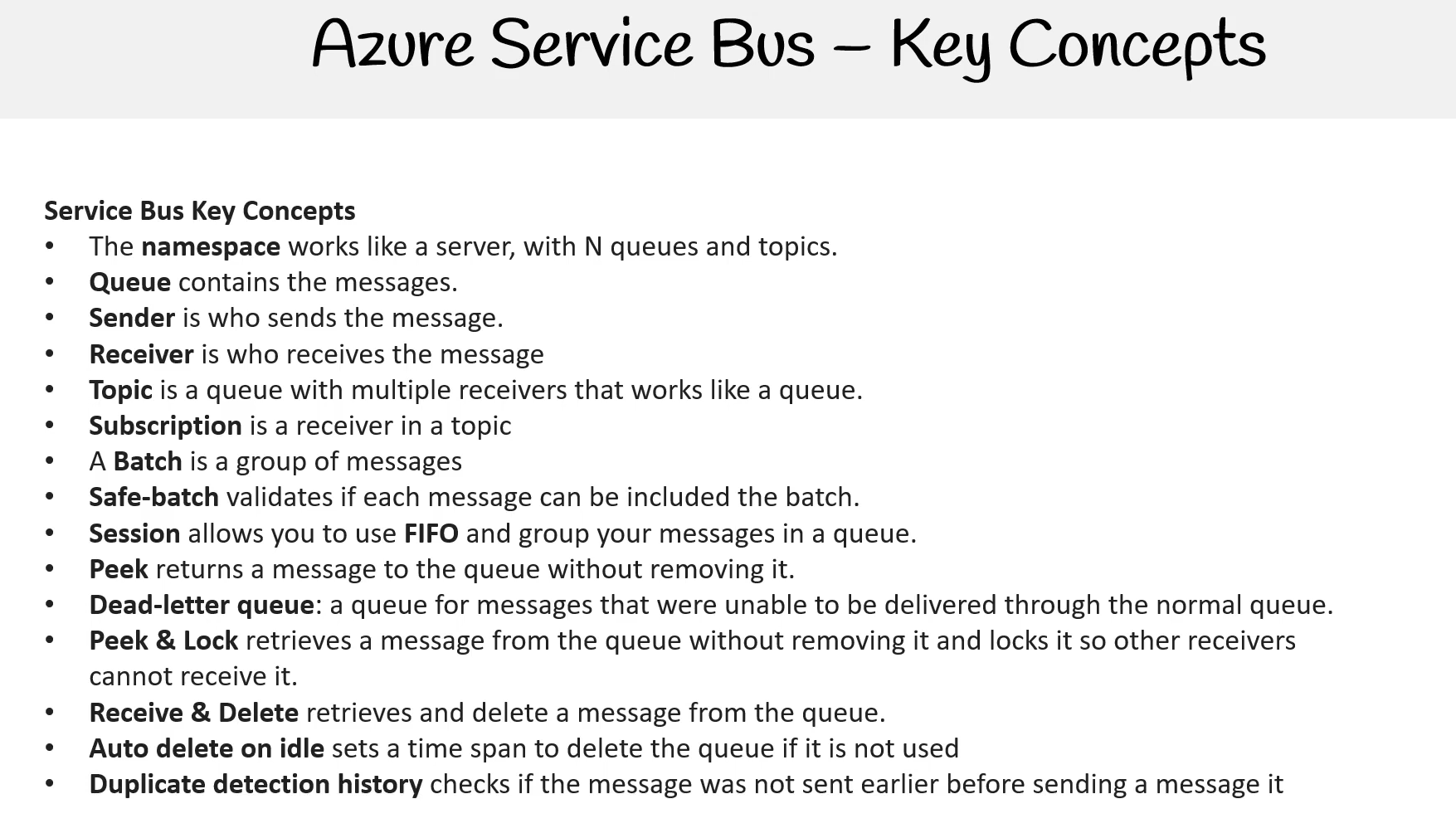


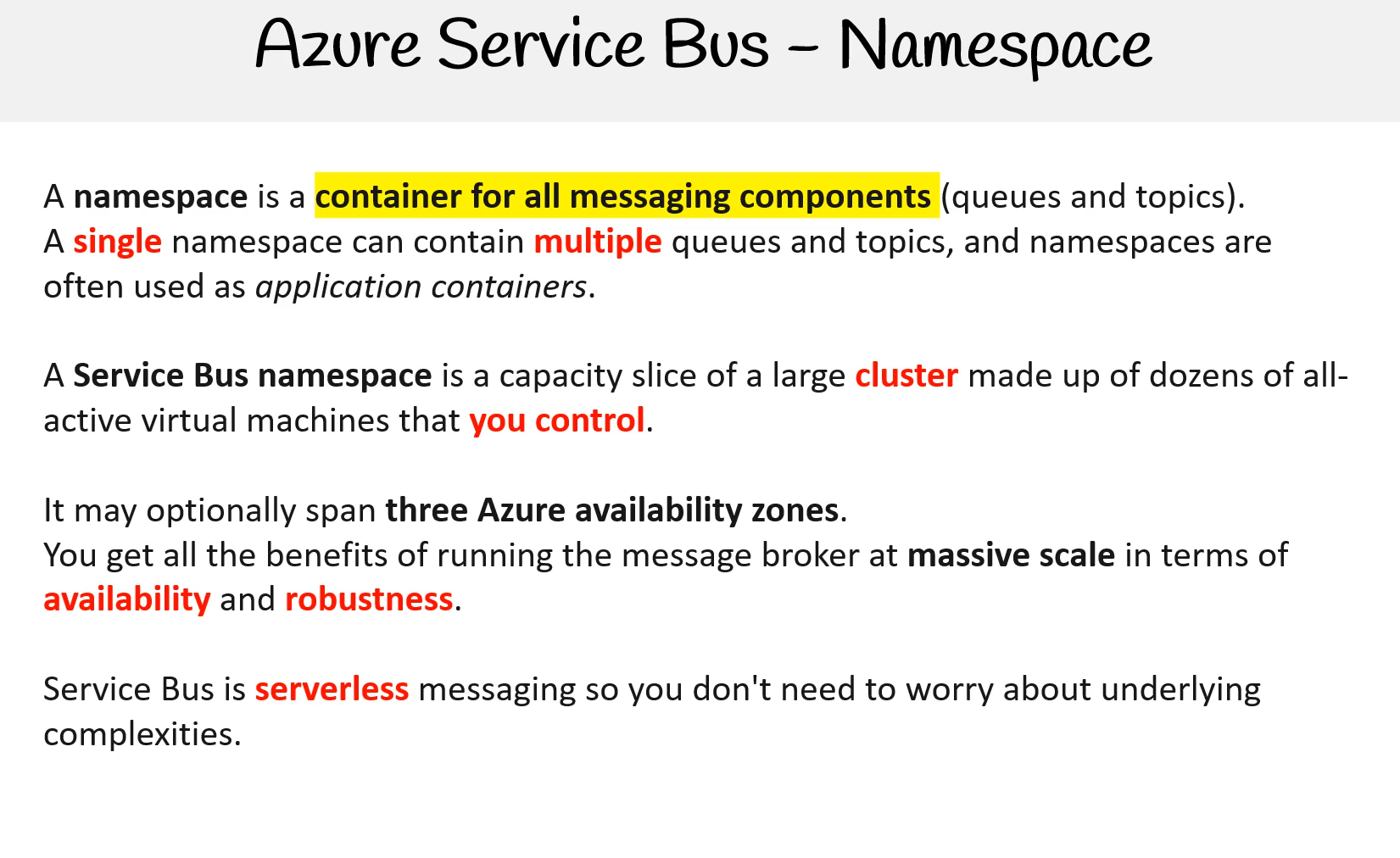
Peek kaydı silmiyor.

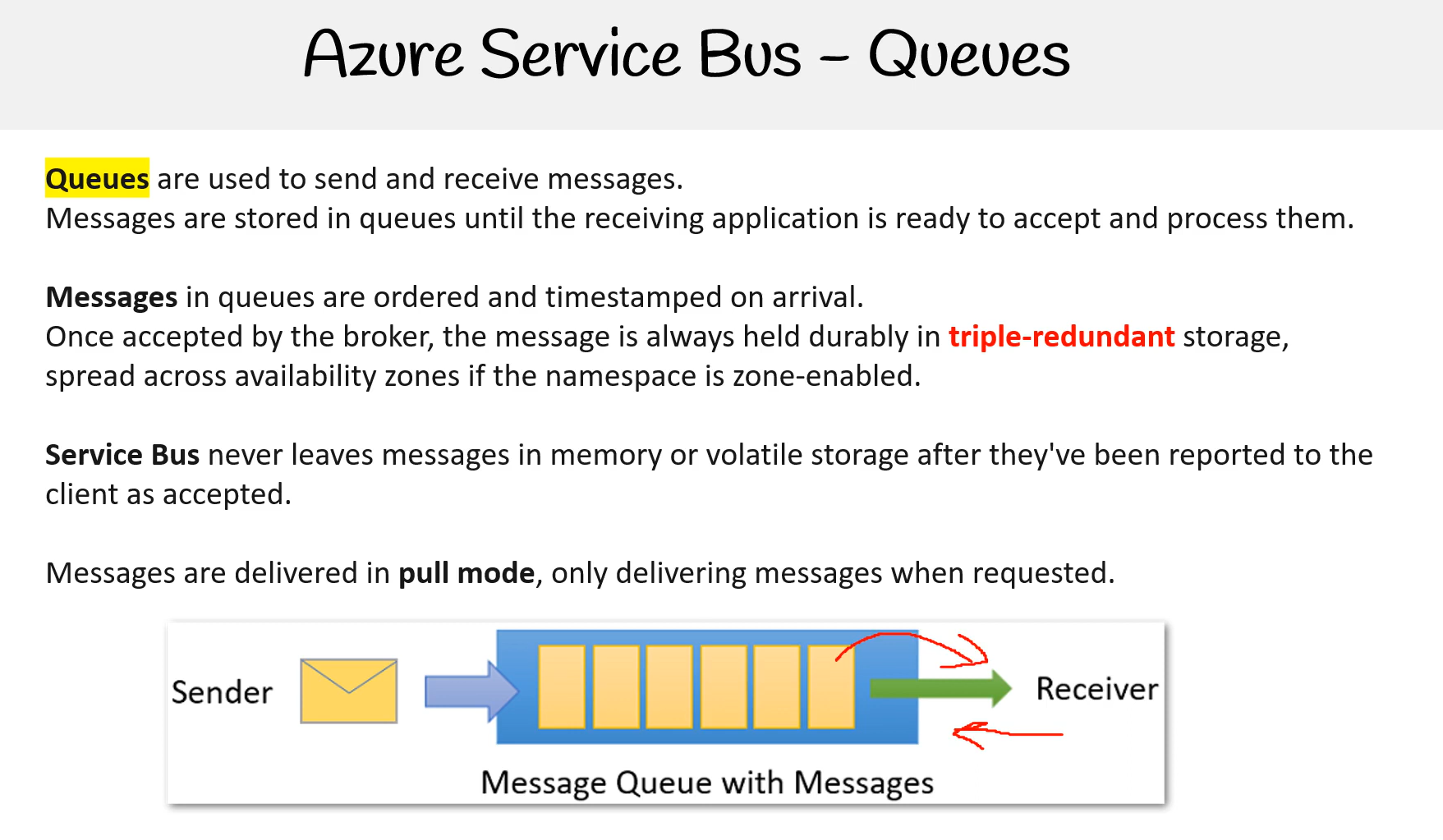
Örnek yapıyor

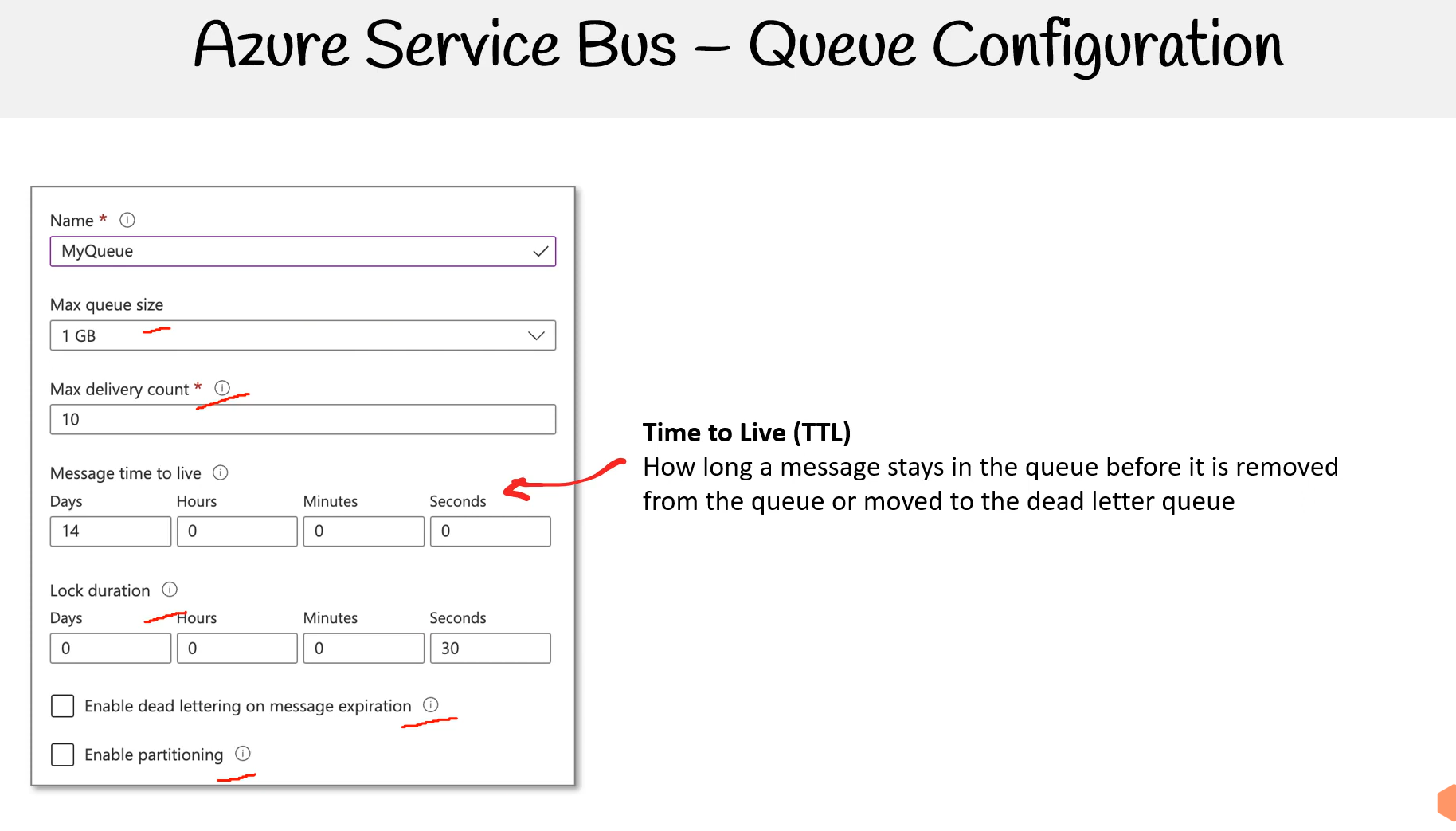
[(375) AZ 204 — Azure Queue Storage Follow Along - YouTube](https://www.youtube.com/watch?v=ZjouZHfxvk0)

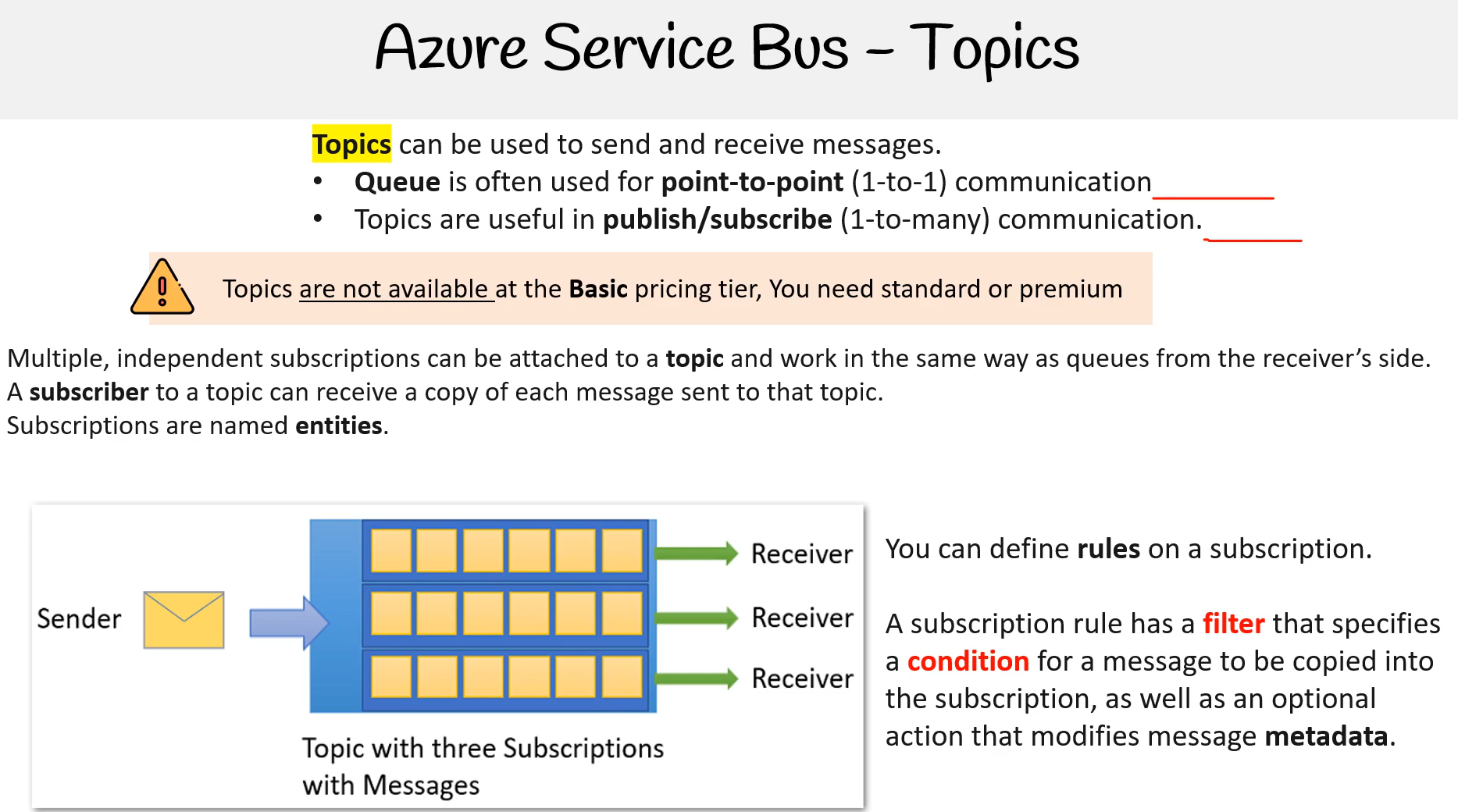


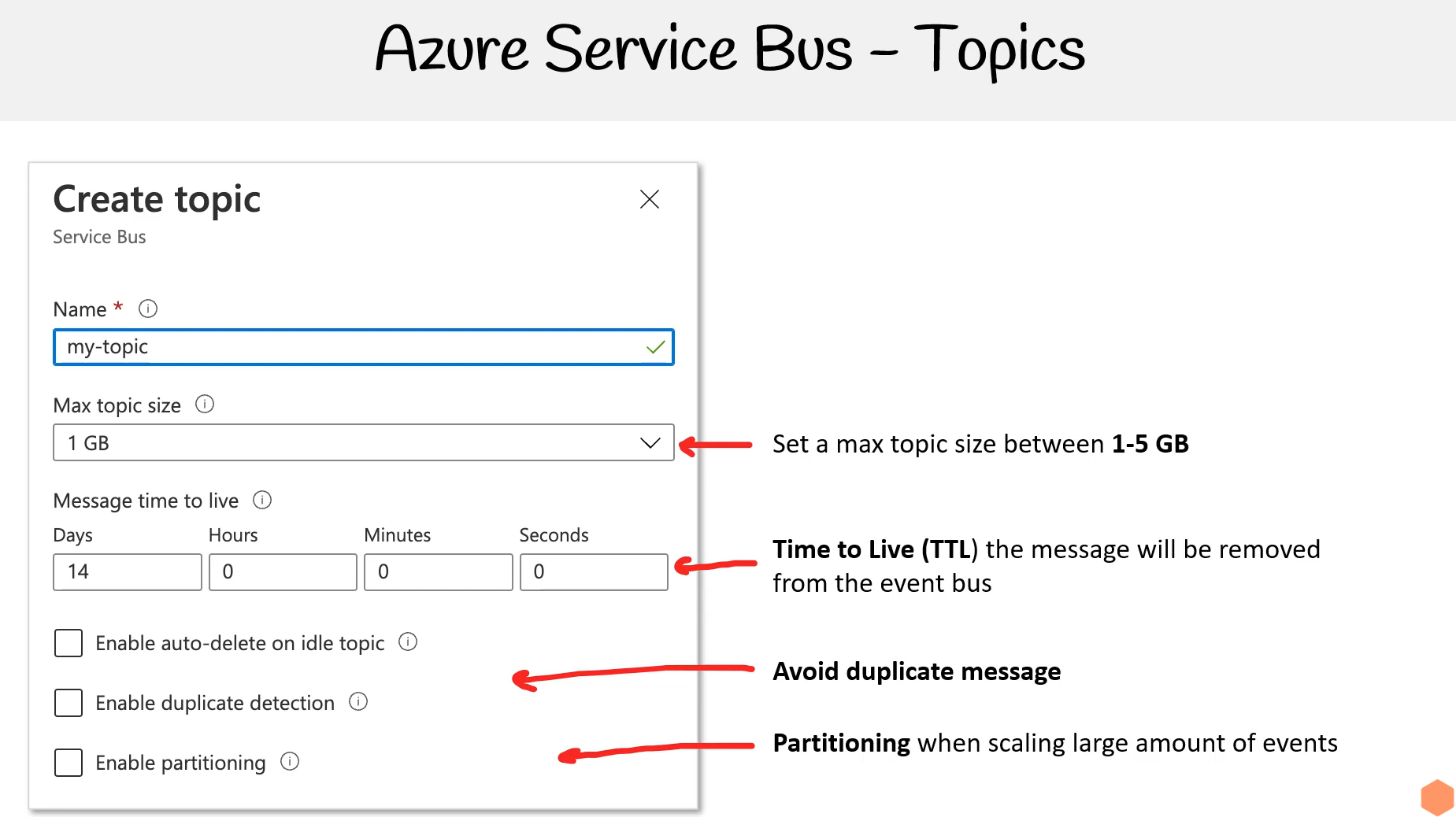


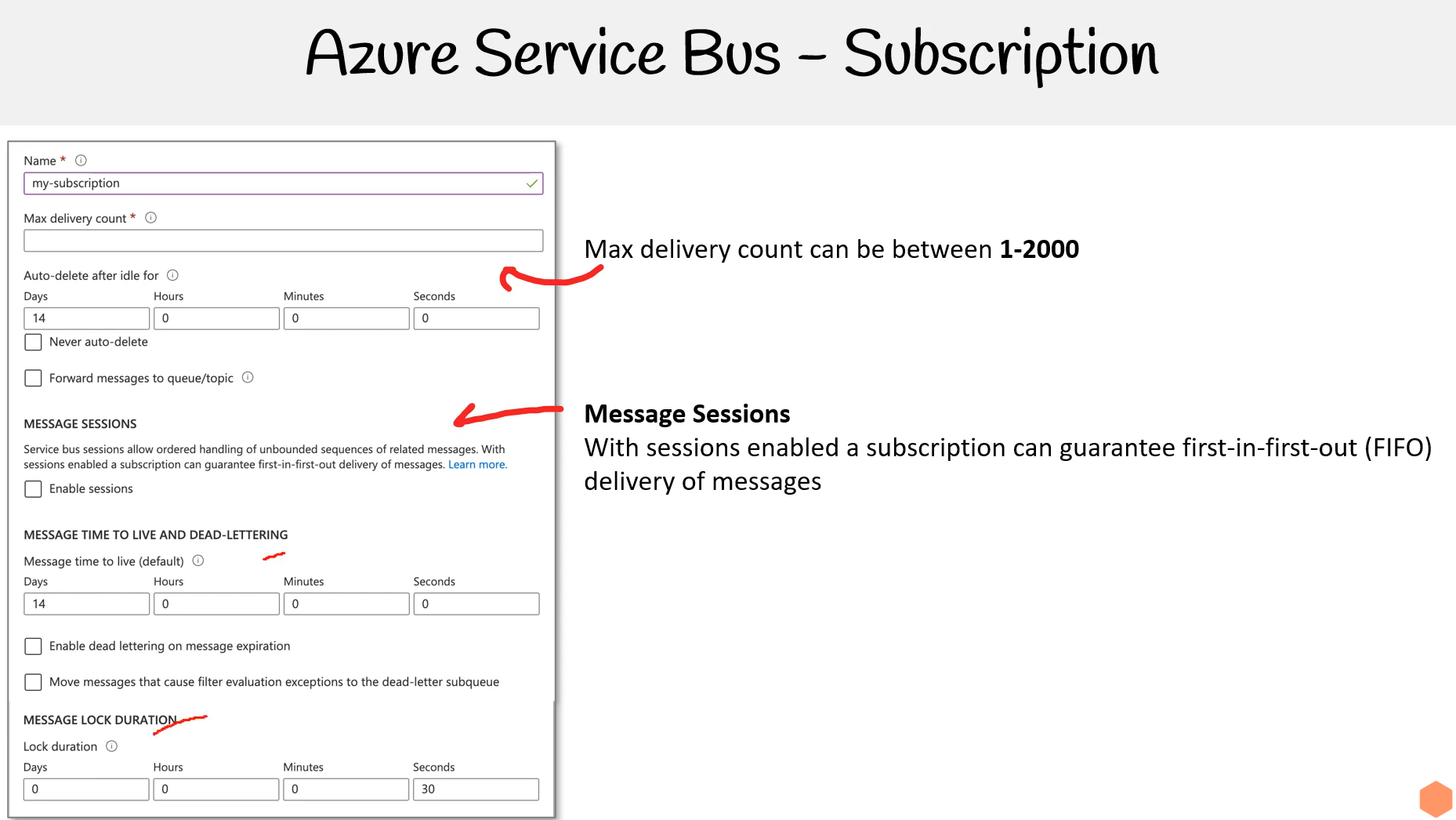


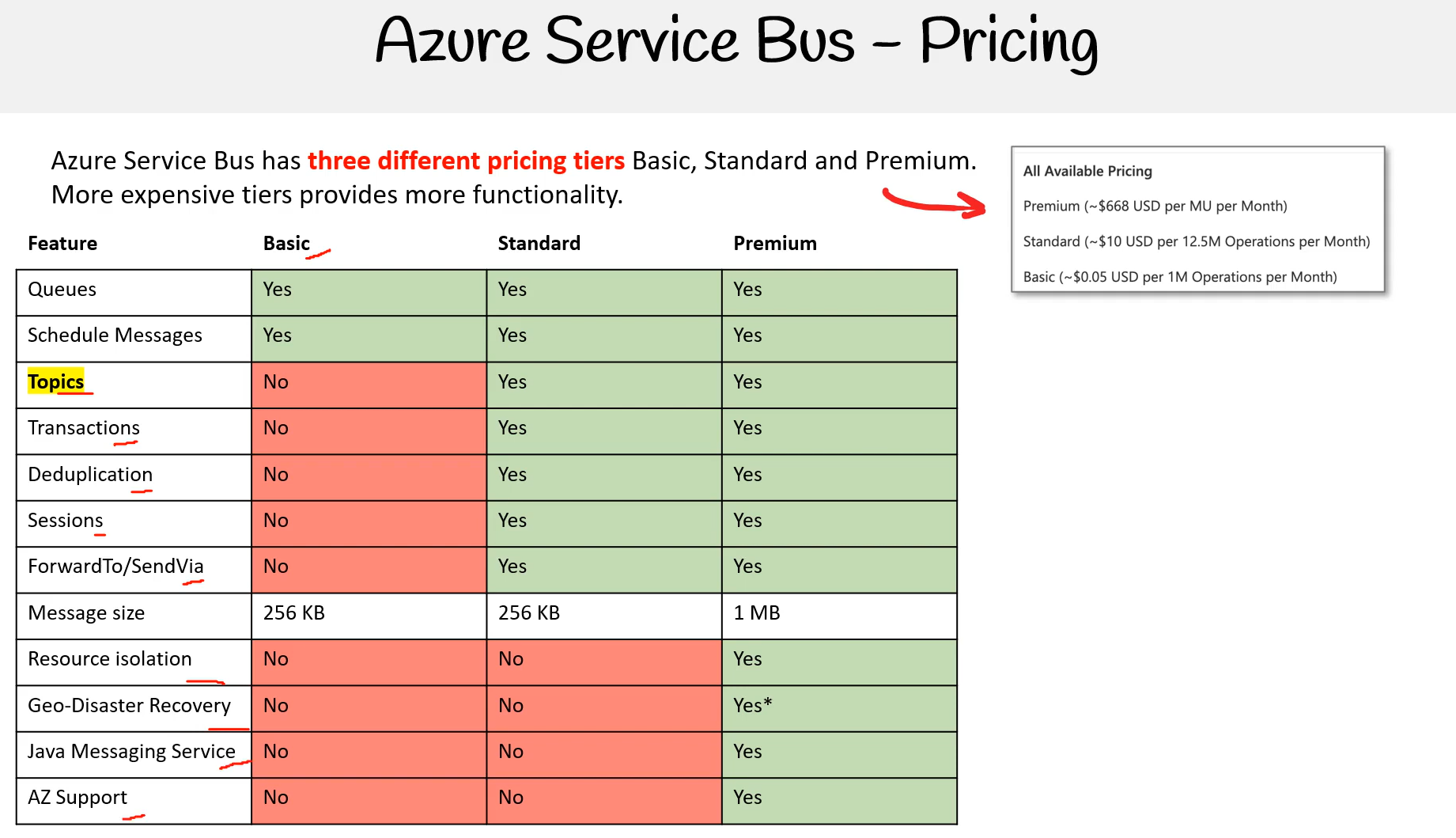


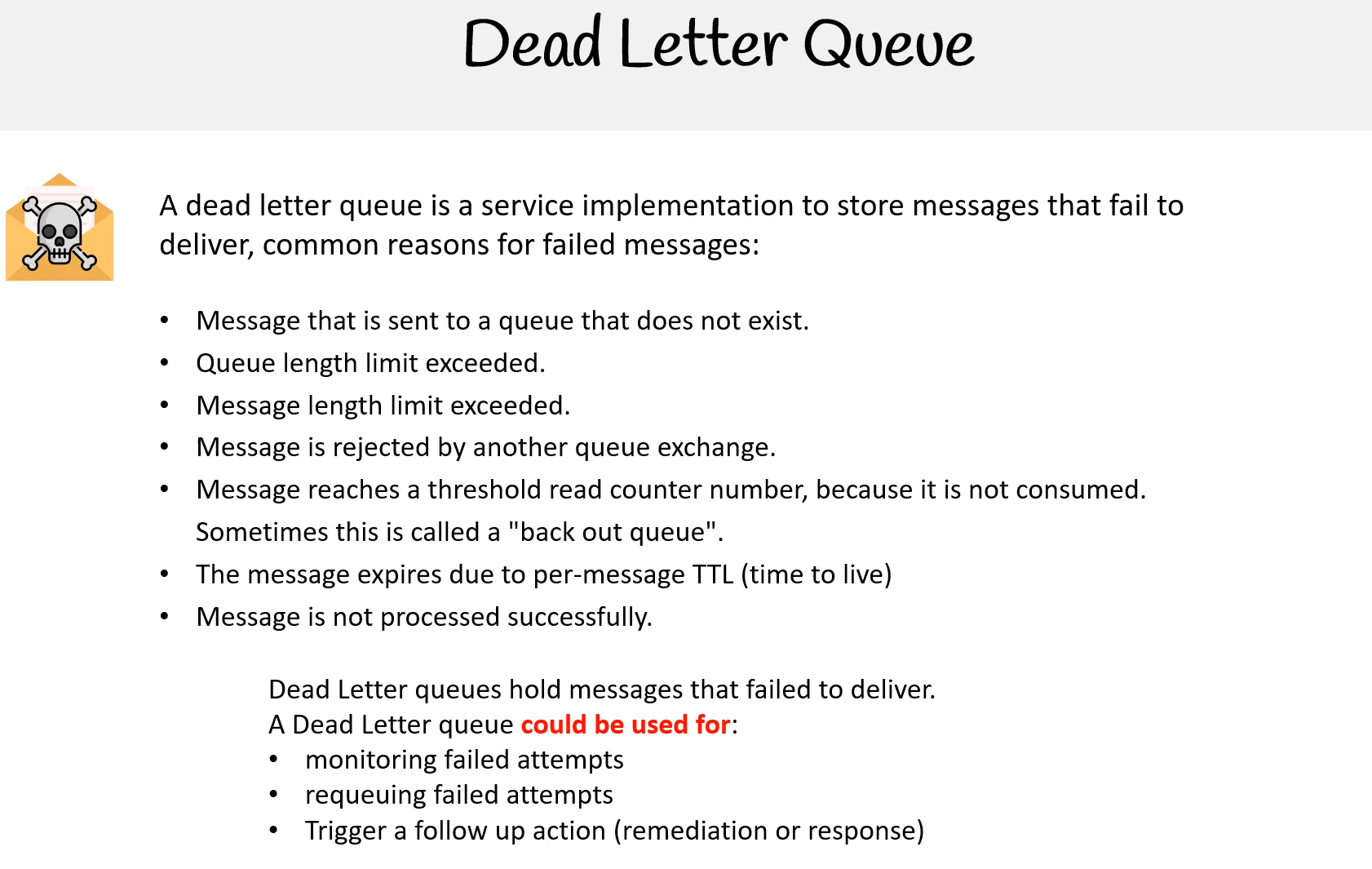


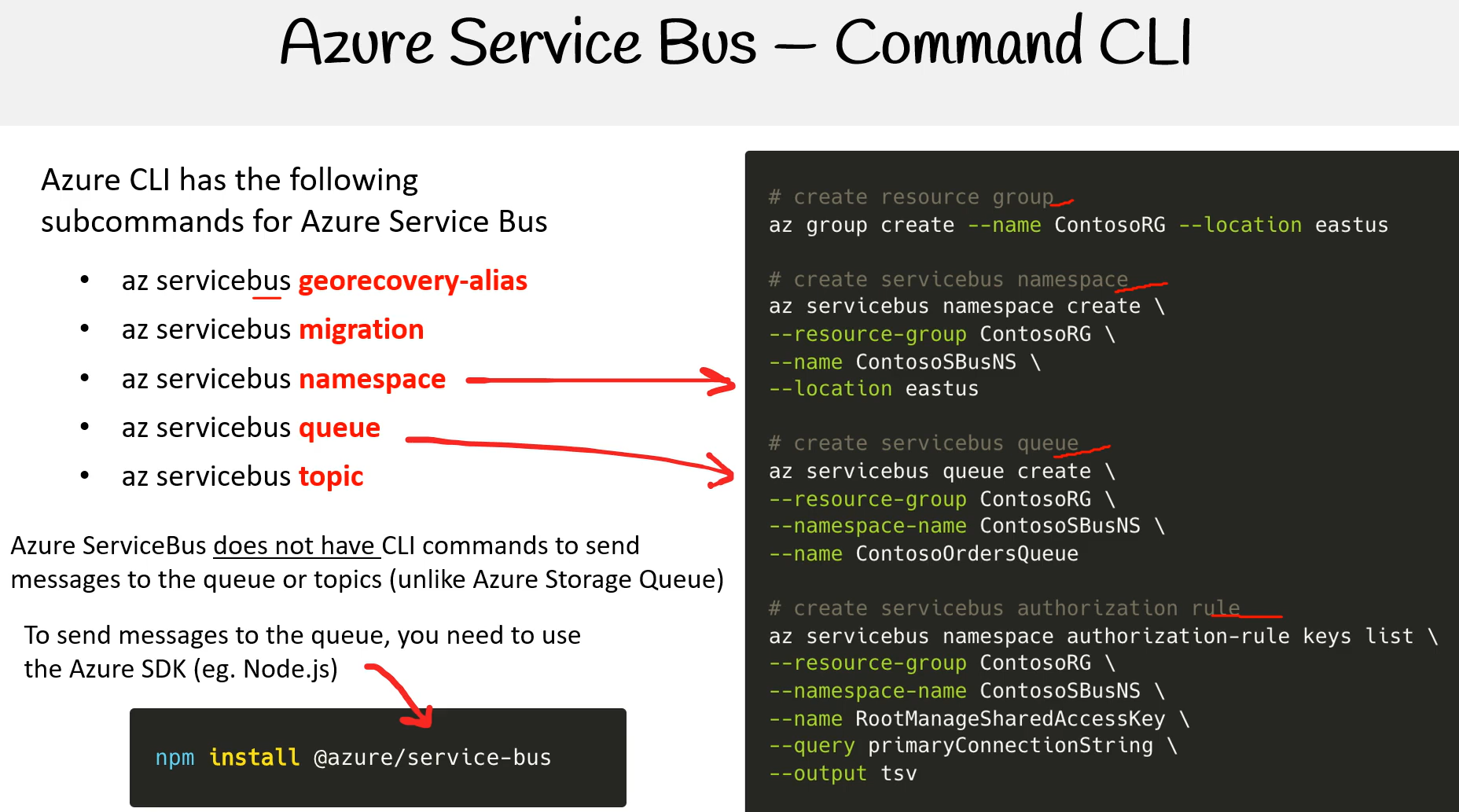












!!ÖNEMLİ CLI üzerinden mesaj gönderilemiyor. SDK ile programmatic olarak ilerletmek gerek. Storage Queue gibi değil.

Örnek Service Bus Queue

[(376) AZ 204 — Azure Service Bus - Queue - YouTube](https://www.youtube.com/watch?v=6_pWgqxAvQA)

Örnek Service Bus Topics

[AZ 204 — Azure Service Bus - Topics - YouTube](https://www.youtube.com/watch?v=Gtp2mX6HEp4&feature=youtu.be)

**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 limit.

**Consider using Storage queues**

As a solution architect/developer, **you should consider using Storage queues** when:

* Your application must store over 80 gigabytes of messages in a queue.
* Your application wants to track progress for processing a message in the queue. It's useful if the worker processing a message crashes. Another worker can then use that information to continue from where the prior worker left off.
* You require server side logs of all of the transactions executed against your queues.

# Explore Azure Service Bus

Microsoft Azure Service Bus is a fully managed enterprise integration message broker. Service Bus can decouple applications and services. Data is transferred between different applications and services using **messages**. A message is a container decorated with metadata, and contains data. The data can be any kind of information, including structured data encoded with the common formats such as the following ones: JSON, XML, Apache Avro, Plain Text.

* *Messaging*. Transfer business data, such as sales or purchase orders, journals, or inventory movements.
* *Decouple applications*. Improve reliability and scalability of applications and services. Client and service don't have to be online at the same time.
* *Topics and subscriptions*. Enable 1:*n* relationships between publishers and subscribers.
* *Message sessions*. Implement workflows that require message ordering or message deferral.

## Service Bus tiers

Service Bus offers a standard and premium tier. The premium tier of Service Bus Messaging addresses common customer requests around scale, performance, and availability for mission-critical applications. The premium tier is recommended for production scenarios. Although the feature sets are nearly identical, these two tiers of Service Bus Messaging are designed to serve different use cases.

Some high-level differences are highlighted in the following table.

| **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 100 MB | Message size up to 256 KB |

## Advanced features

Service Bus includes advanced features that enable you to solve more complex messaging problems. The following table describes several of these features.

| **Feature** | **Description** |
| --- | --- |
| Message sessions | To create a first-in, first-out (FIFO) guarantee in Service Bus, use sessions.  Message sessions enable exclusive, ordered handling of unbounded sequences of  related messages. |
| Autoforwarding | The autoforwarding feature chains a queue or subscription to another queue or  topic that is in the same namespace. |
| Dead-letter queue | Service Bus supports a dead-letter queue (DLQ).  A DLQ holds messages that can't be delivered to any receiver.  Service Bus lets you remove messages from the DLQ and inspect them. |
| Scheduled delivery | You can submit messages to a queue or topic for delayed processing.  You can schedule a job to become available for processing by a system at a certain time. |
| Message deferral | A queue or subscription client can defer retrieval of a message until a later time.  The message remains in the queue or subscription, but it's set aside. |
| Batching | Client-side batching enables a queue or topic client to delay sending a message for a certain period  of time. |
| Transactions | A transaction groups two or more operations together into an execution scope.  Service Bus supports grouping operations against a single messaging entity within  the scope of a single transaction. A message entity can be a queue, topic, or subscription. |
| Filtering 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. |
| Autodelete on idle | Autodelete on idle enables you to specify an idle interval after which a queue is automatically deleted.  The minimum duration is 5 minutes. |
| Duplicate detection | An error could cause the client to have a doubt about the outcome of a send operation. Duplicate detection enables the sender to resend the same message, or for the queue or topic to discard any duplicate copies. |
| Security protocols | Service Bus supports security protocols such as Shared Access Signatures (SAS), Role Based Access Control (RBAC) and Managed identities for Azure resources. |
| Geo-disaster recovery | When Azure regions or datacenters experience downtime, Geo-disaster recovery enables data processing to continue operating in a different region or datacenter. |
| Security | Service Bus supports standard AMQP 1.0 and HTTP/REST protocols. |

## 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. Because messages are stored durably in the queue, producers (senders) and consumers (receivers) don't have to process messages concurrently.

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 has to be able to handle average load instead of peak load.

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.

You can create queues using the Azure portal, PowerShell, CLI, or Resource Manager templates. Then, send and receive messages using clients written in C#, Java, Python, and JavaScript.

## Receive modes

You can specify two different modes in which Service Bus receives messages: **Receive and delete** or **Peek lock**.

# 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. For example, consider a scenario in which the consumer issues the receive request and then crashes before processing it. As Service Bus marks the message as being consumed, the application begins consuming messages upon restart. It misses the message that it consumed before the crash.

# Peek lock

In this mode, the receive operation becomes two-stage, which makes it possible to support applications that can't tolerate missing messages.

1. Finds the next message to be consumed, **locks** it to prevent other consumers from receiving it, and then, return the message to the application.
2. 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 being 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.

## Topics and subscriptions

A queue allows processing of a message by a single consumer. In contrast to queues, topics and subscriptions provide a one-to-many form of communication in a publish and subscribe pattern. It's useful for scaling to large numbers of recipients. Each published message is made available to each subscription registered with the topic. Publisher sends a message to a topic and one or more subscribers receive a copy of the message, depending on filter rules set on these subscriptions. The subscriptions can use more filters to restrict the messages that they want to receive.

Publishers send messages to a topic in the same way that they send messages to a queue. But, consumers don't receive messages directly from the topic. Instead, consumers receive messages from subscriptions of the topic. A topic subscription resembles a virtual queue that receives copies of the messages that are sent to the topic. Consumers receive messages from a subscription identically to the way they receive messages from a queue.

Creating a topic is similar to creating a queue, as described in the previous section. You can create topics and subscriptions using the Azure portal, PowerShell, CLI, or Resource Manager templates. Then, send messages to a topic and receive messages from subscriptions using clients written in C#, Java, Python, and JavaScript.

# Rules and actions

In many scenarios, messages that have specific characteristics must be processed in different ways. To enable this processing, you can configure subscriptions to find messages that have desired properties and then perform certain modifications to those properties. While Service Bus subscriptions see all messages sent to the topic, you can only copy a subset of those messages to the virtual subscription queue. This filtering is accomplished using subscription filters. Such modifications are called **filter actions**. When a subscription is created, you can supply a filter expression that operates on the properties of the message. The properties can be both the system properties (for example, **Label**) and custom application properties (for example, **StoreName**.) The SQL filter expression is optional in this case. Without a SQL filter expression, any filter action defined on a subscription is performed on all the messages for that subscription.

# Explore Service Bus message payloads and serialization

Messages carry a payload and metadata. The metadata is in the form of key-value pair properties, and describes the payload, and gives handling instructions to Service Bus and applications. Occasionally, that metadata alone is sufficient to carry the information that the sender wants to communicate to receivers, and the payload remains empty.

The object model of the official Service Bus clients for .NET and Java maps to and from the wire protocols Service Bus supports.

A Service Bus message consists of a binary payload section that Service Bus never handles in any form on the service-side, and two sets of properties. The broker properties are system defined. These predefined properties either control message-level functionality inside the broker, or they map to common and standardized metadata items. The user properties are a collection of key-value pairs defined and set by the application.

## Message routing and correlation

A subset of the broker properties, specifically To, ReplyTo, ReplyToSessionId, MessageId, CorrelationId, and SessionId, help applications route messages to particular destinations. The following patterns describe the routing:

* **Simple request/reply:** A publisher sends a message into a queue and expects a reply from the message consumer. The publisher owns a queue to receive the replies. The address of that queue is contained in the ReplyTo property of the outbound message. When the consumer responds, it copies the MessageId of the handled message into the CorrelationId property of the reply message and delivers the message to the destination indicated by the ReplyTo property. One message can yield multiple replies, depending on the application context.
* **Multicast request/reply:** As a variation of the prior pattern, a publisher sends the message into a topic and multiple subscribers become eligible to consume the message. Each of the subscribers might respond in the fashion described previously. If ReplyTo points to a topic, such a set of discovery responses can be distributed to an audience.
* **Multiplexing:** This session feature enables multiplexing of streams of related messages through a single queue or subscription such that each session (or group) of related messages, identified by matching SessionId values, are routed to a specific receiver while the receiver holds the session under lock. Learn more about the details of sessions [here](https://learn.microsoft.com/en-us/azure/service-bus-messaging/message-sessions).
* **Multiplexed request/reply:** This session feature enables multiplexed replies, allowing several publishers to share a reply queue. By setting ReplyToSessionId, the publisher can instruct the consumer(s) to copy that value into the SessionId property of the reply message. The publishing queue or topic doesn't need to be session-aware. When the message is sent the publisher can wait for a session with the given SessionId to materialize on the queue by conditionally accepting a session receiver.

Routing inside of a Service Bus namespace uses autoforward chaining and topic subscription rules. Routing across namespaces can be performed using Azure LogicApps. The To property is reserved for future use. Applications that implement routing should do so based on user properties and not lean on the To property; however, doing so now won't cause compatibility issues.

## Payload serialization

When in transit or stored inside of Service Bus, the payload is always an opaque, binary block. The ContentType property enables applications to describe the payload, with the suggested format for the property values being a MIME content-type description according to IETF RFC2045; for example, application/json;charset=utf-8.

Unlike the Java or .NET Standard variants, the .NET Framework version of the Service Bus API supports creating BrokeredMessage instances by passing arbitrary .NET objects into the constructor.

The legacy SBMP protocol serializes objects with the default binary serializer, or with a serializer that is externally supplied. The AMQP protocol serializes objects into an AMQP object. The receiver can retrieve those objects with the GetBody<T>() method, supplying the expected type. With AMQP, the objects are serialized into an AMQP graph of ArrayList and IDictionary<string,object> objects, and any AMQP client can decode them.

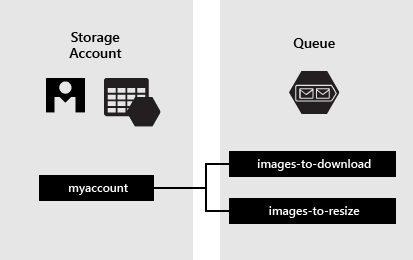
While this hidden serialization magic is convenient, if applications should take explicit control of object serialization and turn their object graphs into streams before including them into a message, they should do the reverse operation on the receiver side. While AMQP has a powerful binary encoding model, it's tied to the AMQP messaging ecosystem and HTTP clients have trouble decoding such payloads.

**ApplicationBuild Yapıyor!!**

# Explore Azure Queue Storage

Azure Queue Storage is a service for storing large numbers of messages. You access messages from anywhere in the world via authenticated calls using HTTP or HTTPS. A queue message can be up to 64 KB in size. A queue may contain millions of messages, up to the total capacity limit of a storage account. Queues are commonly used to create a backlog of work to process asynchronously.

The Queue service contains the following components:



* **URL format:** Queues are addressable using the URL format https://<storage account>.queue.core.windows.net/<queue>. For example, the following URL addresses a queue in the diagram above https://myaccount.queue.core.windows.net/images-to-download
* **Storage account:** All access to Azure Storage is done through a storage account.
* **Queue:** A queue contains a set of messages. All messages must be in a queue. The queue name must be all lowercase.
* **Message:** A message, in any format, of up to 64 KB. For version 2017-07-29 or later, the maximum time-to-live can be any positive number, or -1 indicating that the message doesn't expire. If this parameter is omitted, the default time-to-live is seven days.

