

Messages and Azure Service Bus Overview

Azure Service Bus is a fully managed enterprise messaging system that enables reliable communication between distributed applications.

It decouples application components, allowing them to exchange messages in a secure and asynchronous manner.

In **AzureServiceBusFlow (AsbFlow)**, a **message** represents the fundamental unit of communication between different parts of the system.

Messages can be commands, events, or any other type of information that needs to be transmitted through queues or topics.

What Is a Message?

A **message** is a piece of data sent from one application or service to another. It can represent:

- A request to perform an operation
- A notification that something has occurred
- Data exchange between microservices

Message Flow

1. A **producer** sends a message to a queue or topic.
2. The **Azure Service Bus** stores and delivers the message reliably.
3. A **consumer** receives the message and processes it.



This architecture ensures asynchronous, loosely coupled communication, improving scalability and resilience in distributed systems.

AsbFlow's Role

AzureServiceBusFlow simplifies the process of defining and managing these messages by:

- Providing a clean, strongly typed model for message definition
- Enforcing consistent configuration patterns for message producers and consumers
- Allowing automatic setup of queues, topics, and subscriptions

Commands and Events

In **AzureServiceBusFlow**, messages are generally divided into two conceptual categories: **Commands** and **Events**.

Although they are represented in code using the same message structures, their **purpose and meaning** differ from an architectural perspective.

Commands

A **command** expresses the **intention to perform an action**.

It usually represents a request from one part of the system asking another to execute a specific operation.

Commands are **imperative** — they tell the system *what should be done*.

Examples:

- `CreateOrderCommand`
- `UpdateUserProfileCommand`
- `ProcessPaymentCommand`

A command typically has a **single intended consumer**, as it represents a direct request that must be handled once. **It is possible to configure more than one consumer for a command, although this approach is more commonly applied to events.*



Events

An **event** indicates that **something has already happened**.

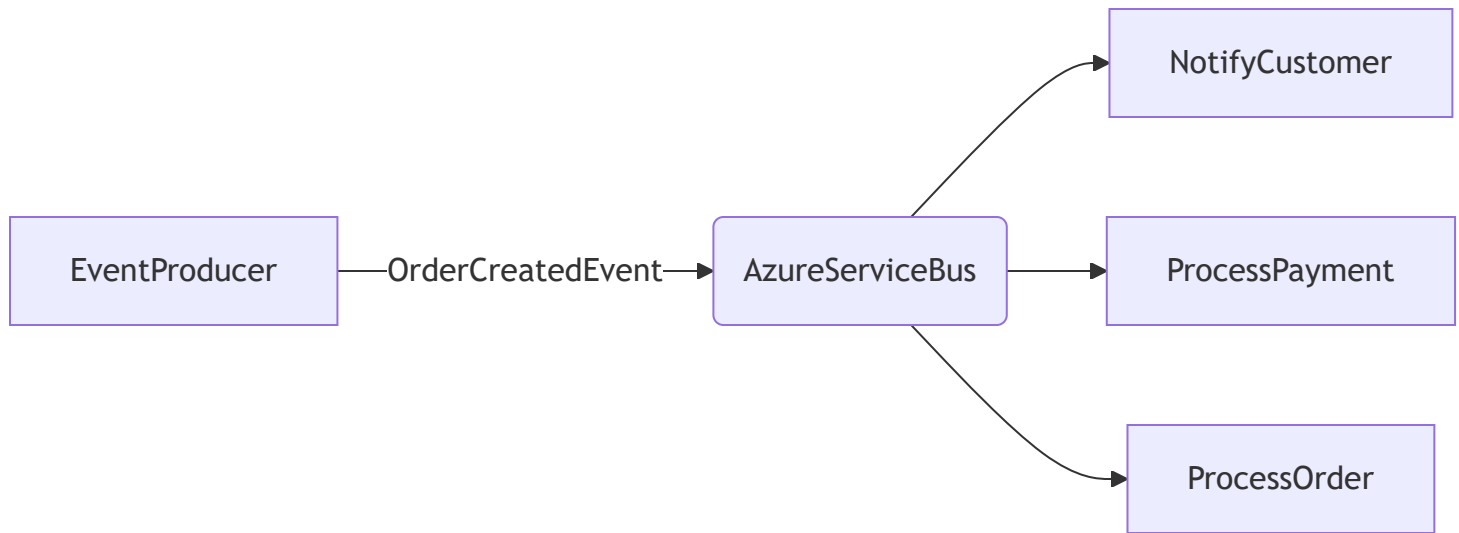
It is a **notification** sent to inform other parts of the system that a specific state change occurred.

Events are **descriptive** — they tell the system *what has happened*.

Examples:

- `OrderCreatedEvent`
- `UserProfileUpdatedEvent`
- `PaymentProcessedEvent`

Unlike commands, events are often **broadcast** to multiple subscribers, allowing different services to react independently.



⚙ Architectural Difference

	Command	Event
Meaning	Intention to perform an action	Notification that something occurred
Temporal Aspect	Future-oriented	Past-oriented
Consumer Count	*Usually one	One or many
Communication Type	Point-to-point	Publish-subscribe
Example	CreateOrderCommand	OrderCreatedEvent

💡 Although they share the same implementation in code, their **semantic purpose** defines whether a message should be treated as a command or an event.

💡 * Even though it is technically possible to have multiple **consumers** for the same **message**, this is not a common pattern for commands. A **command** represents a specific intent to perform a single action, typically handled by one consumer. **Events**, on the other hand, are better suited for multiple consumers, since they signal that something has already happened and can notify several parts of the system independently.

Queues and Topics

Azure Service Bus provides two main types of messaging entities — **queues** and **topics** — each suited for specific communication patterns.



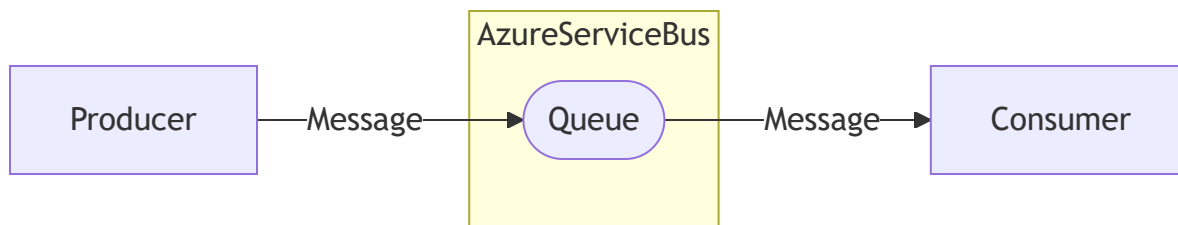
Queues

A **queue** is a simple, point-to-point communication channel.

- Each message is received by **only one** consumer.
- Messages are stored until a consumer successfully processes them.
- Ideal for **command processing** or **task-based** workloads.

Example use cases:

- Order processing
- Email sending jobs
- Background task execution



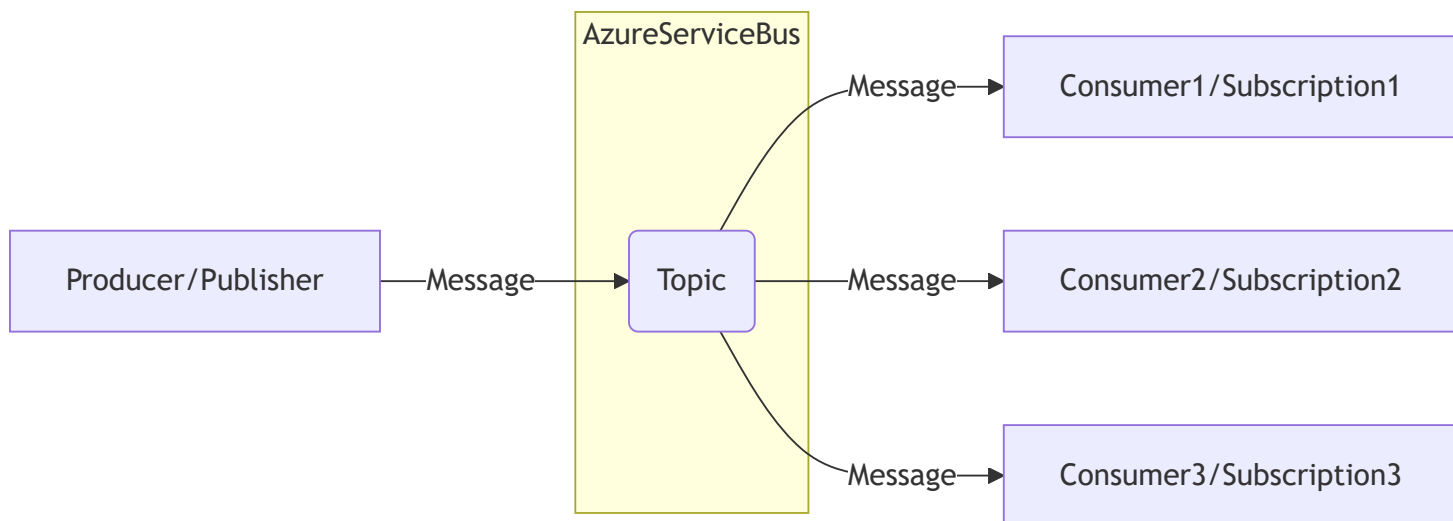
Topics and Subscriptions

A **topic** enables a **publish-subscribe** messaging pattern.

- Producers send messages to a topic.
- Multiple **subscriptions** can be attached to a topic.
- Each subscription receives a **copy** of the message.
- Ideal for **event broadcasting** or **system-wide notifications**.

Example use cases:

- Notifying multiple microservices about an order creation
- Updating caches after a data change
- Triggering independent workflows after an event



Summary

Feature	Queue	Topic
Pattern	Point-to-point	Publish-subscribe
Consumers per message	One	Many (via subscriptions)
Typical usage	Commands	Events
Message copy	Single	One per subscription

AsbFlow automatically configures queues, topics, and subscriptions for you, making setup and maintenance straightforward.

Producers and Consumers

In **AzureServiceBusFlow**, communication between services happens through **producers** and **consumers**, which abstract away the complexity of connecting to Azure Service Bus.

Producers

A **producer** (also known as a **sender**) is responsible for **sending messages** to a queue or topic.

With **AsbFlow**, producers are registered and configured fluently using simple definitions. They can send either **commands** or **events**, depending on the communication model.

Example responsibilities:

- Serializing the message payload
- Sending the message to the appropriate Azure Service Bus entity
- Logging and error handling

Consumers

A **consumer** (or **receiver**) is responsible for **receiving and processing messages** from queues or subscriptions.

In **AsbFlow**, consumers are also registered fluently and can automatically bind to queues or subscriptions, depending on message type.

Example responsibilities:

- Deserializing the message
- Executing the related business logic
- Handling retries and exceptions

Relationship

Component	Description
Producer	Sends messages to Azure Service Bus (queues or topics).
Consumer	Listens for and processes incoming messages.
Queue/Topic	The channel through which messages flow.

This abstraction makes it easy to maintain a **clean architecture**, where producers and consumers are clearly separated and follow consistent configuration patterns.

Messages

In **AzureServiceBusFlow (AsbFlow)**, every message that is sent through the Azure Service Bus — whether it is a **Command** or an **Event** — must implement the `IServiceBusMessage` interface.

This interface defines the basic structure that all messages must follow, ensuring consistency and allowing the framework to automatically handle serialization, metadata, and routing.

IServiceBusMessage Interface

```
public interface IServiceBusMessage
{
    string RoutingKey { get; }
    DateTime CreatedDate { get; }
}
```

Every message must expose:

- **RoutingKey** → used to identify or categorize the message (often a unique ID or business key)
- **CreatedDate** → timestamp of when the message was created

Example of a Command Message

```
public class ExampleCommand1 : IServiceBusMessage
{
    public string RoutingKey => ExampleMessage.Id.ToString();
    public DateTime CreatedDate => DateTime.UtcNow;
    public required ExampleMessage ExampleMessage { get; set; }
}

public class ExampleMessage
{
    public Guid Id { get; set; }
    public string? Cliente { get; set; }
    public decimal Valor { get; set; }
}
```

This structure provides:

- A **strongly typed** message model
- Easy serialization through the built-in producer

- A unified standard for both **commands** and **events**

Architectural Meaning

Although both **commands** and **events** share the same implementation, they differ in intention:

- A **Command** represents the intention to perform an action.
- An **Event** indicates that something has already happened.

The distinction is purely architectural and helps maintain clear separation of responsibilities between services.

Message Handlers

A **Message Handler** defines how a message should be processed once it is received from Azure Service Bus.

In **AzureServiceBusFlow**, every handler must implement the `IMessageHandler<TMessage>` interface, where `TMessage` is the type of message being handled.

This ensures a clean and consistent processing model across all services.

IMessageHandler Interface

```
public interface IMessageHandler<in TMessage>
{
    Task HandleAsync(
        TMessage message,
        ServiceBusReceivedMessage rawMessage,
        CancellationToken cancellationToken);
}
```

Example of a Command Handler

```
public class CommandExample1Handler : IMessageHandler<ExampleCommand1>
{
    public Task HandleAsync(ExampleCommand1 message, ServiceBusReceivedMessage rawMessage,
        CancellationToken cancellationToken)
    {
        // Business logic to handle the command
        Console.WriteLine($"Processing command for client:
{message.ExampleMessage.Cliente}");

        return Task.CompletedTask;
    }
}
```

In this example:

- The handler consumes an `ExampleCommand1` message.
- It receives both the **deserialized message** and the raw **Azure Service Bus message**.
- The method `HandleAsync` is executed automatically when a message is consumed.



Key Benefits

- **Strong typing** ensures compile-time safety.
- **Clear separation** between different message types and business logic.
- **Automatic binding** via dependency injection and the AsbFlow configuration.

Handlers allow your application to stay modular, scalable, and easy to test.

Producers

Producers are responsible for **sending messages** to Azure Service Bus queues or topics.

The **AzureServiceBusFlow** package provides built-in producer interfaces and implementations, allowing you to send both **Commands** and **Events** easily and consistently.

Producer Interfaces

The library defines two main producer interfaces:

```
public interface ICommandProducer<in TCommand> where TCommand : class, IServiceBusMessage
{
    /// <summary>
    /// Produces the command without additional options.
    /// </summary>
    Task ProduceCommandAsync(TCommand command, CancellationToken cancellationToken);

    /// <summary>
    /// Produces the command using explicit message options such as delay or
    application properties.
    /// </summary>
    Task ProduceCommandAsync(TCommand command, MessageOptions messageOptions,
    CancellationToken cancellationToken);

    /// <summary>
    /// Produces the command with a set of application properties that will be attached to
    the message.
    /// </summary>
    Task ProduceCommandAsync(TCommand command, IDictionary<string, object>
    applicationProperties, CancellationToken cancellationToken);

    /// <summary>
    /// Produces the command with a delivery delay before it becomes available
    for processing.
    /// </summary>
    Task ProduceCommandAsync(TCommand command, TimeSpan delay,
    CancellationToken cancellationToken);
}

public interface IEventProducer<in TEvent> where TEvent : class, IServiceBusMessage
{
    Task ProduceEventAsync(TEvent @event, CancellationToken cancellationToken);
}
```

These interfaces abstract the underlying logic of sending messages to the Azure Service Bus.

The overload of `ProduceCommandAsync` by the `ICommandProducer<TCommand>` interface accepts a `MessageOptions` object, defined as:

```
public record MessageOptions(TimeSpan? Delay, IDictionary<string,
object>? ApplicationProperties);
```

This allows you to:

- Set a **delay** for the message to be consumed — for example, you can configure it to be available **3 minutes after being published**.
- Attach custom headers through the `ApplicationProperties` dictionary that travels with the message. These headers can carry metadata useful for all messages, such as tenant information, user identity, or special processing flags (e.g., skipping a message in some scenarios).

The others 2 overloads of `ProduceCommandAsync` allows you to send a message passing only the `delay` or the `applicationProperties`, removing the usage of `MessageOptions` if only one of those properties will be used.

This feature enables flexible message customization without modifying the message payload itself, following best practices for message-driven architectures.

Internal Implementation

Both producers rely on a shared low-level component: the `IServiceBusProducer` interface.

```
public interface IServiceBusProducer<in TMessage> where TMessage : class, IServiceBusMessage
{
    Task ProduceAsync(TMessage message, CancellationToken cancellationToken);
}
```

The default implementation, `ServiceBusProducer<TMessage>`, uses the Azure SDK to serialize and send the message:

```
public class ServiceBusProducer<TMessage> : IServiceBusProducer<TMessage> where TMessage :
class, IServiceBusMessage
{
    public async Task ProduceAsync(TMessage message, CancellationToken cancellationToken)
    {
        var json = JsonConvert.SerializeObject(message);
        var serviceBusMessage = new ServiceBusMessage(json)
```

```

{
    Subject = message.RoutingKey,
    ApplicationProperties =
    {
        { "MessageType", message.GetType().FullName },
        { "CreatedAt", message.CreatedDate.ToString("O") }
    }
};

await _sender.SendMessageAsync(serviceBusMessage, cancellationToken);
_logger.LogInformation("Message {MessageType} published
successfully!", message.GetType().Name);
}
}

```



Example of Usage in a Controller

You don't need to create a custom producer class — AsbFlow already provides the implementations. Simply inject the appropriate interface (**ICommandProducer** or **IEventProducer**) and call the **Produce...Async** method:

```

[Route("api/commands")]
[ApiController]
public class CommandController(ICommandProducer<ExampleCommand1> _producer) : ControllerBase
{
    [HttpPost("command-example-one")]
    public async Task<IActionResult> Example1(Cancellation token cancellationToken)
    {
        ExampleCommand1 command = new()
        {
            ExampleMessage = new ExampleMessage
            {
                Cliente = "jose",
                Id = Guid.NewGuid(),
                Valor = 1111
            }
        };

        await _producer.ProduceCommandAsync(command, cancellationToken);
        return Ok("Command published successfully!");
    }
}

```

Example with Application Properties

In this example we're passing some values in application properties that can be used in MessageHandler.

```
public async Task<IActionResult> Example2WithApplicationProperties(CancellationTokencancellationTokencancellationToken)
{
    ExampleCommand2 command = new()
    {
        ExampleMessage = new ExampleMessage
        {
            Cliente = "jose",
            Id = Guid.NewGuid(),
            Valor = 1111
        }
    };

    var applicationProperties = new Dictionary<string, object>
    {
        { "CorrelationId", Guid.NewGuid().ToString() },
        { "Priority", "High" }
    };

    await _producerTwo.ProduceCommandAsync(command, applicationProperties,
cancellationToken);
    return Ok();
}
```

Example with Delay

In this example we're passing a delay of 5 minutes, meaning that this message should be processed 5 minutes after sending it.

```
public async Task<IActionResult> Example2WithDelay(CancellationTokencancellationToken)
{
    ExampleCommand2 command = new()
    {
        ExampleMessage = new ExampleMessage
        {
            Cliente = "jose",
            Id = Guid.NewGuid(),
            Valor = 1111
        }
    };
};
```



```
var delay = TimeSpan.FromMinutes(5);

await _producerTwo.ProduceCommandAsync(command, delay, cancellationToken);
return Ok();
}
```

You can combine the `applicationProperties` and `delay` in `MessageOptions` if you want the message to be sent with this two parameters together.

Key Points

- Both **CommandProducer** and **EventProducer** share the same implementation logic.
- The difference lies only in **architectural intent** — commands indicate actions to be performed, events indicate actions that already happened.
- You can use dependency injection to directly obtain any producer type.

Middlewares

In the context of `AzureServiceBusFlow`, **Middlewares** are components that run before a message is sent to Azure Service Bus.

They allow you to intercept, inspect, or modify the `ServiceBusMessage` — for example, adding custom properties, performing logging, or applying transformations — without changing the core sending logic.

Each middleware **executes in sequence** through a pipeline, and after all have finished, the message is finally delivered to Azure Service Bus.

Middleware Interface

`AsbFlow` defines an interface for creating custom middlewares.

```
public interface IProducerMiddleware
{
    Task InvokeAsync(ServiceBusMessage message, Func<Task> next);
}
```

This interface defines a single method, `InvokeAsync`, which receives:

- The `ServiceBusMessage` being processed.
- A delegate `next`, which represents the next middleware in the pipeline or the final send operation.

Example of Middleware

```
public class AsbSampleMiddleware : IProducerMiddleware
{
    public async Task InvokeAsync(ServiceBusMessage message, Func<Task> next)
    {
        // Add a custom property if it doesn't exist
        if (!message.ApplicationProperties.ContainsKey("SampleMiddleware"))
        {
            message.ApplicationProperties["SampleMiddleware"] = "Executed";
        }

        // Add a UTC timestamp for debugging or tracking
        message.ApplicationProperties["ProcessedAtUtc"] = DateTime.UtcNow.ToString("O");

        // Continue the pipeline (invoke the next middleware or the actual send)
        await next();
    }
}
```

```
}  
}
```

This sample middleware demonstrates how to enrich a message before it is published. It adds two custom properties:

- `"SampleMiddleware"` – a flag indicating that the middleware has executed.
- `"ProcessedAtUtc"` – a timestamp showing when the message was processed.

By calling `await next()`, it ensures the next middleware (or the send operation) is executed, maintaining the correct order of the pipeline.

⚙️ Registering the Middleware in the Pipeline

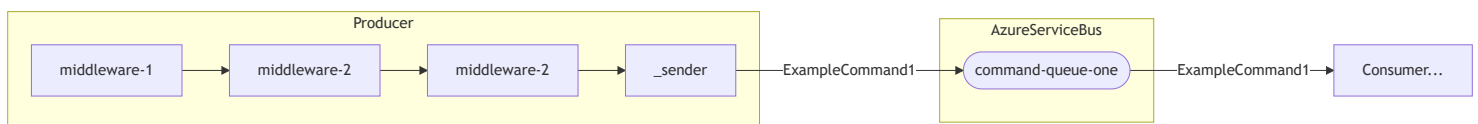
Once your middleware is implemented, you can easily register it in the producer pipeline through the `UseMiddleware()` extension method in your `Program.cs`:

```
builder.Services.AddAzureServiceBus(cfg => cfg  
    .ConfigureAzureServiceBus(azureServiceBusConfig)  
    .AddProducer<ExampleCommand1>(p => p  
        .EnsureQueueExists("command-queue-one")  
        .WithCommandProducer()  
        .UseMiddleware<AsbSampleMiddleware>()  
        .ToQueue("command-queue-one"));
```

In this example:

The `AsbSampleMiddleware` will be executed every time a message is produced for the queue `command-queue-one`.

Multiple middlewares can be chained by calling `UseMiddleware` multiple times — they are executed in the same order they are added.



Configuring your API



Step 1: Install the NuGet Package

Start by installing the **AzureServiceBusFlow** package from NuGet:

```
dotnet add package AzureServiceBusFlow
```

You can also find it [on NuGet.org](https://www.nuget.org/packages/AzureServiceBusFlow/).

This package contains all the necessary components to configure Producers / Consumers, register Queues / Topics and publish Messages (Commands/ Events) to AzureServiceBus.



Step 2: Configure the `appsettings.json`

After installing the package, it's time to configure your `appsettings.json` file. You need to define the basic configuration for Azure Service Bus to work.

Here's a configuration example:

```
"AzureServiceBusConfigurationSettings": {  
  "ConnectionString": "",  
  "ServiceBusReceiveMode": "ReceiveAndDelete",  
  "MaxAutoLockRenewalDurationInSeconds": "1800",  
  "MaxConcurrentCalls": "10",  
  "MaxRetryAttempts": "3"  
}
```

- **ConnectionString:** The full connection string to your Azure Service Bus instance, typically in the format: `Endpoint=sb://<your-servicebus-namespace-name>.servicebus.windows.net/;SharedAccessKeyName=<your-access-key-name>;SharedAccessKey=<your-access-key-value>.`
- **ServiceBusReceiveMode:** Defines how messages are handled after being received.
 - **"PeekLock":** The message is locked for processing and remains in the queue until it is explicitly completed or abandoned. If the receiver fails to complete the message within the lock duration, the message becomes available again for other receivers.
 - **"ReceiveAndDelete":** The message is automatically removed from the queue as soon as it is received.
- **MaxAutoLockRenewalDurationInSeconds:** The maximum duration, in seconds, that the message lock will be automatically renewed while the message is being processed.

- **MaxConcurrentCalls:** Specifies the maximum number of messages that can be processed concurrently.
- **MaxRetryAttempts:** Defines the maximum number of retry attempts the message handler will perform to reprocess a message if an exception occurs.

💡 The `MaxRetryAttempts` property only makes sense when using **ReceiveAndDelete** mode. In **PeekLock** mode, Azure Service Bus retries message processing 10 times by default, so explicit retry configuration is unnecessary.

✖ Step 3: Register AzureServiceBus in Program.cs

Now that your `appsettings.json` is configured, it's time to enable AzureServiceBus in your application.

Inside your `Program.cs`, register the AzureServiceBus in DI container with the `AddAzureServiceBus` extension method and set the configure method.

```
var azureServiceBusConfig = new AzureServiceBusConfiguration
{
    ConnectionString = "",
    ServiceBusReceiveMode =
    Azure.Messaging.ServiceBus.ServiceBusReceiveMode.ReceiveAndDelete,
    MaxConcurrentCalls = 10,
    MaxAutoLockRenewalDurationInSeconds = 1800,
    MaxRetryAttempts = 2
};

builder.Services.AddAzureServiceBus(cfg => cfg
    .ConfigureAzureServiceBus(azureServiceBusConfig));
```

This method need a `AzureServiceBusConfiguration` instance to configure all the necessary properties to work with the Azure Service Bus.

At this time, no queues, topics, producers or consumers are configured and registered. This will be done in the next steps.

💡 I recommend you create a class called `Settings` and map the `appsettings.json` on this class, you can merge your personal `appsettings` config and also add the `AzureServiceBusFlow` properties that are required. For example:

```
{
    "Settings": {
```

```

    "AzureServiceBusConfigurationSettings": {
        "ConnectionString": "",
        "ServiceBusReceiveMode": "ReceiveAndDelete",
        "MaxAutoLockRenewalDurationInSeconds": "1800",
        "MaxConcurrentCalls": "10",
        "MaxRetryAttempts": "3"
    }
}
}

```

```

public class Settings
{
    public AzureServiceBusConfiguration AzureServiceBusConfigurationSettings { get; }
}

```

```

var applicationSettings = builder.Configuration.GetSection("Settings").Get<Settings>();

```

Next Steps: Create Messages, Producers, Consumers, Queues and Topics.

In the next step, we'll explain how to create Messages and Producers that will publish to a specific queue or topic in Azure Service Bus.

Messages

The **AzureServiceBusFlow** defines a standard approach for creating messages that will be published to the bus. This ensures that the **MessageHandler** receives the correct message type and processes it properly.



Creating Messages

To create these messages, you need to define a record or class that implements **IServiceBusMessage** interface from the **AzureServiceBusFlow** package.

```
public class ExampleCommand1 : IServiceBusMessage
{
    public string RoutingKey => ExampleMessage.Id.ToString();
    public DateTime CreatedDate => DateTime.UtcNow;
    public required ExampleMessage ExampleMessage { get; set; }
}

public class ExampleMessage
{
    public Guid Id { get; set; }
    public string? Cliente { get; set; }
    public decimal Valor { get; set; }
}
```

The **RoutingKey** and **CreatedDate** properties are used internally by **AzureServiceBusFlow**, while **ExampleMessage** represents the actual content of the message being sent.

The message above is just an example and the **Content** can be of any type, such as a class, record, struct, integer, string, IEnumerable...

To register which messages will be sent to the bus, we need to configure the **Producers** to publish the messages. [Check out the next page to know how to register it.](#)

Producers

To simplify the process of sending messages to Azure Service Bus, we created an abstraction that encapsulates the publishing logic.

This approach keeps the implementation consistent and allows any type of message to be sent using the same pattern.

Registering a Producer

Sending Messages to a Queue

To register a producer and configure which message it will send, use the `AddProducer()` method inside the `AddAzureServiceBus()` configuration method in `Program.cs`:

```
builder.Services.AddAzureServiceBus(cfg => cfg
    .ConfigureAzureServiceBus(azureServiceBusConfig)
    .AddProducer<ExampleCommand1>(p => p
        .EnsureQueueExists("command-queue-one")
        .WithCommandProducer()
        .ToQueue("command-queue-one"));
```

The code above registers a producer for the `ExampleCommand1` Message created earlier in this documentation ([ExampleCommand1](#)).

- `EnsureQueueExists()`: Ensures that the specified **Queue** exists in Azure Service Bus. If the **Queue** does not exist, it will be created automatically.
- `WithCommandProducer()`: Specifies that the producer is of type **CommandProducer**. Alternatively, you can configure an **EventProducer** or a **CommandProducer**, depending on the message type.
- `ToQueue()`: Defines the **Queue** where the message will be published.

Sending Messages to a Topic

If you want the producer to publish messages to a **Topic** instead of a Queue, you can use the methods `EnsureTopicExists()` and `ToTopic()` in the same configuration pattern:

```
builder.Services.AddAzureServiceBus(cfg => cfg
    .ConfigureAzureServiceBus(azureServiceBusConfig)
    .AddProducer<ExampleEvent1>(p => p
        .EnsureTopicExists("event-topic-one")
        .WithEventProducer()
        .ToTopic("event-topic-one"));
```


- **EnsureTopicExists()**: Ensures that the specified Topic exists in Azure Service Bus. If the Topic does not exist, it will be automatically created.
- **WithEventProducer()**: Specifies that the producer is of type **EventProducer**. Alternatively, you can configure an **CommandProducer** or a **EventProducer**, depending on the message type.
- **ToTopic()**: Defines the Topic where the message will be published.



Publishing Messages

Once the producer is configured, it can be used to publish messages to the bus. Here's an example of how to do it.

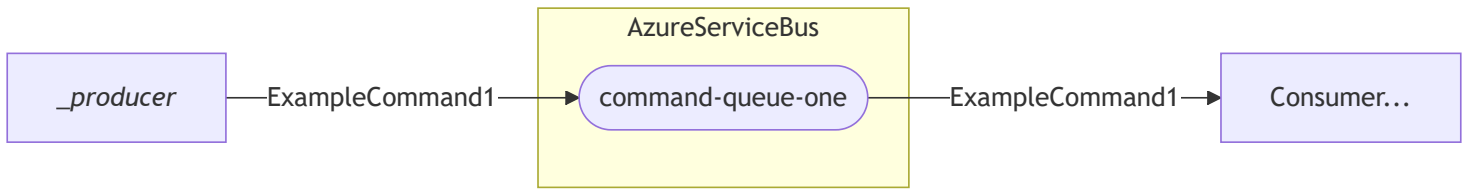
To publish a message as a **Command**, inject an instance of **ICommandProducer** and call **ProduceCommandAsync()**, passing the message as a parameter.

To publish a message as an **Event**, do the same using **IEventProducer** and call **ProduceEventAsync()**.




```
[Route("api/commands")]
[ApiController]
public class CommandController(ICommandProducer<ExampleCommand1> _producer) : ControllerBase
{
    [HttpPost("command-example-one")]
    public async Task<IActionResult> Example1(Cancellation token cancellationToken)
    {
        ExampleCommand1 command = new()
        {
            ExampleMessage = new ExampleMessage
            {
                Cliente = "jose",
                Id = Guid.NewGuid(),
                Valor = 1111
            }
        };

        await _producer.ProduceCommandAsync(command, cancellationToken);
        return Ok();
    }
}
```

The flowchart below shows the flow of sending the message created in the previous example.



Summary

-  **Message** created
-  **Producer** configured
-  **Queue** or **Topic** created and registered

Next step: [Create MessageHandler](#) to consume and process the message.

Consumers

AzureServiceBusFlow also simplify the creation of **Consumers** or **MessageHandlers** by implementing the **IMessageHandler<>** interface and passing the **Message** that will be consumed as the Type of the **Handler**.

This interface provides the **HandleAsync()** method that is used to process the received message.

Creating a Consumer

```
public class CommandExample1Handler : IMessageHandler<ExampleCommand1>
{
    public Task HandleAsync(ExampleCommand1 message, ServiceBusReceivedMessage rawMessage,
        CancellationToken cancellationToken)
    {
        return Task.CompletedTask;
    }
}
```

The MessageHandler above consume a Message of type ExampleCommand1, created earlier in this documentation ([ExampleCommand1](#)).

Registering Consumer

To register the **Consumer**, aka **MessageHandler**, we need to use the **AddConsumer()** extension method in **AddAzureServiceBus()** configuration method in **Program.cs**.

the **AddConsumer()** method requires a Action of type **ServiceBusConsumerConfigurationBuilder** to configure the Consumer using this methods:

- **FromQueue()**: Defines the Queue this Consumer will be receiving messages.
- **FromTopic()**: Defines the Topic this Consumer will be receiving messages.
- **EnsureSubscriptionExists()**: Ensure that the subscription passed exists in AzureServiceBus, creates it if doesn't exists.
- **AddHandler<TMessage, TMessageHandler>()**: Defines the **Consumer / MessageHandler** that is goint to consume a specific Message from a Queue or a Topic. More than one Consumer to the same **Message, Queue or Topic** can be defined in the same **AddConsumer()**

This example shows the full configuration for:

- Registering a **Message**
- Creating a **Producer** for it
- Publishing in a specific **Queue**
- Configuring a **Consumer / MessageHandler** for the **Message**
- Consuming this **Message** from the same **Queue**.

```
builder.Services.AddAzureServiceBus(cfg => cfg
    .ConfigureAzureServiceBus(azureServiceBusConfig)
    .AddProducer<ExampleCommand1>(p => p
        .EnsureQueueExists("command-queue-one")
        .WithCommandProducer()
        .ToQueue("command-queue-one"))
    .AddConsumer(c => c
        .FromQueue("command-queue-one")
        .AddHandler<ExampleCommand1, CommandExample1Handler>())
    );
```

The complete flow of sending a message and consuming it with the **MessageHandler** we just created is shown below.

