📦 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.

This package contains all the necessary components to configure Producers / Consumers, register Queues / Topics and publish Messagens / 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.

🧩 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.

```
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. This configuration class is provided by **AzureServiceBusFlow**.

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

> 💡 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 will 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.

To create these messages, you need to define a record or class that implements the `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

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

📤 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 using a **CommandProducer**, inject an instance of `ICommandProducer` and call `ProduceCommandAsync()`, passing the message as a parameter.

```
[Route("api/commands")]
[ApiController]
public class CommandController(ICommandProducer<ExampleCommand1> _producerOne,
ICommandProducer<ExampleCommand2> _producerTwo) : ControllerBase
{
    [HttpPost("command-example-one")]
    public async Task<IActionResult> Example1(CancellationToken cancellationToken)
    {
        ExampleCommand1 command = new()
        {
            ExampleMessage = new ExampleMessage
            {
                Cliente = "jose",
                Id = Guid.NewGuid(),
                Valor = 1111
            }
        };

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

🧭 Summary

- 💭 **Message** created
- 📤 **Producer** configured
- 📫 **Queue** created and registred

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