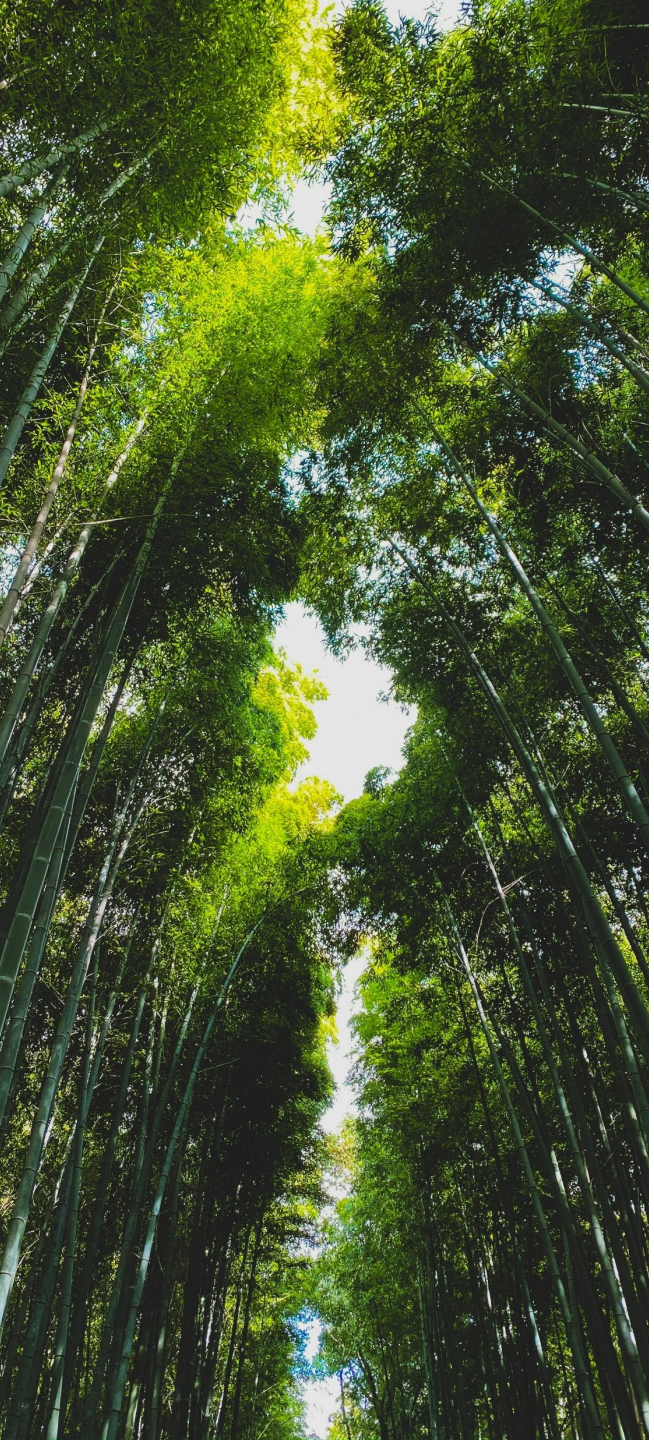
Code.Hub

The first Hub for Developers

Microservices With .NET
Asynchronous Communication with RabbitMQ

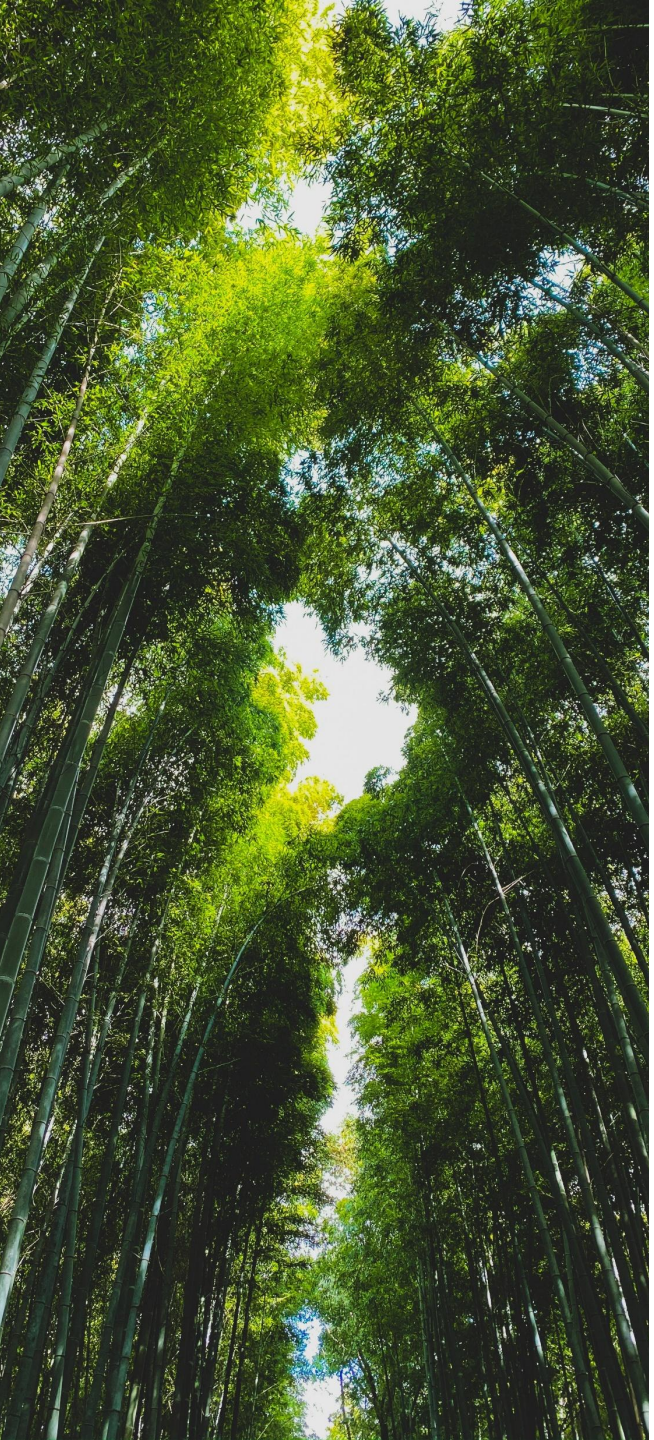


Introduction

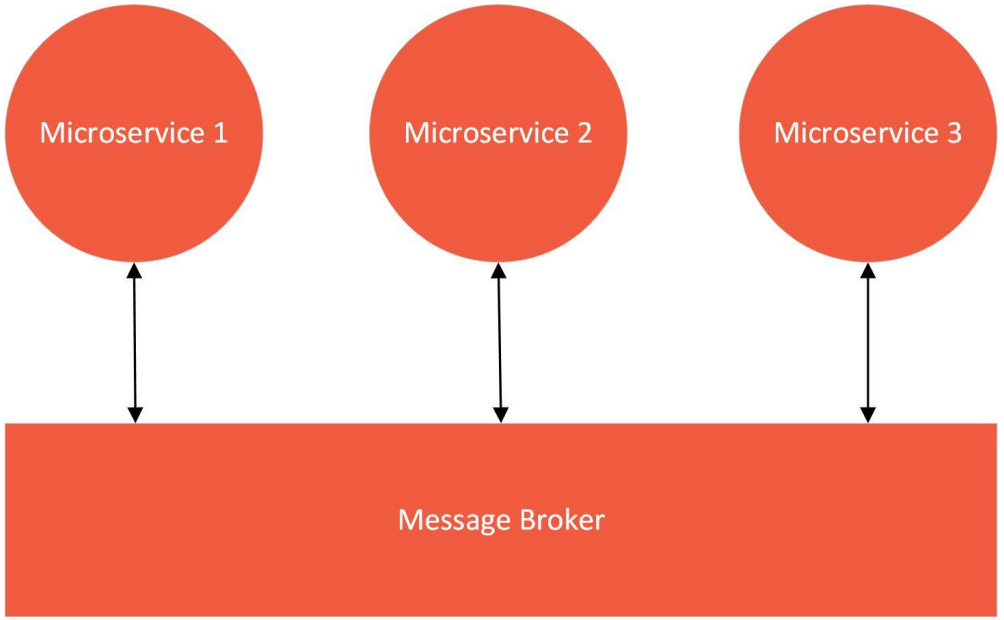
- **Asynchronous Communication** enables microservices to communicate through **messages** traded via a central broker.

- Unlike synchronous communication which follows a **request-reply** pattern, in asynchronous communication, publishers produce messages without expecting an immediate response from the receiver.

- This **significantly** improves message throughput and helps avoid the creation of bottlenecks in the system.



Message Brokers



- **Asynchronous communication is usually facilitated through a Message Broker,** a server application that acts as an intermediary to which messages can be sent and received from.

Code.Hub

RabbitMQ

- **RabbitMQ** is an open-source message broker maintained by *VMWare Tanzu Labs* and available under the *Mozilla Public License*.

- It is written in Erlang, but **clients** are available for a **wide variety of platforms**, including C#, Java, Python, PHP, C/C++, and JavaScript.

- It provides a **rich feature set** such as reliability, performance, and flexible routing.

Installing RabbitMQ (Docker)

- RabbitMQ can be installed through the Docker image, or locally.

- To create a Docker container:
  - Pull the *RabbitMQ:Management* image with `docker image pull rabbitmq:management`

  - Create a container from the image with `docker container run -p 15672:15672 -p 5672:5672 -d rabbitmq:management`

Installing RabbitMQ (Local)

- To install RabbitMQ locally (Windows):
  - First download and install the latest 64-bit Erlang **binary** from https://www.erlang.org/downloads
  - Then download and install the RabitMQ **exe installer** from: https://github.com/rabbitmq/rabbitmq-server/releases
  - Finally, enable the management interface by running the following on a terminal:

```
rabbitmq-plugins enable rabbitmq_management
```

Code.Hub

RabbitMQ Basics

Code.Hub

1. Accessing RabbitMQ
2. Exchanges and Queues
3. Role of Exchanges
4. Exchange Attributes
5. Queue Attributes
6. Exchange Types
7. Message States

RabbitMQ Basics

Code.Hub

Accessing RabbitMQ

- RabbitMQ is based on the AMQP 0-9-1 protocol (although it can also support other protocols via plugins).
- The default port **for the protocol** in RabbitMQ is port `5672`. The **management interface** is available through port `15672`.
- The default credentials are `guest` for **both username and password** (restricted to local host access).

Exchanges and Queues

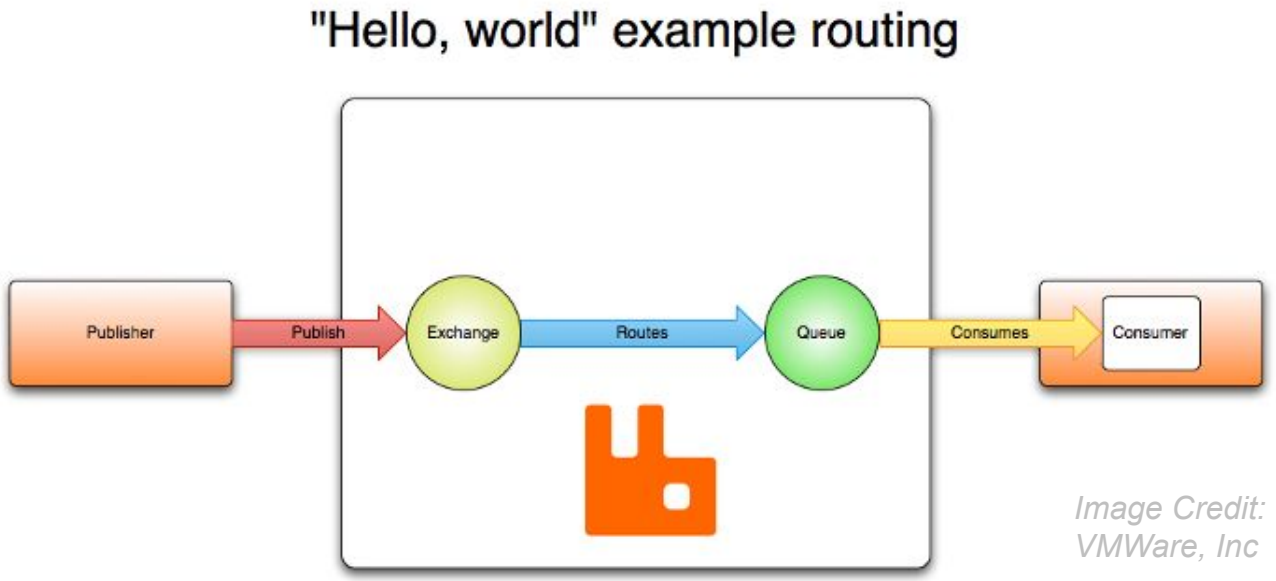


Image Credit: VMWare, Inc

- In the AMPQ protocol, **publishers** create messages that are **pushed to exchanges and stored in queues**. Consumers (receiving applications) can then retrieve messages from the queue.

Code.Hub

Role of Exchanges

- Exchanges are responsible for routing messages to the appropriate queue.
- When needed, messages include a **routing key** based on which the exchange can discern to which queue(s) to push the messages.
- All queues are binded to an exchange. If a queue is not **explicitly** binded to a queue, it is automatically binded to the default exchange.

Exchange Attributes

- Exchanges are declared with several attributes that control their behavior:

Attribute	Function
Name	The name of the exchange.
Durable	If true, the exchange will be stored in the disk drive, and can survive a broker restart.
Auto-delete	If true, the exchange will automatically be deleted when all queues are unbounded from it.
Arguments	Optional arguments used mostly by plugins.

Queue Attributes

- Queues include the following attributes:

Attribute	Function
Name	The name of the queue.
Durable	If true, the queue will be stored in the disk drive, and can survive a broker restart.
Exclusive	If true, the queue can only be used by one connection and will be deleted when the connection closes.
Auto-delete	If true, the queue will automatically be deleted when all consumers unsubscribe.
Arguments	Optional arguments used mostly by plugins.

Exchange Types

Direct	Messages are routed to the **matching exchange** according to the routing key.
Fanout	Messages are always routed to **all queues** binded to the exchange.
Topic	Messages are routed to **one or multiple** queues according to a **structured** routing key.
Headers	Similar to Topic, but routing is based on a **collection of headers** rather than the routing key.

Code.Hub

Message States

Ready

Messages that reach a queue are initially set to read. They will be delivered when a subscriber accesses the queue.

Unacknowledged

Messages that have been delivered to a subscriber but have not been acknowledged as processed are marked us *Unacked*. If the connection to the subscriber is lost, the messages will be moved back to the *Ready* state.

Acknowledged

Messages marked as *Acked* are removed from the queue and are considered by the broker as safe to delete.

Code.Hub

Fanout Exchange

Code.Hub

1. Fanout Exchange Introduction
2. Importing the Client Library
3. Creating an AMQP Connection
4. Declaring a Fanout Exhange
5. Declaring and Binding a Queue
6. Creating a Publisher
7. Publising in Fanout Exchanges
8. Consuming Messages
9. Creating an Eventing Consumer
10. Creating a Subscriber

Fanout Exchange

Introduction

Fanout exchange routing

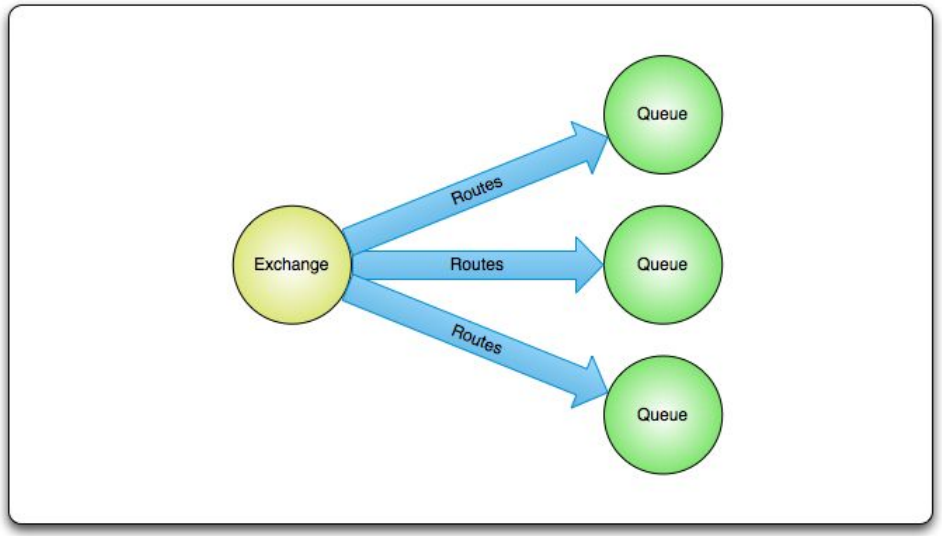


Image Credit: VMWare, Inc

- A **Fanout Exchange** ignores the routing key and rather delivers a copy of the message to **all queues** binded to the exchange.
- It is frequently used to broadcast **information of wide interest to a large number of recipients** such as news for aggregators or scores for a leaderboard.

Important Client Library Types

- The client provides three important types defined in the `RabitMQ.Client` namespace:
  - The `IConnection` type represents a connection to the AMQP 0-9-1 broker server.
  - The `IConnectionFactory` type is used to create instances of `IConnection`.
  - The `IModel` type represents represents an AMQP channel through which most protocol operations can be accessed.

Declaring Exchanges/Queues

- Exchanges and queues can be declared through the management plugin or directly in the code.
- Declaration operators are **transient**. If the exchange or queue already exists, it will be used.
- In order to declare an exchange or queue, a connection must be first established.

Creating an AMPQ Connection

The displayed code presents the creation of a connection.

- First a `ConnectionFactory` is created and UserName, Password as well as HostName provided **through its properties**.
- Then, an **`IConnection` instance is created from the factory** and an **`IModel` instance** is created based on the `IConnection` instance.

```
ConnectionFactory factory = new()
        {
            UserName = "guest",
            Password = "guest",
            HostName = "localhost"
        };

using IConnection connection = factory.CreateConnection();
using IModel channel = connection.CreateModel();
```

Code.Hub

Declaring an Exchange

- Through an `IModel` instance, an exchange or queue can be declared.
- When declaring an exchange, both a **name string** as well as a **type** (through the `ExchangeType` enum) must be specified.
- channel.ExchangeDeclare("CS1225_FanoutEx", ExchangeType.Fanout);

Code.Hub

Declaring a Queue

- When declaring a queue the following parameters are given:
  - *queue* (Queue name, `Boolean`)
  - *durable* (`Boolean`)
  - *exclusive* (`Boolean`)
  - *autoDelete* (`Boolean`)
  - *arguments* (`IDictionaty<string, object>`)

- ```
  channel.QueueDeclare("FanoutEx_Q1",
  false, false, false, null);
  ```
  ```

Binding a Queue

- With both the exchange and the queue declared, the queue can now be binded to the exchange. This can be achived with `IModel`'s `QueueBind()` method. It accepts three arguemtns:
  - *queue* (Queue name, `string`)
  - *exchange* (Exchange name, `string`)
  - *routingKey* (`string`)

- channel.QueueBind("FanoutEx_Q1", "CS1225_FanoutEx", "");

Creating a Publisher

- Messages can be published using `IModel`'s `BasicPublish()` extension method. It accepts four arguments:
  - *exchange* (Exchange name, `string`)
  - *routingKey* (`string`)
  - *basicProperties* (`IBasicProperties`)
  - *body* (`Byte[]`)

- ```
  string message = "Hello World";
  var body = Encoding.UTF8.GetBytes(message);
  channel.BasicPublish("CS1225_FanoutEx", "",
  null, body);
  ```

Code.Hub
  ```

```
static void Main(string[] args)
        {
            ConnectionFactory factory = new()
            {
                UserName = "guest",
                Password = "guest",
                HostName = "localhost"
            };

            using IConnection connection = factory.CreateConnection();
            using IModel channel = connection.CreateModel();

            channel.ExchangeDeclare("CS1225_FanoutEx",
ExchangeType.Fanout);
            channel.QueueDeclare("FanoutEx_Q1", false, false, false,null);

            channel.QueueBind(« FanoutEx_Q1", "CS1225_FanoutEx", "");

            string message = "Hello World";
            var body = Encoding.UTF8.GetBytes(message);

            channel.BasicPublish("CS1225_FanoutEx", "", null, body);

            Console.WriteLine(" FanoutEx.Publisher Sent {0}", message);
}
```

Code.Hub

Publishing in Fanout Exchanges

Received messages can be seen in the queues section of the management portal. In fanout exchanges, the routing key is always ignored and a copy of all messages is delivered to all binded queues.

```
channel.ExchangeDeclare("CS1225_FanoutEx", ExchangeType.Fanout);
channel.QueueDeclare("FanoutEx_Q1",false, false, false, null);
channel.QueueDeclare("FanoutEx_Q2",false, false, false, null);


channel.QueueBind("FanoutEx_Q1", "CS1225_FanoutEx", "");
channel.QueueBind("FanoutEx_Q2", "CS1225_FanoutEx", "q2");
//^ The routing key is ignored in Fanout Exchanges


string message = "Hello World";
var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish("CS1225_FanoutEx", "", null, body);
channel.BasicPublish("CS1225_FanoutEx", "q2", null, body);
//^ Both messages are received by both queues
```

Code.Hub

Consuming Messages

- Messages can be received using a **Consumer**. The consumer maintains an active connection to the broker, which can **notify the consumer** when new messages are available.

- Using consumers, it not necessary for the subscriber to actively check the server for messages (unlike AMQP's `get` method which necessitates an exchange).

- This significantly improves performance and reduces load.

Consuming Messages

- Messages can be received using a **Consumer**. The consumer maintains an active connection to the broker, which can **notify the consumer** when new messages are available.

- Using consumers, it not necessary for the subscriber to actively check the server for messages (unlike AMQP's `get` method which necessitates an exchange).

- This significantly improves performance. In a **new project** named `FanoutEx.Subsriber`, messages can be consumed.

Eventing Consumer

The `EventingBasicConsumer` class allows the instantiation of a **message consumer** that will raise **events** whenever a message is received.

Methods can then be added to the consumer's `Received` **delegate** in order to **handle any incoming messages**.

```
var consumer = new EventingBasicConsumer(channel);

    consumer.Received += (model, ea) =>

    {

    var body = ea.Body.ToArray();

    var message = Encoding.UTF8.GetString(body);

    Console.WriteLine($"FanoutEx.Subscriber received
{message}");

     };
```

Code.Hub

Starting the Consumer

- The consumer is now defined but is **not yet started** and listening for messages.

- A consumer can be started by using the `BasicConsume()` method, which accepts three arguments:
  - `queue` (Name of the queue, String)
  - `autoAck` (If true an ack is automatically sent when the message is received, Boolean)
  - `consumer` (The consumer instance, IBasicConsumer)

Producing a Subscriber

- When `BasicConsume()` is called, the consumer will start and the application will continue to listen for messages until it terminates or until `BasicCanel()` is called.

- A `ReadLine()` statement can be used to prevent the application from automatically exiting.

- ```
  channel.BasicConsume("FanoutEx_Q1",  true,
  consumer);
  Console.WriteLine("Listening for messages.
  Press ENTER anytime to quit...");
  Console.ReadLine();
  ```

Code.Hub
  ```

```
static void Main(string[] args)
{
    ConnectionFactory factory = new()
    {
        UserName = "guest", Password = "guest", HostName = "localhost"
    };

    using IConnection connection = factory.CreateConnection();
    using IModel channel = connection.CreateModel();

    channel.ExchangeDeclare("CS1225_FanoutEx", ExchangeType.Fanout);
    channel.QueueDeclare("FanoutEx_Q1", false, false, false, null);
    channel.QueueBind("FanoutEx_Q1", "CS1225_FanoutEx", "");

    var consumer = new EventingBasicConsumer(channel);
    consumer.Received += (model, ea) =>
    {
        var body = ea.Body.ToArray();
        var message = Encoding.UTF8.GetString(body);
        Console.WriteLine($"FanoutEx.Subscriber received {message}");
    };

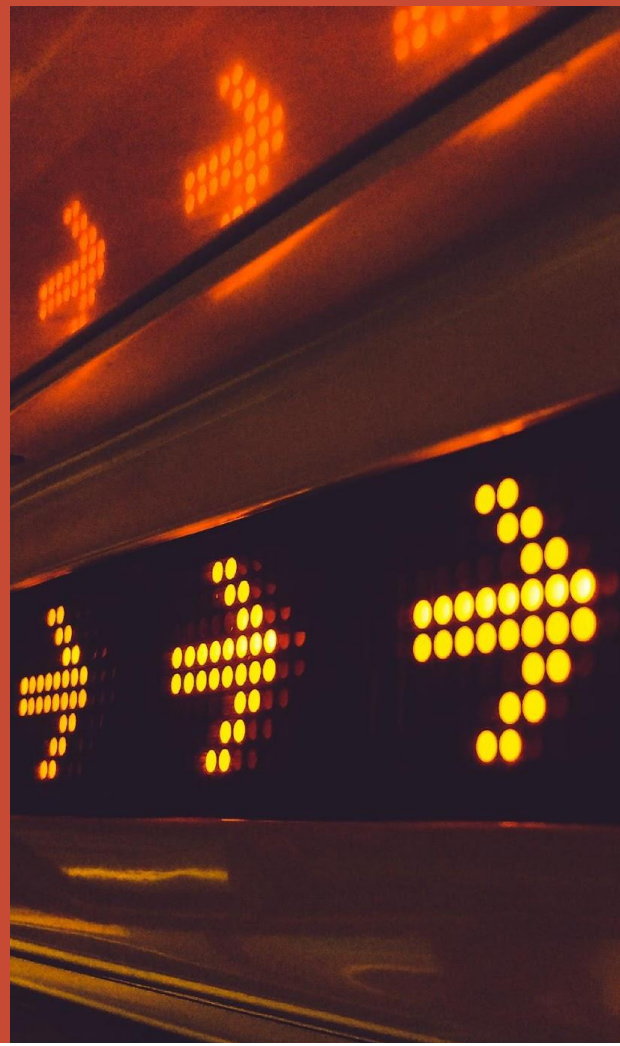
    channel.BasicConsume("FanoutEx_Q1",  true, consumer);

    Console.WriteLine("Listening for messages. Press ENTER anytime to quit...");
    Console.ReadLine();
}
```

Direct Exchange

Code.Hub

1. Direct Exchange Introduction
2. Binding Queues With a Routing Key
3. Publishing to a Direct Exchange
4. Subscribing to a Direct Exchange
5. Configuring Persistence
6. Declaring a Durable Queue
7. Sending Persistent Messages



Direct Exchange

Code.Hub

Introduction

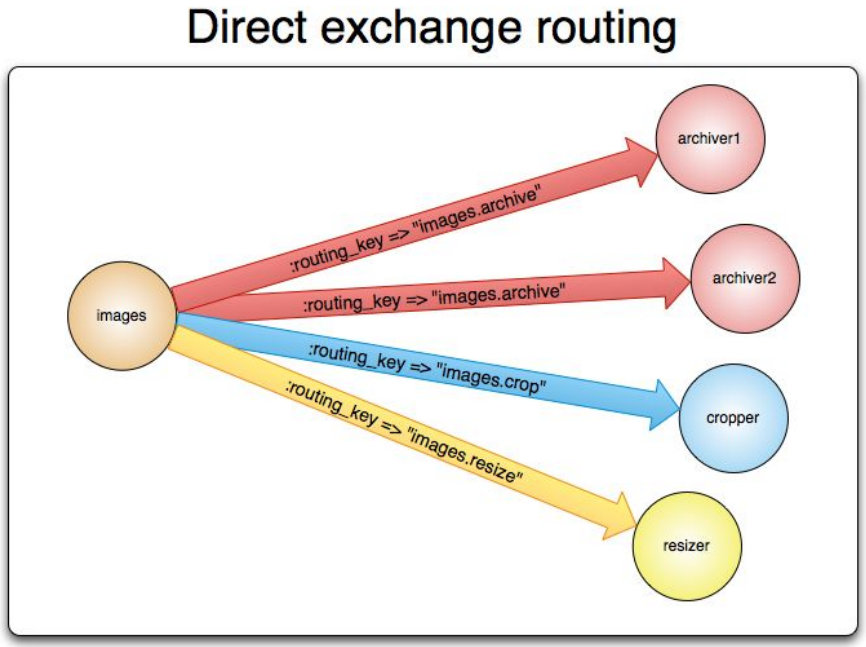


Image Credit: VMWare, Inc

- A **Direct Exchange** routes messages to one or more queues based on the **routing key**.
- Queues bind to the exchange with a routing key. Messages arriving at the exchange will only be routed to the queue(s) with a matching routing key.

Code.Hub

Binding With Routing Key

- Binding a queue to a Direct Exchange, is done in the same manner as in a Fanout Exchange, but a routing key is specified:

```
ConnectionFactory factory = new()
{UserName = "guest", Password = "guest“, HostName = "localhost"};

using IConnection connection = factory.CreateConnection();
using IModel channel = connection.CreateModel();
channel.ExchangeDeclare("CS1225_DirectEx", ExchangeType.Direct);

channel.QueueDeclare(queue: "Green", false, false, false,null);
channel.QueueBind("Green", "CS1225_DirectEx", "green");

channel.QueueDeclare(queue: "Red", false, false, false, null);
channel.QueueBind("Red", "CS1225_DirectEx", "red");
```

Code.Hub

Publishing

- When publishing to a direct exchange the routing key determines which queue(s) will receive the message.
- The following message will only be pushed to any queues banded to the *CS1225_DirectEx* exchange with the routing key *green*:

```
string message = "Hello World to Green!!!";
var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish( "CS1225_DirectEx", "green",
null, body);
```

```
static void Main(string[] args)
        {
            ConnectionFactory factory = new()
            {
                UserName = "guest",
                Password = "guest",
                HostName = "localhost"
            };

            using IConnection connection = factory.CreateConnection();
            using IModel channel = connection.CreateModel();

            channel.ExchangeDeclare("CS1225_DirectEx", ExchangeType.Direct);

            channel.QueueDeclare(queue: "Green", false, false, false,null);
            channel.QueueBind("Green", "CS1225_DirectEx", "green");

            channel.QueueDeclare(queue: "Red", false, false, false, null);
            channel.QueueBind("Red", "CS1225_DirectEx", "red");

            string message = "Hello World to Green!!!";
            var body = Encoding.UTF8.GetBytes(message);

            channel.BasicPublish( "CS1225_DirectEx", "green", null, body);

            Console.WriteLine($"DirectEx.Publisher sent {message}");
        }
```

Code.Hub

Receiving

- Receiving messages from a *Direct* exchange is similar to receiving from a *Fanout.*
- The queue is specified using the queue name, not the routing key. Remember that values are case sensitive.

```
var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) =>
{
    var body = ea.Body.ToArray();
    var message = Encoding.UTF8.GetString(body);
    Console.WriteLine(" [x] Received {0}", message);
};

channel.BasicConsume(queue: "Green", true, consumer);
```

Code.Hub

```
static void Main(string[] args)
        {
            ConnectionFactory factory = new()
            {
                UserName = "guest",
                Password = "guest",
                HostName = "localhost"
            };

            using IConnection connection =
factory.CreateConnection();
            using IModel channel = connection.CreateModel();

            channel.ExchangeDeclare("CS1225_DirectEx",
ExchangeType.Direct);
            channel.QueueDeclare( "green", false, false, false,
null);

            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (model, ea) =>
            {
                var body = ea.Body.ToArray();
                var message = Encoding.UTF8.GetString(body);
                Console.WriteLine(" [x] Received {0}", message);
            };

            channel.BasicConsume(queue: "Green", true, consumer);

            Console.WriteLine("Listening for messages. Press ENTER
anytime to quit...");
            Console.ReadLine();
        }
```

Code.Hub

Durable Queue

- Queues declared as `durable` will be recorded to disk and will **survive a broker restart**.
- The existing *green* queue can be changed to be declared as durable, after deleting the existing instance.
- Trying to **re-declare an existing queue with different properties** will produce an **exception**.

```
channel.QueueDeclare("green", true, false,
false, null);
```

Code.Hub

Implementing Persistence

- In order for messages to be **persisted** and survive a broker restart, their properties must include the property `delivery_mode` with the value of `2`.

- In order to send a message that will persist a broker restart:
  - It must be sent on a **durable** queue.
  - An `IBasicProperties` object with the Persistent property set to **`true`** must be `created`.
  - The `IBasicProperties` object must be passed as the `properties` argument to the publishing method.

Sending Persistent Messages

- Properties are exposed through an *IBasicProperties* object, which can be instantiated with the `Imodel.CreateBasicProperties()` method.

```
channel.ExchangeDeclare("CS1225_DirectEx",
ExchangeType.Direct);
var properties = channel.CreateBasicProperties();
properties.Persistent = true;
```

- When **creating the publisher** the **`IBasicProperties`** object must be passed to the `BasicPublish()` method.

```
channel.BasicPublish( "CS1225_DirectEx", "green",
properties, body);
```

Code.Hub


```
static void Main(string[] args)
{
    ConnectionFactory factory = new()
    {
        UserName = "guest",
        Password = "guest",
        HostName = "localhost"
    };

    using IConnection connection = factory.CreateConnection();
    using IModel channel = connection.CreateModel();
    var properties = channel.CreateBasicProperties();
    properties.Persistent = true;

    channel.ExchangeDeclare("CS1225_DirectEx", ExchangeType.Direct);

    channel.QueueDeclare(queue: "Green", true, false, false,null);
    channel.QueueBind("Green", "CS1225_DirectEx", "green");

    channel.QueueDeclare(queue: "Red", true, false, false, null);
    channel.QueueBind("Red", "CS1225_DirectEx", "red");

    string message = "Hello World to Green!!!";
    var body = Encoding.UTF8.GetBytes(message);

    channel.BasicPublish( "CS1225_DirectEx", "green", properties, body);

    Console.WriteLine($"DirectEx.Publisher sent {message}");
}
```


```
static void Main(string[] args)
{
    ConnectionFactory factory = new()
    {
        UserName = "guest",
        Password = "guest",
        HostName = "localhost"
    };

    using IConnection connection = factory.CreateConnection();
    using IModel channel = connection.CreateModel();

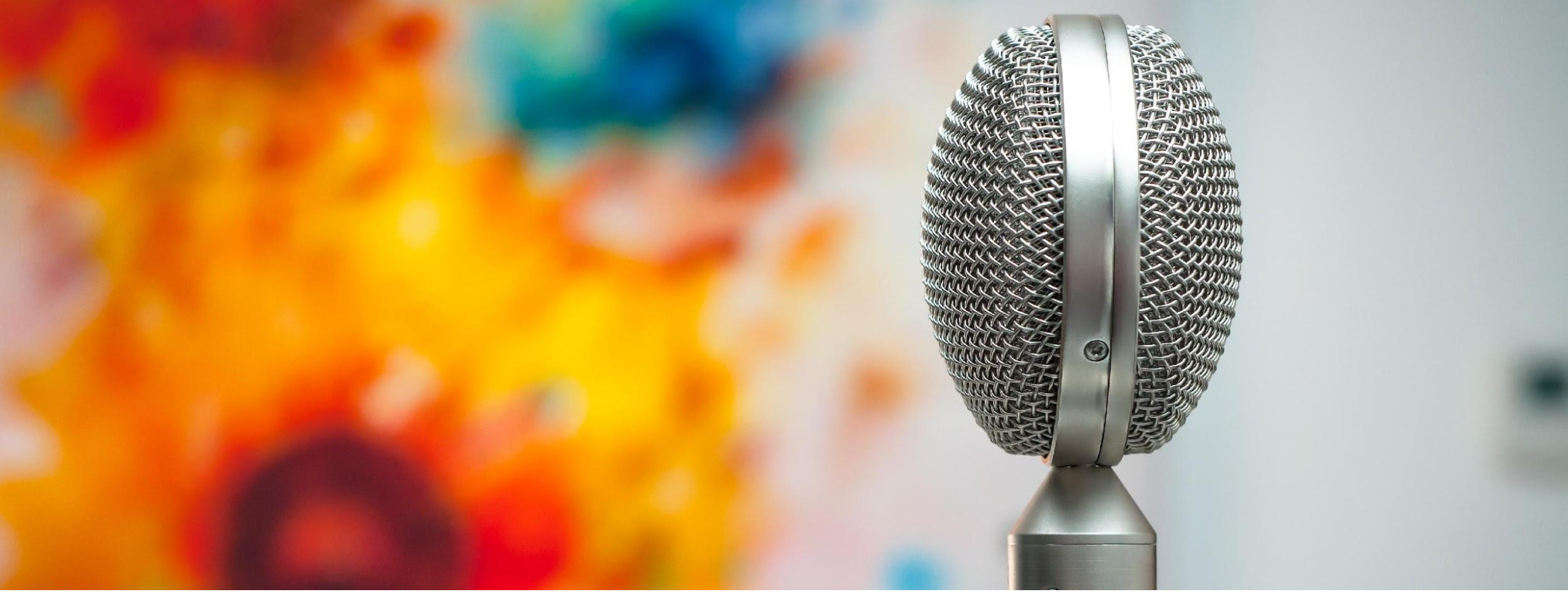
    channel.ExchangeDeclare("CS1225_DirectEx",  ExchangeType.Direct);
    channel.QueueDeclare( "Green", true, false, false, null);

    var consumer = new EventingBasicConsumer(channel);
    consumer.Received += (model, ea) =>
    {
        var body = ea.Body.ToArray();
        var message = Encoding.UTF8.GetString(body);
        Console.WriteLine(" [x] Received {0}", message);
    };

    channel.BasicConsume(queue: "Green", true, consumer);

    Console.WriteLine("Listening for messages. Press ENTER anytime to
quit...");
    Console.ReadLine();
}
```

Code.Hub



Topic Exchange

Code.Hub

1. Topic Exchange Introduction
2. Defining the Example Queues
3. Publishing to a Topic Exchange
4. Subscribing to a Topic Exchange

Topic Exchange

Code.Hub

Introduction

- A **Topic Exchange** also routes messages to one or more queues based on the **routing key**.
- The key format is specific. Consisting of any number of words separated by dots.
- The key can contain the special * and # symbols, to symbolize any single word or any number of words, at any part of the key.

Example Queues

- For this example, four queues will be created:
  - The *BikesSales* (`CRM.Sales.Bikes`) queue will receive only messages for **Bike sales**.
  - The *Clothing* (`CRM.Sales.Clothing`) queue will receive only messages for **clothing sales**.
  - The *AllSales* (`CRM.Sales.*`) queue will receive only messages for **all sales**.
  - The *AllCRM* (`CRM.#`) queue will receive **all** CRM related messages.

Declaring the Queues

The Publisher will declare a series of queues, as per the presented requirements:

```
channel.ExchangeDeclare("CS1225_TopicEx", ExchangeType.Topic);
channel.QueueDeclare(queue: "BikesSales", false, false, false, null);
channel.QueueBind("BikesSales", "CS1225_TopicEx", "CRM.Sales.Bikes");
channel.QueueDeclare(queue: "ClothingSales", false, false, false, null);
channel.QueueBind("ClothingSales", "CS1225_TopicEx", "CRM.Sales.Clothing");

channel.QueueDeclare(queue: "AllSales", false, false, false, null);
channel.QueueBind("AllSales", "CS1225_TopicEx", "CRM.Sales.*");
channel.QueueDeclare(queue: "AllCRM", false, false, false, null);
channel.QueueBind("AllCRM", "CS1225_TopicEx", "CRM.#");
```

Understanding the Routing

- The first and second queues (`CRM.Sales.`Bikes and `CRM.Sales.Clothing`) will only receive messages that **perfectly match their routing keys**, similarly to direct exchanges.

- The third quue will receive messages from both of the above queues (`CRM.Sales.*`). It would also receive messages from any other queue who matches the routing key format, **regardless of the value of the last word**.

- The foruth queue (`CRM.#`) will receive messages from both queues as well, as it would from **any other queue whose routing key starts with *CRM***.

Publishing a Message

- A message can be published to the exchange using an appropriate routing key. The presented message will be sent to all queues except `CRM.Sales.Clothing`.

```
string message = "A Bike Sales Has Occurred!!!";
var body = Encoding.UTF8.GetBytes(message);

channel.BasicPublish("CS1225_TopicEx",
"CRM.Sales.Bikes", null, body);

Console.WriteLine($"TopicEx.Publisher sent
messages to topic exchanges.");
```

```
static void Main(string[] args)
{
    ConnectionFactory factory = new()
    {
        UserName = "guest",
        Password = "guest",
        HostName = "localhost"
    };

    using IConnection connection = factory.CreateConnection();
    using IModel channel = connection.CreateModel();
    channel.ExchangeDeclare("CS1225_TopicEx", ExchangeType.Topic);

    channel.QueueDeclare(queue: "BikesSales", false, false, false, null);
    channel.QueueBind("BikesSales", "CS1225_TopicEx", "CRM.Sales.Bikes");
    channel.QueueDeclare(queue: "ClothingSales", false, false, false, null);
    channel.QueueBind("ClothingSales", "CS1225_TopicEx", "CRM.Sales.Clothing");

    channel.QueueDeclare(queue: "AllSales", false, false, false, null);
    channel.QueueBind("AllSales", "CS1225_TopicEx", "CRM.Sales.*");
    channel.QueueDeclare(queue: "AllCRM", false, false, false, null);
    channel.QueueBind("AllCRM", "CS1225_TopicEx", "CRM.#");


    string message = "A Bike Sales Has Occurred!!!";
    var body = Encoding.UTF8.GetBytes(message);

    channel.BasicPublish("CS1225_TopicEx", "CRM.Sales.Bikes", null, body);
    Console.WriteLine($"TopicEx.Publisher sent messages to topic exchanges.");
}
```

Receiving a Message

- A message can be received by a consumer. The same message may be received by multiple queues.

```
var consumerAll = new EventingBasicConsumer(channel);
consumerAll.Received += (model, ea) =>
{
    var body = ea.Body.ToArray();
    var message = Encoding.UTF8.GetString(body);
    Console.WriteLine($"TopixEx.Subscriber received '{message}' from AllSales Queue");
};
channel.BasicConsume("AllSales", true, consumerAll);

var consumerBikes = new EventingBasicConsumer(channel);
consumerBikes.Received += (model, ea) =>
{
    var body = ea.Body.ToArray();
    var message = Encoding.UTF8.GetString(body);
    Console.WriteLine($"TopixEx.Subscriber received '{message}' from BikesSales Queue");
};
channel.BasicConsume("BikesSales", true, consumerBikes);
```

Code.Hub

```
static void Main(string[] args)
{
    ConnectionFactory factory = new()
    {
        UserName = "guest",
        Password = "guest",
        HostName = "localhost"
    };

    using IConnection connection = factory.CreateConnection();
    using IModel channel = connection.CreateModel();
    channel.ExchangeDeclare("CS1225_TopicEx", ExchangeType.Topic);

    channel.QueueDeclare(queue: "BikesSales", false, false, false, null);
    channel.QueueBind("BikesSales", "CS1225_TopicEx", "CRM.Sales.Bikes");
    channel.QueueDeclare(queue: "ClothingSales", false, false, false, null);
    channel.QueueBind("ClothingSales", "CS1225_TopicEx", "CRM.Sales.Clothing");

    channel.QueueDeclare(queue: "AllSales", false, false, false, null);
    channel.QueueBind("AllSales", "CS1225_TopicEx", "CRM.Sales.*");
    channel.QueueDeclare(queue: "AllCRM", false, false, false, null);
    channel.QueueBind("AllCRM", "CS1225_TopicEx", "CRM.#");

    var consumerAll = new EventingBasicConsumer(channel);
    consumerAll.Received += (model, ea) =>
    {
        var body = ea.Body.ToArray();
        var message = Encoding.UTF8.GetString(body);
        Console.WriteLine($"TopixEx.Subscriber received '{message}' from          AllSales Queue");
    };
    channel.BasicConsume("AllSales", true, consumerAll);

    var consumerBikes = new EventingBasicConsumer(channel);
    consumerBikes.Received += (model, ea) =>
    {
        var body = ea.Body.ToArray();
        var message = Encoding.UTF8.GetString(body);
        Console.WriteLine($"TopixEx.Subscriber received '{message}' from          BikesSales Queue");
    };
    channel.BasicConsume("BikesSales", true, consumerBikes);

    var consumerClothing = new EventingBasicConsumer(channel);
    consumerClothing.Received += (model, ea) =>
    {
        var body = ea.Body.ToArray();
        var message = Encoding.UTF8.GetString(body);
        Console.WriteLine($"TopixEx.Subscriber received '{message}' from          ClothingSales Queue");
    };
    channel.BasicConsume("ClothingSales", true, consumerBikes);

    Console.WriteLine("Listening for messages. Press ENTER anytime to quit...");
    Console.ReadLine();
}
```

Code.Hub

Headers Exchange

Code.Hub

1. Headers Exchange Introduction
2. Binding a Queue With Headers
3. Setting Message Headers
4. Publishing a Message With Headers
5. Subscribing to a Headers Exchange

Topic Exchange

Code.Hub

Introduction

- A **Headers Exchange** is similar to topic and also routes each message to one or more queue, but instead of routing based on the routing key, it routes based on **headers**.
- Headers are specified for a **queue** when it is **bound** to the the headers exchange.
- They specify which messages the queue will receive.
- Headers are also specified when a message is sent.

Binding a Queue

- After declaring an **exchange of headers** type, queues can be bound with headers that specify which messages they will receive.
- The special **`x-match`** header can further specify which messages the queue will receive:
  - **`all`** signifies that the queue should receive messages only when the values **in all headers** match the specified.
  - **any** signifies that queue should match messages when the value in **at least one header** matches the specified.

Binding a Queue

```
ConnectionFactory factory = new()
{ UserName = "guest", Password = "guest", HostName = "localhost"};
using IConnection connection = factory.CreateConnection();
using IModel channel = connection.CreateModel();
channel.ExchangeDeclare("CS1225_HeadersEx", ExchangeType.Headers);

Dictionary<string, object> bindHeaders = new()
{
    { "x-match", "all" },
    { "shape", "square" },
    { "color", "black" }
};
channel.QueueDeclare(queue: "BlackSquares", false, false, false, null);
channel.QueueBind("BlackSquares", "CS1225_HeadersEx", "", bindHeaders);
```

Setting Message Headers

- When publishing a message the headers can be specified within the **IBasicProperpties properties** object, passed in the parmaters of the `BasicPublish()` method.
- The properties object can be initialized with the `IModel.CreateBasicProperties()` method.
- Headers can be specified from IBasicProperties' *`Headers`* field

Publishing a Message

```
string message = "A Black Square has been generated!";
var body = Encoding.UTF8.GetBytes(message);

IBasicProperties props = channel.CreateBasicProperties();
Dictionary<string, object> msgHeaders = new()
{
    { "shape", "square" },
    { "color", "black" }
};
props.Headers = msgHeaders;

channel.BasicPublish("CS1225_HeadersEx", "", props, body);
Console.WriteLine($"HeadersEx.Publisher sent message to the headers exchange.");
```

Code.Hub

```
static void Main(string[] args)
{
    ConnectionFactory factory = new()
    {
        UserName = "guest",
        Password = "guest",
        HostName = "localhost"
    };

    using IConnection connection = factory.CreateConnection();
    using IModel channel = connection.CreateModel();
    channel.ExchangeDeclare("CS1225_TopicEx", ExchangeType.Topic);

    channel.QueueDeclare(queue: "BikesSales", false, false, false, null);
    channel.QueueBind("BikesSales", "CS1225_TopicEx", "CRM.Sales.Bikes");
    channel.QueueDeclare(queue: "ClothingSales", false, false, false, null);
    channel.QueueBind("ClothingSales", "CS1225_TopicEx", "CRM.Sales.Clothing");

    channel.QueueDeclare(queue: "AllSales", false, false, false, null);
    channel.QueueBind("AllSales", "CS1225_TopicEx", "CRM.Sales.*");
    channel.QueueDeclare(queue: "AllCRM", false, false, false, null);
    channel.QueueBind("AllCRM", "CS1225_TopicEx", "CRM.#");


    string message = "A Bike Sales Has Occurred!!!";
    var body = Encoding.UTF8.GetBytes(message);

    channel.BasicPublish("CS1225_TopicEx", "CRM.Sales.Bikes", null, body);
    Console.WriteLine($"TopicEx.Publisher sent messages to topic exchanges.");
}
```

Receiving a Message

- Messages can be received by subscribing to the appropriate queue:

```
var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) =>
{
    var body = ea.Body.ToArray();
    var message = Encoding.UTF8.GetString(body);
    Console.WriteLine($"HeadersEx.Subscriber received '{message}' from BlackSquares Queue");
};
channel.BasicConsume("BlackSquares", true, consumer);

Console.WriteLine("Listening for messages. Press ENTER anytime to quit...");
Console.ReadLine();
```

Code.Hub

```
static void Main(string[] args)
{
    ConnectionFactory factory = new()
    {
        UserName = "guest",
        Password = "guest",
        HostName = "localhost"
    };

    using IConnection connection = factory.CreateConnection();
    using IModel channel = connection.CreateModel();
    channel.ExchangeDeclare("CS1225_HeadersEx", ExchangeType.Headers);

    Dictionary<string, object> bindHeaders = new()
    {
        { "x-match", "all" },
        { "shape", "square" },
        { "color", "black" }
    };
    channel.QueueDeclare(queue: "BlackSquares", false, false, false, null);
    channel.QueueBind("BlackSquares", "CS1225_HeadersEx", "", bindHeaders);

    string message = "A Black Square has been generated!";
    var body = Encoding.UTF8.GetBytes(message);

    IBasicProperties props = channel.CreateBasicProperties();
    Dictionary<string, object> msgHeaders = new()
    {
        { "shape", "square" },
        { "color", "black" }
    };
    props.Headers = msgHeaders;

    channel.BasicPublish("CS1225_HeadersEx", "", props, body);
    Console.WriteLine($"HeadersEx.Publisher sent message to the headers exchange.");
}
```

Worker Service

Code.Hub

1. Worker Service Introduction
2. Background Service Class Methods
3. Creating and Registering the Service

Worker Service

Code.Hub

Introduction

- ASP.NET Core enables **implementation of background tasks** through the `IHostedService` interface and the associated `BackgroudService` base class.
- By creating an extension of *BackgroundService* it is possible to create a background task that will continuously listen for new messages from RabbitMQ while the web application runs.

BackgroundService Methods

- The **BackgroundService** class defines three fundamental methods:
  - `StartAsync()` defines the logic that will be executed when the task is first started. It can be used to initialize properties and open connections.
  - `ExecuteAsync()` defines the main logic that will be executed as the task runs.
  - `StopAsync()` defined the logic that will be used when the task stops gracefully. It can be used to dispose objects and close connections.

```
public class SubscriberService : BackgroundService
{
    private IConnection Connection { get; }
    private IModel Channel { get; }

    public SubscriberService(ILoggerFactory loggerFactory)
    {
        ConnectionFactory factory = new()
        { UserName = "guest", Password = "guest", HostName = "localhost"};

       Connection = factory.CreateConnection();
       Channel = Connection.CreateModel();
    }

    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        stoppingToken.ThrowIfCancellationRequested();

        var consumer = new EventingBasicConsumer(Channel);
        consumer.Received += (ch, ea) =>
        {
            var content = System.Text.Encoding.UTF8.GetString(ea.Body.ToArray());
            Console.WriteLine($"DirectEx.WorkerSubscriber received {content}"); ;
        };

        Channel.BasicConsume("Green", true, consumer);
        return Task.CompletedTask;
    }

    public override void Dispose()
    {
        Channel.Close();
        Connection.Close();
        base.Dispose();
    }
}
```

Code.Hub

Registering the Service

- The presented code allows handling messages while the application is running.
- After the service has been created it can be injected to the **Service Container** by using the `AddHostedService<T>()` method:

```
builder.Services.AddHostedService
    <SubscriberService>();
```

- The written console messages can be seen by running the application directly (not through IIS Express, which will hide console output).

Thank you!

Code.Hub