**Introduction to RabbitMQ**

RabbitMQ is an open source lightweight message broker that implements AMQP (Advance Message Queuing Protocol). It is easy to deploy on premises and in the cloud where it can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements. It helps in scaling the application by deploying a message queuing mechanism in between the two applications/services.

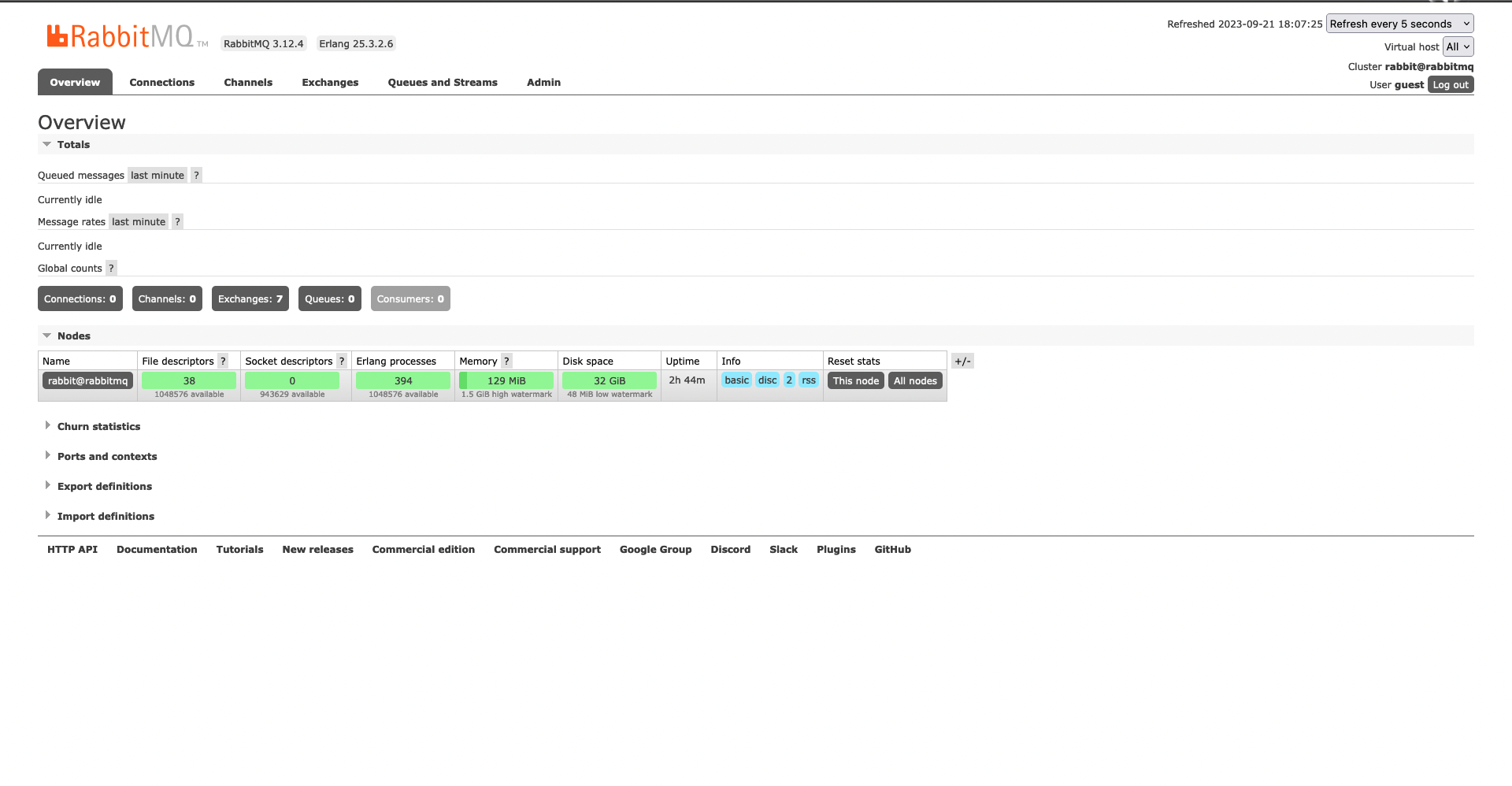
**How to install?**

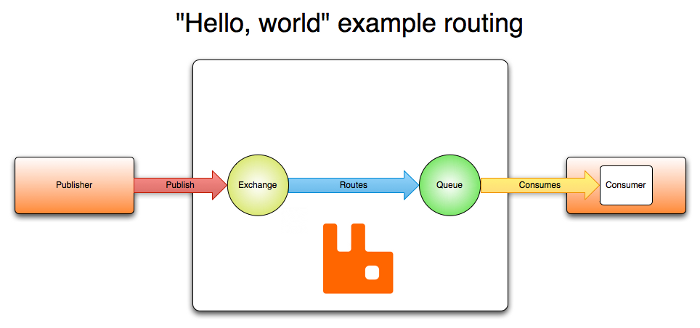
To run the RabbitMQ in the docker, run the following command:

docker run --rm --name rabbit --hostname rabbitmq -p 15672:15672 -p 5672:5672 rabbitmq:management

**Accessing the admin Console.**

Navigate to <http://localhost:15672/#/> to visit RabbitMQ admin console.

Login using the above mentioned username and password i.e username = guest , password = guest.After login we will get this display



**Creating the basic Producer**

To implement RabbitMQ in our producer we need to add **RabbitMQ.Client**

Nuget package in our project. Here is a sample code for the producer.

using System.Collections;

using System.Text;

using RabbitMQ.Client;

//creates the connection factory object and configures it to connect to RabbitMQ server on the machine with host name "localhost"

var factory = new ConnectionFactory { HostName = "localhost" };

//new connection is created

using var connection = factory.CreateConnection();

//creates a communication channel (channel) within an existing connection (connection) to the message broker

using var channel = connection.CreateModel();

//declares a queue named "letterbox" on the RabbitMQ server using the specified parameters. This queue is not durable, not exclusive, and not set to auto-delete

//parameters

//Queue: Name of the queue

//durable:This parameter determines whether the queue should survive a RabbitMQ server restart.

// If set to true, the queue will be durable, meaning it will be saved to disk and will not be lost even if the server restarts.

//exclusive:This parameter specifies whether the queue should be exclusive to the connection that declares it

//auto-delete:This parameter determines whether the queue should be automatically deleted when it is no longer in use (i.e., when there are no consumers or bindings)

//arguments:This parameter allows you to pass additional arguments when declaring the queue

channel.QueueDeclare(queue: "letterbox", durable: false, exclusive: false, autoDelete: false, arguments: null);

//Message that you need to send

var message = "This is my first message";

//encoding the message

var encodedMessage = Encoding.UTF8.GetBytes(message);

// publish a message directly to the "letterbox" queue on the RabbitMQ server using the channel

//exchange:This parameter specifies the exchange to which the message will be sent. The exchange is responsible for routing messages to one or more queues based on routing rules.

//routing Key:This is the routing key associated with the message. The routing key is used by the exchange to determine which queue(s) the message should be routed to

//basicProperties:This parameter allows you to set additional properties for the message, such as message headers, delivery mode (persistent or non-persistent), and more

//body:This is the actual content of the message, represented as a byte array. You encode your message content into bytes and provide it as the body parameter.

channel.BasicPublish(exchange: "", routingKey: "letterbox", basicProperties: null, body: encodedMessage);

Console.WriteLine($"Published message: {message}");

**Creating the basic Consume**

To implement RabbitMQ in our consumer we need to add **RabbitMQ.Client**

Nuget package in our project. Here is a sample code for the consumer.

using System.Text;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

//creates the connection factory object and configures it to connect to RabbitMQ server on the machine with host name "localhost"

var factory = new ConnectionFactory { HostName = "localhost" };

//new connection is created

using var connection = factory.CreateConnection();

//creates a communication channel (channel) within an existing connection (connection) to the message broker

using var channel = connection.CreateModel();

Console.WriteLine("Waiting for the Producer");

//declares a queue named "letterbox" on the RabbitMQ server using the specified parameters. This queue is not durable, not exclusive, and not set to auto-delete

//parameters

//Queue: Name of the queue

//durable:This parameter determines whether the queue should survive a RabbitMQ server restart.

// If set to true, the queue will be durable, meaning it will be saved to disk and will not be lost even if the server restarts.

//exclusive:This parameter specifies whether the queue should be exclusive to the connection that declares it

//auto-delete:This parameter determines whether the queue should be automatically deleted when it is no longer in use (i.e., when there are no consumers or bindings)

//arguments:This parameter allows you to pass additional arguments when declaring the queue

channel.QueueDeclare(queue: "letterbox", durable: false, exclusive: false, autoDelete: false, arguments: null);

//creating the consumer

var consumer = new EventingBasicConsumer(channel);

//registers an event handler for the Received event of the consumer.

//When a message is received from the associated RabbitMQ queue, the event handler extracts the message body, converts it to a string, and prints it to the console,

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

{

var body = ea.Body.ToArray();

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

Console.WriteLine($"Received message {message}");

};

// start a consumer that listens for messages from a letterbox queue

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

Console.ReadKey();

**Competing Consumer:**

The Competing Consumers pattern explains how multiple consumers compete for messages on the same message channelto process multiple messages concurrently.This pattern is useful when you want to process a discrete set of tasks asynchronously by distributing them among parallel consumers. In return, you’ll get a scalable, reliable, and resilient message processing system.

Here in our code we are using it to consume the message of our queue because the producer is producing faster than the consumer is consuming. Here is the sample code for same.

**Producer**:

using System.Text;

using RabbitMQ.Client;

var factory = new ConnectionFactory { HostName = "localhost" };

var random = new Random();

using var connection = factory.CreateConnection();

using var channel = connection.CreateModel();

var messageId = 1;

channel.QueueDeclare(queue: "letterbox", durable: false, exclusive: false, autoDelete: false, arguments: null);

while (true)

{

var publishingTime = random.Next(1, 4);

var message = $"Sending messageId: {messageId} ";

var encodedMessage = Encoding.UTF8.GetBytes(message);

channel.BasicPublish("", "letterbox", null, encodedMessage);

Console.WriteLine($"Published message: {message}");

Task.Delay(TimeSpan.FromSeconds(publishingTime)).Wait();

messageId++;

}

**Consumer**:

Here we have two consumers in the code. We can add multiple client but for our code two consumers are enough to consume the message.

using System.Text;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

class Program

{

private static async Task Main(string[] args)

{

var factory = new ConnectionFactory { HostName = "localhost" };

using var connection = factory.CreateConnection();

var random = new Random();

var random2 = new Random();

using var channel = connection.CreateModel();

using var channel2 = connection.CreateModel();

Console.WriteLine("Wating for the Producer");

channel.QueueDeclare(queue: "letterbox", durable: false, exclusive: false, autoDelete: false, arguments: null);

Test test = new Test(channel, random, "first");

Test test1 = new Test(channel2, random2, "second");

Thread t1 = new Thread(test.firstConsumer);

t1.Start();

Thread t2 = new Thread(test1.firstConsumer);

t2.Start();

// firstConsumer(channel, random);

//secondConsumer(channel, random);

Console.ReadKey();

//return Task.CompletedTask;

}

}

class Test

{

public IModel channel;

public Random random;

string consumer;

public Test(IModel channnel, Random random, string consumer)

{

this.channel = channnel;

this.random = random;

this.consumer = consumer;

}

public void firstConsumer()

{

//This line helps to send the message to the free consumer rather than sending it in round robin format

//if we remove this line the consumer will wait for its turn to consume the message rather than consuming after being free

channel.BasicQos(prefetchSize: 0, prefetchCount: 1, global: false);

var consumer1 = new EventingBasicConsumer(channel);

consumer1.Received += (model, ea) =>

{

var processingTime = random.Next(1, 6);

var body = ea.Body.ToArray();

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

Console.WriteLine($"Receving message: {message} and it take {processingTime} to process {consumer}");

Thread.Sleep(processingTime \* 1000);

channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);

};

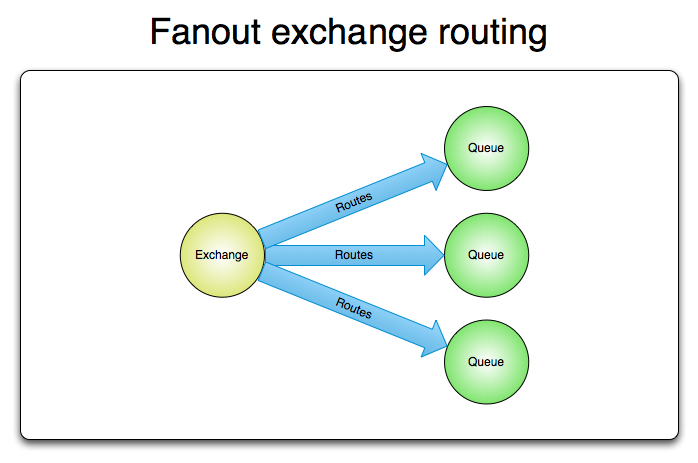
channel.BasicConsume(queue: "letterbox", autoAck: false, consumer: consumer1);

}

}

**Can more than one consumer consume the same message?**

Yes, it is possible for more than one consumer to consume the same message by using the concept of message queues and multiple consumer instances. This is known as the "publish/subscribe" or "fanout" messaging pattern. In this pattern we declare a exchange of type Fanout and bind this exchange with all the queues in which we need to publish the same message. It is a broadcast mechanism.



**Producer Code**

The queue is commented here because the queue is not necessary as the message is published at the exchange. And while consuming the message

using System.Text;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

var factory = new ConnectionFactory { HostName = "localhost" };

using (var connection = factory.CreateConnection())

{

// Create a channel

using (var channel = connection.CreateModel())

{

// Declare a fanout exchange

channel.ExchangeDeclare(exchange: "my-fanout-exchange", type: ExchangeType.Fanout);

// channel.QueueDeclare(queue: "queue-1", durable: false, exclusive: false, autoDelete: false, arguments: null);

// channel.QueueDeclare(queue: "queue-2", durable: false, exclusive: false, autoDelete: false, arguments: null);

// channel.QueueBind(queue: "queue-1", exchange: "my-fanout-exchange", routingKey: "");

// channel.QueueBind(queue: "queue-2", exchange: "my-fanout-exchange", routingKey: "");

var message = Encoding.UTF8.GetBytes("Hello, World!");

channel.BasicPublish(exchange: "my-fanout-exchange", routingKey: "", basicProperties: null, body: message);

}

}

**Fanout Consumer:**

using System.Text;

using RabbitMQ.Client;

using RabbitMQ.Client.Events;

var factory = new ConnectionFactory { HostName = "localhost" };

using var connection = factory.CreateConnection();

var channel = connection.CreateModel();

channel.ExchangeDeclare(exchange: "my-fanout-exchange", type: ExchangeType.Fanout);

channel.QueueDeclare(queue: "queue-1", durable: false, exclusive: false, autoDelete: false, arguments: null);

channel.QueueDeclare(queue: "queue-2", durable: false, exclusive: false, autoDelete: false, arguments: null);

channel.QueueBind(queue: "queue-1", exchange: "my-fanout-exchange", routingKey: "");

channel.QueueBind(queue: "queue-2", exchange: "my-fanout-exchange", routingKey: "");

var consumer1 = new EventingBasicConsumer(channel);

var consumer2 = new EventingBasicConsumer(channel);

consumer1.Received += (model, ea) =>

{

var body = ea.Body;

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

Console.WriteLine("Queue-1 Received: " + message);

};

consumer2.Received += (model, ea) =>

{

var body = ea.Body;

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

Console.WriteLine("Queue-2 Received: " + message);

};

channel.BasicConsume(queue: "queue-1", autoAck: true, consumer: consumer1);

channel.BasicConsume(queue: "queue-2", autoAck: true, consumer: consumer2);

Console.WriteLine("");

Console.WriteLine("Press [enter] to exit.");

Console.ReadLine();

**References:**

[**https://www.educative.io/answers/what-is-rabbitmq**](https://www.educative.io/answers/what-is-rabbitmq)

[**https://youtube.com/playlist?list=PLalrWAGybpB-UHbRDhFsBgXJM1g6T4IvO&feature=shared**](https://youtube.com/playlist?list=PLalrWAGybpB-UHbRDhFsBgXJM1g6T4IvO&feature=shared)

[**https://www.rabbitmq.com/getstarted.html**](https://www.rabbitmq.com/getstarted.html)