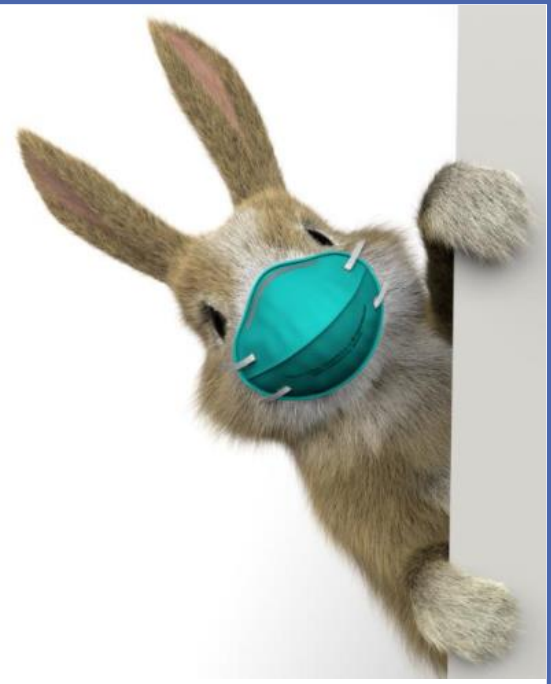
RabbitMQ

Introduction to RabbitMQ



AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

What is RabbitMQ?

- RabbitMQ is a message queuing software also known as a message broker or queue manager.
- RabbitMQ accepts, stores and forwards binary blobs of data → messages.

- RabbitMQ is where queues can be defined, and applications may connect and transfer a message onto it.
- Message queues enable asynchronous communication, which means that other applications (endpoints) that are producing and consuming messages interact with the queue instead of communicating directly with each other.

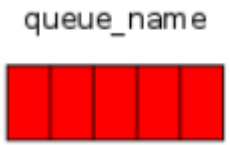
What is RabbitMQ?

- RabbitMQ is open source message broker software that implements the Advanced Message Queuing Protocol (AMQP).
- The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover.
- Client libraries to interface with the broker are available for all major programming languages.

Messages are sent from a sender to a receiver

queue

- A queue is the name for a post box which lives inside RabbitMQ.
- Although messages flow through RabbitMQ and your applications, they can only be stored inside a queue.
- **A queue is only bound by the host's memory & disk limits, it's essentially a large message buffer**.
- Many producers can send messages that go to one queue, and many consumers can try to receive data from one queue.



Message

- A message can include any information.
- It could, for example, contain information about a process or job that should start on another application, possibly even on another server.
- Or it might be a simple text message.

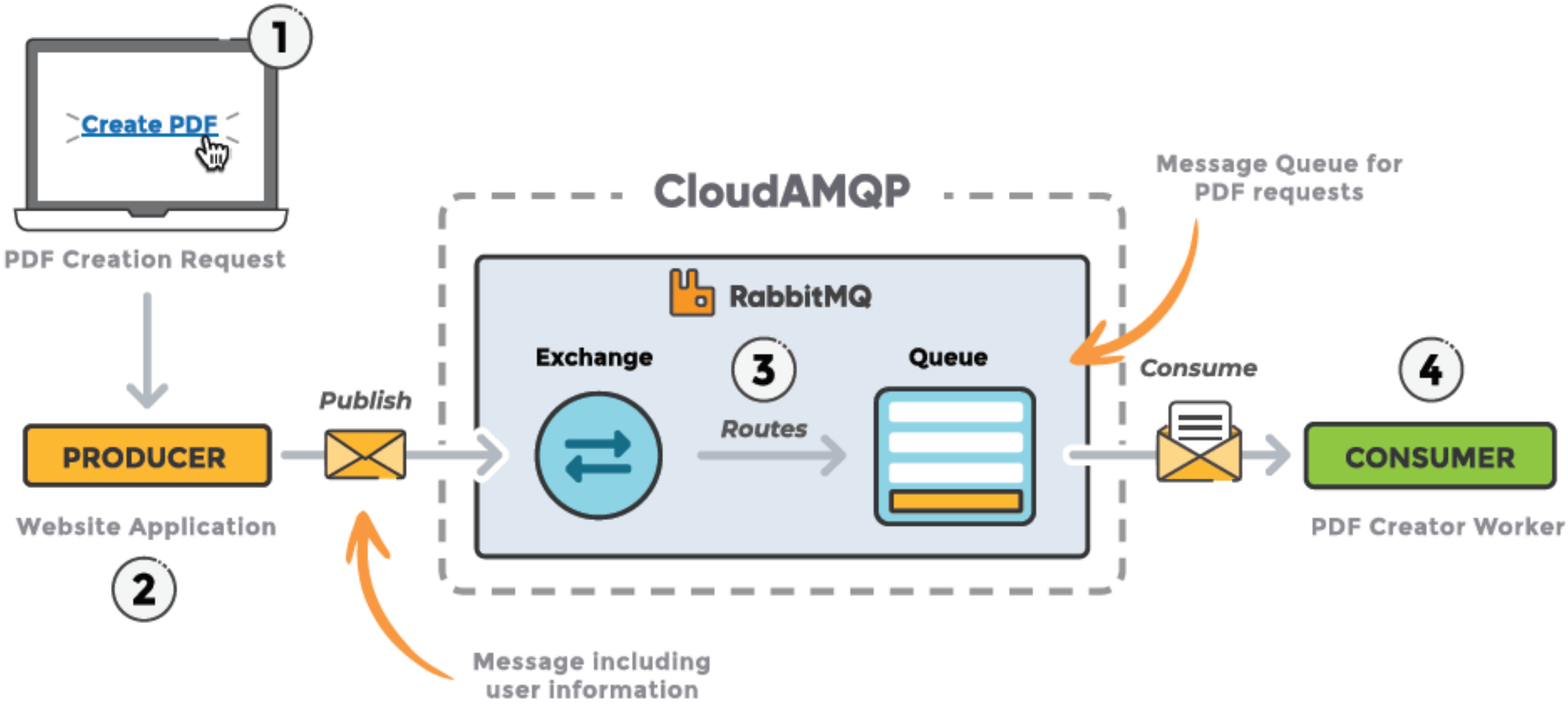
A sketch of the RabbitMQ workflow

The broker holds the message queue

When to use RabbitMQ

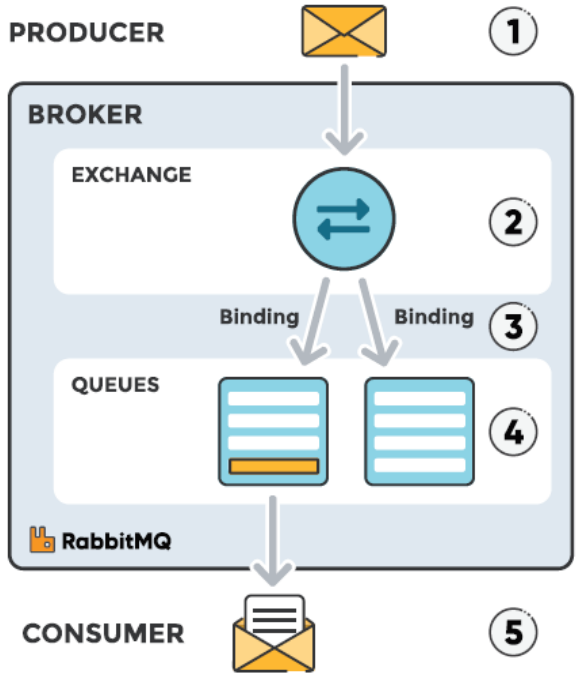
- Message queuing allows web servers to respond to requests in their own time instead of being forced to perform resource-heavy procedures immediately.
- Message queuing is also useful for distributing a message to multiple recipients for consumption or when balancing the load between workers.
- The consumer can be on an entirely different server than the publisher or they can be located on the same server, it makes no difference.
- Requests can be created in one programming language and handled in another programming language, as the two applications only communicate through the messages they are sending to each other.
- The two services have what is known as ‘low coupling’ between the sender and the receiver.

Example


1
Create PDF
PDF Creation Request
CloudAMQP
Message Queue for
PDF requests
RabbitMQ
Exchange
3
Queue
Publish
Routes
Consume
4
PRODUCER
CONSUMER
Website Application
PDF Creator Worker
2
Message including
user information

Exchanges

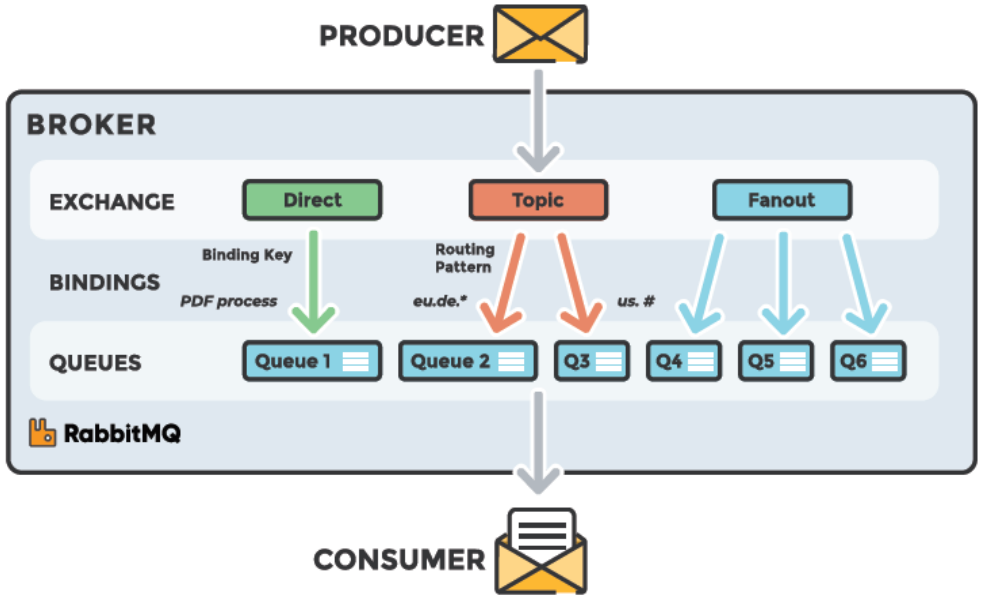
- Messages are not published directly to a queue.
- The producer sends a message to an exchange.
- The job of an exchange is to accept messages from the producer applications and route them to the correct message queues.
- It does this with the help of bindings and routing keys.
- A binding is a link between a queue and an exchange.



- When creating an exchange, its type must be specified.
- The exchange looks at different message attributes and keys depending on the exchange type.

Types of Exchanges

- Direct
  - A direct exchange delivers messages to queues based on a message routing key.
  - In a direct exchange, the message is routed to the queue with the exact match of binding key as the routing key of the message.
- Topic
  - The topic exchange performs a wildcard match between the routing key and the routing pattern specified in the binding.
- Fanout
  - A fanout exchange routes messages to all of the queues that are bound to it.
  - The routing key is ignored.
- Headers
  - A header exchange uses the message header attributes for routing purposes.



The default exchange

- The default exchange is a direct exchange with no name (empty string) pre-declared by the broker.
- The default exchange has a default binding that says that the message will arrive at a queue with the same name as the routing key.

Other RabbitMQ concepts

- **Connection**
  - TCP connection between the application and the RabbitMQ broker.
- **Channel**
  - A virtual connection inside a *connection*.
  - When publishing or consuming messages or subscribing to a queue, it's all done over a channel.
- **Routing Key**
  - The key that the exchange looks at to decide how to route the message to queues.
  - Think of the routing key as the destination address of a message.
- **AMQP**
  - Advanced Message Queuing Protocol
  - The primary protocol used by RabbitMQ for messaging.
- **Users**
  - It's possible to connect to RabbitMQ with a given username and password, with assigned permissions such as rights to read, write and configure.
- **Vhost**
  - Virtual host or Vhost segregate applications that are using the same RabbitMQ instance.
  - Different users can have different access privileges to different vhosts and queues, and exchanges can be created so that they only exist in one vhost.
- **Acknowledgments and Confirms**
  - Indicators that messages have been received or acted upon.
  - Acknowledgements can be used in both directions

Connections

RabbitMQ supports several protocols:

- **AMQP 0-9-1** with extensions
- AMQP 1.0
- MQTT 3.1.1
- STOMP 1.0 through 1.2

Port Access
RabbitMQ nodes bind to ports (open server TCP sockets) in order to accept client and CLI tool connections.
4369: epmd, a peer discovery service used by RabbitMQ nodes and CLI tools
5672, 5671: used by AMQP 0-9-1 and 1.0 clients without and with TLS
25672: …
15672: HTTP API clients, management UI and rabbitmqadmin (only if the management plugin is enabled)
61613, 61614: STOMP clients without and with TLS.

- Each protocol has its own set of client libraries.
- All protocols supported by RabbitMQ are TCP-based and assume long-lived connections.
- In order for a client to successfully connect, target RabbitMQ node must allow for connections on a certain protocol-specific port.

MQTT

- The Standard for IoT Messaging.
- MQTT is an OASIS standard messaging protocol for the Internet of Things (IoT).
- It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth.
- MQTT today is used in a wide variety of industries, such as automotive, manufacturing, telecommunications, oil and gas, etc.
- MQTT can scale to connect with millions of IoT devices.
- Many different servers/brokers and clients:
  https://mqtt.org/software/
- For a quick overview: https://en.wikipedia.org/wiki/MQTT

CloudAMQP

- CloudAMQP is a hosted RabbitMQ solution
  - RabbitMQ as a Service
  - All that is required is to sign up for an account and create an instance.
- There is no need to set up and install RabbitMQ or care about cluster handling, as CloudAMQP will handle that.
- RabbitMQ is available free with the plan Little Lemur.
  - Go to the plan page (www.cloudamqp.com/plans.html) and sign up for an appropriate plan.
  - Click on “details” of the cloud-hosted RabbitMQ instance to find the username, password, and **connection URL**.

Downloading and Installing RabbitMQ

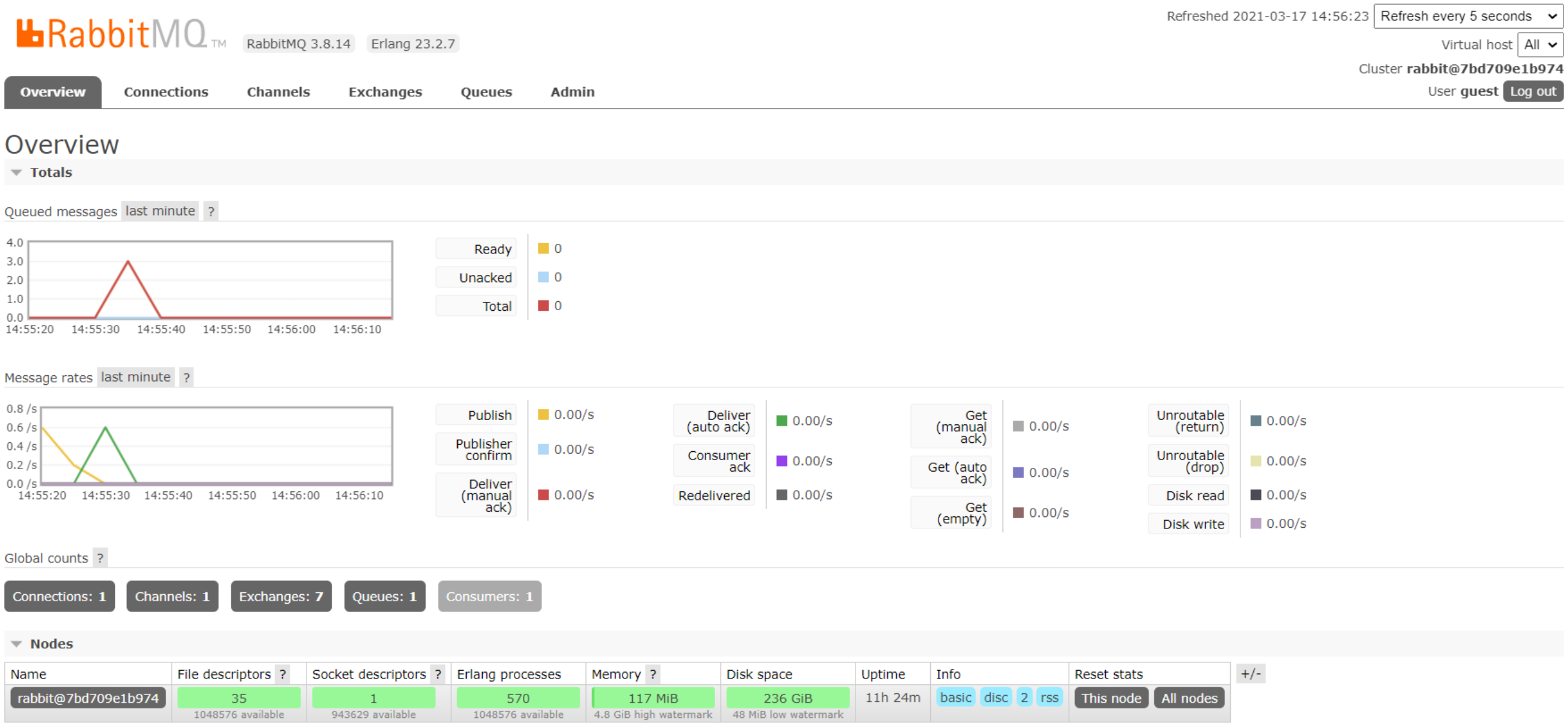
- To experimenting with RabbitMQ on your workstation, try the community Docker image:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

- Se other options here: https://www.rabbitmq.com/download.html

Admin interface: http://localhost:15672

Admin interface



Publish and Consume Messages

- RabbitMQ speaks the AMQP protocol by default.
  - But RabbitMQ speaks multiple protocols.
- We must use a library that understands the same protocol to be able to communicate with RabbitMQ.
- A RabbitMQ client library abstract the complexity of the AMQP protocol into simple methods.
- The methods should be used when connecting to the RabbitMQ broker using the given parameters, hostname, port number, etc.
- There is a choice of libraries for all major programming languages.

Steps to setup a queue

1. Create a connection object.
   - Here, the username, password, connection URL, port, etc., are specified.
   - A TCP connection will be set up between the application and RabbitMQ.
2. Open a channel
   - Use the connection interface to that.
3. Declare (create) a queue.
   - Declaring a queue will cause it to be created if it does not already exist.
   - All queues need to be declared before they can be used.
4. Set up exchanges and bind a queue to an exchange.
   - Messages are only routed to a queue if the queue is bound to an exchange.
   - The default exchange is used if you don’t specify one.

using the amqp.node client

- Install node package:

```
npm install amqplib
```

Send - JavaScript

```
const amqp = require('amqplib/callback_api');
amqp.connect('amqp://localhost', function (error0, connection) {
    if (error0) {
        throw error0;
    }
    connection.createChannel(function (error1, channel) {
        if (error1) {
            throw error1;
        }
        var queue = 'hello';
        var msg = 'Hello world';
        channel.assertQueue(queue, {
            durable: false
        });
        channel.sendToQueue(queue, Buffer.from(msg));
        console.log(" [x] Sent %s", msg);
    });
    setTimeout(function() {
        connection.close();
        process.exit(0);
    }, 500);
});
```

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

code should not be considered production ready!

Receive - JavaScript

```
const amqp = require('amqplib/callback_api');
amqp.connect('amqp://localhost', function (error0, connection) {
    if (error0) {
        throw error0;
    }
    connection.createChannel(function (error1, channel) {
        if (error1) {
            throw error1;
        }
        var queue = 'hello';
        channel.assertQueue(queue, {
            durable: false
        });
        console.log(" [*] Waiting for messages in %s. To exit press CTRL+C", queue);

        channel.consume(queue, function (msg) {
            console.log(" [x] Received %s", msg.content.toString());
        }, {
            noAck: true
        });
    });
});
```

Send – C#

```
static void Main(string[] args)
{
    var factory = new ConnectionFactory() { HostName = "localhost" };
    using (var connection = factory.CreateConnection())
    {
        using (var channel = connection.CreateModel())
        {
            channel.QueueDeclare(queue: "hello",
                                 durable: false,
                                 exclusive: false,
                                 autoDelete: false,
                                 arguments: null);
            string message = "Hello World!";
            var body = Encoding.UTF8.GetBytes(message);

            channel.BasicPublish(exchange: "",
                                 routingKey: "hello",
                                 basicProperties: null,
                                 body: body);
            Console.WriteLine(" [x] Sent {0}", message);
        }
    }
}
```

code should not be considered production ready!

Receive – C#

```
static void Main(string[] args) {
    var factory = new ConnectionFactory() { HostName = "localhost" };
    using (var connection = factory.CreateConnection()) {
        using (var channel = connection.CreateModel()){
            channel.QueueDeclare(queue: "hello",
                                 durable: false,
                                 exclusive: false,
                                 autoDelete: false,
                                 arguments: 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: "hello",
                                 autoAck: true,
                                 consumer: consumer);
            Console.WriteLine(" Press [enter] to exit."); Console.ReadLine();
        }
    }
}
```

How to send json?

- AMQP messages also have a payload, which AMQP brokers treat as an opaque byte array.
  - The broker will not inspect or modify the payload.
  - It is possible for messages to contain only attributes and no payload.
- It is common to use serialisation formats like:
  - JSON
  - Thrift
  - Protocol Buffers
  - MessagePack

  to serialize structured data in order to publish it as the message payload.
- AMQP peers typically use the "content-type" and "content-encoding" fields to communicate this information, but this is by convention only.

Json example

Producer:

```
let payloadAsString = JSON.stringify(payload);
channel.sendToQueue(queue, Buffer.from(payloadAsString));
```

Consumer:

```
let payload = JSON.parse(msg.content.toString());
//then access the object as you normally do, i.e. :
let id = payload.id;
```

References & Links

- "The Optimal RabbitMQ Guide”
  https://www.cloudamqp.com/rabbitmq_ebook.html

- RabbitMQ
  https://www.rabbitmq.com/
  https://www.rabbitmq.com/tutorials/tutorial-one-javascript.html
  https://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html

- Docker Official Images
  https://hub.docker.com/_/rabbitmq

- CloudAMQP at Heroku Getting started
  https://www.cloudamqp.com/docs/heroku.html