Advanced topics

1. Messaging systems

Intro

- Messaging provides a mechanism for loosely-coupled integration of systems
- The central unit of processing in a message is a message which typically contains a **body** and a **header**
- Use cases include:
  - Log aggregation between systems
  - Event propagation between systems - някакви събития се fire-ват
  - Offloading log-running tasks to worker nodes - the result of the task then to be sent to a third systems for example
- Messaging solutions implement different protocols for transferring of messages such as **AMQP** (binary protocol), XMPP, MQTT and many more like XML, JSON, etc.
- The variety of protocols implies vendor lock-in when using a particular messaging solution (also called a messaging broker) - ако е специфичен протокола лошо. Т.е. протокола е добре да е такъв, че да може да се използва от различни message broker systems

- Message brokers
  - ActiveMQ - using JMS (Java Messaging System) Java EE
  - RabbitMQ
  - Qpid
  - TIBCO
  - WebSphere MQ
  - Msmq

- Messaging solutions provide means for:
  - Securing message transfer, authenticating and authorizing messaging endpoints
  - Routing messages between endpoints
  - Subscribing to the broker

- An **enterprise service bus (ESB)** is one layer of abstraction above a messaging solution that further provides:
  - Adapters for different messaging protocols
  - Translation of messages between the different types of protocols

I. RabbitMQ

Info

- An open source message broker written in Erlang
  - Заема малко ранм памет и процесор
  - при Erlang няма context switching като при JVM
  - reliability - дете ако не се изпълни, то родителят му го пуска за изпълнение наново
- **Implements the AMQP protocol** (Advanced Message Queueing Protocol)
- Has a pluggable architecture and provides extension for other protocols such as HTTP, STOMP and MQTT

- AMQP is a binary protocol that aims to standardize middleware communication
- The AMQP protocol derives its origins form the financial industry - processing of large volumes of financial data between different systems is a classic use case of messaging
- The AMQP protocol defines:
    - **Exchanges** - the message broker endpoints that receive messages
    - **Queues** - the message broker endpoints that store messages from exchanges and are used by subscribers for retrieval of messages. The Queue can also be persistent - messages can be saved.
    - **Bindings** - rules that bind exchanges and queues
- The AMQP protocol is programmable - which means that the above entities can be created/ modified/ deleted by applications

- The AMQP protocol defines multiple connection channels inside a single TCP connection in order to remove the overhead of opening a large number of TCP connections to the message broker

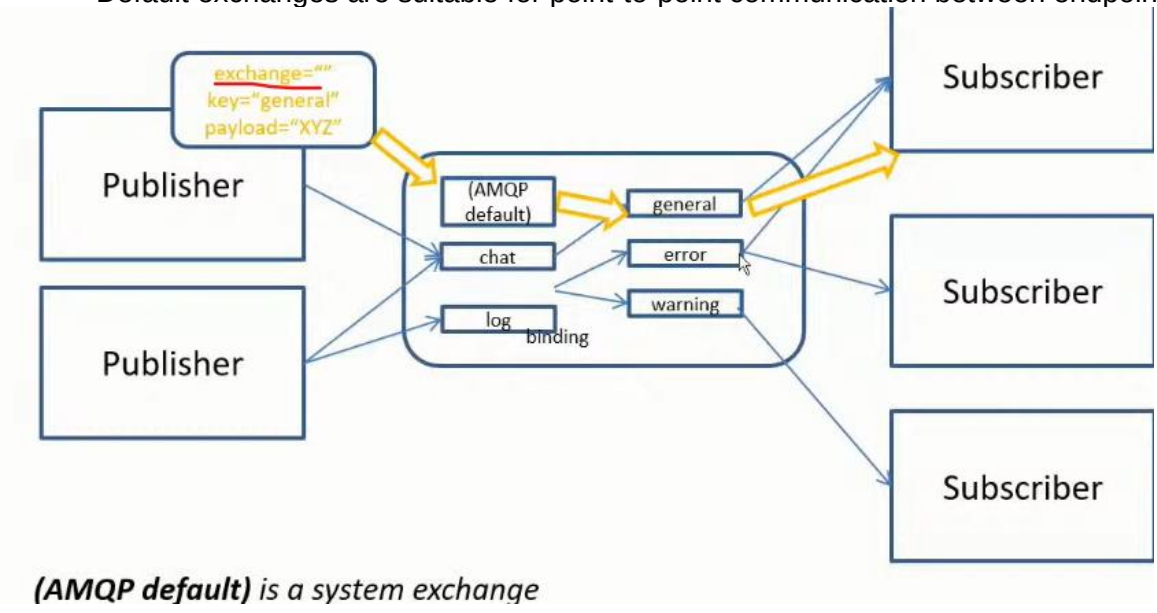
- Each message can be published with a **routing key**
- Each binding between an exchange and a queue has a **binding key**
- Routing of messages is determined based on matching between the **routing key** and the **binding key**

Messaging patterns with RabbitMQ

- Different types of messaging patterns are implemented by means of different types of exchanges
- RabbitMQ provides the following types of exchanges:
    - default - без име като търси съвпадение на **routing key** с **binding key**
    - direct - има име като търси съвпадение на **routing key** с **binding key**
    - fanout
    - topic
    - headers- на база мачинг по хедъри също

Default exchange

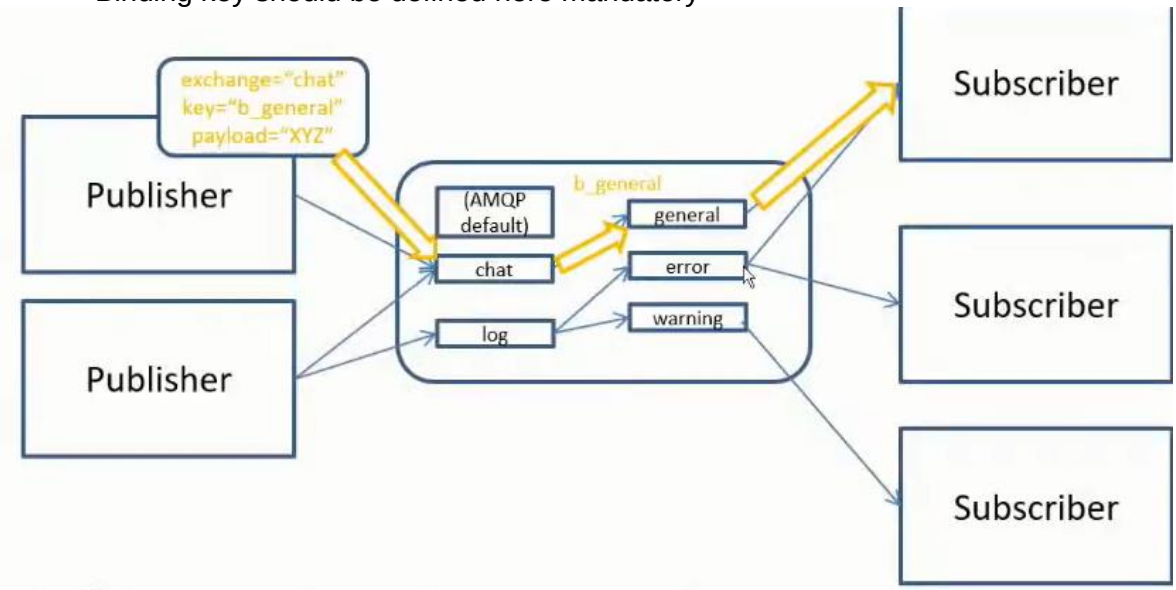
- A default exchange has **the empty string as a name** and routes messages to a queue if the routing key of the message matches the queue name (no binding needs to be declared between a default exchange and a queue)
- Default exchanges are suitable for point-to-point communication between endpoints



***(AMQP default)** is a system exchange*

Direct exchange

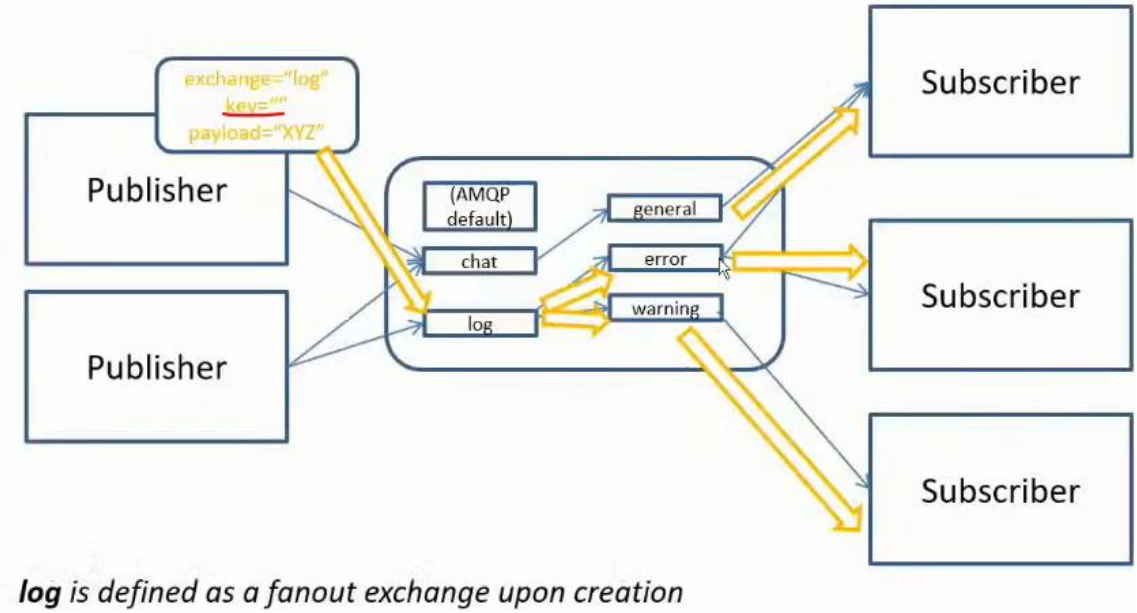
- A direct exchange routes messages to a queue if the routing key of the message matches the binding key between the direct exchange and the queue
- Direct exchanges are suitable for point-to-point communication between endpoints
- Binding key should be defined here mandatory



***chat** is defined as a direct exchange upon creation*

Fanout exchange

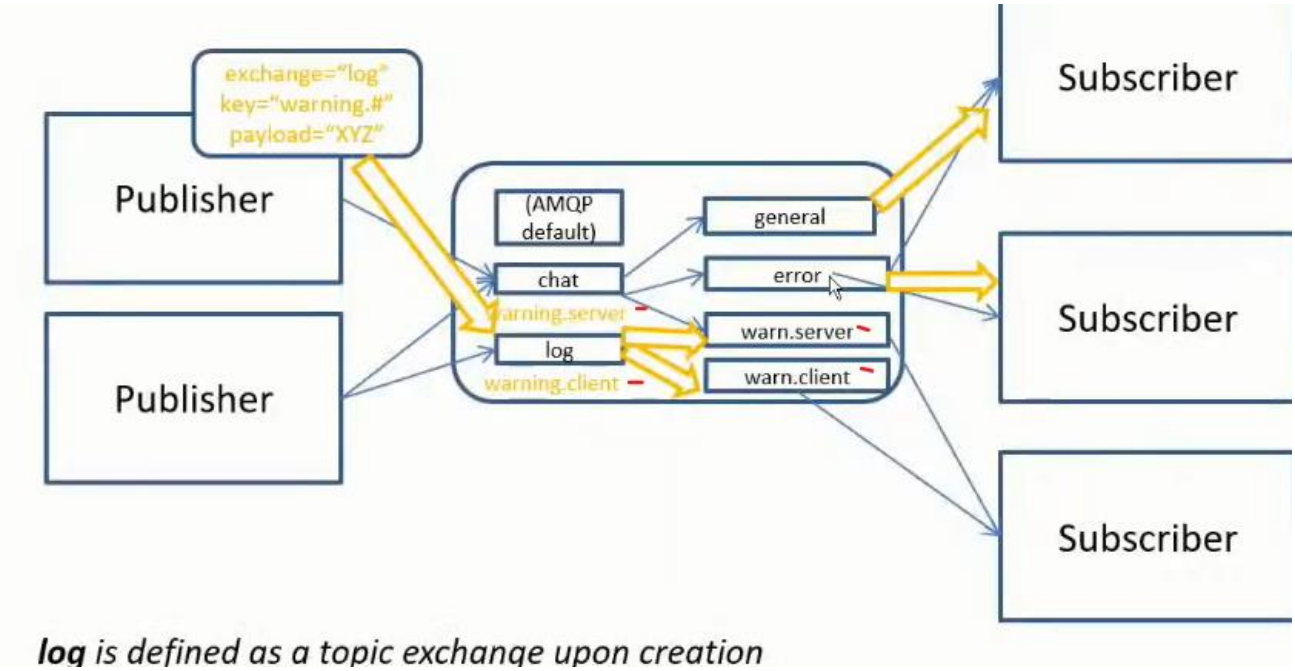
- A fanout exchange routes (broadcasts) messages to all queues that are bound to it (the binding key is not used)
- Fanout exchanges are suitable for publish-subscribe communication between endpoints



***log** is defined as a fanout exchange upon creation*

Topic exchange

- A topic exchange routes (multicasts) messages to all queues that have a binding key (can be a pattern) that matches the routing key of the message
- Topic exchanges are suitable for routing messages to different queues based on the type of message
- Диеса след *warning.* може да е всякаква дума

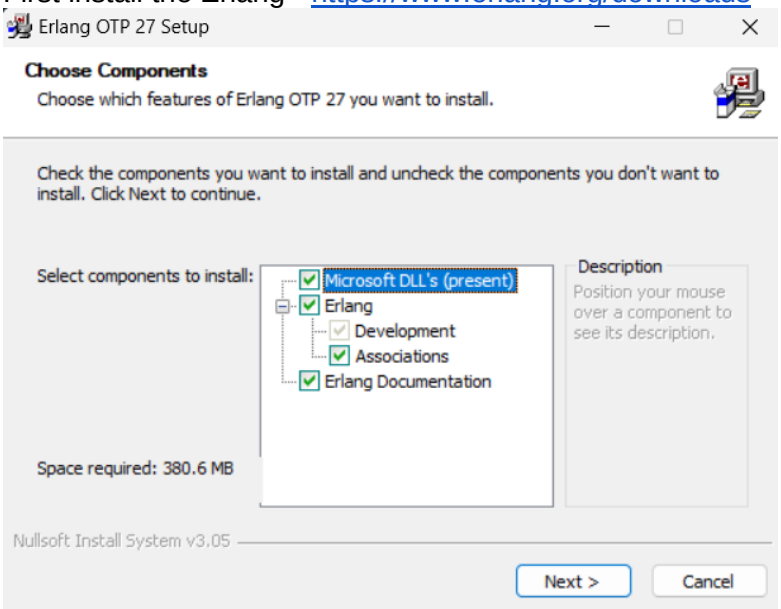


Headers exchange

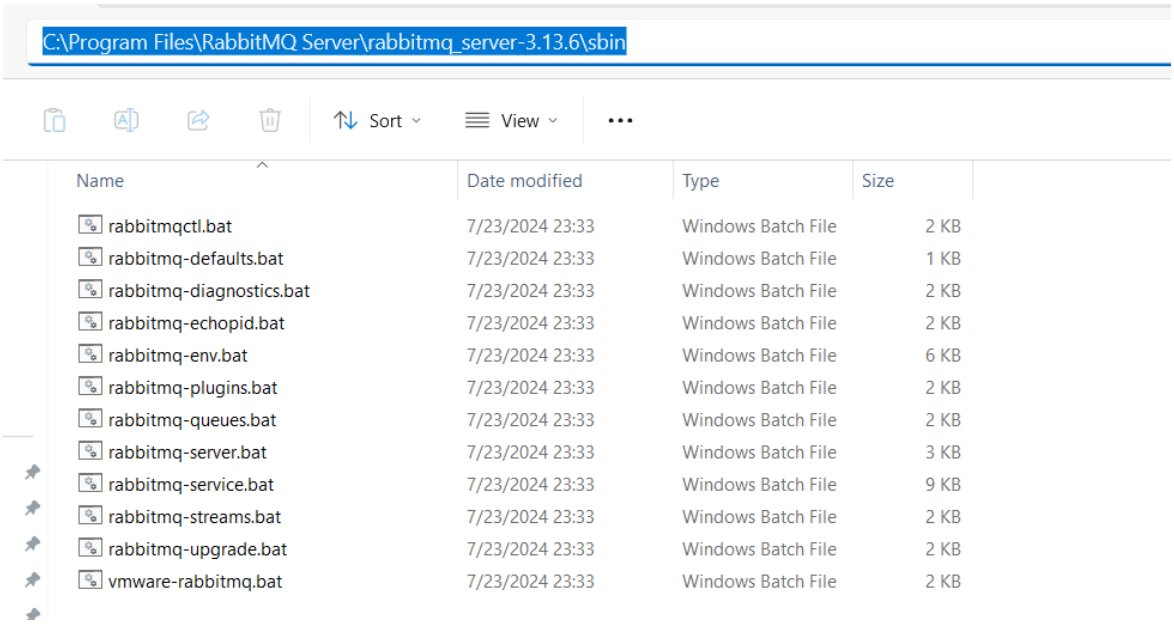
- A headers exchange routes messages based on a custom message header
- Header exchanges are suitable for routing messages to different queues based on more than one attribute

Installation of the RabbitMQ server

First install the Erlang - https://www.erlang.org/downloads



Then install the RabbitMQ server - https://www.rabbitmq.com/docs/download
rabbitmq-service.bat
rabbitmq-service.bat



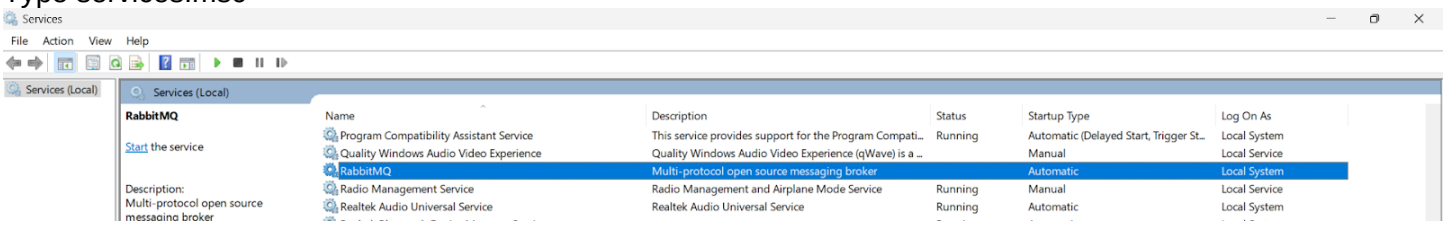
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.13.6\sbin>**rabbitmq-plugins.bat enable** rabbitmq_management
Enabling plugins on node rabbit@SVILKATA:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@SVILKATA...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch

set 3 plugins.
Offline change; changes will take effect at broker restart.

През CommandPrompt като администратор:
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.13.6\sbin>rabbitmq-plugins.bat list
Listing plugins with pattern ".*" ...
 Configured: E = explicitly enabled; e = implicitly enabled
 | Status: [failed to contact rabbit@SVILKATA - status not shown]
 |/
[ ] rabbitmq_amqp1_0 3.13.6
[ ] rabbitmq_auth_backend_cache 3.13.6
[ ] rabbitmq_auth_backend_http 3.13.6
[ ] rabbitmq_auth_backend_ldap 3.13.6
[ ] rabbitmq_auth_backend_oauth2 3.13.6
[ ] rabbitmq_auth_mechanism_ssl 3.13.6
[ ] rabbitmq_consistent_hash_exchange 3.13.6
[ ] rabbitmq_event_exchange 3.13.6
[ ] rabbitmq_federation 3.13.6
[ ] rabbitmq_federation_management 3.13.6
[ ] rabbitmq_jms_topic_exchange 3.13.6
[*] rabbitmq_management 3.13.6
[*] rabbitmq_management_agent 3.13.6
[ ] rabbitmq_mqtt 3.13.6

```
[  ] rabbitmq_peer_discovery_aws     3.13.6
[  ] rabbitmq_peer_discovery_common        3.13.6
[  ] rabbitmq_peer_discovery_consul 3.13.6
[  ] rabbitmq_peer_discovery_etcd    3.13.6
[  ] rabbitmq_peer_discovery_k8s     3.13.6
[  ] rabbitmq_prometheus             3.13.6
[  ] rabbitmq_random_exchange        3.13.6
[  ] rabbitmq_recent_history_exchange  3.13.6
[  ] rabbitmq_sharding               3.13.6
[  ] rabbitmq_shovel            3.13.6
[  ] rabbitmq_shovel_management    3.13.6
[  ] rabbitmq_stomp             3.13.6
[  ] rabbitmq_stream            3.13.6
[  ] rabbitmq_stream_management            3.13.6
[  ] rabbitmq_top               3.13.6
[  ] rabbitmq_tracing           3.13.6
[  ] rabbitmq_trust_store            3.13.6
[*] rabbitmq_web_dispatch          3.13.6
[  ] rabbitmq_web_mqtt               3.13.6
[  ] rabbitmq_web_mqtt_examples    3.13.6
[  ] rabbitmq_web_stomp              3.13.6
[  ] rabbitmq_web_stomp_examples           3.13.6
```

Type services.msc

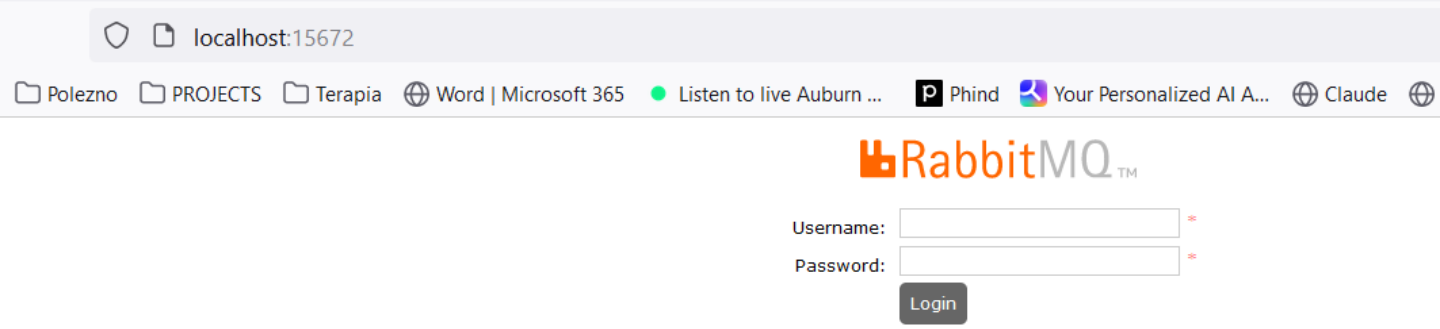



```
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.13.6\sbin>rabbitmq-server.bat
2024-07-31 11:52:47.571000+03:00 [warning] <0.134.0> Using RABBITMQ_ADVANCED_CONFIG_FILE:
c:/Users/svilk/AppData/Roaming/RabbitMQ/advanced.config
2024-07-31 11:52:52.145000+03:00 [notice] <0.45.0> Application syslog exited with reason: stopped
2024-07-31 11:52:52.145000+03:00 [notice] <0.213.0> Logging: switching to configured handler(s); following
messages may not be visible in this log output

  ##  ##      RabbitMQ 3.13.6
  ##  ##
  ##########  Copyright (c) 2007-2024 Broadcom Inc and/or its subsidiaries
  ######  ##
  ##########  Licensed under the MPL 2.0. Website: https://rabbitmq.com

  Erlang:      27.0.1 [jit]
  TLS Library: OpenSSL - OpenSSL 3.1.0 14 Mar 2023
  Release series support status: see https://www.rabbitmq.com/release-information

  Doc guides:  https://www.rabbitmq.com/docs
  Support:     https://www.rabbitmq.com/docs/contact
  Tutorials:   https://www.rabbitmq.com/tutorials
  Monitoring:  https://www.rabbitmq.com/docs/monitoring
  Upgrading:   https://www.rabbitmq.com/docs/upgrade

  Logs: <stdout>
```

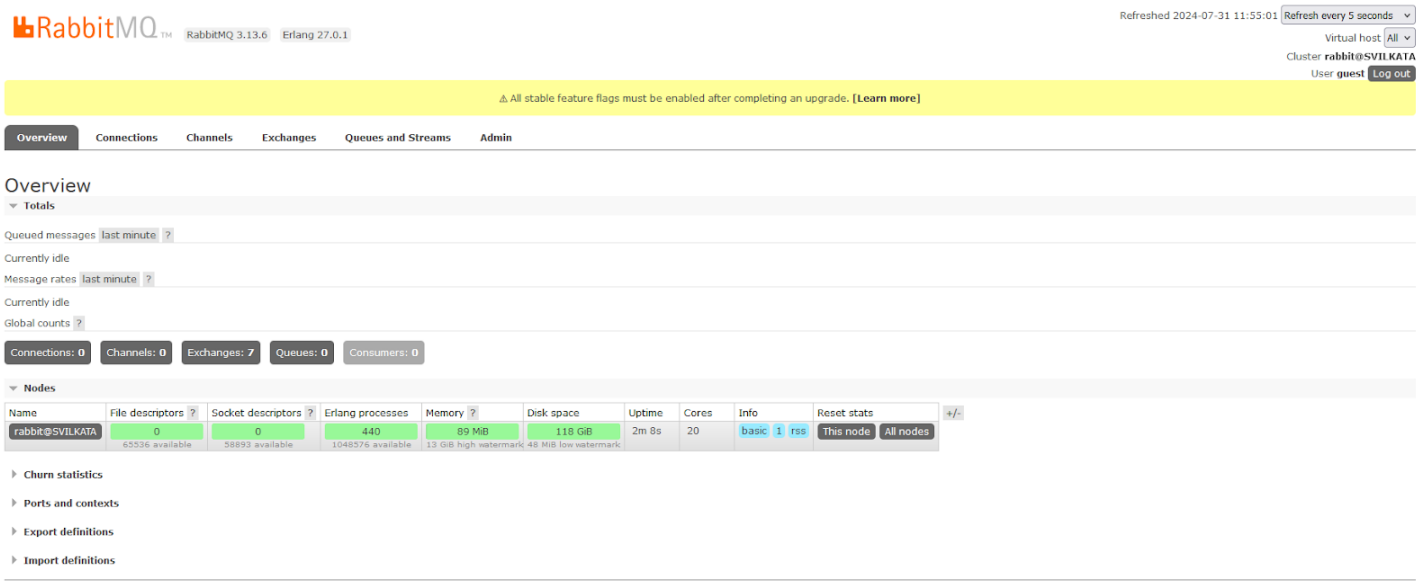
c:/Users/svilk/AppData/Roaming/RabbitMQ/log/rabbit@SVILKATA.log

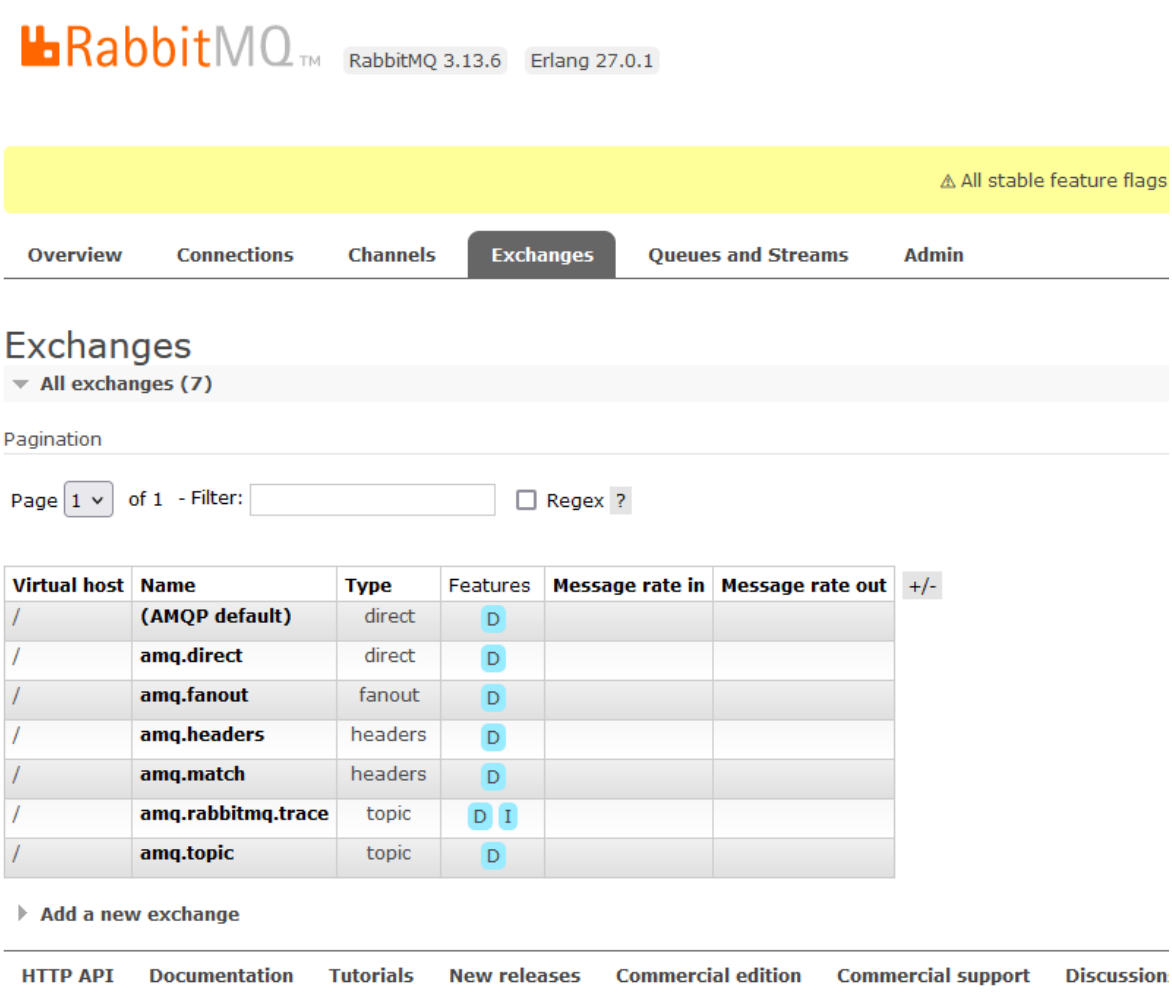
Config file(s): c:/Users/svilk/AppData/Roaming/RabbitMQ/advanced.config

Starting broker... **completed with 3 plugins.**



username: guest
password: guest



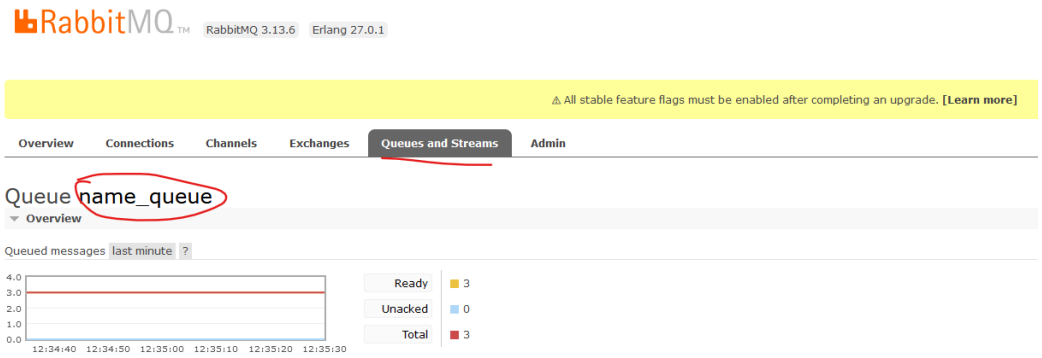


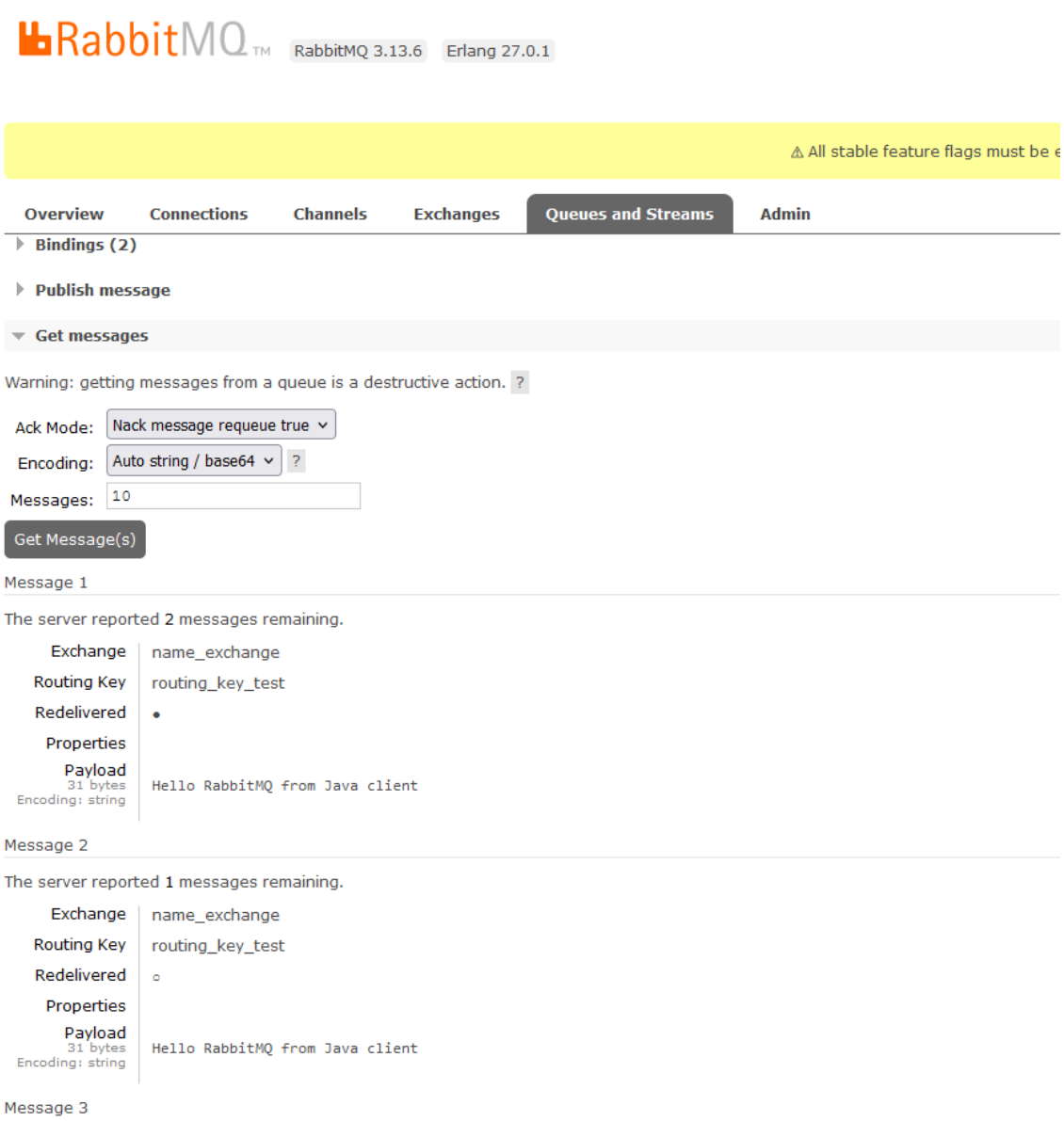
Using the Java Client

```
<dependency>
 <groupId>com.rabbitmq</groupId>
 <artifactId>amqp-client</artifactId>
 <version>5.20.0</version>
</dependency>
```

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeoutException;

public class Publisher {
   public static void main(String[] args) throws IOException, TimeoutException {
       Connection connection = null;
       Channel channel = null;

       try {
           ConnectionFactory connectionFactory = new ConnectionFactory();
           connectionFactory.setHost("localhost"); //by default on port 15672
           connection = connectionFactory.newConnection();
           channel = connection.createChannel();
```

```
            channel.exchangeDeclare("name_exchange", "direct"); //created only once on
the RabbitMQ server
            channel.queueDeclare("name_queue", false, false, false, null); //created
only once on the RabbitMQ server
            channel.queueBind("name_queue", "name_exchange", "routing_key_test");

            channel.basicPublish("name_exchange", "routing_key_test", null,
                    "Hello RabbitMQ from Java
client".getBytes(StandardCharsets.UTF_8));
        } finally {
            if (channel != null) {
                channel.close();
            }
            if (connection != null) {
                connection.close();
            }
        }
    }
}
```






```
import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class Subscriber {
   public static void main(String[] args) throws IOException, TimeoutException,
InterruptedException {
       Connection connection = null;
       Channel channel = null;

       try {
           ConnectionFactory connectionFactory = new ConnectionFactory();
           connectionFactory.setHost("localhost"); //by default on port 15672
           connection = connectionFactory.newConnection();
           channel = connection.createChannel();
```

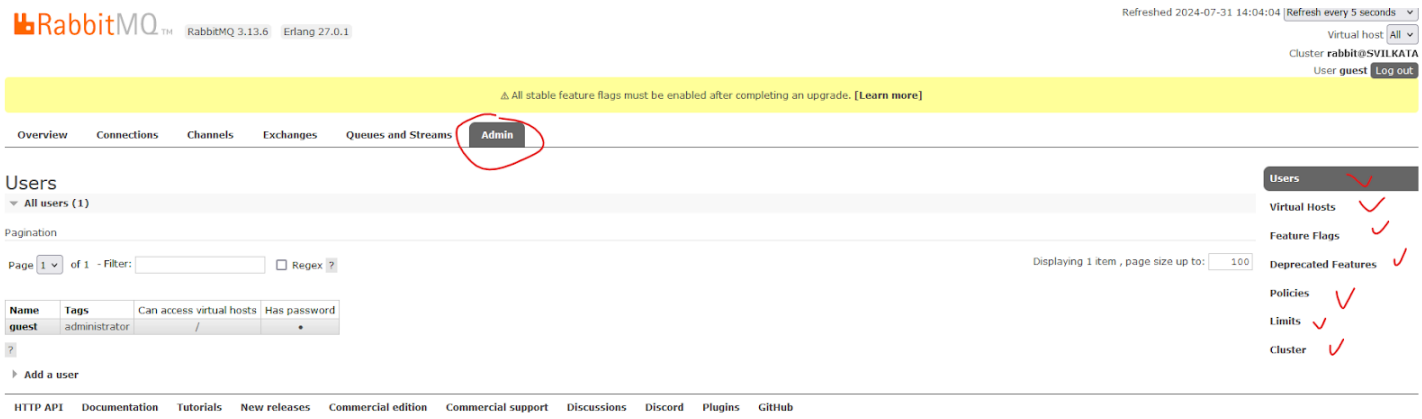
```
            channel.exchangeDeclare("name_exchange", "direct"); //created only once on
the RabbitMQ server
            channel.queueDeclare("name_queue", false, false, false, null); //created
only once on the RabbitMQ server
            channel.queueBind("name_queue", "name_exchange", "routing_key_test");

            while (true) {
                channel.basicConsume("name_queue", true, new DefaultConsumer(channel)
{
                    @Override
                    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                        // super.handleDelivery(consumerTag, envelope, properties,
body);  no-op no work to do
                        System.out.println(new String(body));
                    }
                });

                Thread.sleep(3000);
            }
        } finally {
            if (channel != null) {
                channel.close();
            }
            if (connection != null) {
                connection.close();
            }
        }
    }
}
```

Administration

- Administration of the broker includes a number of activities such as:
  - Updating the broker
  - Backing up the broker database
  - installing/uninstalling and configuring plug-ins
  - Configuring the various components of the broker

- Apart from queues, exchanges and bindings we can also manage the following types of components:
  - vhosts (virtual hosts) - for logical separation of broker components
  - users
  - Parameters - defining upstream links to another brokers
  - Policies - for queue mirroring

- Administration of single instance or an entire cluster can be performed in several ways:
  - Using the management Web interface



- Using the management HTTP API - rest API
- Using the **rabbitmq-admin.py /** **rabbitmqadmin.py** script - written on Python
- Using the **rabbitmqctl** utility

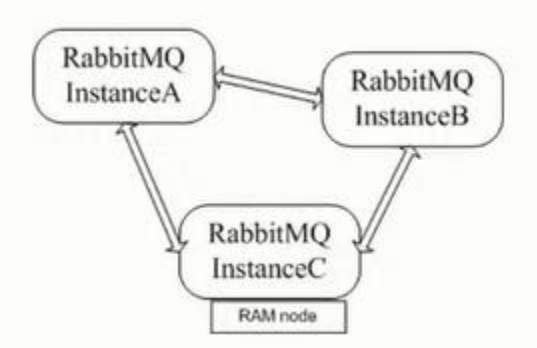
Scalability and High Availability in RabbitMQ

Basic default configuration

- RabbitMQ provides clustering support that allows new RabbitMQ nodes to be added on the fly
- Clustering by default does not guarantee that message loss may or may not occur - ако дадена инстанция примерно падне

- Nodes in a RabbitMQ cluster can be:
  - DISK - data is persisted in the node database
  - RAM - data is buffered only in-memory - когато не е критично да се запазват данните след рестарт например
- **Nodes share only broker metadata - messages are not replicated among nodes!! - съобщението не се репликира в останалите node-ве**

Example:
A и B са DISK nodes, а C е Ram node.



Instance A node DISK

```
set RABBITMQ_NODENAME=instanceA &
set RABBITMQ_NODE_PORT=5770 &
set RABBITMQ_SERVER_START_ARGS=
      -rabbitmq_management listener [{port,33333}] &
rabbitmq-server.bat -detached
```

Instance B node DISK

Пускаме инстанция B, спираме я, присъединяваме я след това

```
set RABBITMQ_NODENAME=instanceB &
set RABBITMQ_NODE_PORT=5771 &
rabbitmq-server.bat -detached
rabbitmqctl.bat -n instanceB stop_app
rabbitmqctl.bat -n instanceB join_cluster instanceA@MARTIN
rabbitmqctl.bat -n instanceB start_app
```

Instance C node RAM

Пускаме инстанция C, спираме я, присъединяваме я след това

```
set RABBITMQ_NODENAME=instanceC &
set RABBITMQ_NODE_PORT=5772 &
rabbitmq-server.bat -detached
rabbitmqctl.bat -n instanceC stop_app
rabbitmqctl.bat -n instanceC join_cluster -ram instanceA@MARTIN
rabbitmqctl.bat -n instanceC start_app
```

- If a node that hosts a queue buffers unprocessed messages goes down, then messages are lost
- Default clustering mechanism provides scalability in terms of queues rather than high availability

Mirrored queues

- **Mirrored queues** are an extension to the default clustering mechanism that can be used to establish **high availability** at the broker level
- Mirrored queues provide queue replication over different nodes that allows a message to survive node failure
- Queue mirroring is establishing by means of a mirroring policy that specifies:
  - Number of nodes to use for queue replication
  - Particular nodes designated by name for queue replication
  - All nodes for queue replication

- The node where the queue is created is the master node - all other nodes are slaves
- A new master node can be promoted in case the original one goes down
- A slave node is promoted to/as the new master in case it is fully synchronized with the old master

Example:

Let's define the test queue in the cluster and mirror it over all other nodes:

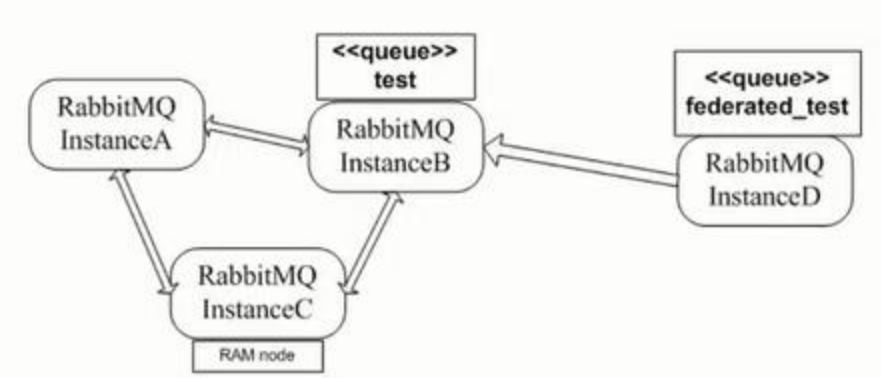
```
rabbitmqadmin.py -N instanceA declare queue name=test
durable=false
rabbitmqctl -n instanceA set_policy ha-all "test" "{""ha-
mode"":""all""}"
```

Federation and Shovel plugins

- The RabbitMQ clustering mechanism uses Erlang message passing along with a message cookie in order to establish communication between the nodes…….. which is **not reliable** over the Wide Area Networks!!
- In order to establish high availability among nodes in different geographic locations you can use the **federation, federation_management** and **shovel** plug-ins
- The shovel plug-in works at a lower level than the federation plug-in

Federation

- Ако искаме да репликираме опашката на отдалечена инстанция D:




```
set RABBITMQ_NODENAME=instanceD &
set RABBITMQ_NODE_PORT=6001 &
set RABBITMQ_SERVER_START_ARGS=
      -rabbitmq_management listener [{port,44444}] &
rabbitmq-server.bat -detached

rabbitmq-plugins -n instanceD enable rabbitmq_federation
rabbitmq-plugins -n instanceD enable
rabbitmq_federation_management
```

- Declare the **federated_test** queue

```
rabbitmqadmin.py -N instanceD -P 44444 declare queue
name=federated_test durable=false
```

Declare the upstream to the initial cluster and set a federation link to the **test** queue:

```
rabbitmqctl -n instanceD set_parameter federation-upstream
upstream
"{""uri"":""amqp://localhost:5770"",""expires"":3600000,
""queue"":""test""}"

rabbitmqctl -n instanceD set_policy federate-queue
--apply-to queues "federated_test"
"{""federation-upstream"":""upstream""}"
```

Shovel

The shovel plug-in provides two variants:

- **static** - all links between the source/destination nodes/clusters are defined statically in the RabbitMQ configuration file
- **dynamic** - all links between the source/destination nodes/clusters are defined dynamically via the RabbitMQ parameters

source \ destination	exchange	queue
exchange	federation dynamic shovel	dynamic shovel
queue	static shovel dynamic shovel	federation dynamic shovel

Integrations

Info

- RabbitMQ provides integrations with other protocols such as STOMP, MQTT and LDAP by means of RabbitMQ plug-ins
- Using the Java Client - already discussed above
- The Spring framework provides integration with AMQP protocol and RabbitMQ in particular
- The **Spring AMQP framework** provides:
  - **RabbitAdmin** class for automatically declaring queues, exchanges and bindings
  - **Listener container** for asynchronous processing of inbound messages
  - **RabbitTemplate** class for sending and receiving messages

- Utilities of the Spring AMQP framework can be used directly in Java or preconfigured in the Spring configuration
- The **Spring Integration framework** to **Spring Boot** provides adapters for the AMQP protocol

- Integration with Quarkus framework

Spring AMQP framework

```
<dependencies>
    <dependency>
        <groupId>org.springframework.amqp</groupId>
        <artifactId>spring-rabbit</artifactId>
        <version>1.4.5.RELEASE</version>
    </dependency>
</dependencies>
```

The **RabbitAdmin** class:

```
CachingConnectionFactory factory = new
       CachingConnectionFactory("localhost");
RabbitAdmin admin = new RabbitAdmin(factory);
Queue queue = new Queue("sample-queue");
admin.declareQueue(queue);
TopicExchange exchange = new TopicExchange("sample-topic-
       exchange");
admin.declareExchange(exchange);
admin.declareBinding(BindingBuilder.bind(queue).to(exchange)
       .with("sample-key"));
factory.destroy();
```

Listener container

```
CachingConnectionFactory factory =
       new CachingConnectionFactory(
"localhost");
SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer(
       factory);
Object listener = new Object() {
    public void handleMessage(String message) {
    System.out.println("Message received: " +
    message);
}};
MessageListenerAdapter adapter = new
       MessageListenerAdapter(listener);
container.setMessageListener(adapter);
container.setQueueNames("sample-queue");
container.start();
```

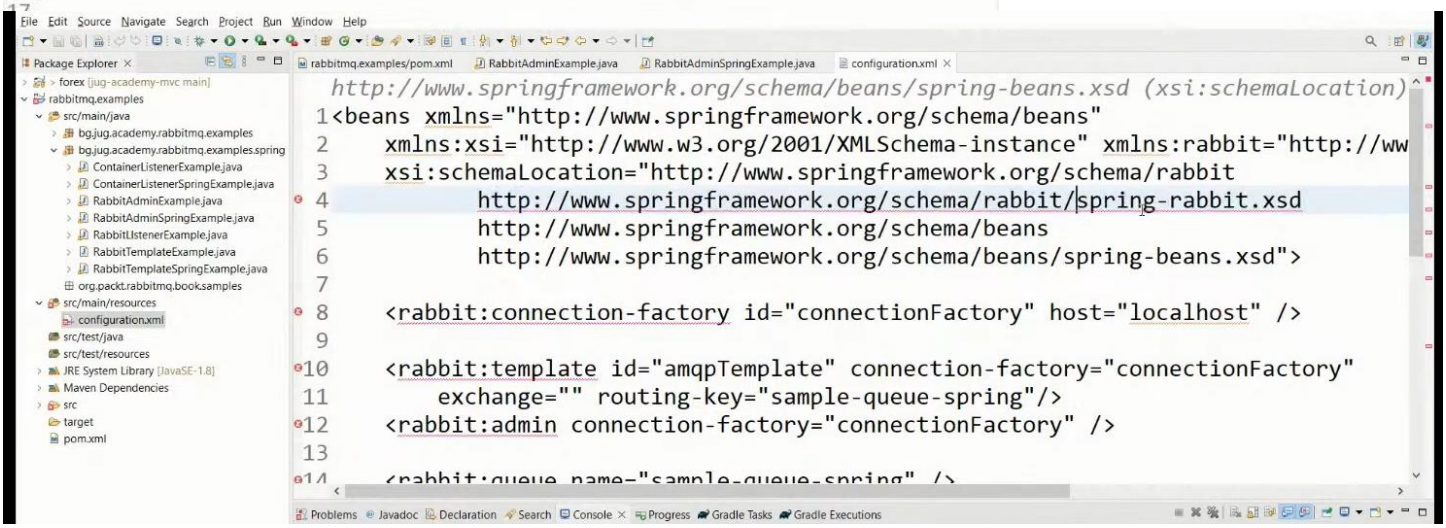
The **RabbitTemplate** class:

```
CachingConnectionFactory factory =
       new CachingConnectionFactory("localhost");
RabbitTemplate template = new
RabbitTemplate(factory);
template.convertAndSend("", "sample-queue",
        "sample-queue test message!");
```

- All of the above Spring AMQP framework examples can be configured using the Spring configuration - so that to be cleaner and to decouple RabbitMQ configuration from the business logic/Java code

For example within a **configuration.xml** file:

```
7 public class RabbitAdminSpringExample {
8
9     public static void main(String[] args) {
10
11         AbstractApplicationContext context = new ClassPathXmlApplicationContext(
12                 "configuration.xml");
13         RabbitAdmin admin = context.getBean(RabbitAdmin.class);
14     }
15
16 }
```



Spring Boot Starter AMQP - Spring Integration framework

In gradle

implementation 'org.springframework.boot:**spring-boot-starter-amqp**'

Предоставя ни бийнове за **RabbitAdmin** class, **Listener container** and **RabbitTemplate** class.

Посредством дефиниране на бийнове - можем да си декларираме **exchange, queue** или **queueBinding**

```
@Component
public class EventingServiceImpl implements EventingService {

    private final RabbitTemplate template;

    public EventingServiceImpl(RabbitTemplate template) {
        this.template = template;
    }

    @Bean
    public Queue createExchangeQueue() {
        return new Queue("exchange_rate_queue");
    @Override
    public void publish(String message) {
        template.convertAndSend("exchange_rate_queue", message);
    }

}
```

Quarkus framework

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-smallrye-reactive-messaging-rabbitmq</artifactId>
</dependency>
```

In application.properties file

```
mp.messaging.outgoing.bi.use-ssl=true
mp.messaging.outgoing.bi.connector=smallrye-rabbitmq
mp.messaging.outgoing.bi.exchange.type=direct
mp.messaging.outgoing.bi.port=5672
```

```
import io.smallrye.reactive.messaging.rabbitmq.OutgoingRabbitMQMetadata;

@Inject
@Channel("asd")  //from Microprofile
Emitter<String> emitter;  //from Microprofile

public void emmitMessage(RabbitMqPayload payload) {
   String messagePayload = JsonUtils.toJsonString(List.of(payload));
   LOGGER.debugf("Emitting Bi message to RabbitMQ %s", messagePayload);
   OutgoingRabbitMQMetadata metadata = new OutgoingRabbitMQMetadata.Builder()
           .withRoutingKey(payload.routingKey())
           .build();

   Message<String> message = Message.of(messagePayload, Metadata.of(metadata));
   biEmitter.send(message); //from Microprofile
   LOGGER.infof("Bi message emitted to route %s", payload.routingKey());
}
```

Security

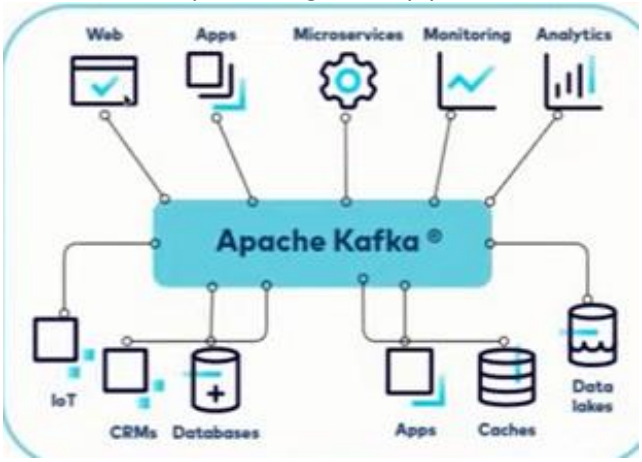
- RabbitMQ uses SASL Simple Authentication Security Layer for authentication (SASL PLAIN used by default)
- RabbitMQ uses access control lists (permissions) for authorization
- SSL/TLS support can be enabled for the AMQP communication channels
- SSL/TLS support can be enabled for node communication between nodes in a cluster
- SSL/TLS support can be enabled for the federation and shovel plug-ins

II. Apache Kafka

Book
Kafka – The Definitive Guide – Real-Time Data and Stream Processing at Scale

Use cases

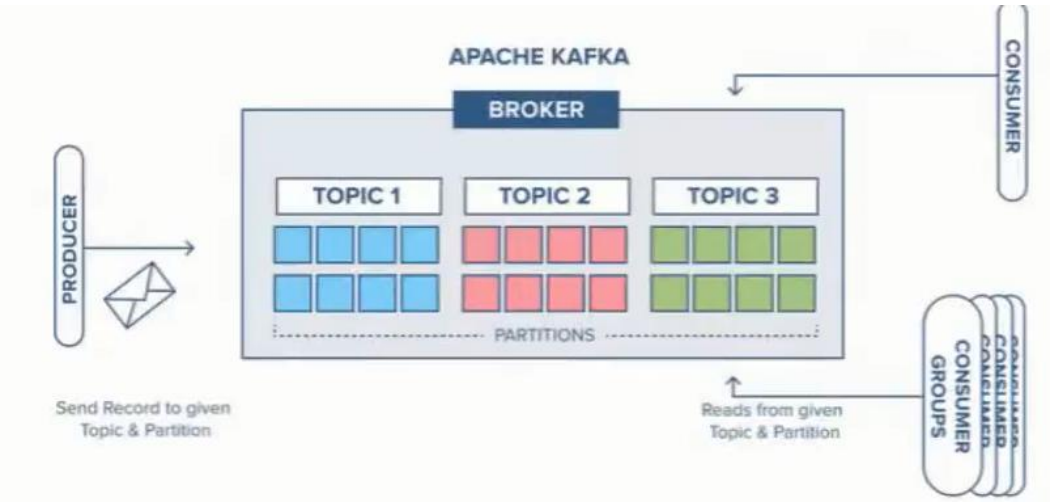
- For activity tracking – например LinkedIn да знае кой профил гледате/се гледа най-много
- Messaging
- Metrics and logging
- Commit log
- Stream processing – data pipeline – за един ден колко човека са кликнали на моята страница



Brokers and Clusters

A single Kafka server/instance is called a ***broker***. The broker receives messages from producers, assigns offsets to them, and writes the messages to storage on disk. It also services consumers, responding to fetch requests for partitions and responding with the messages that have been published. Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands and millions of messages per second.

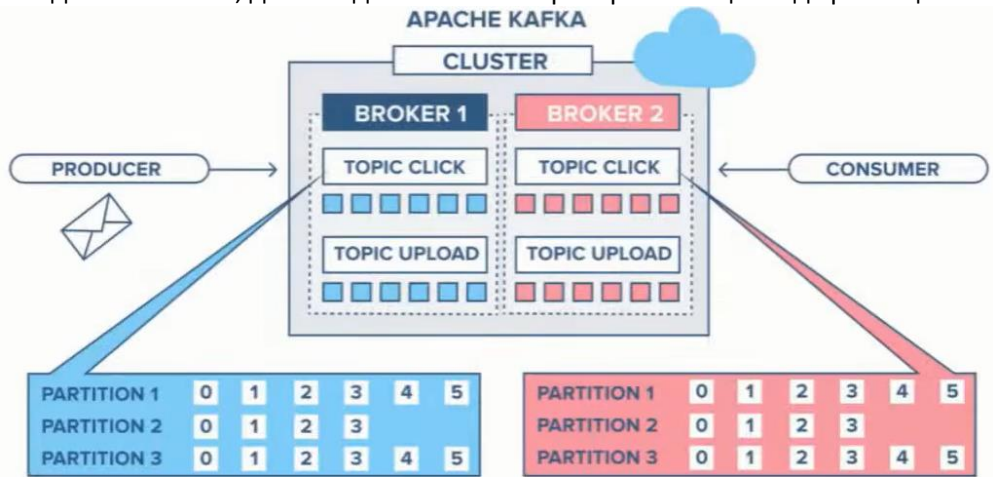
В един Kafka инстанция може да има 1 или повече топици, а във всеки топик има определен брой partitions.



Cluster – комбинация от няколко брокера (поне 3), т.е. няколко брокера които работят заедно и се координират заедно.

С други думи да има Fault tolerance, и когато един broker instance падне/или целият cluster падне, то да има опция да се репликират данните.

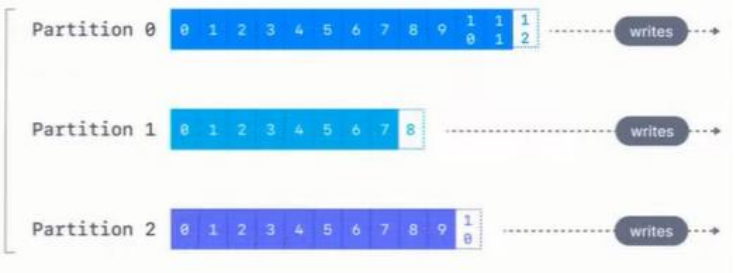
Като си правим cluster, то самите broker instances да бъдат географски на различни места – че ако падне нета на единия instance, да не паднат всичките брокер инстанции и дефакто целия клъстер.



Topics and Partitions

- With key – Messages are appended in the same partition if they have the same key
- Without key – messages are appended to the next partition in a round-robin fashion
- The order is guaranteed only in a partition and only if we have/use **key**

Kafka topic – основната единица



Колкото partitions има в даден топик, то толкова паралелни consumers може да имаме.

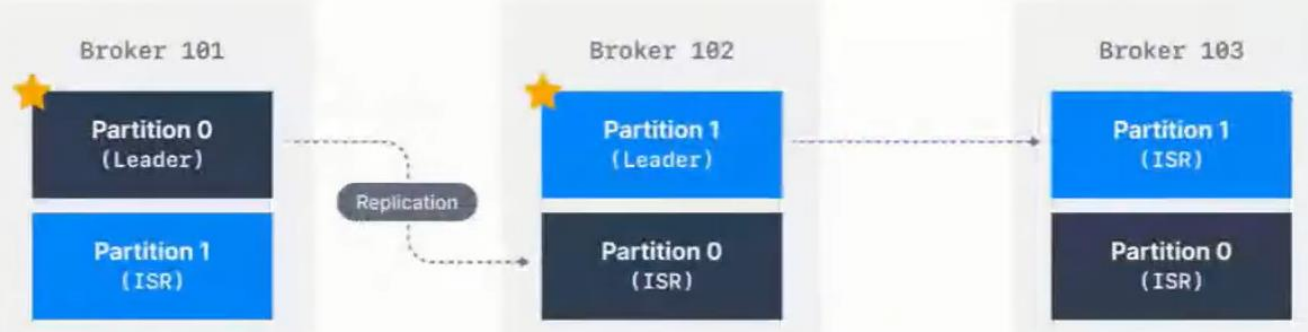
Ако имаме повече от 1 partition в даден топик, то нямаме гаранция за подредба. Освен разбира се ако не използваме **key!**

Partition представлява множество съобщения/елементи едно след друго, с определен индекс който наричаме **offset!**

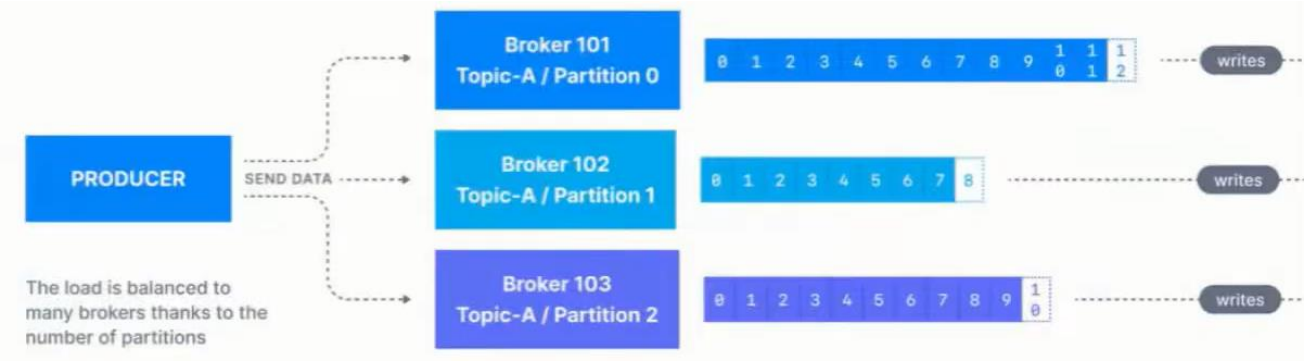
Topic replication

- **min.insync.replicas** – how many brokers (including the leader) should have the data before it is considered stored and ready to serve. Recommended is **replication.factor -1** (можем да настроим да я има тази информация на само 1 брокер, на още 2 брокера, и т.н. – зависи от use case-а и от наличните брокери)

Each partition is replicated separately, has its own broker leader and ISR.
Обикновено единия брокер се води Leader на partition-а.

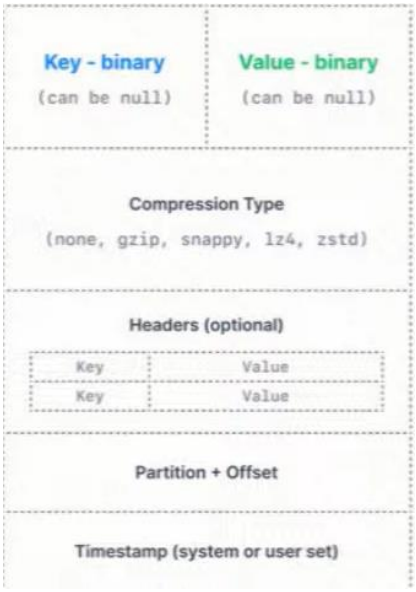


Когато изпращаме данни, ние изпращаме данни към съответния Leader на съответния partition.



What is a Kafka message

И key и value са **binary**! За Кафка всичко е byte-ове.
Възможност за компресия. Запазват се sequential – на следващия сегмент на диска.
Headers – например информация през кои сървиси е минало
Информация на кой partition е, и на кой офсет индекс е даденото съобщение.
Timestamp by default се слага от producer-а, а не вътрешно от Кафка брокера!

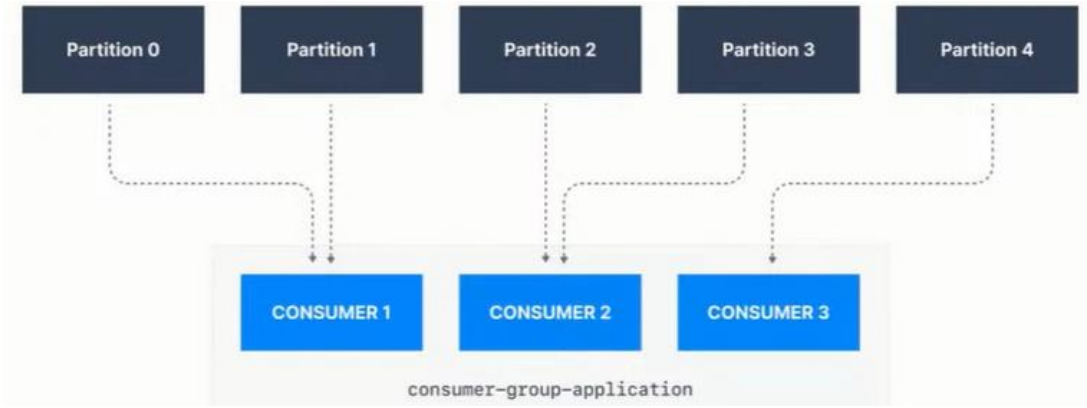


Consumers

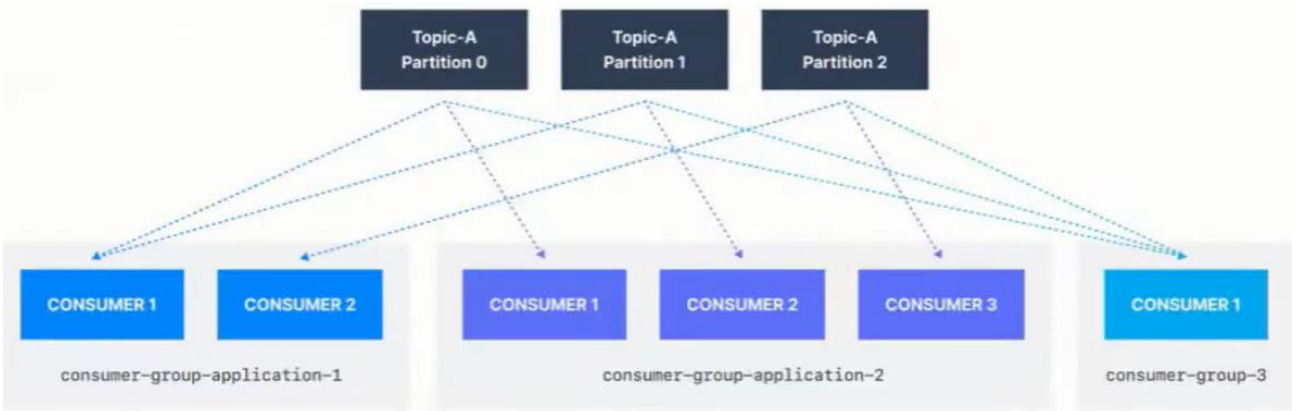
Kafka скалира не на базата на топици, а на базата на partitions!!!

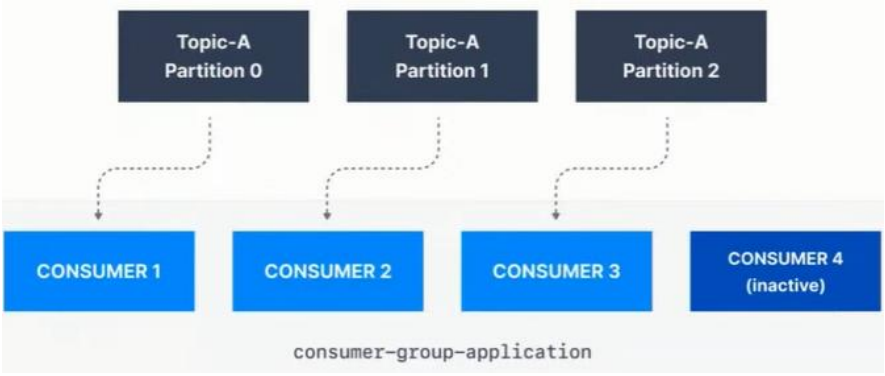
Consumer group

Пример: Имаме 5 партишъни. И разпределяме партишъните на различни consumers.
В текущият пример можем да сложим общо макс 5 consumers!!! Или с други думи правилото е че можем да скалираме consumers до броя на partitions!



Many consumers groups

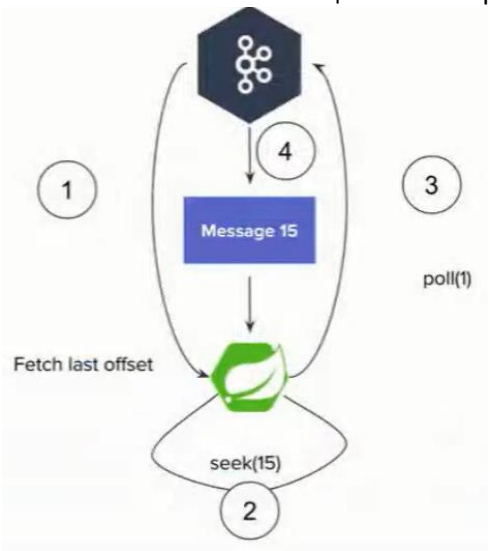




Commit

- Commit is the process of telling Kafka to remember that you have processed **(both produced and consumed)** the messages of a partition to a specific offset
- When first started, the consumer looks up where it left off via the kafka commit facility
- A call to **seek()** is made to initialize an internal counter which determines the messages that will be poll()-ed for the lifetime of the **consumer**

Poll означава вземи съобщението от брокера, и го процесни при теб (при Spring framework-а например)



Producer

Batching

spring.kafka.producer.batch-size=100
Sets the maximum number of records to be sent om one request. Sends the batch as soon as it is filled.

spring.kafka.producer.properties[linger.ms]
Gives the batch that much ms to be filled and sent automatically (see above). After the duration, if the batch is still not filled, sends it as it is.

spring.kafka.producer.buffer-memory
If a batch cannot be sent due to broker being down or whatever other recoverable issue, fills new batches in the buffer. After the buffer is filled, the **send()** method blocks for **spring.kafka.producer.properties[max.block.ms]**. This buffer **must be large enough** to contain at least one full batch or such batches will be lost.

spring.kafka.producer.compression-type=lz4 avro gzip snappy

The batch will be compressed before sending. This directly increases throughput and latency (not much latency)
Следните неща ни интересуват като изпращаме съобщения:

- **acks=0**

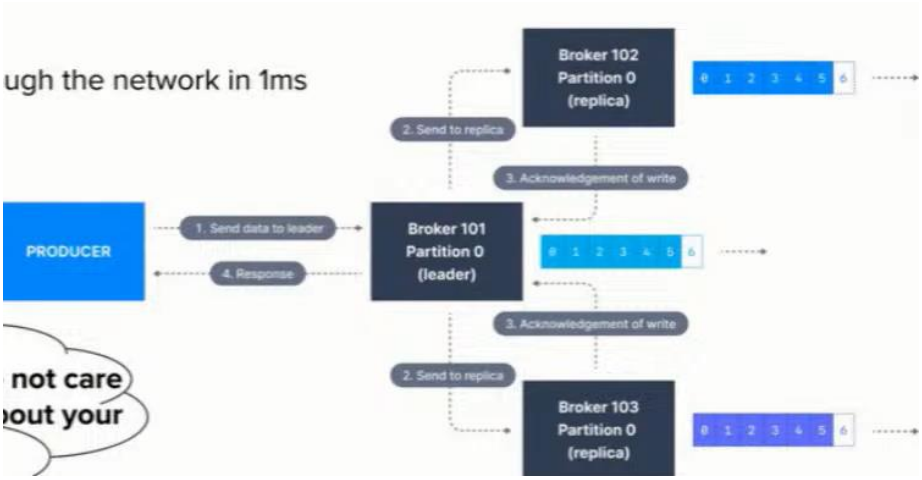
Does not wait for a response to consider the sent successful

- **acks=1**

Wait for the leader to commit the message/s but do not wait for the replicas to commit!!
Изчакай поне Leader да ти върне, че е записано съобщението, но не чакаме да се случи репликацията. Проблем би бил ако leader-а умре/падне, и тогава съобщението се губи

- **acks=all**

Wait for the leader and all replicas(или еди колко си на брой реплики) to commit, so that to consider the message/s sent.
Note: all == -1



x2 slower – докато се получат повтържденията за репликите, и се забавя
Assuming that a message/s is sent through the network in 1ms. This leads to 2.5ms latency.
1000 / 2ms = 2micro s per message

Consumer

Batching again

spring.kafka.consumer.max-poll-records=500
Sets the maximum number of records that will be returned by a poll()

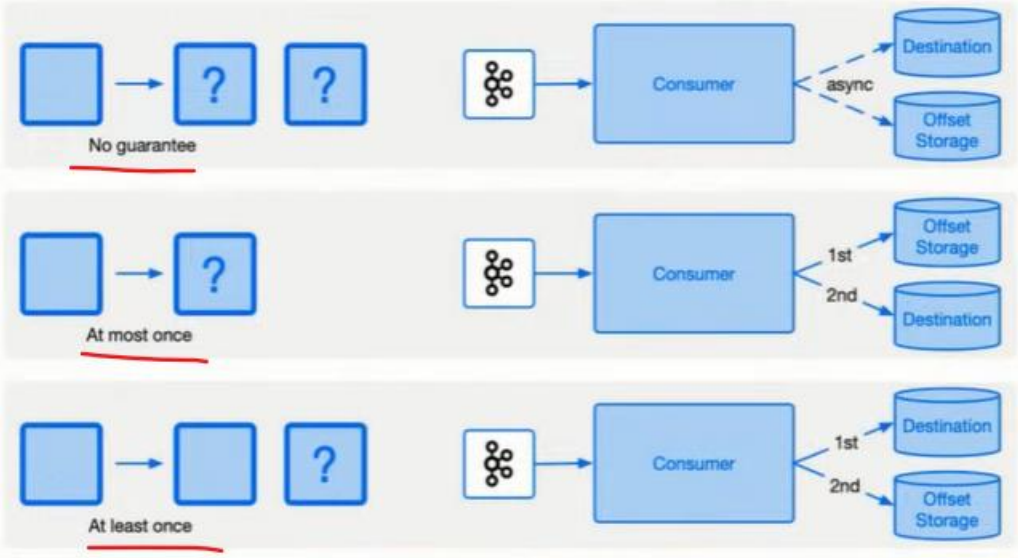
spring.kafka.listener.ack-mode=batch
Auto commit is mostly evil. Don't use it by default unless actually verifiably ok for your data
Като процеснеш batch-а, то commit-ни ръчно тогава offset-а. Ако се закача наново, то да не го процесна същото съобщение.
spring.kafka.consumer.enable-auto-commit=false

max.poll.interval.ms=5min
You must be able to process a full batch in that time or a rebalance will be triggered. На колко време да вземаме съобщенията обратно.

Delivery guarantees

Това съобщение защо сме го процеснали 2 пъти.
Или това съобщение защо изобщо не сме го процеснали.

At **least once** means not that a single message might be duplicated but a whole batch it is scary with big batches 😊

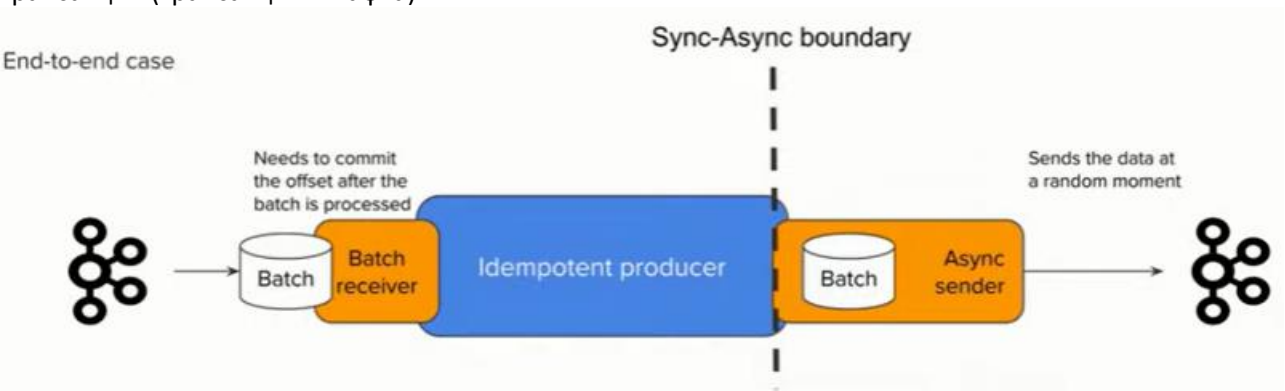


Idempotent producer – получаване на дуплицирани съобщения
Поради network проблем може Кафка брокера да е приел съобщението и да не отговори, и ние да изпратим съобщението наново.

Idempotent producer
spring.kafka.producer.properties[max.in.flight.requests.per.connection]=5
spring.kafka.producer.properties[enable.idempotence]=true
It works for the lifespan of the producer. Required but not enough for proper idempotency across restarts.

Ако е Network проблем, това решава дуплицирането.

Consumer – producer problem
Ако чета в Кафка, и пиша в Кафка – имам проблем. Защото трябва да имаме/използваме нещо, което се казва трансакции (трансакции в Кафка).



kafkaTemplate.send(msg)
This is fully async. You don't know when the data is actually going to be sent. The method returns a future that can be waited but to do it for each message breaks the batching and is extremely slow.

Demo with kafka-clients

build.gradle – Groovy style

```
dependencies {
    implementation group: 'org.apache.kafka', name: 'kafka-clients', version: '3.6.1'
    //SLF4J Api
    implementation 'org.slf4j:slf4j-api:1.7.32' //Use the latest version available

    //Logback (SLF$J implementation)
    implementation 'ch.qos.logback:logback-classic:1.2.6' //Use the latest version
available

}
```

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class ProducerExample {
    public static void main(String[] args) {
        Properties properties = new Properties();
```

```
        properties.put("bootstrap.servers", "localhost:29092");
        properties.put("key.serializer", "org.apache.kafka.common.serialization");
        properties.put("value.serializer", "org.apache.kafka.common.serialization");

        Producer<String, String> producer = new KafkaProducer<>(properties);

        producer.send(new ProducerRecord<>("bgjug", "KafkaKey", "Welcome to Kafka"));
        producer.close();
    }
}
```

```
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class ConsumerExample {

    private static final Logger logger = LoggerFactory.getLogger(ConsumerExample.class);

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:29092");
//        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "your.group.id");
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "svilen");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        Consumer<String, String> consumer = new KafkaConsumer<>(properties);
        consumer.subscribe(Collections.singletonList("bgjug"));

        while (true) {
            ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));

            records.forEach( record -> {
                logger.info("Consumed message - Topic: {}, Partition: {}, Offset: {},
Key: {}, Value: {}",
                        record.topic(), record.partition(), record.offset(),
record.key(), record.value());
            });
        }

    }
}
```

Demo with Spring