Queuing System

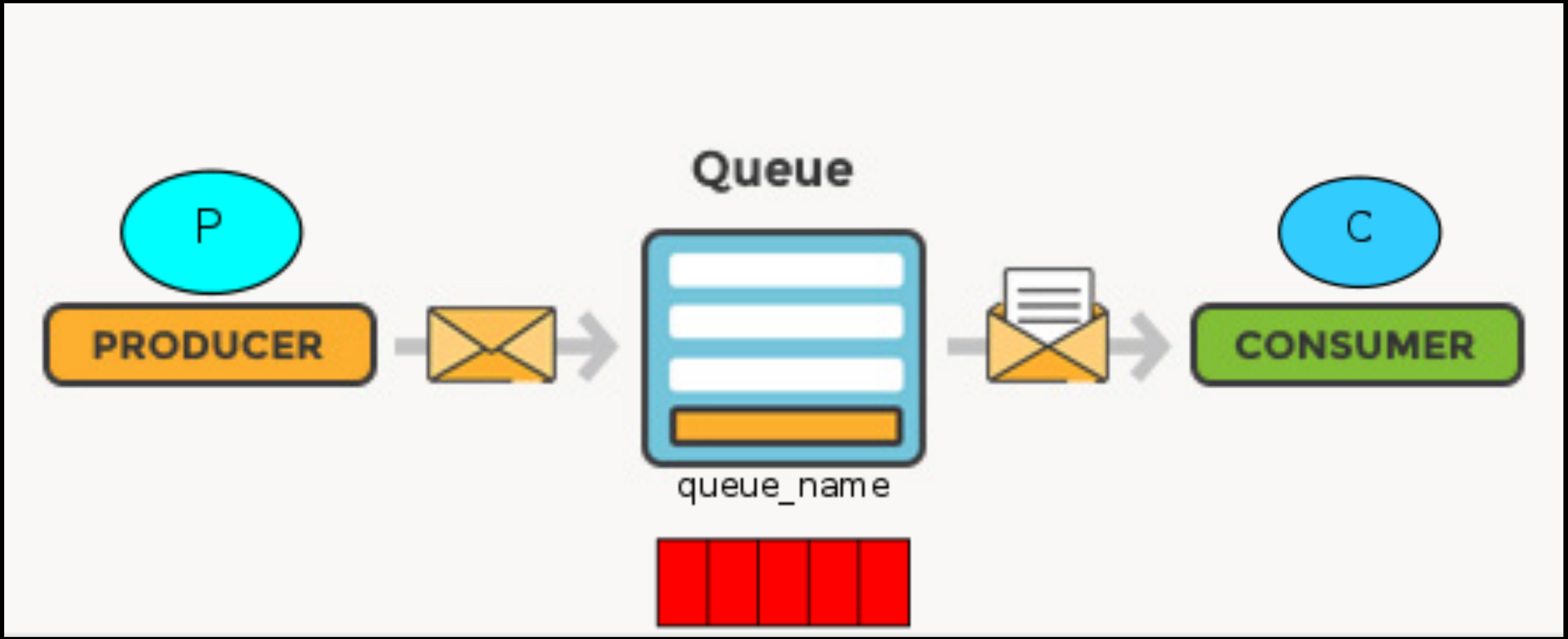
Friday Presentation

Shrijan Subedi, 16th October 2020

What is Message Queing?

- Message queuing is a way of communication between application by passing small units of information called messages.
- A message queue provides an **asynchronous communications protocol i.e.** a message is loaded in a message queue and does not require an immediate response to continuing processing.
- Example: Email delivery system,

Architecture of a Queuing System



- Producers = client applications that create messages and deliver them to the message queue
- Consumers = connects to the queue and gets the messages to be processed.
- Producer, consumer, and broker do not have to reside on the same host. An application can be both a producer and consumer, too.

Architecture of a Queuing System

- A **queue** is a line of messages waiting to be consumed. It includes a sequence of work objects waiting to be processed.
- A queue is only bound by the host's memory & disk limits.
- A **message** is the data transported between the sender and the receiver application
- It's essentially a byte array with some headers at the top.
- Example: Message to start processing a task, Message containing result of a task

Use cases

Why would you require a message queue?

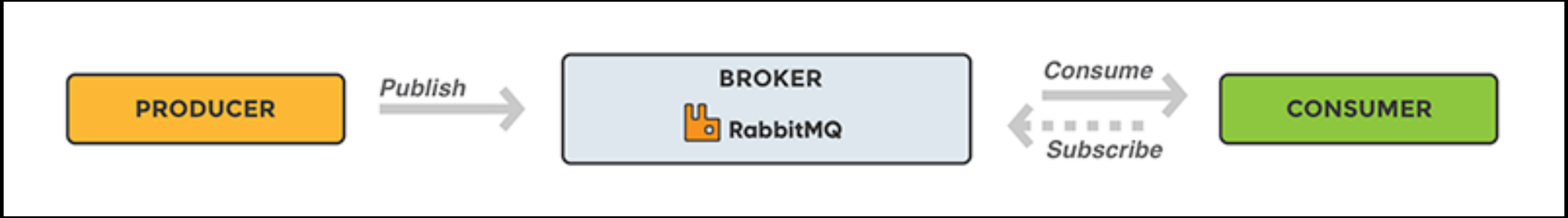
- **Decoupling and Scalability**: We can have a decoupled application system where the components communicate asynchronously using message queues. Communication will not depend on nature of the systems. Individual components can also be scaled without regard for other systems.
- **Fault Tolerance and Availability**: Placing a queue between a web service and the processing service is ideal when we require high availability and fault tolerance. If one process in a decoupled system fails to process messages from the queue, other messages can still be added to the queue and be processed when the system has recovered. This guaranties availability.
- **Responsive systems**: Asynchronous nature of the queuing system allows certain parts of the application to process while making other parts available to the user
- **Load balancing**: Message queueing is also good when you want to distribute a message to multiple consumers or to balance loads between workers.

Where do Message Brokers fit in?
RabbitMQ, IBM MQ, Amazon MQ, Kafka

- A **message broker** is an intermediary that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver.
- It mediates communication among applications, reducing the need of awareness application must have of each other i.e decoupling.
- The primary purpose of a broker is to take incoming messages from applications and perform some action on them.

Where do Message Brokers fit in?

- Using a real world analogy, it acts a post office; or rather a post box, a post office and a postman as a whole.
- It doesn’t deal with papers but it accepts, stores and forwards binary blobs of data called messages.



Where do Message Brokers fit in?

Functions performed by a message broker

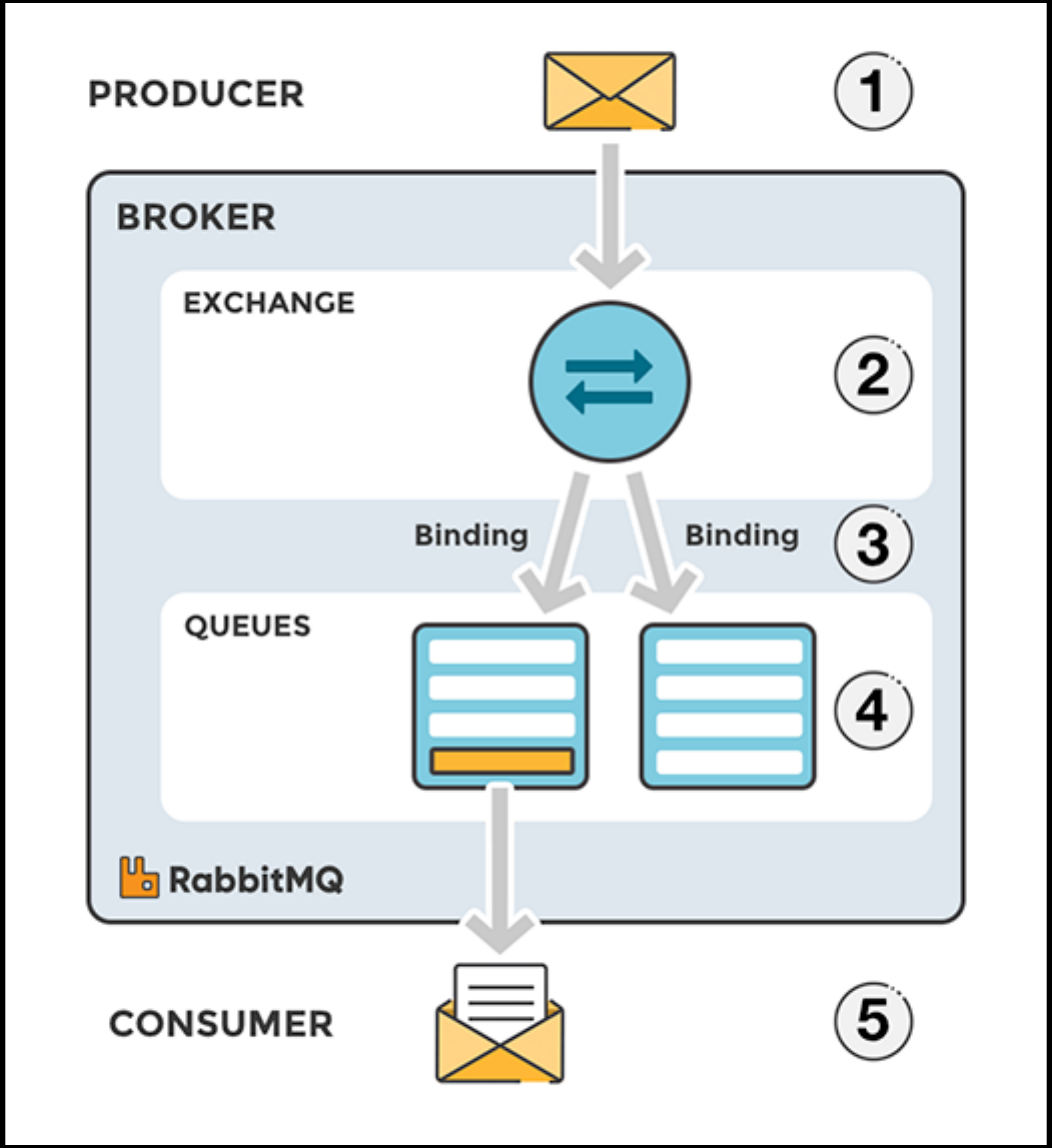
- Route messages to one or more destinations
- Respond to events or errors
- Invoke web services to retrieve data
- Perform message aggregation, decomposition.
- and so on..

RabbitMQ

- RabbitMQ is a open-source message broker: whose primary function is to accept and forward messages.
- It originally used AMQP but can be extended to STOMP, MQTT and other protocols.
- It has client libraries for many langugaes including Java, .NET, Erlang and Python
- Celery is such a client library for Python language.

RabbitMQ

Message Flow in RabbitMQ



RabbitMQ

Message Flow in RabbitMQ

- 1. The producer publishes a message to an exchange. When creating an exchange, the type must be specified.
- 2. The exchange receives the message and is now responsible for routing the message. The exchange takes different message attributes into account, such as the routing key, depending on the exchange type.
- 3. Bindings must be created from the exchange to queues. In this case, there are two bindings to two different queues from the exchange. The exchange routes the message into the queues depending on message attributes.
- 4. The messages stay in the queue until they are handled by a consumer
- 5. The consumer handles the message.

Exchanges

- Messages are not published directly to a queue; instead, the producer sends messages to an exchange.
- Exchanges are message routing agents, defined by the virtual host within RabbitMQ.
- An exchange is responsible for routing the messages to different queues with the help of bindings and routing keys.

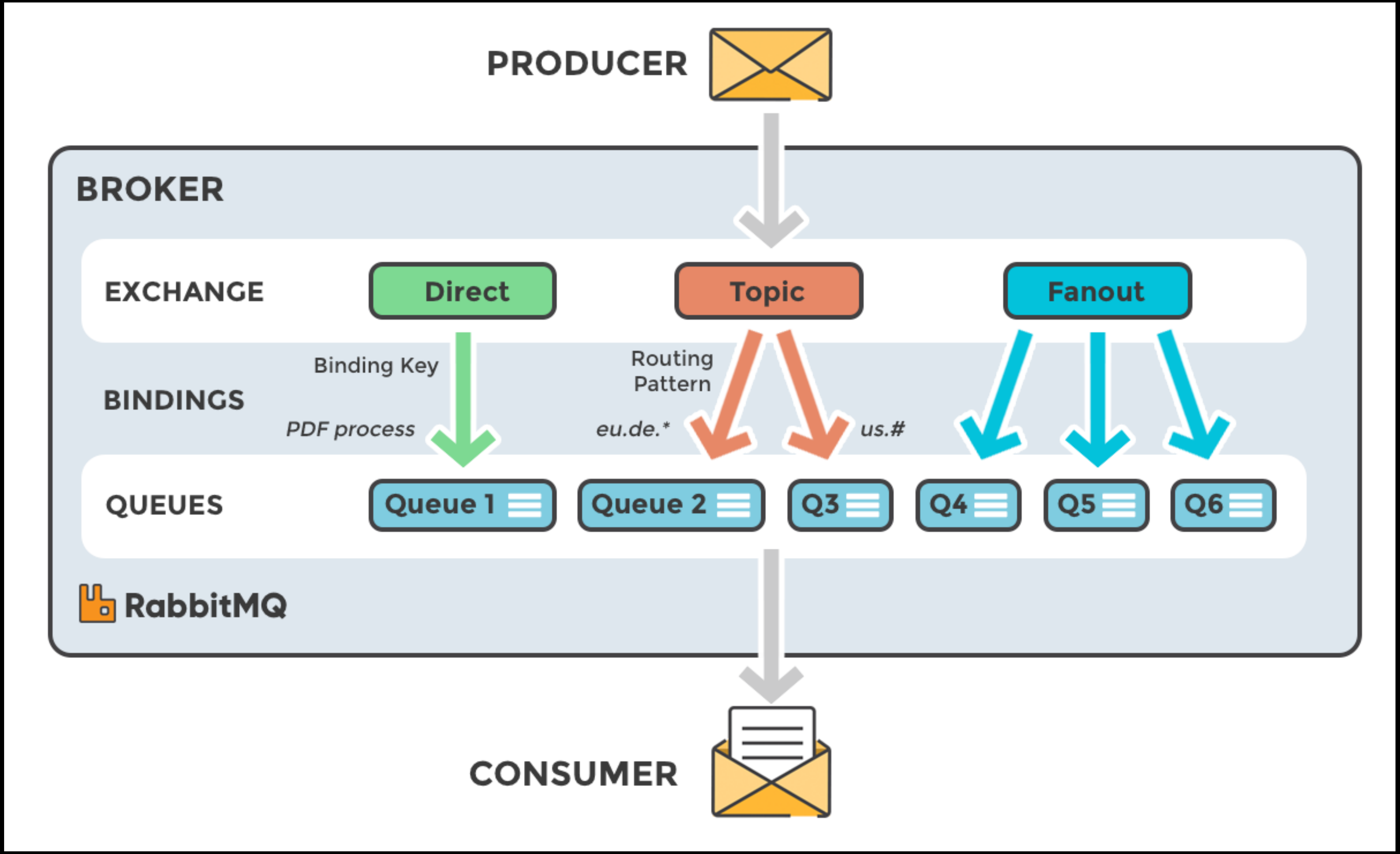
Exchanges

Binding and Routing keys

- A **binding** is a "link" that you set up to bind a queue to an exchange.
- The **routing key** is a message attribute the exchange looks at when deciding how to route the message to queues (depending on exchange type).
- Routing pattern is a pattern that is used in binding an exchange and queue. Topic exchanges used routing pattern and routing key to find correct queue.

Exchanges

Types of Exchanges



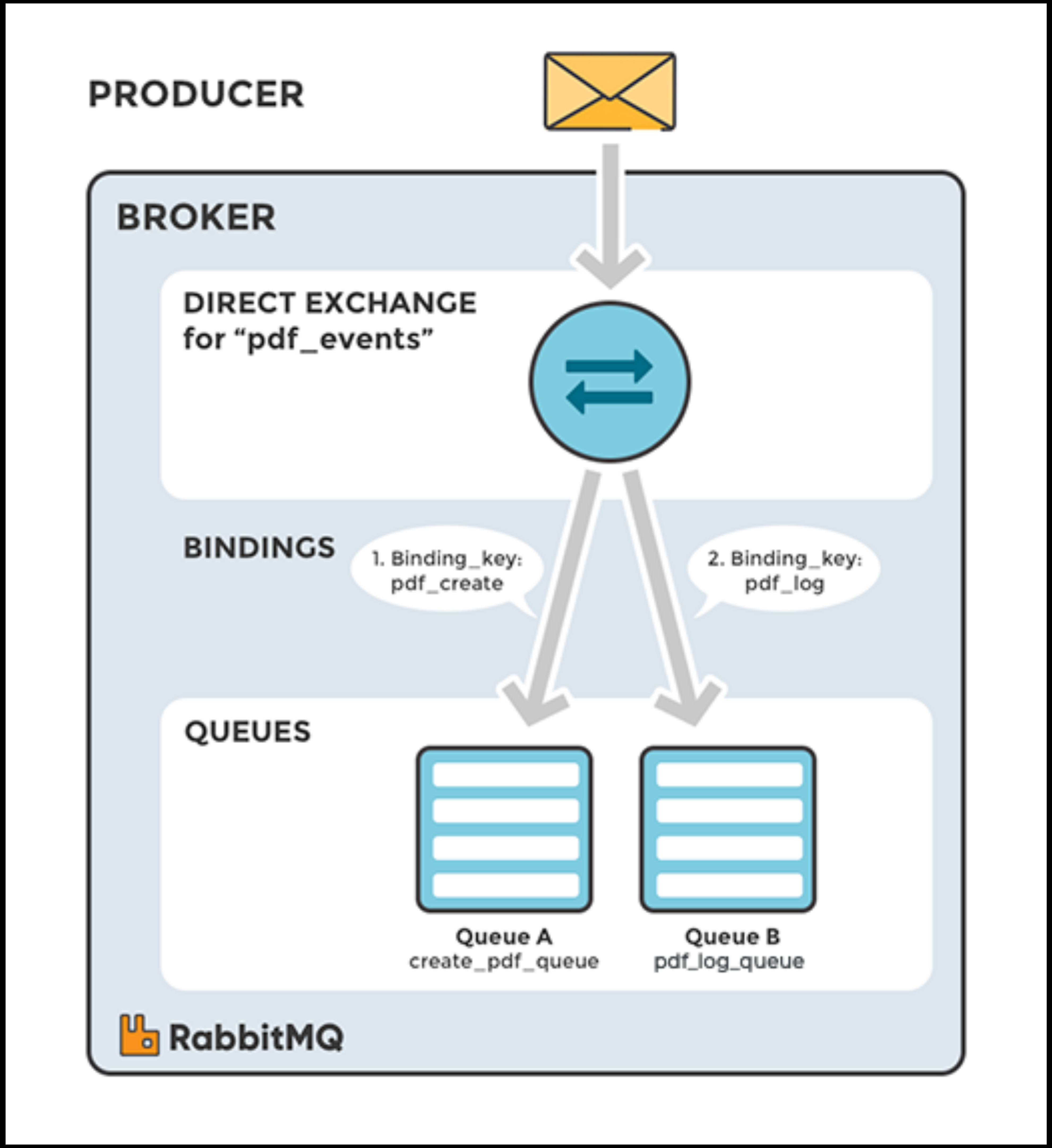
Exchanges

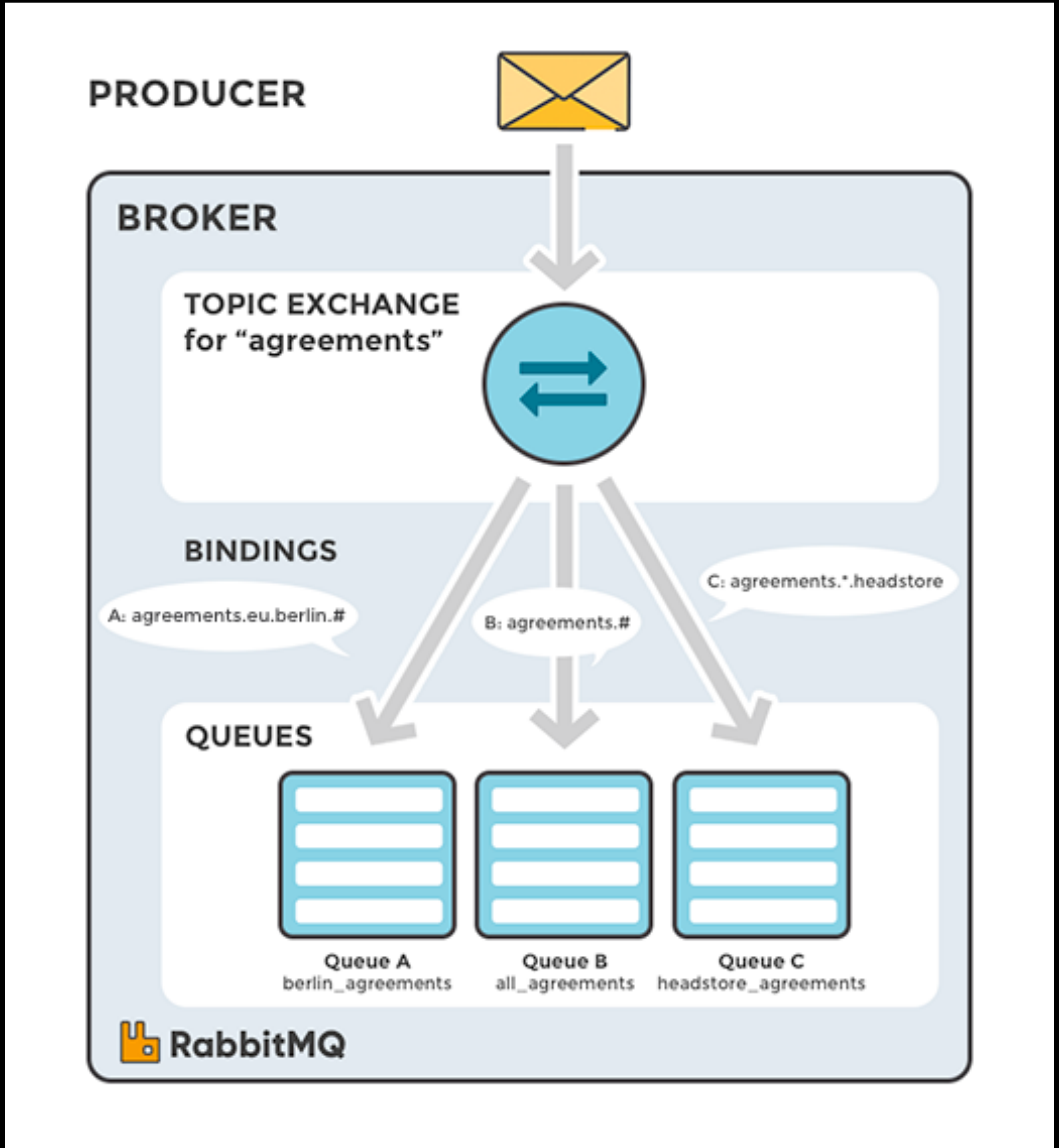
Types of Exchanges

- In RabbitMQ, there are four different types of exchanges that route the message differently using different parameters and bindings setups:
- Direct : message is routed to the queues whose binding key exactly matches the routing key of the message. Eg: if the queue is bound to the exchange with the binding key *pdfprocess, a message published to* the exchange with a routing key *pdfprocess is routed to that queue.*
- Fanout : A fanout exchange copies and routes messages to all of the queues bound to it.
- Topic : The topic exchange does a wildcard match between the routing key and the routing pattern specified in the binding.
- Headers : Headers exchanges use the message header attributes for routing.

Exchanges

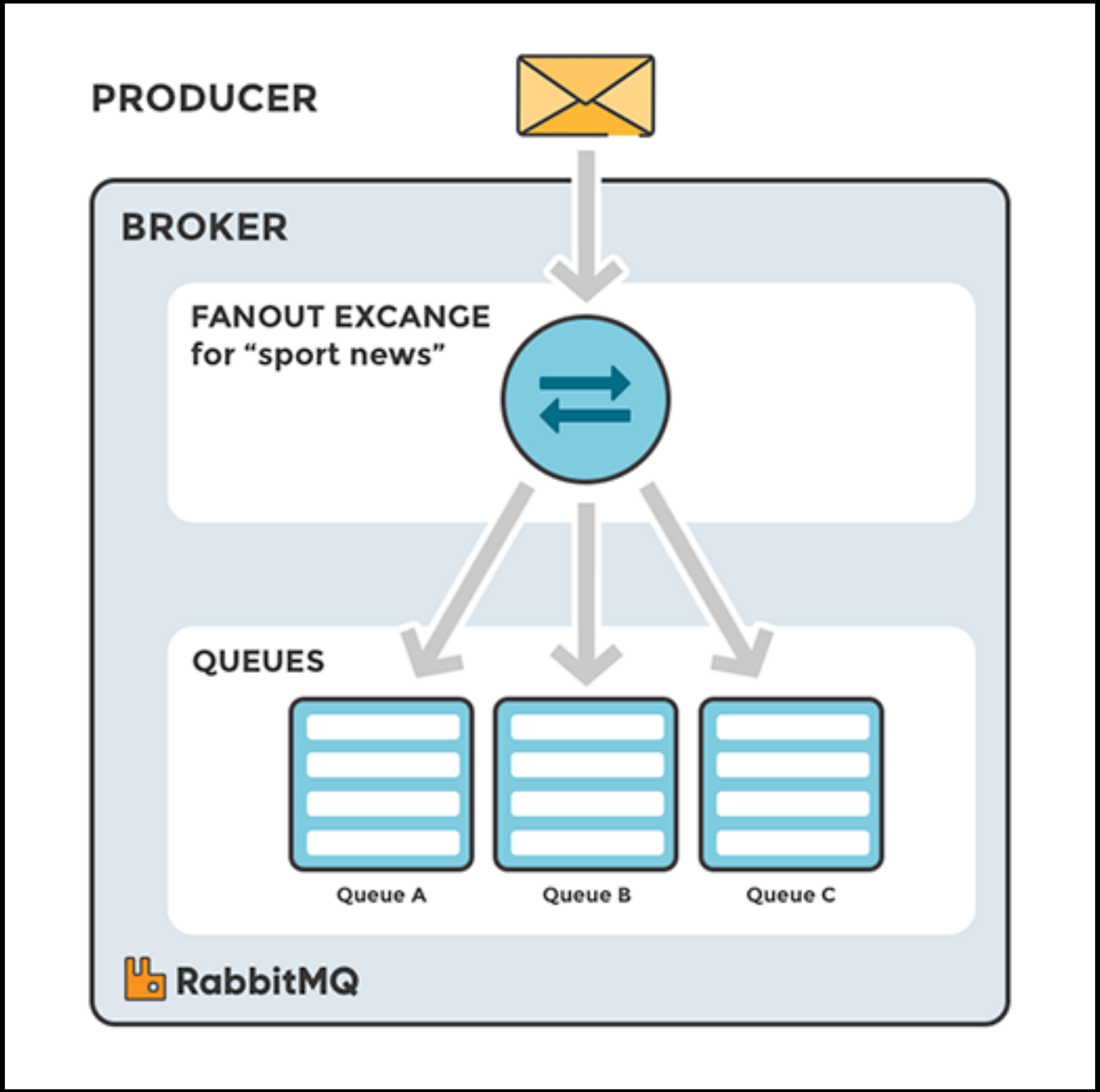
Types of Exchanges

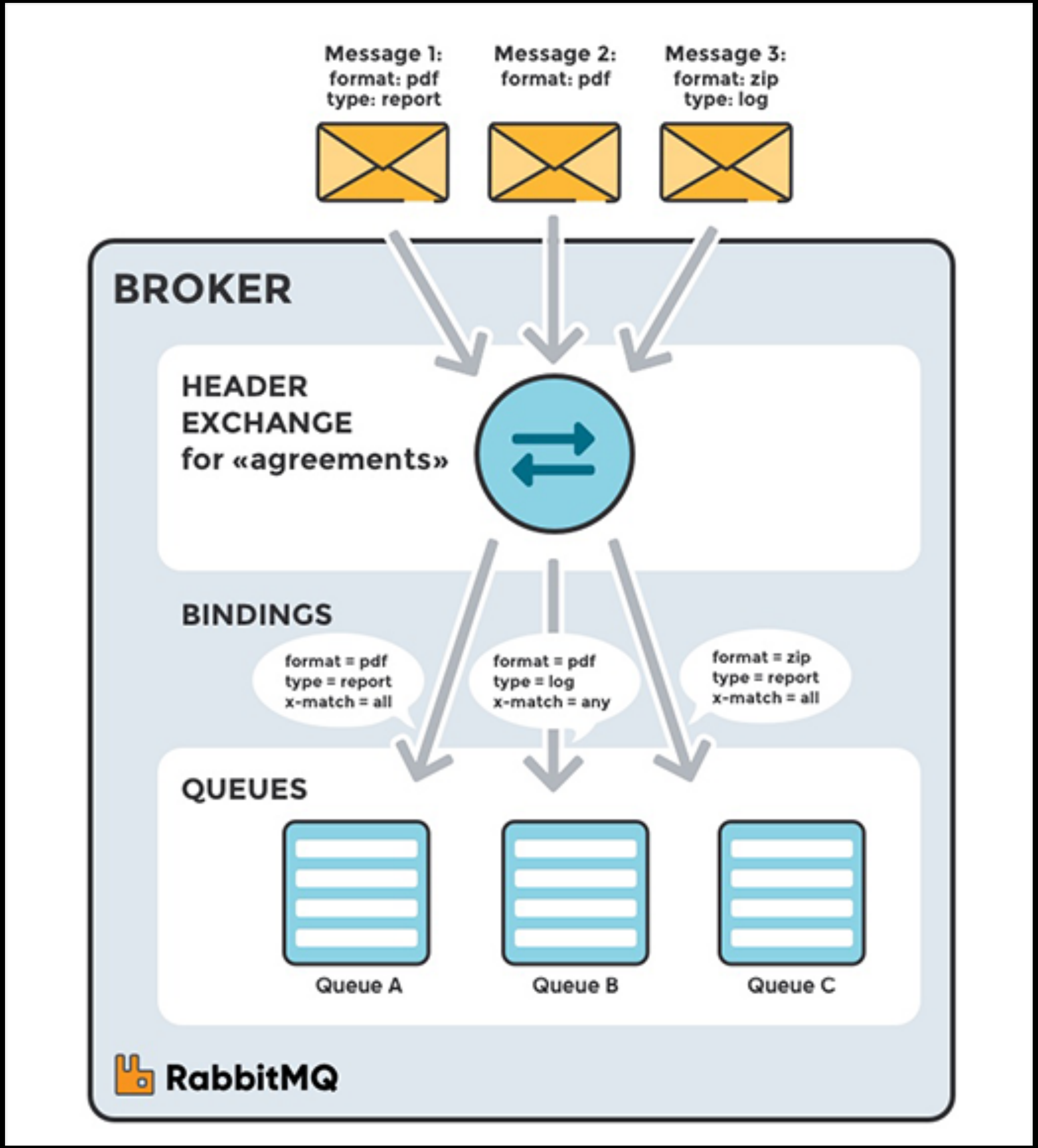



PRODUCER
BROKER
TOPIC EXCHANGE for "agreements"
BINDINGS
A: agreements.eu.berlin.#
B: agreements.#
C: agreements.*.headstore
QUEUES
Queue A berlin_agreements
Queue B all_agreements
Queue C headstore_agreements
RabbitMQ

Exchanges

Types of Exchanges





Exchanges

Message Durability

- Exchanges, connections, and queues can be configured with parameters such as *durable, temporary,* and *auto delete* upon creation.
- Durable exchanges survive server restarts and last until they are explicitly deleted.
- Temporary exchanges exist until RabbitMQ is shut down.
- Auto-deleted exchanges are removed once the last bound object is unbound from the exchange.

Terminologies

- Producer, Consumer, Queue, Message
- **Connection:** A TCP connection between your application and the RabbitMQ broker.
- **Channel:** A virtual connection inside a connection. When publishing or consuming messages from a queue - it's all done over a channel.
- Exchange, Binding, Routing key

RabbitMQ Management Interface

- The RabbitMQ Management is a user-friendly interface that let you monitor and handle your RabbitMQ server from a web browser.
- Queues, connections, channels, exchanges, users and user permissions can be handled - created, deleted and listed in the browser.
- You can monitor message rates and send/receive messages manually.

Celery

Task Queue Library for Python

- Celery is a distributed system to process vast amounts of messages, with included batteries for operations
- It’s a task queue with focus on real-time processing, while also supporting task scheduling.

Celery

Getting Started with Celery

* Installing Celery:

Celery needs to be installed

```
$ pip install celery
```

* Create an instance

The first thing we need is a Celery instance. This instance is used as the entry-point for everything we want to do in Celery, like creating tasks and managing workers.

```
-> app = Celery('tasks', broker=BROKER_URL)
```

1st argument is the module name, and 2nd argument specifies the url of the broker we intend to use.

Celery

Getting Started with Celery

* Create a task

We can use @app.task decorator to define a task

```python
@app.task
def add(x, y):
    return x + y
```

Each task that we define using the decorator will be runnable through the Celery instance

Celery

Getting Started with Celery

* Running the celery worker server

```
$ celery -A tasks worker —loglevel=info
```

The worker can be run in the background as a daemon using tools like supervisord

* Calling the Task

To call our task we can use the `delay()` method.

```
>>> from tasks import add
>>> add.delay(parameters_to_task)
```

Celery

Getting Started with Celery

* Keeping the results

We can use dedicated backends to keep track of the tasks’ states. We use an optional argument to the celery instance to define a results backend

app = Celery(‘tasks', backend=‘rabbitmqqt://localhost' ,broker=BROKER_URL)

Now we can store the AsyncResult instance when calling a task

>>> result = add.delay(**parameters**)

ready() returns whether task has finished processing or not

>>> result.ready()

get() returns the result of the task

>>> result.get()

Celery

Configuration

There are many options that can be configured to make Celery work exactly as needed. For configuring many settings at once we can use update

```
app.conf.update(
    task_serializer='json',
    accept_content=['json']
    result_serializer='json',
    timezone='Europe/Oslo',
    enable_utc=True,)
```

To verify configuration we can use

```
>>> python -m config_module_name
```

Celery

Configuration

You can tell your Celery instance to use a configuration module by calling the app.config_from_object() method

```
>>> app.config_from_object(‘celeryconfig’)
```

celeryconfig.py:

```
broker_url = 'pyamqp://'
result_backend =‘rpc://'

task_serializer = 'json'
result_serializer = 'json'
accept_content = ['json']
```

Celery

Configuration Examples

We can specify many behaviors in configuration module:

Routing a misbehaving task:

```
>>> task_routes = {
    'tasks.add': 'low-priority',
}
```

Limiting the no of tasks per minute

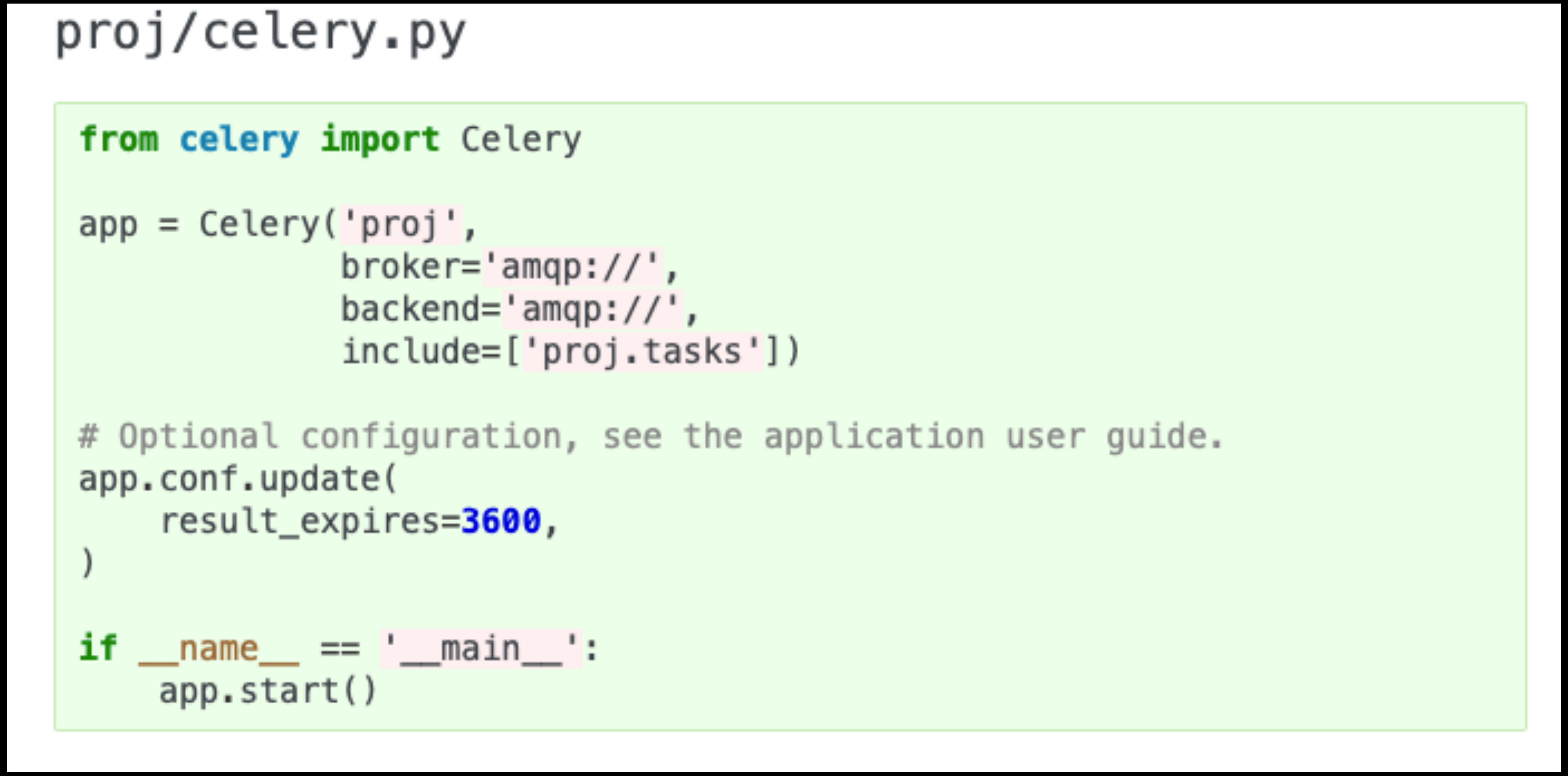
```
>>> task_annotations = {
    'tasks.add': {'rate_limit': '10/m'}
}
```

Celery

Using Celery in an Application

- To use celery in our application we might have the following layout structure:

proj/__init__.py
    /celery.py
    /tasks.py



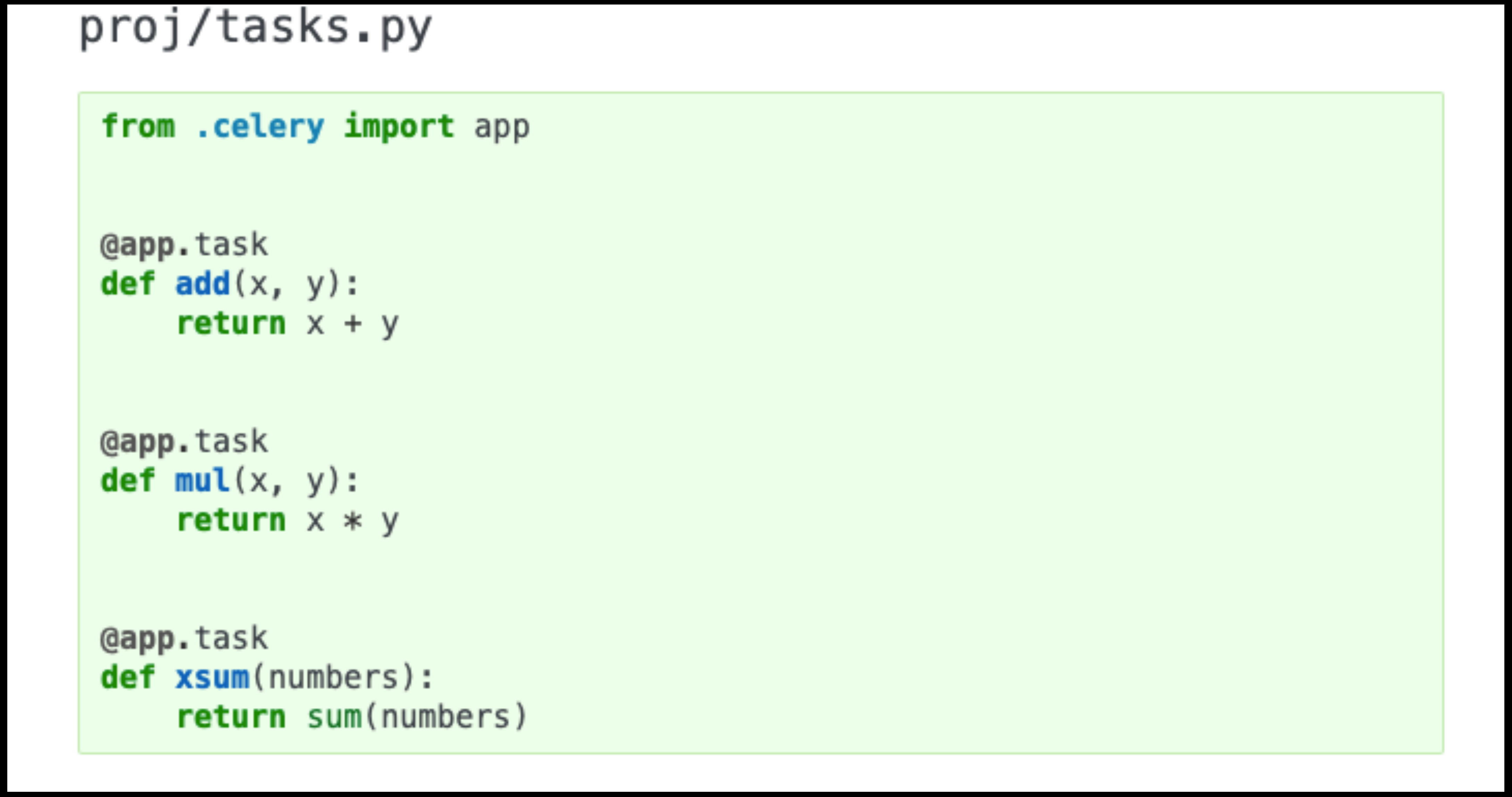
proj/celery.py

```python
from celery import Celery

app = Celery('proj',
             broker='amqp://',
             backend='amqp://',
             include=['proj.tasks'])

# Optional configuration, see the application user guide.
app.conf.update(
    result_expires=3600,
)

if __name__ == '__main__':
    app.start()
```



proj/tasks.py

```python
from .celery import app


@app.task
def add(x, y):
    return x + y


@app.task
def mul(x, y):
    return x * y


@app.task
def xsum(numbers):
    return sum(numbers)
```

Celery

Using Celery in an Application

To start the worker:

```
$ celery -A proj worker -l info
```

When the worker starts we see a banner and messages showing our current configuration such as broker, app, concurrency of worker, events and queues

We can use ctrl+c to stop the worker server

Celery

Running in the background

Daemonization scripts uses the **celery multi** command to start one or more workers in the background:

```
$ celery multi start w1 -A proj -l info
```

To restart the server:

```
$ celery  multi restart w1 -A proj -l info
```

To stop the server:

```
$ celery multi stop w1 -A proj -l info
```

To stop when all executing tasks are completed:

```
$ celery multi stopwait w1 -A proj -l info
```

Celery

Running in the background

Celery multi creates pid and log files in the current directory. To protect against multiple workers launching on top of each other we can put these in a dedicated directory:

```
$ mkdir -p /var/run/celery

$ mkdir -p /var/log/celery

$ celery multi start w1 -A proj -l info --pidfile=/var/run/celery/%n.pid \
                                        --logfile=/var/log/celery/%n%I.log
```

Celery

Calling the tasks

We can call a task using delay() method

```
>>>add.delay(2,2)
```

We can also use apply_async() to provide additional arguments such as the queue and countdown

```
>>>add.apply_async((2,2), queue=‘lopri’, countdown=10)
```

These return an AsyncResult instance that can used to keep track of task execution state (requires result backend). We can also obtain the result of the task using get()

```
>>>result=add.delay(2,2)

>>>res.get( )  / res.failed( ) / res.successful( ) / res.state
```

Celery

Task States

A task can only be in a single state, but it can progress through several states.
The stages of a typical task can be:

PENDING -> STARTED -> SUCCESS

Started state is recorded if (track_started=True)

Pending state acts as the default state for any task id that’s unknown

Retrying the app adds several complex states:

PENDING -> STARTED -> RETRY -> STARTED -> RETRY -> STARTED -> SUCCESS

Celery

Signatures

- A signature wraps the arguments and execution options of a single task invocation in such a way that it can be passed to functions or even serialized and sent across the wire.

- We can create a signature using signature() method and arguments:

```
>>> add.signature((2, 2), countdown=10)
```

- Or a star argument shortcut

```
>>> add.s(2, 2)
```

Signature instances also support the calling API, meaning they have delay and apply_async methods.

Celery

Signatures

- A signature may already have arguments specified which matches the arguments of the task, a complete signature.
- But, we can also make incomplete signatures to create partials:

```
>>> s2 = add.s(2)
```

s2 is now a partial signature that needs another argument to be complete, and this can be resolved when calling the signature:

```
>>> res = s2.delay(8)
```

- Keyword arguments can also be added later; these are then merged with any existing keyword arguments, but with new arguments taking precedence:

```
>>> s3 = add.s(2, 2, debug=True)

>>> s3.delay(debug=False) #due to precedence debug is now false
```

Celery

Primitives

- Primitives are signature objects that can be combined in any number of ways to compose complex work-flows.
- Some of primitives are:
- Group, Chain, Chord, Map, Starmap, Chunks
- Since these primitives are all of the signature type they can be combined almost however we want.

Celery

Primitives: Group

- A group calls a list of tasks in parallel, and it returns a special result instance that lets you inspect the results as a group, and retrieve the return values in order.

```
>>> from celery import group

>>> from proj.tasks import add

>>> group(add.s(i, i) for i in range(10 ) ) ( ).get( )

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- We can also group our tasks using comma sepertaion

```
>>> group(add.s(2,2) , mul.s(4,4) ) ( ).get( )
```

- Partial group

```
>>> g = group( add.s(i) for i in range(10) )

>>> g(10).get()
```

Celery

Primitives: Chain

- Tasks can be linked together so that after one task returns the other is called:

```
>>> from celery import chain

>>> from proj.tasks import add,mul

>>>chain( add.s(4,4) | mul.s(8) ) ( ).get( )
```

(4+4 is chained with * 8=64)

- Partial Chain

```
>>> c = chain(add.s(4) | mul.s(8) )

>>> c(4).get()
```

We can use shorthand : `>>> ( add.s(4, 4) | mul.s(8) ) ( ).get( )`

Celery

Primitives: Chord

- A chord is a group with a callback:

```
>>> from celery import chord
>>> from proj.tasks import add, xsum
>>> chord( ( add.s(i, i) for i in range(10) ), xsum.s( ) )( ).get( )
```

First we get [0, 2, 4, 6, 8, 10, 12, 14, 16, 18] and then add to Get 90

- A group chained to another task will be automatically converted to a chord:

```
>>> ( group( add.s(i, i) for i in range(10)) | xsum.s( ) ) ( ).get( )
```

We can use shorthand : `>>> ( add.s(4, 4) | mul.s(8) ) ( ).get( )`

Celery
Routing

- Celery supports all of the routing facilities provided by AMQP, but it also supports simple routing where messages are sent to named queues.
- The task_routes setting enables us to route tasks by name and keep everything centralized in one location.
- Queues are defined with the following settings:

```
{'exchange': 'add',

 'exchange_type': ‘direct',

 'routing_key': 'add'}
```

Celery

Routing : Automatic Routing

- Automatic routing hides routing complexity and makes it easier to define routes.
- We can use task_routes to define routes for the specific tasks:

```
 >>>  task_routes = {
        'proj.tasks.add': {'queue': 'hipri'},
   }
```

Now the add task will be routed to hipri queue, while all other tasks will be routed to default queue.

- We can also use glob pattern matching to match a selection of tasks:

```
>>>  app.conf.task_routes={‘proj.tasks.* ’:{‘queue’ : ’hipri’}
```

Celery

Routing : Automatic Routing

- After configuring the router, we can start server to process only a certain queue:

```
$ celery -A proj worker -Q hipri
```

- We may specify as many queues as required:

```
$ celery -A proj worker -Q feeds,celery
```

- The default route name can be changed as:

```
app.conf.task_default_queue = ‘default'
```

Celery

Routing : Manual Routing

- We can use to following configuration for manual routing:

```
from kombu import Queue


app.conf.task_default_queue = 'default'

app.conf.task_queues = (

    Queue(‘default', routing_key='task.#'),

    Queue('feed_tasks', routing_key='feed.#'),

)

app.conf.task_default_exchange = 'tasks'

app.conf.task_default_exchange_type = 'topic'

app.conf.task_default_routing_key = 'task.default'
```

Celery

Routing : Manual Routing

- To route a task to feed_tasks we can add an entry in task_routes:

```
>>> task_routes = {
        'feeds.tasks.import_feed': {
            'queue': 'feed_tasks',
            'routing_key': 'feed.import',
        },
}
```

Celery

Routing: Message Priorities

- Queues can be configured to support priorities by setting the x-max-priority argument:

```
>>> from kombu import Exchange, Queue
>>> app.conf.task_queues = [
    Queue('tasks', Exchange('tasks'), routing_key='tasks',
          queue_arguments={'x-max-priority': 10}),
]
```

- Default value can be set as:

```
>>> app.conf.task_queue_max_priority = 10
```

Celery

Retry

- app.Task.retry() can be used to re-execute the task, for example in the event of recoverable errors.
- When you call retry it'll send a new message, using the same task-id, and it'll take care to make sure the message is delivered to the same queue as the originating task.
- When a task is retried this is also recorded as a task state, so that you can track the progress of the task using the result instance
- We can specify various options such as max_retries, auto_retry for known exception etc.

Celery

Retry

- An example:

```
@app.task(bind=True) #Here bind allows to bind self to the current task instance
def add_by_input(self):
    try:
        res=5+'5'
        return res
    except TypeError as exc:
        raise self.retry(exc=exc)
```

Celery
Periodic Tasks

- Celery uses “celery beat” to schedule periodic tasks.
- Celery beat runs tasks at regular intervals, which are then executed by celery workers.
- We can use @periodic_task decorator along with some options or configure beat_schedule settings to run a task periodically.
- To run the celery beat service we use:

```
$ celery -A proj beat
```

Or

```
$ celery -A proj worker -B  #convenient if not running more than one worker
```

Celery

Periodic Tasks: Configuration

- We can schedule the configuration by changing beat_schedule settings.
- Example: Run the tasks.add task every 30 seconds.

```
app.conf.beat_schedule = {
    'add-every-30-seconds': {
        'task': 'tasks.add',
        'schedule': 30.0,
        'args': (16, 16)
    },
}
```

Celery

Periodic Tasks: Scheduling

- Contrab Schedules: If we want more control over when the task is executed, for example, a particular time of day or day of the week, we can use the crontab schedule type:

```
'schedule': contrab(hour=7, minute=30, day_of_week=1)
```

- If you have a task that should be executed according to sunrise, sunset, dawn or dusk, you can use the solar schedule type:

```
'schedule': solar('sunset', -37.81753, 144.96715),
```

Celery

Help with the commands

Help:

To get a complete list of worker commands we can use

```
$  celery worker —help
```

Or for other commands

```
$ celery help
```

References

- https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html
- https://www.youtube.com/watch?v=oUJbuFMyBDk
- https://en.wikipedia.org/wiki/RabbitMQ
- https://www.rabbitmq.com/tutorials/tutorial-one-python.html
- https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html#message-flow-in-rabbitmq
- https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html
- https://docs.celeryproject.org/en/stable/index.html
- https://docs.celeryproject.org/en/stable/userguide/routing.html#id7
- https://docs.celeryproject.org/en/stable/getting-started/first-steps-with-celery.html#id7