HumusAmqpModule Documentation

Release 0.1.0

Sascha-Oliver Prolic

June 18, 2016

Contents

1 Contents **3**

1.1 Guides overview . . . 3
1.2 Getting Started with Humus AMQP Module, RabbitMQ and Zend Framework 2 . . . 5
1.3 Connecting to RabbitMQ from Zend Framework 2 with Humus AMQP Module . . . 9
1.4 Exchanges and Producers . . . 12
1.5 Queues . . . 27
1.6 Bindings . . . 33
1.7 Consumers . . . 36
1.8 CLI Usage . . . 40
1.9 Durability . . . 40
1.10 RabbitMQ Extensions . . . 42
1.11 Error Handling . . . 47
1.12 Troubleshooting . . . 49
1.13 Deployment Strategies . . . 49
1.14 License Information . . . 51

2 Indices and tables **53**

Documentation for Humus AMQP Module

The Humus AMQP Module incorporates messaging in your zf2 application via RabbitMQ using the PHP AMQP Extension.

This module implements several messaging patterns comparable to the Thumper library.

A lot of ideas and even some implementation details came from the RabbitMQ Java Client and the RabbitMqBundle, special thanks to Alvaro Videla and the contributors of this project.

The documentation is based on the Bunny documentation, texts and images are partly copied from there, special thanks to Michael Klishin for the permission to build on this work.

CHAPTER 1

Contents

1.1 Guides overview

We recommend that you read these guides, if possible, in this order:

1.1.1 Getting started

An overview of HumusAmqpModule with a quick tutorial that helps you to get started with it. It should take about 20 minutes to read and study the provided code examples.

1.1.2 AMQP 0.9.1 Model Concepts

This guide covers:

- AMQP 0.9.1 model overview
- What are channels
- What are vhosts
- What are queues
- What are exchanges
- What are bindings
- What are AMQP 0.9.1 classes and methods

1.1.3 Connecting to RabbitMQ from Zend Framework 2 with Humus AMQP Module

This guide covers:

- How to connect to RabbitMQ with HumusAmqpModule
- How to use connection URI to connect to RabbitMQ (also: in PaaS environments such as Heroku and CloudFoundry)
- How to open a channel
- How to close a channel
- How to disconnect

1.1.4 Queues and Consumers

This guide covers:

- How to declare AMQP queues with HumusAmqpModule
- Queue properties
- How to declare server-named queues
- How to declare temporary exclusive queues
- How to consume messages ("push API")
- How to fetch messages ("pull API")
- Message and delivery properties
- Message acknowledgements
- How to purge queues
- How to delete queues
- Other topics related to queues

1.1.5 Exchanges and Publishing

This guide covers:

- Exchange types
- How to declare AMQP exchanges with HumusAmqpModule
- How to publish messages
- Exchange properties
- Fanout exchanges
- Direct exchanges
- Topic exchanges
- Default exchange
- Message and delivery properties
- Message routing
- Bindings
- How to delete exchanges
- Other topics related to exchanges and publishing

1.1.6 Bindings

This guide covers:

- How to bind exchanges to queues
- How to unbind exchanges from queues
- Other topics related to bindings

1.1.7 Durability and Related Matters

This guide covers:

- Topics related to durability of exchanges and queues
- Durability of messages

1.1.8 RabbitMQ Extensions to AMQP 0.9.1

This guide covers RabbitMQ extensions and how they are used in Humus AMQP Module:

- How to use exchange-to-exchange bindings
- How to the alternate exchange extension
- How to set per-queue message TTL
- How to set per-message TTL
- What are consumer cancellation notifications and how to use them
- Message *dead lettering* and the dead letter exchange

1.1.9 Error Handling and Recovery

This guide covers:

- AMQP 0.9.1 protocol exceptions
- How to deal with network failures
- Other things that may go wrong

1.1.10 Troubleshooting

This guide covers:

- What to check when your apps that use Humus AMQP Module and RabbitMQ misbehave

Tell Us What You Think!

Please take a moment to tell us what you think about this guide: Send an e-mail or raise an issue on Github.

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.2 Getting Started with Humus AMQP Module, RabbitMQ and Zend Framework 2

1.2.1 About this guide

This guide is a quick tutorial that helps you to get started with RabbitMQ and HumusAmqpModule. It should take about 20 minutes to read and study the provided code examples. This guide covers:

- Installing RabbitMQ, a mature popular messaging broker server.
- Installing HumusAmqpModule via Composer.
- Installing HumusAmqpDemoModule.
- Producing and consuming messages from cli.
- Running a rpc server and and a client.

1.2.2 Installing RabbitMQ

The RabbitMQ site has a good installation guide that addresses many operating systems. On Mac OS X, the fastest way to install RabbitMQ is with Homebrew:

```
$ brew install rabbitmq
```

then run it:

```
$ rabbitmq-server
```

On Debian and Ubuntu, you can either download the RabbitMQ .deb package and install it with dpkg or make use of the apt repository that the RabbitMQ team provides.

For RPM-based distributions like RedHat or CentOS, the RabbitMQ team provides an RPM package.

1.2.3 Installing HumusAmqpModule & HumusAmqpDemoModule

1. Make sure you have the php-amqp extension installed
2. The minimum required version is 1.4.0

c) Make sure that you have a running Zend Framework 2 Skeleton Application

4. You can use composer to install HumusAmqpModule

```
$ cd path/to/zf2app
$ php composer.phar require prolic/humus-amqp-module dev-master
$ php composer.phar require prolic/humus-amqp-demo-module dev-master
```

5. Adding HumusAmqpModule & HumusAmqpDemoModule to application configuration

Edit your config/application.config.php

```
<?php
    return array(
        // This should be an array of module namespaces used in the application.
        'modules' => array(
            'Application',
    ),
```

to

```
<?php
    return array(
    // This should be an array of module namespaces used in the application.
    'modules' => array(
        'Application',
        'HumusAmqpModule',
        'HumusAmqpDemoModule'
    ),
```

1.2.4 Running demos from CLI

Demo-Consumer and Producer

Open up 2 terminals.

Then run the demo consumer

```
$ php public/index.php humus amqp consumer demo-consumer
```

Next, open another shell and run the demo producer

```
$ php public/index.php humus amqp stdin-producer demo-producer "demo-message"
```

You should see the output in the demo consumer's shell. It should look something like this:

```
hallo
2014-08-27T18:43:30+02:00 DEBUG (7): Acknowledged 1 messages at 0 msg/s
```

If you run the command multiple times, you can see that it the consumer will also ack bundles of messages. You noticed perhabs, that you run it with the stdin-producer command, but what does this mean? Try this:

```
$ cat README.md | xargs -0 php public/index.php humus amqp stdin-producer demo-producer
$ echo "my test message" | xargs -0 php public/index.php humus amqp stdin-producer demo-producer
```

For now, let's check what a demo consumer looks like and how to configure it.

The EchoCallback is the implementation part of the consumer. As you can see, you simply provide a callable, you get the parameters (message and queue) and you're ready to start. You don't need to extend or even write yourself the consumer implementation.

The required connection configuration can be found at: module.config.php#L85-L95.

The required exchange configuration is also there: module.config.php#L27-L37.

The required queue configuration: module.config.php#L56-L64.

The required consumer configuration: module.config.php#L112-L119.

And finally, the required producer configuration: module.config.php#L98-L105.

More information about the configuration of the Humus AMQP module, check the other sections of the manual.

That's it, send a SIGUSR1-signal (kill -10) to stop the consumer. You probably noticed, that there is an error-exchange configured for the demo exchange.

That's a nice exercise: Go and change the consumer callback to "return false;", so the messages get a nack and will be routed to the error exchange. Attach a queue to that exchange and create the consumer configuration. You can also reuse the already existing EchoErrorCallback.

Topic consumer and producer example

First, run the consumer again:

```
$ php public/index.php humus amqp consumer topic-consumer-error
```

This consumer is only interested in routing keys matching #.err, so let's send some messages with different routing keys.

```
$ php public/index.php humus amqp stdin-producer topic-producer --route=level.err err
$ php public/index.php humus amqp stdin-producer topic-producer --route=level.warn warn
$ php public/index.php humus amqp stdin-producer topic-producer --route=level.info info
$ php public/index.php humus amqp stdin-producer topic-producer --route=level.debug debug
```

As you can see, only the first message in interessting for the consumer, all others are trashed. Go, send a lot of messages:

```
$ php public/index.php humus amqpdemo topic-producer 1000
```

This will send 1000 messages that will be consumed by the topic-consumer-error. You probably noticed, that by default, the consumer will never ack more than 3 messages at once, even if you send tons of messages. You can change that, go to the module.config.php file:

```
'topic-consumer-error' => array(
     'queues' => array(
         'info-queue',
     ),
     'callback' => 'HumusAmqpDemoModule\Demo\EchoCallback',
     'qos' => array(
         'prefetch_count' => 100
     ),
     'auto_setup_fabric' => true
 ),
```

If you set the prefetch count to 100, the consumer will ack up to 100 messages at once. For more information, see: Consumer Prefetch.

Running RPC-client & -server example

Open up 3 terminals.

Then run 2 rpc-servers

```
$ php public/index.php humus amqp rpc-server demo-rpc-server
$ php public/index.php humus amqp rpc-server demo-rpc-server2
```

Before we start the client, let's see, how a rpc-server get's configured what in the demo servers.

First, we need exchanges: module.config.php#L46-L53.

Queues, too: module.config.php#L65-L76.

And here's how the servers/ clients are configured: module.config.phpL128-L145.

You can check the callbacks here.

The PowerOfTwoCallback does a sleep(1) before returning, the RandomIntCallback does a sleep(2); With this, it's more easy to show real parallel processing.

Start the rpc-client

```
$ php public/index.php humus amqpdemo rpc-client 5
```

This will send 5 messages, you can see in the server output, that the messages are acknowledged and the response in the client afterwards. No let's send messages to both:

```
$ php public/index.php humus amqpdemo rpc-client 5 --parallel
```

What? Don't believe it? It's truly parallel!

```
$ time php public/index.php humus amqpdemo rpc-client 5 --parallel
```

Enjoy!

See *Running from CLI* get know more about Humus AMQP Module's CLI commands.

1.2.5 What to read next

Documentation is organized as a number of *guides*, covering all kinds of topics including use cases for various exchange types, fault-tolerant message processing with acknowledgements and error handling.

We recommend that you read the following guides next, if possible, in this order:

- AMQP 0.9.1 Model Explained. A simple 2 page long introduction to the AMQP Model concepts and features. Understanding the AMQP 0.9.1 Model will make a lot of other documentation, both for Bunny and RabbitMQ itself, easier to follow. With this guide, you don't have to waste hours of time reading the whole specification.
- *Connecting to RabbitMQ from Zend Framework 2 with Humus AMQP Module*. This guide explains how to connect to an RabbitMQ and how to integrate Bunny into standalone and Web applications.
- *Queues*. This guide focuses on features that consumer applications use heavily.
- *Exchanges and Producers*. This guide focuses on features that producer applications use heavily.
- *Error Handling*. This guide explains how to handle protocol errors, network failures and other things that may go wrong in real world projects.

1.2.6 Tell Us What You Think!

Please take a moment to tell us what you think about this guide: Send an e-mail or raise an issue on Github.

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.3 Connecting to RabbitMQ from Zend Framework 2 with Humus AMQP Module

1.3.1 Connection configuration

Map options that Humus AMQP Module will recognize are

- `:host` - amqp.host The host to connect too. Note: Max 1024 characters.
- `:port` - amqp.port Port on the host.
- `:vhost` - amqp.vhost The virtual host on the host. Note: Max 128 characters.
- `:login` - amqp.login The login name to use. Note: Max 128 characters.
- `:password` - amqp.password Password. Note: Max 128 characters.
- `:persistent` - Establish a persistent connection with the AMQP broker, if set to true.
- `:read_timeout` - Timeout in for income activity. Note: 0 or greater seconds. May be fractional.
- `:write_timeout` - Timeout in for outcome activity. Note: 0 or greater seconds. May be fractional.

1.3.2 Default parameters

Default connection parameters are

```
[
  'host'         => "127.0.0.1",
  'port'         => 5672,
  'vhost'        => "/",
  'login'        => "guest",
  'password'     => "guest",
  'persistent'   => false,
  'readTimeout'  => 1.0,
  'writeTimeout' => 1.0
]
```

Note: The persistent parameter is only used by the Humus AMQP Module's ConnectionAbstractServiceFactory.

Based on this parameter the module will decide, whether to call pconnect() or connect() on the connection.

1.3.3 Creating a connection

```
<?php

$conn = new AMQPConnection();
$conn->setLogin('demouser');
$conn->setPassword('password');
...
$conn->pconnect();
```

1.3.4 Opening a Channel

Some applications need multiple connections to RabbitMQ. However, it is undesirable to keep many TCP connections open at the same time because doing so consumes system resources and makes it more difficult to configure firewalls. AMQP 0-9-1 connections are multiplexed with channels that can be thought of as "lightweight connections that share a single TCP connection".

To open a channel:

```
<?php

$conn = new AMQPConnection();
$conn->connect();
$ch   = new AMQPChannel($conn);
```

Channels are typically long lived: you open one or more of them and use them for a period of time, as opposed to opening a new channel for each published message, for example.

1.3.5 Disconnecting

To close a connection, use the disconnect() method. This will automatically close all channels of that connection first:

```php
<?php

$conn = new AMQPConnection();
$conn->connect();
$ch   = new AMQPChannel($conn);
$conn->disconnect();
```

1.3.6 Module Configuration

You can simply configure as many connection as needed and simply give them a name. You can also set a default connection, using the `default_connection` configuration key.

```php
<?php

return array(
    'humus_amqp_module' => array(
        'default_connection' => 'default',
        'connections' => array(
            'default' => array(
                'host' => 'localhost',
                'port' => 5672,
                'login' => 'guest',
                'password' => 'guest',
                'vhost' => '/',
                'persistent' => true,
            )
        )
    )
);
```

1.3.7 Getting a connection

All connections are handled by the HumusAmqpModulePluginManagerConnection. To grab a connection simply call:

```php
<?php

$connectionManager = $serviceManager->get('HumusAmqpModule\PluginManager\Connection');
$defaultConnection = $connectionManager->get('default');
```

1.3.8 Troubleshooting

If you have read this guide and still have issues with connecting, check our *Troubleshooting guide* and feel free to raise an issue at Github.

1.3.9 What to Read Next

The documentation is organized as *a number of guides*, covering various topics.

We recommend that you read the following guides first, if possible, in this order:

- *Queues and Consumers*
- *Exchanges and Publishing*

- *Bindings*
- RabbitMQ Extensions to AMQP 0.9.1
- *Durability and Related Matters*
- *Error Handling and Recovery*
- *Troubleshooting*

1.3.10 Tell Us What You Think!

Please take a moment to tell us what you think about this guide: Send an e-mail or raise an issue on Github.

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.4 Exchanges and Producers

1.4.1 Exchanges in AMQP 0.9.1 — Overview

What are AMQP exchanges?

An *exchange* accepts messages from a producer application and routes them to message queues. They can be thought of as the "mailboxes" of the AMQP world. Unlike some other messaging middleware products and protocols, in AMQP, messages are *not* published directly to queues. Messages are published to exchanges that route them to queue(s) using pre-arranged criteria called *bindings*.

There are multiple exchange types in the AMQP 0.9.1 specification, each with its own routing semantics. Custom exchange types can be created to deal with sophisticated routing scenarios (e.g. routing based on geolocation data or edge cases) or just for convenience.

Concept of Bindings

A *binding* is an association between a queue and an exchange. A queue must be bound to at least one exchange in order to receive messages from publishers. Learn more about bindings in the *Bindings Guide*.

Exchange attributes

Exchanges have several attributes associated with them:

- Name
- Type (direct, fanout, topic, headers or some custom type)
- Durability
- Whether the exchange is auto-deleted when no longer used
- Other metadata (sometimes known as *X-arguments*)

1.4.2 Exchange types

There are four built-in exchange types in AMQP v0.9.1:

- Direct
- Fanout
- Topic
- Headers

As stated previously, each exchange type has its own routing semantics and new exchange types can be added by extending brokers with plugins. Custom exchange types begin with "x-", much like custom HTTP headers, e.g. x-consistent-hash exchange or x-random exchange.

1.4.3 Message attributes

Before we start looking at various exchange types and their routing semantics, we need to introduce message attributes. Every AMQP message has a number of *attributes*. Some attributes are important and used very often, others are rarely used. AMQP message attributes are metadata and are similar in purpose to HTTP request and response headers.

Every AMQP 0.9.1 message has an attribute called *routing key*. The routing key is an "address" that the exchange may use to decide how to route the message. This is similar to, but more generic than, a URL in HTTP. Most exchange types use the routing key to implement routing logic, but some ignore it and use other criteria (e.g. message content).

1.4.4 Fanout exchanges

How fanout exchanges route messages

A fanout exchange routes messages to all of the queues that are bound to it and the routing key is ignored. If N queues are bound to a fanout exchange, when a new message is published to that exchange a *copy of the message* is delivered to all N queues. Fanout exchanges are ideal for the broadcast routing of messages.

Graphically this can be represented as:

Fig. 1.1: fanout exchange routing

Declaring a fanout exchange

You can simply define an exchange in the module configuration.

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'myexchange' => array(
                'name' => 'myexchange',
                'type' => 'fanout'
            )
        )
    )
);
```

Fanout use cases

Because a fanout exchange delivers a copy of a message to every queue bound to it, its use cases are quite similar:

- Massively multiplayer online (MMO) games can use it for leaderboard updates or other global events
- Sport news sites can use fanout exchanges for distributing score updates to mobile clients in near real-time
- Distributed systems can broadcast various state and configuration updates
- Group chats can distribute messages between participants using a fanout exchange (although AMQP does not have a built-in concept of presence, so XMPP may be a better choice)

Pre-declared fanout exchanges

AMQP 0.9.1 brokers must implement a fanout exchange type and pre-declare one instance with the name of `"amq.fanout"`.

Applications can rely on that exchange always being available to them. Each vhost has a separate instance of that exchange, it is *not shared across vhosts* for obvious reasons.

1.4.5 Direct exchanges

How direct exchanges route messages

A direct exchange delivers messages to queues based on a *message routing key*, an attribute that every AMQP v0.9.1 message contains.

Here is how it works:

- A queue binds to the exchange with a routing key K
- When a new message with routing key R arrives at the direct exchange, the exchange routes it to the queue if K = R

A direct exchange is ideal for the unicast routing of messages (although they can be used for multicast routing as well).

Here is a graphical representation:

Fig. 1.2: direct exchange routing

Declaring a direct exchange

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'myexchange' => array(
                'name' => 'myexchange',
                'type' => 'direct'
            )
        )
    )
);
```

Direct routing example

Since direct exchanges use the *message routing key* for routing, message producers need to specify it:

The routing key will then be compared for equality with routing keys on bindings, and consumers that subscribed with the same routing key each get a copy of the message.

Direct Exchanges and Load Balancing of Messages

Direct exchanges are often used to distribute tasks between multiple workers (instances of the same application) in a round robin manner. When doing so, it is important to understand that, in AMQP 0.9.1, *messages are load balanced between consumers and not between queues*.

The *Queues and Consumers* guide provides more information on this subject.

Pre-declared direct exchanges

AMQP 0.9.1 brokers must implement a direct exchange type and pre-declare two instances:

- `amq.direct`
- *“”* exchange known as *default exchange* (unnamed, referred to as an empty string)

Applications can rely on those exchanges always being available to them. Each vhost has separate instances of those exchanges, they are *not shared across vhosts* for obvious reasons.

Default exchange

The default exchange is a direct exchange with no name pre-declared by the broker. It has one special property that makes it very useful for simple applications, namely that *every queue is automatically bound to it with a routing key which is the same as the queue name*.

For example, when you declare a queue with the name of “search.indexing.online”, RabbitMQ will bind it to the default exchange using “search.indexing.online” as the routing key. Therefore a message published to the default exchange with routing key = “search.indexing.online” will be routed to the queue “search.indexing.online”. In other words, the default exchange makes it *seem like it is possible to deliver messages directly to queues*, even though that is not technically what is happening.

Direct Exchange Use Cases

Direct exchanges can be used in a wide variety of cases:

- Direct (near real-time) messages to individual players in an MMO game
- Delivering notifications to specific geographic locations (for example, points of sale)
- Distributing tasks between multiple instances of the same application all having the same function, for example, image processors
- Passing data between workflow steps, each having an identifier (also consider using headers exchange)
- Delivering notifications to individual software services in the network

1.4.6 Topic Exchanges

How Topic Exchanges Route Messages

Topic exchanges route messages to one or many queues based on matching between a message routing key and the pattern that was used to bind a queue to an exchange. The topic exchange type is often used to implement various publish/subscribe pattern variations.

Topic exchanges are commonly used for the multicast routing of messages.

Topic exchanges can be used for broadcast routing, but fanout exchanges are usually more efficient for this use case.

Declaring a topic exchange

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'myexchange' => array(
                'name' => 'myexchange',
                'type' => 'topic',
                'connection' => 'my_other_connection'
            )
        )
    )
);
```

As you can see, you can also specify to which connection the exchange belongs to. If nothing is present, the default connection will be used. You can set the default like this:

```
<?php

return array(
    'humus_amqp_module' => array(
        'default_connection' => 'my_other_connection'
    )
);
```

Topic Exchange Routing Example

Two classic examples of topic-based routing are stock price updates and location-specific data (for instance, weather broadcasts). Consumers indicate which topics they are interested in (think of it like subscribing to a feed for an individual tag of your favourite blog as opposed to the full feed).

A routing pattern consists of several words separated by dots, in a similar way to URI path segments being joined by slash. A few of examples:

- asia.southeast.thailand.bangkok
- sports.basketball
- usa.nasdaq.aapl
- tasks.search.indexing.accounts

The following routing keys match the "americas.south.#" pattern:

- americas.south
- americas.south.*brazil*
- americas.south.*brazil.saopaolo*
- americas.south.*chile.santiago*

In other words, the "#" part of the pattern matches 0 or more words.

For the pattern "americas.south.*", some matching routing keys are:

- americas.south.*brazil*
- americas.south.*chile*
- americas.south.*peru*

but not

- americas.south
- americas.south.chile.santiago

As you can see, the "*" part of the pattern matches 1 word only.

Topic Exchange Use Cases

Topic exchanges have a very broad set of use cases. Whenever a problem involves multiple consumers/applications that selectively choose which type of messages they want to receive, the use of topic exchanges should be considered. To name a few examples:

- Distributing data relevant to specific geographic location, for example, points of sale
- Background task processing done by multiple workers, each capable of handling specific set of tasks
- Stocks price updates (and updates on other kinds of financial data)
- News updates that involve categorization or tagging (for example, only for a particular sport or team)
- Orchestration of services of different kinds in the cloud
- Distributed architecture/OS-specific software builds or packaging where each builder can handle only one architecture or OS

1.4.7 Publishing messages

To publish a message to an exchange, first we need a configured producer.

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'myexchange' => array(
                'name' => 'myexchange',
                'type' => 'topic',
            )
        ),
        'producers' => array(
            'my-producer' => array(
```

```
                'exchange' => 'myexchange',
                'auto_setup_fabric' => true
            )
        )
    )
);
```

You define a producer name (my-producer) and tell to which exchange it should publish. Additionally you can set "auto_setup_fabric" to true. This will automatically create the exchange if none is present.

To publish a message, get the producer plugin manager, get the producer and publish:

```
<?php

$pm = $serivceManager->get('HumusAmqpModule\PluginManager\Producer');
$producer = $pm->get('my-producer');

$producer->publish('foo', 'my.routing.key');

$messages = array('foo', 'bar', 'baz');

$producer->publishBatch($messages, 'my.routing.key');
```

The method accepts message body, a routing key and some message attributes. Routing key can be blank (`""`) but never `null`. The body needs to be a string. The message payload is completely opaque to the library and is not modified by Bunny or RabbitMQ in any way.

Data serialization

You are encouraged to take care of data serialization before publishing (i.e. by using JSON, Thrift, Protocol Buffers or some other serialization library). Note that because AMQP is a binary protocol, text formats like JSON largely lose their advantage of being easy to inspect as data travels across the network, so if bandwidth efficiency is important, consider using MessagePack or Protocol Buffers.

A few popular options for data serialization are:

- JSON
- BSON
- Message Pack
- XML
- Protocol Buffers

Message attributes

RabbitMQ messages have various metadata attributes that can be set when a message is published. Some of the attributes are well-known and mentioned in the AMQP 0.9.1 specification, others are specific to a particular application. Well-known attributes are listed here as options that Humus AMQP Module takes:

- `:persistent`
- `:delivery_mode`
- `:mandatory`
- `:timestamp`

- :expiration
- :type
- :reply_to
- :content_type
- :content_encoding
- :correlation_id
- :priority
- :cluster_id
- :user_id
- :app_id
- :message_id

All other attributes can be added to a *headers table*.

An example:

```
<?php

$now = microtime(1);

$attribs = new MessageAttributes()
$attribs->setAppId('amqp.example');
$attribs->setAppId(8);
$attribs->setType('kinda.checkin';
$attribs->setHeaders(array(
    'latitude' => 59.35,
    'longituide' => 18.0666667
));
$attribs->setTimestamp($now)
$attribs->setCorrelationId('r-1');
$attribs->setContentType('application/json');

$producer->publish('{"foo": "bar"}', '', $attribs);
```

:routing_key

Used for routing messages depending on the exchange type and configuration.

:persistent

When set to true, RabbitMQ will persist message to disk.

:mandatory

This flag tells the server how to react if the message cannot be routed to a queue. If this flag is set to true, the server will return an unroutable message to the producer with a `basic.return` AMQP method. If this flag is set to false, the server silently drops the message.

:content_type

MIME content type of message payload. Has the same purpose/semantics as HTTP Content-Type header.

:content_encoding

MIME content encoding of message payload. Has the same purpose/semantics as HTTP Content-Encoding header.

:priority

Message priority, from 0 to 9.

:message_id

Message identifier as a string. If applications need to identify messages, it is recommended that they use this attribute instead of putting it into the message payload.

:reply_to

Commonly used to name a reply queue (or any other identifier that helps a consumer application to direct its response). Applications are encouraged to use this attribute instead of putting this information into the message payload.

:correlation_id

ID of the message that this message is a reply to. Applications are encouraged to use this attribute instead of putting this information into the message payload.

:type

Message type as a string. Recommended to be used by applications instead of including this information into the message payload.

:user_id

Sender's identifier. Note that RabbitMQ will check that the value of this attribute is the same as username AMQP connection was authenticated with, it SHOULD NOT be used to transfer, for example, other application user ids or be used as a basis for some kind of Single Sign-On solution.

:app_id

Application identifier string, for example, "eventoverse" or "webcrawler"

:timestamp

Timestamp of the moment when message was sent, in seconds since the Epoch

:expiration

Message expiration specification as a string

:arguments

A map of any additional attributes that the application needs. Nested hashes are supported. Keys must be strings.

It is recommended that application authors use well-known message attributes when applicable instead of relying on custom headers or placing information in the message body. For example, if your application messages have priority, publishing timestamp, type and content type, you should use the respective AMQP message attributes instead of reinventing the wheel.

Validated User ID

In some scenarios it is useful for consumers to be able to know the identity of the user who published a message. RabbitMQ implements a feature known as validated User ID. If this property is set by a publisher, its value must be the same as the name of the user used to open the connection. If the user-id property is not set, the publisher's identity is not validated and remains private.

Note: Validated user id not yet implemented in Humus AMQP Module.

Publishing Callbacks and Reliable Delivery in Distributed Environments

A commonly asked question about RabbitMQ clients is "how to execute a piece of code after a message is received".

Message publishing with Bunny happens in several steps:

- AMQPExchange::publish takes a payload and various metadata attributes
- Resulting payload is staged for writing
- On the next event loop tick, data is transferred to the OS kernel using one of the underlying NIO APIs
- OS kernel buffers data before sending it
- Network driver may also employ buffering

As you can see, "when data is sent" is a complicated issue and while methods to flush buffers exist, flushing buffers does not guarantee that the data was received by the broker because it might have crashed while data was travelling down the wire.

The only way to reliably know whether data was received by the broker or a peer application is to use message acknowledgements. This is how TCP works and this approach is proven to work at the enormous scale of the modern Internet. AMQP 0.9.1 fully embraces this fact and Bunny follows.

In cases when you cannot afford to lose a single message, AMQP 0.9.1 applications can use one (or a combination of) the following protocol features:

- Publisher confirms (a RabbitMQ-specific extension to AMQP 0.9.1)
- Publishing messages as mandatory
- Transactions (these introduce noticeable overhead and have a relatively narrow set of use cases)

A more detailed overview of the pros and cons of each option can be found in a blog post that introduces Publisher Confirms extension by the RabbitMQ team. The next sections of this guide will describe how the features above can be used with Bunny.

Publishing messages as mandatory

When publishing messages, it is possible to use the `:mandatory` option to publish a message as "mandatory". When a mandatory message cannot be *routed* to any queue (for example, there are no bindings or none of the bindings match), the message is returned to the producer.

Note: The PHP AMQP Extension currently has not full support of the mandatory flag.

Returned messages

When a message is returned, the application that produced it can handle that message in different ways:

- Store it for later redelivery in a persistent store
- Publish it to a different destination
- Log the event and discard the message

A returned message handler has access to AMQP method (`basic.return`) information, message metadata and payload (as a byte array). The metadata and message body are returned without modifications so that the application can store the message for later redelivery.

Publishing Persistent Messages

Messages potentially spend some time in the queues to which they were routed before they are consumed. During this period of time, the broker may crash or experience a restart. To survive it, messages must be persisted to disk. This has a negative effect on performance, especially with network attached storage like NAS devices and Amazon EBS. AMQP 0.9.1 lets applications trade off performance for durability, or vice versa, on a message-by-message basis.

To publish a persistent message, use the `:persistent` option:

```
<?php

$attribs = new MessageAttributes();
$attribs->setPersistent(true);
$producer->publish($data, '', $attribs);
```

Note that in order to survive a broker crash, the messages MUST be persistent and the queue that they were routed to MUST be durable.

Durability and Message Persistence provides more information on the subject.

Publishing In Multi-threaded Environments

When using Humus AMQP Module in multi-threaded environments, the rule of thumb is: avoid sharing channels across threads.

In other words, publishers in your application that publish from separate threads should use their own channels. The same is a good idea for consumers.

1.4.8 Headers exchanges

Now that message attributes and publishing have been introduced, it is time to take a look at one more core exchange type in AMQP 0.9.1. It is called the *headers exchange type* and is quite powerful.

How headers exchanges route messages

An Example Problem Definition

The best way to explain headers-based routing is with an example. Imagine a distributed continuous integration system that distributes builds across multiple machines with different hardware architectures (x86, IA-64, AMD64, ARM family and so on) and operating systems. It strives to provide a way for a community to contribute machines to run tests on and a nice build matrix like the one WebKit uses. One key problem such systems face is build distribution. It would be nice if a messaging broker could figure out which machine has which OS, architecture or combination of the two and route build request messages accordingly.

A headers exchange is designed to help in situations like this by routing on multiple attributes that are more easily expressed as message metadata attributes (headers) rather than a routing key string.

Routing on Multiple Message Attributes

Headers exchanges route messages based on message header matching. Headers exchanges ignore the routing key attribute. Instead, the attributes used for routing are taken from the "headers" attribute. When a queue is bound to a headers exchange, the `:arguments` attribute is used to define matching rules:

```php
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'header-exchange' => array(
                'name' => 'header-exchange',
                'type' => 'headers'
            )
        ),
        'queues' => array(
            'myqueue-1' => array(
                'name' => 'myqueue',
                'exchange' => 'header-exchange',
                'arguments' => array(
                    'os' => 'linux',
                    'x-match' => 'all'
                )
            ),
            'myqueue-2' => array(
                'name' => 'myqueue',
                'exchange' => 'header-exchange',
                'arguments' => array(
                    'os' => 'osx',
                    'x-match' => 'any'
                )
            )
        ),
        'producers' => array(
            'my-producer' => array(
                'exchange' => 'exchanges',
                'auto_setup_fabric' => true
            )
        )
    )
);
```

When matching on one header, a message is considered matching if the value of the header equals the value specified upon binding. An example that demonstrates headers routing:

```php
<?php

$attribs = new MessageAttributes();

$attribs->setHeaders(array(
    'os' => 'linux',
    'cores' => 8
));
$producer->publish('8 cores/Linux', '', $attribs);

$attribs->setHeaders(array(
    'os' => 'osx',
    'cores' => 8
));

$producer->publish('4 cores/OS X', '', $attribs);
```

When executed, it outputs

```
myqueue-2 received 8 cores/Linux
```

The myqueue-1 has not matched, because of x-match: all

Matching All vs Matching One

It is possible to bind a queue to a headers exchange using more than one header for matching. In this case, the broker needs one more piece of information from the application developer, namely, should it consider messages with any of the headers matching, or all of them? This is what the "x-match" binding argument is for.

When the `"x-match"` argument is set to `"any"`, just one matching header value is sufficient. So in the example above, any message with a "cores" header value equal to 8 will be considered matching.

Headers Exchange Routing

When there is just one queue bound to a headers exchange, messages are routed to it if any or all of the message headers match those specified upon binding. Whether it is "any header" or "all of them" depends on the `"x-match"` header value. In the case of multiple queues, a headers exchange will deliver a copy of a message to each queue, just like direct exchanges do. Distribution rules between consumers on a particular queue are the same as for a direct exchange.

Headers Exchange Use Cases

Headers exchanges can be looked upon as "direct exchanges on steroids" and because they route based on header values, they can be used as direct exchanges where the routing key does not have to be a string; it could be an integer or a hash (dictionary) for example.

Some specific use cases:

- Transfer of work between stages in a multi-step workflow (routing slip pattern)
- Distributed build/continuous integration systems can distribute builds based on multiple parameters (OS, CPU architecture, availability of a particular package).

Pre-declared Headers Exchanges

RabbitMQ implements a headers exchange type and pre-declares one instance with the name of `"amq.match"`. RabbitMQ also pre-declares one instance with the name of `"amq.headers"`. Applications can rely on those exchanges always being available to them. Each vhost has a separate instance of those exchanges and they are *not shared across vhosts* for obvious reasons.

1.4.9 Custom Exchange Types

consistent-hash

The consistent hashing AMQP exchange type is a custom exchange type developed as a RabbitMQ plugin. It uses consistent hashing to route messages to queues. This helps distribute messages between queues more or less evenly.

A quote from the project README:

In various scenarios, you may wish to ensure that messages sent to an exchange are consistently and equally distributed across a number of different queues based on the routing key of the message. You could arrange for this to occur yourself by using a direct or topic exchange, binding queues to that exchange and then publishing messages to that exchange that match the various binding keys.

However, arranging things this way can be problematic:

It is difficult to ensure that all queues bound to the exchange will receive a (roughly) equal number of messages without baking in to the publishers quite a lot of knowledge about the number of queues and their bindings.

If the number of queues changes, it is not easy to ensure that the new topology still distributes messages between the different queues evenly.

Consistent Hashing is a hashing technique whereby each bucket appears at multiple points throughout the hash space, and the bucket selected is the nearest higher (or lower, it doesn't matter, provided it's consistent) bucket to the computed hash (and the hash space wraps around). The effect of this is that when a new bucket is added or an existing bucket removed, only a very few hashes change which bucket they are routed to.

In the case of Consistent Hashing as an exchange type, the hash is calculated from the hash of the routing key of each message received. Thus messages that have the same routing key will have the same hash computed, and thus will be routed to the same queue, assuming no bindings have changed.

x-random

The x-random AMQP exchange type is a custom exchange type developed as a RabbitMQ plugin by Jon Brisbin. A quote from the project README:

> It is basically a direct exchange, with the exception that, instead of each consumer bound to that exchange with the same routing key getting a copy of the message, the exchange type randomly selects a queue to route to.

This plugin is licensed under Mozilla Public License 1.1, same as RabbitMQ.

1.4.10 Using the Publisher Confirms Extension

Please refer to *RabbitMQ Extensions guide*

Message Acknowledgements and Their Relationship to Transactions and Publisher Confirms

Consumer applications (applications that receive and process messages) may occasionally fail to process individual messages, or might just crash. Additionally, network issues might be experienced. This raises a question - "when should the RabbitMQ remove messages from queues?" This topic is covered in depth in the *Queues guide*, including prefetching and examples.

In this guide, we will only mention how message acknowledgements are related to AMQP transactions and the Publisher Confirms extension. Let us consider a publisher application (P) that communications with a consumer (C) using AMQP 0.9.1. Their communication can be graphically represented like this:

We have two network segments, S1 and S2. Each of them may fail. A publisher (P) is concerned with making sure that messages cross S1, while the broker (B) and consumer (C) are concerned with ensuring that messages cross S2 and are only removed from the queue when they are processed successfully.

Message acknowledgements cover reliable delivery over S2 as well as successful processing. For S1, P has to use transactions (a heavyweight solution) or the more lightweight Publisher Confirms, a RabbitMQ-specific extension.

1.4.11 Binding Queues to Exchanges

Queues are bound to exchanges. This topic is described in detail in the *Queues and Consumers guide*.

1.4.12 Unbinding Queues from Exchanges

Queues are unbound from exchanges using. This topic is described in detail in the *Queues and Consumers guide*.

1.4.13 Deleting Exchanges

Explicitly Deleting an Exchange

Exchanges are deleted using the `AMQPExchange#delete`:

```
<?php

/* @var $exchange \AMQPExchange */
$exchange->delete();
```

Auto-deleted exchanges

Exchanges can be *auto-deleted*. To declare an exchange as auto-deleted, use the `:auto_delete` option on declaration:

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'header-exchange' => array(
                'name' => 'header-exchange',
                'type' => 'headers',
                'auto_delete' => true
            )
        ),
```

An auto-deleted exchange is removed when the last queue bound to it is unbound.

1.4.14 Exchange durability vs Message durability

See *Durability guide*

1.4.15 Wrapping Up

Publishers publish messages to exchanges. Messages are then routed to queues according to rules called bindings that applications define. There are 4 built-in exchange types in RabbitMQ and it is possible to create custom types.

Messages have a set of standard properties (e.g. type, content type) and can carry an arbitrary map of headers.

1.4.16 What to Read Next

The documentation is organized as *a number of guides*, covering various topics.

We recommend that you read the following guides first, if possible, in this order:

- *Bindings*
- RabbitMQ Extensions to AMQP 0.9.1
- *Durability and Related Matters*
- *Error Handling and Recovery*
- *Troubleshooting*

1.4.17 Tell Us What You Think!

Please take a moment to tell us what you think about this guide: Send an e-mail or raise an issue on Github.

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.5 Queues

1.5.1 Queues in AMQP 0.9.1: Overview

What are AMQP Queues?

Queues store and forward messages to consumers. They are similar to mailboxes in SMTP. Messages flow from producing applications to *Exchanges* that route them to queues and finally, queues deliver the messages to consumer applications (or consumer applications fetch messages as needed).

Note: Note that unlike some other messaging protocols/systems, messages are not delivered directly to queues. They are delivered to exchanges that route messages to queues using rules known as *bindings*.

AMQP 0.9.1 is a programmable protocol, so queues and bindings alike are declared by applications.

Concept of Bindings

A *binding* is an association between a queue and an exchange. Queues must be bound to at least one exchange in order to receive messages from publishers. Learn more about bindings in the *Bindings guide*.

Queue Attributes

Queues have several attributes associated with them:

- Name
- Exclusivity
- Durability
- Whether the queue is auto-deleted when no longer used

- Other metadata (sometimes called *X-arguments*)

These attributes define how queues can be used, their life-cycle, and other aspects of queue behavior.

1.5.2 Queue Names and Declaring Queues

Every AMQP queue has a name that identifies it. Queue names often contain several segments separated by a dot ".", in a similar fashion to URI path segments being separated by a slash "/", although almost any string can represent a segment (with some limitations - see below).

Before a queue can be used, it has to be *declared*. Declaring a queue will cause it to be created if it does not already exist. The declaration will have no effect if the queue does already exist and its attributes are the *same as those in the declaration*. When the existing queue attributes are not the same as those in the declaration a channel-level exception is raised. This case is explained later in this guide.

Explicitly Named Queues

Applications may pick queue names or ask the broker to generate a name for them. The configure a queue with an explicit name:

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct'
            )
        ),
        'queues' => array(
            'my-queue' => array(
                'name' => 'my-queue',
                'exchange' => 'demo-exchange'
            )
        )
    )
);
```

Server-named queues

To ask an AMQP broker to generate a unique queue name for you, pass an *empty string* as the queue name argument. A generated queue name (like *amq.gen-JZ46KgZEOZWg-pAScMhhig*) will be assigned to the queue instance that the method returns:

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct'
            )
        ),
```

```
        'queues' => array(
            'my-queue' => array(
                'name' => '',
                'exchange' => 'demo-exchange'
            )
        )
    )
);
```

Note: While it is common to declare server-named queues as `:exclusive`, it is not necessary.

Reserved Queue Name Prefix

Queue names starting with "amq." are reserved for server-named queues and queues for internal use by the broker. Attempts to declare a queue with a name that violates this rule will result in an AMQPExchangeException with reply code `403` and an exception message similar to this:

```
Server channel error: 403, message: ACCESS_REFUSED - exchange name 'amq.queue' contains reserved pre
```

This error results in the channel that was used for the declaration being forcibly closed by RabbitMQ. If the program subsequently tries to communicate with RabbitMQ using the same channel without re-opening it then the AMQP Extension will throw an `AMQPChannelException' with message 'Could not create exchange. No channel available`.

Queue Re-Declaration With Different Attributes

When queue declaration attributes are different from those that the queue already has, a channel-level exception with code `406` will be raised. The reply text will be similar to this:

```
Server channel error: 406, message: PRECONDITION_FAILED - cannot redeclare exchange 'foo' in vhost '
with different type, durable, internal or autodelete value
```

This error results in the channel that was used for the declaration being forcibly closed by RabbitMQ. If the program subsequently tries to communicate with RabbitMQ using the same channel without re-opening it then Bunny will throw an `AMQPChannelException' with message 'Could not create exchange. No channel available`. In order to continue communications in the same program after such an error, a different channel would have to be used.

1.5.3 Queue Life-cycle Patterns

According to the AMQP 0.9.1 specification, there are two common message queue life-cycle patterns:

- Durable queues that are shared by many consumers and have an independent existence: i.e. they will continue to exist and collect messages whether or not there are consumers to receive them.
- Temporary queues that are private to one consumer and are tied to that consumer. When the consumer disconnects, the message queue is deleted.

There are some variations of these, such as shared message queues that are deleted when the last of many consumers disconnects.

Let us examine the example of a well-known service like an event collector (event logger). A logger is usually up and running regardless of the existence of services that want to log anything at a particular point in time. Other applications

know which queues to use in order to communicate with the logger and can rely on those queues being available and able to survive broker restarts. In this case, explicitly named durable queues are optimal and the coupling that is created between applications is not an issue.

Another example of a well-known long-lived service is a distributed metadata/directory/locking server like Apache Zookeeper, Google's Chubby or DNS. Services like this benefit from using well-known, not server-generated, queue names and so do any other applications that use them.

A different sort of scenario is in "a cloud setting" when some kind of worker/instance might start and stop at any time so that other applications cannot rely on it being available. In this case, it is possible to use well-known queue names, but a much better solution is to use server-generated, short-lived queues that are bound to topic or fanout exchanges in order to receive relevant messages.

Imagine a service that processes an endless stream of events — Twitter is one example. When traffic increases, development operations may start additional application instances in the cloud to handle the load. Those new instances want to subscribe to receive messages to process, but the rest of the system does not know anything about them and cannot rely on them being online or try to address them directly. The new instances process events from a shared stream and are the same as their peers. In a case like this, there is no reason for message consumers not to use queue names generated by the broker.

In general, use of explicitly named or server-named queues depends on the messaging pattern that your application needs. Enterprise Integration Patterns discusses many messaging patterns in depth and the RabbitMQ FAQ also has a section on use cases.

1.5.4 Declaring a Durable Shared Queue

To declare a durable shared queue, you pass a queue name that is a non-blank string and use the `:durable` option:

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct'
            )
        ),
        'queues' => array(
            'my-queue' => array(
                'name' => 'demo-queue',
                'exchange' => 'demo-exchange',
                'durable' => true
            )
        )
    )
);
```

1.5.5 Declaring a Temporary Exclusive Queue

To declare a server-named, exclusive, auto-deleted queue, pass "" (an empty string) as the queue name and use the `:exclusive` option:

```
<?php

return array(
```

```
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct'
            )
        ),
        'queues' => array(
            'my-queue' => array(
                'name' => '',
                'exchange' => 'demo-exchange',
                'exclusive' => true
            )
        )
    )
);
```

Exclusive queues may only be accessed by the current connection and are deleted when that connection closes. The declaration of an exclusive queue by other connections is not allowed and will result in a channel-level exception with the code `405 (RESOURCE_LOCKED)`

Exclusive queues will be deleted when the connection they were declared on is closed.

1.5.6 Checking if a Queue Exists

Sometimes it's convenient to check if a queue exists. To do so, at the protocol level you use `queue.declareQueue` with `passive` set to `true`. In response RabbitMQ responds with a channel exception if the queue does not exist. This will lead to an 'AMQPQueueException' with message 'Server channel error: 404, message: NOT_FOUND - no queue 'test-queue' in vhost '/'

1.5.7 Binding Queues with Routing Keys

In order to receive messages, a queue needs to be bound to at least one exchange. Most of the time binding is explcit (done by applications). **Please note:** All queues are automatically bound to the default unnamed RabbitMQ direct exchange with a routing key that is the same as the queue name (see Exchanges and Publishing guide for more details).

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct'
            )
        ),
        'queues' => array(
            'my-queue' => array(
                'name' => 'demo-queue',
                'exchange' => 'demo-exchange',
                'routingKeys => array(
                    'v1.0.*',
                    'v1.1.0',
                    'v2.0.0'
                )
```

```
            )
        )
    )
);
```

1.5.8 Unbinding Queues From Exchanges

To unbind a queue from an exchange use the `AMQPQueue#unbind` function:

```
<?php

$queue->unbind('exchange-name');
```

Note: Trying to unbind a queue from an exchange that the queue was never bound to will result in a channel-level exception.

1.5.9 Purging queues

It is possible to purge a queue (remove all of the messages from it) using the `AMQPQueue#purge` method:

```
<?php

$queue->purge();
```

Note: When a server named queue is declared, it is empty, so for server-named queues, there is no need to purge them before they are used.

1.5.10 Deleting Queues

Queues can be deleted either indirectly or directly. To delete a queue indirectly you can include either of the following two arguments in the queue declaration:

- `:exclusive => true`
- `:auto_delete => true`

If the *exclusive* flag is set to true then the queue will be deleted when the connection that was used to declare it is closed.

If the *auto_delete* flag is set to true then the queue will be deleted when there are no more consumers subscribed to it. The queue will remain in existence until at least one consumer accesses it.

To delete a queue directly, use the `AMQPQueue#delete` method:

```
<?php

$queue->delete();
```

When a queue is deleted, all of the messages in it are deleted as well.

1.5.11 Queue Durability vs Message Durability

See Durability guide

1.5.12 RabbitMQ Extensions Related to Queues

See RabbitMQ Extensions guide

1.5.13 Wrapping Up

In RabbitMQ, queues can be client-named or server-named. For messages to be routed to queues, queues need to be bound to exchanges.

1.5.14 What to Read Next

The documentation is organized as a number of guides, covering various topics.

We recommend that you read the following guides first, if possible, in this order:

- Exchanges and Publishing
- Bindings
- RabbitMQ Extensions to AMQP 0.9.1
- Durability and Related Matters
- Error Handling and Recovery
- Concurrency Considerations
- Troubleshooting
- Using TLS (SSL) Connections

1.5.15 Tell Us What You Think!

Please take a moment to tell us what you think about this guide on Twitter or the Bunny mailing list

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.6 Bindings

1.6.1 What Are AMQP 0.9.1 Bindings

Bindings are rules that exchanges use (among other things) to route messages to queues. To instruct an exchange E to route messages to a queue Q, Q has to *be bound* to E. Bindings may have an optional *routing key* attribute used by some exchange types. The purpose of the routing key is to selectively match only specific (matching) messages published to an exchange to the bound queue. In other words, the routing key acts like a filter.

To draw an analogy:

- Queue is like your destination in New York city

- Exchange is like JFK airport
- Bindings are routes from JFK to your destination. There may be no way, or more than one way, to reach it

Some exchange types use routing keys while some others do not (routing messages unconditionally or based on message metadata). If an AMQP message cannot be routed to any queue (for example, because there are no bindings for the exchange it was published to), it is either dropped or returned to the publisher, depending on the message attributes that the publisher has set.

If an application wants to connect a queue to an exchange, it needs to *bind* them. The opposite operation is called *unbinding*.

1.6.2 Binding Queues to Exchanges

In order to receive messages, a queue needs to be bound to at least one exchange. Most of the time binding is explcit (done by applications).

Example:

```
.. code-block:: php

<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct'
            )
        ),
        'queues' => array(
            'my-queue' => array(
                'name' => 'my-queue',
                'exchange' => 'demo-exchange'
            )
        )
    )
);
```

1.6.3 Unbinding queues from exchanges

```
<?php

$queue->unbind('exchange-name');
```

1.6.4 Exchange-to-Exchange Bindings

Exchange-to-Exchange bindings is a RabbitMQ extension to AMQP 0.9.1. It is covered in the RabbitMQ extensions guide.

1.6.5 Bindings, Routing and Returned Messages

How RabbitMQ Routes Messages

After a message reaches RabbitMQ and before it reaches a consumer, several things happen:

- RabbitMQ needs to find one or more queues that the message needs to be routed to, depending on type of exchange
- RabbitMQ puts a copy of the message into each of those queues or decides to return the message to the publisher
- RabbitMQ pushes message to consumers on those queues or waits for applications to fetch them on demand

A more in-depth description is this:

- RabbitMQ needs to consult bindings list for the exchange the message was published to in order to find one or more queues that the message needs to be routed to (step 1)
- If there are no suitable queues found during step 1 and the message was published as mandatory, it is returned to the publisher (step 1b)
- If there are suitable queues, a *copy* of the message is placed into each one (step 2)
- If the message was published as mandatory, but there are no active consumers for it, it is returned to the publisher (step 2b)
- If there are active consumers on those queues and the basic.qos setting permits, message is pushed to those consumers (step 3)

The important thing to take away from this is that messages may or may not be routed and it is important for applications to handle unroutable messages.

Handling of Unroutable Messages

Unroutable messages are either dropped or returned to producers. RabbitMQ extensions can provide additional ways of handling unroutable messages: for example, RabbitMQ's Alternate Exchanges extension makes it possible to route unroutable messages to another exchange. Bunny support for it is documented in the RabbitMQ Extensions guide.

Exchanges and Publishing documentation guide provides more information on the subject, including full code examples.

1.6.6 What to Read Next

The documentation is organized as a number of guides, covering various topics.

We recommend that you read the following guides first, if possible, in this order:

- RabbitMQ Extensions to AMQP 0.9.1
- Durability and Related Matters
- Error Handling and Recovery
- Troubleshooting

1.6.7 Tell Us What You Think!

Please take a moment to tell us what you think about this guide: Send an e-mail or raise an issue on Github.

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.7 Consumers

The Humus AMQP Module provides a default consumer implementation that suites most use-cases. If you have a special use-case, you can extend this class or implement the consumer interface yourself.

1.7.1 Consumer Callbacks

In order do reduce extending consumer classes and avoid factory duplication, the consumer expects a delivery callback. This callback get executed every time a new message gets delivered to the consumer. The consumer expects the callback to take 3 arguments: The envelope, the queue and used consumer. A very simple callback would look like this:

```
<?php

$callback = function(AMQPEnvelope $envelope, AMQPQueue $queue, ConsumerInterface $consumer) {
    echo $envelope->getBody();
    return true;
}
```

or

```
<?php

class MyCallback
{
    public function __invoke(AMQPEnvelope $envelope, AMQPQueue $queue, ConsumerInterface $consumer)
    {
        echo $envelope->getBody();
        return true;
    }
}
```

The “return true” at the end of the callback will signal the consumer that the message was processed correctly and the consumer will send an ack. If you return false, the messages will get rejected and requeued once. When the same message gets rejected the second time, the message will not be requeued again. If you don’t return anything (or return null) the message will get deferred, until the block size is reache or an timeout occurs. So you can handle blocks of messages.

Another possiblity of returning is to return ConsumerInterace::MSG_ACK, ::MSG_DEFER, ::MSG_REJECT, or ::MSG_REJECT_REQUEUE.

1.7.2 Handling Messages in Batches

If you have collected messages (returned null or ConsumerIntrace::MSG_DEFER in the delivery callback) and the block size or timeout is reached, the flush callback will get executed. If you did not specify a flush callback, it will return true leading to all messages collected being acknowleged at once. You have the possibility to add a custom flush

callback where you have to take care whether or not you return true or false. Note that any other value than true, will lead to all messages rejected in the current block.

1.7.3 Message Acknowledgements & Rejecting

Consumer applications — applications that receive and process messages , may occasionally fail to process individual messages, or will just crash. There is also the possibility of network issues causing problems. This raises a question — "When should the AMQP broker remove messages from queues?"

The AMQP 0.9.1 specification proposes two choices:

- After broker sends a message to an application (using either basic.deliver or basic.get-ok methods).
- After the application sends back an acknowledgement (using basic.ack AMQP method).

The former choice is called the *automatic acknowledgement model*, while the latter is called the *explicit acknowledgement model*. With the explicit model, the application chooses when it is time to send an acknowledgement. It can be right after receiving a message, or after persisting it to a data store before processing, or after fully processing the message (for example, successfully fetching a Web page, processing and storing it into some persistent data store).

Note: Acknowledgements are channel-specific. Applications MUST NOT receive messages on one channel and acknowledge them on another.

1.7.4 Logging

The consumer expects you to inject a logger instance, if you don't provide one, a null-logger will be created and injected for you.

1.7.5 Error-Handling

By default, all errors are logged on the configured logger. If you want to, you can specify your own error callback that will get executed instead.

1.7.6 QoS — Prefetching messages

For cases when multiple consumers share a queue, it is useful to be able to specify how many messages each consumer can be sent at once before sending the next acknowledgement. This can be used as a simple load balancing technique to improve throughput if messages tend to be published in batches. For example, if a producing application sends messages every minute because of the nature of the work it is doing.

Imagine a website that takes data from social media sources like Twitter or Facebook during the Champions League (european soccer) final (or the Superbowl), and then calculates how many tweets mentioned a particular team during the last minute. The site could be structured as 3 applications:

- A crawler that uses streaming APIs to fetch tweets/statuses, normalizes them and sends them in JSON for processing by other applications ("app A").
- A calculator that detects what team is mentioned in a message, updates statistics and pushes an update to the Web UI once a minute ("app B").
- A Web UI that fans visit to see the stats ("app C").

In this imaginary example, the "tweets per second" rate will vary, but to improve the throughput of the system and to decrease the maximum number of messages that the AMQP broker has to hold in memory at once, applications can be designed in such a way that application "app B", the "calculator", receives 5000 messages and then acknowledges them all at once. The broker will not send message 5001 unless it receives an acknowledgement.

In AMQP 0.9.1 parlance this is known as *QoS* or *message prefetching*. Prefetching is configured on a per-channel basis.

The default implementation of the Humus AMQP Module's consumer will work with prefetch count, so if you set the prefetch count to 50, a block size of 50 messages will be used.

1.7.7 Timeouts

The idle timeout takes effect, when there are no more messages coming in and you expect a block size > 1. The wait timeout applies every time the consumer tries to fetch a message from the queue but doesn't receive any.

1.7.8 Set up the consumer

```
<?php

return array(
    'humus_amqp_module' => array(
        'consumers' => array(
            'demo-consumer' => array(
                'queues' => array(
                    'queue1',
                    'queue2'
                ),
                'auto_setup_fabric' => true,
                'callback' => 'echoCallback',
                'flush_callback' => 'flushCallback',
                'error_callback' => 'errorCallback',
                'idle_timeout' => 5.0, // secs
                'wait_timeout' => 5000, // microsecs
                'logger' => 'consumer-logger'
            )
        ),
        'plugin_managers' => array(
            'callback' => array(
                'invokables' => array(
                    'echoCallback' => 'My\Callback\Echo',
                    'flushCallback' => 'My\Callback\Flush',
                    'errorCallback' => 'My\Callback\Error',
                )
            )
        )
    )
);
```

1.7.9 Using Multiple Consumers Per Queue

It is possible to have multiple non-exclusive consumers on queues. In that case, messages will be distributed between them according to prefetch levels of their channels (more on this later in this guide). If prefetch values are equal for all consumers, each consumer will get about the same number of messages.

1.7.10 Starting a consumer

```
php public/index.php humus amqp consumer my-consumer
```

See: *CLI Usage* for more informations.

1.7.11 Killing a Consumer gracefully

You can send a SIGUSER1 signal to gracefully shutdown the consumer (if started from the consumer controller in Humus AMQP Module).

```
kill -10 23453
```

Where 23453 is the process id of the consumer process.

QoS — Prefetching messages

For cases when multiple consumers share a queue, it is useful to be able to specify how many messages each consumer can be sent at once before sending the next acknowledgement. This can be used as a simple load balancing technique to improve throughput if messages tend to be published in batches. For example, if a producing application sends messages every minute because of the nature of the work it is doing.

Imagine a website that takes data from social media sources like Twitter or Facebook during the Champions League (european soccer) final (or the Superbowl), and then calculates how many tweets mentioned a particular team during the last minute. The site could be structured as 3 applications:

- A crawler that uses streaming APIs to fetch tweets/statuses, normalizes them and sends them in JSON for processing by other applications ("app A").
- A calculator that detects what team is mentioned in a message, updates statistics and pushes an update to the Web UI once a minute ("app B").
- A Web UI that fans visit to see the stats ("app C").

In this imaginary example, the "tweets per second" rate will vary, but to improve the throughput of the system and to decrease the maximum number of messages that the AMQP broker has to hold in memory at once, applications can be designed in such a way that application "app B", the "calculator", receives 5000 messages and then acknowledges them all at once. The broker will not send message 5001 unless it receives an acknowledgement.

In AMQP 0.9.1 parlance this is known as *QoS* or *message prefetching*. Prefetching is configured on a per-channel basis.

Note: The prefetch setting is ignored for consumers that do not use explicit acknowledgements.

1.7.12 What to Read Next

The documentation is organized as *a number of guides*, covering various topics.

We recommend that you read the following guides first, if possible, in this order:

- *Error Handling and Recovery*
- *Troubleshooting*

1.7.13 Tell Us What You Think!

Please take a moment to tell us what you think about this guide: Send an e-mail or raise an issue on Github.

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.8 CLI Usage

1.8.1 Overview of available commands

You can quickly get an overview of all available cli commands (incl. the commands from the Humus AMQP Module) by running:

```
$ php public/index.php
```

1.9 Durability

1.9.1 Entity durability and message persistence

Exchange Durability

AMQP separates the concept of entity durability (queues, exchanges) from message persistence. Exchanges can be durable or transient. Durable exchanges survive broker restart, transient exchanges do not (they have to be redeclared when the broker comes back online), however, not all scenarios and use cases mandate exchanges to be durable.

To create a durable exchange, declare it with the `:durable => true` argument.

Queue Durability

Queues can be durable or transient. Durable queues survive broker restart, transient queues do not (they have to be redeclared when the broker comes back online), however, not all scenarios and use cases mandate queues to be durable.

To create a durable queue, declare it with the `:durable => true` argument.

Durability of a queue does not make *messages* that are routed to that queue durable. If a broker is taken down and then brought back up, durable queues will be re-declared during broker startup, however, only *persistent* messages will be recovered.

Binding Durability

Bindings of durable queues to durable exchanges are automatically durable and are restored after a broker restart. The AMQP 0.9.1 specification states that the binding of durable queues to transient exchanges must be allowed. In this case, since the exchange would not survive a broker restart, neither would any bindings to such and exchange.

Message Persistence

Messages may be published as persistent and this, in conjunction with queue durability, is what makes an AMQP broker persist them to disk. If the server is restarted, the system ensures that received persistent messages in durable queues are not lost. Simply publishing a message to a durable exchange or the fact that a queue to which a message is

routed is durable does not make that message persistent. Message persistence depends on the persistence mode of the message itself.

Note: Publishing persistent messages affects performance (just like with data stores, durability comes at a certain cost to performance).

Pass the `:persistent => true` argument to the `AMQPExchange#publish` method to publish your message as persistent.

Clustering and High Availability

To achieve the degree of durability that critical applications need, it is necessary but not enough to use durable queues, exchanges and persistent messages. You need to use a cluster of brokers because otherwise, a single hardware problem may bring a broker down completely.

RabbitMQ offers a number of high availability features for both scenarios with more (LAN) and less (WAN) reliable network connections.

See the RabbitMQ clustering and high availability guides for in-depth discussion of this topic.

Highly Available (Mirrored) Queues

Whilst the use of clustering provides for greater durability of critical systems, in order to achieve the highest level of resilience for queues and messages, high availability configuration should be used. This is because although exchanges and bindings survive the loss of individual nodes by using clustering, messages do not. Without mirroring, queue contents reside on exactly one node, thus the loss of a node will cause message loss.

See the RabbitMQ high availability guide for more information about mirrored queues.

1.9.2 What to Read Next

The documentation is organized as a number of guides, covering various topics.

We recommend that you read the following guides first, if possible, in this order:

- Queues and Consumers
- Exchanges and Publishing
- Bindings
- RabbitMQ Extensions to AMQP 0.9.1

1.9.3 Tell Us What You Think!

Please take a moment to tell us what you think about this guide on Twitter or the Bunny mailing list

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.10 RabbitMQ Extensions

Humus AMQP Module supports all RabbitMQ extensions to AMQP 0.9.1 that the PHP AMQP Extension supports, too:

- Negative acknowledgements (basic.nack)
- Exchange-to-Exchange Bindings
- Alternate Exchanges
- Per-queue Message Time-to-Live
- Per-message Time-to-Live
- Queue Leases
- Sender-selected Distribution
- Dead Letter Exchanges

The following RabbitMQ extensions are note supported due to php extension limitations:

- Publisher confirms
- Validated user_id

This guide briefly describes how to use these extensions with Humus AMQP Module.

1.10.1 Enabling RabbitMQ Extensions

You don't need to require any additional files to make Humus AMQP Module support RabbitMQ extensions. The support is built into the core.

1.10.2 Per-queue Message Time-to-Live

Per-queue Message Time-to-Live (TTL) is a RabbitMQ extension to AMQP 0.9.1 that allows developers to control how long a message published to a queue can live before it is discarded. A message that has been in the queue for longer than the configured TTL is said to be dead. Dead messages will not be delivered to consumers and cannot be fetched.

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct'
            )
        ),
        'queues' => array(
            'my-queue' => array(
                'name' => 'my-queue',
                'exchange' => 'demo-exchange',
                'arguments' => array(
                    'x-message-ttl' => 1000
                )
            )
```

```
        )
    )
);
```

When a published message is routed to multiple queues, each of the queues gets a *copy of the message*. If the message subsequently dies in one of the queues, it has no effect on copies of the message in other queues.

Learn More

See also rabbitmq.com section on Per-queue Message TTL

1.10.3 basic.nack

The AMQP 0.9.1 specification defines the basic.reject method that allows clients to reject individual, delivered messages, instructing the broker to either discard them or requeue them. Unfortunately, basic.reject provides no support for negatively acknowledging messages in bulk.

To solve this, RabbitMQ supports the basic.nack method that provides all of the functionality of basic.reject whilst also allowing for bulk processing of messages.

How To Use It With Humus AMQP Module

The Humus AMQP Module makes already use of the nack method to reject a block of messages. You don't need to take care of that, unless you want to write a consumer yourself.

Learn More

See also rabbitmq.com section on basic.nack

1.10.4 Alternate Exchanges

The Alternate Exchanges RabbitMQ extension to AMQP 0.9.1 allows developers to define "fallback" exchanges where unroutable messages will be sent.

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct',
                'arguments' => array(
                    'alternate_exchange' => 'alternate-exchange-name'
                )
            )
        ),
    )
);
```

Learn More

See also rabbitmq.com section on Alternate Exchanges

1.10.5 Exchange-To-Exchange Bindings

RabbitMQ supports exchange-to-exchange bindings to allow even richer routing topologies as well as a backbone for some other features (e.g. tracing).

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct',
                'exchange_bindings' => array(
                    'exchange1' => array(
                        'routingKey.1',
                        'routingKey.2'
                    ),
                    'exchange2' => array(
                        'routingKey.3'
                    )
                )
            )
        ),
    )
);
```

Learn More

See also rabbitmq.com section on Exchange-to-Exchange Bindings

1.10.6 Queue Leases

Queue Leases is a RabbitMQ feature that lets you set for how long a queue is allowed to be *unused*. After that moment, it will be deleted. *Unused* here means that the queue

- has no consumers
- is not redeclared
- no message fetches happened

```
<?php

return array(
    'humus_amqp_module' => array(
        'exchanges' => array(
            'demo-exchange' => array(
                'name' => 'demo-exchange',
                'type' => 'direct',
                'arguments' => array(
                    'x-expires' => 10000
```

```
                    )
                )
            ),
        )
);
```

Learn More

See also rabbitmq.com section on Queue Leases

1.10.7 Per-Message Time-to-Live

A TTL can be specified on a per-message basis, by setting the `:expiration` property when publishing.

```
<?php

$attribs = new MessageAttributes()
$attribs->setExpiration(5000);

$producer->publish('some message', '', $attribs);
```

Learn More

See also rabbitmq.com section on Per-message TTL

1.10.8 Sender-Selected Distribution

Generally, the RabbitMQ model assumes that the broker will do the routing work. At times, however, it is useful for routing to happen in the publisher application. Sender-Selected Routing is a RabbitMQ feature that lets clients have extra control over routing.

The values associated with the `"CC"` and `"BCC"` header keys will be added to the routing key if they are present. If neither of those headers is present, this extension has no effect.

```
<?php

$attribs = new MessageAttributes()
$attribs->setHeaders(array(
    'CC' => array('two', 'three')
));

$producer->publish('some message', '', $attribs);
```

Learn More

See also rabbitmq.com section on Sender-Selected Distribution

1.10.9 Dead Letter Exchange (DLX)

The x-dead-letter-exchange argument to queue.declare controls the exchange to which messages from that queue are 'dead-lettered'. A message is dead-lettered when any of the following events occur:

The message is rejected (basic.reject or basic.nack) with requeue=false; or the TTL for the message expires.

How To Use It With Bunny 0.9+

Dead-letter Exchange is a feature that is used by specifying additional queue arguments:

- `"x-dead-letter-exchange"` specifies the exchange that dead lettered messages should be published to by RabbitMQ
- `"x-dead-letter-routing-key"` specifies the routing key that should be used (has to be a constant value)

```
<?php

return array(
    'humus_amqp_module' => array(
        'queues' => array(
            'foo' => array(
                'name' => 'foo',
                'exchange' => 'demo',
                'arguments' => array(
                    'x-dead-letter-exchange' => 'demo.error'
                ),
        ),
    )
);
```

Learn More

See also rabbitmq.com section on Dead Letter Exchange

1.10.10 Wrapping Up

RabbitMQ provides a number of useful extensions to the AMQP 0.9.1 specification.

Bunny 0.9 and later releases have RabbitMQ extensions support built into the core. Some features are based on optional arguments for queues, exchanges or messages, and some are Bunny public API features. Any future argument-based extensions are likely to be useful with Bunny immediately, without any library modifications.

1.10.11 What to Read Next

The documentation is organized as a number of guides, covering various topics.

We recommend that you read the following guides first, if possible, in this order:

- Durability and Related Matters
- Error Handling and Recovery
- Troubleshooting

1.10.12 Tell Us What You Think!

Please take a moment to tell us what you think about this guide: Send an e-mail or raise an issue on Github.

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.11 Error Handling

1.11.1 Client Exceptions

Here is the break-down of exceptions that can be raised by the PHP AMQP Extension:

```
AMQPChannelException
AMQPConnectionException
AMQPExchangeException
AMQPQueueException
```

Additionally the Humus AMQP Module throws some of the following exceptions

```
HumusAmqpModule\Exception\BadFunctionCallException
HumusAmqpModule\Exception\BadMethodCallException
HumusAmqpModule\Exception\ExtensionNotLoadedException
HumusAmqpModule\Exception\InvalidArgumentException
HumusAmqpModule\Exception\RuntimeException
```

The first 3 exceptions only occur, when you don't have ext-pcntl installed but you try to start the consumer without the –without-signals switch.

The InvalidArgumentException occurs, when you have a wrong amqp module configuration.

1.11.2 Initial RabbitMQ Connection Failures

When applications connect to the broker, they need to handle connection failures. Networks are not 100% reliable, even with modern system configuration tools like Chef or Puppet misconfigurations happen and the broker might also be down. Error detection should happen as early as possible. To handle TCP connection failure, catch the `AMQPConnection` exception.

1.11.3 Authentication Failures

Another reason why a connection may fail is authentication failure. Handling authentication failure is very similar to handling initial TCP connection failure.

When you try to access RabbitMQ with invalid credentials, you'll get an 'AMQPConnectionException' with message 'Library error: a socket error occurred - Potential login failure.'.

In case you are wondering why the exception name has "potential" in it: AMQP 0.9.1 spec requires broker implementations to simply close TCP connection without sending any more data when an exception (such as authentication failure) occurs before AMQP connection is open. In practice, however, when broker closes TCP connection between successful TCP connection and before AMQP connection is open, it means that authentication has failed.

1.11.4 Channel-level Exceptions

Channel-level exceptions are more common than connection-level ones and often indicate issues applications can recover from (such as consuming from or trying to delete a queue that does not exist).

Common channel-level exceptions and what they mean

A few channel-level exceptions are common and deserve more attention.

406 Precondition Failed

Description

The client requested a method that was not allowed because some precondition failed.

What might cause it

AMQP entity (a queue or exchange) was re-declared with attributes different from original declaration. Maybe two applications or pieces of code declare the same entity with different attributes. Note that different RabbitMQ client libraries historically use slightly different defaults for entities and this may cause attribute mismatches.

PRECONDITION_FAILED - parameters for queue 'examples.channel_exception' in vhost '/' not equivalent

PRECONDITION_FAILED - channel is not transactional

405 Resource Locked

Description

The client attempted to work with a server entity to which it has no access because another client is working with it.

What might cause it

Multiple applications (or different pieces of code/threads/processes/routines within a single application) might try to declare queues with the same name as exclusive.

Multiple consumer across multiple or single app might be registered as exclusive for the same queue.

Example RabbitMQ error message

RESOURCE_LOCKED - cannot obtain exclusive access to locked queue 'examples.queue' in vhost '/'

404 Not Found

Description

The client attempted to use (publish to, delete, etc) an entity (exchange, queue) that does not exist.

What might cause it

Application miscalculates queue or exchange name or tries to use an entity that was deleted earlier

Example RabbitMQ error message

NOT_FOUND - no queue 'queue_that_should_not_exist0.6798199937619038' in vhost '/'

403 Access Refused

Description

The client attempted to work with a server entity to which it has no access due to security settings.

What might cause it

Application tries to access a queue or exchange it has no permissions for (or right kind of permissions, for example, write permissions)

Example RabbitMQ error message

ACCESS_REFUSED - access to queue 'examples.channel_exception' in vhost '_testbed' refused for user '_reader'

1.11.5 What to Read Next

The documentation is organized as a number of guides, covering various topics.

We recommend that you read the following guides first, if possible, in this order:

- Troubleshooting

1.11.6 Tell Us What You Think!

Please take a moment to tell us what you think about this guide: Send an e-mail or raise an issue on Github.

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.12 Troubleshooting

1.13 Deployment Strategies

1.13.1 Shut down the system, update and restart

While the easiest way to deploy a RabbitMQ environment is to just destroy the old one and create a new one, that's not always possible. Sometimes the consumers are not part of your application or it's complete unacceptable to put the payment system offline and not to process messages even for half an hour, things get more complicated.

This guide tries to show some different deployment strategies. Note, that there is no general way to do this, it will always depend on the use-case you have.

1.13.2 Create a new node and switch configuration

Let's say you have a running rabbitmq configuration on a given node (or vhost). An easy way to update configuration would be to just deploy a new node (or vhost) and switch you application configuration to use that new node (vhost) instead of the old one.

1.13.3 Message-Versioning

Message-Versioning with Routing Keys

You can use the routing keys for versioning, like this:

```
<?php

$producer->publish('some message', 'v1.0.0');
```

Then you just bind the queue to the routing key. When you update your system, newer messages (e.g. 2.0.0) are not delivered to queue that is not able to process them. On the other hand you can write consumers, that are backwards-compatible, so the queues used will bind to a variety of routing keys (v1.0.0 & v2.0.0 or v1.0.*).

Note: Use semantic versioning.

Message-Versioning with extra Attributes

Use some custom message headers to specify version constraints:

```
<?php

$attribs = new MessageAttributes();
$attribs->setHeaders(array(
    'x-version' => 'v1.0.0'
));
$producer->publish('some message', '', $attribs);
```

This way a message with a version not handled by your consumer, will also be at least delivered to him. The consumer will then need to check for the x-version header and process if possible or throw an exception. Compared to the versioning strategy with routing keys, you'll always know, that somebody is sending still messages in old version, as you see the exceptions in the log file. If you just deploy a consumer that is only able to acknowlegde messages of e.g. v2.0.0, than it's hard to recognize that there are still messages in version 1.0.0 going through the system. On the other hand, there are messages send over network, that nobody cares, so routing keys are often preferable.

1.13.4 Updating Exchanges

Most times it's not neccessary to update an exchange configuration. However if required, you have to remove the old exchange before you can redeclare the new one with the same name. Keep in mind, that when you put an exchange down, no messages can be delivered any more. It's also possible to use an exchange name like: "update-stuff-v1-0-0", so you don't have to deleted the old exchange when deploying the new one.

1.13.5 Updating Queues

Updating queues can be sometimes a little more tricky then updating an exchange. If you bind the new queue before the old one is destroyed, you'll get duplicated messages in your system. But if this is done in concert with a new exchange you can prevent duplicate messages.

1.13.6 Getting queues empty first

Sometimes you might want to upgrade the consumers because of a new message format and you don't want to maintain backwards compatibility. If you don't want to lose any messages, stop all producers first, untill the queues are empty, then you can to the switch without losing messages.

1.13.7 What to Read Next

The documentation is organized as *a number of guides*, covering various topics.

We recommend that you read the following guides first, if possible, in this order:

- *Error Handling and Recovery*
- *Troubleshooting*

1.13.8 Tell Us What You Think!

Please take a moment to tell us what you think about this guide: Send an e-mail or raise an issue on Github.

Let us know what was unclear or what has not been covered. Maybe you do not like the guide style or grammar or discover spelling mistakes. Reader feedback is key to making the documentation better.

1.14 License Information

Indices and tables

- search