pika

Release 1.2.1

Apr 27, 2022

Contents

Pika is a pure-Python implementation of the AMQP 0-9-1 protocol that tries to stay fairly independent of the underlying network support library.

If you have not developed with Pika or RabbitMQ before, the *Introduction to Pika* documentation is a good place to get started.

CHAPTER 1

Installing Pika

Pika is available for download via PyPI and may be installed using easy_install or pip:

```
pip install pika
```

or:

```
easy_install pika
```

To install from source, run "python setup.py install" in the root source directory.

CHAPTER 2

Using Pika

2.1 Introduction to Pika

2.1.1 IO and Event Looping

As AMQP is a two-way RPC protocol where the client can send requests to the server and the server can send requests to a client, Pika implements or extends IO loops in each of its asynchronous connection adapters. These IO loops are blocking methods which loop and listen for events. Each asynchronous adapter follows the same standard for invoking the IO loop. The IO loop is created when the connection adapter is created. To start an IO loop for any given adapter, call the `connection.ioloop.start()` method.

If you are using an external IO loop such as Tornado's `IOLoop` you invoke it normally and then add the Pika Tornado adapter to it.

Example:

```
import pika

def on_open(connection):
    # Invoked when the connection is open
    pass

# Create our connection object, passing in the on_open method
connection = pika.SelectConnection(on_open_callback=on_open)

try:
    # Loop so we can communicate with RabbitMQ
    connection.ioloop.start()
except KeyboardInterrupt:
    # Gracefully close the connection
    connection.close()
    # Loop until we're fully closed, will stop on its own
    connection.ioloop.start()
```

2.1.2 Continuation-Passing Style

Interfacing with Pika asynchronously is done by passing in callback methods you would like to have invoked when a certain event completes. For example, if you are going to declare a queue, you pass in a method that will be called when the RabbitMQ server returns a Queue.DeclareOk response.

In our example below we use the following five easy steps:

1. We start by creating our connection object, then starting our event loop.
2. When we are connected, the *on_connected* method is called. In that method we create a channel.
3. When the channel is created, the *on_channel_open* method is called. In that method we declare a queue.
4. When the queue is declared successfully, *on_queue_declared* is called. In that method we call `channel.basic_consume` telling it to call the handle_delivery for each message RabbitMQ delivers to us.
5. When RabbitMQ has a message to send us, it calls the handle_delivery method passing the AMQP Method frame, Header frame, and Body.

Note: Step #1 is on line #28 and Step #2 is on line #6. This is so that Python knows about the functions we'll call in Steps #2 through #5.

Example:

```
import pika

# Create a global channel variable to hold our channel object in
channel = None

# Step #2
def on_connected(connection):
    """Called when we are fully connected to RabbitMQ"""
    # Open a channel
    connection.channel(on_open_callback=on_channel_open)

# Step #3
def on_channel_open(new_channel):
    """Called when our channel has opened"""
    global channel
    channel = new_channel
    channel.queue_declare(queue="test", durable=True, exclusive=False, auto_
↪delete=False, callback=on_queue_declared)

# Step #4
def on_queue_declared(frame):
    """Called when RabbitMQ has told us our Queue has been declared, frame is the␣
↪response from RabbitMQ"""
    channel.basic_consume('test', handle_delivery)

# Step #5
def handle_delivery(channel, method, header, body):
    """Called when we receive a message from RabbitMQ"""
    print(body)

# Step #1: Connect to RabbitMQ using the default parameters
parameters = pika.ConnectionParameters()
connection = pika.SelectConnection(parameters, on_open_callback=on_connected)
```

(continues on next page)

(continued from previous page)

```
try:
    # Loop so we can communicate with RabbitMQ
    connection.ioloop.start()
except KeyboardInterrupt:
    # Gracefully close the connection
    connection.close()
    # Loop until we're fully closed, will stop on its own
    connection.ioloop.start()
```

2.1.3 Credentials

The `pika.credentials` module provides the mechanism by which you pass the username and password to the `ConnectionParameters` class when it is created.

Example:

```
import pika
credentials = pika.PlainCredentials('username', 'password')
parameters = pika.ConnectionParameters(credentials=credentials)
```

2.1.4 Connection Parameters

There are two types of connection parameter classes in Pika to allow you to pass the connection information into a connection adapter, `ConnectionParameters` and `URLParameters`. Both classes share the same default connection values.

2.1.5 TCP Backpressure

As of RabbitMQ 2.0, client side Channel.Flow has been removed[1]. Instead, the RabbitMQ broker uses TCP Backpressure to slow your client if it is delivering messages too fast. If you pass in backpressure_detection into your connection parameters, Pika attempts to help you handle this situation by providing a mechanism by which you may be notified if Pika has noticed too many frames have yet to be delivered. By registering a callback function with the `add_backpressure_callback` method of any connection adapter, your function will be called when Pika sees that a backlog of 10 times the average frame size you have been sending has been exceeded. You may tweak the notification multiplier value by calling the `set_backpressure_multiplier` method passing any integer value.

Example:

```
import pika

parameters = pika.URLParameters('amqp://guest:guest@rabbit-server1:5672/%2F?
↪backpressure_detection=t')
```

2.2 Core Class and Module Documentation

For the end user, Pika is organized into a small set of objects for all communication with RabbitMQ.

[1] "more effective flow control mechanism that does not require cooperation from clients and reacts quickly to prevent the broker from exhausting memory - see http://lists.rabbitmq.com/pipermail/rabbitmq-announce/attachments/20100825/2c672695/attachment.txt

- A *connection adapter* is used to connect to RabbitMQ and manages the connection.
- *Connection parameters* are used to instruct the `Connection` object how to connect to RabbitMQ.
- *Authentication Credentials* are used to encapsulate all authentication information for the `ConnectionParameters` class.
- A `Channel` object is used to communicate with RabbitMQ via the AMQP RPC methods.
- *Exceptions* are raised at various points when using Pika when something goes wrong.

2.2.1 Connection Adapters

Pika uses connection adapters to provide a flexible method for adapting pika's core communication to different IOLoop implementations. In addition to asynchronous adapters, there is the `BlockingConnection` adapter that provides a more idiomatic procedural approach to using Pika.

Adapters

BlockingConnection

The blocking connection adapter module implements blocking semantics on top of Pika's core AMQP driver. While most of the asynchronous expectations are removed when using the blocking connection adapter, it attempts to remain true to the asynchronous RPC nature of the AMQP protocol, supporting server sent RPC commands.

The user facing classes in the module consist of the `BlockingConnection` and the `BlockingChannel` classes.

Be sure to check out examples in *Usage Examples*.

class `pika.adapters.blocking_connection.`**`BlockingConnection`**(*parameters=None*, *_impl_class=None*)

The BlockingConnection creates a layer on top of Pika's asynchronous core providing methods that will block until their expected response has returned. Due to the asynchronous nature of the *Basic.Deliver* and *Basic.Return* calls from RabbitMQ to your application, you can still implement continuation-passing style asynchronous methods if you'd like to receive messages from RabbitMQ using `basic_consume` or if you want to be notified of a delivery failure when using `basic_publish`.

For more information about communicating with the blocking_connection adapter, be sure to check out the `BlockingChannel` class which implements the `Channel` based communication for the blocking_connection adapter.

To prevent recursion/reentrancy, the blocking connection and channel implementations queue asynchronously-delivered events received in nested context (e.g., while waiting for *BlockingConnection.channel* or *BlockingChannel.queue_declare* to complete), dispatching them synchronously once nesting returns to the desired context. This concerns all callbacks, such as those registered via *BlockingConnection.call_later*, *BlockingConnection.add_on_connection_blocked_callback*, *BlockingConnection.add_on_connection_unblocked_callback*, *BlockingChannel.basic_consume*, etc.

Blocked Connection deadlock avoidance: when RabbitMQ becomes low on resources, it emits Connection.Blocked (AMQP extension) to the client connection when client makes a resource-consuming request on that connection or its channel (e.g., *Basic.Publish*); subsequently, RabbitMQ suspsends processing requests from that connection until the affected resources are restored. See http://www.rabbitmq.com/connection-blocked.html. This may impact *BlockingConnection* and *BlockingChannel* operations in a way that users might not be expecting. For example, if the user dispatches *BlockingChannel.basic_publish* in non-publisher-confirmation mode while RabbitMQ is in this low-resource state followed by a synchronous request (e.g., *BlockingConnection.channel*, *BlockingChannel.consume*, *BlockingChannel.basic_consume*, etc.), the synchronous request

will block indefinitely (until Connection.Unblocked) waiting for RabbitMQ to reply. If the blocked state persists for a long time, the blocking operation will appear to hang. In this state, *BlockingConnection* instance and its channels will not dispatch user callbacks. SOLUTION: To break this potential deadlock, applications may configure the *blocked_connection_timeout* connection parameter when instantiating *BlockingConnection*. Upon blocked connection timeout, this adapter will raise ConnectionBlockedTimeout exception‘. See *pika.connection.ConnectionParameters* documentation to learn more about the *blocked_connection_timeout* configuration.

add_callback_threadsafe(*callback*)

Requests a call to the given function as soon as possible in the context of this connection's thread.

NOTE: This is the only thread-safe method in *BlockingConnection*. All other manipulations of *BlockingConnection* must be performed from the connection's thread.

NOTE: the callbacks are dispatched only in the scope of specially-designated methods: see *BlockingConnection.process_data_events()* and *BlockingChannel.start_consuming()*.

For example, a thread may request a call to the *BlockingChannel.basic_ack* method of a *BlockingConnection* that is running in a different thread via:

```
connection.add_callback_threadsafe(
    functools.partial(channel.basic_ack, delivery_tag=...))
```

NOTE: if you know that the requester is running on the same thread as the connection it is more efficient to use the *BlockingConnection.call_later()* method with a delay of 0.

Parameters **callback** (*callable*) – The callback method; must be callable

Raises ***pika.exceptions.ConnectionWrongStateError*** – if connection is closed

add_on_connection_blocked_callback(*callback*)

RabbitMQ AMQP extension - Add a callback to be notified when the connection gets blocked (*Connection.Blocked* received from RabbitMQ) due to the broker running low on resources (memory or disk). In this state RabbitMQ suspends processing incoming data until the connection is unblocked, so it's a good idea for publishers receiving this notification to suspend publishing until the connection becomes unblocked.

NOTE: due to the blocking nature of BlockingConnection, if it's sending outbound data while the connection is/becomes blocked, the call may remain blocked until the connection becomes unblocked, if ever. You may use *ConnectionParameters.blocked_connection_timeout* to abort a BlockingConnection method call with an exception when the connection remains blocked longer than the given timeout value.

See also *Connection.add_on_connection_unblocked_callback()*

See also *ConnectionParameters.blocked_connection_timeout*.

Parameters **callback** (*callable*) – Callback to call on *Connection.Blocked*, having the signature *callback(connection, pika.frame.Method)*, where connection is the *BlockingConnection* instance and the method frame's *method* member is of type *pika.spec.Connection.Blocked*

add_on_connection_unblocked_callback(*callback*)

RabbitMQ AMQP extension - Add a callback to be notified when the connection gets unblocked (*Connection.Unblocked* frame is received from RabbitMQ) letting publishers know it's ok to start publishing again.

Parameters **callback** (*callable*) – Callback to call on Connection.Unblocked‘, having the signature *callback(connection, pika.frame.Method)*, where connection is the *BlockingConnection* instance and the method

frame's *method* member is of type *pika.spec.Connection.Unblocked*

basic_nack
Specifies if the server supports basic.nack on the active connection.

Return type bool

basic_nack_supported
Specifies if the server supports basic.nack on the active connection.

Return type bool

call_later(*delay*, *callback*)
Create a single-shot timer to fire after delay seconds. Do not confuse with Tornado's timeout where you pass in the time you want to have your callback called. Only pass in the seconds until it's to be called.

NOTE: the timer callbacks are dispatched only in the scope of specially-designated methods: see *BlockingConnection.process_data_events()* and *BlockingChannel.start_consuming()*.

Parameters

- **delay** (*float*) – The number of seconds to wait to call callback
- **callback** (*callable*) – The callback method with the signature callback()

Returns Opaque timer id

Return type int

channel(*channel_number=None*)
Create a new channel with the next available channel number or pass in a channel number to use. Must be non-zero if you would like to specify but it is recommended that you let Pika manage the channel numbers.

Return type *pika.adapters.blocking_connection.BlockingChannel*

close(*reply_code=200*, *reply_text='Normal shutdown'*)
Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

Raises ***pika.exceptions.ConnectionWrongStateError*** – if called on a closed connection (NEW in v1.0.0)

consumer_cancel_notify
Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

consumer_cancel_notify_supported
Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

exchange_exchange_bindings
Specifies if the active connection supports exchange to exchange bindings.

Return type bool

exchange_exchange_bindings_supported
Specifies if the active connection supports exchange to exchange bindings.

Return type bool

is_closed
Returns a boolean reporting the current connection state.

is_open
Returns a boolean reporting the current connection state.

process_data_events(*time_limit=0*)
Will make sure that data events are processed. Dispatches timer and channel callbacks if not called from the scope of BlockingConnection or BlockingChannel callback. Your app can block on this method.

Parameters **time_limit** (*float*) – suggested upper bound on processing time in seconds. The actual blocking time depends on the granularity of the underlying ioloop. Zero means return as soon as possible. None means there is no limit on processing time and the function will block until I/O produces actionable events. Defaults to 0 for backward compatibility. This parameter is NEW in pika 0.10.0.

publisher_confirms
Specifies if the active connection can use publisher confirmations.

Return type bool

publisher_confirms_supported
Specifies if the active connection can use publisher confirmations.

Return type bool

remove_timeout(*timeout_id*)
Remove a timer if it's still in the timeout stack

Parameters **timeout_id** – The opaque timer id to remove

sleep(*duration*)
A safer way to sleep than calling time.sleep() directly that would keep the adapter from ignoring frames sent from the broker. The connection will "sleep" or block the number of seconds specified in duration in small intervals.

Parameters **duration** (*float*) – The time to sleep in seconds

class pika.adapters.blocking_connection.**BlockingChannel**(*channel_impl*, *connection*)
The BlockingChannel implements blocking semantics for most things that one would use callback-passing-style for with the `Channel` class. In addition, the *BlockingChannel* class implements a generator that allows you to *consume messages* without using callbacks.

Example of creating a BlockingChannel:

```
import pika

# Create our connection object
connection = pika.BlockingConnection()

# The returned object will be a synchronous channel
channel = connection.channel()
```

add_on_cancel_callback(*callback*)
Pass a callback function that will be called when Basic.Cancel is sent by the broker. The callback function should receive a method frame parameter.

Parameters **callback** (*callable*) – a callable for handling broker's Basic.Cancel notification with the call signature: callback(method_frame) where method_frame is of type *pika.frame.Method* with method of type *spec.Basic.Cancel*

add_on_return_callback(*callback*)

Pass a callback function that will be called when a published message is rejected and returned by the server via *Basic.Return*.

Parameters **callback** (*callable*) – The method to call on callback with the signature callback(channel, method, properties, body), where

- channel: pika.Channel
- method: pika.spec.Basic.Return
- properties: pika.spec.BasicProperties
- body: bytes

basic_ack(*delivery_tag=0*, *multiple=False*)

Acknowledge one or more messages. When sent by the client, this method acknowledges one or more messages delivered via the Deliver or Get-Ok methods. When sent by server, this method acknowledges one or more messages published with the Publish method on a channel in confirm mode. The acknowledgement can be for a single message or a set of messages up to and including a specific message.

Parameters

- **delivery_tag** (*int*) – The server-assigned delivery tag
- **multiple** (*bool*) – If set to True, the delivery tag is treated as "up to and including", so that multiple messages can be acknowledged with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates acknowledgement of all outstanding messages.

basic_cancel(*consumer_tag*)

This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer. The client may receive an arbitrary number of messages in between sending the cancel method and receiving the cancel-ok reply.

NOTE: When cancelling an auto_ack=False consumer, this implementation automatically Nacks and suppresses any incoming messages that have not yet been dispatched to the consumer's callback. However, when cancelling a auto_ack=True consumer, this method will return any pending messages that arrived before broker confirmed the cancellation.

Parameters **consumer_tag** (*str*) – Identifier for the consumer; the result of passing a consumer_tag that was created on another channel is undefined (bad things will happen)

Returns

(NEW IN pika 0.10.0) empty sequence for a auto_ack=False consumer; for a auto_ack=True consumer, returns a (possibly empty) sequence of pending messages that arrived before broker confirmed the cancellation (this is done instead of via consumer's callback in order to prevent reentrancy/recursion. Each message is four-tuple: (channel, method, properties, body)

- channel: BlockingChannel
- method: spec.Basic.Deliver
- properties: spec.BasicProperties
- body: bytes

Return type list

basic_consume(*queue*, *on_message_callback*, *auto_ack=False*, *exclusive=False*, *consumer_tag=None*, *arguments=None*)

Sends the AMQP command Basic.Consume to the broker and binds messages for the consumer_tag to the

consumer callback. If you do not pass in a consumer_tag, one will be automatically generated for you. Returns the consumer tag.

NOTE: the consumer callbacks are dispatched only in the scope of specially-designated methods: see *BlockingConnection.process_data_events* and *BlockingChannel.start_consuming*.

For more information about Basic.Consume, see: http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.consume

Parameters

- **queue** (*str*) – The queue from which to consume
- **on_message_callback** (*callable*) – Required function for dispatching messages to user, having the signature: on_message_callback(channel, method, properties, body)
  - channel: BlockingChannel
  - method: spec.Basic.Deliver
  - properties: spec.BasicProperties
  - body: bytes
- **auto_ack** (*bool*) – if set to True, automatic acknowledgement mode will be used (see http://www.rabbitmq.com/confirms.html). This corresponds with the 'no_ack' parameter in the basic.consume AMQP 0.9.1 method
- **exclusive** (*bool*) – Don't allow other consumers on the queue
- **consumer_tag** (*str*) – You may specify your own consumer tag; if left empty, a consumer tag will be generated automatically
- **arguments** (*dict*) – Custom key/value pair arguments for the consumer

Returns consumer tag

Return type str

Raises ***pika.exceptions.DuplicateConsumerTag*** – if consumer with given consumer_tag is already present.

basic_get(*queue*, *auto_ack=False*)

Get a single message from the AMQP broker. Returns a sequence with the method frame, message properties, and body.

Parameters

- **queue** (*str*) – Name of queue from which to get a message
- **auto_ack** (*bool*) – Tell the broker to not expect a reply

Returns a three-tuple; (None, None, None) if the queue was empty; otherwise (method, properties, body); NOTE: body may be None

Return type (spec.Basic.GetOk|None, spec.BasicProperties|None, str|None)

basic_nack(*delivery_tag=0*, *multiple=False*, *requeue=True*)

This method allows a client to reject one or more incoming messages. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Parameters

- **delivery_tag** (*int*) – The server-assigned delivery tag

- **multiple** (*bool*) – If set to True, the delivery tag is treated as "up to and including", so that multiple messages can be acknowledged with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates acknowledgement of all outstanding messages.
- **requeue** (*bool*) – If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

basic_publish(*exchange*, *routing_key*, *body*, *properties=None*, *mandatory=False*)

Publish to the channel with the given exchange, routing key, and body.

For more information on basic_publish and what the parameters do, see:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.publish

NOTE: mandatory may be enabled even without delivery confirmation, but in the absence of delivery confirmation the synchronous implementation has no way to know how long to wait for the Basic.Return.

Parameters

- **exchange** (*str*) – The exchange to publish to
- **routing_key** (*str*) – The routing key to bind on
- **body** (*bytes*) – The message body; empty string if no body
- **properties** (pika.spec.BasicProperties) – message properties
- **mandatory** (*bool*) – The mandatory flag

Raises

- ***UnroutableError*** – raised when a message published in publisher-acknowledgments mode (see *BlockingChannel.confirm_delivery*) is returned via *Basic.Return* followed by *Basic.Ack*.
- ***NackError*** – raised when a message published in publisher-acknowledgements mode is Nack'ed by the broker. See *BlockingChannel.confirm_delivery*.

basic_qos(*prefetch_size=0*, *prefetch_count=0*, *global_qos=False*)

Specify quality of service. This method requests a specific quality of service. The QoS can be specified for the current channel or for all channels on the connection. The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

Parameters

- **prefetch_size** (*int*) – This field specifies the prefetch window size. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls into other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply. The prefetch-size is ignored if the no-ack option is set in the consumer.
- **prefetch_count** (*int*) – Specifies a prefetch window in terms of whole messages. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both prefetch windows (and those at the channel and connection level) allow it. The prefetch-count is ignored if the no-ack option is set in the consumer.
- **global_qos** (*bool*) – Should the QoS apply to all channels on the connection.

basic_recover(*requeue=False*)

This method asks the server to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method replaces the asynchronous Recover.

Parameters **requeue** (*bool*) – If False, the message will be redelivered to the original recipient. If True, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

basic_reject(*delivery_tag=0*, *requeue=True*)

Reject an incoming message. This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Parameters

- **delivery_tag** (*int*) – The server-assigned delivery tag
- **requeue** (*bool*) – If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

cancel()

Cancel the queue consumer created by *BlockingChannel.consume*, rejecting all pending ackable messages.

NOTE: If you're looking to cancel a consumer issued with BlockingChannel.basic_consume then you should call BlockingChannel.basic_cancel.

Returns The number of messages requeued by Basic.Nack. NEW in 0.10.0: returns 0

Return type int

channel_number

Channel number

close(*reply_code=0*, *reply_text='Normal shutdown'*)

Will invoke a clean shutdown of the channel with the AMQP Broker.

Parameters

- **reply_code** (*int*) – The reply code to close the channel with
- **reply_text** (*str*) – The reply text to close the channel with

confirm_delivery()

Turn on RabbitMQ-proprietary Confirm mode in the channel.

For more information see: https://www.rabbitmq.com/confirms.html

connection

The channel's BlockingConnection instance

consume(*queue*, *auto_ack=False*, *exclusive=False*, *arguments=None*, *inactivity_timeout=None*)

Blocking consumption of a queue instead of via a callback. This method is a generator that yields each message as a tuple of method, properties, and body. The active generator iterator terminates when the consumer is cancelled by client via *BlockingChannel.cancel()* or by broker.

Example:

```
for method, properties, body in channel.consume('queue'):
    print body
    channel.basic_ack(method.delivery_tag)
```

You should call *BlockingChannel.cancel()* when you escape out of the generator loop.

If you don't cancel this consumer, then next call on the same channel to *consume()* with the exact same (queue, auto_ack, exclusive) parameters will resume the existing consumer generator; however, calling with different parameters will result in an exception.

Parameters

- **queue** (*str*) – The queue name to consume
- **auto_ack** (*bool*) – Tell the broker to not expect a ack/nack response
- **exclusive** (*bool*) – Don't allow other consumers on the queue
- **arguments** (*dict*) – Custom key/value pair arguments for the consumer
- **inactivity_timeout** (*float*) – if a number is given (in seconds), will cause the method to yield (None, None, None) after the given period of inactivity; this permits for pseudo-regular maintenance activities to be carried out by the user while waiting for messages to arrive. If None is given (default), then the method blocks until the next event arrives. NOTE that timing granularity is limited by the timer resolution of the underlying implementation. NEW in pika 0.10.0.

Yields tuple(spec.Basic.Deliver, spec.BasicProperties, str or unicode)

Raises

- **ValueError** – if consumer-creation parameters don't match those of the existing queue consumer generator, if any. NEW in pika 0.10.0
- ***ChannelClosed*** – when this channel is closed by broker.

consumer_tags

Property method that returns a list of consumer tags for active consumers

Return type list

exchange_bind(*destination*, *source*, *routing_key=''*, *arguments=None*)

Bind an exchange to another exchange.

Parameters

- **destination** (*str*) – The destination exchange to bind
- **source** (*str*) – The source exchange to bind to
- **routing_key** (*str*) – The routing key to bind on
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

Returns Method frame from the Exchange.Bind-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Exchange.BindOk*

exchange_declare(*exchange*, *exchange_type=<ExchangeType.direct: 'direct'>*, *passive=False*, *durable=False*, *auto_delete=False*, *internal=False*, *arguments=None*)

This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected class.

If passive set, the server will reply with Declare-Ok if the exchange already exists with the same name, and raise an error if not and if the exchange does not already exist, the server MUST raise a channel exception with reply code 404 (not found).

Parameters

- **exchange** (*str*) – The exchange name consists of a non-empty sequence of these characters: letters, digits, hyphen, underscore, period, or colon.
- **exchange_type** (*str*) – The exchange type to use

- **passive** (*bool*) – Perform a declare or just check to see if it exists
- **durable** (*bool*) – Survive a reboot of RabbitMQ
- **auto_delete** (*bool*) – Remove when no more queues are bound to it
- **internal** (*bool*) – Can only be published to by other exchanges
- **arguments** (*dict*) – Custom key/value pair arguments for the exchange

Returns Method frame from the Exchange.Declare-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Exchange.DeclareOk*

exchange_delete(*exchange=None*, *if_unused=False*)

Delete the exchange.

Parameters

- **exchange** (*str*) – The exchange name
- **if_unused** (*bool*) – only delete if the exchange is unused

Returns Method frame from the Exchange.Delete-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Exchange.DeleteOk*

exchange_unbind(*destination=None*, *source=None*, *routing_key=''*, *arguments=None*)

Unbind an exchange from another exchange.

Parameters

- **destination** (*str*) – The destination exchange to unbind
- **source** (*str*) – The source exchange to unbind from
- **routing_key** (*str*) – The routing key to unbind
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

Returns Method frame from the Exchange.Unbind-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Exchange.UnbindOk*

flow(*active*)

Turn Channel flow control off and on.

NOTE: RabbitMQ doesn't support active=False; per https://www.rabbitmq.com/specification.html: "active=false is not supported by the server. Limiting prefetch with basic.qos provides much better control"

For more information, please reference:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#channel.flow

Parameters **active** (*bool*) – Turn flow on (True) or off (False)

Returns True if broker will start or continue sending; False if not

Return type bool

get_waiting_message_count()

Returns the number of messages that may be retrieved from the current queue consumer generator via *BlockingChannel.consume* without blocking. NEW in pika 0.10.0

Returns The number of waiting messages

Return type int

is_closed
Returns True if the channel is closed.

Return type bool

is_open
Returns True if the channel is open.

Return type bool

queue_bind(*queue*, *exchange*, *routing_key=None*, *arguments=None*)
Bind the queue to the specified exchange

Parameters

- **queue** (*str*) – The queue to bind to the exchange
- **exchange** (*str*) – The source exchange to bind to
- **routing_key** (*str*) – The routing key to bind on
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

Returns Method frame from the Queue.Bind-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Queue.BindOk*

queue_declare(*queue*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=False*, *arguments=None*)
Declare queue, create if needed. This method creates or checks a queue. When creating a new queue the client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue.

Use an empty string as the queue name for the broker to auto-generate one. Retrieve this auto-generated queue name from the returned *spec.Queue.DeclareOk* method frame.

Parameters

- **queue** (*str*) – The queue name; if empty string, the broker will create a unique queue name
- **passive** (*bool*) – Only check to see if the queue exists and raise *ChannelClosed* if it doesn't
- **durable** (*bool*) – Survive reboots of the broker
- **exclusive** (*bool*) – Only allow access by the current connection
- **auto_delete** (*bool*) – Delete after consumer cancels or disconnects
- **arguments** (*dict*) – Custom key/value arguments for the queue

Returns Method frame from the Queue.Declare-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Queue.DeclareOk*

queue_delete(*queue*, *if_unused=False*, *if_empty=False*)
Delete a queue from the broker.

Parameters

- **queue** (*str*) – The queue to delete
- **if_unused** (*bool*) – only delete if it's unused
- **if_empty** (*bool*) – only delete if the queue is empty

Returns Method frame from the Queue.Delete-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Queue.DeleteOk*

queue_purge(*queue*)

Purge all of the messages from the specified queue

Parameters **queue** (*str*) – The queue to purge

Returns Method frame from the Queue.Purge-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Queue.PurgeOk*

queue_unbind(*queue*, *exchange=None*, *routing_key=None*, *arguments=None*)

Unbind a queue from an exchange.

Parameters

- **queue** (*str*) – The queue to unbind from the exchange
- **exchange** (*str*) – The source exchange to bind from
- **routing_key** (*str*) – The routing key to unbind
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

Returns Method frame from the Queue.Unbind-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Queue.UnbindOk*

start_consuming()

Processes I/O events and dispatches timers and *basic_consume* callbacks until all consumers are cancelled.

NOTE: this blocking function may not be called from the scope of a pika callback, because dispatching *basic_consume* callbacks from this context would constitute recursion.

Raises

- ***pika.exceptions.ReentrancyError*** – if called from the scope of a *BlockingConnection* or *BlockingChannel* callback
- ***ChannelClosed*** – when this channel is closed by broker.

stop_consuming(*consumer_tag=None*)

Cancels all consumers, signalling the *start_consuming* loop to exit.

NOTE: pending non-ackable messages will be lost; pending ackable messages will be rejected.

tx_commit()

Commit a transaction.

Returns Method frame from the Tx.Commit-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Tx.CommitOk*

tx_rollback()

Rollback a transaction.

Returns Method frame from the Tx.Commit-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Tx.CommitOk*

tx_select()

Select standard transaction mode. This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

Returns Method frame from the Tx.Select-ok response

Return type *pika.frame.Method* having *method* attribute of type *spec.Tx.SelectOk*

Select Connection Adapter

A connection adapter that tries to use the best polling method for the platform pika is running on.

class pika.adapters.select_connection.**SelectConnection**(*parameters=None*, *on_open_callback=None*, *on_open_error_callback=None*, *on_close_callback=None*, *custom_ioloop=None*, *internal_connection_workflow=True*)

An asynchronous connection adapter that attempts to use the fastest event loop adapter for the given platform.

add_on_close_callback(*callback*)

Add a callback notification when the connection has closed. The callback will be passed the connection and an exception instance. The exception will either be an instance of *exceptions.ConnectionClosed* if a fully-open connection was closed by user or broker or exception of another type that describes the cause of connection closure/failure.

Parameters **callback** (*callable*) – Callback to call on close, having the signature: callback(pika.connection.Connection, exception)

add_on_connection_blocked_callback(*callback*)

RabbitMQ AMQP extension - Add a callback to be notified when the connection gets blocked (*Connection.Blocked* received from RabbitMQ) due to the broker running low on resources (memory or disk). In this state RabbitMQ suspends processing incoming data until the connection is unblocked, so it's a good idea for publishers receiving this notification to suspend publishing until the connection becomes unblocked.

See also *Connection.add_on_connection_unblocked_callback()*

See also *ConnectionParameters.blocked_connection_timeout*.

Parameters **callback** (*callable*) – Callback to call on *Connection.Blocked*, having the signature *callback(connection, pika.frame.Method)*, where the method frame's *method* member is of type *pika.spec.Connection.Blocked*

add_on_connection_unblocked_callback(*callback*)

RabbitMQ AMQP extension - Add a callback to be notified when the connection gets unblocked (*Connection.Unblocked* frame is received from RabbitMQ) letting publishers know it's ok to start publishing again.

Parameters **callback** (*callable*) – Callback to call on *Connection.Unblocked*, having the signature *callback(connection, pika.frame.Method)*, where the method frame's *method* member is of type *pika.spec.Connection.Unblocked*

add_on_open_callback(*callback*)

Add a callback notification when the connection has opened. The callback will be passed the connection instance as its only arg.

Parameters **callback** (*callable*) – Callback to call when open

add_on_open_error_callback(*callback*, *remove_default=True*)

Add a callback notification when the connection can not be opened.

The callback method should accept the connection instance that could not connect, and either a string or an exception as its second arg.

Parameters

- **callback** (*callable*) – Callback to call when can't connect, having the signature _(Connection, Exception)

- **remove_default** (*bool*) – Remove default exception raising callback

basic_nack

Specifies if the server supports basic.nack on the active connection.

Return type bool

channel(*channel_number=None*, *on_open_callback=None*)

Create a new channel with the next available channel number or pass in a channel number to use. Must be non-zero if you would like to specify but it is recommended that you let Pika manage the channel numbers.

Parameters

- **channel_number** (*int*) – The channel number to use, defaults to the next available.
- **on_open_callback** (*callable*) – The callback when the channel is opened. The callback will be invoked with the *Channel* instance as its only argument.

Return type *pika.channel.Channel*

close(*reply_code=200*, *reply_text='Normal shutdown'*)

Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

Raises ***pika.exceptions.ConnectionWrongStateError*** – if connection is closed or closing.

consumer_cancel_notify

Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

classmethod create_connection(*connection_configs*, *on_done*, *custom_ioloop=None*, *workflow=None*)

Implement **:py:classmethod:'pika.adapters.BaseConnection.create_connection()'**.

exchange_exchange_bindings

Specifies if the active connection supports exchange to exchange bindings.

Return type bool

ioloop

Returns the native I/O loop instance underlying async services selected by user or the default selected by the specialized connection adapter (e.g., Twisted reactor, *asyncio.SelectorEventLoop*, *select_connection.IOLoop*, etc.)

Return type object

is_closed

Returns a boolean reporting the current connection state.

is_closing

Returns True if connection is in the process of closing due to client-initiated *close* request, but closing is not yet complete.

is_open

Returns a boolean reporting the current connection state.

publisher_confirms
Specifies if the active connection can use publisher confirmations.

Return type bool

Tornado Connection Adapter

Use pika with the Tornado IOLoop

Be sure to check out the *asynchronous examples* including the Tornado specific *consumer* example.

class pika.adapters.tornado_connection.**TornadoConnection**(*parameters=None*, *on_open_callback=None*, *on_open_error_callback=None*, *on_close_callback=None*, *custom_ioloop=None*, *internal_connection_workflow=True*)

The TornadoConnection runs on the Tornado IOLoop.

add_on_close_callback(*callback*)
Add a callback notification when the connection has closed. The callback will be passed the connection and an exception instance. The exception will either be an instance of *exceptions.ConnectionClosed* if a fully-open connection was closed by user or broker or exception of another type that describes the cause of connection closure/failure.

Parameters **callback** (*callable*) – Callback to call on close, having the signature: callback(pika.connection.Connection, exception)

add_on_connection_blocked_callback(*callback*)
RabbitMQ AMQP extension - Add a callback to be notified when the connection gets blocked (*Connection.Blocked* received from RabbitMQ) due to the broker running low on resources (memory or disk). In this state RabbitMQ suspends processing incoming data until the connection is unblocked, so it's a good idea for publishers receiving this notification to suspend publishing until the connection becomes unblocked.

See also *Connection.add_on_connection_unblocked_callback()*

See also *ConnectionParameters.blocked_connection_timeout*.

Parameters **callback** (*callable*) – Callback to call on *Connection.Blocked*, having the signature *callback(connection, pika.frame.Method)*, where the method frame's *method* member is of type *pika.spec.Connection.Blocked*

add_on_connection_unblocked_callback(*callback*)
RabbitMQ AMQP extension - Add a callback to be notified when the connection gets unblocked (*Connection.Unblocked* frame is received from RabbitMQ) letting publishers know it's ok to start publishing again.

Parameters **callback** (*callable*) – Callback to call on *Connection.Unblocked*, having the signature *callback(connection, pika.frame.Method)*, where the method frame's *method* member is of type *pika.spec.Connection.Unblocked*

add_on_open_callback(*callback*)
Add a callback notification when the connection has opened. The callback will be passed the connection instance as its only arg.

Parameters **callback** (*callable*) – Callback to call when open

add_on_open_error_callback(*callback*, *remove_default=True*)

Add a callback notification when the connection can not be opened.

The callback method should accept the connection instance that could not connect, and either a string or an exception as its second arg.

Parameters

- **callback** (*callable*) – Callback to call when can't connect, having the signature _(Connection, Exception)
- **remove_default** (*bool*) – Remove default exception raising callback

basic_nack

Specifies if the server supports basic.nack on the active connection.

Return type bool

channel(*channel_number=None*, *on_open_callback=None*)

Create a new channel with the next available channel number or pass in a channel number to use. Must be non-zero if you would like to specify but it is recommended that you let Pika manage the channel numbers.

Parameters

- **channel_number** (*int*) – The channel number to use, defaults to the next available.
- **on_open_callback** (*callable*) – The callback when the channel is opened. The callback will be invoked with the *Channel* instance as its only argument.

Return type *pika.channel.Channel*

close(*reply_code=200*, *reply_text='Normal shutdown'*)

Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

Raises ***pika.exceptions.ConnectionWrongStateError*** – if connection is closed or closing.

consumer_cancel_notify

Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

classmethod create_connection(*connection_configs*, *on_done*, *custom_ioloop=None*, *workflow=None*)

Implement **:py:classmethod:'pika.adapters.BaseConnection.create_connection()'**.

exchange_exchange_bindings

Specifies if the active connection supports exchange to exchange bindings.

Return type bool

ioloop

Returns the native I/O loop instance underlying async services selected by user or the default selected by the specialized connection adapter (e.g., Twisted reactor, *asyncio.SelectorEventLoop*, *select_connection.IOLoop*, etc.)

Return type object

is_closed
Returns a boolean reporting the current connection state.

is_closing
Returns True if connection is in the process of closing due to client-initiated *close* request, but closing is not yet complete.

is_open
Returns a boolean reporting the current connection state.

publisher_confirms
Specifies if the active connection can use publisher confirmations.

Return type bool

Twisted Connection Adapter

Using Pika with a Twisted reactor.

The interfaces in this module are Deferred-based when possible. This means that the connection.channel() method and most of the channel methods return Deferreds instead of taking a callback argument and that basic_consume() returns a Twisted DeferredQueue where messages from the server will be stored. Refer to the docstrings for TwistedProtocolConnection.channel() and the TwistedChannel class for details.

class pika.adapters.twisted_connection.**TwistedProtocolConnection**(*parameters=None*, *custom_reactor=None*)

A Pika-specific implementation of a Twisted Protocol. Allows using Twisted's non-blocking connectTCP/connectSSL methods for connecting to the server.

TwistedProtocolConnection objects have a *ready* instance variable that's a Deferred which fires when the connection is ready to be used (the initial AMQP handshaking has been done). You *have* to wait for this Deferred to fire before requesting a channel.

Once the connection is ready, you will be able to use the *closed* instance variable: a Deferred which fires when the connection is closed.

Since it's Twisted handling connection establishing it does not accept connect callbacks, you have to implement that within Twisted. Also remember that the host, port and ssl values of the connection parameters are ignored because, yet again, it's Twisted who manages the connection.

channel(*channel_number=None*)
Create a new channel with the next available channel number or pass in a channel number to use. Must be non-zero if you would like to specify but it is recommended that you let Pika manage the channel numbers.

Parameters **channel_number** (*int*) – The channel number to use, defaults to the next available.

Returns a Deferred that fires with an instance of a wrapper around the Pika Channel class.

Return type Deferred

connectionLost(*reason=<twisted.python.failure.Failure twisted.internet.error.ConnectionDone: Connection was closed cleanly.>*)
Called when the connection is shut down.

Clear any circular references here, and any external references to this Protocol. The connection has been closed.

@type reason: L{twisted.python.failure.Failure}

connectionMade()
Called when a connection is made.

This may be considered the initializer of the protocol, because it is called when the connection is completed. For clients, this is called once the connection to the server has been established; for servers, this is called after an accept() call stops blocking and a socket has been received. If you need to send any greeting or initial message, do it here.

connectionReady()
This method will be called when the underlying connection is ready.

dataReceived(*data*)
Called whenever data is received.

Use this method to translate to a higher-level message. Usually, some callback will be made upon the receipt of each complete protocol message.

@param data: a string of indeterminate length. Please keep in mind that you will probably need to buffer some data, as partial (or multiple) protocol messages may be received! I recommend that unit tests for protocols call through to this method with differing chunk sizes, down to one byte at a time.

logPrefix()
Return a prefix matching the class name, to identify log messages related to this protocol instance.

makeConnection(*transport*)
Make a connection to a transport and a server.

This sets the 'transport' attribute of this Protocol, and calls the connectionMade() callback.

class pika.adapters.twisted_connection.**TwistedChannel**(*channel*)
A wrapper around Pika's Channel.

Channel methods that normally take a callback argument are wrapped to return a Deferred that fires with whatever would be passed to the callback. If the channel gets closed, all pending Deferreds are errbacked with a ChannelClosed exception. The returned Deferreds fire with whatever arguments the callback to the original method would receive.

Some methods like basic_consume and basic_get are wrapped in a special way, see their docstrings for details.

add_on_return_callback(*callback*)
Pass a callback function that will be called when a published message is rejected and returned by the server via *Basic.Return*.

Parameters **callback** (*callable*) – The method to call on callback with the message as only argument. The message is a named tuple with the following attributes: channel: this TwistedChannel method: pika.spec.Basic.Return properties: pika.spec.BasicProperties body: bytes

basic_ack(*delivery_tag=0*, *multiple=False*)
Acknowledge one or more messages. When sent by the client, this method acknowledges one or more messages delivered via the Deliver or Get-Ok methods. When sent by server, this method acknowledges one or more messages published with the Publish method on a channel in confirm mode. The acknowledgement can be for a single message or a set of messages up to and including a specific message.

Parameters

- **delivery_tag** (*integer*) – int/long The server-assigned delivery tag
- **multiple** (*bool*) – If set to True, the delivery tag is treated as "up to and including", so that multiple messages can be acknowledged with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates acknowledgement of all outstanding messages.

basic_cancel(*consumer_tag=''*)

This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer. The client may receive an arbitrary number of messages in between sending the cancel method and receiving the cancel-ok reply. It may also be sent from the server to the client in the event of the consumer being unexpectedly cancelled (i.e. cancelled for any reason other than the server receiving the corresponding basic.cancel from the client). This allows clients to be notified of the loss of consumers due to events such as queue deletion.

This method wraps `Channel.basic_cancel` and closes any deferred queue associated with that consumer.

Parameters **consumer_tag** (*str*) – Identifier for the consumer

Returns Deferred that fires on the Basic.CancelOk response

Return type Deferred

Raises **ValueError** –

basic_consume(*queue*, *auto_ack=False*, *exclusive=False*, *consumer_tag=None*, *arguments=None*)

Consume from a server queue.

Sends the AMQP 0-9-1 command Basic.Consume to the broker and binds messages for the consumer_tag to a `ClosableDeferredQueue`. If you do not pass in a consumer_tag, one will be automatically generated for you.

For more information on basic_consume, see: Tutorial 2 at http://www.rabbitmq.com/getstarted.html http://www.rabbitmq.com/confirms.html http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.consume

Parameters

- **queue** (*str*) – The queue to consume from. Use the empty string to specify the most recent server-named queue for this channel.
- **auto_ack** (*bool*) – if set to True, automatic acknowledgement mode will be used (see http://www.rabbitmq.com/confirms.html). This corresponds with the 'no_ack' parameter in the basic.consume AMQP 0.9.1 method
- **exclusive** (*bool*) – Don't allow other consumers on the queue
- **consumer_tag** (*str*) – Specify your own consumer tag
- **arguments** (*dict*) – Custom key/value pair arguments for the consumer

Returns

Deferred that fires with a tuple `(queue_object, consumer_tag)`. The Deferred will errback with an instance of `exceptions.ChannelClosed` if the call fails. The queue object is an instance of `ClosableDeferredQueue`, where data received from the queue will be stored. Clients should use its `get()` method to fetch an individual message, which will return a Deferred firing with a namedtuple whose attributes are:

- channel: this TwistedChannel
- method: pika.spec.Basic.Deliver
- properties: pika.spec.BasicProperties
- body: bytes

Return type Deferred

basic_get (*queue*, *auto_ack=False*)

Get a single message from the AMQP broker.

Will return If the queue is empty, it will return None. If you want to be notified of Basic.GetEmpty, use the Channel.add_callback method adding your Basic.GetEmpty callback which should expect only one parameter, frame. Due to implementation details, this cannot be called a second time until the callback is executed. For more information on basic_get and its parameters, see:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.get

This method wraps `Channel.basic_get`.

Parameters

- **queue** (*str*) – The queue from which to get a message. Use the empty string to specify the most recent server-named queue for this channel.
- **auto_ack** (*bool*) – Tell the broker to not expect a reply

Returns

Deferred that fires with a namedtuple whose attributes are:

- channel: this TwistedChannel
- method: pika.spec.Basic.GetOk
- properties: pika.spec.BasicProperties
- body: bytes

If the queue is empty, None will be returned.

Return type Deferred

Raises `pika.exceptions.DuplicateGetOkCallback` –

basic_nack (*delivery_tag=None*, *multiple=False*, *requeue=True*)

This method allows a client to reject one or more incoming messages. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Parameters

- **delivery_tag** (*integer*) – int/long The server-assigned delivery tag
- **multiple** (*bool*) – If set to True, the delivery tag is treated as "up to and including", so that multiple messages can be acknowledged with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates acknowledgement of all outstanding messages.
- **requeue** (*bool*) – If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

basic_publish (*exchange*, *routing_key*, *body*, *properties=None*, *mandatory=False*)

Publish to the channel with the given exchange, routing key and body.

This method wraps `Channel.basic_publish`, but makes sure the channel is not closed before publishing.

For more information on basic_publish and what the parameters do, see:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.publish

Parameters

- **exchange** (*str*) – The exchange to publish to

- **routing_key** (*str*) – The routing key to bind on
- **body** (*bytes*) – The message body
- **properties** (pika.spec.BasicProperties) – Basic.properties
- **mandatory** (*bool*) – The mandatory flag

Returns A Deferred that fires with the result of the channel's basic_publish.

Return type Deferred

Raises

- ***UnroutableError*** – raised when a message published in publisher-acknowledgments mode (see *BlockingChannel.confirm_delivery*) is returned via *Basic.Return* followed by *Basic.Ack*.
- ***NackError*** – raised when a message published in publisher-acknowledgements mode is Nack'ed by the broker. See *BlockingChannel.confirm_delivery*.

basic_qos (*prefetch_size=0*, *prefetch_count=0*, *global_qos=False*)

Specify quality of service. This method requests a specific quality of service. The QoS can be specified for the current channel or for all channels on the connection. The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

Parameters

- **prefetch_size** (*int*) – This field specifies the prefetch window size. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls into other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply. The prefetch-size is ignored by consumers who have enabled the no-ack option.
- **prefetch_count** (*int*) – Specifies a prefetch window in terms of whole messages. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both prefetch windows (and those at the channel and connection level) allow it. The prefetch-count is ignored by consumers who have enabled the no-ack option.
- **global_qos** (*bool*) – Should the QoS apply to all channels on the connection.

Returns Deferred that fires on the Basic.QosOk response

Return type Deferred

basic_recover (*requeue=False*)

This method asks the server to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method replaces the asynchronous Recover.

Parameters **requeue** (*bool*) – If False, the message will be redelivered to the original recipient. If True, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

Returns Deferred that fires on the Basic.RecoverOk response

Return type Deferred

basic_reject (*delivery_tag*, *requeue=True*)

Reject an incoming message. This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Parameters

- **delivery_tag** (*integer*) – int/long The server-assigned delivery tag

- **requeue** (*bool*) – If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

Raises TypeError

callback_deferred(*deferred*, *replies*)

Pass in a Deferred and a list replies from the RabbitMQ broker which you'd like the Deferred to be callbacked on with the frame as callback value.

Parameters

- **deferred** (*Deferred*) – The Deferred to callback
- **replies** (*list*) – The replies to callback on

close(*reply_code=0*, *reply_text='Normal shutdown'*)

Invoke a graceful shutdown of the channel with the AMQP Broker.

If channel is OPENING, transition to CLOSING and suppress the incoming Channel.OpenOk, if any.

Parameters

- **reply_code** (*int*) – The reason code to send to broker
- **reply_text** (*str*) – The reason text to send to broker

Raises ***ChannelWrongStateError*** – if channel is closed or closing

confirm_delivery()

Turn on Confirm mode in the channel. Pass in a callback to be notified by the Broker when a message has been confirmed as received or rejected (Basic.Ack, Basic.Nack) from the broker to the publisher.

For more information see: http://www.rabbitmq.com/confirms.html#publisher-confirms

Returns Deferred that fires on the Confirm.SelectOk response

Return type Deferred

exchange_bind(*destination*, *source*, *routing_key=''*, *arguments=None*)

Bind an exchange to another exchange.

Parameters

- **destination** (*str*) – The destination exchange to bind
- **source** (*str*) – The source exchange to bind to
- **routing_key** (*str*) – The routing key to bind on
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

Raises **ValueError** –

Returns Deferred that fires on the Exchange.BindOk response

Return type Deferred

exchange_declare(*exchange*, *exchange_type=<ExchangeType.direct: 'direct'>*, *passive=False*, *durable=False*, *auto_delete=False*, *internal=False*, *arguments=None*)

This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected class.

If passive set, the server will reply with Declare-Ok if the exchange already exists with the same name, and raise an error if not and if the exchange does not already exist, the server MUST raise a channel exception with reply code 404 (not found).

Parameters

- **exchange** (*str*) – The exchange name consists of a non-empty sequence of these characters: letters, digits, hyphen, underscore, period, or colon
- **exchange_type** (*str*) – The exchange type to use
- **passive** (*bool*) – Perform a declare or just check to see if it exists
- **durable** (*bool*) – Survive a reboot of RabbitMQ
- **auto_delete** (*bool*) – Remove when no more queues are bound to it
- **internal** (*bool*) – Can only be published to by other exchanges
- **arguments** (*dict*) – Custom key/value pair arguments for the exchange

Returns Deferred that fires on the Exchange.DeclareOk response

Return type Deferred

Raises **ValueError** –

exchange_delete(*exchange=None*, *if_unused=False*)

Delete the exchange.

Parameters

- **exchange** (*str*) – The exchange name
- **if_unused** (*bool*) – only delete if the exchange is unused

Returns Deferred that fires on the Exchange.DeleteOk response

Return type Deferred

Raises **ValueError** –

exchange_unbind(*destination=None*, *source=None*, *routing_key=''*, *arguments=None*)

Unbind an exchange from another exchange.

Parameters

- **destination** (*str*) – The destination exchange to unbind
- **source** (*str*) – The source exchange to unbind from
- **routing_key** (*str*) – The routing key to unbind
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

Returns Deferred that fires on the Exchange.UnbindOk response

Return type Deferred

Raises **ValueError** –

flow(*active*)

Turn Channel flow control off and on.

Returns a Deferred that will fire with a bool indicating the channel flow state. For more information, please reference:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#channel.flow

Parameters **active** (*bool*) – Turn flow on or off

Returns Deferred that fires with the channel flow state

Return type Deferred

Raises **ValueError** –

is_closed
Returns True if the channel is closed.

Return type bool

is_closing
Returns True if client-initiated closing of the channel is in progress.

Return type bool

is_open
Returns True if the channel is open.

Return type bool

open()
Open the channel

queue_bind(*queue*, *exchange*, *routing_key=None*, *arguments=None*)
Bind the queue to the specified exchange

Parameters

- **queue** (*str*) – The queue to bind to the exchange
- **exchange** (*str*) – The source exchange to bind to
- **routing_key** (*str*) – The routing key to bind on
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

Returns Deferred that fires on the Queue.BindOk response

Return type Deferred

Raises **ValueError** –

queue_declare(*queue*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=False*, *arguments=None*)
Declare queue, create if needed. This method creates or checks a queue. When creating a new queue the client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue.

Use an empty string as the queue name for the broker to auto-generate one

Parameters

- **queue** (*str*) – The queue name; if empty string, the broker will create a unique queue name
- **passive** (*bool*) – Only check to see if the queue exists
- **durable** (*bool*) – Survive reboots of the broker
- **exclusive** (*bool*) – Only allow access by the current connection
- **auto_delete** (*bool*) – Delete after consumer cancels or disconnects
- **arguments** (*dict*) – Custom key/value arguments for the queue

Returns Deferred that fires on the Queue.DeclareOk response

Return type Deferred

Raises **ValueError** –

queue_delete(*queue*, *if_unused=False*, *if_empty=False*)
Delete a queue from the broker.

This method wraps `Channel.queue_delete`, and removes the reference to the queue object after it gets deleted on the server.

Parameters

- **queue** (*str*) – The queue to delete
- **if_unused** (*bool*) – only delete if it's unused
- **if_empty** (*bool*) – only delete if the queue is empty

Returns Deferred that fires on the Queue.DeleteOk response

Return type Deferred

Raises **ValueError** –

queue_purge(*queue*)
Purge all of the messages from the specified queue

Parameters **queue** (*str*) – The queue to purge

Returns Deferred that fires on the Queue.PurgeOk response

Return type Deferred

Raises **ValueError** –

queue_unbind(*queue*, *exchange=None*, *routing_key=None*, *arguments=None*)
Unbind a queue from an exchange.

Parameters

- **queue** (*str*) – The queue to unbind from the exchange
- **exchange** (*str*) – The source exchange to bind from
- **routing_key** (*str*) – The routing key to unbind
- **arguments** (*dict*) – Custom key/value pair arguments for the binding

Returns Deferred that fires on the Queue.UnbindOk response

Return type Deferred

Raises **ValueError** –

tx_commit()
Commit a transaction.

Returns Deferred that fires on the Tx.CommitOk response

Return type Deferred

Raises **ValueError** –

tx_rollback()
Rollback a transaction.

Returns Deferred that fires on the Tx.RollbackOk response

Return type Deferred

Raises **ValueError** –

tx_select()
Select standard transaction mode. This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

Returns Deferred that fires on the Tx.SelectOk response

Return type Deferred

Raises `ValueError` –

class pika.adapters.twisted_connection.**ClosableDeferredQueue**(*size=None*, *backlog=None*)
Like the normal Twisted DeferredQueue, but after close() is called with an exception instance all pending Deferreds are errbacked and further attempts to call get() or put() return a Failure wrapping that exception.

close(*reason*)
Closes the queue.

Errback the pending calls to `get()`.

get()
Returns a Deferred that will fire with the next item in the queue, when it's available.

The Deferred will errback if the queue is closed.

Returns Deferred that fires with the next item.

Return type Deferred

put(*obj*)
Like the original `DeferredQueue.put()` method, but returns an errback if the queue is closed.

2.2.2 Channel

The Channel class provides a wrapper for interacting with RabbitMQ implementing the methods and behaviors for an AMQP Channel.

Channel

class pika.channel.**Channel**(*connection*, *channel_number*, *on_open_callback*)
A Channel is the primary communication method for interacting with RabbitMQ. It is recommended that you do not directly invoke the creation of a channel object in your application code but rather construct a channel by calling the active connection's channel() method.

add_callback(*callback*, *replies*, *one_shot=True*)
Pass in a callback handler and a list replies from the RabbitMQ broker which you'd like the callback notified of. Callbacks should allow for the frame parameter to be passed in.

Parameters

- **callback** (*callable*) – The callback to call
- **replies** (*list*) – The replies to get a callback for
- **one_shot** (*bool*) – Only handle the first type callback

add_on_cancel_callback(*callback*)
Pass a callback function that will be called when the basic_cancel is sent by the server. The callback function should receive a frame parameter.

Parameters **callback** (*callable*) – The callback to call on Basic.Cancel from broker

add_on_close_callback(*callback*)

Pass a callback function that will be called when the channel is closed. The callback function will receive the channel and an exception describing why the channel was closed.

If the channel is closed by broker via Channel.Close, the callback will receive *ChannelClosedByBroker* as the reason.

If graceful user-initiated channel closing completes successfully (either directly of indirectly by closing a connection containing the channel) and closing concludes gracefully without Channel.Close from the broker and without loss of connection, the callback will receive *ChannelClosedByClient* exception as reason.

If channel was closed due to loss of connection, the callback will receive another exception type describing the failure.

Parameters **callback** (*callable*) – The callback, having the signature: callback(Channel, Exception reason)

add_on_flow_callback(*callback*)

Pass a callback function that will be called when Channel.Flow is called by the remote server. Note that newer versions of RabbitMQ will not issue this but instead use TCP backpressure

Parameters **callback** (*callable*) – The callback function

add_on_return_callback(*callback*)

Pass a callback function that will be called when basic_publish is sent a message that has been rejected and returned by the server.

Parameters **callback** (*callable*) – The function to call, having the signature callback(channel, method, properties, body) where

- channel: pika.channel.Channel
- method: pika.spec.Basic.Return
- properties: pika.spec.BasicProperties
- body: bytes

basic_ack(*delivery_tag=0*, *multiple=False*)

Acknowledge one or more messages. When sent by the client, this method acknowledges one or more messages delivered via the Deliver or Get-Ok methods. When sent by server, this method acknowledges one or more messages published with the Publish method on a channel in confirm mode. The acknowledgement can be for a single message or a set of messages up to and including a specific message.

Parameters

- **delivery_tag** (*integer*) – int/long The server-assigned delivery tag
- **multiple** (*bool*) – If set to True, the delivery tag is treated as "up to and including", so that multiple messages can be acknowledged with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates acknowledgement of all outstanding messages.

basic_cancel(*consumer_tag=''*, *callback=None*)

This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer. The client may receive an arbitrary number of messages in between sending the cancel method and receiving the cancel-ok reply. It may also be sent from the server to the client in the event of the consumer being unexpectedly cancelled (i.e. cancelled for any reason other than the server receiving the corresponding basic.cancel from the client). This allows clients to be notified of the loss of consumers due to events such as queue deletion.

Parameters

- **consumer_tag** (*str*) – Identifier for the consumer
- **callback** (*callable*) – callback(pika.frame.Method) for method Basic.CancelOk. If None, do not expect a Basic.CancelOk response, otherwise, callback must be callable

Raises **ValueError** –

basic_consume(*queue*, *on_message_callback*, *auto_ack=False*, *exclusive=False*, *consumer_tag=None*, *arguments=None*, *callback=None*)

Sends the AMQP 0-9-1 command Basic.Consume to the broker and binds messages for the consumer_tag to the consumer callback. If you do not pass in a consumer_tag, one will be automatically generated for you. Returns the consumer tag.

For more information on basic_consume, see: Tutorial 2 at http://www.rabbitmq.com/getstarted.html http://www.rabbitmq.com/confirms.html http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.consume

Parameters

- **queue** (*str*) – The queue to consume from. Use the empty string to specify the most recent server-named queue for this channel
- **on_message_callback** (*callable*) – The function to call when consuming with the signature on_message_callback(channel, method, properties, body), where
  - channel: pika.channel.Channel
  - method: pika.spec.Basic.Deliver
  - properties: pika.spec.BasicProperties
  - body: bytes
- **auto_ack** (*bool*) – if set to True, automatic acknowledgement mode will be used (see http://www.rabbitmq.com/confirms.html). This corresponds with the 'no_ack' parameter in the basic.consume AMQP 0.9.1 method
- **exclusive** (*bool*) – Don't allow other consumers on the queue
- **consumer_tag** (*str*) – Specify your own consumer tag
- **arguments** (*dict*) – Custom key/value pair arguments for the consumer
- **callback** (*callable*) – callback(pika.frame.Method) for method Basic.ConsumeOk.

Returns Consumer tag which may be used to cancel the consumer.

Return type str

Raises **ValueError** –

basic_get(*queue*, *callback*, *auto_ack=False*)

Get a single message from the AMQP broker. If you want to be notified of Basic.GetEmpty, use the Channel.add_callback method adding your Basic.GetEmpty callback which should expect only one parameter, frame. Due to implementation details, this cannot be called a second time until the callback is executed. For more information on basic_get and its parameters, see:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.get

Parameters

- **queue** (*str*) – The queue from which to get a message. Use the empty string to specify the most recent server-named queue for this channel
- **callback** (*callable*) – The callback to call with a message that has the signature callback(channel, method, properties, body), where:

- channel: pika.channel.Channel
- method: pika.spec.Basic.GetOk
- properties: pika.spec.BasicProperties
- body: bytes

- **auto_ack** (*bool*) – Tell the broker to not expect a reply

Raises **ValueError** –

basic_nack(*delivery_tag=0*, *multiple=False*, *requeue=True*)

This method allows a client to reject one or more incoming messages. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Parameters

- **delivery_tag** (*integer*) – int/long The server-assigned delivery tag
- **multiple** (*bool*) – If set to True, the delivery tag is treated as "up to and including", so that multiple messages can be acknowledged with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates acknowledgement of all outstanding messages.
- **requeue** (*bool*) – If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

basic_publish(*exchange*, *routing_key*, *body*, *properties=None*, *mandatory=False*)

Publish to the channel with the given exchange, routing key and body. For more information on basic_publish and what the parameters do, see:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#basic.publish

Parameters

- **exchange** (*str*) – The exchange to publish to
- **routing_key** (*str*) – The routing key to bind on
- **body** (*bytes*) – The message body
- **properties** (pika.spec.BasicProperties) – Basic.properties
- **mandatory** (*bool*) – The mandatory flag

basic_qos(*prefetch_size=0*, *prefetch_count=0*, *global_qos=False*, *callback=None*)

Specify quality of service. This method requests a specific quality of service. The QoS can be specified for the current channel or for all channels on the connection. The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

Parameters

- **prefetch_size** (*int*) – This field specifies the prefetch window size. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls into other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply. The prefetch-size is ignored by consumers who have enabled the no-ack option.
- **prefetch_count** (*int*) – Specifies a prefetch window in terms of whole messages. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both prefetch windows (and those at the channel and connection level) allow it. The prefetch-count is ignored by consumers who have enabled the no-ack option.

- **global_qos** (*bool*) – Should the QoS apply to all channels on the connection.
- **callback** (*callable*) – The callback to call for Basic.QosOk response

Raises **ValueError** –

basic_reject(*delivery_tag=0*, *requeue=True*)

Reject an incoming message. This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Parameters

- **delivery_tag** (*integer*) – int/long The server-assigned delivery tag
- **requeue** (*bool*) – If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

Raises TypeError

basic_recover(*requeue=False*, *callback=None*)

This method asks the server to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method replaces the asynchronous Recover.

Parameters

- **requeue** (*bool*) – If False, the message will be redelivered to the original recipient. If True, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.
- **callback** (*callable*) – Callback to call when receiving Basic.RecoverOk
- **callback** – callback(pika.frame.Method) for method Basic.RecoverOk

Raises **ValueError** –

close(*reply_code=0*, *reply_text='Normal shutdown'*)

Invoke a graceful shutdown of the channel with the AMQP Broker.

If channel is OPENING, transition to CLOSING and suppress the incoming Channel.OpenOk, if any.

Parameters

- **reply_code** (*int*) – The reason code to send to broker
- **reply_text** (*str*) – The reason text to send to broker

Raises ***ChannelWrongStateError*** – if channel is closed or closing

confirm_delivery(*ack_nack_callback*, *callback=None*)

Turn on Confirm mode in the channel. Pass in a callback to be notified by the Broker when a message has been confirmed as received or rejected (Basic.Ack, Basic.Nack) from the broker to the publisher.

For more information see: https://www.rabbitmq.com/confirms.html

Parameters

- **ack_nack_callback** (*callable*) – Required callback for delivery confirmations that has the following signature: callback(pika.frame.Method), where method_frame contains either method *spec.Basic.Ack* or *spec.Basic.Nack*.
- **callback** (*callable*) – callback(pika.frame.Method) for method Confirm.SelectOk

Raises **ValueError** –

consumer_tags

Property method that returns a list of currently active consumers

Return type list

exchange_bind(*destination*, *source*, *routing_key=''*, *arguments=None*, *callback=None*)

Bind an exchange to another exchange.

Parameters

- **destination** (*str*) – The destination exchange to bind
- **source** (*str*) – The source exchange to bind to
- **routing_key** (*str*) – The routing key to bind on
- **arguments** (*dict*) – Custom key/value pair arguments for the binding
- **callback** (*callable*) – callback(pika.frame.Method) for method Exchange.BindOk

Raises **ValueError** –

exchange_declare(*exchange*, *exchange_type=<ExchangeType.direct: 'direct'>*, *passive=False*, *durable=False*, *auto_delete=False*, *internal=False*, *arguments=None*, *callback=None*)

This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected class.

If passive set, the server will reply with Declare-Ok if the exchange already exists with the same name, and raise an error if not and if the exchange does not already exist, the server MUST raise a channel exception with reply code 404 (not found).

Parameters

- **exchange** (*str*) – The exchange name consists of a non-empty sequence of these characters: letters, digits, hyphen, underscore, period, or colon
- **exchange_type** (*str*) – The exchange type to use
- **passive** (*bool*) – Perform a declare or just check to see if it exists
- **durable** (*bool*) – Survive a reboot of RabbitMQ
- **auto_delete** (*bool*) – Remove when no more queues are bound to it
- **internal** (*bool*) – Can only be published to by other exchanges
- **arguments** (*dict*) – Custom key/value pair arguments for the exchange
- **callback** (*callable*) – callback(pika.frame.Method) for method Exchange.DeclareOk

Raises **ValueError** –

exchange_delete(*exchange=None*, *if_unused=False*, *callback=None*)

Delete the exchange.

Parameters

- **exchange** (*str*) – The exchange name
- **if_unused** (*bool*) – only delete if the exchange is unused
- **callback** (*callable*) – callback(pika.frame.Method) for method Exchange.DeleteOk

Raises **ValueError** –

exchange_unbind(*destination=None*, *source=None*, *routing_key=''*, *arguments=None*, *callback=None*)

Unbind an exchange from another exchange.

Parameters

- **destination** (*str*) – The destination exchange to unbind
- **source** (*str*) – The source exchange to unbind from
- **routing_key** (*str*) – The routing key to unbind
- **arguments** (*dict*) – Custom key/value pair arguments for the binding
- **callback** (*callable*) – callback(pika.frame.Method) for method Exchange.UnbindOk

Raises **ValueError** –

flow(*active*, *callback=None*)

Turn Channel flow control off and on. Pass a callback to be notified of the response from the server. active is a bool. Callback should expect a bool in response indicating channel flow state. For more information, please reference:

http://www.rabbitmq.com/amqp-0-9-1-reference.html#channel.flow

Parameters

- **active** (*bool*) – Turn flow on or off
- **callback** (*callable*) – callback(bool) upon completion

Raises **ValueError** –

is_closed

Returns True if the channel is closed.

Return type bool

is_closing

Returns True if client-initiated closing of the channel is in progress.

Return type bool

is_open

Returns True if the channel is open.

Return type bool

open()

Open the channel

queue_bind(*queue*, *exchange*, *routing_key=None*, *arguments=None*, *callback=None*)

Bind the queue to the specified exchange

Parameters

- **queue** (*str*) – The queue to bind to the exchange
- **exchange** (*str*) – The source exchange to bind to
- **routing_key** (*str*) – The routing key to bind on
- **arguments** (*dict*) – Custom key/value pair arguments for the binding
- **callback** (*callable*) – callback(pika.frame.Method) for method Queue.BindOk

Raises **ValueError** –

queue_declare(*queue*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=False*, *arguments=None*, *callback=None*)

Declare queue, create if needed. This method creates or checks a queue. When creating a new queue the

client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue.

Use an empty string as the queue name for the broker to auto-generate one

Parameters

- **queue** (*str*) – The queue name; if empty string, the broker will create a unique queue name
- **passive** (*bool*) – Only check to see if the queue exists
- **durable** (*bool*) – Survive reboots of the broker
- **exclusive** (*bool*) – Only allow access by the current connection
- **auto_delete** (*bool*) – Delete after consumer cancels or disconnects
- **arguments** (*dict*) – Custom key/value arguments for the queue
- **callback** (*callable*) – callback(pika.frame.Method) for method Queue.DeclareOk

Raises **ValueError** –

queue_delete(*queue*, *if_unused=False*, *if_empty=False*, *callback=None*)

Delete a queue from the broker.

Parameters

- **queue** (*str*) – The queue to delete
- **if_unused** (*bool*) – only delete if it's unused
- **if_empty** (*bool*) – only delete if the queue is empty
- **callback** (*callable*) – callback(pika.frame.Method) for method Queue.DeleteOk

Raises **ValueError** –

queue_purge(*queue*, *callback=None*)

Purge all of the messages from the specified queue

Parameters

- **queue** (*str*) – The queue to purge
- **callback** (*callable*) – callback(pika.frame.Method) for method Queue.PurgeOk

Raises **ValueError** –

queue_unbind(*queue*, *exchange=None*, *routing_key=None*, *arguments=None*, *callback=None*)

Unbind a queue from an exchange.

Parameters

- **queue** (*str*) – The queue to unbind from the exchange
- **exchange** (*str*) – The source exchange to bind from
- **routing_key** (*str*) – The routing key to unbind
- **arguments** (*dict*) – Custom key/value pair arguments for the binding
- **callback** (*callable*) – callback(pika.frame.Method) for method Queue.UnbindOk

Raises **ValueError** –

tx_commit(*callback=None*)

Commit a transaction

Parameters **callback** (*callable*) – The callback for delivery confirmations

Raises **ValueError** –

tx_rollback(*callback=None*)
Rollback a transaction.

Parameters **callback** (*callable*) – The callback for delivery confirmations

Raises **ValueError** –

tx_select(*callback=None*)
Select standard transaction mode. This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

Parameters **callback** (*callable*) – The callback for delivery confirmations

Raises **ValueError** –

2.2.3 Connection

The *Connection* class implements the base behavior that all connection adapters extend.

class pika.connection.**Connection**(*parameters=None*, *on_open_callback=None*, *on_open_error_callback=None*, *on_close_callback=None*, *internal_connection_workflow=True*)
This is the core class that implements communication with RabbitMQ. This class should not be invoked directly but rather through the use of an adapter such as SelectConnection or BlockingConnection.

add_on_close_callback(*callback*)
Add a callback notification when the connection has closed. The callback will be passed the connection and an exception instance. The exception will either be an instance of *exceptions.ConnectionClosed* if a fully-open connection was closed by user or broker or exception of another type that describes the cause of connection closure/failure.

Parameters **callback** (*callable*) – Callback to call on close, having the signature: callback(pika.connection.Connection, exception)

add_on_connection_blocked_callback(*callback*)
RabbitMQ AMQP extension - Add a callback to be notified when the connection gets blocked (*Connection.Blocked* received from RabbitMQ) due to the broker running low on resources (memory or disk). In this state RabbitMQ suspends processing incoming data until the connection is unblocked, so it's a good idea for publishers receiving this notification to suspend publishing until the connection becomes unblocked.

See also *Connection.add_on_connection_unblocked_callback()*

See also *ConnectionParameters.blocked_connection_timeout*.

Parameters **callback** (*callable*) – Callback to call on *Connection.Blocked*, having the signature *callback(connection, pika.frame.Method)*, where the method frame's *method* member is of type *pika.spec.Connection.Blocked*

add_on_connection_unblocked_callback(*callback*)
RabbitMQ AMQP extension - Add a callback to be notified when the connection gets unblocked (*Connection.Unblocked* frame is received from RabbitMQ) letting publishers know it's ok to start publishing again.

Parameters **callback** (*callable*) – Callback to call on *Connection.Unblocked*, having the signature *callback(connection, pika.frame.Method)*, where the method frame's *method* member is of type *pika.spec.Connection.Unblocked*

add_on_open_callback(*callback*)

Add a callback notification when the connection has opened. The callback will be passed the connection instance as its only arg.

Parameters **callback** (*callable*) – Callback to call when open

add_on_open_error_callback(*callback*, *remove_default=True*)

Add a callback notification when the connection can not be opened.

The callback method should accept the connection instance that could not connect, and either a string or an exception as its second arg.

Parameters

- **callback** (*callable*) – Callback to call when can't connect, having the signature _(Connection, Exception)
- **remove_default** (*bool*) – Remove default exception raising callback

basic_nack

Specifies if the server supports basic.nack on the active connection.

Return type bool

channel(*channel_number=None*, *on_open_callback=None*)

Create a new channel with the next available channel number or pass in a channel number to use. Must be non-zero if you would like to specify but it is recommended that you let Pika manage the channel numbers.

Parameters

- **channel_number** (*int*) – The channel number to use, defaults to the next available.
- **on_open_callback** (*callable*) – The callback when the channel is opened. The callback will be invoked with the *Channel* instance as its only argument.

Return type *pika.channel.Channel*

close(*reply_code=200*, *reply_text='Normal shutdown'*)

Disconnect from RabbitMQ. If there are any open channels, it will attempt to close them prior to fully disconnecting. Channels which have active consumers will attempt to send a Basic.Cancel to RabbitMQ to cleanly stop the delivery of messages prior to closing the channel.

Parameters

- **reply_code** (*int*) – The code number for the close
- **reply_text** (*str*) – The text reason for the close

Raises ***pika.exceptions.ConnectionWrongStateError*** – if connection is closed or closing.

consumer_cancel_notify

Specifies if the server supports consumer cancel notification on the active connection.

Return type bool

exchange_exchange_bindings

Specifies if the active connection supports exchange to exchange bindings.

Return type bool

is_closed

Returns a boolean reporting the current connection state.

is_closing
Returns True if connection is in the process of closing due to client-initiated *close* request, but closing is not yet complete.

is_open
Returns a boolean reporting the current connection state.

publisher_confirms
Specifies if the active connection can use publisher confirmations.

Return type bool

2.2.4 Authentication Credentials

The credentials classes are used to encapsulate all authentication information for the `ConnectionParameters` class.

The `PlainCredentials` class returns the properly formatted username and password to the `Connection`.

To authenticate with Pika, create a `PlainCredentials` object passing in the username and password and pass it as the credentials argument value to the `ConnectionParameters` object.

If you are using `URLParameters` you do not need a credentials object, one will automatically be created for you.

If you are looking to implement SSL certificate style authentication, you would extend the `ExternalCredentials` class implementing the required behavior.

PlainCredentials

class `pika.credentials.`**PlainCredentials**(*username*, *password*, *erase_on_connect=False*)
A credentials object for the default authentication methodology with RabbitMQ.

If you do not pass in credentials to the ConnectionParameters object, it will create credentials for 'guest' with the password of 'guest'.

If you pass True to erase_on_connect the credentials will not be stored in memory after the Connection attempt has been made.

Parameters

- **username** (*str*) – The username to authenticate with
- **password** (*str*) – The password to authenticate with
- **erase_on_connect** (*bool*) – erase credentials on connect.

erase_credentials()
Called by Connection when it no longer needs the credentials

response_for(*start*)
Validate that this type of authentication is supported

Parameters **start** (`spec.Connection.Start`) – Connection.Start method

Return type tuple(str|None, str|None)

ExternalCredentials

class pika.credentials.**ExternalCredentials**

The ExternalCredentials class allows the connection to use EXTERNAL authentication, generally with a client SSL certificate.

erase_credentials()

Called by Connection when it no longer needs the credentials

response_for(*start*)

Validate that this type of authentication is supported

Parameters **start** (spec.Connection.Start) – Connection.Start method

Return type tuple(str or None, str or None)

2.2.5 Exceptions

Pika specific exceptions

exception pika.exceptions.**AMQPChannelError**

exception pika.exceptions.**AMQPConnectionError**

exception pika.exceptions.**AMQPError**

exception pika.exceptions.**AMQPHeartbeatTimeout**

Connection was dropped as result of heartbeat timeout.

exception pika.exceptions.**AuthenticationError**

exception pika.exceptions.**BodyTooLongError**

exception pika.exceptions.**ChannelClosed**(*reply_code*, *reply_text*)

The channel closed by client or by broker

reply_code

NEW in v1.0.0 :rtype: int

reply_text

NEW in v1.0.0 :rtype: str

exception pika.exceptions.**ChannelClosedByBroker**(*reply_code*, *reply_text*)

Channel.Close from broker; may be passed as reason to channel's on-closed callback of non-blocking connection adapters or raised by *BlockingConnection*.

NEW in v1.0.0

exception pika.exceptions.**ChannelClosedByClient**(*reply_code*, *reply_text*)

Channel closed by client upon receipt of *Channel.CloseOk*; may be passed as reason to channel's on-closed callback of non-blocking connection adapters, but not raised by *BlockingConnection*.

NEW in v1.0.0

exception pika.exceptions.**ChannelError**

exception pika.exceptions.**ChannelWrongStateError**

Channel is in wrong state for the requested operation.

exception pika.exceptions.**ConnectionBlockedTimeout**

RabbitMQ-specific: timed out waiting for connection.unblocked.

exception pika.exceptions.**ConnectionClosed**(*reply_code*, *reply_text*)

reply_code
NEW in v1.0.0 :rtype: int

reply_text
NEW in v1.0.0 :rtype: str

exception pika.exceptions.**ConnectionClosedByBroker**(*reply_code*, *reply_text*)
Connection.Close from broker.

exception pika.exceptions.**ConnectionClosedByClient**(*reply_code*, *reply_text*)
Connection was closed at request of Pika client.

exception pika.exceptions.**ConnectionOpenAborted**
Client closed connection while opening.

exception pika.exceptions.**ConnectionWrongStateError**
Connection is in wrong state for the requested operation.

exception pika.exceptions.**ConsumerCancelled**

exception pika.exceptions.**DuplicateConsumerTag**

exception pika.exceptions.**DuplicateGetOkCallback**

exception pika.exceptions.**IncompatibleProtocolError**

exception pika.exceptions.**InvalidChannelNumber**

exception pika.exceptions.**InvalidFieldTypeException**

exception pika.exceptions.**InvalidFrameError**

exception pika.exceptions.**MethodNotImplemented**

exception pika.exceptions.**NackError**(*messages*)
This exception is raised when a message published in publisher-acknowledgements mode is Nack'ed by the broker.

Used by BlockingChannel.

exception pika.exceptions.**NoFreeChannels**

exception pika.exceptions.**ProbableAccessDeniedError**

exception pika.exceptions.**ProbableAuthenticationError**

exception pika.exceptions.**ProtocolSyntaxError**

exception pika.exceptions.**ProtocolVersionMismatch**

exception pika.exceptions.**ReentrancyError**
The requested operation would result in unsupported recursion or reentrancy.

Used by BlockingConnection/BlockingChannel

exception pika.exceptions.**ShortStringTooLong**

exception pika.exceptions.**StreamLostError**
Stream (TCP) connection lost.

exception pika.exceptions.**UnexpectedFrameError**

exception pika.exceptions.**UnroutableError**(*messages*)

Exception containing one or more unroutable messages returned by broker via Basic.Return.

Used by BlockingChannel.

In publisher-acknowledgements mode, this is raised upon receipt of Basic.Ack from broker; in the event of Basic.Nack from broker, *NackError* is raised instead

exception pika.exceptions.**UnsupportedAMQPFieldException**

2.2.6 Connection Parameters

To maintain flexibility in how you specify the connection information required for your applications to properly connect to RabbitMQ, pika implements two classes for encapsulating the information, `ConnectionParameters` and `URLParameters`.

ConnectionParameters

The classic object for specifying all of the connection parameters required to connect to RabbitMQ, `ConnectionParameters` provides attributes for tweaking every possible connection option.

Example:

```
import pika

# Set the connection parameters to connect to rabbit-server1 on port 5672
# on the / virtual host using the username "guest" and password "guest"
credentials = pika.PlainCredentials('guest', 'guest')
parameters = pika.ConnectionParameters('rabbit-server1',
                                       5672,
                                       '/',
                                       credentials)
```

class pika.connection.**ConnectionParameters**(*host=<class 'pika.connection.ConnectionParameters._DEFAULT'>, port=<class 'pika.connection.ConnectionParameters._DEFAULT'>, virtual_host=<class 'pika.connection.ConnectionParameters._DEFAULT'>, credentials=<class 'pika.connection.ConnectionParameters._DEFAULT'>, channel_max=<class 'pika.connection.ConnectionParameters._DEFAULT'>, frame_max=<class 'pika.connection.ConnectionParameters._DEFAULT'>, heartbeat=<class 'pika.connection.ConnectionParameters._DEFAULT'>, ssl_options=<class 'pika.connection.ConnectionParameters._DEFAULT'>, connection_attempts=<class 'pika.connection.ConnectionParameters._DEFAULT'>, retry_delay=<class 'pika.connection.ConnectionParameters._DEFAULT'>, socket_timeout=<class 'pika.connection.ConnectionParameters._DEFAULT'>, stack_timeout=<class 'pika.connection.ConnectionParameters._DEFAULT'>, locale=<class 'pika.connection.ConnectionParameters._DEFAULT'>, blocked_connection_timeout=<class 'pika.connection.ConnectionParameters._DEFAULT'>, client_properties=<class 'pika.connection.ConnectionParameters._DEFAULT'>, tcp_options=<class 'pika.connection.ConnectionParameters._DEFAULT'>, **kwargs*)

Connection parameters object that is passed into the connection adapter upon construction.

blocked_connection_timeout

Returns blocked connection timeout. Defaults to *DEFAULT_BLOCKED_CONNECTION_TIMEOUT*.

Return type float|None

channel_max

Returns max preferred number of channels. Defaults to *DEFAULT_CHANNEL_MAX*.

Return type int

client_properties

Returns client properties used to override the fields in the default client properties reported to RabbitMQ via *Connection.StartOk* method. Defaults to *DEFAULT_CLIENT_PROPERTIES*.

Return type dict|None

connection_attempts

Returns number of socket connection attempts. Defaults to *DEFAULT_CONNECTION_ATTEMPTS*. See also *retry_delay*.

Return type int

credentials

Return type one of the classes from *pika.credentials.VALID_TYPES*. Defaults to *DEFAULT_CREDENTIALS*.

frame_max

Returns desired maximum AMQP frame size to use. Defaults to *DEFAULT_FRAME_MAX*.

Return type int

heartbeat

Returns AMQP connection heartbeat timeout value for negotiation during connection tuning or callable which is invoked during connection tuning. None to accept broker's value. 0 turns heartbeat off. Defaults to *DEFAULT_HEARTBEAT_TIMEOUT*.

Return type int|callable|None

host

Returns hostname or ip address of broker. Defaults to *DEFAULT_HOST*.

Return type str

locale

Returns locale value to pass to broker; e.g., 'en_US'. Defaults to *DEFAULT_LOCALE*.

Return type str

retry_delay

Returns interval between socket connection attempts; see also *connection_attempts*. Defaults to *DEFAULT_RETRY_DELAY*.

Return type float

socket_timeout

Returns socket connect timeout in seconds. Defaults to *DEFAULT_SOCKET_TIMEOUT*. The value None disables this timeout.

Return type float|None

stack_timeout

Returns full protocol stack TCP/[SSL]/AMQP bring-up timeout in seconds. Defaults to *DEFAULT_STACK_TIMEOUT*. The value None disables this timeout.

Return type float

ssl_options

Returns None for plaintext or *pika.SSLOptions* instance for SSL/TLS.

Return type 'pika.SSLOptions'|None

port

Returns port number of broker's listening socket. Defaults to *DEFAULT_PORT*.

Return type int

virtual_host

Returns rabbitmq virtual host name. Defaults to *DEFAULT_VIRTUAL_HOST*.

Return type str

tcp_options

Returns None or a dict of options to pass to the underlying socket

Return type dict|None

URLParameters

The *URLParameters* class allows you to pass in an AMQP URL when creating the object and supports the host, port, virtual host, ssl, username and password in the base URL and other options are passed in via query parameters.

Example:

```
import pika

# Set the connection parameters to connect to rabbit-server1 on port 5672
# on the / virtual host using the username "guest" and password "guest"
parameters = pika.URLParameters('amqp://guest:guest@rabbit-server1:5672/%2F')
```

class pika.connection.**URLParameters**(*url*)

Connect to RabbitMQ via an AMQP URL in the format:

```
amqp://username:password@host:port/<virtual_host>[?query-string]
```

Ensure that the virtual host is URI encoded when specified. For example if you are using the default "/" virtual host, the value should be *%2f*.

See *Parameters* for default values.

Valid query string values are:

- **channel_max:** Override the default maximum channel count value
- **client_properties:** dict of client properties used to override the fields in the default client properties reported to RabbitMQ via *Connection.StartOk* method
- **connection_attempts:** Specify how many times pika should try and reconnect before it gives up
- **frame_max:** Override the default maximum frame size for communication
- **heartbeat:** Desired connection heartbeat timeout for negotiation. If not present the broker's value is accepted. 0 turns heartbeat off.
- **locale:** Override the default *en_US* locale value
- **ssl_options:** None for plaintext; for SSL: dict of public ssl context-related arguments that may be passed to `ssl.SSLSocket()` as kwargs, except *sock*, *server_side*,'do_handshake_on_connect', *family*, *type*, *proto*, *fileno*.
- **retry_delay:** The number of seconds to sleep before attempting to connect on connection failure.
- **socket_timeout:** Socket connect timeout value in seconds (float or int)
- **stack_timeout:** Positive full protocol stack (TCP/[SSL]/AMQP) bring-up timeout in seconds. It's recommended to set this value higher than *socket_timeout*.
- **blocked_connection_timeout:** Set the timeout, in seconds, that the connection may remain blocked (triggered by Connection.Blocked from broker); if the timeout expires before connection becomes unblocked, the connection will be torn down, triggering the connection's on_close_callback
- **tcp_options:** Set the tcp options for the underlying socket.

Parameters **url** (*str*) – The AMQP URL to connect to

ssl_options

Returns None for plaintext or *pika.SSLOptions* instance for SSL/TLS.

Return type ‘pika.SSLOptions‘|None

host

Returns hostname or ip address of broker. Defaults to *DEFAULT_HOST*.

Return type str

port

Returns port number of broker's listening socket. Defaults to *DEFAULT_PORT*.

Return type int

credentials

Return type one of the classes from *pika.credentials.VALID_TYPES*. Defaults to *DEFAULT_CREDENTIALS*.

virtual_host

Returns rabbitmq virtual host name. Defaults to *DEFAULT_VIRTUAL_HOST*.

Return type str

blocked_connection_timeout

Returns blocked connection timeout. Defaults to *DEFAULT_BLOCKED_CONNECTION_TIMEOUT*.

Return type float|None

channel_max

Returns max preferred number of channels. Defaults to *DEFAULT_CHANNEL_MAX*.

Return type int

client_properties

Returns client properties used to override the fields in the default client properties reported to RabbitMQ via *Connection.StartOk* method. Defaults to *DEFAULT_CLIENT_PROPERTIES*.

Return type dict|None

connection_attempts

Returns number of socket connection attempts. Defaults to *DEFAULT_CONNECTION_ATTEMPTS*. See also *retry_delay*.

Return type int

frame_max

Returns desired maximum AMQP frame size to use. Defaults to *DEFAULT_FRAME_MAX*.

Return type int

heartbeat

Returns AMQP connection heartbeat timeout value for negotiation during connection tuning or callable which is invoked during connection tuning. None to accept broker's value. 0 turns heartbeat off. Defaults to *DEFAULT_HEARTBEAT_TIMEOUT*.

Return type int|callable|None

locale

Returns locale value to pass to broker; e.g., 'en_US'. Defaults to *DEFAULT_LOCALE*.

Return type str

retry_delay

Returns interval between socket connection attempts; see also *connection_attempts*. Defaults to *DEFAULT_RETRY_DELAY*.

Return type float

socket_timeout

Returns socket connect timeout in seconds. Defaults to *DEFAULT_SOCKET_TIMEOUT*. The value None disables this timeout.

Return type float|None

stack_timeout

Returns full protocol stack TCP/[SSL]/AMQP bring-up timeout in seconds. Defaults to *DEFAULT_STACK_TIMEOUT*. The value None disables this timeout.

Return type float

tcp_options

Returns None or a dict of options to pass to the underlying socket

Return type dict|None

2.2.7 pika.spec

AMQP Specification

This module implements the constants and classes that comprise AMQP protocol level constructs. It should rarely be directly referenced outside of Pika's own internal use.

Note: Auto-generated code by codegen.py, do not edit directly. Pull

requests to this file without accompanying `utils/codegen.py` changes will be rejected.

class `pika.spec.`**Connection**

INDEX = 10

NAME = 'Connection'

class Start(*version_major=0*, *version_minor=9*, *server_properties=None*, *mechanisms='PLAIN'*, *locales='en_US'*)

INDEX = 655370

NAME = 'Connection.Start'

synchronous

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class StartOk(*client_properties=None*, *mechanism='PLAIN'*, *response=None*, *locale='en_US'*)

INDEX = 655371

NAME = 'Connection.StartOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Secure(*challenge=None*)

INDEX = 655380

NAME = 'Connection.Secure'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class SecureOk(*response=None*)

INDEX = 655381

NAME = 'Connection.SecureOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Tune(*channel_max=0*, *frame_max=0*, *heartbeat=0*)

INDEX = 655390

NAME = 'Connection.Tune'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class TuneOk(*channel_max=0*, *frame_max=0*, *heartbeat=0*)

INDEX = 655391

NAME = 'Connection.TuneOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.

Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Open(*virtual_host='/'*, *capabilities=''*, *insist=False*)

INDEX = 655400

NAME = 'Connection.Open'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class OpenOk(*known_hosts=''*)

INDEX = 655401

NAME = 'Connection.OpenOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Close(*reply_code=None*, *reply_text=''*, *class_id=None*, *method_id=None*)

INDEX = 655410

NAME = 'Connection.Close'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class CloseOk

INDEX = 655411

NAME = 'Connection.CloseOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Blocked(*reason=''*)

INDEX = 655420

NAME = 'Connection.Blocked'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Unblocked

INDEX = 655421

NAME = 'Connection.Unblocked'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Channel**

INDEX = 20

NAME = 'Channel'

class Open(*out_of_band=''*)

INDEX = 1310730

NAME = 'Channel.Open'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class OpenOk(*channel_id=''*)

INDEX = 1310731

NAME = 'Channel.OpenOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Flow(*active=None*)

INDEX = 1310740

NAME = 'Channel.Flow'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class FlowOk(*active=None*)

INDEX = 1310741

NAME = 'Channel.FlowOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Close(*reply_code=None*, *reply_text=''*, *class_id=None*, *method_id=None*)

INDEX = 1310760

NAME = 'Channel.Close'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class CloseOk

INDEX = 1310761

NAME = 'Channel.CloseOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Access**

INDEX = 30

NAME = 'Access'

class Request(*realm='/data'*, *exclusive=False*, *passive=True*, *active=True*, *write=True*, *read=True*)

INDEX = 1966090

NAME = 'Access.Request'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class RequestOk(*ticket=1*)

INDEX = 1966091

NAME = 'Access.RequestOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Exchange**

INDEX = 40

NAME = 'Exchange'

class Declare(*ticket=0*, *exchange=None*, *type=<ExchangeType.direct: 'direct'>*, *passive=False*, *durable=False*, *auto_delete=False*, *internal=False*, *nowait=False*, *arguments=None*)

INDEX = 2621450

NAME = 'Exchange.Declare'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class DeclareOk

INDEX = 2621451

NAME = 'Exchange.DeclareOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Delete(*ticket=0*, *exchange=None*, *if_unused=False*, *nowait=False*)

INDEX = 2621460

NAME = 'Exchange.Delete'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class DeleteOk

INDEX = 2621461

NAME = 'Exchange.DeleteOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Bind(*ticket=0*, *destination=None*, *source=None*, *routing_key=''*, *nowait=False*, *arguments=None*)

INDEX = 2621470

NAME = 'Exchange.Bind'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class BindOk

INDEX = 2621471

NAME = 'Exchange.BindOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.

Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Unbind(*ticket=0, destination=None, source=None, routing_key=", nowait=False, arguments=None*)

INDEX = 2621480

NAME = 'Exchange.Unbind'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded, offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class UnbindOk

INDEX = 2621491

NAME = 'Exchange.UnbindOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded, offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Queue**

INDEX = 50

NAME = 'Queue'

class Declare(*ticket=0, queue=", passive=False, durable=False, exclusive=False, auto_delete=False, nowait=False, arguments=None*)

INDEX = 3276810

NAME = 'Queue.Declare'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class DeclareOk(*queue=None*, *message_count=None*, *consumer_count=None*)

INDEX = 3276811

NAME = 'Queue.DeclareOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Bind(*ticket=0*, *queue=''*, *exchange=None*, *routing_key=''*, *nowait=False*, *arguments=None*)

INDEX = 3276820

NAME = 'Queue.Bind'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.

Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class BindOk

INDEX = 3276821

NAME = 'Queue.BindOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Purge(*ticket=0*, *queue=''*, *nowait=False*)

INDEX = 3276830

NAME = 'Queue.Purge'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class PurgeOk(*message_count=None*)

INDEX = 3276831

NAME = 'Queue.PurgeOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Delete(*ticket=0*, *queue=''*, *if_unused=False*, *if_empty=False*, *nowait=False*)

INDEX = 3276840

NAME = 'Queue.Delete'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class DeleteOk(*message_count=None*)

INDEX = 3276841

NAME = 'Queue.DeleteOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Unbind(*ticket=0*, *queue=''*, *exchange=None*, *routing_key=''*, *arguments=None*)

INDEX = 3276850

NAME = 'Queue.Unbind'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class UnbindOk

INDEX = 3276851

NAME = 'Queue.UnbindOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Basic**

INDEX = 60

NAME = 'Basic'

class Qos(*prefetch_size=0*, *prefetch_count=0*, *global_qos=False*)

INDEX = 3932170

NAME = 'Basic.Qos'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class QosOk

INDEX = 3932171

NAME = 'Basic.QosOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Consume(*ticket=0*, *queue=''*, *consumer_tag=''*, *no_local=False*, *no_ack=False*, *exclusive=False*, *nowait=False*, *arguments=None*)

INDEX = 3932180

NAME = 'Basic.Consume'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class ConsumeOk(*consumer_tag=None*)

INDEX = 3932181

NAME = 'Basic.ConsumeOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Cancel(*consumer_tag=None*, *nowait=False*)

INDEX = 3932190

NAME = 'Basic.Cancel'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class CancelOk(*consumer_tag=None*)

INDEX = 3932191

NAME = 'Basic.CancelOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Publish(*ticket=0*, *exchange=''*, *routing_key=''*, *mandatory=False*, *immediate=False*)

INDEX = 3932200

NAME = 'Basic.Publish'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Return(*reply_code=None*, *reply_text=''*, *exchange=None*, *routing_key=None*)

INDEX = 3932210

NAME = 'Basic.Return'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Deliver(*consumer_tag=None*, *delivery_tag=None*, *redelivered=False*, *exchange=None*, *routing_key=None*)

INDEX = 3932220

NAME = 'Basic.Deliver'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Get(*ticket=0*, *queue=''*, *no_ack=False*)

INDEX = 3932230

NAME = 'Basic.Get'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class GetOk(*delivery_tag=None*, *redelivered=False*, *exchange=None*, *routing_key=None*, *message_count=None*)

INDEX = 3932231

NAME = 'Basic.GetOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class GetEmpty(*cluster_id=”*)

INDEX = 3932232

NAME = 'Basic.GetEmpty'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Ack(*delivery_tag=0*, *multiple=False*)

INDEX = 3932240

NAME = 'Basic.Ack'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Reject(*delivery_tag=None*, *requeue=True*)

INDEX = 3932250

NAME = 'Basic.Reject'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class RecoverAsync(*requeue=False*)

INDEX = 3932260

NAME = 'Basic.RecoverAsync'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Recover(*requeue=False*)

INDEX = 3932270

NAME = 'Basic.Recover'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class RecoverOk

INDEX = 3932271

NAME = 'Basic.RecoverOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Nack(*delivery_tag=0*, *multiple=False*, *requeue=True*)

INDEX = 3932280

NAME = 'Basic.Nack'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Tx**

INDEX = 90

NAME = 'Tx'

class Select

INDEX = 5898250

NAME = 'Tx.Select'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class SelectOk

INDEX = 5898251

NAME = 'Tx.SelectOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Commit

INDEX = 5898260

NAME = 'Tx.Commit'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class CommitOk

INDEX = 5898261

NAME = 'Tx.CommitOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class Rollback

INDEX = 5898270

NAME = 'Tx.Rollback'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class RollbackOk

INDEX = 5898271

NAME = 'Tx.RollbackOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class pika.spec.**Confirm**

INDEX = 85

NAME = 'Confirm'

class Select(*nowait=False*)

INDEX = 5570570

NAME = 'Confirm.Select'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.
Return type str|unicode

get_properties()
Return the properties if they are set.
Return type pika.frame.Properties

class SelectOk

INDEX = 5570571

NAME = 'Confirm.SelectOk'

synchronous
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

decode(*encoded*, *offset=0*)

encode()

get_body()
Return the message body if it is set.

Return type str|unicode

get_properties()

Return the properties if they are set.

Return type pika.frame.Properties

class pika.spec.**BasicProperties**(*content_type=None*, *content_encoding=None*, *headers=None*, *delivery_mode=None*, *priority=None*, *correlation_id=None*, *reply_to=None*, *expiration=None*, *message_id=None*, *timestamp=None*, *type=None*, *user_id=None*, *app_id=None*, *cluster_id=None*)

CLASS

alias of *Basic*

INDEX = 60

NAME = 'BasicProperties'

FLAG_CONTENT_TYPE = 32768

FLAG_CONTENT_ENCODING = 16384

FLAG_HEADERS = 8192

FLAG_DELIVERY_MODE = 4096

FLAG_PRIORITY = 2048

FLAG_CORRELATION_ID = 1024

FLAG_REPLY_TO = 512

FLAG_EXPIRATION = 256

FLAG_MESSAGE_ID = 128

FLAG_TIMESTAMP = 64

FLAG_TYPE = 32

FLAG_USER_ID = 16

FLAG_APP_ID = 8

FLAG_CLUSTER_ID = 4

decode(*encoded*, *offset=0*)

encode()

pika.spec.**has_content**(*methodNumber*)

2.3 Usage Examples

Pika has various methods of use, between the synchronous BlockingConnection adapter and the various asynchronous connection adapter. The following examples illustrate the various ways that you can use Pika in your projects.

2.3.1 Using URLParameters

Pika has two methods of encapsulating the data that lets it know how to connect to RabbitMQ, *pika.connection.ConnectionParameters* and *pika.connection.URLParameters*.

Note: If you're connecting to RabbitMQ on localhost on port 5672, with the default virtual host of / and the default username and password of *guest* and *guest*, you do not need to specify connection parameters when connecting.

Using `pika.connection.URLParameters` is an easy way to minimize the variables required to connect to RabbitMQ and supports all of the directives that `pika.connection.ConnectionParameters` supports.

The following is the format for the URLParameters connection value:

```
scheme://username:password@host:port/virtual_host?key=value&key=value
```

As you can see, by default, the scheme (amqp, amqps), username, password, host, port and virtual host make up the core of the URL and any other parameter is passed in as query string values.

Example Connection URLS

The default connection URL connects to the / virtual host as guest using the guest password on localhost port 5672. Note the forwardslash in the URL is encoded to %2F:

```
amqp://guest:guest@localhost:5672/%2F
```

Connect to a host *rabbit1* as the user *www-data* using the password *rabbit_pwd* on the virtual host *web_messages*:

```
amqp://www-data:rabbit_pwd@rabbit1/web_messages
```

Connecting via SSL is pretty easy too. To connect via SSL for the previous example, simply change the scheme to *amqps*. If you do not specify a port, Pika will use the default SSL port of 5671:

```
amqps://www-data:rabbit_pwd@rabbit1/web_messages
```

If you're looking to tweak other parameters, such as enabling heartbeats, simply add the key/value pair as a query string value. The following builds upon the SSL connection, enabling heartbeats every 30 seconds:

```
amqps://www-data:rabbit_pwd@rabbit1/web_messages?heartbeat=30
```

Options that are available as query string values:

- backpressure_detection: Pass in a value of *t* to enable backpressure detection, it is disabled by default.
- channel_max: Alter the default channel maximum by passing in a 32-bit integer value here.
- connection_attempts: Alter the default of 1 connection attempt by passing in an integer value here.
- frame_max: Alter the default frame maximum size value by passing in a long integer value[1].
- heartbeat: Pass a value greater than zero to enable heartbeats between the server and your application. The integer value you pass here will be the number of seconds between heartbeats.
- locale: Set the locale of the client using underscore delimited posix Locale code in ll_CC format (en_US, pt_BR, de_DE).
- retry_delay: The number of seconds to wait before attempting to reconnect on a failed connection, if connection_attempts is > 0.
- socket_timeout: Change the default socket timeout duration from 0.25 seconds to another integer or float value. Adjust with caution.
- **ssl_options: A url encoded dict of values for the SSL connection. The available keys are:**

[1] The AMQP specification states that a server can reject a request for a frame size larger than the value it passes during content negotiation.

- ca_certs
- cert_reqs
- certfile
- keyfile
- ssl_version

For an information on what the ssl_options can be set to reference the official Python documentation. Here is an example of setting the client certificate and key:

```
amqp://www-data:rabbit_pwd@rabbit1/web_messages?heartbeat=30&ssl_options=%7B%27keyfile
↪%27%3A+%27%2Fetc%2Fssl%2Fmykey.pem%27%2C+%27certfile%27%3A+%27%2Fetc%2Fssl%2Fmycert.
↪pem%27%7D
```

The following example demonstrates how to generate the ssl_options string with Python's urllib:

```
import urllib
urllib.parse.urlencode({'ssl_options': {'certfile': '/etc/ssl/mycert.pem', 'keyfile':
↪'/etc/ssl/mykey.pem'}})
```

2.3.2 Connecting to RabbitMQ with Callback-Passing Style

When you connect to RabbitMQ with an asynchronous adapter, you are writing event oriented code. The connection adapter will block on the IOLoop that is watching to see when pika should read data from and write data to RabbitMQ. Because you're now blocking on the IOLoop, you will receive callback notifications when specific events happen.

Example Code

In the example, there are three steps that take place:

1. Setup the connection to RabbitMQ
2. Start the IOLoop
3. Once connected, the on_open method will be called by Pika with a handle to the connection. In this method, a new channel will be opened on the connection.
4. Once the channel is opened, you can do your other actions, whether they be publishing messages, consuming messages or other RabbitMQ related activities.:

```
import pika

# Step #3
def on_open(connection):
    connection.channel(on_open_callback=on_channel_open)

# Step #4
def on_channel_open(channel):
    channel.basic_publish('exchange_name',
                          'routing_key',
                          'Test Message',
                          pika.BasicProperties(content_type='text/plain',
                                               type='example'))

# Step #1: Connect to RabbitMQ
```

(continues on next page)

(continued from previous page)

```
connection = pika.SelectConnection(on_open_callback=on_open)

try:
    # Step #2 - Block on the IOLoop
    connection.ioloop.start()

# Catch a Keyboard Interrupt to make sure that the connection is closed cleanly
except KeyboardInterrupt:

    # Gracefully close the connection
    connection.close()

    # Start the IOLoop again so Pika can communicate, it will stop on its own␣
↪when the connection is closed
    connection.ioloop.start()
```

2.3.3 Using the Blocking Connection to get a message from RabbitMQ

The `BlockingChannel.basic_get` method will return a tuple with the members.

If the server returns a message, the first item in the tuple will be a `pika.spec.Basic.GetOk` object with the current message count, the redelivered flag, the routing key that was used to put the message in the queue, and the exchange the message was published to. The second item will be a `BasicProperties` object and the third will be the message body.

If the server did not return a message a tuple of None, None, None will be returned.

Example of getting a message and acknowledging it:

```
import pika

connection = pika.BlockingConnection()
channel = connection.channel()
method_frame, header_frame, body = channel.basic_get('test')
if method_frame:
    print(method_frame, header_frame, body)
    channel.basic_ack(method_frame.delivery_tag)
else:
    print('No message returned')
```

2.3.4 Using the Blocking Connection to consume messages from RabbitMQ

The `BlockingChannel.basic_consume` method assign a callback method to be called every time that RabbitMQ delivers messages to your consuming application.

When pika calls your method, it will pass in the channel, a `pika.spec.Basic.Deliver` object with the delivery tag, the redelivered flag, the routing key that was used to put the message in the queue, and the exchange the message was published to. The third argument will be a `pika.spec.BasicProperties` object and the last will be the message body.

Example of consuming messages and acknowledging them:

```
import pika
```

(continues on next page)

(continued from previous page)

```
def on_message(channel, method_frame, header_frame, body):
    print(method_frame.delivery_tag)
    print(body)
    print()
    channel.basic_ack(delivery_tag=method_frame.delivery_tag)


connection = pika.BlockingConnection()
channel = connection.channel()
channel.basic_consume('test', on_message)
try:
    channel.start_consuming()
except KeyboardInterrupt:
    channel.stop_consuming()
connection.close()
```

2.3.5 Using the Blocking Connection with connection recovery with multiple hosts

RabbitMQ nodes can be clustered. In the absence of failure clients can connect to any node and perform any operation. In case a node fails, stops, or becomes unavailable, clients should be able to connect to another node and continue.

To simplify reconnection to a different node, connection recovery mechanism should be combined with connection configuration that specifies multiple hosts.

The BlockingConnection adapter relies on exception handling to check for connection errors:

```
import pika
import random

def on_message(channel, method_frame, header_frame, body):
    print(method_frame.delivery_tag)
    print(body)
    print()
    channel.basic_ack(delivery_tag=method_frame.delivery_tag)

## Assuming there are three hosts: host1, host2, and host3
node1 = pika.URLParameters('amqp://node1')
node2 = pika.URLParameters('amqp://node2')
node3 = pika.URLParameters('amqp://node3')
all_endpoints = [node1, node2, node3]

while(True):
    try:
        print("Connecting...")
        ## Shuffle the hosts list before reconnecting.
        ## This can help balance connections.
        random.shuffle(all_endpoints)
        connection = pika.BlockingConnection(all_endpoints)
        channel = connection.channel()
        channel.basic_qos(prefetch_count=1)
        ## This queue is intentionally non-durable. See http://www.rabbitmq.com/ha.
↪html#non-mirrored-queue-behavior-on-node-failure
        ## to learn more.
        channel.queue_declare('recovery-example', durable = False, auto_delete = True)
```

(continues on next page)

(continued from previous page)

```
        channel.basic_consume('recovery-example', on_message)
        try:
            channel.start_consuming()
        except KeyboardInterrupt:
            channel.stop_consuming()
            connection.close()
            break
    except pika.exceptions.ConnectionClosedByBroker:
        # Uncomment this to make the example not attempt recovery
        # from server-initiated connection closure, including
        # when the node is stopped cleanly
        #
        # break
        continue
    # Do not recover on channel errors
    except pika.exceptions.AMQPChannelError as err:
        print("Caught a channel error: {}, stopping...".format(err))
        break
    # Recover on all other connection errors
    except pika.exceptions.AMQPConnectionError:
        print("Connection was closed, retrying...")
        continue
```

Generic operation retry libraries such as retry can prove useful.

To run the following example, install the library first with *pip install retry*.

In this example the *retry* decorator is used to set up recovery with delay:

```
import pika
import random
from retry import retry

def on_message(channel, method_frame, header_frame, body):
    print(method_frame.delivery_tag)
    print(body)
    print()
    channel.basic_ack(delivery_tag=method_frame.delivery_tag)

## Assuming there are three hosts: host1, host2, and host3
node1 = pika.URLParameters('amqp://node1')
node2 = pika.URLParameters('amqp://node2')
node3 = pika.URLParameters('amqp://node3')
all_endpoints = [node1, node2, node3]

@retry(pika.exceptions.AMQPConnectionError, delay=5, jitter=(1, 3))
def consume():
    random.shuffle(all_endpoints)
    connection = pika.BlockingConnection(all_endpoints)
    channel = connection.channel()
    channel.basic_qos(prefetch_count=1)

    ## This queue is intentionally non-durable. See http://www.rabbitmq.com/ha.html
↪#non-mirrored-queue-behavior-on-node-failure
    ## to learn more.
    channel.queue_declare('recovery-example', durable = False, auto_delete = True)
    channel.basic_consume('recovery-example', on_message)
```

(continues on next page)

(continued from previous page)

```
    try:
        channel.start_consuming()
    except KeyboardInterrupt:
        channel.stop_consuming()
        connection.close()
    except pika.exceptions.ConnectionClosedByBroker:
        # Uncomment this to make the example not attempt recovery
        # from server-initiated connection closure, including
        # when the node is stopped cleanly
        # except pika.exceptions.ConnectionClosedByBroker:
        #     pass
        continue

consume()
```

2.3.6 Using the BlockingChannel.consume generator to consume messages

The `BlockingChannel.consume` method is a generator that will return a tuple of method, properties and body.

When you escape out of the loop, be sure to call consumer.cancel() to return any unprocessed messages.

Example of consuming messages and acknowledging them:

```
import pika

connection = pika.BlockingConnection()
channel = connection.channel()

# Get ten messages and break out
for method_frame, properties, body in channel.consume('test'):

    # Display the message parts
    print(method_frame)
    print(properties)
    print(body)

    # Acknowledge the message
    channel.basic_ack(method_frame.delivery_tag)

    # Escape out of the loop after 10 messages
    if method_frame.delivery_tag == 10:
        break

# Cancel the consumer and return any pending messages
requeued_messages = channel.cancel()
print('Requeued %i messages' % requeued_messages)

# Close the channel and the connection
channel.close()
connection.close()
```

If you have pending messages in the test queue, your output should look something like:

```
(pika)gmr-0x02:pika gmr$ python blocking_nack.py
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=1', 'exchange=test'])>
```

(continues on next page)

(continued from previous page)

```
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=2', 'exchange=test'])>
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=3', 'exchange=test'])>
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=4', 'exchange=test'])>
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=5', 'exchange=test'])>
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=6', 'exchange=test'])>
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=7', 'exchange=test'])>
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=8', 'exchange=test'])>
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=9', 'exchange=test'])>
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
<Basic.Deliver(['consumer_tag=ctag1.0', 'redelivered=True', 'routing_key=test',
↪'delivery_tag=10', 'exchange=test'])>
<BasicProperties(['delivery_mode=1', 'content_type=text/plain'])>
Hello World!
Requeued 1894 messages
```

2.3.7 Comparing Message Publishing with BlockingConnection and SelectConnection

For those doing simple, non-asynchronous programming, `pika.adapters.blocking_connection.BlockingConnection()` proves to be the easiest way to get up and running with Pika to publish messages.

In the following example, a connection is made to RabbitMQ listening to port *5672* on *localhost* using the username *guest* and password *guest* and virtual host */*. Once connected, a channel is opened and a message is published to the *test_exchange* exchange using the *test_routing_key* routing key. The BasicProperties value passed in sets the message to delivery mode *1* (non-persisted) with a content-type of *text/plain*. Once the message is published, the connection is closed:

```
import pika

parameters = pika.URLParameters('amqp://guest:guest@localhost:5672/%2F')
```

(continues on next page)

(continued from previous page)

```
connection = pika.BlockingConnection(parameters)

channel = connection.channel()

channel.basic_publish('test_exchange',
                      'test_routing_key',
                      'message body value',
                      pika.BasicProperties(content_type='text/plain',
                                           delivery_mode=pika.DeliveryMode.Transient))

connection.close()
```

In contrast, using *pika.adapters.select_connection.SelectConnection()* and the other asynchronous adapters is more complicated and less pythonic, but when used with other asynchronous services can have tremendous performance improvements. In the following code example, all of the same parameters and values are used as were used in the previous example:

```
import pika

# Step #3
def on_open(connection):

    connection.channel(on_open_callback=on_channel_open)

# Step #4
def on_channel_open(channel):

    channel.basic_publish('test_exchange',
                          'test_routing_key',
                          'message body value',
                          pika.BasicProperties(content_type='text/plain',
                                               delivery_mode=pika.DeliveryMode.
↪Transient))

    connection.close()

# Step #1: Connect to RabbitMQ
parameters = pika.URLParameters('amqp://guest:guest@localhost:5672/%2F')

connection = pika.SelectConnection(parameters=parameters,
                                   on_open_callback=on_open)

try:

    # Step #2 - Block on the IOLoop
    connection.ioloop.start()

# Catch a Keyboard Interrupt to make sure that the connection is closed cleanly
except KeyboardInterrupt:

    # Gracefully close the connection
    connection.close()

    # Start the IOLoop again so Pika can communicate, it will stop on its own when␣
↪the connection is closed
    connection.ioloop.start()
```

2.3.8 Using Delivery Confirmations with the BlockingConnection

The following code demonstrates how to turn on delivery confirmations with the BlockingConnection and how to check for confirmation from RabbitMQ:

```
import pika

# Open a connection to RabbitMQ on localhost using all default parameters
connection = pika.BlockingConnection()

# Open the channel
channel = connection.channel()

# Declare the queue
channel.queue_declare(queue="test", durable=True, exclusive=False, auto_delete=False)

# Turn on delivery confirmations
channel.confirm_delivery()

# Send a message
try:
    channel.basic_publish(exchange='test',
                          routing_key='test',
                          body='Hello World!',
                          properties=pika.BasicProperties(content_type='text/plain',
                                                          delivery_mode=pika.
↪DeliveryMode.Transient)):
    print('Message publish was confirmed')
except pika.exceptions.UnroutableError:
    print('Message could not be confirmed')
```

2.3.9 Ensuring message delivery with the mandatory flag

The following example demonstrates how to check if a message is delivered by setting the mandatory flag and handling exceptions when using the BlockingConnection:

```
import pika
import pika.exceptions

# Open a connection to RabbitMQ on localhost using all default parameters
connection = pika.BlockingConnection()

# Open the channel
channel = connection.channel()

# Declare the queue
channel.queue_declare(queue="test", durable=True, exclusive=False, auto_delete=False)

# Enabled delivery confirmations. This is REQUIRED.
channel.confirm_delivery()

# Send a message
try:
    channel.basic_publish(exchange='test',
                          routing_key='test',
                          body='Hello World!',
```

(continues on next page)

(continued from previous page)

```
                            properties=pika.BasicProperties(content_type='text/plain',
                                                            delivery_mode=pika.
↪DeliveryMode.Transient),
                            mandatory=True)
    print('Message was published')
except pika.exceptions.UnroutableError:
    print('Message was returned')
```

2.3.10 Asynchronous consumer example

The following example implements a consumer that will respond to RPC commands sent from RabbitMQ. For example, it will reconnect if RabbitMQ closes the connection and will shutdown if RabbitMQ cancels the consumer or closes the channel. While it may look intimidating, each method is very short and represents a individual actions that a consumer can do.

Asynchronous Consumer Example

2.3.11 Asynchronous publisher example

The following example implements a publisher that will respond to RPC commands sent from RabbitMQ and uses delivery confirmations. It will reconnect if RabbitMQ closes the connection and will shutdown if RabbitMQ closes the channel. While it may look intimidating, each method is very short and represents a individual actions that a publisher can do.

Asynchronous Publisher Example

2.3.12 Twisted Consumer Example

Example of writing an application using the `Twisted connection adapter`::.

Twisted Example

2.3.13 Tornado Consumer

The following example implements a consumer using the `Tornado adapter` for the Tornado framework that will respond to RPC commands sent from RabbitMQ. For example, it will reconnect if RabbitMQ closes the connection and will shutdown if RabbitMQ cancels the consumer or closes the channel. While it may look intimidating, each method is very short and represents a individual actions that a consumer can do.

consumer.py:

```
import logging
import pika
from pika import adapters
from pika.adapters.tornado_connection import TornadoConnection
from pika.exchange_type import ExchangeType

LOG_FORMAT = ('%(levelname) -10s %(asctime)s %(name) -30s %(funcName) '
              '-35s %(lineno) -5d: %(message)s')
LOGGER = logging.getLogger(__name__)
```

(continues on next page)

(continued from previous page)

```
class ExampleConsumer(object):
    """This is an example consumer that will handle unexpected interactions
    with RabbitMQ such as channel and connection closures.

    If RabbitMQ closes the connection, it will reopen it. You should
    look at the output, as there are limited reasons why the connection may
    be closed, which usually are tied to permission related issues or
    socket timeouts.

    If the channel is closed, it will indicate a problem with one of the
    commands that were issued and that should surface in the output as well.

    """
    EXCHANGE = 'message'
    EXCHANGE_TYPE = ExchangeType.topic
    QUEUE = 'text'
    ROUTING_KEY = 'example.text'

    def __init__(self, amqp_url):
        """Create a new instance of the consumer class, passing in the AMQP
        URL used to connect to RabbitMQ.

        :param str amqp_url: The AMQP url to connect with

        """
        self._connection = None
        self._channel = None
        self._closing = False
        self._consumer_tag = None
        self._url = amqp_url

    def connect(self):
        """This method connects to RabbitMQ, returning the connection handle.
        When the connection is established, the on_connection_open method
        will be invoked by pika.

        :rtype: pika.SelectConnection

        """
        LOGGER.info('Connecting to %s', self._url)
        return TornadoConnection(
            pika.URLParameters(self._url),
            self.on_connection_open,
        )

    def close_connection(self):
        """This method closes the connection to RabbitMQ."""
        LOGGER.info('Closing connection')
        self._connection.close()

    def add_on_connection_close_callback(self):
        """This method adds an on close callback that will be invoked by pika
        when RabbitMQ closes the connection to the publisher unexpectedly.

        """
        LOGGER.info('Adding connection close callback')
        self._connection.add_on_close_callback(self.on_connection_closed)
```

(continues on next page)

(continued from previous page)

```
    def on_connection_closed(self, connection, reason):
        """This method is invoked by pika when the connection to RabbitMQ is
        closed unexpectedly. Since it is unexpected, we will reconnect to
        RabbitMQ if it disconnects.

        :param pika.connection.Connection connection: The closed connection obj
        :param Exception reason: exception representing reason for loss of
            connection.

        """
        self._channel = None
        if self._closing:
            self._connection.ioloop.stop()
        else:
            LOGGER.warning('Connection closed, reopening in 5 seconds: %s',
                           reason)
            self._connection.ioloop.call_later(5, self.reconnect)

    def on_connection_open(self, unused_connection):
        """This method is called by pika once the connection to RabbitMQ has
        been established. It passes the handle to the connection object in
        case we need it, but in this case, we'll just mark it unused.

        :param pika.SelectConnection _unused_connection: The connection

        """
        LOGGER.info('Connection opened')
        self.add_on_connection_close_callback()
        self.open_channel()

    def reconnect(self):
        """Will be invoked by the IOLoop timer if the connection is
        closed. See the on_connection_closed method.

        """
        if not self._closing:

            # Create a new connection
            self._connection = self.connect()

    def add_on_channel_close_callback(self):
        """This method tells pika to call the on_channel_closed method if
        RabbitMQ unexpectedly closes the channel.

        """
        LOGGER.info('Adding channel close callback')
        self._channel.add_on_close_callback(self.on_channel_closed)

    def on_channel_closed(self, channel, reason):
        """Invoked by pika when RabbitMQ unexpectedly closes the channel.
        Channels are usually closed if you attempt to do something that
        violates the protocol, such as re-declare an exchange or queue with
        different parameters. In this case, we'll close the connection
        to shutdown the object.

        :param pika.channel.Channel: The closed channel
```

(continues on next page)

(continued from previous page)

```
        :param Exception reason: why the channel was closed

        """
        LOGGER.warning('Channel %i was closed: %s', channel, reason)
        self._connection.close()

    def on_channel_open(self, channel):
        """This method is invoked by pika when the channel has been opened.
        The channel object is passed in so we can make use of it.

        Since the channel is now open, we'll declare the exchange to use.

        :param pika.channel.Channel channel: The channel object

        """
        LOGGER.info('Channel opened')
        self._channel = channel
        self.add_on_channel_close_callback()
        self.setup_exchange(self.EXCHANGE)

    def setup_exchange(self, exchange_name):
        """Setup the exchange on RabbitMQ by invoking the Exchange.Declare RPC
        command. When it is complete, the on_exchange_declareok method will
        be invoked by pika.

        :param str|unicode exchange_name: The name of the exchange to declare

        """
        LOGGER.info('Declaring exchange %s', exchange_name)
        self._channel.exchange_declare(
            callback=self.on_exchange_declareok,
            exchange=exchange_name,
            exchange_type=self.EXCHANGE_TYPE,
        )

    def on_exchange_declareok(self, unused_frame):
        """Invoked by pika when RabbitMQ has finished the Exchange.Declare RPC
        command.

        :param pika.Frame.Method unused_frame: Exchange.DeclareOk response frame

        """
        LOGGER.info('Exchange declared')
        self.setup_queue(self.QUEUE)

    def setup_queue(self, queue_name):
        """Setup the queue on RabbitMQ by invoking the Queue.Declare RPC
        command. When it is complete, the on_queue_declareok method will
        be invoked by pika.

        :param str|unicode queue_name: The name of the queue to declare.

        """
        LOGGER.info('Declaring queue %s', queue_name)
        self._channel.queue_declare(
            queue=queue_name,
            callback=self.on_queue_declareok,
```

(continues on next page)

(continued from previous page)

```
        )

    def on_queue_declareok(self, method_frame):
        """Method invoked by pika when the Queue.Declare RPC call made in
        setup_queue has completed. In this method we will bind the queue
        and exchange together with the routing key by issuing the Queue.Bind
        RPC command. When this command is complete, the on_bindok method will
        be invoked by pika.

        :param pika.frame.Method method_frame: The Queue.DeclareOk frame

        """
        LOGGER.info('Binding %s to %s with %s',
                    self.EXCHANGE, self.QUEUE, self.ROUTING_KEY)
        self._channel.queue_bind(
            queue=self.QUEUE,
            exchange=self.EXCHANGE,
            routing_key=self.ROUTING_KEY,
            callback=self.on_bindok,
        )

    def add_on_cancel_callback(self):
        """Add a callback that will be invoked if RabbitMQ cancels the consumer
        for some reason. If RabbitMQ does cancel the consumer,
        on_consumer_cancelled will be invoked by pika.

        """
        LOGGER.info('Adding consumer cancellation callback')
        self._channel.add_on_cancel_callback(self.on_consumer_cancelled)

    def on_consumer_cancelled(self, method_frame):
        """Invoked by pika when RabbitMQ sends a Basic.Cancel for a consumer
        receiving messages.

        :param pika.frame.Method method_frame: The Basic.Cancel frame

        """
        LOGGER.info('Consumer was cancelled remotely, shutting down: %r',
                    method_frame)
        if self._channel:
            self._channel.close()

    def acknowledge_message(self, delivery_tag):
        """Acknowledge the message delivery from RabbitMQ by sending a
        Basic.Ack RPC method for the delivery tag.

        :param int delivery_tag: The delivery tag from the Basic.Deliver frame

        """
        LOGGER.info('Acknowledging message %s', delivery_tag)
        self._channel.basic_ack(delivery_tag)

    def on_message(self, unused_channel, basic_deliver, properties, body):
        """Invoked by pika when a message is delivered from RabbitMQ. The
        channel is passed for your convenience. The basic_deliver object that
        is passed in carries the exchange, routing key, delivery tag and
        a redelivered flag for the message. The properties passed in is an
```

(continues on next page)

(continued from previous page)

```
        instance of BasicProperties with the message properties and the body
        is the message that was sent.

        :param pika.channel.Channel unused_channel: The channel object
        :param pika.Spec.Basic.Deliver: basic_deliver method
        :param pika.Spec.BasicProperties: properties
        :param bytes body: The message body

        """
        LOGGER.info('Received message # %s from %s: %s',
                    basic_deliver.delivery_tag, properties.app_id, body)
        self.acknowledge_message(basic_deliver.delivery_tag)

    def on_cancelok(self, unused_frame):
        """This method is invoked by pika when RabbitMQ acknowledges the
        cancellation of a consumer. At this point we will close the channel.
        This will invoke the on_channel_closed method once the channel has been
        closed, which will in-turn close the connection.

        :param pika.frame.Method unused_frame: The Basic.CancelOk frame

        """
        LOGGER.info('RabbitMQ acknowledged the cancellation of the consumer')
        self.close_channel()

    def stop_consuming(self):
        """Tell RabbitMQ that you would like to stop consuming by sending the
        Basic.Cancel RPC command.

        """
        if self._channel:
            LOGGER.info('Sending a Basic.Cancel RPC command to RabbitMQ')
            self._channel.basic_cancel(self.on_cancelok, self._consumer_tag)

    def start_consuming(self):
        """This method sets up the consumer by first calling
        add_on_cancel_callback so that the object is notified if RabbitMQ
        cancels the consumer. It then issues the Basic.Consume RPC command
        which returns the consumer tag that is used to uniquely identify the
        consumer with RabbitMQ. We keep the value to use it when we want to
        cancel consuming. The on_message method is passed in as a callback pika
        will invoke when a message is fully received.

        """
        LOGGER.info('Issuing consumer related RPC commands')
        self.add_on_cancel_callback()
        self._consumer_tag = self._channel.basic_consume(
            on_message_callback=self.on_message,
            queue=self.QUEUE,
        )

    def on_bindok(self, unused_frame):
        """Invoked by pika when the Queue.Bind method has completed. At this
        point we will start consuming messages by calling start_consuming
        which will invoke the needed RPC commands to start the process.

        :param pika.frame.Method unused_frame: The Queue.BindOk response frame
```

(continues on next page)

(continued from previous page)

```
        """
        LOGGER.info('Queue bound')
        self.start_consuming()

    def close_channel(self):
        """Call to close the channel with RabbitMQ cleanly by issuing the
        Channel.Close RPC command.

        """
        LOGGER.info('Closing the channel')
        self._channel.close()

    def open_channel(self):
        """Open a new channel with RabbitMQ by issuing the Channel.Open RPC
        command. When RabbitMQ responds that the channel is open, the
        on_channel_open callback will be invoked by pika.

        """
        LOGGER.info('Creating a new channel')
        self._connection.channel(on_open_callback=self.on_channel_open)

    def run(self):
        """Run the example consumer by connecting to RabbitMQ and then
        starting the IOLoop to block and allow the SelectConnection to operate.

        """
        self._connection = self.connect()
        self._connection.ioloop.start()

    def stop(self):
        """Cleanly shutdown the connection to RabbitMQ by stopping the consumer
        with RabbitMQ. When RabbitMQ confirms the cancellation, on_cancelok
        will be invoked by pika, which will then closing the channel and
        connection. The IOLoop is started again because this method is invoked
        when CTRL-C is pressed raising a KeyboardInterrupt exception. This
        exception stops the IOLoop which needs to be running for pika to
        communicate with RabbitMQ. All of the commands issued prior to starting
        the IOLoop will be buffered but not processed.

        """
        LOGGER.info('Stopping')
        self._closing = True
        self.stop_consuming()
        self._connection.ioloop.start()
        LOGGER.info('Stopped')


def main():
    logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)
    example = ExampleConsumer('amqp://guest:guest@localhost:5672/%2F')
    try:
        example.run()
    except KeyboardInterrupt:
        example.stop()
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

2.3.14 TLS parameters example

This example demonstrates a TLS session with RabbitMQ using mutual authentication (server and client authentication). It was tested against RabbitMQ 3.7.4, using Python 3.6.5 and Pika 1.0.0.

See the RabbitMQ TLS/SSL documentation for certificate generation and RabbitMQ TLS configuration. Please note that the RabbitMQ TLS (x509 certificate) authentication mechanism must be enabled for these examples to work.

tls_example.py:

```
import logging
import pika
import ssl

logging.basicConfig(level=logging.INFO)
context = ssl.create_default_context(
    cafile="PIKA_DIR/testdata/certs/ca_certificate.pem")
context.load_cert_chain("PIKA_DIR/testdata/certs/client_certificate.pem",
                        "PIKA_DIR/testdata/certs/client_key.pem")
ssl_options = pika.SSLOptions(context, "localhost")
conn_params = pika.ConnectionParameters(port=5671,
                                        ssl_options=ssl_options)

with pika.BlockingConnection(conn_params) as conn:
    ch = conn.channel()
    ch.queue_declare("foobar")
    ch.basic_publish("", "foobar", "Hello, world!")
    print(ch.basic_get("foobar"))
```

rabbitmq.config:

```
# Enable AMQPS
listeners.ssl.default = 5671
ssl_options.cacertfile = PIKA_DIR/testdata/certs/ca_certificate.pem
ssl_options.certfile = PIKA_DIR/testdata/certs/server_certificate.pem
ssl_options.keyfile = PIKA_DIR/testdata/certs/server_key.pem
ssl_options.verify = verify_peer
ssl_options.fail_if_no_peer_cert = true

# Enable HTTPS
management.listener.port = 15671
management.listener.ssl = true
management.listener.ssl_opts.cacertfile = PIKA_DIR/testdata/certs/ca_certificate.pem
management.listener.ssl_opts.certfile = PIKA_DIR/testdata/certs/server_certificate.pem
management.listener.ssl_opts.keyfile = PIKA_DIR/testdata/certs/server_key.pem
```

To perform mutual authentication with a Twisted connection:

```
from pika import ConnectionParameters
from pika.adapters import twisted_connection
from pika.credentials import ExternalCredentials
```

(continues on next page)

(continued from previous page)

```
from twisted.internet import defer, protocol, ssl, reactor

@defer.inlineCallbacks
def publish(connection):
    channel = yield connection.channel()
    yield channel.basic_publish(
        exchange='amq.topic',
        routing_key='hello.world',
        body='Hello World!',
    )
    print("published")

def connection_ready(conn):
    conn.ready.addCallback(lambda _ :conn)
    return conn.ready

# Load the CA certificate to validate the server's identity
with open("PIKA_DIR/testdata/certs/ca_certificate.pem") as fd:
    ca_cert = ssl.Certificate.loadPEM(fd.read())

# Load the client certificate and key to authenticate with the server
with open("PIKA_DIR/testdata/certs/client_key.pem") as fd:
    client_key = fd.read()
with open("PIKA_DIR/testdata/certs/client_certificate.pem") as fd:
    client_cert = fd.read()
client_keypair = ssl.PrivateCertificate.loadPEM(client_key + client_cert)

context_factory = ssl.optionsForClientTLS(
    "localhost",
    trustRoot=ca_cert,
    clientCertificate=client_keypair,
)
params = ConnectionParameters(credentials=ExternalCredentials())
cc = protocol.ClientCreator(
    reactor, twisted_connection.TwistedProtocolConnection, params)
deferred = cc.connectSSL("localhost", 5671, context_factory)
deferred.addCallback(connection_ready)
deferred.addCallback(publish)
reactor.run()
```

2.3.15 TLS parameters example

This examples demonstrates a TLS session with RabbitMQ using server authentication.

It was tested against RabbitMQ 3.6.10, using Python 3.6.1 and pre-release Pika *0.11.0*

Note the use of *ssl.PROTOCOL_TLSv1_2*. The recent versions of RabbitMQ disable older versions of SSL due to security vulnerabilities.

See https://www.rabbitmq.com/ssl.html for certificate creation and rabbitmq SSL configuration instructions.

tls_example.py:

```
import ssl
import pika
import logging
```

(continues on next page)

(continued from previous page)

```
logging.basicConfig(level=logging.INFO)

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.load_verify_locations('/Users/me/tls-gen/basic/testca/cacert.pem')

cp = pika.ConnectionParameters(ssl_options=pika.SSLOptions(context))

conn = pika.BlockingConnection(cp)
ch = conn.channel()
print(ch.queue_declare("sslq"))
ch.publish("", "sslq", "abc")
print(ch.basic_get("sslq"))
```

rabbitmq.conf:

```
%% In this example, both the client and RabbitMQ server are assumed to be running on␣
↪the same machine
%% with a self-signed set of certificates generated using https://github.com/
↪michaelklishin/tls-gen.
%%
%% To find out the default rabbitmq.conf location, see https://www.rabbitmq.com/
↪configure.html.
%%
%% The contents of the example config file are for demonstration purposes only.
%% See https://www.rabbitmq.com/ssl.html to learn how to use TLS for client␣
↪connections in RabbitMQ.
%%
%% The example below allows clients without a certificate to connect
%% but performs peer verification on those that present a certificate chain.

listeners.ssl.default = 5671

ssl_options.cacertfile = /Users/me/tls-gen/basic//ca_certificate.pem
ssl_options.certfile = /Users/me/tls-gen/basic//server_certificate.pem
ssl_options.keyfile = /Users/me/tls-gen/basic/server_key.pem
ssl_options.verify = verify_peer
ssl_options.fail_if_no_peer_cert = false
```

2.3.16 Ensuring well-behaved connection with heartbeat and blocked-connection timeouts

This example demonstrates explicit setting of heartbeat and blocked connection timeouts.

Starting with RabbitMQ 3.5.5, the broker's default heartbeat timeout decreased from 580 seconds to 60 seconds. As a result, applications that perform lengthy processing in the same thread that also runs their Pika connection may experience unexpected dropped connections due to heartbeat timeout. Here, we specify an explicit lower bound for heartbeat timeout.

When RabbitMQ broker is running out of certain resources, such as memory and disk space, it may block connections that are performing resource-consuming operations, such as publishing messages. Once a connection is blocked, RabbitMQ stops reading from that connection's socket, so no commands from the client will get through to the broker on that connection until the broker unblocks it. A blocked connection may last for an indefinite period of time, stalling the connection and possibly resulting in a hang (e.g., in BlockingConnection) until the connection is unblocked.

Blocked Connection Timeout is intended to interrupt (i.e., drop) a connection that has been blocked longer than the given timeout value.

Example of configuring heartbeat and blocked-connection timeouts:

```
import pika


def main():

    # NOTE: These parameters work with all Pika connection types
    params = pika.ConnectionParameters(heartbeat=600,
                                       blocked_connection_timeout=300)

    conn = pika.BlockingConnection(params)

    chan = conn.channel()

    chan.basic_publish('', 'my-alphabet-queue', "abc")

    # If publish causes the connection to become blocked, then this conn.close()
    # would hang until the connection is unblocked, if ever. However, the
    # blocked_connection_timeout connection parameter would interrupt the wait,
    # resulting in ConnectionClosed exception from BlockingConnection (or the
    # on_connection_closed callback call in an asynchronous adapter)
    conn.close()


if __name__ == '__main__':
    main()
```

2.4 Frequently Asked Questions

- Is Pika thread safe?

  Pika does not have any notion of threading in the code. If you want to use Pika with threading, make sure you have a Pika connection per thread, created in that thread. It is not safe to share one Pika connection across threads, with one exception: you may call the connection method *add_callback_threadsafe* from another thread to schedule a callback within an active pika connection.

- How do I report a bug with Pika?

  The main Pika repository is hosted on Github and we use the Issue tracker at https://github.com/pika/pika/issues.

- Is there a mailing list for Pika?

  Yes, Pika's mailing list is available on Google Groups and the email address is pika-python@googlegroups.com, though traditionally questions about Pika have been asked on the RabbitMQ mailing list.

- How can I contribute to Pika?

  You can fork the project on Github and issue Pull Requests when you believe you have something solid to be added to the main repository.

2.5 Contributors

The following people have directly contributes code by way of new features and/or bug fixes to Pika:

- Gavin M. Roy
- Tony Garnock-Jones
- Vitaly Kruglikov
- Michael Laing
- Marek Majkowski
- Jan Urbański
- Brian K. Jones
- Ask Solem
- ml
- Will
- atatsu
- Fredrik Svensson
- Pedro Abranches
- Kyösti Herrala
- Erik Andersson
- Charles Law
- Alex Chandel
- Tristan Penman
- Raphaël De Giusti
- Jozef Van Eenbergen
- Josh Braegger
- Jason J. W. Williams
- James Mutton
- Cenk Alti
- Asko Soukka
- Antti Haapala
- Anton Ryzhov
- cellscape
- cacovsky
- bra-fsn
- ateska
- Roey Berman
- Robert Weidlich

- Riccardo Cirimelli
- Perttu Ranta-aho
- Pau Gargallo
- Kane
- Kamil Kisiel
- Jonty Wareing
- Jonathan Kirsch
- Jacek 'Forger' Całusiński
- Garth Williamson
- Erik Olof Gunnar Andersson
- David Strauss
- Anton V. Yanchenko
- Alexey Myasnikov
- Alessandro Tagliapietra
- Adam Flynn
- skftn
- saarni
- pavlobaron
- nonleaf
- markcf
- george y
- eivanov
- bstemshorn
- a-tal
- Yang Yang
- Stuart Longland
- Sigurd Høgsbro
- Sean Dwyer
- Samuel Stauffer
- Roberto Decurnex
- Rikard Hultén
- Richard Boulton
- Ralf Nyren
- Qi Fan
- Peter Magnusson
- Pankrat

- Olivier Le Thanh Duong
- Njal Karevoll
- Milan Skuhra
- Mik Kocikowski
- Michael Kenney
- Mark Unsworth
- Luca Wehrstedt
- Laurent Eschenauer
- Lars van de Kerkhof
- Kyösti Herrala
- Juhyeong Park
- JuhaS
- Josh Hansen
- Jorge Puente Sarrín
- Jeff Tang
- Jeff Fein-Worton
- Jeff
- Hunter Morris
- Guruprasad
- Garrett Cooper
- Frank Slaughter
- Dustin Koupal
- Bjorn Sandberg
- Axel Eirola
- Andrew Smith
- Andrew Grigorev
- Andrew
- Allard Hoeve
- A.Shaposhnikov

Contributors listed by commit count.

Contributing

Test Coverage

To contribute to Pika, please make sure that any new features or changes to existing functionality **include test coverage**.

Pull requests that add or change code without coverage have a much lower chance of being accepted.

Prerequisites

Pika test suite has a couple of requirements:

- Dependencies from *test-requirements.txt* are installed
- A RabbitMQ node with all defaults is running on *localhost:5672*

Installing Dependencies

To install the dependencies needed to run Pika tests, use

pip install -r test-requirements.txt

which on Python 3 might look like this

pip3 install -r test-requirements.txt

Running Tests

To run all test suites, use

nose2

Note that some tests are OS-specific (e.g. epoll on Linux or kqueue on MacOS and BSD). Those will be skipped automatically.

If you would like to run TLS/SSL tests, use the following procedure:

- Create a *rabbitmq.conf* file:

  ` sed -e "s#PIKA_DIR#$PWD#g" ./testdata/rabbitmq.conf.in > ./testdata/rabbitmq.conf `

- Start RabbitMQ and use the configuration file you just created. An example command that works with the *generic-unix* package is as follows:

  ` $ RABBITMQ_CONFIG_FILE=/path/to/pika/testdata/rabbitmq.conf ./sbin/rabbitmq-server `

- Run the tests indicating that TLS/SSL connections should be used:

  ` PIKA_TEST_TLS=true nose2 `

Code Formatting

Please format your code using [yapf](http://pypi.python.org/pypi/yapf) with `google` style prior to issuing your pull request. *Note: only format those lines that you have changed in your pull request. If you format an entire file and change code outside of the scope of your PR, it will likely be rejected.*

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V