SAMPLE CHAPTER

RabbitMQ IN DEPTH

Gavin M. Roy

MANNING

RabbitMQ in Depth

by Gavin Roy

Chapter 2

brief contents

2 How to speak Rabbit: the AMQ Protocol

This chapter covers

- Communicating with RabbitMQ via the AMQ Protocol
- Framing the AMQ Protocol at a low level
- Publishing messages into RabbitMQ
- Getting messages from RabbitMQ

The process that RabbitMQ and client libraries go through in order to get a message from your application into RabbitMQ and from RabbitMQ into consumer applications can be complex. If you're processing critical information, such as sales data, reliably delivering the canonical source of information about the sale should be a top priority. At the protocol level, the AMQP specification defines the semantics for client and broker to negotiate and speak to each other about the process for relaying your information. Oftentimes the lexicon defined in the AMQP specification bubbles its way up into RabbitMQ client libraries, with the classes and methods used by applications communicating with RabbitMQ mirroring the protocol-level classes and methods. Understanding how this communication takes place will help you learn not just the "how" of communicating with RabbitMQ but also the "why."

Even though the commands in client libraries tend to resemble or even directly copy the actions defined in the AMQP specification, most client libraries attempt to hide the complexity of communicating via the AMQ Protocol. This tends to be a good thing when you're looking to write an application and you don't want to worry about the intricacies of how things work. But skipping over the technical foundation of what RabbitMQ clients are doing isn't very helpful when you want to truly understand what's going on with your application. Whether you want to know why your application is slower to publish than you might expect, or you just want to know what steps a client must take in order to establish that first connection with RabbitMQ, knowing how your client is talking to RabbitMQ will make that process much easier.

To better illustrate the how and why, in this chapter you'll learn how AMQP splits communication between the client and broker into chunks of data called *frames,* and how these frames detail the actions your client application wants RabbitMQ to take and the actions RabbitMQ wants your client application to take. In addition, you'll learn how these frames are constructed at the protocol level, and how they provide the mechanism by which messages are delivered and consumed.

Building on this information, you'll write your first application in Python using a RabbitMQ client library written as a teaching aid for this book. This application will use AMQP to define an exchange and queue and then bind them together. Finally, you'll write a consumer application that will read the messages from the newly defined queue and print the contents of the message. If you're already comfortable doing these things, you should still dive into this chapter. I found that it was only after I fully understood the semantics of AMQP, the "why" instead of just the "how," that I understood RabbitMQ.

2.1 AMQP as an RPC transport

As an AMQP broker, RabbitMQ speaks a strict dialect for communication, utilizing a *remote procedure call* (RPC) pattern in nearly every aspect of communication with the core product. A remote procedure call is a type of communication between computers that allows one computer to execute a program or its methods on the other. If you've done web programming where you're talking to a remote API, you're using a common RPC pattern.

However, the RPC conversations that take place when communicating with RabbitMQ are unlike most web-based API calls. In most web API definitions, there are RPC conversations where the client issues commands and the server responds—the server doesn't issue commands back to the client. In the AMQP specification, both the server and the client can issue commands. For a client application, this means that it should be listening for communication from the server that may have little to do with what the client application is doing.

To illustrate how RPC works when a client is talking to RabbitMQ, let's consider the connection negotiation process.

2.1.1 Kicking off the conversation

When you're communicating with someone new in a foreign country, it's inevitable that one of you will kick off the conversation with a greeting, something that lets you and the other person know if you're both capable of speaking the same language. When speaking AMQP, this greeting is the *protocol header*, and it's sent by the client to the server. This greeting shouldn't be considered a request, however, as unlike the rest of the conversation that will take place, it's not a command. RabbitMQ starts the command/response sequence by replying to the greeting with a `Connection.Start` command, and the client responds to the RPC request with `Connection.StartOk` response frame (figure 2.1).

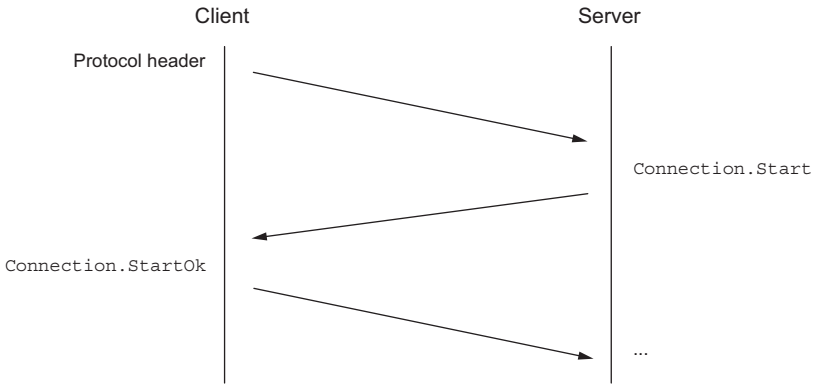


Figure 2.1 The initial communication negotiation with RabbitMQ demonstrates the RPC process in AMQP.

The full conversation for initiating a connection isn't terribly important unless you're writing a client library, but it's worth noting that to fully connect to RabbitMQ, there's a sequence of three synchronous RPC requests to start, tune, and open the connection. Once this sequence has finished, RabbitMQ will be ready for your application to make requests.

There are a whole range of different commands your application can send to RabbitMQ and that RabbitMQ can send to your client. You'll learn a small subset of these commands later in the chapter, but before that happens, you have to open a channel.

2.1.2 Tuning in to the right channel

Similar in concept to channels on a two-way radio, the AMQP specification defines channels for communicating with RabbitMQ. Two-way radios transmit information to each other using the airwaves as the connection between them. In AMQP, channels use the negotiated AMQP connection as the conduit for transmitting information to each other, and like channels on a two-way radio, they isolate their transmissions from other conversations that are happening. A single AMQP connection can have multiple

channels, allowing multiple conversations between a client and server to take place. In technical terms, this is called *multiplexing*, and it can be useful for multithreaded or asynchronous applications that perform multiple tasks.

> **TIP** In creating your client applications, it's important not to overcomplicate things with too many channels. On the wire in marshaled frames, channels are nothing more than an integer value that's assigned to the messages that are passed between a server and client; in the RabbitMQ server and client, they represent more. There are memory structures and objects set up for each channel. The more channels you have in a connection, the more memory RabbitMQ must use to manage the message flow for that connection. If you use them judiciously, you'll have a happier RabbitMQ server and a less complicated client application.

2.2 AMQP's RPC frame structure

Very similar in concept to object-oriented programming in languages such as C++, Java, and Python, AMQP uses classes and methods, referred to as *AMQP commands*, to create a common language between clients and servers. The classes in AMQP define a scope of functionality, and each class contains methods that perform different tasks. In the connection negotiation process, the RabbitMQ server sends a `Connection.Start` command, marshaled into a frame, to the client. As illustrated in figure 2.2, the `Connection.Start` command is composed of two components: the AMQP *class* and *method*.

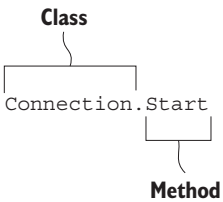


Figure 2.2 The AMQP `Connection` class and the `Start` method comprise the `Connection.Start` RPC request.

There are many commands in the AMQP specification, but if you're like me, you'll want to skip through all of that and get to the important bits of sending and receiving messages. It's important, however, to understand how the commands you'll be sending and receiving with RabbitMQ are represented on the wire to truly appreciate what's happening in your applications.

2.2.1 AMQP frame components

When commands are sent to and from RabbitMQ, all of the arguments required to execute them are encapsulated in data structures called frames that encode the data for transmission. Frames provide an efficient way for the command and its arguments to be encoded and delimited on the wire. You can think of frames as being like freight cars on a train. As a generalization, freight cars have the same basic structure and are

differentiated by what they contain. The same is true with low-level AMQP frames. As figure 2.3 illustrates, a low-level AMQP frame is composed of five distinct components:

1. Frame type
2. Channel number
3. Frame size in bytes
4. Frame payload
5. End-byte marker (ASCII value 206)

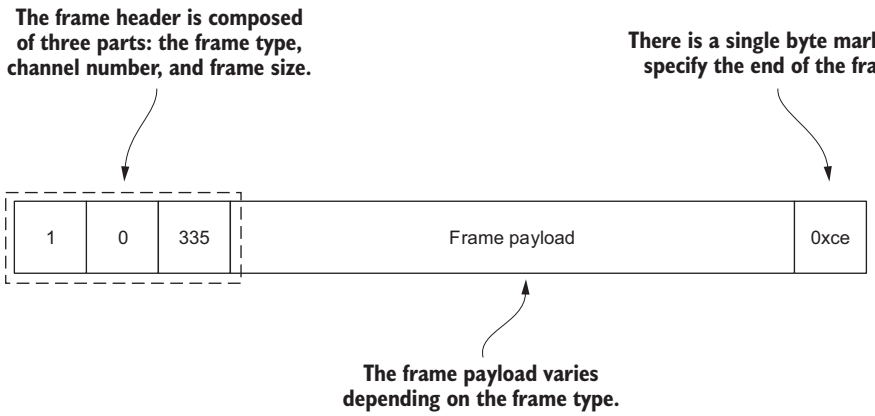


Figure 2.3 The anatomy of a low-level AMQP frame

A low-level AMQP frame starts off with three fields, referred to as a *frame header* when combined. The first field is a single byte indicating the frame type, and the second field specifies the channel the frame is for. The third field carries the byte size of the frame payload. The frame header, along with the end-byte marker, creates the structure for the frame.

Carried inside the frame, after the header and before the end-byte marker, is the frame payload. Much like the freight car protecting its contents on a train, the frame is designed to protect the integrity of the content it carries.

2.2.2 Types of frames

The AMQP specification defines five types of frames: a protocol header frame, a method frame, a content header frame, a body frame, and a heartbeat frame. Each frame type has a distinct purpose, and some are used much more frequently than others:

- The protocol header frame is only used once, when connecting to RabbitMQ.
- A method frame carries with it the RPC request or response that's being sent to or received from RabbitMQ.
- A content header frame contains the size and properties for a message.

- Body frames contain the content of messages.
- The heartbeat frame is sent to and from RabbitMQ as a check to ensure that both sides of the connection are available and working properly.

Whereas the protocol header and heartbeat frames are generally abstracted away from developers when using a client library, the method, content header, and body frames and their constructs are usually surfaced when writing applications that communicate with RabbitMQ. In the next section, you'll learn how messages that are sent into and received from RabbitMQ are marshaled into a method frame, a content header frame, and one or more body frames.

NOTE The heartbeat behavior in AMQP is used to ensure that both client and server are responding to each other, and it's a perfect example of how AMQP is a bidirectional RPC protocol. If RabbitMQ sends a heartbeat to your client application, and it doesn't respond, RabbitMQ will disconnect it. Oftentimes developers in single-threaded or asynchronous development environments will want to increase the timeout to some large value. If you find your application blocks communication in a way that makes heartbeats difficult to work with, you can turn them off by setting the heartbeat interval to 0 when creating your client connection. If, instead, you choose to use a much higher value than the default of 600 seconds, you can change RabbitMQ's maximum heartbeat interval value by changing the `heartbeat` value in the rabbitmq.config file.

2.2.3 Marshaling messages into frames

When publishing a message to RabbitMQ, the method, header, and body frames are used. The first frame sent is the method frame carrying the command and the parameters required to execute it, such as the exchange and routing key. Following the method frame are the content frames: a content header and body. The content header frame contains the message properties along with the body size. AMQP has a maximum frame size, and if the body of your message exceeds that size, the content will be split into multiple body frames. These frames are always sent in the same order over the wire: a method frame, content header frame, and one or more body frames (figure 2.4).

As figure 2.4 illustrates, when sending a message to RabbitMQ, a `Basic.Publish` command is sent in the method frame, and that's followed by a content header frame with the message's properties, such as the message's content type and the time when the message was sent. These properties are encapsulated in a data structure defined in the AMQP specification as `Basic.Properties`. Finally, the content of the message is marshaled into the appropriate number of body frames.

NOTE Although the default frame size is 131 KB, client libraries can negotiate a larger or smaller maximum frame size during the connection process, up to a 32-bit value for the number of bytes in a frame.

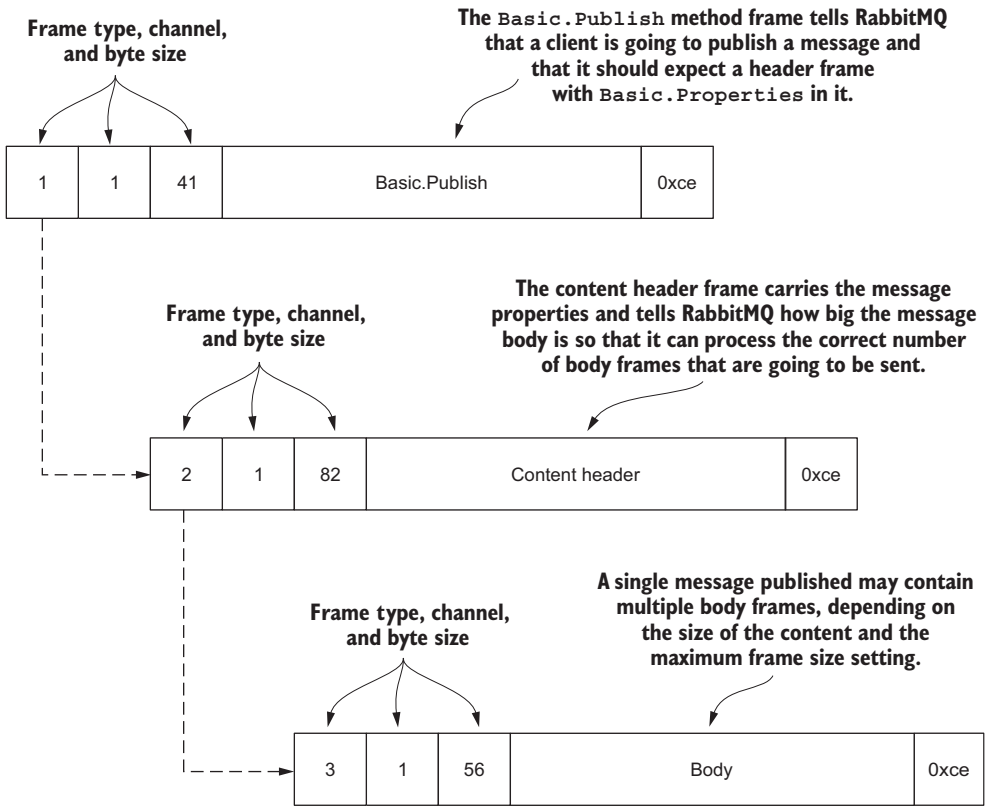


Figure 2.4 A single message published into RabbitMQ is composed of three frame types: the method frame for the `Basic.Publish` RPC call, a header frame, and one or more body frames.

In order to be more efficient and minimize the size of the data being transferred, the content in the method frame and content header frame is binary packed data and is not human-readable. Unlike the method and header frames, the message content carried inside the body frame isn't packed or encoded in any way and may be anything from plain text to binary image data.

To further illustrate the anatomy of an AMQP message, let's examine these three frame types in more detail.

2.2.4 The anatomy of a method frame

Method frames carry with them the class and method your RPC request is going to make as well as the arguments that are being passed along for processing. In figure 2.5, the method frame carrying a `Basic.Publish` command carries the binary packed data describing the command, and the request arguments that are passing along with it. The first two fields are numeric representations of the `Basic` class and the `Publish`

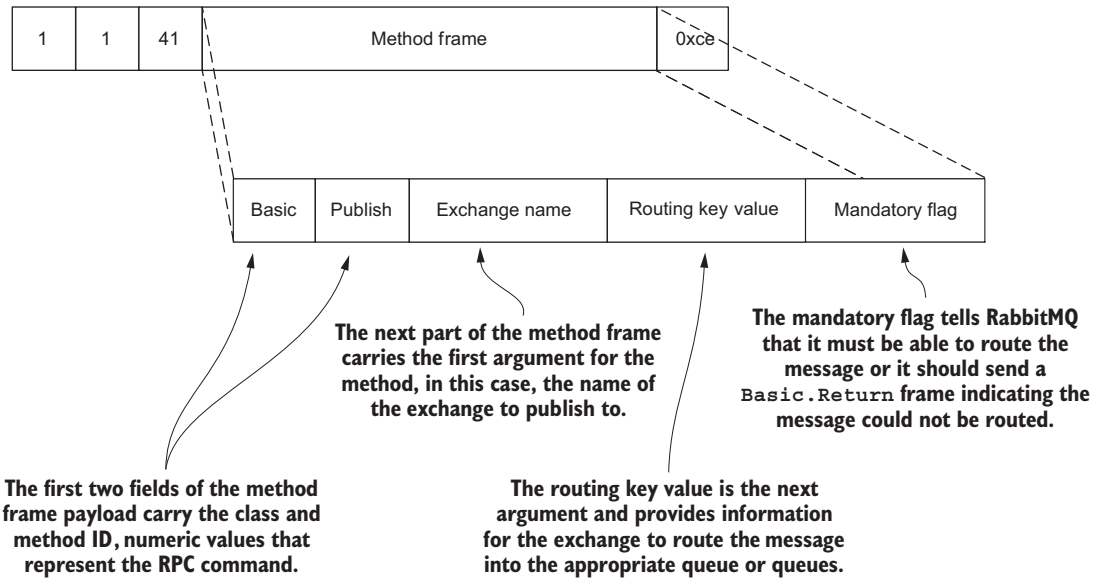


Figure 2.5 The `Basic.Publish` method frame is composed of five components: the class type and method type that identifies it as a `Basic.Publish` RPC request, the exchange name, a routing key value, and a mandatory flag.

method. These fields are followed by the string values for the exchange name and the routing key. As previously mentioned, these attributes instruct RabbitMQ on how to route a message. The `mandatory` flag tells RabbitMQ that the message must be delivered or the publishing of the message should fail.

Each data value in the method frame payload is encoded in a data-type-specific format. This format is designed to minimize byte size on the wire, ensure data integrity, and ensure that data marshaling and unmarshaling are as fast as possible. The actual format varies depending on the data type, but it's usually a single byte followed by numeric data, or a single byte followed by a byte-size field and then text data.

NOTE Usually, sending a message using the `Basic.Publish` RPC request is a single-sided conversation. In fact, the AMQP specification goes as far as to say that success, as a general rule, is silent, whereas errors should be as noisy and intrusive as possible. But if you're using the `mandatory` flag when publishing your messages, your application should be listening for a `Basic.Return` command sent from RabbitMQ. If RabbitMQ isn't able to meet the requirements set by the `mandatory` flag, it will send a `Basic.Return` command to your client on the same channel. More information about `Basic.Return` is covered in chapter 4.

2.2.5 The content header frame

The headers that are sent along after the method frame carry more than the data that tells RabbitMQ how big your message is. As illustrated in figure 2.6, the header frame also carries attributes about your message that describe the message to both the RabbitMQ server and to any application that may receive it. These attributes, as values in a `Basic.Properties` table, may contain data that describes the content of your message or they may be completely blank. Most client libraries will prepopulate a minimal set of fields, such as the content type and the delivery mode.

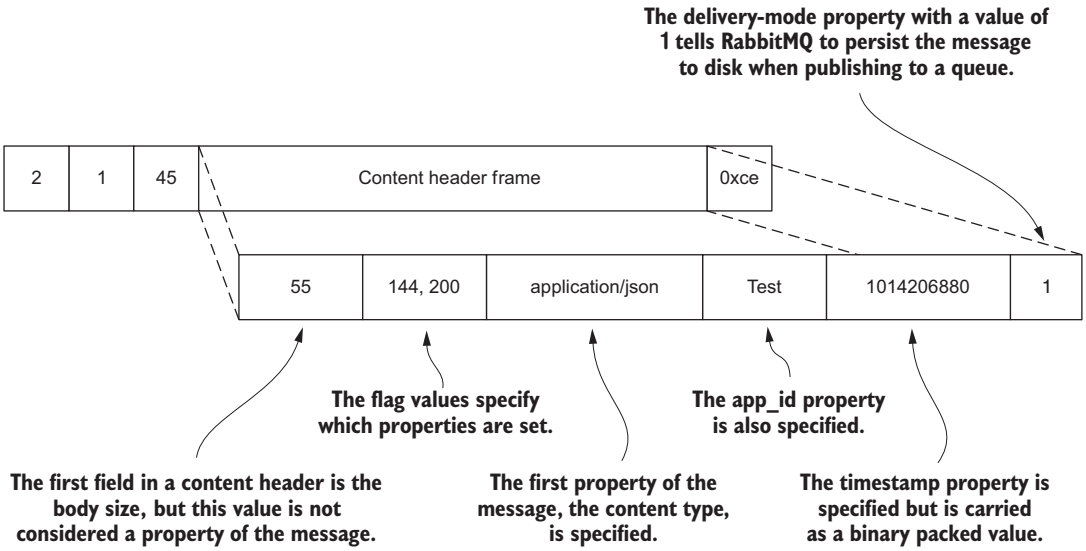


Figure 2.6 A message header carries the body size and a `Basic.Properties` table.

Properties are powerful tools in composing your message. They can be used to create a contract between publishers and consumers about the content of the message, allowing for a large amount of specificity about the message. You'll learn about `Basic.Properties` and the various possible uses for each field the data structure can carry in chapter 3.

2.2.6 The body frame

The body frame for a message is agnostic to the type of data being transferred, and it may contain either binary or text data. Whether you're sending binary data such as a JPEG image or serialized data in a JSON or XML format, the message body frame is the structure in the message that carries the actual message data (figure 2.7).

Together, the message properties and body form a powerful encapsulation format for your data. Marrying the descriptive attributes of the message with the content-agnostic body ensures you can use RabbitMQ for any type of data you deem appropriate.

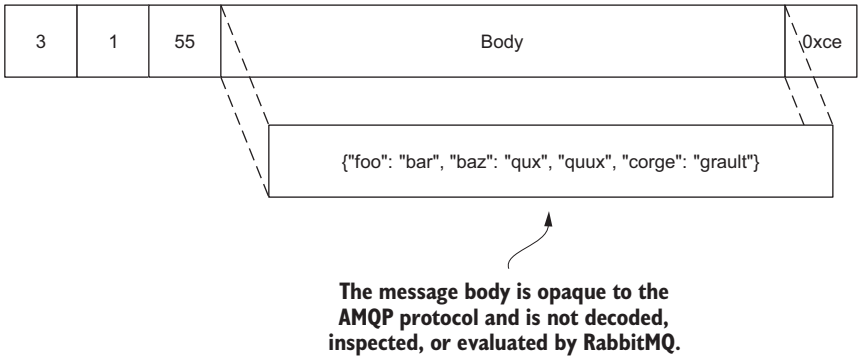


Figure 2.7 A message body embedded in an AMQP frame

2.3 Putting the protocol to use

There are a few configuration-related steps you must take care of before you can publish messages into a queue. At a minimum, you must set up both an exchange and a queue, and then bind them together.

But before you actually perform those steps, let's look at what needs to happen at a protocol level to enable a message to be published, routed, queued, and delivered, starting with setting up an exchange for routing messages.

2.3.1 Declaring an exchange

Exchanges, like queues, are first-rate citizens in the AMQ model. As such, each has its own class in the AMQP specification. Exchanges are created using the `Exchange.Declare` command, which has arguments that define the name of the exchange, its type, and other metadata that may be used for message processing.

Once the command has been sent and RabbitMQ has created the exchange, an `Exchange.DeclareOk` method frame is sent in response (figure 2.8). If, for whatever reason, the command should fail, RabbitMQ will close the channel that the `Exchange.Declare` command was sent on by sending a `Channel.Close` command. This response will include a numeric reply code and text value indicating why the `Exchange.Declare` failed and the channel was closed.

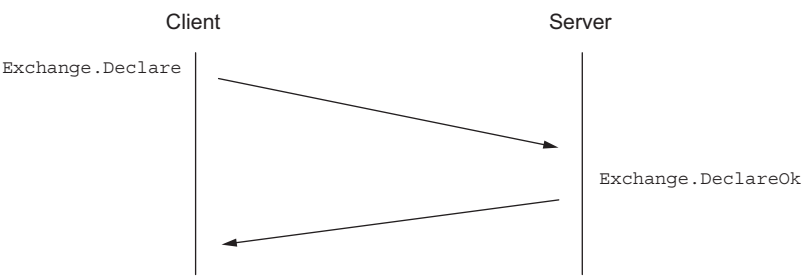


Figure 2.8 The communication sequence that occurs when declaring an exchange

2.3.2 Declaring a queue

Once the exchange has been created, it's time to create a queue by sending a `Queue.Declare` command to RabbitMQ. Like the `Exchange.Declare` command, there's a simple communication sequence that takes place (figure 2.9), and should the `Queue.Declare` command fail, the channel will be closed.

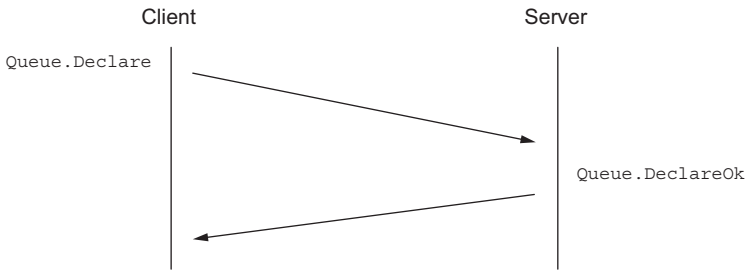


Figure 2.9 A queue-declare communication sequence consists of a `Queue.Declare` command and a `Queue.DeclareOk` response.

When declaring a queue, there's no harm in issuing the same `Queue.Declare` command more than once. RabbitMQ will consider subsequent queue declares to be passive and will return useful information about the queue, such as the number of pending messages in the queue and the number of consumers subscribed to it.

Handling errors gracefully

When you try to declare a queue with different properties than an existing queue with the same name, RabbitMQ will close the channel that the RPC request was issued on. This behavior is consistent with any other type of error that your client application may make in issuing commands to the broker. For example, if you issue a `Queue.Declare` command with a user that doesn't have *configuration* access on the virtual host, the channel will close with a 403 error.

To correctly handle errors, your client application should be listening for a `Channel.Close` command from RabbitMQ so it can respond appropriately. Some client libraries may present this information as an exception for your application to handle, whereas others may use a callback passing style where you register a method that's called when a `Channel.Close` command is sent.

If your client application isn't listening for or handling events coming from the server, you may lose messages. If you're publishing on a non-existent or closed channel, RabbitMQ may close the connection. If your application is consuming messages and doesn't know that RabbitMQ closed the channel, it may not know that RabbitMQ stopped sending your client messages and could still think that it's functioning properly and is subscribed to an empty queue.

2.3.3 Binding a queue to an exchange

Once the exchange and queue have been created, it's time to bind them together. Like with `Queue.Declare`, the command to bind a queue to an exchange, `Queue.Bind`, can only specify one queue at a time. Much like the `Exchange.Declare` and `Queue.Declare` commands, after you issue a `Queue.Bind` command, your application will receive a `Queue.BindOk` method frame if it was processed successfully (figure 2.10).

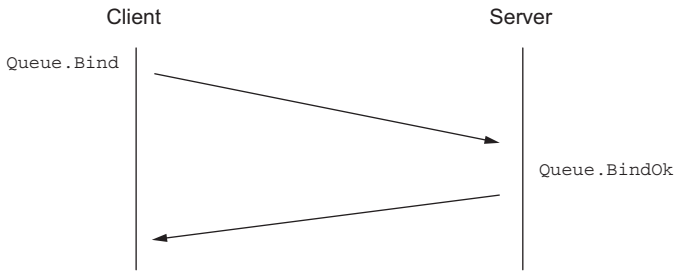


Figure 2.10 After the client successfully issues a `Queue.Bind` command to bind a queue to an exchange with a routing key, the client will receive a `Queue.BindOk` method frame in response.

As basic examples of RPC interactions between a RabbitMQ server and client, the `Exchange.Declare`, `Queue.Declare`, and `Queue.Bind` commands illustrate a common pattern that's mimicked by all synchronous commands in the AMQP specification. But there are a few asynchronous commands that break from the simple "Action" and "ActionOk" pattern. These commands deal with sending and receiving messages from RabbitMQ.

2.3.4 Publishing a message to RabbitMQ

As you previously learned, when publishing messages to RabbitMQ, multiple frames encapsulate the message data that's sent to the server. Before the actual message content ever reaches RabbitMQ, the client application sends a `Basic.Publish` method frame, a content header frame, and at least one body frame (figure 2.11).

When RabbitMQ receives all of the frames for a message, it will inspect the information it needs from the method frame before determining the next steps. The `Basic.Publish` method frame carries with it the exchange name and routing key for

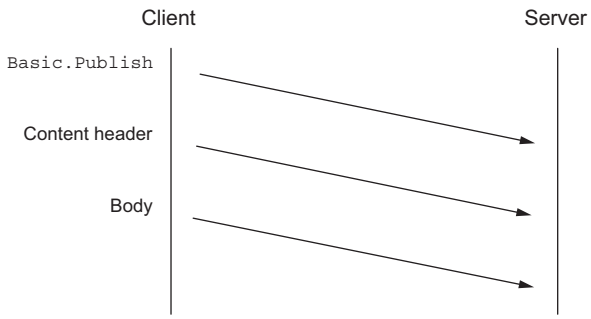


Figure 2.11 When publishing a message to RabbitMQ, at least three frames are sent: the `Basic.Publish` method frame, a content header frame, and a body frame.

the message. When evaluating this data, RabbitMQ will try to match the exchange name in the `Basic.Publish` frame against its database of configured exchanges.

> **TIP** By default, if you're publishing messages with an exchange that doesn't exist in RabbitMQ's configuration, it will silently drop the messages. To ensure your messages are delivered, either set the mandatory flag to true when publishing, or use delivery confirmations. These options are detailed in chapter 4. Be aware that using either of these methods may negatively impact the message publishing speed of your application.

When RabbitMQ finds a match to the exchange name in the `Basic.Properties` method frame, it evaluates the bindings in the exchange, looking to match queues with the routing key. When the criterion for a message matches any bound queues, the RabbitMQ server will enqueue the message in a FIFO order. Instead of putting the actual message into a queue data structure, a reference to the message is added to the queue. When RabbitMQ is ready to deliver the message, it will use the reference to compose the marshaled message and send it over the wire. This provides a substantial optimization for messages that are published to multiple queues. Holding only one instance of the message takes less physical memory when it's published to multiple destinations. The disposition of a message in a queue, whether consumed, expired, or sitting idle, will not impact the disposition of that message in any other queue. Once RabbitMQ no longer needs the message, because all copies of it have been delivered or removed, the single copy of the message data will be removed from memory in RabbitMQ.

By default, as long as there are no consumers listening to the queue, messages will be stored in the queue. As you add more messages, the queue will grow in size. RabbitMQ can keep these messages in memory or write them to disk, depending on the `delivery-mode` property specified in the message's `Basic.Properties`. The `delivery-mode` property is so important that it will be discussed in the next chapter and in even more detail in chapter 4.

2.3.5 Consuming messages from RabbitMQ

Once a published message has been routed and enqueued to one or more queues, there's not much left to discuss but its consumption. To consume messages from a queue in RabbitMQ, a consumer application subscribes to the queue in RabbitMQ by issuing a `Basic.Consume` command. Like the other synchronous commands, the server will respond with `Basic.ConsumeOk` to let the client know it's going to open the floodgates and release a torrent of messages, or at least a trickle. At RabbitMQ's discretion, the consumer will start receiving messages in the unsurprising form of `Basic.Deliver` methods and their content header and body frame counterparts (figure 2.12).

Once the `Basic.Consume` has been issued, it will stay active until one of a few things occurs. If a consumer wants to stop receiving messages, it can issue a `Basic.Cancel`

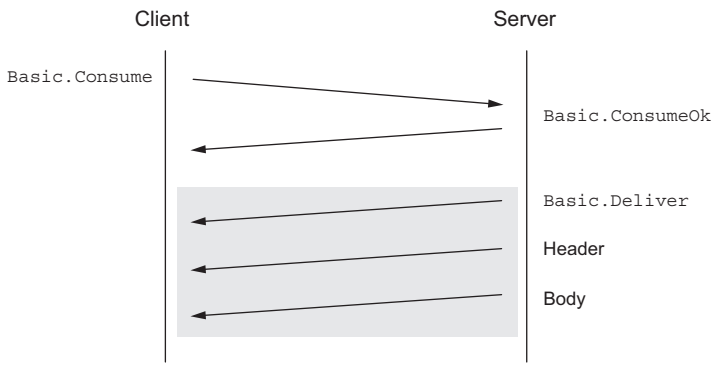


Figure 2.12 The logical frame delivery order between client and server when subscribing to a queue and receiving messages

command. It's worth noting that this command is issued asynchronously while RabbitMQ may still be sending messages, so a consumer can still receive any number of messages RabbitMQ has preallocated for it prior to receiving a `Basic.CancelOk` response frame.

When consuming messages, there are several settings that let RabbitMQ know how you want to receive them. One such setting is the `no_ack` argument for the `Basic.Consume` command. When set to true, RabbitMQ will send messages continuously until the consumer sends a `Basic.Cancel` command or the consumer is disconnected. If the `no_ack` flag is set to false, a consumer must acknowledge each message that it receives by sending a `Basic.Ack` RPC request (figure 2.13).

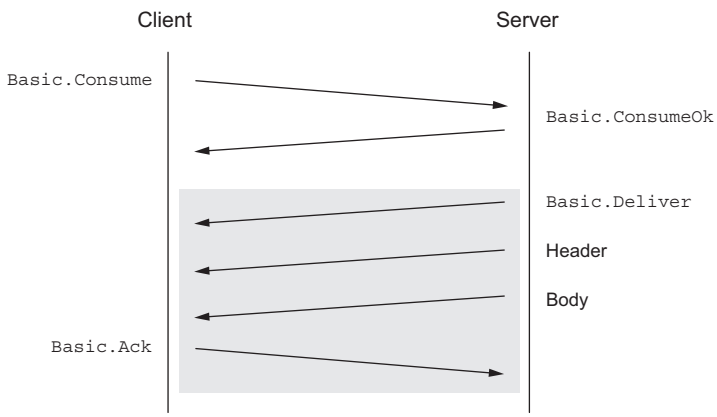


Figure 2.13 Each message successfully delivered by RabbitMQ to the client will be responded to with a `Basic.Ack`, until a `Basic.Cancel` command is sent. If `no_ack` is specified, the `Basic.Ack` step is omitted.

When the `Basic.Ack` response frame is sent, the consumer must pass with it an argument from the `Basic.Deliver` method frame called the *delivery tag*. RabbitMQ uses the delivery tag along with the channel as a unique identifier to communicate message acknowledgement, rejection, and negative acknowledgement. You'll learn more about these options in chapter 5.

2.4 Writing a message publisher in Python

Now that you have a healthy knowledge of AMQP fundamentals under your belt, it's time to turn theory into practice and write both a publisher and consumer. To do this we'll use the rabbitpy library. There are many libraries for communicating with RabbitMQ, but I created rabbitpy as a teaching aid for this book to keep the programming examples simple and concise while attempting to stay true to the AMQP command syntax. If you haven't done so yet, please install rabbitpy by following the VM installation instructions in the appendix.

To start this exercise, you'll make use of the IPython Notebook Server installed as part of the RabbitMQ in Depth virtual machine. If you've yet to do so, please follow the steps outlined in the appendix to set up the virtual machine on your local computer. Open your browser to http://localhost:8888 and you should see a page similar to figure 2.14.

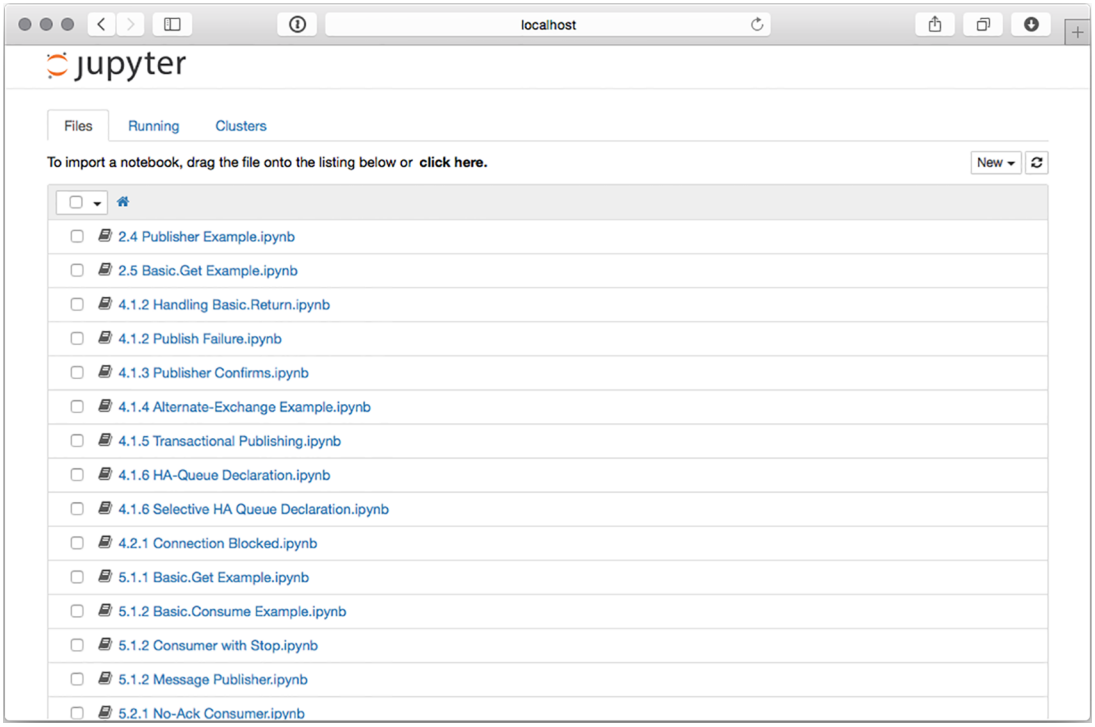


Figure 2.14 The IPython Notebook index page

The "2.4 Publisher Example" notebook in the index contains all of the code outlined in this page in order to communicate with RabbitMQ. You must import the rabbitpy library so that the Python interpreter allows you to use it:

```
In [1]: # Import the RabbitMQ Client Library
        import rabbitpy
```

If you press the Play button or the Run Cell button in the toolbar or if you press Shift-Enter, the cell containing that code will execute. In the first cell of the notebook, the rabbitpy library will be imported.

You should also have seen the asterisk (*) change to the number 1. The active cell has automatically advanced from the first to the next one. As you read through this example code, you should execute each cell as you encounter it, advancing through the code in the IPython Notebook.

Now, with the rabbitpy library imported, you'll need to create an AMQP connection URL. The format for the URL is very similar to the format used for HTTP requests:

```
In [2]: # Specify the URL to connect to
        url = 'amqp://guest:guest@localhost:5672/%2F'
```

This AMQP URL specifies that you'll connect over a normal AMQP connection using the username "guest" and the password "guest". It will connect you to localhost on port number 5672 with the default "/" vhost. This URL expects that you'll be connecting to RabbitMQ on your local machine with the default configuration. If you've set up RabbitMQ on a remote server or have changed the configuration of the RabbitMQ broker, you'll have to change the values accordingly.

Now that the URL has been defined, it's time to open a connection to RabbitMQ:

```
In [3]: # Connect to RabbitMQ using the URL above
        connection = rabbitpy.Connection(url)
```

If you didn't receive an exception, you're now connected to RabbitMQ. If you did receive one, the most likely scenario is that RabbitMQ isn't running on your local machine. Please ensure that it's running and try again.

If you're successfully connected, it's time to open a channel to communicate with RabbitMQ:

```
In [4]: # Open a new channel on the connection
        channel = connection.channel()
```

With the channel open, you can now declare an exchange by creating a new instance of the `rabbitpy.Exchange` class. Pass in the channel and the name of the exchange you'd like to create. I suggest using `chapter2-example` for now.

```
In [5]: # Create a new exchange object, passing in the channel to use
        exchange = rabbitpy.Exchange(channel, 'chapter2-example')
```

Once it's constructed, use the exchange object's `declare` method to send the command, declaring the exchange in RabbitMQ:

```
In [6]: # Declare the exchange on the RabbitMQ server
        exchange.declare()
```

Now that you've declared the exchange, you can set up the queue and bind it to the exchange. To do this, you first create the `Queue` object, passing in the channel and the name of the queue. In the example that follows, the name of the queue is `example`.

```
In [7]: # Create a new queue object, passing in the channel to use
        queue = rabbitpy.Queue(channel, 'example')
```

Once the object has been created and the instance returned as the `queue` variable, you can send the `Queue.Declare` command to RabbitMQ using the `declare` method. What you should see is an output line that has a Python tuple data structure with the number of messages in the queue and the number of consumers for the queue. A tuple is an immutable set of Python objects. In this case they are integer values.

```
In [8]: # Declare the queue on the RabbitMQ server
        queue.declare()
Out[8]: (10, 0)
```

Now that the queue has been created, you must bind it in order for it to receive messages. To bind the queue to the exchange, send the `Queue.Bind` command by invoking the queue object's `bind` method, passing in the exchange and the routing key. In the following example, the routing key is `example-routing-key`. When the execution of this cell returns, you should see the output `True`, indicating that the binding was successful.

```
In [9]: # Bind the queue to the exchange on the RabbitMQ server
        queue.bind(exchange, 'example-routing-key')
Out[9]: True
```

In your application, I recommend that you use semantically appropriate period-delimited keywords to namespace your routing keys. The *Zen of Python* states that "Namespaces are one honking great idea—let's do more of those!" and this is true in RabbitMQ as well. By using period-delimited keywords, you'll be able to route messages based upon patterns and subsections of the routing key. You'll learn more about this in chapter 6.

> **TIP** Queue and exchange names, along with routing keys, can include Unicode characters.

With your exchange and queue created and bound, you can now publish test messages into RabbitMQ that will be stored in the `example` queue. To make sure you have enough messages to play with, the following example publishes 10 test messages into the queue.

```
In [10]: for message_number in range(0, 10):
             message = rabbitpy.Message(channel,
                                        'Test message #%i' % message_number,
                                        {'content_type': 'text/plain'},
                                        opinionated=True)
             message.publish(exchange, 'example-routing-key')
```

To publish test messages, a new `rabbitpy.Message` object is created in each loop iteration, passing in the channel, a message body, and a dictionary of message properties. Once the message is created, the `publish` method is invoked, creating the `Basic.Publish` method frame, the content header frame, and one body frame, and delivering them all to RabbitMQ.

> **TIP** When you write publishers for your production environment, use a data serialization format such as JSON or XML so that your consumers can easily deserialize the messages and so they're easier to read when you're troubleshooting any problems that may arise.

You should now go to the RabbitMQ web management console and see if your messages made it into the queue: Open your web browser and visit the management UI at http://localhost:15672/#/queues/%2F/example (if your broker is on a different machine, change *localhost* in the URL to the appropriate server). Once authenticated, you should see a page resembling the screenshot in figure 2.15.

If you look toward the bottom of the page, you'll see a Get Messages section. If you change the Messages field value from 1 to 10 and click Get Messages, you should see each of the 10 messages you previously published. Make sure you leave the Requeue field value set to Yes. It tells RabbitMQ to add the messages back into the queue when RabbitMQ retrieves them for display in the management UI. If you didn't, don't worry; just go back and rerun the publishing code.

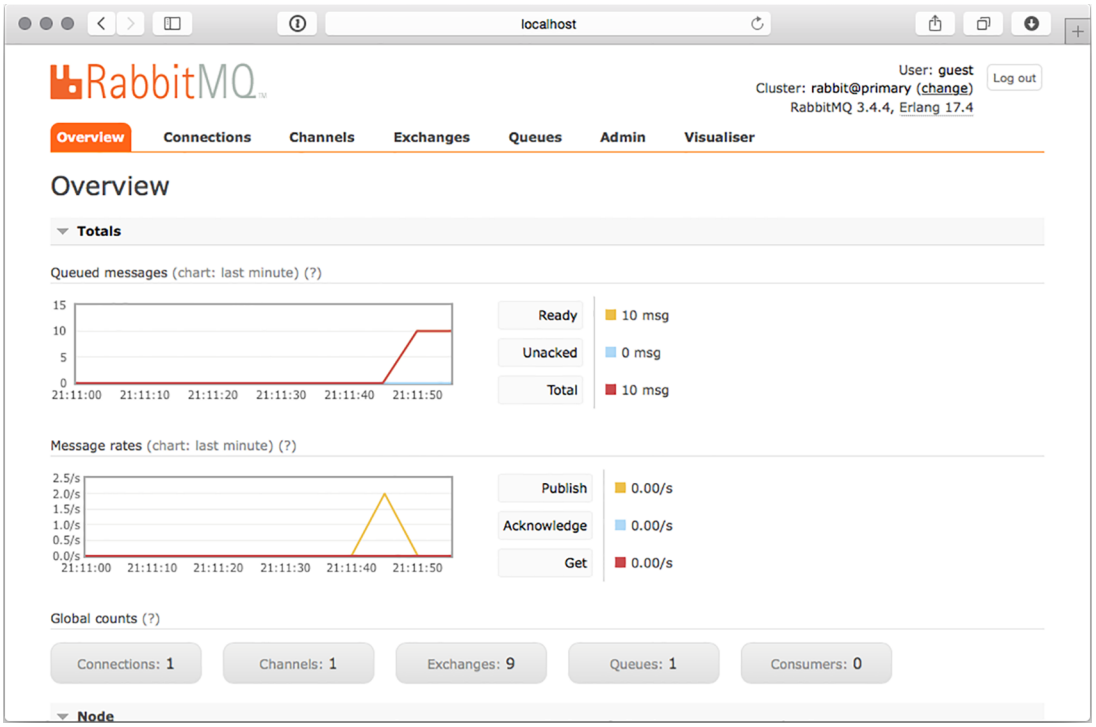


Figure 2.15 The RabbitMQ web management UI showing 10 messages in the order-processing queue.

2.5 Getting messages from RabbitMQ

Now that you know how to publish messages, it's time to retrieve them. The following listing pulls together the repetitive, yet import, connection elements from the publishing code discussed in the last section, allowing you to get messages from RabbitMQ. This code is in the "2.5 Basic.Get Example" notebook. This notebook has six cells in it when using the IPython Notebook interface. You can click the Cell dropdown and then Run All instead of running each cell as in the previous example.

```
import rabbitpy

url = 'amqp://guest:guest@localhost:5672/%2F'
connection = rabbitpy.Connection(url)
channel = connection.channel()
queue = rabbitpy.Queue(channel, 'example')

while len(queue) > 0:
  message = queue.get()
  print 'Message:'
```

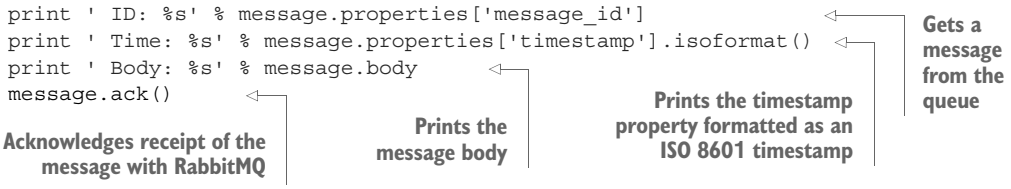
Creates a new connection object, connecting to RabbitMQ

Opens a channel to communicate on

Creates a new queue object for getting messages with

Loops while there are messages in the queue

Retrieves the message



After typing in and executing the preceding consumer code, you should see each of the 10 messages you previously published. If you were looking closely, you may have noticed that although you didn't specify the `message_id` or `timestamp` properties when publishing the messages, each message printed from the consumer has them. The rabbitpy client library will automatically populate these properties for you if you don't specify them. In addition, had you sent a Python `dict` as the message, rabbitpy would automatically serialize the data as JSON and set the content-type property as `application/json`.

2.6 Summary

The AMQP 0.9.1 specification defines a communication protocol that uses RPC-style commands to communicate between the RabbitMQ server and client. Now that you know how these commands are framed and how the protocol functions, you should be better equipped for writing and troubleshooting applications that interact with RabbitMQ. You've already covered a large majority of the process of communicating with RabbitMQ for publishing and consuming messages. Many applications contain little more code than what you've already implemented to work with your RabbitMQ instance.

In the next chapter you'll learn even more about using message properties, allowing your publishers and consumers to use a common contract for the messages your applications exchange.

SOFTWARE DEVELOPMENT

RabbitMQ IN DEPTH

Gavin M. Roy • Technical Editor James Titcumb

At the heart of most modern distributed applications is a queue that buffers, prioritizes, and routes message traffic. RabbitMQ is a high-performance message broker based on the Advanced Message Queueing Protocol. It's battle tested, ultrafast, and powerful enough to handle anything you can throw at it. It requires a few simple setup steps, and you can instantly start using it to manage low-level service communication, application integration, and distributed system message routing.

RabbitMQ in Depth is a practical guide to building and maintaining message-based applications. This book provides detailed coverage of RabbitMQ with an emphasis on why it works the way it does. You'll find examples and detailed explanations based in real-world systems ranging from simple networked services to complex distributed designs. You'll also find the insights you need to make core architectural choices and develop procedures for effective operational management.

What's Inside

- AMQP, the Advanced Message Queueing Protocol
- Communicating via MQTT, Stomp, and HTTP
- Valuable troubleshooting techniques
- Database integration

Written for programmers with a basic understanding of messaging-oriented systems.

Gavin M. Roy is an active, open source evangelist and advocate who has been working with internet and enterprise technologies since the mid-90s. Technical editor **James Titcumb** is a freelance developer, trainer, speaker, and active contributor to open source projects.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/rabbitmq-in-depth

"An excellent resource for beginners and experts alike ... shows how to integrate RabbitMQ into a successful enterprise application."
—Ian Dallas, Hewlett-Packard

"The most comprehensive source for everything RabbitMQ. From terms to code to patterns, it's all here!"
—Andrew Meredith
Quantum Metric

"A cheat sheet for getting started and troubleshooting the migration process to RabbitMQ."
—Nadia Noori
La Salle University Barcelona

"Filled with pragmatic advice and pearls of wisdom."
—Miloš Milivojević, Mozzart Bet

ISBN-13: 978-1-61729-100-5
ISBN-10: 1-61729-100-5
55999
9 781617 291005

MANNING $59.99 / Can $79.99 [INCLUDING eBOOK]