Community Experience Distilled

Build and optimize efficient messaging applications with ease

Learning RabbitMQ

RabbitMQ is an open source message broker and an ideal candidate for large-scale projects ranging from e-commerce and finance to big data and social networking.

Learning RabbitMQ starts with a concise description of a number of message patterns along with their implementation in RabbitMQ. This is followed by a number of practical examples and tips on administration and management of the broker. The knowledge is further expanded by exploring how to establish clustering and high availability at the level of the message broker and how to integrate RabbitMQ with a number of technologies, such as Spring, and ESBs such as MuleESB and WSO2. The book dives into advanced topics such as performance tuning, troubleshooting, secure messaging, and the internals of RabbitMQ. Along the way we use a demo project called Corporate Social Network (CSN) to illustrate the various concepts being discussed.

Who this book is written for

If you are a developer or system administrator with a basic knowledge of messaging who wants to learn RabbitMQ, or if you want to further enhance your knowledge of working with a message broker, then this book is ideal for you. To fully understand some examples in the book, a basic knowledge of the Java programming language is required.

What you will learn from this book

- Apply messaging patterns using the message broker
- Administer RabbitMQ using the command line, the management web console, or management REST services
- Create a cluster of scalable and highly-available RabbitMQ instances
- Use RabbitMQ with Spring Framework, MuleESB, WSO2, and Oracle database
- Deploy RabbitMQ using Puppet, Vagrant, or Docker
- Fine-tune the performance of RabbitMQ
- Monitor RabbitMQ using Nagios, Munin, or Munin
- Secure, troubleshoot, and extend RabbitMQ


$ 44.99 US
£ 28.99 UK

Prices do not include local sales tax or VAT

Martin Toshev

Learning RabbitMQ

PUBLISHING

PACKT open source

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Design Patterns with RabbitMQ'
- A synopsis of the book’s content
- More information on Learning RabbitMQ
Martin Toshev is a software developer and Java enthusiast with more than eight years of experience and vast expertise originating from projects in areas such as enterprise Java, social networking, source code analysis, Internet of Things, and investment banking in companies such as Cisco and Deutsche Telekom. He is a graduate of computer science from the University of Sofia. He is also a certified Java professional (SCJP6) and a certified IBM cloud computing solution advisor. His areas of interest include a wide range of Java-related technologies (Servlets, JSP, JAXB, JAXP, JMS, JMX, JAX-RS, JAX-WS, Hibernate, Spring Framework, Liferay Portal, and Eclipse RCP), cloud computing technologies, cloud-based software architectures, enterprise application integration, and relational and NoSQL databases. Martin is one of the leaders of the Bulgarian Java Users group (BGJUG), a regular speaker at Java conferences, and one of the organizers behind the jPrime conference in Bulgaria (http://jprime.io/).
Preface

Learning RabbitMQ provides you with a practical guide for the notorious message broker and covers the essentials required to start using it. The reader is able to build up knowledge along the way – starting from the very basics (such as what is RabbitMQ and what features does it provide) and reaching the point where more advanced topics, such as RabbitMQ troubleshooting and internals, are discussed. Best practices and important tips are provided in a variety of scenarios; some of them are related to external systems that provide integration with the message broker or that are integrated as part of the message broker in the form of a RabbitMQ plugin. Practical examples are also provided for most of these scenarios that can be applied in a broader context and used as a good starting point.

An example system called CSN (Corporate Social Network) is used to illustrate the various concepts provided throughout the chapters.

Each chapter ends with an Exercises section that allows the reader to test his understanding on the presented topic.

What this book covers

Chapter 1, Introducing RabbitMQ, provides you with a brief recap on enterprise messaging and a short overview of RabbitMQ along with its features. Other similar technologies are mentioned and an installation guide for the message broker is provided at the end of the chapter. The basic terminology behind RabbitMQ such as exchanges, queues, and bindings is introduced.

Chapter 2, Design Patterns with RabbitMQ, discusses what messaging patterns can be implemented using RabbitMQ, including point-to-point, publish-subscribe, request-reply, and message router types of communication. The patterns are implemented using the building blocks provided by the message broker and using the Java client API.
Chapter 3, Administration, Configuration and Management, reveals how to administer and configure RabbitMQ instances, how to install and manage RabbitMQ plugins, and how to use the various utilities provided as part of the RabbitMQ installation in order to accomplish a number of administrative tasks. A brief overview of the RabbitMQ management HTTP API is provided.

Chapter 4, Clustering, discusses what built-in clustering support is provided in the message broker and how it can be used to enable scalability in terms of message queues. A sample RabbitMQ cluster is created in order to demonstrate how nodes can be added/removed from a cluster and how RabbitMQ clients can connect to the cluster.

Chapter 5, High Availability, extends on the concepts of clustering by providing an overview of how a RabbitMQ cluster can be made more reliable in terms of mirrored queues and how messages can be replicated between remote instances using the Federation and Shovel plugins. High availability in terms of client connections and reliable delivery is also discussed with AMQP transactions, publisher confirms, and client reconnections.

Chapter 6, Integrations, provides you with a number of practical scenarios for integration of the message broker with the Spring framework, with ESB (enterprise services bus) systems such as MuleESB and WS02, and with database management systems (RDBMS and NoSQL). Deployment options for RabbitMQ using systems such as Puppet, Docker, and Vagrant are discussed in the chapter. A brief overview of how RabbitMQ applications can be tested using third-party frameworks is provided at the end of the chapter.

Chapter 7, Performance Monitoring and Tuning, gives a detailed list of factors that must be considered in terms of performance tuning of the message broker. The PerfTest tool is used to demonstrate how the RabbitMQ performance can be tested. At the end of the chapter, several monitoring solutions that provide support for RabbitMQ such as Nagios, Munin, and Monit are used to demonstrate how the message broker can be monitored in terms of stability and performance.

Chapter 8, Troubleshooting, illustrates a number of problems that can occur during the startup of the message broker and normal operation along with the various causes and resolutions in such cases. A brief primer on the Erlang programming language is provided for the purpose of understanding and analyzing the RabbitMQ crash dump—either directly or using the Crashdump Viewer for convenience.

Chapter 9, Security, provides a high-level overview of the vulnerability landscape related to the message broker along with a number of techniques to secure a RabbitMQ setup. Authentication, authorization, and secure communication are among the most important concepts covered in the chapter.
Chapter 10, Internals, discusses the internal architecture of the message broker and provides a detailed overview on the most important components that RabbitMQ comprises of.

Appendix A, Contributing to RabbitMQ, provides a short guide on how to get the RabbitMQ sources, how to set up a development environment, and how to build the message broker. A short discussion on how to contribute to the RabbitMQ ecosystem is provided as part of the appendix.
Design Patterns with RabbitMQ

As a robust messaging solution, RabbitMQ provides different utilities for distributing messages between endpoints in the communication channel. These utilities provide an implementation of best practices and design patterns that apply to messaging solutions and form the backbone of a messaging broker such as RabbitMQ.

Topics covered in the chapter:

- Messaging patterns in RabbitMQ
- Point-to-point communication
- Publish-subscribe communication
- Request-reply communication
- Message router

Messaging patterns in RabbitMQ

Messaging patterns in RabbitMQ are implemented based on exchanges, queues, and the bindings between them. We can distinguish between the different approaches for implementing a design pattern with RabbitMQ:

- For point-to-point communication between the publisher and the broker you can use a default or a direct exchange in order to deliver a message to a single queue. However, note that there might be multiple subscribers to this single queue, thus implementing publish-subscribe between the broker and the message receivers bound to that queue.
• For publish-subscribe, we can use a fanout exchange, which will deliver a message from an exchange to all queues that are bound to this exchange; in this manner, we may have a queue-per-subscriber strategy for implementing publish-subscribe.

• For request-response communication, we can use two separate exchanges and two queues; the publisher sets a message identifier in the message header and sends the request message to the request exchange, which in turn delivers the message to the request queue. The subscriber retrieves the message from the request queue, processes it, and sends a response message to the response exchange by also setting the same message identifier found in the request message to the response message header. The response exchange then delivers the message to a response queue. The publisher is subscribed to a response queue in order to retrieve response messages and uses the message identifier from the response message header to map the response message to the corresponding request message.

• For message routing we can use a topic exchange in order to deliver messages based on a binding key pattern or a headers exchange based on one or more headers.

It is important to remember that AMQP 0-9-1 protocol messages are load-balanced between consumers in a round-robin fashion. In this case, if there are multiple consumers on a message queue (bound using the `basic.consume` AMQP protocol command) then only one of them will receive the message, signifying that we have competing consumers. The same applies for the `basic.get` AMQP protocol command that retrieves a message from a queue on-demand (pull style) rather than by consumption (push style). If a message arrives on a queue that has no subscribers then the message will stay in the queue until a new subscriber is bound to the queue or the message is explicitly requested using `basic.get`. A message can also be rejected using the `basic.reject` AMQP protocol command. We will illustrate each of the preceding message patterns with concrete examples in subsequent sections. Before trying out the examples, you have to include the AMQP client library for Java. If you are using Maven, you can include the following dependencies for the client library along with the `slf4j` dependencies that provide the `slf4j` logging features used to provide logging capabilities in the examples:

```xml
<dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>3.4.1</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
</dependency>
```
In order to send messages to RabbitMQ, the Sender class will be used:

```java
import java.io.IOException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;

public class Sender {
    private final static String QUEUE_NAME = "event_queue";
    private final static Logger LOGGER = LoggerFactory.getLogger(Sender.class);
    private static final String DEFAULT_EXCHANGE = "";
    private Channel channel;
    private Connection connection;
}
```

The initialize() method is used to initialize the message sender by doing the following:

- Creating a ConnectionFactory that is used to create AMQP connections to a running RabbitMQ server instance; in this case, this is an instance running on localhost and accepting connections on the default port (5672)
- Creating a new connection using the connection factory
Creating a new channel for sending messages in the created connection:

```java
public void initialize() {
    try {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        connection = factory.newConnection();
        channel = connection.createChannel();
    } catch (IOException e) {
        LOGGER.error(e.getMessage(), e);
    }
}
```

The `send()` method has two variants: one that accepts a message and sends it to the default queue and a second one that accepts an exchange name, exchange type, and the message to send. The first variant is appropriate for point-to-point communication and does the following:

- Declares a queue in the message broker using the `queueDeclare()` method; if the queue is already created then it is not recreated by the method
- Publishes a message on the default exchange that is delivered to that queue

The second variant of `send()` is appropriate for the publish-subscribe type of communication and does the following:

- Declares the specified exchange along with its type on the message bus using the `exchangeDeclare()` method; the exchange is not recreated if it exists on the message bus
- Sends a message to this exchange with a routing key equal to the empty string (we are indicating that we will not use the routing key with this variant of the method):

```java
public void send(String message) {
    try {
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        channel.basicPublish(DEFAULT_EXCHANGE, QUEUE_NAME, null,
                             message.getBytes());
    } catch (IOException e) {
        LOGGER.error(e.getMessage(), e);
    }
}
```

```
public void send(String exchange, String type, String message) {
```
try {
    channel.exchangeDeclare(exchange, type);
    channel.basicPublish(exchange, "", null,
                         message.getBytes());
} catch (IOException e) {
    LOGGER.error(e.getMessage(), e);
}

The `destroy()` method is used to close the connection and all connection channels to the message broker:

```java
public void destroy() {
    try {
        if (connection != null) {
            connection.close();
        }
    } catch (IOException e) {
        LOGGER.warn(e.getMessage(), e);
    }
}
```

**Point-to-point communication**

The following diagram provides an overview of the scenario that we will implement:

![Diagram](image)

For point-to-point communication, the sender can use either the default exchange or a direct exchange (that uses the routing key to determine to which queue a message must be sent; the routing key should match the binding key between the exchange and the queue). The `CompetingReceiver` class can be used to subscribe to a particular queue and receive messages from that queue:

```java
import java.io.IOException;
import org.slf4j.Logger;
```
import org.slf4j.LoggerFactory;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.ConsumerCancelledException;
import com.rabbitmq.client.QueueingConsumer;
import com.rabbitmq.client.ShutdownSignalException;

public class CompetingReceiver {

    private final static String QUEUE_NAME = "event_queue";
    private final static Logger LOGGER = LoggerFactory.getLogger(Sender.class);
    private Connection connection = null;
    private Channel channel = null;
    public void initialize() {
        try {
            ConnectionFactory factory =
                new ConnectionFactory();
            factory.setHost("localhost");
            connection = factory.newConnection();
            channel = connection.createChannel();
        } catch (IOException e) {
            LOGGER.error(e.getMessage(), e);
        }
    }
}

The receive() method is used to receive a message from the queue named event_queue by doing the following:

- Creating the event_queue in the message broker, if not already created, using the queueDeclare() method
- Creating a QueueingConsumer instance that is used as the handler for messages from the event_queue queue
• Registering the QueueingConsumer as a message consumer using the basicConsume() method of the Channel instance that represents the AMQP channel to the message broker

• Consuming a message from the event_queue queue using the nextDeliver() method of the QueueingConsumer instance, which blocks until a message arrives on the queue; QueueingConsumer.Delivery represents the received message:

```java
public String receive() {
    if (channel == null) {
        initialize();
    }
    String message = null;
    try {
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        QueueingConsumer consumer =
            new QueueingConsumer(channel);
        channel.basicConsume(QUEUE_NAME, true,
            consumer);
        QueueingConsumer.Delivery delivery =
            consumer.nextDelivery();
        message = new String(delivery.getBody());
        LOGGER.info("Message received: " + message);
        return message;
    } catch (IOException e) {
        LOGGER.error(e.getMessage(), e);
    } catch (ShutdownSignalException e) {
        LOGGER.error(e.getMessage(), e);
    } catch (ConsumerCancelledException e) {
        LOGGER.error(e.getMessage(), e);
    } catch (InterruptedException e) {
        LOGGER.error(e.getMessage(), e);
    }
    return message;
}
```
The `destroy()` method closes the AMQP connection and must be called explicitly when needed; closing the connection closes all AMQP channels created in that connection:

```java
public void destroy() {
    if (connection != null) {
        try {
            connection.close();
        } catch (IOException e) {
            LOGGER.warn(e.getMessage(), e);
        }
    }
}
```

In order to demonstrate the usage of the `CompetingConsumer` class in a point-to-point channel, we can use the `DefaultExchangeSenderDemo` class to send a message to the default exchange:

```java
public class DefaultExchangeSenderDemo {

    public static void sendToDefaultExchange() {
        Sender sender = new Sender();
        sender.initialize();
        sender.send("Test message.");
        sender.destroy();
    }

    public static void main(String[] args) {
        sendToDefaultExchange();
    }
}
```
When invoking the `main()` method, a message is sent to the RabbitMQ server instance running on localhost; if no instance is running then a `java.net.ConnectionException` is thrown from the client. Assuming that there are no defined queues yet in the message broker, if you open the RabbitMQ management console you will notice the following before invoking the `main()` method:

![RabbitMQ console](image)

After invoking the `main()` method, you will notice that the `event_queue` is created:

![RabbitMQ console](image)
Moreover, there is one unprocessed message in the queue; the Ready section gives the number of unprocessed messages on the particular queue. In order to consume the message CompetingReceiverDemo class, perform the following:

```java
public class CompetingReceiverDemo {

    public static void main(String[] args) throws InterruptedException {
        final CompetingReceiver receiver1 = new CompetingReceiver();
        receiver1.initialize();
        final CompetingReceiver receiver2 = new CompetingReceiver();
        receiver2.initialize();

        Thread t1 = new Thread(new Runnable() {
            public void run() {
                receiver1.receive();
            }
        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                receiver2.receive();
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        receiver1.destroy();
        receiver2.destroy();
    }
}
```

We create two CompetingReceiver instances and invoke the receive() methods of the two instances in separate threads so that we have two subscribers for the same queue waiting for a message. The two threads are joined to the main application thread so that method execution continues once both consumers receive a message from the queue. Since our queue already has one message, one of the two consumers will receive the message while the other will continue to wait for a message. If we invoke the main() method of the DefaultExchangeSenderDemo class once again, the other consumer will also receive a message from the queue and the main() method of CompetingReceiverDemo() will terminate.
Publish-subscribe communication

The following diagram provides an overview of the scenario that we will implement:

For publish-subscribers we can use a fanout exchange and bind any number of queues to that exchange regardless of the binding key. The PublishSubscribeReceiver class can be used to bind a specified queue to a fanout exchange and receive messages from it:

```java
public class PublishSubscribeReceiver {
    private final static String EXCHANGE_NAME = "pubsub_exchange";
    private final static Logger LOGGER = LoggerFactory.getLogger(Sender.class);
    private Channel channel = null;
    private Connection connection = null;

    public void initialize() {
        try {
            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost("localhost");
            connection = factory.newConnection();
```
channel = connection.createChannel();
} catch (IOException e) {
    LOGGER.error(e.getMessage(), e);
}
...

The `receive()` method can be used to retrieve a message from a queue that is bound to the `pubsub_exchange` fanout exchange and does the following:

- Creates the `pubsub_exchange`, if not already created
- Creates the specified queue if not already created
- Binds the queue to the `pubsub_exchange` using the `queueBind()` method of the Channel instance that represents the AMQP channel for the receiver; notice that in this case we don’t specify any particular binding key and for that reason we are using the empty string
- Creates a new `QueueingConsumer` instance, registered using the AMQP channel, and the `nextDelivery()` method is called to receive a message from the channel:

```java
public String receive(String queue) {
    if (channel == null) {
        initialize();
    }

    String message = null;
    try {
        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
        channel.queueDeclare(queue, false, false, false, null);
        channel.queueBind(queue, EXCHANGE_NAME, ");
        QueueingConsumer consumer = new QueueingConsumer(channel);
        channel.basicConsume(queue, true, consumer);
        QueueingConsumer.Delivery delivery = consumer.nextDelivery();
        message = new String(delivery.getBody());
        LOGGER.info("Message received: " + message);
        return message;
    } catch (IOException e) {
```
And we also have a `destroy()` method:

```java
public void destroy() {
    try {
        if (connection != null) {
            connection.close();
        }
    } catch (IOException e) {
        LOGGER.warn(e.getMessage(), e);
    }
}
```

In order to demonstrate the usage of `QueueingConsumer` for establishing a publish-subscribe communication channel, we will use the `FanoutExchangeSenderDemo` class to send a message to the `pubsub_exchange` fanout exchange:

```java
public class FanoutExchangeSenderDemo {

    private static final String FANOUT_EXCHANGE_TYPE = "fanout";

    public static void sendToFanoutExchange(String exchange) {
        Sender sender = new Sender();
        sender.initialize();
        sender.send(exchange, FANOUT_EXCHANGE_TYPE, "Test message.");
        sender.destroy();
    }

    public static void main(String[] args) {
        sendToFanoutExchange("pubsub_exchange");
    }
}
```
When you invoke the `main()` method of the `FanoutExchangeSenderDemo` class, you may notice from the management console that the `pubsub_exchange` exchange is created in the RabbitMQ server instance separate from the predefined exchanges:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Policy</th>
<th>Parameters</th>
<th>Message rate in</th>
<th>Message rate out</th>
</tr>
</thead>
<tbody>
<tr>
<td>(AMQP default)</td>
<td>direct</td>
<td>D</td>
<td></td>
<td>0.00/s</td>
<td>0.00/s</td>
</tr>
<tr>
<td>amq.direct</td>
<td>direct</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>amq.fanout</td>
<td>fanout</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>amq.headers</td>
<td>headers</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>amq.match</td>
<td>headers</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>amq.rabbitmq.log</td>
<td>topic</td>
<td>D</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>amq.rabbitmq.trace</td>
<td>topic</td>
<td>D</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>amq.topic</td>
<td>topic</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pubsub_exchange</td>
<td>fanout</td>
<td>D</td>
<td></td>
<td>0.00/s</td>
<td></td>
</tr>
</tbody>
</table>

If you restart the RabbitMQ instance then you will not see the `pubsub_exchange` from the management console again, because the exchange is not marked as durable. In order to mark a queue/exchange as durable, you can provide an additional parameter to the `queueDeclare()`/`exchangeDeclare()` methods of the `Channel` class. In order to provide further message delivery guarantees on the broker, you can use the publisher confirms of the extension.
The PublishSubscribeReceiverDemo class provides a demonstration of the PublishSubscribeReceiver class for the establishment of a publish-subscribe channel:

```java
public class PublishSubscribeReceiverDemo {

    public static void main(String[] args)
    throws InterruptedException {
        final PublishSubscribeReceiver receiver1 =
        new PublishSubscribeReceiver();
        receiver1.initialize();
        final PublishSubscribeReceiver receiver2 =
        new PublishSubscribeReceiver();
        receiver2.initialize();
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                receiver1.receive("pubsub_queue1");
            }
        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                receiver2.receive("pubsub_queue2");
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        receiver1.destroy();
        receiver2.destroy();
    }
}
```

The `main()` method creates two receivers that bind to two different queues: `pubsub_queue1` and `pubsub_queue2`. If you have already sent a message to the `pubsub_exchange` exchange, it will be delivered to both queues and thus sent to both consumers.
Request-reply communication

The following diagram provides an overview of the scenario that we will implement:

The sender will send a message to the default exchange with a routing key that matches the name of the designated request queue. The request receiver is a subscriber to the request queue. After a request message is received, the request receiver retrieves the value of the replyTo property from the message header, creates a response message, and sends it to the default exchange with a routing key that matches the replyTo property. This means that the replyTo property points to a queue that handles response messages and the sender is subscribed to that queue in order to receive a response.

Let’s extend our Sender class with the following sendRequest() method, which sends a message to the request_exchange exchange, and the receiveResponse() method, which receives a message from the response_queue queue as follows:

```java
private static final String REQUEST_QUEUE = "request_queue";
private static final String RESPONSE_QUEUE = "response_queue";
public void sendRequest(String requestQueue, String message, String correlationId) {
    try {
        channel.queueDeclare(REQUEST_QUEUE, false, false, false, null);
        channel.queueDeclare(RESPONSE_QUEUE, false, false, false, null);
        AMQP.BasicProperties amqpProps = new AMQP.BasicProperties();
        amqpProps = amqpProps.builder()
            .correlationId(String.valueOf(correlationId))
            .replyTo(RESPONSE_QUEUE).build();
        channel.basicPublish(DEFAULT_EXCHANGE, REQUEST_QUEUE, amqpProps, message.getBytes());
    } catch (IOException e) {
    }
```
public String waitForResponse(final String correlationId) {
    QueueingConsumer consumer = new QueueingConsumer(channel);
    String result = null;
    try {
      channel.basicConsume(RESPONSE_QUEUE, true, consumer);
      QueueingConsumer.Delivery delivery = consumer.nextDelivery(3000);
      String message = new String(delivery.getBody());
      if (delivery.getProperties() != null) {
        String msgCorrelationId = delivery.getProperties()
          .getCorrelationId();
        if (!correlationId.equals(msgCorrelationId)) {
          LOGGER.warn("Received response of another request.");
        } else {
          result = message;
        }
      }
      LOGGER.info("Message received: " + message);
    } catch (IOException e) {
      LOGGER.error(e.getMessage(), e);
    } catch (ShutdownSignalException e) {
      LOGGER.error(e.getMessage(), e);
    } catch (ConsumerCancelledException e) {
      LOGGER.error(e.getMessage(), e);
    } catch (InterruptedException e) {
      LOGGER.error(e.getMessage(), e);
    }
    return result;
  }

The sendRequest() method crafts an AMQP.BasicProperties instance and provides the replyTo and correlationId properties. The correlationId must be a unique identifier that is passed back in the response message and can be used by the sender to determine the request for which a response is received.
The RequestReceiver class provides a sample implementation of a request receiver:

```java
public class RequestReceiver {
    private static final String DEFAULT_QUEUE = "";
    private static final String REQUEST_QUEUE = "request_queue";
    private final static Logger LOGGER = LoggerFactory.getLogger(Sender.class);
    private Connection connection = null;
    private Channel channel = null;

    public void initialize() {
        try {
            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost("localhost");
            connection = factory.newConnection();
            channel = connection.createChannel();
        } catch (IOException e) {
            LOGGER.error(e.getMessage(), e);
        }
    }

    public void receive() {
        if (channel == null) {
            initialize();
        }

        String message = null;
        try {
            channel.queueDeclare(REQUEST_QUEUE, false, false, false, null);
            QueueingConsumer consumer = new QueueingConsumer(channel);
            channel.basicConsume(REQUEST_QUEUE, true, consumer);
            QueueingConsumer.Delivery delivery = consumer.nextDelivery();
            message = new String(delivery.getBody());
        }
    }
}
```

The receive() method is used to read a request message from a queue:
LOGGER.info("Request received: " + message);

// do something with the request message ...

BasicProperties properties = delivery.getProperties();
if (properties != null) {
    AMQP.BasicProperties amqpProps =
    new AMQP.BasicProperties();
    amqpProps = amqpProps.builder().correlationId(
        String.valueOf(properties.getCorrelationId())).build();
    channel.basicPublish(DEFAULT_QUEUE,
        properties.getReplyTo(), amqpProps, "Response message.".getBytes());
} else {
    LOGGER.warn("Cannot determine response
destination for message.");
}

} catch (IOException e) {
    LOGGER.error(e.getMessage(), e);
} catch (ShutdownSignalException e) {
    LOGGER.error(e.getMessage(), e);
} catch (ConsumerCancelledException e) {
    LOGGER.error(e.getMessage(), e);
} catch (InterruptedException e) {
    LOGGER.error(e.getMessage(), e);
}

})

And again we have a destroy() method – it is important to make sure that you
close your connections to the broker if you are no longer using them:

public void destroy() {
    if (connection != null) {
        try {
            connection.close();
        } catch (IOException e) {
            LOGGER.warn(e.getMessage(), e);
        }
    }
}

}
In order to send a request message we can use the `RequestSenderDemo` class:

```java
public class RequestSenderDemo {

    private static final String REQUEST_QUEUE = "request_queue";

    public static String sendToRequestReplyQueue() {
        Sender sender = new Sender();
        sender.initialize();
        sender.sendRequest(REQUEST_QUEUE, "Test message.", "MSG1");
        String result = sender.waitForResponse("MSG1");
        sender.destroy();
        return result;
    }

    public static void main(String[] args) {
        sendToRequestReplyQueue();
    }
}
```

In order to receive the request message and send a response message, you can use the `RequestReceiverDemo` class:

```java
public class RequestReceiverDemo {

    public static void main(String[] args) throws InterruptedException {
        final RequestReceiver receiver = new RequestReceiver();
        receiver.initialize();
        receiver.receive();
        receiver.destroy();
    }
}
```
Message router

The following diagram provides an overview of the scenario that we will implement:

Let's say we have a service that triggers an event upon the creation of a new programming seminar, or hackathon, for a given community. We want to send all seminar events to a particular destination receiver and all hackaton events to another destination receiver. Moreover, we want to send messages to the same exchange. For that setup, a topic exchange is a rational choice; one queue will be bound to the topic exchange with the seminar.# routing key and another queue will be bound with hackaton.# routing key. The # character is special and serves as a pattern that matches any character sequence.

We can implement this type of message sending by further extending our Sender class:

```java
private static final String SEMINAR_QUEUE = "seminar_queue";
private static final String HACKATON_QUEUE = "hackaton_queue";
private static final String TOPIC_EXCHANGE = "topic_exchange";

public void sendEvent(String exchange, String message, String messageKey) {
    try {
        channel.exchangeDeclare(TOPIC_EXCHANGE, "topic");
        channel.queueDeclare(SEMINAR_QUEUE, false, false, false, null);
        channel.queueDeclare(HACKATON_QUEUE, false, false, false, null);
        channel.queueBind(SEMINAR_QUEUE, TOPIC_EXCHANGE, "seminar.#");
        channel.queueBind(HACKATON_QUEUE, TOPIC_EXCHANGE, "hackaton.#");
        channel.basicPublish(TOPIC_EXCHANGE, messageKey, null,
```
In order to demonstrate event sending, we can use the TopicSenderDemo class:

```java
public class TopicSenderDemo {

    private static final String TOPIC_EXCHANGE = "topic_exchange";

    public static void sendToTopicExchange() {
        Sender sender = new Sender();
        sender.initialize();
        sender.sendEvent(TOPIC_EXCHANGE, "Test message 1.", "seminar.java");
        sender.sendEvent(TOPIC_EXCHANGE, "Test message 2.", "seminar.rabbitmq");
        sender.sendEvent(TOPIC_EXCHANGE, "Test message 3.", "hackaton.rabbitmq");
        sender.destroy();
    }

    public static void main(String[] args) {
        sendToTopicExchange();
    }
}
```
Case study: Initial design of the CSN

The following diagram extends the general overview of a CSN in regard to a client browser that provides client-side interaction with the system:

Now that we have seen how to implement messaging patterns in RabbitMQ, we can apply this to implement the following:

- Global event handling; we can use the default exchange along with a single queue called `event_queue`. The worker nodes as illustrated in the preceding diagram will subscribe to the `event_queue` and start handling events for long-running tasks in a round-robin fashion; the `CompetingReceiver` class is a proper alternative for the implementation of a point-to-point receiver on the worker nodes.

- Chat service; each user of the system will have a separate queue that will receive messages for that queue. You can use a variant—a point-to-point channel—to send a message from one user to the other. For group chatting, you can have a fanout or topic exchange (based on the implementation strategy) for the particular group that will be used to deliver messages to all use queues.
To implement a chat client that is displayed in the client's browser you have a number of alternatives, such as:

- Using the WebSocket protocol, since it allows two-way communication between the browser and the CSN frontend server; the frontend server sends the message to the RabbitMQ server for further handling. In this case, you may need to create a mapping between the WebSocket endpoints and AMQP queues.
- Implementing a browser plugin that makes use of the AMQP protocol directly; this allows you to connect clients directly to the RabbitMQ broker.
- Ajax requests with long polling; this option is not preferred since it implies a heavy footprint on network bandwidth but it is still another alternative.

Summary

In this chapter, we saw how to implement various messaging patterns in RabbitMQ. We also discussed how to design the various components of a CSN (Corporate Social Network) that makes use of such messaging patterns, with RabbitMQ as the message broker used in the system. In the next chapter we will see how to configure and administer RabbitMQ.

Exercises

1. How can you implement different enterprise integration patterns with RabbitMQ other than the ones listed in this chapter? Refer to the book Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf.
2. Can you think of any non-standard applications of RabbitMQ in CSN? List them and think of a general design for implementing them in CSN.
Where to buy this book

You can buy Learning RabbitMQ from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.