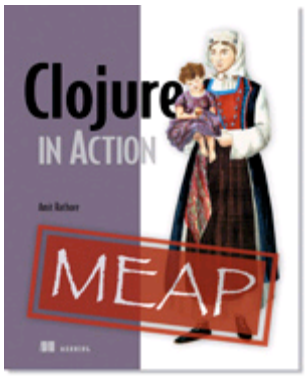
MANNING PUBLICATIONS

Clojure and RabbitMQ

An article from



[Clojure in Action](#)
EARLY ACCESS EDITION

Amit Rathore
MEAP Release: November 2009
Softbound print: Early 2011 | 475 pages
ISBN: 9781935182597

This article is taken from the book Clojure in Action. *The author explains the AMPQ protocol basics and shows how to send and receive messages over RabbitMQ.*

In this article, we're going to write code for some basic communication tasks between Clojure processes and RabbitMQ. RabbitMQ uses the AMQP protocol, so a basic understanding of that protocol will help you get the most out of RabbitMQ. Therefore, our first stop will be to review essential elements of AMQP. We'll then use the Java client library to send and receive messages over RabbitMQ. Finally, we'll create a convenient abstraction over incoming RabbitMQ messages for our programs that need to process messages asynchronously.

Which version to use?

The RabbitMQ project is evolving at a steady pace. In fact, the project was recently acquired by SpringSource, which is part of VMWare. The version of RabbitMQ we're going to be working with is 1.7.2 and can be downloaded from www.rabbitmq.com.

AMQP basics

Let's look at a few key concepts of RabbitMQ, which are reflective of the AMQP protocol underneath. Specifically, it is important to understand the basic concepts of the message queue, the exchange, and the routing key. Once these topics are understood, it becomes quite easy to understand the message flow and how message passing should be configured in a given application.

Message queue

A message queue stores messages in memory or on a disk and delivers these to various consumers (usually) in the order they come in. Message queues can be created to have a varying set of properties—private or shared, durable or not, permanent or temporary. The right combination of these properties can result in the various commonly used patterns of message queues—store and forward, temporary reply queue, and the classic publish/subscribe model. Message queues can be named so that clients can refer to the specific ones they created during the routing configuration process.

Exchange

The exchange is a sort of clearinghouse that accepts the incoming messages and then, based on certain specified criteria, routes them to specific message queues. The criteria are called bindings and are specified by the clients themselves.

AMQP defines a few exchange types, such as direct and fanout exchanges. By naming both exchanges and queues, clients can configure them to work with each other.

Routing key

An exchange can route messages to message-queues based on a variety of message properties. The most common way, though, is dependent on a single property called the routing key. It can be thought of as a virtual address that the exchange can use to route messages to the queue.

Now that we've seen the fundamental constructs of AMQP, we're ready to start playing with a server that implements it. In the next section, we'll begin this using RabbitMQ.

Connecting to RabbitMQ

We'll now see how to connect to a RabbitMQ server. Further, we'll abstract away the idea of a connection to the server by creating a var for that purpose. First off, here's the code to create a new connection:

```
(ns chapter14-rabbitmq
  (:import (com.rabbitmq.client ConnectionParameters
                                ConnectionFactory QueueingConsumer)))

 (defn new-connection [q-host q-username q-password]
  (let [params (doto (ConnectionParameters.)
                 (.setVirtualHost "/")
                 (.setUsername q-username)
                 (.setPassword q-password))]
    (.newConnection (ConnectionFactory. params) q-host)))
```

Now for the var—let's create one called `*rabbitmq-connection*` as follows:

```
(def *rabbit-connection*)
```

We can bind this to a call to `new-connection` whenever we need to do something with RabbitMQ. In order to make this easy, we can create a macro to do it for us. One might call such a macro `with-rabbit` and it might look like this:

```
(defmacro with-rabbit [[mq-host mq-username mq-password] & exprs]
  `(with-open [connection# (new-connection ~mq-host
                                           ~mq-username ~mq-password)]
     (binding [*rabbit-connection* connection#]
       (do ~@exprs))))
```

Note that, by using the `with-open` macro, the connection held in `*rabbit-connection*` gets closed when we exit the `with-rabbit` form. Now, with the `with-rabbit` macro in hand, we're ready to write more code to do things like sending messages to the server and receiving messages from it. The next couple of sections show how to do just that.

Sending messages over RabbitMQ

As described in the section on AMQP, the physical manifestation of a connection to RabbitMQ is the Channel. Therefore, the first thing we'll need to do is get hold of a channel. Here's the function that sends a message to the server, given a specific routing key:

```
(defn send-message [routing-key message-object]
  (with-open [channel (.createChannel *rabbit-connection*)]
    (.basicPublish channel "" routing-key nil
                                  (.getBytes (str message-object)))))
```

In order to send a message through the RabbitMQ server, we can now just call our newly defined `send-message` function, like so:

```
(send-message "chapter14-test" "chapter 14 test method")
```

Our next task is to receive such messages from the server. We will do this in the next section.

Receiving messages from RabbitMQ

We're going to take a stab at a simple function that just accepts a queue-name (which, in this case must match the routing key used to send messages out) and listens for messages addressed to it. Once we have this function, we'll put the send-message function to use in order to test our capability to receive messages by printing it to the console. Once we have this working, we'll see how to write a longer-living process that handles a stream of incoming messages.

Let's begin with our function to fetch a message from the RabbitMQ server. Consider this implementation where `next-message-from` is our final, outward facing function:

```
(defn delivery-from [channel consumer]                                  #A
  (let [delivery (.nextDelivery consumer)]
    (.basicAck channel (.. delivery getEnvelope getDeliveryTag) false)
    (String. (.getBody delivery))))

(defn consumer-for [channel queue-name]
  (let [consumer (QueueingConsumer. channel)]
    (.queueDeclare channel queue-name)                                  #B
    (.basicConsume channel queue-name consumer)
    consumer))

(defn next-message-from [queue-name]
  (with-open [channel (.createChannel *rabbit-connection*)]             #C
    (let [consumer (consumer-for channel queue-name)]
      (delivery-from channel consumer))))
```

#A Acknowledge after processing in production!
#B Declare queue even if it exists
#C Declare vars without root-binding

What's happening here is that we create an instance of `QueueingConsumer`, which is a class provided by the RabbitMQ client library. We then declare the queue-name we were passed in, following which we attach the `consumer` to the queue-name via the call to `basicConsume`. The `delivery-from` function calls the `nextDelivery` method on the `consumer` object, and that is a blocking call. The execution proceeds only when the consumer receives something from RabbitMQ, and then all we do is acknowledge the message by calling `basicAck` and then returning the contents of the message as a string.

We can confirm that this works by starting the following program up as a Clojure script:

```
(ns chapter14-receiver
  (:use chapter14-rabbitmq))
```

```
(println "Waiting...")
(with-rabbit ["localhost" "guest" "guest"]
  (println (next-message-from "chapter14-test")))
```

This will cause the program to block (since next-message-from blocks until a message is delivered to it from the RabbitMQ server) and, in order to unblock it and see that it prints the message to the console, we have to send it a message. This following program, which can also be run as a script, does just that:

```
(ns chapter14-sender
  (:use chapter14-rabbitmq))

(println "Sending...")
(with-rabbit ["localhost" "guest" "guest"]
  (send-message "chapter14-test" "chapter 14 test method"))
(println "done!")
```

If you run the above two programs in two separate shells (run the receiver first!), you'll see our two functions—`send-message` and `next-message-from` in action. Now that we've got basic communication going over RabbitMQ, let's write a version of the receiver program that can handle multiple messages from the queue.

Receiving multiple messages

In typical projects involving messaging, a program like the one we wrote in the previous section might be useful for quick tests and debugging. Most production services that are meant to handle messages asynchronously essentially wait for such messages in a loop. These long-running processes can form the basis of a compute cluster that can offload work from services that must respond to the end-user quickly. Let's rewrite the earlier program to do its job in a loop:

```
(ns chapter14-receiver-multiple1
  (:use chapter14-rabbitmq))

(defn print-multiple-messages []
  (loop [message (next-message-from "chapter14-test")]
    (println "Message: " message)
    (recur (next-message-from "chapter14-test"))))

(with-rabbit ["localhost" "guest" "guest"]
  (println "Waiting for messages...")
  (print-multiple-messages))
```

This program can also be tested in a similar fashion to our previous example. When you run this program, it will block, waiting for the first message to be delivered to it by the server. We can use our message sender program from the previous section to send it a few messages. Output might look like this:

```
Waiting for messages...
Message:  chapter 14 test method
Message:  chapter 14 test method
Message:  chapter 14 test method
```

Press `control-c` to exit this program.

This program works but is not particularly reusable since it mixes up the work of printing the incoming messages with the logic of waiting for messages in a loop. Let's fix that problem by creating a higher order function that will accept a function that knows how to handle a single message. Here's the code:

```
(ns chapter14-receiver-multiple2
  (:use chapter14-rabbitmq))

(defn handle-multiple-messages [handler]
  (loop [message (next-message-from "chapter14-test")]
    (handler message)
```

```
    (recur (next-message-from "chapter14-test"))))

(with-rabbit ["localhost" "guest" "guest"]
  (println "Waiting for messages...")
  (handle-multiple-messages println))
```

With this higher-order function called `handle-multiple-messages`, we can now do whatever we please with an incoming stream of messages. Indeed, if you run several instances of this program in parallel and send messages using the same sender program, you'll see RabbitMQ deliver messages to each in a roughly round-robin manner. As said earlier, this can form the basis of a compute cluster of some kind.

One thing to note is that our `next-message-from` function is quite inefficient. It creates a new channel each time it is called. If we know we're going to process multiple messages, we really should improve this state of affairs. Indeed, while we're at it, we should recognize the fact that an incoming sequence of messages can be modeled as a Clojure sequence. We'll do this in the next section.

message-seq: a sequence abstraction for receiving messages

If we consider the job of `next-message-from`, we can think of it producing an element of a lazy sequence each time a message is delivered to it. Using that as a foundation, we can create a new abstraction to deal with messages from the RabbitMQ server. We can call it `message-seq` and it might be implemented as follows:

```
(defn- lazy-message-seq [channel consumer]                          #A
  (lazy-seq
   (let [message (delivery-from channel consumer)]
     (cons message (lazy-message-seq channel consumer)))))

(defn message-seq [queue-name]
  (let [channel (.createChannel *rabbit-connection*)
        consumer (consumer-for channel queue-name)]
    (lazy-message-seq channel consumer)))
```

#A creating an infinite sequences of messages off RabbitMQ

Note that `lazy-message-seq` is a helper function, which is private to this namespace. The message-seq function is the one that we can use in our programs. Let's write a version of multiple message handler that uses it:

```
(ns chapter14-receiver-multiple3
  (:use chapter14-rabbitmq))

(defn handle-multiple-messages [handler]
  (doseq [message (message-seq "chapter14-test")]
    (handler message)))

(with-rabbit ["localhost" "guest" "guest"]
  (println "Waiting for messages...")
  (handle-multiple-messages println))
```

There are several advantages of this approach. The first is that it's more efficient since we're no longer creating a new channel (and consumer) for each message. Perhaps more importantly, since `message-seq` is a real Clojure sequence, we can use the full Clojure sequence library on it. The example above shows the usage of `doseq`, but we can now `map` across it, `filter` out only those messages that we like, and so on. Here's an example where we only print the messages in pairs:

```
(ns chapter14-receiver-multiple4
  (:use chapter14-rabbitmq clojure.contrib.str-utils))

(defn print-two-messages [messages]
  (println (str-join "::" messages)))

(with-rabbit ["localhost" "guest" "guest"]
  (println "Waiting for messages...")
```

```
(let [message-pairs (partition 2 (message-seq "chapter14-test"))]
  (doseq [message-pair message-pairs]
    (print-two-messages message-pair))))
```

We can test this by sending it messages using our sender program, as usual. The output of a test run might look like the following:

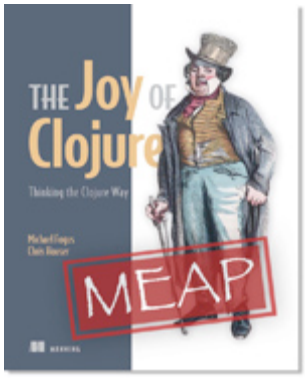
```
Waiting for messages...
chapter 14 test method::chapter 14 test method
chapter 14 test method::chapter 14 test method
```

The fact that we can now bring the full power of the Clojure sequence library to bear on a series of messages from RabbitMQ allows us to write idiomatic Clojure programs. No function outside the scope of `message-seq` needs to know that the sequence is lazily being fed by a RabbitMQ server, somewhere on the network.

Summary

We can use the code we wrote in this article as a basis for real applications to handle events in an asynchronous manner. Further, we can expand our event processing capacity simply by starting up more instances of our handlers. This idea forms the basis of using messaging to scale up applications.

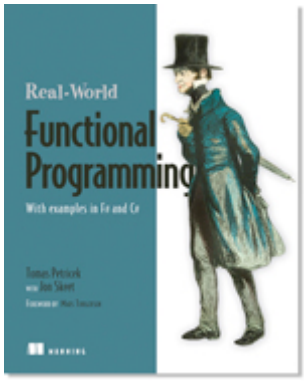
Here are some other Manning titles you might be interested in:



The Joy of Clojure

EARLY ACCESS EDITION

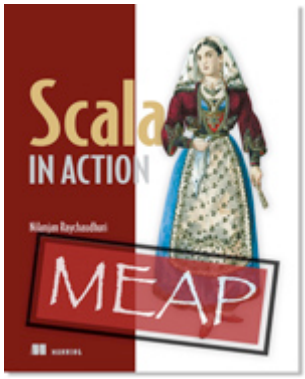
Michael Fogus and Chris Houser
MEAP Release: January 2010
Softbound print: December 2010 | 400 pages
ISBN: 9781935182641



Real-World Functional Programming

IN PRINT

Tomas Petricek with Jon Skeet
December 2009 | 560 pages
ISBN: 9781933988924



Scala in Action

EARLY ACCESS EDITION

Nilanjan Raychaudhuri
MEAP Began: March 2010
Softbound print: Spring 2011 | 525 pages
ISBN: 9781935182757