

SAMPLE CHAPTER

RabbitMQ

IN ACTION

Distributed messaging for everyone

Alvaro Videla
Jason J.W. Williams

FOREWORD BY
ALEXIS RICHARDSON



MANNING



RabbitMQ in Action

Alvaro Videla

Jason J.W. Williams

Chapter 1

Copyright 2012 Manning Publications

brief contents

- 1 ■ Pulling RabbitMQ out of the hat 1
- 2 ■ Understanding messaging 12
- 3 ■ Running and administering Rabbit 37
- 4 ■ Solving problems with Rabbit: coding and patterns 60
- 5 ■ Clustering and dealing with failure 87
- 6 ■ Writing code that survives failure 107
- 7 ■ Warrens and Shovels: failover and replication 120
- 8 ■ Administering RabbitMQ from the Web 137
- 9 ■ Controlling Rabbit with the REST API 154
- 10 ■ Monitoring: Houston, we have a problem 167
- 11 ■ Supercharging and securing your Rabbit 195
- 12 ■ Smart Rabbits: extending RabbitMQ 216

Pulling RabbitMQ out of the hat

This chapter covers

- The need for an open protocol—AMQP
- Brief history of RabbitMQ
- Installing RabbitMQ
- First program—Hello World

We live in a world where real-time information is constantly available, and the applications we write need easy ways to be routed to multiple receivers reliably and quickly. More important, we need ways to change who gets the information our apps create without constantly rewriting them. Too often, our application’s information becomes siloed, inaccessible by new programs that need it without rewriting (and probably breaking) the original producers. You might be saying to yourself, “Sure, but how can message queuing or RabbitMQ help me fix that?” Let’s start by asking whether the following scenario sounds familiar.

You’ve just finished implementing a great authentication module for your company’s killer web app. It’s beautiful. On every page hit, your code efficiently coordinates with the authentication server to make sure your users can only access what

they should. You're feeling smug, because every page hit on your company's world-class avocado distribution website activates your code. That's about when your boss walks in and tells you the company needs a way to log every successful and failed permission attempt so that it can be data mined. After you lightly protest that that's the job of the authentication server, your boss not so gently informs you that there's no way to access that data. The authentication server logs it in a proprietary format; hence this is now your problem. Mulling over the situation causes a four-aspirin headache, as you realize you're going to have to modify your authentication module and probably break every page in the process. After all, that wonderful code of yours touches *every* access to the site. Let's stop for a moment though. Let's punch the Easy button and time warp back to the beginning of the development of that great auth module. Let's assume you leveraged message queuing heavily in its design from day one.

With RabbitMQ in place, you brilliantly leveraged message queuing to decouple your module from the authentication server. With every page request, your authentication module is designed to place an authorization request message into RabbitMQ. The authentication server then listens on a RabbitMQ queue that receives that request message. Once the request is approved, the auth server puts a reply message back into RabbitMQ where it's routed to the queue that your module is listening on. In this world, your boss's request doesn't faze you. You realize you don't need to touch your module or even write a new one. All you need to do is write a small app that connects to RabbitMQ and subscribes to the authorization requests your auth module is already publishing. No code changes. Nothing you already wrote knows anything has changed. It's so simple a smile almost breaks out on your face. That's the power of messaging to make your day job easier.

Message queuing is simply connecting your applications together with messages that are routed between them by a message broker like RabbitMQ. It's like putting in a post office just for your applications. The reality is that this approach isn't just a solution to the real-time problems of the financial industry; it's a solution to the problems we all face as developers every day. We, the authors, don't come from a financial services background. We had no idea what "enterprise messaging" was when we needed to scale. We were simply devs like you with an itch that needed scratching: an itch to deal with real-time volumes of information and route it to multiple consumers quickly. We needed to do it all without blocking the producers of that information ... and without them needing to know who the final consumers might be. RabbitMQ helped us to solve those common problems easily, and in a standards-based way that ensured any app of ours could talk to any other app, be it Python, PHP, or even Scala.

Over the next few chapters, we'll take you on a ride. It starts by explaining how message queuing works, its history, and how RabbitMQ fits in. Then we'll take you all the way through to real-world examples you can apply to your own scalability and interoperability challenges ... ending with how to make Rabbit purr like a well-oiled machine in a "downtime is not acceptable!" environment.

This is the book we wished was on the shelves when we entered the messaging wilderness. We hope it will help you benefit from our experience and battle scars and free you to make amazing applications with less pain. Before we're done in this chapter, you'll have a short history of messaging under your belt, and RabbitMQ up and running. Without further ado, let's take a look at where all this messaging fun started.

1.1 Living in other people's dungeons

The world of message queuing didn't start out the dank and cramped one it is today, with most folks subservient to lock-in overlords. It started with a ray of light in an otherwise byzantine software landscape. It was 1983 when a 26-year-old engineer from Mumbai had a radical question: why wasn't there a common software "bus"—a communication system that would do the heavy lifting of communicating information from one interested application to another? Coming from an education in hardware design at MIT, Vivek Ranadivé envisioned a common bus like the one on a motherboard, only this would be a software bus that applications could plug into. (See <http://hbswk.hbs.edu/archive/1884.html>.) Thus, in 1983 Teknekron was born. A freshly minted Harvard MBA in his hand and this powerful idea in his head, Vivek started plowing a path that would help developers everywhere.

Having the idea was one thing, but finding a killer application for it was something completely different. It was at Goldman Sachs in 1985 that Ranadivé found his first customer and the problem his software bus was born to solve: financial trading. A trader's stall at that time was packed to the brim with different terminals for each type of information the trader needed to do his job. Teknekron saw an opportunity to replace all those terminals and their siloed applications. In their place would be Ranadivé's software bus. What would remain would be a single workstation whose display programs could now plug into the Teknekron software bus as consumers and allow the trader to "subscribe" to the information the trader wanted to see. Publish-subscribe (PubSub) was born, as was the world's first modern message queuing software: Teknekron's *The Information Bus (TIB)*.

It didn't take long for this model of data transfer to find many more killer uses. After all, an application publishing data and an application consuming it no longer had to directly connect to each other. Heck, they didn't even have to know each other existed. What Teknekron's TIB allowed application developers to do was establish a set of rules for describing message content. As long as the messages were published according to those rules, any consuming application could subscribe to a copy of the messages tagged with topics it was interested in. Producers and consumers of information could now be completely decoupled and flexibly mixed on-the-fly. Either side of the PubSub model (producer/consumer) was completely interchangeable without breaking the opposite side. The only thing that needed to remain stable was the TIB software and the rules for tagging and routing the information. Since the financial trading industry is full of information with a constantly changing set of interested folks, TIB spread like wildfire in that sector. It was also noticed by telecommunications

and especially news organizations, who also had information that needed timely delivery to a dynamically changing set of interested consumers. That's why mega news outfit Reuters purchased Teknekron in 1994.

Meanwhile, this burgeoning new segment of enterprise software didn't go unnoticed by Big Blue. After all, many of IBM's biggest customers were in the financial services industry. Also, Teknekron's TIB software was frequently run on IBM hardware and operating systems ... all without the boys in White Plains getting a cut. Thus, in the late '80s IBM began research into developing their own message-queuing software, leveraging their extensive experience in information delivery from developing DB2 (see <http://www-01.ibm.com/software/integration/wmq/MQ15Anniversary.html>). Development began in 1990 at IBM's Hursely Park Laboratories near Winchester, United Kingdom. What emerged three years later was the IBM MQSeries family of message-queuing server software. In the 17 years since, MQSeries has evolved into WebSphere MQ and is today the dominant commercial message-queuing platform. During that time, Ranadivé's TIB hardly disappeared into the bowels of Reuters. Instead it has remained the other major player in enterprise messaging, thriving through a renaming to *Rendezvous* and Teknekron's re-emergence as an independent company in the form of TIBCO in 1997. The same year, Microsoft's first crack at the messaging market emerged: Microsoft Message Queue (MSMQ).

Through all of this evolution, *message queuing (MQ)* software primarily remained the domain of large-budgeted organizations with a need for reliable, decoupled, real-time message delivery. Why didn't MQ find a larger audience? How did it survive the information boom that was the late '90s internet bubble without experiencing explosive adoption? After all, everyone today from Twitter to Salesforce.com is scrambling to create internal solutions to the PubSub problems that The Information Bus solved 25 years ago. Two words: vendor lock-in. The commercial MQ vendors wanted to help applications interoperate, not create standard interfaces that would allow different MQ products to interoperate or, Heaven forbid, allow applications to change MQ platforms. Vendor lock-in has kept prices and margins high, and commercial MQ software out of reach of the startups and Web 2.0 companies that are abounding today.

As it turned out, smaller tech companies weren't the only ones unhappy about the high-priced walled gardens of MQ vendors. The financial services companies that formed the bread and butter of the MQ industry weren't thrilled either. Inevitably, the size of financial companies meant that MQ products were in place from multiple vendors servicing different internal applications. If an application subscribing to information on a TIBCO MQ suddenly needed to consume messages from an IBM MQ, it couldn't easily be done. They used different APIs, different wire protocols, and definitely couldn't be federated together into a single bus. From this problem was born the *Java Message Service (JMS)* in 2001 (see http://en.wikipedia.org/wiki/Java_Message_Service). JMS attempted to solve the lock-in and interoperability problem by providing a common Java API that hides the actual interface to the individual vendor MQ products. Technically, a Java application only needs to be written to the JMS API, with the appropriate MQ drivers selected. JMS takes care of the rest ... supposedly. The

problem is you're trying to glue a single standard interface over multiple diverse interfaces. It's like gluing together different types of cloth: eventually the seams come apart and the reality breaks through. Applications could become more brittle with JMS, not less. A new standards-based approach to messaging was needed.

1.2 AMQP to the rescue

In 2004, JPMorgan Chase required a better solution to the problem and started development of the *Advanced Message Queuing Protocol (AMQP)* with iMatix Corporation (see http://en.wikipedia.org/wiki/Advanced_Message_Queueing_Protocol#Development). AMQP from the get-go was designed to be an open standard that would solve the vast majority of message queuing needs and topologies. By virtue of being an open standard, anyone can implement it, and anyone who codes to the standard can interoperate with MQ servers from any AMQP vendor.

In many ways, AMQP promises to liberate us from the dungeons of vendors and fulfill Ranadivé's original vision: dynamically connecting information in real time from any publisher to any interested consumer over a software bus.

1.3 A brief history of RabbitMQ

In the early 2000s, a young entrepreneur out of the London financial sector co-founded a company for caching Java objects: Metalogic. For Alexis Richardson, the theory was simple enough: use Java objects for distributed computing and cache them in transit for performance. The reality was far different. Varying versions of the Java Virtual Machine, as well as differing libraries on the client and server, could make the

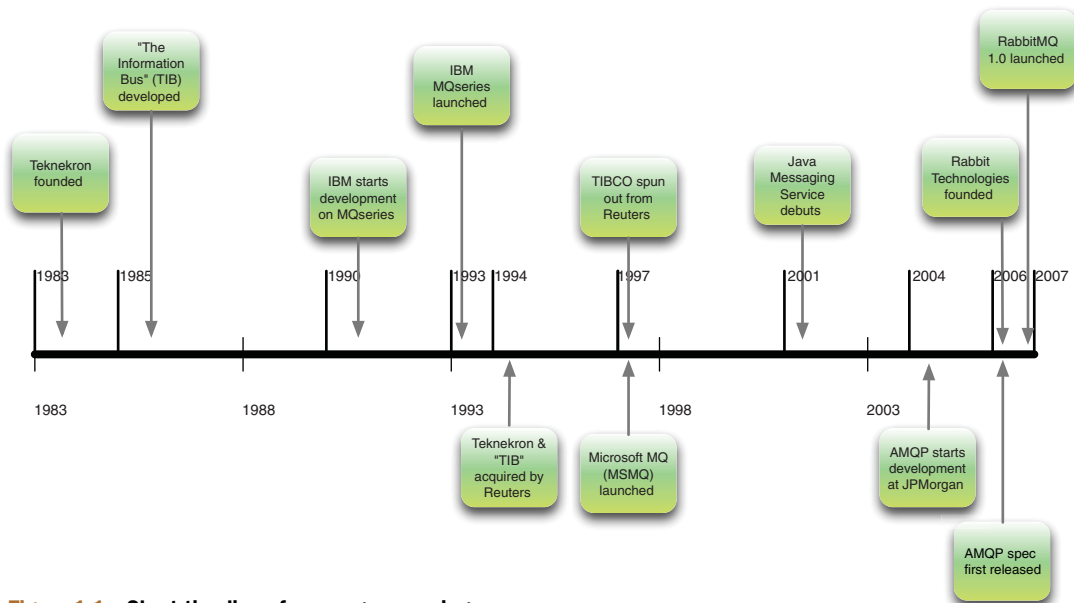


Figure 1.1 Short timeline of message queuing

objects unusable when they arrived. There were too many environment variables in the real world for Metalogic's approach to be widely successful. What did come out of Metalogic was Alexis meeting Matthias Radestock.

Matthias was working for LShift, where Alexis was subleasing office space while at Metalogic. LShift at the time was heavily involved in language modeling and distributed computing contracts for a major software vendor. The background in these areas triggered Matthias's interest in Erlang, the programming language that Ericsson had originally developed for their telephone switching gear. What grabbed Matthias's attention was that Erlang excelled at distributed programming and robust failure recovery, but unfortunately at the time it wasn't open source. In the meantime, Metalogic had closed operations and LShift was in the process of winding down their primary distributed computing contract. But Alexis had learned two valuable lessons from his experience at Metalogic: what works in a distributed computing environment, and what companies want for those environments.

Alexis knew he wanted to start a new company to solve the problems of communicating in a distributed environment. He also knew the next company he started would be open source and build on the model just proved successful by JBoss and MySQL. Looking back at where the Metalogic solution had run into problems, Alexis started to see that messaging was the right answer to distributed computing. More important, in the tech world circa 2004 a huge gap existed for open source messaging. No one was providing a messaging solution except for the big commercial vendors, and while "enterprise" open source was flourishing with databases (MySQL) and application servers (JBoss), no one was touching the missing component: messaging. Interestingly, it was in 2004 that AMQP was just starting to be developed at JPMorgan Chase. Through his background in the financial industry, Alexis had been introduced to the principal driver of AMQP at JPMorgan, John O'Hara (future founder of the AMQP Working Group). Through O'Hara, Alexis became acquainted with AMQP, and started lining up the building blocks for what would become RabbitMQ.

Around 2005, Alexis cofounded CohesiveFT. He and his cofounders in the U.S. started the company to provide an application stack and tools for what has today become cloud computing. That a key part of that stack would be distributed messaging seemed obvious to Alexis, who (still in the same office as LShift) started talking to Matthias about AMQP. What was clear to Matthias was that he'd just found the application he'd been looking for to write in Erlang. But before any of this could get started, Alexis and Matthias focused on three questions that they knew would be critical to an open source version of AMQP being successful if it was written in Erlang:

- 1 Would large financial institutions care whether their messaging broker was written in Erlang?
- 2 Was Erlang really a good choice for writing an AMQP server?
- 3 If it was written in Erlang, would that slow down adoption in the open source community?

The first issue was quickly dispatched by a financial company who confirmed they didn't care what it was written in if it helped reduce their integration costs. The second question was answered by Francesco Cesarini at Erlang Solutions: from his analysis of AMQP, the specification implied an architecture present in every telephone switch. In other words, you couldn't pick a better implementation language than Erlang for building an AMQP broker. The final question was put to rest by an entirely different messaging server: ejabberd. By 2005, *Extensible Messaging and Presence Protocol (XMPP)* had become a respected standard for open instant messaging, and one of the foremost implementations was the Erlang-based ejabberd server package by Alexey Shchepin. ejabberd was widely in use by many different organizations, and its implementation in Erlang didn't seem to be slowing anyone down.

With the three major questions answered, Alexis and Matthias convinced CohesiveFT and LShift to jointly back the project. The first thing they did was contract Matthew Sackman (now a core Rabbit developer) to write a prototype in Erlang to test latency. They quickly discovered that using the distributed computing libraries built into Erlang produced incredible latency that was comparable to using raw sockets. There was also the question of what to call this thing: everyone agreed on Rabbit. After all, rabbits are fast and they multiply like crazy, making it a great name for distributed software. Not the least of the reasons for this choice is that Rabbit is easy to remember. Thus, in 2006 Rabbit Technologies was born: a joint venture between CohesiveFT and LShift that would hold the intellectual property for what we know today as RabbitMQ.

The timing couldn't have been more perfect because, around the same time, the first public draft of the AMQP specification had become available. As a new specification, AMQP was rapidly changing. This was an area where Erlang proved critical. By using Erlang, RabbitMQ could be developed quickly and keep pace with a moving target: the AMQP standard. Amazingly, version 1.0 of RabbitMQ was written in only two and a half months by core developer Tony Garnock-Jones. From the beginning, RabbitMQ has implemented a key feature of AMQP that differentiates it from TIBCO and IBM: provisioning resources like queues and exchanges can be done from within the protocol itself. With the commercial vendors, provisioning is done by specialized staff at specialized administrative consoles. RabbitMQ's provisioning capabilities make it the perfect communication bus for anyone building a distributed application, particularly one that leverages cloud-based resources and rapid deployment.

That brings us to today, where RabbitMQ is used by everyone from small Silicon Valley startups to some of the largest names on the internet. That's perhaps the best thing about RabbitMQ, and the thing that surprised its founders: its largest block of users are tech firms, not financial companies. RabbitMQ fulfills Ranadivé's vision for the rest of us with smaller budgets and the same real problems. That's what drew us to RabbitMQ. We didn't know that we were looking for message-queueing software. All we knew was that we had real problems to solve integrating applications and serving high transaction loads. RabbitMQ provides a powerful toolkit for solving those problems, and brings to the masses the rich history of messaging ... and finally a pluggable information bus for everyone that needs one.

1.4 **Picking RabbitMQ out of the hat (and other open options)**

Today, RabbitMQ isn't the only game in town for open messaging. Options like ActiveMQ, ZeroMQ, and Apache Qpid all providing different open source approaches to message queuing. The question is, why do we think you should pick RabbitMQ?

- Except for Qpid, RabbitMQ is the only broker implementing the AMQP open standard.
- Clustering is ridiculously simple on RabbitMQ because of Erlang.
- Your mileage may vary, but we've found RabbitMQ to be far more reliable and crash resistant than its competitors.

Perhaps the most important reason is that RabbitMQ is incredibly easy to install and use. Whether you need a simple one-node setup for your workstation, or a seven-server cluster to power your web infrastructure, RabbitMQ can be up and running in about 30 minutes. With that in mind, it's about time we fired up the little critter.

1.5 **Installing RabbitMQ on Unix systems**

So far we've discussed the motivation behind the AMQP protocol and the history of the RabbitMQ server. Now it's time to get the broker up and running and start doing cool stuff with it. The operating system requirements for running RabbitMQ are flexible because we can run it on several platforms including Linux, Windows, Mac OS X, and other Unix-like systems. In this chapter we'll go through the process of setting up the server for a generic Unix system (all examples and instructions in the book assume a UNIX environment unless otherwise noted). Since RabbitMQ is written in Erlang, we need to have installed the language libraries to run the broker.

1.5.1 **Why environment matters—living la vida Erlang**

We recommend that you use the latest version of Erlang, which at the time of this writing is R14A. You can obtain a copy of Erlang from its website (<http://www.erlang.org/>). Please follow the installation instructions provided there. By running the latest version of Erlang on your system, you'll be sure to have all the updates and improvements for the foundations RabbitMQ will run on. Every new release of Erlang includes performance improvements that are worth having.

Once you have RabbitMQ dependencies solved, create a folder where you can perform our tests. Assuming that you're running a Unix-flavored system, fire up a terminal to start typing commands:

```
$ mkdir rabbitmqinaction
$ cd rabbitmqinaction
```

1.5.2 Getting the package

Then download the RabbitMQ Server from the server download page: <http://www.rabbitmq.com/server.html>. Select the package for a generic Unix system and download it:¹

```
$ wget http://www.rabbitmq.com/releases/rabbitmq-server/v2.7.0/\
rabbitmq-server-generic-unix-2.7.0.tar.gz
```

Your next step is to unpack the tarball and change to the `rabbitmq_server-2.7.0` directory inside the package:

```
$ tar -xzf rabbitmq-server-generic-unix-2.7.0.tar.gz
$ cd rabbitmq_server-2.7.0/
```

1.5.3 Setting up the folder structure

You're nearly ready to start the broker, but there are a couple of folders to create before you do that. The first one is where RabbitMQ will write the logs. You can look into this folder in case you need to troubleshoot your installation. The second folder is for the Mnesia database that RabbitMQ uses to store information about the broker, like queue metadata, virtual hosts, and so on. Type the following commands at the terminal:

```
$ mkdir -p /var/log/rabbitmq
$ mkdir -p /var/lib/rabbitmq/mnesia/rabbit
```

You may need to run those commands as a super user. If you have to do so, then don't forget to *chown* the folders to your system user.

TIP When we run RabbitMQ in production, we usually create a `rabbitmq` user and then we grant the folder privileges to that user instead of running all the commands with a normal user account.

1.5.4 Firing Rabbit up for the first time

Now you're all set to fire up the server. Type the final command to do so:

```
$ sbin/rabbitmq-server
```

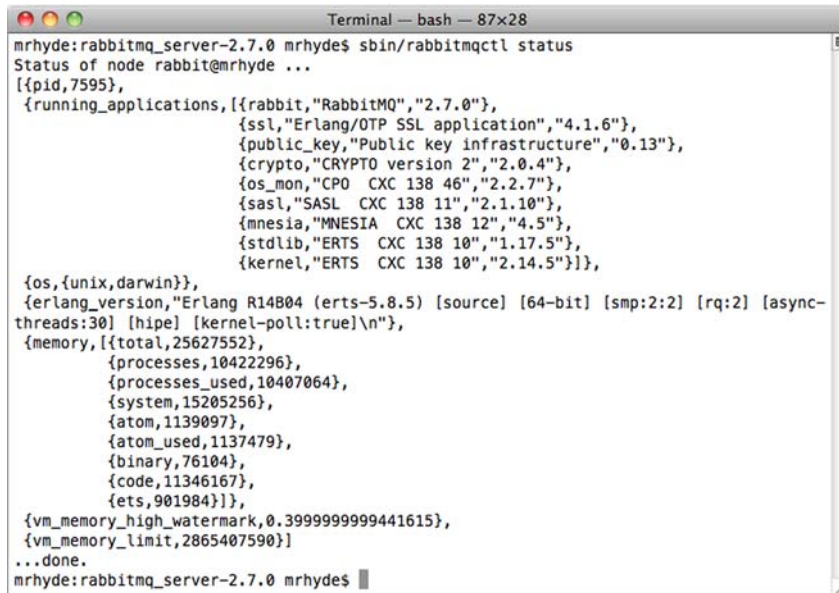
RabbitMQ will output some information about the startup progress. If all went as expected, you'll see ASCII art of the RabbitMQ logo and the message broker running, as seen in figure 1.2.

Now open a new terminal window and check the status of the server. Type the following:²

```
$ cd path/to/rabbitmqinaction/rabbitmq_server-2.7.0/
$ sbin/rabbitmqctl status
```

¹ Pre-build installation packages for RabbitMQ are available for Windows, Debian/Ubuntu and RedHat (RPM) from <http://www.rabbitmq.com/download.html>.

² If you installed from an RPM or Ubuntu/Debian package, you may need to run `rabbitmqctl` as root.



```

Terminal — bash — 87x28
mrhyde:rabbitmq_server-2.7.0 mrhyde$ sbin/rabbitmqctl status
Status of node rabbit@mrhyde ...
[{pid,7595},
 {running_applications,[{rabbit,"RabbitMQ","2.7.0"},
                        {ssl,"Erlang/OTP SSL application","4.1.6"},
                        {public_key,"Public key infrastructure","0.13"},
                        {crypto,"CRYPTO version 2","2.0.4"},
                        {os_mon,"CPO CXC 138 46","2.2.7"},
                        {sasl,"SASL CXC 138 11","2.1.10"},
                        {mnesia,"MNESIA CXC 138 12","4.5"},
                        {stdlib,"ERTS CXC 138 10","1.17.5"},
                        {kernel,"ERTS CXC 138 10","2.14.5"}]}],
 {os,{unix,darwin}},
 {erlang_version,"Erlang R14B04 (erts-5.8.5) [source] [64-bit] [smp:2:2] [rq:2] [async-threads:30] [hipe] [kernel-poll:true]\n"},
 {memory,[{total,25627552},
          {processes,10422296},
          {processes_used,10407064},
          {system,15205256},
          {atom,1139097},
          {atom_used,1137479},
          {binary,76104},
          {code,11346167},
          {ets,901984}}],
 {vm_memory_high_watermark,0.39999999999441615},
 {vm_memory_limit,2865407590}]
...done.
mrhyde:rabbitmq_server-2.7.0 mrhyde$

```

Figure 1.3 Checking RabbitMQ status

financial traders, to message routing monsters that are the beating heart of everything related to financial exchanges, to the manufacturing lines at semiconductor fabs. Now you have that kind of power running on your dev laptop, and we've only finished chapter 1! With RabbitMQ running, it's time to dive into the building blocks of messaging: queues, bindings, exchanges, and virtual hosts. Let's see how they all fit together and get Rabbit saying "Hello World"!

RabbitMQ IN ACTION

Videla • Williams



There's a virtual switchboard at the core of most large applications where messages race between servers, programs, and services. RabbitMQ is an efficient and easy-to-deploy queue that handles this message traffic effortlessly in all situations, from web startups to massive enterprise systems.

RabbitMQ in Action teaches you to build and manage scalable applications in multiple languages using the RabbitMQ messaging server. It's a snap to get started. You'll learn how message queuing works and how RabbitMQ fits in. Then, you'll explore practical scalability and interoperability issues through many examples. By the end, you'll know how to make Rabbit run like a well-oiled machine in a 24 x 7 x 365 environment.

What's Inside

- Learn fundamental messaging design patterns
- Use patterns for on-demand scalability
- Glue a PHP frontend to a backend written in anything
- Implement a PubSub-alerting service in 30 minutes flat
- Configure RabbitMQ's built-in clustering
- Monitor, manage, extend, and tune RabbitMQ

Written for developers familiar with Python, PHP, Java, .NET, or any other modern programming language. No RabbitMQ experience required.

Alvaro Videla is a developer and architect specializing in MQ-based applications.

Jason J. W. Williams is CTO of DigiTar, a messaging service provider, where he directs design and development.

For access to the book's forum and a free eBook in all formats, owners of this book should visit manning.com/RabbitMQinAction

“In this outstanding work, two experts share their years of experience running large-scale RabbitMQ systems.”

— Alexis Richardson, VMware
Author of the Foreword

“Well-written, thoughtful, and easy to follow.”

—Karsten Strøbæk, Microsoft

“Soup to nuts on RabbitMQ; a wide variety of in-depth examples.”

—Patrick Lemiux, Voxel Internap

“This book will take you to a messaging wonderland.”

—David Dossot
coauthor of *Mule in Action*



\$44.99 / Can \$47.99 [INCLUDING eBook]

ISBN 13: 978-1-935182-97-9
ISBN 10: 1-935182-97-8



9 781935 182979