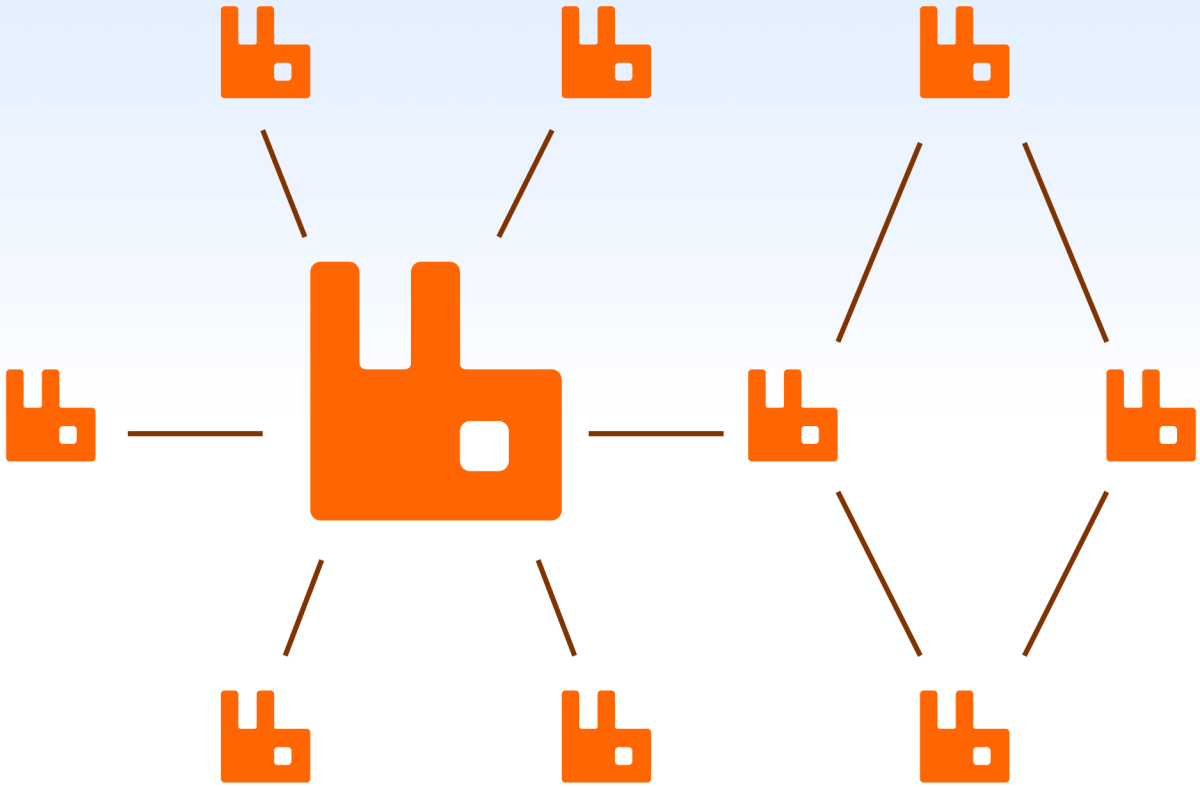
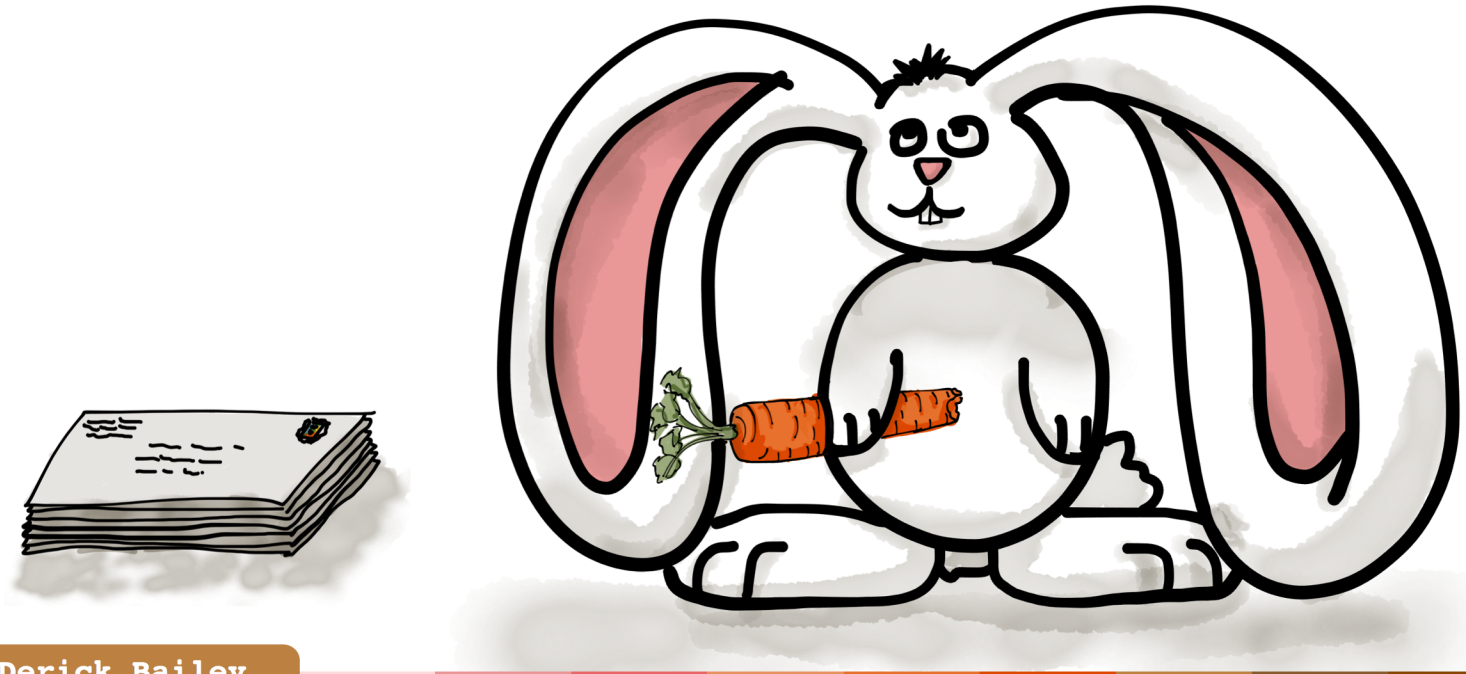
RabbitMQ Layout

Basic Structure and Topology for Applications





By Derick Bailey

RabbitMQ Layout

Basic Structure and Topology for Applications

Derick Bailey

Tweet This Book!

Please help Derick Bailey by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm learning how to organize my messaging infrastructure thanks to @derickbailey.

The suggested hashtag for this book is #RabbitMQTopology.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#RabbitMQTopology

Also By Derick Bailey

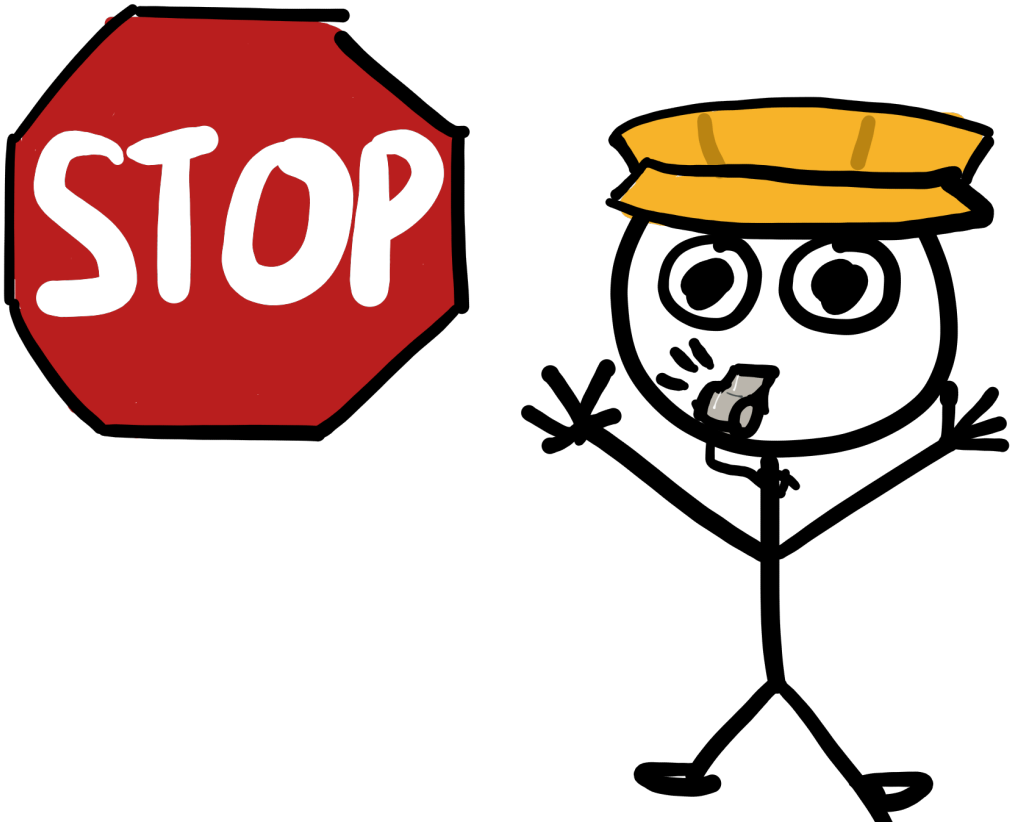
Building Backbone Plugins

5 Rules For Mastering JavaScript's "this"

RabbitMQ: Patterns for Applications

Contents

Before You Start Reading This Book ...



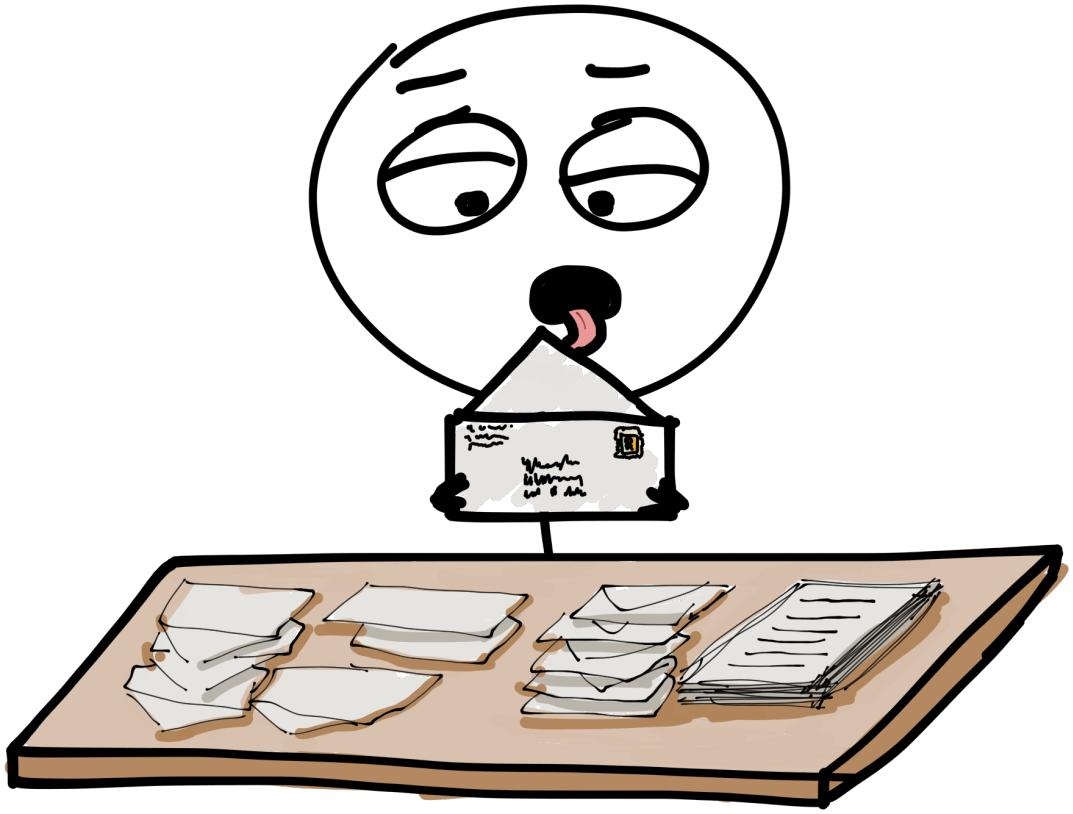
Did you know that there's a companion email course to go along with this book? It's a free guided tour set up to help you get the gist of each chapter! You'll get 1 email every 3 days to help you understand the background and the main lessons in each chapter. It's a great companion to the book, and it's free!

Sign up for the FREE guided tour[1] and get all the inside info!

[1] https://www.getdrip.com/forms/5862535/submissions/new

Preface

Have you ever stuffed envelopes full of the same things - cards, invitations, junk mail or otherwise - only to label each envelope with a different address?



If you've ever sent out Christmas cards, birthday invitations or thank-you cards, you probably understand the tedious nature of doing this.

1. Grab an envelope
2. Open the envelope
3. Grab an insert
4. Put the insert in
5. Close the envelope

6. Address the envelope
7. Put a stamp on it
8. Put it in a big pile of envelopes
9. Goto step 1

Tedious may be a generous word, here.

One Card, Many Deliveries

Imagine, for a moment, a situation where you can take one copy of that birthday invitation and have it sent to everyone on your list at the same time. Instead of having to stuff and label each individual envelope, you only need a list of addresses to go with one invitation. Everyone on the address list gets a copy when you send it, and you don't have to worry about creating multiple copies.

Sounds great, right?

It also sounds like email. You probably do this every day when you send email to multiple recipients. It would certainly be nice to have an email-like capability with physical mail, wouldn't it?

But what if you don't want to deal with individual addresses?

Perhaps you want to have a block party and invite the whole neighborhood. Wouldn't it be nice if you could send out one invitation to one "address" or "group", and have a copy of the invitation show up in the mailbox of everyone in the neighborhood? And maybe you want to skip that grumpy neighbor down the block - send it to everyone except for them. Ok, sure - include your friend from that other neighborhood, as well. Everyone here knows them, anyways. And then there's ...

It would be nice if you could figure out a way to have this one invitation correctly routed to all of the people that you want to invite, and not have to label so many envelopes, stuffed with copies of the invitation.

Enter RabbitMQ

In the world of physical mail, there is a struggle to address letters and envelopes individually. In the world of email, it gets easier because you can send one letter to multiple address. But what do you do in the software world, when you need one application to communicate with multiple external processes, systems and other applications?

The "easy" way is like stuffing envelopes with copies of the same message. Only in this case, you have to re-write the message with a slightly different format and sent it to different API endpoints.

With messaging systems like RabbitMQ, though, you can have all the benefits of one message for many recipients, and individually addressing envelopes.

You can make a single copy of a message and address to it a single destination. You can have one message addressed to many specific destinations. You can exclude or include destinations based on pattern matching. You can also send messages to anyone that happens to be listening at the moment, ignoring anyone that recently left the conversation or joined in later.

There is a lot of flexibility with RabbitMQ messaging and topologies. But, there isn't a lot of information about how to organize things; how to create an exchange, routing and queue layout that makes sense for your application.

That's where this book comes in.

About This Book

In this book, you'll learn about the basics of RabbitMQ's Exchanges, Queues and the bindings between them. You'll learn about the 3 primary types of exchanges, and one exchange type that we won't bother with. These include:

- Direct Exchanges
- Fanout Exchanges
- Topic Exchanges

and worth mentioning, but not something that will be covered: Header Exchanges.

Part 1: RabbitMQ Exchanges

Part 1 of the book will introduce the three main exchange types that you will be using with RabbitMQ.

Chapter 1 will introduce the core concepts of RabbitMQ topologies. Chapters 2 through 4 offer a brief discussion of the features and behavior of each Exchange type, and binding them to destination queues.

Part 2: Understanding The Design And Decisions

Part 2 moves in to the application of RabbitMQ's topology options, providing a unique look at making decisions. With a myriad of options and features available, it is not always clear when and where you should use what.

Rather than providing a strictly technical approach to show you how to use RabbitMQ, part 2 takes a story-telling approach to guide you through the decision making paths of real world projects and systems. Through a narrative style that puts you in the mind of another developer, you will see the struggles, the success and some of the failures in designing a message based system with RabbitMQ.

Chapter 5 will show a sample job scheduling application, following a developer through a decision to implement RabbitMQ and organize it appropriately. The running jobs in the schedule can be put in to various states through an administrative web application, but how should the queues and exchanges be organized to facilitate this? The inspiration for the solution seems to come out of nowhere.

Chapter 6 follows a developer through the often painful process of discovering and documenting performance problems, and finding solutions to the issue. RabbitMQ is already in place, in this case, but was it the right thing to do? Should RabbitMQ be taken out of the equation in order to reduce the latency between requesting a download and actually sending the file back to the user? Find out whether or not RabbitMQ makes the cut in this story of struggling to improve the performance of a file sharing service.

Chapter 7 follows a developer with a side project that is quickly growing in popularity, and slowing in speed. The current setup is working fine for the end-user, but the background processing is grinding to a halt and causing massive delays between a database call and integration with a 3rd party service that seems to fail more than it succeeds. Follow along with a developer that is bringing on a second team member, while trying to understand how to keep a queue from becoming a backlog of thousands of messages.

Chapter 8 is a final look at RabbitMQ's topology options, building an application that takes the "Internet of Things" (IoT) to a new level. Step inside the mind of a developer who winds up working for a decidedly low-tech company, using cutting edge technologies to facilitate communication between work out in the field and maintenance bays back at the company headquarters.

Who Should Read This Book?

I wrote this book to speak to the problems that I found when learning RabbitMQ on my own. There were plenty of books, blogs and other resources available for the "how-to" of RabbitMQ, but I found the lack of "why-to" resources to be particularly frustrating.

If you're looking at RabbitMQ as a solution to your problems, and need to understand some of the decisions that go into building a RabbitMQ based system, then this book is for you. If you're in the middle of your first or second RabbitMQ implementation and are struggling to understand when you would use a Direct exchange vs a Fanout exchange, then look no further. If you're a technical leader or manager in a team of technical gurus, and need a high level overview of the decisions and criteria that your team will face, this book has you covered.

This book is for you - the application and system architect, the team lead and software developer. You will learn how to keep your RabbitMQ topology sane for your application.

You will learn not only the different types of exchanges in RabbitMQ and how they work, but when to use them and how to organize your RabbitMQ topology around them - all from an application perspective.

Who Should NOT Read This Book?

If you're looking for the authoritative "how-to" with RabbitMQ, you're looking at the wrong book. While I do provide a list of my favorite resources to cover this aspect of RabbitMQ, this book does not directly cover it.

If you're looking for a definitive, rules based guide to picking your exchange type, you will likely be disappointed. The truth is, most of the decisions made around messaging topology is fuzzy logic at best. This book will offer some real-world examples to illustrate areas where each exchange type works well, but it will not give you absolute constraints under which you should choose a specific option.

Additionally, this book is not going to teach you what you need to know about integrating disparate systems. It does not cover enterprise level integration needs, even if it does provide some discussion on the basic patterns and implementation details within RabbitMQ.

If you're looking for a guide to integration multiple large-scale or enterprise systems, this is *not* a book that you should be reading. To cover that material, I highly recommend the Enterprise Integration Patterns[2] book instead.

With that said, I hope you are set for an intriguing look inside the minds of other developers as they work their way through the struggles and the success of working with RabbitMQ.

[2] http://bit.ly/db-rmq-eip

Part 1: Exchanges, Queues and Bindings

From an over-simplified perspective, RabbitMQ allows you to send and receive messages much like a mail service does. In this case, though, the messages are not sent through a post office. Instead, they are sent through exchanges that are bound to queues in various ways.

The specific bindings from exchanges to queues, and how messages flow from the exchanges to the queues, is determined by a couple of things:

1. The type of exchange used to publish the message
2. The routing key used when publishing the message

You can think of the routing key as an address label for an envelope. When you send a message to an address, the exchange figures out how to actually deliver it to the right location. To figure out what the right location is - and if there are multiple locations which should receive the message - routing keys are matched to bindings in various ways.

The specifics of how routing keys are matched depends on the type of exchange being used.

There are three core exchange types that you will use with RabbitMQ:

- Direct Exchanges
- Fanout Exchanges
- Topic Exchanges

The next three chapters will discuss each of these exchange types in turn. You'll see how messages are routed with routing keys in each case, and how you can take advantage of the different exchange types when handling various scenarios.

Chapter 1: The Basics Of Sending Messages

If you've ever delivered an item through a postal service, there is a good chance you took an envelope, wrote an address on it, put a stamp on the envelope and delivered it to either a post drop box or to a post office. From there the post office became responsible for routing and delivery of the message to the intended recipient.

Messaging architectures in software work much the same way.

With RabbitMQ, you have an exchange - a place where you take your message and drop it off (publish it). The message may be labelled with a specific address (a routing key) if it needs one. The routing key is used to tell the postal delivery personnel where the message is to be delivered. The message is then delivered (routed) to the mailbox (queue) in question.

On the other end of the system, something may be there, waiting for messages to show up in the queue. When a message arrives in the queue and a message consumer is available, it picks up the message, reads the contents and takes appropriate action.

Sometimes the message requires a signature - acknowledgement that the message was delivered, picked up and handled. Other times, the message is shoved in to the queue and no one cares whether or not it was properly handled.

There is a lot of flexibility in how messages are routed, how they are stuffed in to the queue on the other end (assuming there is one), and how the message is handled (if at all). The number of options seem to create a nearly endless combination of behaviors and features, and it can be overwhelming at first. With a little bit of knowledge on how each of the different types of exchanges works, though, the flexibility and possibilities begin to make sense.

Routing Messages

At its core, RabbitMQ allows messages to be published through an exchange and routed to zero or more queues. To receive messages from an exchange, a queue is bound to that exchange using a routing key. Depending on the type of exchange and the routing key specified with a message, zero or more queues may receive the message.

Think of it this way:

If you put a fake address on an envelope, stamped it and sent it to the post office, what would happen? If you bothered to put a return address on the envelope, it would be returned to you with a note saying the address was not found (or something similar). If you didn't bother putting a return address on the envelope, though, the post office would throw away the letter. It wouldn't be delivered anywhere.

Messaging architectures tend to work the same way, and RabbitMQ is no exception.

You are free to add an invalid routing key to your messages whenever you want. When you publish that message, though, the exchange will simply dump it in to the void, never to be seen again.

Therefore, it is important that you have a well known set of routing keys that are appropriately bound in your exchanges and can be used with your messages.

Invalid Keys

Note: While there are ways around invalid routing keys and how to handle them, this subject is not covered in this book. For more information on that, see the RabbitMQ documentation on alternate exchanges[3].

[3] https://www.rabbitmq.com/ae.html

RabbitMQ Exchange Types

Knowing your routing key is valid will be sufficient for a message publisher - you only need to specify the exchange name and routing key to publish a message successfully. When it comes to routing the message to the correct queue(s), however, knowing what type of exchange is being used and how the routing keys are configured is important.

RabbitMQ provides 4 types of exchanges - only 3 of which you will generally care about and use.

1. Direct exchange
2. Fanout exchange
3. Topic exchange
4. Header exchange

Each of these exchange types each provides a unique set of routing behaviors that are configured in the exchange bindings.

Direct exchanges require an exact match between the routing key on an exchange's binding, and the routing key on a message. If the routing key does not match exactly, the message does not get delivered to that queue.

Fanout exchanges don't use routing keys. Instead, they route all message to every bound exchange, always.

Topic exchanges provide pattern matching for routing keys. Messages can be routed to specific queues with full routing keys, or wild-cards can be used to send message to more than one queue while limiting which queues receive the message.

Header exchanges are rarely used. They have a performance impact and are generally to be avoided because of this. The vast majority of use cases for header exchanges can be handled in the other exchange types by using a more appropriate routing configuration. Header exchanges will not be discussed further.

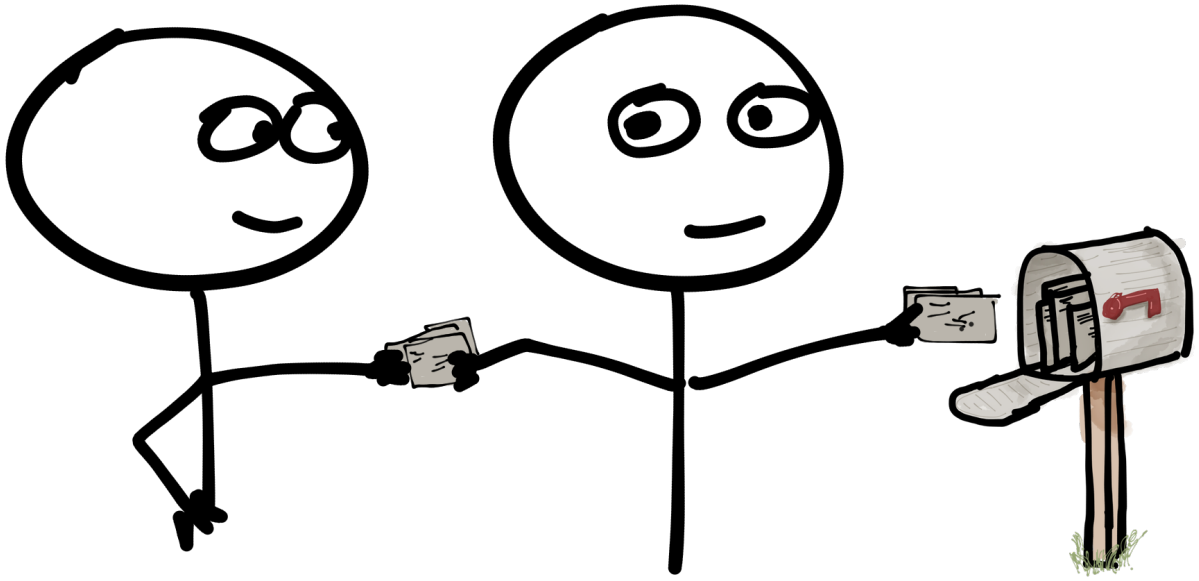
One Exchange to Rule Them All?

The truth about exchange types is that there is no "master" type - not one to be used as a default, or most of the time. Sure, a given application may have its needs served by a single exchange or exchange type, but this will not always be the case. Even with in a single system, there may be a need to route messages in different ways and have them end up in the same queue.

If you find yourself in a situation where choosing one of the above exchange types will preclude a needed set of routing behaviors for your messages, use more than one exchange. You can route from any number of exchanges to a single queue, or from a single exchange to any number of queues.

Don't limit your systems routing needs to a single exchange type for any given message or destination. Take advantage of each one, as needed.

Chapter 2: Direct Exchanges



A direct exchange allows you to bind a queue to an exchange with a routing key that is matched, case sensitively. This may be the most straight-forward exchange of them all, as there is no pattern matching or other behavior to track and consider. If a routing key from a message matches the routing key of a binding in the exchange, the message is routed.

An Example: Job Requests

Say you are building a system that can run an arbitrary number of jobs as background processes. These jobs may be part of a web application, but will take more time than is reasonable for a website request.

In this situation, you may have an exchange named “jobs”. This exchange could have a binding with a routing key of “job-request”. That routing key may route messages to a queue name “job-queue”.

An example configuration, using the wascally library for NodeJS, may look like this:

```
var wascally = require("wascally");
wascally.configure({
  connection: { /* ... */ },

  exchanges: [
    {name: "jobs", type: "direct"}
  ],

  queues: [
    {name: "job-queue"}
  ],

  bindings: [{
    exchange: "jobs",
    target: "job-queue",
    keys: ["job-request"]
  }]

});
```

With this configuration any message that is sent to the “jobs” exchange with a routing key of “job-request” will have the message routed to the “job-queue” queue.

Routing Keys

Routing keys for direct exchanges and their bindings must match exactly. Any deviation from the specified key will result in the message not being routed correctly (if it is routed at all).

Route Key Matching

Given the configuration shown above, a message that is published to the “jobs” queue with a routing key of “job-request” will be queued in the “job-queue” queue.

```
// publish a message to the "jobs" queue
// and route it with the "job-request" key

wascally.publish("jobs", {
  // ...

  key: "job-request",

  // ...
});
```

If you change the routing key to anything other than “job-request”, the message may not be routed correctly. Even if the only change is capitalization of a single letter, it is considered a different key.

```
// publish a message to the "jobs" queue
// and route it with an incorrect key

wascally.publish("jobs", {
  // ...

  key: "job-Request",

  // ...
});
```

In this example, the capitalized “R” in “Request” will make the message unroutable. Of course, you could add a route binding for this variation, but having to create common variations or misspellings will quickly pollute your exchange bindings and RabbitMQ configuration. It would be best to keep your bindings simple and ensure you are using the correct routing key.

Default Routing Keys

With direct exchanges you may publish a message with an empty routing key. This is sometimes called the “default” routing key, or just an empty routing key.

Whatever you call it, the routing key specified on a message must match the routing key in a binding. If a message is sent without a routing key but there are no bindings with an empty routing key specified, the message will not be routed.

Many Exchanges Bound To Many Queues

Direct exchanges allow for any number of bindings to use the same key. This allows you to have a single message route to many different queues, if needed. Having a message show up in many queues will allow many different processes to receive the message.

You may also use as many different routing keys to bind to one or more queues. Each routing key + destination queue is a new binding.

When To Use Direct Exchanges

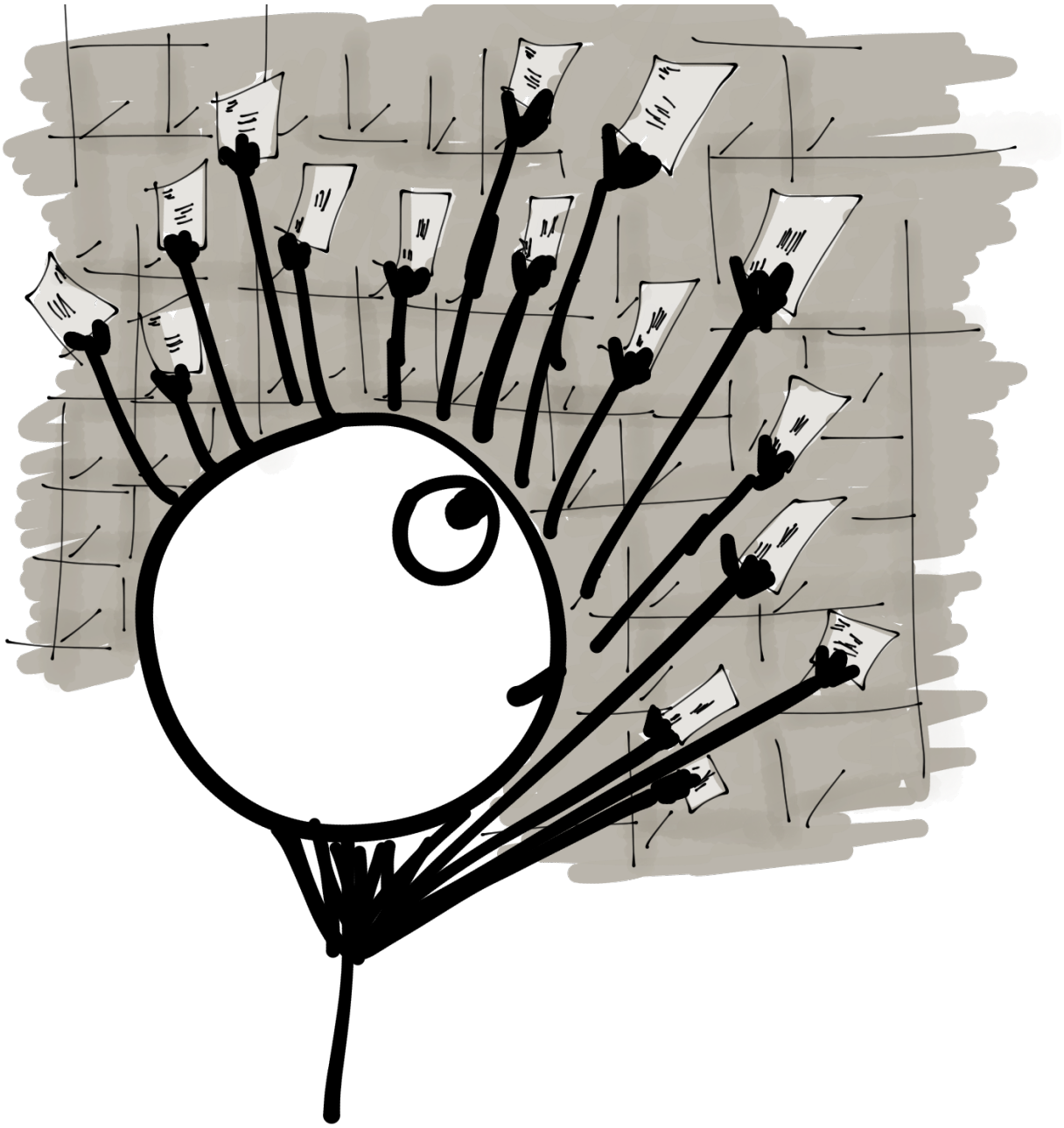
Direct exchanges are a good default to use for RabbitMQ. They are relatively simple to configure and easy to reason about.

> "This named exchange with this routing key goes to this queue."

If you need a message to go to a specific queue or to a specific message consumer, consider using a direct exchange.

You can use a direct exchange to route a single message to multiple queues, as well. To do that, you need multiple bindings - one for each queue that should receive the message. However, there may be better options for multiple queues receiving a message, in the fanout or topic exchange types.

Chapter 3: Fanout Exchanges



Fanout exchanges allow you to broadcast a message to every queue bound to an exchange, with no way to filter which queues receive the message. If a queue is bound to a fanout exchange, it will

receive any message published through that exchange.

Fanout exchanges work similarly to publish / subscribe systems like event emitters in JavaScript (Backbone.Events, jQuery Events, NodeJS' EventEmitter). Any queue that is bound to the exchange will receive any message sent through the exchange, at the time it is sent. If a queue is not currently bound to the exchange when the message is sent, it does not receive the message.

An Example: Notifications

Say you have a system that produces useful information that other systems need. You could have a specific set of queues in place for each of the systems that need notification of this information being available. In that situation, a fanout exchange can be used to broadcast the message to all available subscribers.

Where this stands out is not in the ability to send messages to multiple recipient queues. Any exchange type can do that. The difference is that you can have queues show up and disappear as needed, creating a binding to the queue, and not having to worry about routing keys.

In the notification example, there may be external systems that only need updates some of the time. Imagine, for example, a social network that uses message queues for each person that is logged in (please note: this is probably a bad example, but illustrates the point).

When a user logs in, the application can create a queue with a binding to the notifications exchange.

```
var userId = "{some user id}";

wascally.configure({

  queues: [
    { name: userId, exclusive: true, autoDelete: true }
  },

  exchanges: [
    { name: "notifications" }
  ],

  bindings: [
    { exchange: "notifications", target: userId }
  ]

});
```

When the user logs out, the application can destroy the queue (and thereby, the exchange binding). Or, better yet, the application can use a queue that is set up to auto-delete and marked as exclusive

as shown above. This ensures only the one application instance can use the queue (private) and that the queue will disappear when the client disconnects (autoDelete).

By using a fanout exchange, neither the message producer nor consumer has to worry about routing keys. Any application instance that is connected to a queue will receive a copy of all notifications. When the application disconnects, the queue goes away but the message producer doesn't care. It just keeps sending messages, whether or not any queues are there to listen.

Routing Keys

Fanout exchanges don't use routing keys in the bindings. You may be able to specify keys in the bindings or in the messages that are sent, but they will be ignored.

One Exchange, Many Queues

All messages sent to a fanout exchange will make their way to every queue that is bound to the exchange.

You could a single queue bound to a fanout exchange as well as other exchanges. This is likely not the best use case for fanout exchanges, though.

It would be a better idea to have one fanout exchange bound to multiple queues, and to have those queues dedicated to receiving messages from that one exchange.

When To Use Fanout Exchanges

When an application doesn't not need to know about messages that are sent when the application is not connected to the queue, this is a good scenario for a fanout exchange with private, auto-deleted queues.

Unlike direct exchanges that need to be configured with routing keys, fanout exchanges are rather specialized and don't need any routing keys. This makes them very well suited for scenarios where you need to allow applications to bind and unbind queues as needed.

The notifications example from above is an adequate example, but there are likely other use cases as well.

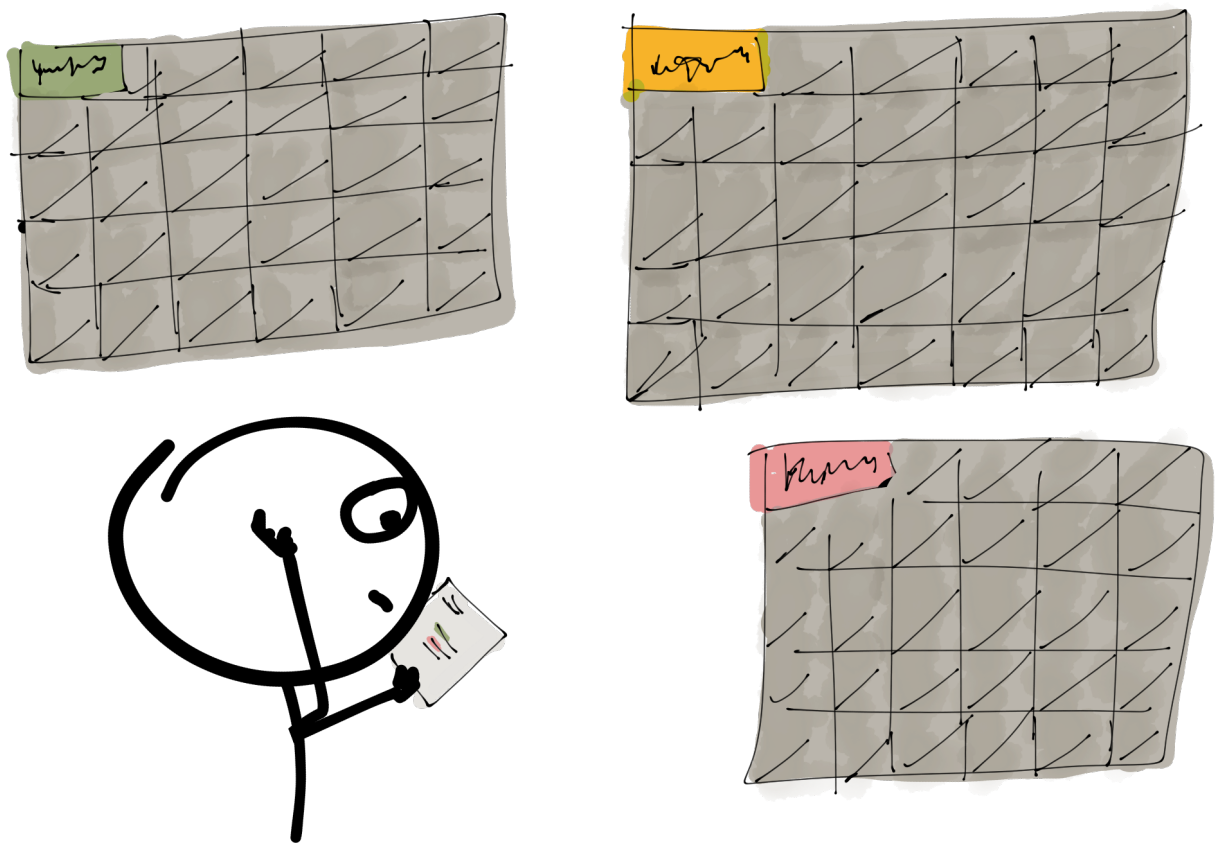
You may want to have log messages sent to many different logging systems, for example. But if one of those logging systems is offline, you may not want to have a large queue of messages waiting for it when it comes back online. Those log messages are temporal, and if the logging application is offline, it doesn't matter that it did not receive those messages.

In another scenario, you may have events that are triggered by one system and other systems need to take some appropriate action based on those events. Since events tend to be "fire-and-forget" modes

of communication - effectively tossing something over a wall and not caring if anything is there to catch it - a fanout exchange may be a good choice.

When events are triggered, any connected queue will receive the event. When the application that owns the queue is no longer running or has destroyed it's event queue, it will no longer receive events.

Chapter 4: Topic Exchanges



A topic exchange is similar to a direct exchange in that it uses routing keys. Unlike a direct exchange, though, the routing keys do not have to match exactly for a message to be routed. Topic exchanges allow you to specify wild-card matching of "topics" (routing keys) in your bindings. This lets you receive messages from more than one routing key and provides a level of flexibility not found in the other exchange types.

An Example: Logging

Say you need to log messages from various parts of your system. You may want to have different logging mechanisms available depending on the log severity. You may want to have a "master" log file that catches all log messages, no matter what the severity is. Additionally, there may be times when you need additional applications to handle messages of varying severity.

In a scenario like this, a topic exchange is going to serve you well. With the flexible nature of routing keys and pattern matching, you will be able to quickly and easily bind the right queues to the right routing keys.

Sending Log Messages With Routing Keys

In your application, you can have code send logging messages with the severity set in the routing key. Debug logging may have a key of "log.debug" while errors may have a key of "log.error" and so-on.

```
wascally.publish("log.ex", {
  // ...
  routingKey: "log.debug"
});

wascally.publish("log.ex", {
  // ...
  routingKey: "log.error"
});
```

With these routing keys in place, you will be able to assign the messages to queues using some simple pattern matching.

Routing All Messages

There are a few different ways to route all log messages in this scenario. The truly global "catch all" requires the use of a # (pound-sign) for the routing key.

```
exchanges: [
  { name: "log.ex", type: "topic" }
],

queues: [
  { name: "log.all-messages" }
],

bindings: [
  {
    exchange: "log.ex",
    target: "log.all",
    routingKeys: "#"
  }
]
```

Note the use of `#` in the routingKeys for the binding. This tells RabbitMQ to send all messages that are published through this exchange, no matter the routing key used, to this queue.

Routing All Log Messages

In addition to the `#` shown above, you can use simple pattern matching to route all log messages that start with "log".

RabbitMQ's pattern matching is similar to that of most file systems. You can can "log.*" to catch all messages that have a routing key starting with "log", separated in to parts with a "." (dot or period).

For example, a log message with a routing key of "log.debug" will be matched to a route binding of "log.*", as will "log.error", "log.info", or "log.any-other-words-you-want".

```
bindings: [
  {
    exchange: "log.ex",
    target: "log.debug.q",
    routingKeys: "log.debug"
  }, {
    exchange: "log.ex",
    target: "log.error",
    routingKeys: "log.error"
  }, {
    exchange: "log.ex",
    target: "log.all",
    routingKeys: "log.*"
  }
]
```

In this example, there are three separate logging queues. The first two use standard routing key matching with no wild cards. The last one sets up a binding to put all messages with a routing key of "log.*" in to the "log.all" queue. This binding will match all log messages that have a routing key starting with "log.".

Routing Keys

The examples above are over-simplifed to illustrate common use cases for pattern matching in routing keys. In truth, the `#` and `*` symbols both have very specific meaning and matching characteristics.

Word Separator .

When dealing with "words" in routing keys, you should know that each word is separated by a "." in the key. Therefore "log.error" is 2 (two) words, while "log-error" is considered a single word.

Matching Single Words With *

The * (asterisk) is used when you want to match a single word in a specified position.

A binding of just * will match a single word routing key - any word, but only a routing key with a single word as the key.

A binding of `log.*` will match routing keys of "log.error" and "log.debug", but will not match "log.error.app-1" or "log.error.app-2".

Matching Zero Or More Words With

The # (pound) is used when you want to match zero or more words in a specified position.

A binding of # will match zero or more words in a routing key. In other words, it will match all routing keys, including empty routing keys (zero words).

A binding of `log.#` will match all routing keys that start with "log." - no matter how many separators are used in the routing key.

Complex Pattern Matching

With the combined ability of * and # matchers, you can create some rather complex matching patterns.

A binding of `log.*.#` will match all routing keys that have at least two words, starting with "log" as the first word.

A binding of `#.error.*` will match any number of words at the beginning, with the word "error" just prior to the final word in the key.

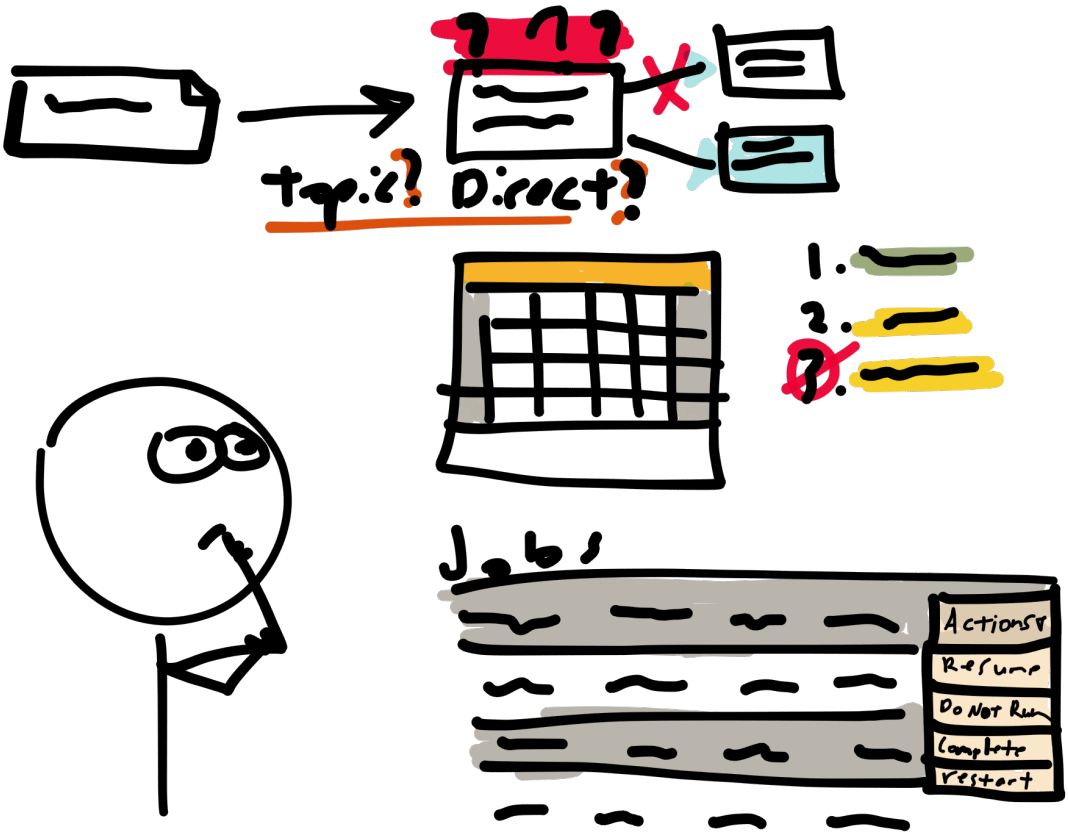
When To Use Topic Exchanges

Topic exchanges are useful when you have a highly structured set of routing keys that need complex rules for which queues will receive the messages. Although logging is the most typical example of complex routing rules, you will find other situations where mutliple queues should receive a message based on a word in a specific position of the routing key.

Part 2: Applications, Topologies and Decisions

Knowing the different types of exchanges and how they handle routing keys is only half the battle that you will have with RabbitMQ. In order to effectively work with RabbitMQ over time, you need to properly manage the topology - that is, the layout of the exchanges and queues, and the bindings that tie them together.

Chapter 5: Changing Job States In A Scheduling System



For the last year, your team has been working on a scheduling system for the company's batch processes. This system allows company IT professionals to create schedules that run nightly, weekly, monthly or however often they need to run. Each of the schedules is made up from one or more jobs that are executed by sending a message across RabbitMQ. An "agent" (job execution code) will pick up the job request from the queue and run the requested job.

After deploying the scheduling system and having success in running the first few nights, some new requirements come in from the users. The support staff that monitors the jobs at night need to be able to change the currently running jobs.

These changes they need to make include:

- mark a as job as already having completed
- restart a job that is in an error state
- prevent a job from running at all
- resume a job that had been marked to not run

All of these new features need to be shown as "actions" to take for a specific job, from within the administrative website of the scheduling system.

Inventory of Existing Queues and Exchanges

Before diving in to the new requirements, it's worth checking the existing topology of queues and exchanges. There may be some existing part of the RabbitMQ topology that can be re-used.

Unfortunately, reading the current documentation doesn't give you much to work with. You haven't found anything about the RabbitMQ setup in the docs, so you decide to dive in to the code directly.

Reading the code produces some useful information, but running the system to see what messages are sent provides the most value. Through exercising the system, you find several exchanges and queues that are related to your new feature requests.

Table 1. Existing Job Topology

exchange (type)	routing key	queue
job-request (direct)	job-request	job-request
job-result (direct)	job-result	job-result
job-log (fanout)		file-logger db-logger sys-logger

It looks like the existing exchanges and queues have an easy to follow pattern in their naming. It's a simple pattern, and it makes sense. None of these exchanges look like they will support your new requirements, but that doesn't does phase you. With this simple naming convention, it looks like you'll have your new requirements configured in RabbitMQ, quickly.

A Simple 1:1 Exchange:Queue Setup

Following the pattern from the existing topology, you'll need to add 4 new exchanges and queues to your RabbitMQ configuration. Each of these exchanges will correspond to the feature requested, with a queue named the same as the exchange.

Before digging in to the code for this, however, you decide it would be a good idea to save the documentation that you previously started. From there, you add your own exchanges and queues.

Table 2. Updated Job Topology

exchange (type)	routing key	queue
job-request (direct)	job-request	job-request
job-result (direct)	job-result	job-result
job-log (fanout)		file-logger db-logger sys-logger
job-complete (direct)	job-complete	job-complete
job-restart (direct)	job-restart	job-restart
job-dotNotRun (direct)	job-doNotRun	job-doNotRun
job-resume (direct)	job-resume	job-resume

Satisfied that your design has followed the existing naming conventions, you save your documentation to the project wiki.

There are other people also working on the project, as well, and they might appreciate the documentation. So you email everyone and let them know about what you've worked up. The response from your team is positive, thanking you for your effort.

A few moments later, however, the project's tech lead pops in to your cubicle to discuss your proposed exchange and queue setup.

Reducing Topology Bloat

The conversation with your tech lead starts out positively. She's very thankful for the time and effort in documenting the existing topology. That was something she had meant to do a long time ago, but she kept getting pulled in to more critical things.

After some additional discussion about what you found in your research of the code, she admits that the current setup for exchanges and queues isn't ideal. It works, clearly, as your system has been running in production for well over a week now. But there are some bad decisions left-over from almost a year ago in this topology - choices that would have been very different if she were making the same decisions today. Over-all, she's ok with the way things are, though. After all, the need for change is a natural part of the learning process with a new technology and with the growth of an application.

Before heading back to her work, the project tech lead says that she wants you to look in to a different approach for these new features. She want you to keep the number of exchanges down to a minimum instead of adding a new exchange for every message.

The goal, she says, is to use routing keys and bindings more intelligently with fewer exchanges. By

her quick estimation, you should be able to use a single exchange to enable the four new features that you're working on.

You don't need to worry about the existing topology, though. Fixing that can come later, when you have more time.

A Single Exchange: Commonalities

It's frustrating to have your ideas shot down like that. But, you're happy to be given a chance to fix things and help create a new convention for your application's RabbitMQ configuration. With curiosity and a bit of trepidation, you dive in to some resources for exchanges, queues and bindings.

After some scanning of documentation and ebooks, it's easy to rule out fanout exchanges for these new features. You don't want to send these new messages to every single queue bound to the exchange. That would require your code to be intelligent and know which messages it can handle. From previous conversations about RabbitMQ, you know this is a bad idea. This kind of behavior is built in to the features and capabilities of RabbitMQ's routing, and coding it yourself would needlessly complicate things.

The decision between a topic exchange and a direct exchange isn't quite as simple, however. Both of these exchange types would get the job done. Both would allow you to have a single exchange with queues bound to routing keys for each type of message.

Which one should you choose, then?

Taking a step back to look at what your features are, you notice a common theme.

- Job Complete is a change in the job status to say this job is complete
- Job Restart will put the job in to a status of restarted
- Do Not Run Job will also change the status so that your scheduling system ignores the job
- Lastly, Job Resume reverts the "do not run" status, letting the schedule run the job

You could create a set of routing rules and bindings for each status, sending messages from one exchange to four different queues. That solves half the problem, but still doesn't answer the question of which type of exchange to use.

Using a topic exchange would let you add extra features, if you need to. You could log the status changes using pattern matching on the routing keys, for example. And with pattern matching, you would also be able to send the status change to the right queue. in a way that is more flexible than with direct exchanges. But in thinking about this, you remember that there are logging messages in your code already and you don't want to duplicate those. Perhaps a topic exchange adds complexity that you don't need.

With that in mind, you decide to create an easier binding setup with a direct exchange. The features in question deal with changing the status of a job, making it obvious to you that this should be

a “job-status” exchange. Following the existing naming convention, the routing keys and queues should named for the state of each feature request as well.

The resulting change to the topology looks like this:

Table 2. Single Exchange Topology

exchange (type)	routing key	queue
job-status (direct)	job.complete	job-complete
	job.resume	job-resume
	job.doNotRun	job-doNotRun
	job.resume	job-resume

After updating the project wiki, you check with your tech lead to see what she thinks about this topology design.

Are These Just Status Updates?

The conversation with your tech lead starts out great. She’s happy that you found a way to use a single exchange, and she likes the routing keys and queue names that you chose.

When she gets to the exchange name, though, she questions why this is a job “status” exchange. As you are explaining how each of these new features will result in a status change, she nods in understanding. But, something about this is still bothering her, judging by the expression on her face.

A moment later she has a question, wondering if the purpose of these messages is to change the status of the job or if that is a result of the action being taken. Slightly confused by the question, you ask for clarification while again pointing out how the job state will be affected in each case.

Your tech lead stops for a moment, to think. Then, with a look of clarity in her eyes, she then asks you to describe the code that you would write if this were not a message-based architecture. She wants you to think about this in terms of the objects being manipulated.

The answer to these questions come quickly and you describe how the job object would have a “complete” method, a “doNotRun” method, a “resume” method and a “restart” method. Each of these methods would handle the necessary logic to complete the requested action, resulting in the status of the job changing to …

Aha! That’s what she was getting at a moment ago!

The feature requests are not merely changes in the state of the jobs. They are commands to take action, and they happen to result in the state of the job being changed. Saying a job is now “complete”, for example, has to shut down the current job if it is executing, stop all the code related to waiting for that job to complete, and more. Only after all of these actions are taken, will the job be set to the “complete” status.

But… how does that change the way you would lay out the exchanges and queues?

After asking for clarification while considering the command-like nature of methods on objects brings only more question, however. The tech lead tells you that she'll leave it up to you at this point. She wants you to make the final decision on whether or not the current exchanges and queues are the right way to go.

The only hint she gives you in solving this puzzle is to consider the intent of the exchange, and the messages that will travel through it.

Command and Conquer

Frustrated by the lack of answers, but determined to find the "right" solution, you dig further in to the idea of commands.

There has recently been some discussion in your team about command/query separation. That is, having methods that either query the state of things, or command things to take action - but never both at the same time. These ideas have you wondering about the relationship between commands, queries and messaging.

If you were building objects, you would have command methods to restart the job, resume the job, etc. A job is the object on which actions are being taken, or commands are being executed. The job has command methods.

Maybe these thoughts around objects can be translated to the message topology as well? What would that look like?

This train of thought feels like beating your head against a wall, so you walk away from your desk to get some fresh air.

While you're taking in the warm sun and cool breeze outside, a friend a former soldier in the Army comes out to chat. They recount yet another story of a drill sergeant berating them, and commanding them to run 10 miles for the 3rd time that day.

As you listen to your friend's story, their voice trails off in to the background. Your thoughts have gone back to the design problem, but are now intertwined with the drill sergeant's commands.

Something clicks. A rush of excitement is felt, and you thank your friend profusely as you rush back to your desk, leaving them confused and wondering how they helped with anything.

Back at your desk, your mind is racing.

What if you thought about the methods on an object not as command methods, but as command messages? Like the drill sergeant barking orders at someone? The sergeant isn't executing a method on your friend to make them run 10 miles. The sergeant is commanding your friend to run 10 miles with a message… a very loud, rude and degrading message… but still, a message!

So, if a job is an object that has command methods, perhaps a "job" can be an exchange through which command messages are routed!

It makes sense. But, would it work? And what about the type of exchange to use? Would it matter if you stick with a direct exchange or should you go back to a topic exchange?

If you're dealing with an object that has a method, then you're directly calling that method on the object. You don't need fuzzy logic or pattern matching to figure out which methods are the appropriate ones to call. You are directly calling the method.

So, topic queues seem to be out - at least for this scenario. You don't need pattern matching for queues because you are directly commanding work to be done on a job.

With a direct queue, then, you can model the same relationship as object->method. In this case it will be exchange->(routingKey->queue). The object "job" is now the exchange "job". The routing key + queue are now the name of the command (the object's method name), and the contents of the message tell the code on the other end how to execute the command (what parameters to use).

With this in mind, you rebuild the wiki documentation for your new job topology:

Table 3. Command Message Topology

exchange (type)	routing key	queue
job (direct)	job.complete	job-complete
	job.resume	job-resume
	job.doNotRun	job-doNotRun
	job.resume	job-resume

Upon looking at this you realize the only significant difference in this new topology, compared to the first, is the name of exchange. But you also know that the intent is very different. And much like the design patterns that you deal with in object-oriented code, you guess that the intention of the topology is as important as the implementation.

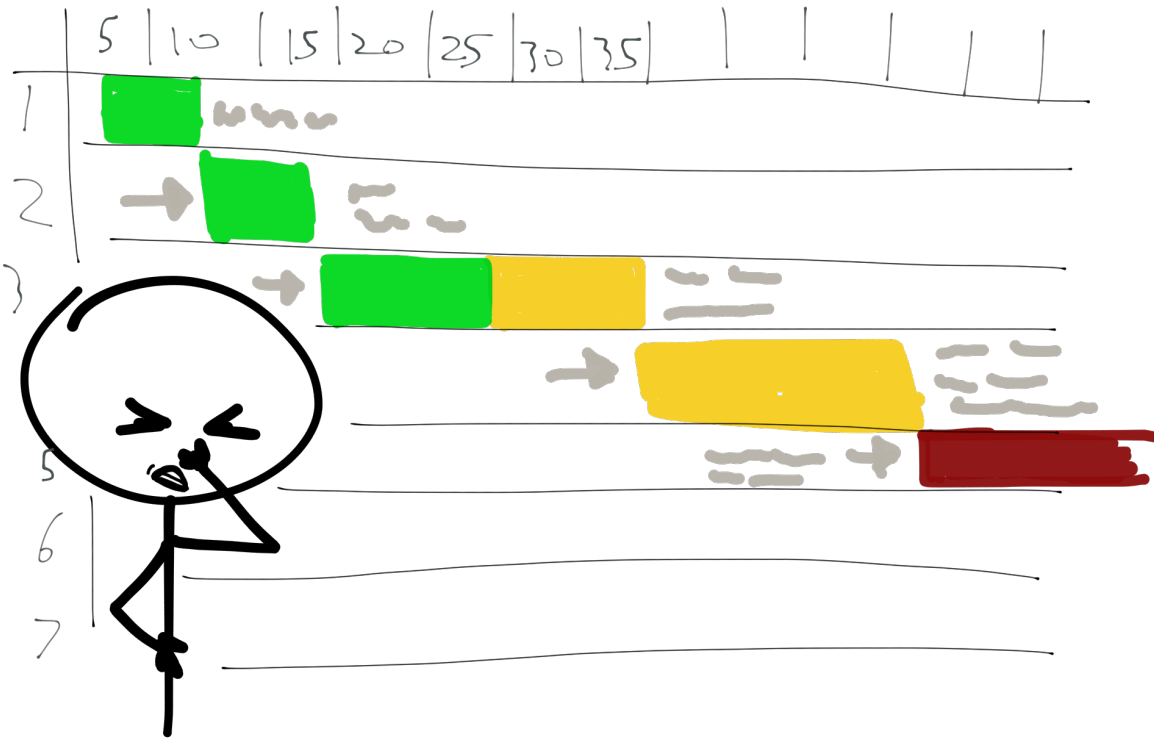
Epilogue

After taking some time to review your plan and ensure you have thought through the entire process, you know you're ready. You've covered all the scenarios you can think of. And, you have documented the new topology with notes to explicitly state the command oriented intention of this exchange and it's bindings. Rather than sending status updates and inert "documents" that only contain data to be stored, the intention of this topology is to send commands - message that carry instructions on what and how to execute the code on the other end of the queue.

Satisfied with your efforts and the results, you bring the project tech lead back to your cube to discuss the new layout. On the short walk to your cube, she reads the documentation changes you made, via her phone.

A smile slowly spreads across her face.

Chapter 6: Reducing Response Time



It's been a rough year with the new application.

Sure, it launched well. It was well received by the first users, too! But now things are different. People are complaining about how slow the app is and your manager is breathing down your neck to fix the performance problems.

Today is no different. You've got another issue ticket to fix a performance problem waiting for you when you get to your desk. Only this time, you're not sure what you can do.

The problem is in the file downloads. It takes twenty to thirty seconds to start, and people are canceling the download before it starts. This is making the stats look wrong and causing complaints from a lot of users.

Before you get started for the day, you check with the tech lead to see if this is an old bug. After reminding him that the Content Delivery Network ("CDN") went in to place two weeks ago, you're wondering if this bug didn't get tracked fast enough. Maybe it was left over from prior to having the content delivery network serving files? Shouldn't the CDN have taken care of the download speed problems?

Sadly, the issue was filed yesterday… by your own project manager. She knows about the CDN, clearly, and she said that it does help the speed of downloads once the download starts. But, now the problem is waiting for the download to begin in the first place. It takes too long to start and this is where people are canceling and complaining.

Documenting The Process / Problems

Finding the problems takes a few days of grueling work. Tracing through code and running profilers is not your idea of fun. Involving network specialists in the IT department makes it that much more painful, too. You had to rule out the network configuration, though. And of course it wasn't that, to your chagrin as the the arrogance of the network team stood up to the test.

After countless bad paths and red herrings examined, you're starting to get a clear picture of what's happening, though. Maybe. At least, you think you are. Hopefully.

The problem seems to be centered around a few key areas in the code. From a high level perspective, there are too many calls out to network services. Some of these service are known to be unreliable when under heavy load, but your system must ensure they are called. You can't send a file back to the user without these calls. They are not optional.

To make your case on where the problem is - and hopefully get some insight in to why these services are used - you document the complete flow from the initial download request to the HTTP redirect when the user is given the location of the file on the CDN.

It is not a pretty sight.

Table 1. Download Request Process

Download Step	HTTP Code	Avg. Resp. Time
1. download_controller receives HTTP request		
2. authorization check for file		5 sec
2a. send authorization request across RabbitMQ		
2b. wait for response from RabbitMQ authorization check		
2c. if not authorized, return HTTP not authorized. exit.	401	
3. CDN call to verify existence of file		12 sec.
3a. if file not found, CDN call to get file from source		

Table 1. Download Request Process

Download Step	HTTP Code	Avg. Resp. Time
3b. if the existence check fails, try again 1 second later		
3c. if 3 failures, send server error. exit	500	
4. send HTTP redirect to client, to start file download	307	3 sec.
	Total:	20 sec.

Armed with your analysis and average response time data, you schedule a Q&A session with the project manager and tech lead. Hopefully they will have some direction for you. The problem of high latency, error prone and slow services is a good combination for having a bad time.

Questioning The Current Design

Your tech lead and project manager are both late. Typical. Although, you do have to cut them some slack since they were in a meeting with the executives ... probably getting their faces ripped off with more complaints. Better them than you, right? At least they know how to handle that kind of meeting.

When they finally arrive, you hand each of them a copy of the analysis and response times documentation. Both of their expressions change from the pain of the previous meeting to that of being impressed with your analysis. That, at least, is a good way to start the meeting.

Your analysis is insightful. The tech lead knew that each of these services were in place, but the project manager had never looked at the complete picture until now. Having all of this collected in one document helps them to understand the scale of the problem.

With the pleasantries of your effort out of the way, they want to get in to the details as quickly as possible.

Why is RabbitMQ in Here?

Your first question, going from the top of the list, is about the authorization check. You know what RabbitMQ does - at least at a very basic level - but you're not sure why it was needed at this point in the process.

The project manager admits that this was a left-over design decision from almost 2 years ago, before you were on the project. The original requirement for authorization was fuzzy at best. The team at the time decided to hide the decision behind a request/response setup to isolate changes they expected. They had been playing with RabbitMQ for some new ideas, and thought it would be a good fit.

While the RabbitMQ calls started out fine, it added an extra layer in between the app and the authorization check. In the end, the real authorization check happens very quickly. The high latency for the call comes from poorly written RabbitMQ code and having to travel through that code multiple times. Maybe you can you do something about that now?

The project manager didn't like the idea of using RabbitMQ when it was first suggested. It was the "new toy", she said, and that made her nervous. But, at the same time, she wanted to leave the technical decisions to the technical team. In hindsight she wishes she had at least asked more questions about the decision.

You have more questions about RabbitMQ, but you also want to get through the initial list before diving in deep on any subject. And, hopefully the CDN calls will be an easier problem to tackle.

Do We Have To Verify The File, Every Time?

Looking for at least one small win, you ask whether or not the code needs to verify the file on the CDN every time it is called. With that question, you finally get some good news: no, you don't!

… but …

You cannot send allow downloads of a file until you know with certainty that the file is there. Once you know the file is there, you could cache that knowledge. If you can do that, you wouldn't have to check every time.

Unfortunately, the CDN is not optimized for this check - and everyone knows it. As your site traffic has increased, the load on the CDN for this check has grown constantly. The CDN provider has complained about your system many times, and so has your team. It needs to be fixed.

Perhaps there is a way around the current CDN problems with the file existence checks. The response could be cached, as your tech lead noted. But at the same time, you don't want that first call to have to wait for the check. If there were multiple calls for a non-existent file at roughly the same time, you would still make the file check multiple times. That is not what you want.

Before digging in to the CDN checks further, though, you decide to move on to the last issue.

Redirect Issues?

The HTTP 307 redirect is really only a problem on high latency networks and mobile devices, it turns out. There isn't much you can do about that… at least not in the immediate future. This is up to the end-user's network latency and speed.

Luckily, the project manager tells you not to worry about this. Fix the other problems, as they are in your control.

Any Ideas?

Wanting to continue the conversation, you ask about ideas for solving the CDN problem. You're wondering if there is some way to ensure that the check for the file existence only happens once. If the one check can be guaranteed, and the result can be cached, then a lot of that problem would go away.

Before he can answer, though, the tech lead gets a phone call that can't be ignored and steps out of the conference room.

While he's out, you ask the project manager about the RabbitMQ problems. Knowing her technical background, you're hoping she'll have a good understanding of the issues. Unfortunately, she doesn't remember the exact problem. Issues were noted a long time ago, but no one had time to fix them. She remembers something about opening too many channels or connections, but isn't entirely sure what that meant since she had not been in the code at that point in time.

Before you can dig deeper with more questions, the tech lead pops back in to the room. The call he had taken was from another tech lead on a related project. Both he and the project manager were needed for yet another upper management meeting about the status of problems.

Just before leaving, the project manager suggests you read up on RabbitMQ. She thinks it should be removed, but wants you to make sure this is the right decision. The tech lead also suggests reading the how-to articles and basic API documentation for the CDN. There's probably something in the CDN that would help.

As they walk out the door, your frustration grows. Why should you read up and learn more about RabbitMQ if you're just going to can it? At least the tech lead's request to read about the CDN seems a little more relevant, even if it does leave you with more questions than answers.

Research and Planning

When you first joined this team, the tech lead gave you a bit of advice: "solve the hard problems, first. The rest of it will fall in place." While this may not always work out the way he said it, it has generally been good advice.

RabbitMQ seems to be the hard problem at the moment, so you dig in to see what's going on.

RabbitMQ: Friend or Foe?

The work is tremendously frustrating at first. You're only aware of what RabbitMQ does and how it works, and you're having a hard time wrapping you're head around it. The idea of channels vs connections doesn't make sense. What are exchanges, and why can't you just write a message to a queue? Why would anybody want this extra layer in between servers, anyways?

You spend countless hours of reading, watching screencasts, and trying to write at least a basic working example of RabbitMQ, over the next week. Through this, you begin to see some problems

in the current application code. According to what you've read, each process should only have one connection to RabbitMQ. It looks like the code in the authorization request is not doing that.

… Really?! …

How could the previous team not understand that? You've only spent a week on this, and you know at least that much. But, whatever. There's no point in getting mad about it now.

Researching the authorization request further, you're starting to find RabbitMQ to be somewhat intriguing. But, the more you read about the authorization API, the more you think that RabbitMQ should be eliminated from the equation. The authorization API is a well-known end-point on your internal servers. This service provides access rights to hundreds of thousands of users across dozens of applications, every hour. Someone on the original team for your project clearly didn't understand that.

RabbitMQ is part of the problem in the authorization request, in your opinion. But, is it really all that bad? Or was it just applied incorrectly - or in the wrong place - in this project?

For now, you have a plan to fix the authorization request. It's time to move on to the CDN and see what can be done, there.

Don't Call Us. We'll Call You.

The research on the CDN is significantly faster and easier than RabbitMQ - which is a much needed relief after last week.

It turns out the CDN has a rather extensive API that you can use. The check for the existence of a file is only a small part of it, and wasn't meant to be used the way your project is currently using it. This particular method is part of an out-dated API set that was originally intended to be part of a push based strategy for delivering content to the CDN.

In looking at the most recent API versions for the CDN… (well, ok - the API versions that have been around for at least as long as your project has been going… who wrote this code in your project? and why didn't they use an up to date API when they did?) … you've found a webhooks section in the documentation. It looks *very* promising.

Checking if a file exists when a user requests it is a bad idea, as you already knew. With the webhooks that the CDN provides, you won't need to. The CDN will tell you when the file has been published. You can have your system tell the CDN to pull the file up, and then wait for the webhook notification before saying the file is available to download.

The result will be a mild slowdown in the time from someone uploading a file to your service, to the time that it is available to download by others. But the resulting increase in speed of starting a download will be worth it.

The real question will be how to handle the webhook. The documentation for the webhook says the endpoint it calls must be available at all times, and must respond within 100 milliseconds. If you drop or miss too many requests, the CDN will blacklist your webhook … And you know, from

recent discussions with the network people, that your back-end systems are often unavailable. The web server stays up pretty much all the time, but the services on the back-end that would need to be updated with the CDN info do not.

But if you could accept the API call for the webhook via the web server, and then process that message at a later time (when needed…) that would certainly make life easier. Then you could have the webhook respond to the CDN quickly, like they require, and not worry about the back-end services being down.

With a look a both revelation and confusion, you suddenly have an answer - or at least a starting point.

Grabbing your incomplete notes, you walk over to the tech lead and ask for a quick meeting with him and the project manager.

You Want To Use What?!

Your tech lead is thoroughly confused. He thought they told you to get rid of RabbitMQ. And, now you're saying that you want to use it more?!

The project manager is only smiling, at this point. It appears she either understands what you want to do or is just having fun watching the tech lead squirm. Maybe both.

Calming the tech lead down takes some work, but you explain to him that yes, the problem with the authorization calls is partially RabbitMQ. The real problem with RabbitMQ is how it was coded, though. Explaining the difference between a connection and a channel is difficult, but he finally understands your point. The RabbitMQ code in the authorization call is terrible and wasting network resources on an already high network traffic server.

With a sense of relief washing over the tech lead, you explain that you do want to remove RabbitMQ from the authorization call. It shouldn't be used there. The authorization checks should make a call to the service directly, like every other system in your company.

However, the CDN problem is going to be solved with a more appropriate use of RabbitMQ.

Webhooks are Messages

The epiphany came, you explain, as you were trying to figure out how to deal with the CDN's webhooks. When you started thinking about storing the webhoook request and dealing with it at a later time, your first thought was to put the request in a database somewhere. But then the idea of integrating two separate systems through your database made you uneasy, and you wished you could just send the webhook message to another server.

The word "message", in your own thought stream, brought RabbitMQ rushing back. Of course! RabbitMQ would solve this problem elegantly! Your webhook will send a message through RabbitMQ and the code on the other end will pick it up when it's good and ready. It doesn't matter

if the back-end systems go down or not. When they come back up, they'll start pulling messages from the queue and processing them.

The project manager is now smiling directly at you, instead of smiling while laughing at the tech lead. Clearly, she likes the idea and she says so in no uncertain terms. Reluctantly, the tech lead agrees.

Designing The RabbitMQ Solution

It takes more time than you had expected to come up with a good solution for the proposed use of RabbitMQ. It turns out there's more to RabbitMQ properly than you had previously thought. The connection vs channel makes sense now, but the notion of which type of exchange to use is confusing.

Your first solution is to just throw everything in to direct queues and to pile up as many exchanges and queues as you needed - with no real routing of messages to speak of. But your project manager has more knowledge of RabbitMQ than she let on before. She quickly points out the confusion and explosion of exchanges that this would cause over time, and she wants you to look at using routing keys.

With a little design assistance from the project lead, and a lot of code review from your tech lead, you head down the design path of the RabbitMQ topology.

The Webhooks

Your website has a dedicated API sub-domain, so the webhooks end points will live there. Once the API receives a webhooks call, it will send a message to RabbitMQ to say which file is ready. The CDN allows you to specify arbitrary meta-data with your file uploads, and it sends some of those fields back to you through the webhook. This lets you easily identify which file upload it was that completed.

After initially wanting to use a direct exchange for the webhooks, the tech lead informs you that there will likely be more webhooks to handle in the future - from places other than the CDN. Considering this for a moment, you're still ok with a direct exchange. Routing keys will handle the needs of adding new webhook sources.

The layout that you design for the first version of the webhooks solution looks like this:

Table 1. RabbitMQ Layout

Exchange	Ex. Type	Routing Key	Queues
webhooks	direct	cdn.file.uploaded	cdn.file.uploaded

Explaining how the routing keys are matched, you tell your tech lead that this configuration will

allow you flexibility in the future. You can add different words to the various parts of the routing key, telling RabbitMQ to send the messages to different queues.

You had thought about using a topic exchange in this case, but didn't see a need to have flexible routing keys for the webhooks. You're confident that these messages only need to go to one place. And if you need a message to go somewhere else, you can add a new binding.

The tech lead isn't quite as confident about using a direct queue, and wants a topic queue. But he leaves the decision to you, wondering if you'll be able to change it later if you need to. While you're not one-hundred-percent confident in being able to change it later, you think it should be possible and decide to leave the exchange type as-is… for now, at least.

Epilogue

Almost four months after your initial investigation, your solution to the performance problems has been a tremendous success. Within the first first few days of release, the support team noticed a dramatic reduction in complaints about download speeds. Within the first month, these complaints had all but stopped.

A New Request

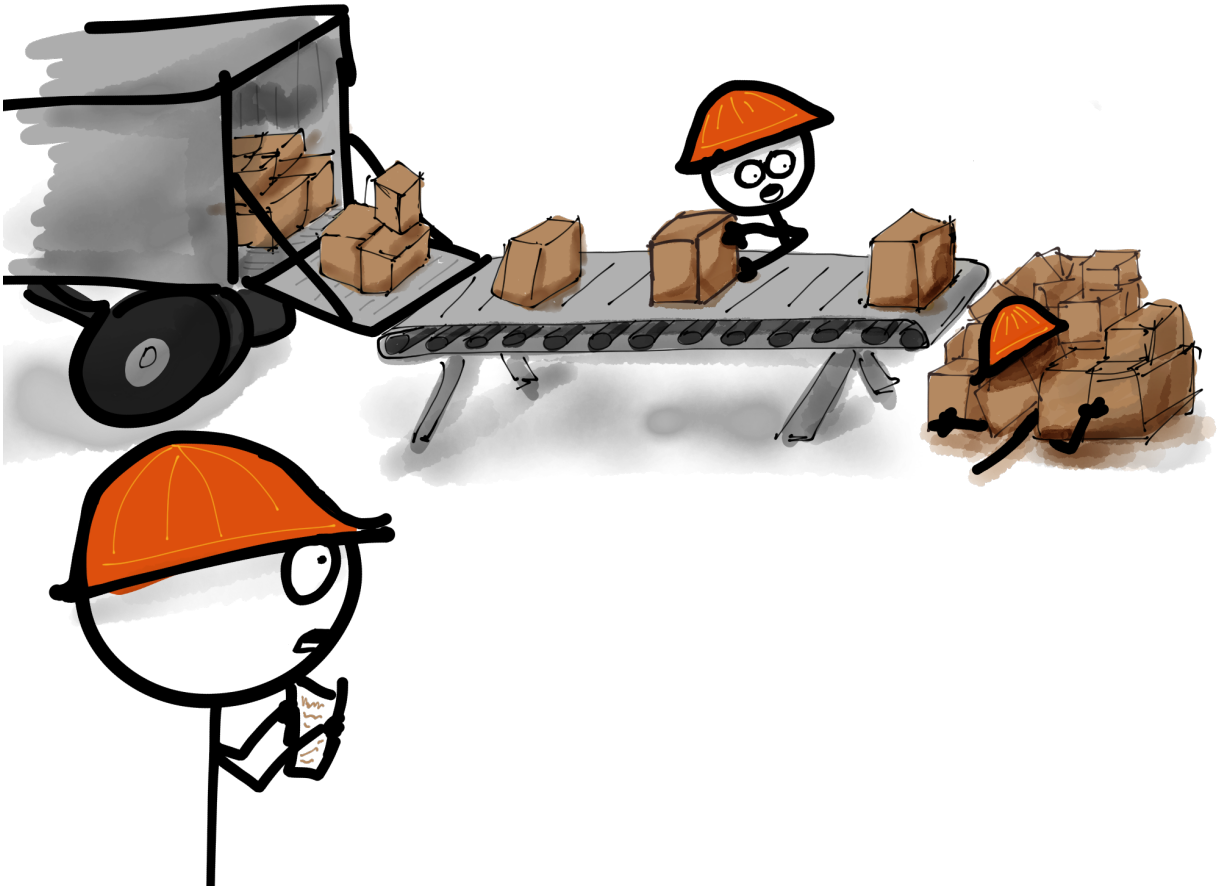
You've been riding high on the success of this deployment for well over a month when the project lead stops by your desk. She has a new request for you, from some new complaints that the support team is receiving on an increasingly frequent basis.

Some of the people that upload files to the system are getting confused and uploading the same file multiple times. Nearly every person interviewed about the problem has said they didn't think the file made it up to the service. The small delay - typically less than a minute - between uploading a file and seeing that is as available, is causing them to think the upload failed and they are uploading the file a second time.

The project lead wants you to investigate solutions to this new problem. Perhaps some kind of status message about the upload, letting the user know that it is being processed, would help. She asks if you could do something like that, or find another solution to the issue.

With a smile, you recall a recent screencast series that you purchased. In one particular episode, you saw how RabbitMQ can be used to send status updates for long running background processes to a web application. Looking up at the project lead, you let her know that you have just what the support staff needs and you should have the solution ready well before the upcoming release of the system, next week.

Chapter 7: Increasing Workload and Workers



Your podcast hosting service has been growing for the last 6 months. Slowly, but surely, you're earning new customers and publishing more podcast episodes for them. Then, recently, you landed a couple of "big" podcasts - big by your standards at least. Having these two podcasts on your service is proving to be tremendously beneficial, too. The hosts love what you have built and are telling everyone about you!

In spite of the problems you have run in to, things are going well. You're loving this little side project that you started building just over one year ago. Customers are happy, and you enjoy listening to the shows that are being served by the service you provide.

Slowly Getting Slower

Performance is a continuing problem with your system, it seems. In spite of all the effort you've put in to making this thing fast, the continuing increase in traffic means everything continuously gets slower.

You've got the worst of the problem behind you having put RabbitMQ in place recently. But, you still have performance issues that need to be dealt with.

Adding RabbitMQ

Thinking back to a few months ago, you noticed a problem with the performance of the application. As traffic was increasing, the response time of the episode downloads was increasing. To solve this, you added RabbitMQ and off-loaded most of the long running work to a back-end process.

In particular, you had noticed that there were times when the analytics service call was slow to respond. By adding RabbitMQ in to the mix, you were able to move that call to a separate process. This made the front-end system significantly more responsive, effectively solving the performance problem.

The layout for your RabbitMQ set up was simple, since you only had the one background process.

Table 1. RabbitMQ Layout

Exchange	Ex. Type	Routing Key	Queues
analytics	direct	(none)	analytics

But now you find yourself needing more than just this one analytics call, running from your back-end process. The analytics service has sometimes been unavailable, due to network issues between your hosting provider and the analytics service. This failure is not something you want to tolerate, but you don't want to leave that service behind, either.

Try / Catch Database Calls

The first idea that comes to your mind for solving the unavailability, is to retry the analytics call. You can put a `try / catch` around the code that makes this call. If it fails, a simple `setTimeout` with a delay of a few minutes should suffice for retrying.

That seems a simple enough fix, but you're also concerned about having too many messages stuck in an infinite loop. Adding a counter to check for the number of times a message was retried is simple enough and solves that problem.

An initial retry limit of three times seems reasonable, with a 1 minute delay in between the retries. It's been a rare occasion that your network hiccup has lasted more than a minute, so this should cover the majority of the problems.

A Growing Team, A Growing Problem

Your service has continued to grow, and you're having the time of your life supporting everyone. Things are getting a little more difficult to keep up with, though, and it's about time to bring on a second person - at least part time - to help out with the code while you handle support requests.

Fortunately, you know who to bring on - a former coworker that has been keeping up with your work for the last year. She's a phenomenal developer, a good friend, and she's got some RabbitMQ experience which means she'll be able to pick up that side of the work without a lot of training.

You Did What, With These Failed Calls?!

Your new teammate / contractor - and good friend who's opinion you trust - just about hit the floor when she saw your recent "fix" for the latency in the analytics service.

Once again, you explain that the three tries for the analytics service are there because of the high percentage of dropped connections when trying to write data. The database is there as a backup for when the analytics service fails. It logs each request that should go to the analytics service. And more recently, you added a nightly batch process that looks for entries in the database. When it finds entries in storage that are not in the analytics system, it makes the needed adjustments.

Yes, of course she understands that. There are enough comments in the code to make that process apparent. The real issue, according to her, is the tight coupling of the database and analytics calls. She says something about how this code misses half the functionality of what RabbitMQ can do for the system, but at this point your frustrated reaction is preventing you from really understanding what she is saying.

Fortunately, you recognize your own bad reaction before you respond verbally and things get really out of hand. Calming yourself for a moment, you ask for a better explanation of what she sees as the problem and what she would do to fix things.

Fire And Forget

Seeing your frustration in your face, your new teammate calms herself and takes a moment to collect her thoughts before continuing.

Before starting the explanation for which you asked, she does say that you're on the right path. She is glad to see RabbitMQ already handling the majority of the work. That, at least, is a small win and makes you happy to hear.

By putting RabbitMQ in between the web server and the database and analytics services, she knows you were able to reduce these services to a single network call that does not wait for a response.

That "fire-and-forget moment" - where you shove a message in to RabbitMQ and let a separate back-end process handle the call to the database server and analytics service - reduced the response time on the site significantly. The back-end process can take all the time it wants, at this point - a

near infinite amount of time if needed. But having the database call and the analytics call wrapped up together is going to cause additional problems - maybe some of the current problems that you're seeing.

Separating The Two Operations

When you were first talking with her about coming on to the project, you had explained some of the issues that you're currently having. One of them - probably the most important one at the moment - included a growing wait time for analytics data to be available.

When you first put the RabbitMQ solution in place, the calls to the analytics service were keeping up just fine. People were able to see their analytics updating in near real-time. As the customer base continues to grow, though, the analytics are getting slow and slower to be updated.

You had looked at the RabbitMQ server to see what was going on, and found your queue to be backed up to nearly 10,000 messages. The current rate of processing only a few messages per second wasn't cutting it anymore.

As your new teammate is recounting these details, you are nodding in understanding but wondering where she's going with this. Fortunately, you realize she is just about to get to the point.

A moment later, she is recommending you separate the database write from the analytics write, for several reasons:

1. They both require network calls that are "expensive" (take a long time)
2. One is not actually dependent on the other succeeding - they can both succeed or fail independently
3. With proper routing keys and bindings, there won't be any additional network calls from the web server to RabbitMQ
4. RabbitMQ can help you with the failed tries, offering more robustness than an in-memory counter with try / catch blocks

Slowly, the light bulb in your mind begins to glow with understanding. If you split the two processes, then they can run independently. Removing the dependency on the database calls means the analytics call can run faster. That should reduce the time to write the analytics data, hopefully returning to a near real-time analytics report for your customers!

You're not sure about the retries, off-hand, but that's ok. That problem can be solved later. For the time being, you'll start with separating the two processes by using a more intelligent routing key.

Re-Routing The Database and Analytics Message

The decision to update your RabbitMQ configuration and back-end processes was easy enough. The debate over the exchange type to be used, on the other hand, isn't as quick and clean.

Your first pass at the topology involves adding a new exchange to your configuration, so you can send another message through RabbitMQ.

Table 1. RabbitMQ Layout

Exchange	Ex. Type	Routing Key	Queues
analytics	direct	(none)	analytics
db	direct	(none)	db

Having written that down, though, it doesn't make sense. You were assured that the solution could be implemented with a single RabbitMQ call. And, wouldn't the extra exchange require another call?

That's when your teammate reminds you that you can have multiple routing keys for a single exchange, sending a single message to multiple queues. She quickly recounts the types of exchanges that are available - direct, fanout and topic - and asks which you would prefer to use.

Thinking about multiple queues receiving one message makes you wonder if this would be a good use for a fanout exchange. There are a few reasons why this would be appropriate, in your mind:

- fanout exchanges send the message to all bound queues
- there is no need to worry about routing keys
- you have multiple services that need the message

But on listing these reasons, you suddenly find yourself less than convinced. It seemed like a good idea at first, but the list of reasons seems a bit shallow. Technically, any exchange type would work under these circumstances. Not needing a routing key doesn't seem like a great use case of a fanout exchange in this scenario.

Perhaps it would be better to use a topic exchange? This would let you have messages sent to multiple queues and gives the advantage of pattern matching the routing keys. You only need to add a new binding for each destination queue.

The Value Of A Topic

From your teammate's perspective, a topic exchange doesn't make sense here. It's not that a topic exchange wouldn't work - clearly it would, she points out. But what is the value of the topic exchange, really? Both a direct exchange and a topic exchange would support multiple destination queues with multiple bindings. A lot of the value in pattern matching for routing keys, though, is backward from what you were thinking.

A topic exchange is not important in that you can have a single message go to multiple queues. Any exchange type allows this. Rather, the value of topics and pattern matching is to have multiple routing keys go to a single queue. Pattern matching creates a rich set of capabilities when you need multiple destinations per message and potentially multiple messages per destination queue with as few bindings as possible.

Confused, you ask for clarification. The explanation you receive is what she calls the "classic example" of log messages.

She says that a log message could have a routing key like `log.appname.critcal` or `log.appname.debug`. Both a direct or topic exchange could take these routing keys and send the message to multiple queues. The downside is that you need to add a new bindings for each destination queue to make to work.

Table 3. Logging w/ Direct Exchange

Exchange	Ex. Type	Routing Key	Queues
logging	direct	log.fooApp.critical	log.critical
logging	direct	log.fooApp.critical	log.all
logging	direct	log.fooApp.debug	log.debug
logging	direct	log.fooApp.debug	log.all

A topic exchange would also allow the same configuration. And, honestly, to get a message to multiple queues, you will need multiple bindings. However, a topic exchange would allow you to have a single binding to send *all* of these messages to a single queue.

Table 4. Logging w/ Topic Exchange

Exchange	Ex. Type	Routing Key	Queues
logging	topic	log.#	log.all
logging	topic	log.#.critical	log.critical
logging	topic	log.#.debug	log.debug

Having the ability to log all log messages to a single queue is powerful. It allows unified logging across all applications, which can be stored and filtered later with an analytics service. By setting up a pattern matching of `log.*.critical` or `log.#.critical` routing keys, you can have any application publish a "critical" message to a queue that would be monitored by support staff. The code behind this queue could immediately alert any on-call staff of the critical problem. You could have new applications sending log messages of a critical nature, send messages that are specific to that application, and still have the support staff be notified of critical issues. Similarly, you can have developers receive debug notifications to help track down problems.

at this point you're not trying to send multiple messages to a single queue. Rather, you are trying to send a single message to multiple queues. This gives you a moment of pause, once again wondering if a fanout exchange would be most appropriate.

Try It and See?

Having mentioned fanout exchanges again, your teammate says she thinks you should try a topic exchange. It would require additional bindings if you need to add a new queue and process, later.

But having the topic based routing keys will make it more flexible. Not having a strong argument for either fanout or topic, you decide to go with her suggestion and see how it works out.

Before committing the changes to code, you write out the new topology on a whiteboard, updating the exchange type to be "topic".

Table 5. Updated RabbitMQ Layout

Exchange	Ex. Type	Routing Key	Queues
analytics	topic	(n/a)	analytics

Looking at this change, you see a problem with the naming.

Using the word "analytics" for the exchange and queues is too generic. What analytics are these referring to? Do you have more than one type of analytics that all need to flow through the same exchange? This should change to something more appropriate.

With the current name, it is possible for the exchanges and queues to being over-used with messages for other analytics that you want to add, later. That could cause big problems with the back-end code in the future, as it would have to be smart enough to handle a lot of different messages. It's obvious your teammate is not liking the current name either, based on the way she is looking at the whiteboard.

Thinking for a moment, you remember that a friend of yours had shown you how he likes to invert relationships when dealing with object models. The idea of taking what you thought was a child object and making it a parent that represents a new context seems to fit here. You currently have a high level abstraction of "analytics" with a child of "download". By inverting this, you would have a high level of "download" which needs to flow through both "analytics" and "logging". It makes sense in your head - how would it look on the whiteboard when you try to explain it?

Quick with a whiteboard eraser and marker, you make the change while explaining the idea.

Table 6. Inverted Layout

Exchange	Ex. Type	Routing Key	Queues
download	topic	download.logging download.analytics	download.logging download.analytics

By reversing the intended use of the exchange and queues, you explain that they won't be overrun with complicating code later on. The code that listens to the "download.logging" queue will only be concerned with logging the download request to the database. The code listening to the "download.analytics" queue, on the other hand, will only be concerned with sending a message to the analytics service for a download request. It will never have to deal with other analytics requests.

The explanation sits well with your teammate, but she points out the routing key needing to be the same for both of these bindings. If you don't use the same routing key for both bindings, you

would have to send multiple messages through RabbitMQ. And that is certainly something that can be avoided in this scenario.

Table 7. Correcting Routing Keys

Exchange	Ex. Type	Routing Key	Queues
download	topic	download.requested	download.logging
		download.requested	download.analytics

The room is all smiles and nods, looking at the new layout of queues and exchanges.

An Overloaded Queue

It has been almost two months since you changed the way the analytics and database logging for analytics calls were made. The code has been running beautifully, and you're happy with how easy it has been to update the code as needed. Keeping the database call and the analytics call separate has allowed you to change each of those responsibilities independently of the other. That alone was worth the time you spent to make the original change.

With the continuously increasing number of customers, however, your backlog of requests hasn't completely cleaned itself up, yet.

Drain, Delete, or Transfer?

During the initial switch over to the new layout, you kept the old queue handler in place for a while. There were, after all, more than 10,000 messages sitting in that queue to be processed. Rather than trying to move those messages to a new queue, you decided it would be easier to just let the current code finish the existing messages while the new code handled new ones.

The strategy worked well, found in an article by your teammate, worked well. She said this was called "draining the queue", and that it was one of three options she read about.

- "drain" the queue
- delete the queue and all the messages
- transfer all the messages to a new queue

There was no way you could delete the queues with all of the messages in it. That would have lost days worth of analytics for your customers, and made a lot of people very unhappy. The amount of work involved in transferring the messages to a new queue seemed like it wasn't worth the trouble. You would have to re-write the current queue handler code completely. Instead of actually handling the messages, it would transform them in to a new message and re-publish them to the new exchange and queue. It would take a few days of coding and testing to make sure this worked.

By your estimates this would be longer than the time to just process all the messages. This would also keep you from other work, pushing everything back further than you wanted.

In the end, it seemed like the "drain" would be the best option. With your teammate's agreement, you left the existing queue handler code in place and let it run it's course while you worked on other things.

And it did work! It took a few days to drain, but the old queue eventually went to zero messages.

After letting it sit for a few more day just - just in case - you had your teammate shut off the old code and delete the old queue. Since then, both of you have been monitoring the new queues. Slowly, but surely, you've seen the backlog of messages creeping up once more.

A little disheartened and more than a little bit frustrated, the two of you head back to the white board to discuss potential solutions for the continuing backup of message.

More Queues?

The primary problem, it seems, is the continuing network hiccups between your servers and the analytics service. You've contacted both your hosting provider and the analytics service and neither of them have been able to figure out where the problem is. And while it would be possible to move your server to a new host, that isn't something you want to do right now. There are far more important things to take care of, since you already have several layers of insulation for this particular problem.

The backup of messages in the queue, however, is not a problem that you want to ignore.

The discussion starts with a quick outline of the exchange and queue layout, and the code in question.

Table 8. Current Topology.

Exchange	Ex. Type	Routing Key	Queues
download	topic	download.requested	download.logging
		download.requested	download.analytics

1. Download is requested
2. Message is sent to "download" exchange, using "download.requested" routing key
3. Web server sends file to requesting application and the web server is done
4. Download requested message is routed to the logging and analytics queues
5. The logging queue is handled, putting a record in the database
6. The analytics queue is handled, attempting to send to the service
    1. If the analytics call fails, wait for a minute and try it again
    2. If the message fails for a 3rd time, send an email to say there is a problem and throw away the original message

3. Failed messages will be handled by the database log and a nightly batch process

You've been thinking about this for a while now, and your teammate has been as well. It's good to see the entire flow of code listed, though, as it can be easy to get buried in the weeds and miss things.

The first suggestion she makes is to add another queue to the mix - you could have half the messages for the analytics service go to one queue and the other half go to the other queue. This would double the amount of work that can be done by having multiple workers in place.

Table 9. Doubling A Queue.

Exchange	Ex. Type	Routing Key	Queues
download	topic	download.requested	download.logging
		download.requested	download.analytics
		download.requested.2	download.analytics.2

This would require a new routing key, though. Having a second routing key would require a second call to RabbitMQ, to use this key. It would also mean that you have to keep track of which routing key should be used next, from your web server. All of this seems like a lot of trouble to get another worker in place for this message.

Not happy with this option, but also not having any ideas of your own at this point, you head back to the books and search engines to see if there are any better options. A few minutes later, you come back to the whiteboard a mixed look of embarrassment and surprise.

More Handlers

Every source that you looked at in the last few minutes has confirmed that yes, RabbitMQ will handle this scenario for you. In fact, this isn't something that you need to configure or build or add support for at all.

It turns out that this "round-robin" style of alternating a message is built in to queues directly. All you have to do is stand up another instance of the analytics queue handler and RabbitMQ will immediately begin alternating which queue receives messages, one after the other. You can add as many queue handlers as you need, and the messages will be evenly distributed across all of them.

So, scratch the extra queue, extra routing key and extra call to RabbitMQ. Let RabbitMQ do the work for you.

Table 10. Multiple Handlers.

Exchange	Ex. Type	Routing Key	Queues	Handlers
download	topic	download.requested	download.logging	1
		download.requested	download.analytics	2

Less than 10 minutes later, you have a second instance of the queue handler for the analytics server up and running. The effect is immediate - the analytics queue is now draining instead of filling up! Sure, it's a very slow process. Estimating from the last few minutes of watching the queue, it should take about a week to completely drain the queue. But, hey! It's working once again!

Once the new set of queue handlers catches up on the backlog of messages, it will stay caught up, too. And if it doesn't? Well, you can always add another handler instance to take care of any backlog you have.

Epilogue

A few weeks after standing up the second handler for the analytics service, your queue looks to be healthy and happy.

There are peak usage times when the queue bumps up to a few dozen messages, and it takes a few hours of continuously processing new messages before the peak period drops. But with a few dozen messages backed up, it only takes a few minutes for it to drain once the high traffic subsides.

So far, everything has been running smoothly. But, it's good to monitor the queue and make sure you are not seeing a permanent backlog.

Retry Queues

During a round of code cleanup recently, your teammate asked about the way the analytics service retries were being handled. She understand how it was working - the code to wait for a minute was fairly intuitive. But she didn't like the idea of having the code wait like that. If the back-end service dies, so does the message. You would lose however many messages were in a wait state, at that point.

After doing a little research on the subject, she came up with what looks like a great option for this: a "retry" queue. Technically, it's not a type of queue. It's a combination of RabbitMQ's "dead letter exchange" and TTL (time to live) settings. She tells you that you can have a queue with a TTL setting of 1 minute, combined with a dead letter exchange to create the retry queue setup.

The idea sounds intriguing, and it would certainly solve the problem. Adding it to your list of things to do this week, you sit down for some research on the idea and find a number of blog posts surrounding it. You also find a plugin for RabbitMQ, to handle this exact scenario in a much better manner.

It looks like you'll be brushing up on your RabbitMQ configuration and management skills over the next few days. Time to settle in and get started.

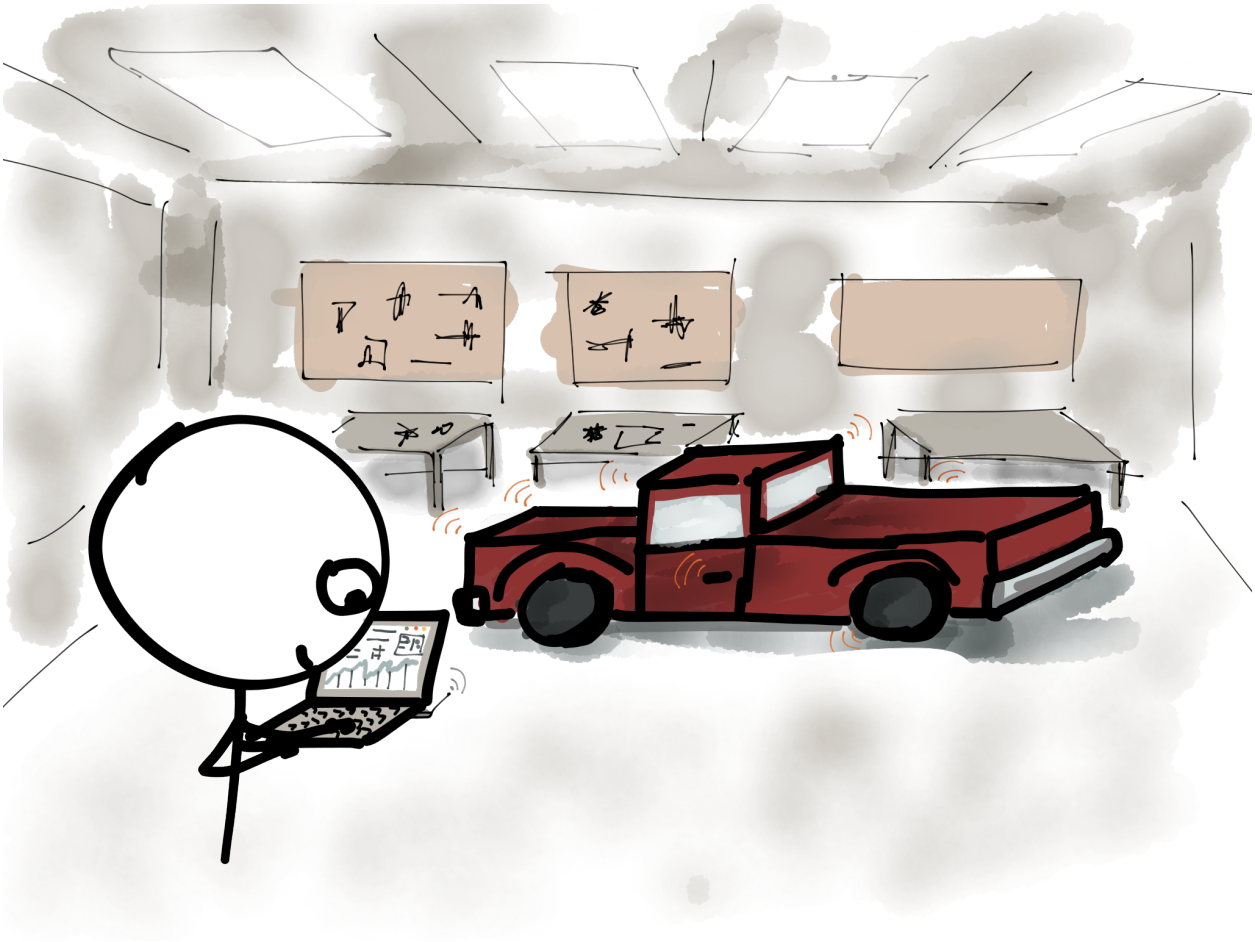
Fixing The Network Glitch

Just as you get in to the zone of configuring your local RabbitMQ instance, your phone interrupts you. It's your hosting service provider, and they say they have identified a few issues in their network

routing equipment. It looks like some bad entries in the routing tables may be causing the hiccups with your analytics service.

With a smile, you let the service provider know how thankful you are for the continued effort to track down the problem. You also tell them that they can wait until the next maintenance window - 3 weeks out - to fix the problem. You've got a working solution to the problem for now.

Chapter 8: Remote Diagnostics and Maintenance



If you could go back in time 10 years and tell your younger self that *this* would be your dream job, the younger you would have laughed. Hard.

It all started 6 months ago. You were somewhat desperate for a job at the time, having just left another position that was not suiting you at all. You had some savings built up, fortunately, so you weren't ready to take the first offer that came along. You also had realistic expectations of the job market and were willing to interview at places that didn't fit your usual high-tech profile. When a friend suggested the vehicle maintenance shop at this company, you thought he was joking at first. But after reading up on the company and what the shop was looking for, you found it intriguing. Maybe you should interview there just to see if the job description was a lie… or if there really was

something to it.

It might not be a high-tech company, but the work you're doing certainly is! And it still doesn't seem right when you stop to think about it, some times - the idea of vehicle maintenance being built on the back of the "Internet of Things" (IoT) movement, with wi-fi enabled sensors all over the vehicles.

But, here you are. You're building maintenance software for a vehicle fleet. You're using JavaScript to power sensors on vehicles. You're compiling a profile of a vehicle on the fly, so a maintenance bay can be ready for it.

And you're having the time of your life!

A Not-So-Working Prototype

Over the last 3 months, you've spent a countless number of hours working with these little sensors that are going in the vehicles.

The hardware design team has been doing some amazing things - especially when it comes to the software that runs the devices. They've built it all with prototyping electronics boards that allow JavaScript to be used, instead of C. This has given you a chance to work with the hardware directly a few times, and helps you to understand how these devices operate. The end result has been worth the time and effort, as your knowledge of the working hardware and how they will communicate has directly influenced your design and implementation of the software to capture all of the data.

Basic Working Design

The initial system design has all the parts working together over the same wi-fi network. Each of the sensors will connect to a pre-configured network when they are powered up. The network configuration is stored in the device's EEPROM - a 1K chunk of non-volatile memory. It's just enough room to store the required network configuration and a few other bits to tell the device which vehicle is it working within.

You've had a lot of success with these devices, so far. Each of them broadcasts their location over a discoverable network protocol, identifying which vehicle they are part of and what the sensor does. Your application code reads the network discovery protocol, logs all of the available sensors and organizes them in to a "vehicle" grouping.

When the application user want to get information about the vehicle, they select it from the list of vehicles that are within the same wi-fi network. Once the vehicle is selected, the sensors are read via HTTP based API calls. Each of the sensors returns a JSON document to your software which is used to create a complete picture of the current of the vehicle's status. Everything from miles driven to amount of fuel used, from average time spent in traffic to how many miles the tires have left on them is captured. The full "health" of each vehicle is known, allowing the maintenance shop to very quickly and efficiently work on each vehicle and keep the entire fleet operating.

Your application is working well so far. You've demo'd it to "The Boss" (as everyone calls him) - a surprisingly tech-savvy man in his 50's, who has more grease and grit from vehicle maintenance than you have years of life - and he has been extremely happy so far.

It comes as a bit of a surpise, then, when the hardware team starts making noise about "serious complications", "missing deadlines" and "extensive rework".

API Design Failure

With some concern about the project's likelihood of success, The Boss called for an all-hands meeting. Fortunately, "all hands" means a total of 5 people other than yourself. It's a small team, but that is part of the secret to success, here. The Boss believes in small teams of highly engaged and multi-skilled people, rather than a larger organization of people that are highly specialized and less motivated.

The hardware design lead starts the conversation with her usual complimentary style. She's always an incredibly positive person, looking for opportunity to let everyone around her know that they are doing a great job. This meeting is no exception, either. She has no shortage of praise for the hardware team, yourself and the QA team - who, incidently, are also mechanics that work on the vehicles. But the usually positive and smiling face quickly turns somber as she begins to describe the problems that they've run in to.

Everyone has known, from the beginning of the project, that the final hardware design would involve re-writing hardware to use C. It would allow the team to produce much smaller, less power-hungry devices. The goal, though, was to produce the first batch of sensors with the more flexible prototyping boards and get in to serious hardware design later.

Unfortunately, she says that "later" has become "now", as the hardware team has been unable to crack through a set of problems with their current JavaScript based hardware.

The problem is not the language - she is very clear about that. The problem is the current set of JavaScript bindings for the hardware they are using. It is preventing them from being able to re-configure the network settings the way they need to, because of a bug in the JavaScript version of the EEPROM API.

The hardware team has spent a few weeks working with the manufacturer of the hardware, discussing the problem, digging in to the JavaScript bindings and helping to look at the lower-level C code that facilitates the bindings. The sad reality, though, is that the hardware team has been told that it will take 6 months to fix that particular problem… minimum. It's not that the problem is difficult to track down. It has more to do with the current priorities of the API team for the hardware manufacturer.

The time frame that she was given is not a timeframe that will be acceptable for your company. The first batch of prototype sensors are due to be installed in production vehicles in 2 months. The current hardware will be fine, as the team can hard code the network settings for now. However, once the company puts the next round of sensors in to the other shops around the state, things become unmanageable quickly.

The Boss nods, with a quiet and thoughtful expression. He understands how difficult hardware manufacturers can be. He also understands how difficult it is to work around hardware limitations with software. Just prior to ending the meeting, he asks the hardware team to come up with an estimate for the rework of the prototype hardware and software. He also asks the team, as a whole, to think about design alternatives - where the EEPROM of the device does not need to be modified before installing the sensors.

Quietly, and with more than a little concern about the project team's ability to complete the work on time, the team disperses back to their respective jobs.

Push Notifications

Tuesday rolls around again, and you're happy to see this day. It's been an odd week at the shop. The usually loud and energetic atmosphere is gone. The hardware team has been scrambling to prototype, staying holed up in their lab. The QA team has been back on the maintenance floor trying to play catch up with the other half of their work. And you've been sitting in your office, working away at the application side of things, hoping that someone comes up with a solution to the problem.

But Tuesdays are an office tradition no matter the problems being faced, no matter the deadlines the team is one. Every Tuesday evening, the project team - hardware, software, and QA - all head across the street for food and drinks. It's a time to relax, to talk about non-work things and to hopefully "inspire" some creativity by getting out of the office. It's also a common thing for The Boss to show up and pay the whole tab - and while it doesn't happen all that often, you're hopeful this week as The Boss has a tendency to drop in when things are particularly tough at the shop.

Smiling Faces And Day-Dreams

It's good to see the smiling faces again, as the crew relaxes a little and unwinds from the stress of the project. The hardware lead, especially, has been taking this hard. She's been working late hours so that her team doesn't have to. Of course, that only inspires her team to work late, too - in spite of her telling everyone else to go home at 5pm every day. But here, on Tuesday, everyone has a few hours to forget about the extra work and talk about something else for a while. At least, until the inevitable shop-talk starts up in about an hour.

When a moment of laughter turns in to an awkward silence, it's the hardware lead that brings up work, first. She's been playing with some interesting tech in the last few days and has some real options. But one of the problems she keeps running in to, is figuring out how to make an HTTP based API for your software to consume. Most of the hardware that supports what they need, doesn't have that kind of capability.

One of the more interesting bits of tech that she is playing with, though, is some kind of mesh-network setup. The idea sounds cool - each device is connected to other devices within a short distance to create a network between all of them. You can reach devices out of your range by connecting to devices within range and having them forward your connection to the other one.

This solves a lot of problems they are having with Wi-Fi configuration - because there is no Wi-Fi configuration. But that only makes the problem of you getting the data off the mesh network more difficult.

As the night moves on, you can't help but think about the mesh network and how interesting that idea is. Sitting back in to the booth, getting comfortable for a moment, you imagine your cell phone being on a mesh network. What would it look like if it were communicating all over the globe by connecting to all the other phones in the restaurant or within two or three blocks. They go out to the next hop of in-range devices, which continue to go out from there. It would be like that William Hertling novel where artificial intelligence has replaced TCP/IP with mesh based networks. Wouldn't that be awesome?

Your enthusiastic, child-like smile is suddenly wiped from your face as your phone dings and vibrates with a new email. Annoyed at the interruption to your daydream, you open the phone settings to turn off push notifications.

That's when it hits you - the solution to the API problems with the mesh of sensors, the ability to get data from them, and how to make this whole thing work! At least, you're hoping this is a viable solution.

With a sudden energy and vision that needs to be shared, you practically jump out of your seat to find the hardware lead and let her know what you're thinking.

Messages, Mesh Networks and Push Notifications

Barely able to contain your excitement, you sit down next to the hardware team lead and tell her that you have an idea. In spite of her initial sigh, she says she's interested in hearing it.

Launching in to your vision of a future filled with mesh-network phones, you've moved off topic quickly and don't seem to be coming back any time soon. Confused, the hardware lead says she isn't quite sure where you're heading. With a short apology, you promise that the story does have a point.

When you finally get to the point, her eyes go wide. She thinks it's brilliant! Gathering the rest of the team around the remains of the shared appetizers and left-over dinners, the hardware lead asks you to re-tell the story, promising everyone else that it will be worth listening it.

You start off by recounting your day-dream about your phone, and how Hertling's novel has Artificial Intelligence replacing TCP/IP networks with mesh networks. You make the obvious connection to the mesh network hardware that the team has been playing with, and everyone nods with understanding. The problem that has already been pointed out, is how to get the information from the mesh network and out in to your software. You can't require every diagnostics computer to have the mesh hardware retrofit in to it. Most of these computers barely have a USB port, let alone expansion slots for new hardware.

The moment of inspiration came when got that email notification on your phone and went in to turn off push notifications. At that moment, you thought your current application design and how

you're using RabbitMQ to publish usage stats and other information back to the central server for the maintenance bay. The connection between push notifications and RabbitMQ was you thinking about how the push notifications handled your phone severing the connection. It probably works like RabbitMQ, you thought, where the push is received by anything connected and the message is thrown away if nothing is there to receive it. It was that connection - RabbitMQ and push notifications - combined with your mesh network day-dream that brought the solution to mind.

Rather than having your software reaching out to find and poll the sensors on a Wi-Fi network, invert the relationship. You could have the mesh network talking to a central "brain" or "hub" within the vehicle - a hub that runs a full-featured operating system like Linux. There are plenty of options out there for miniaturized hardware prototyping that run full operating systems like this. You could easily add one of the mesh network adapters to a development board like this. As it reads data from the mesh of sensors, it can publish that data through a local RabbitMQ server. Much like the push notifications of your phone, you can have any connected system read the messages as they come through the queues - and, boom! Problems solved!

Around the table, there are looks of interest and looks of confusion. Sensing that not everyone at the table is as excited as you and the hardware lead, she asks for questions from the obviously confused persons.

The first question is how this gets around the Wi-Fi issue. Aren't you just moving that problem to the "hub"? It's true, you are. But that problem is easier to solve with these miniature computers. You can configure them to have an ad-hoc Wi-Fi network using security based on public/private encryption keys. On the laptop side of things, you can use dirt-cheap USB Wi-Fi dongles that can be configured through your software to use the ad-hoc network and provide the correct encryption key.

The second question, after seeing additional nods of approval, is how long will this take to implement?

Mildly startled, the entire table turns and looks up at The Boss. Apparently, he had shown up just in time to hear you recounting your story of inspiration and problem solving. He is grinning nearly ear to ear as he drops a credit card down on the bill for the food and drinks, and repeats his question to you and the hardware team.

A Now-Working Prototype

It took a few weeks of trial, error, research and trying again. But the hardware team, augmented with your experience in RabbitMQ, finally found a miniature computer that could support a message queue server, an ad-hoc wireless network configured with encryption keys for access and the needed GPIO pins for the mesh network hardware module.

But it works! And oh, does it work well!

Configuring The Exchanges

Your first attempt at an exchange and queue layout was through a topic exchange. It seemed like a good idea at first, given the flexibility that you may want in the future.

Table 1. Topic Exchange Layout

Exchange	Ex. Type	Routing Key	Queue
sensor	topic	sensor.data	sensor

This worked well for your initial testing. You had demo hardware for the new mesh network and "hub" computer set up on your desk, and you were able to read the sensors correctly.

The best part was the ease of changing your software. You were already publishing the sensor data to a back-end processor with RabbitMQ. All you did for this change, then, was add a queue handler for this new sensor queue server in your software. This handler took messages out of the "sensor" queue and pushed it through your central RabbitMQ server to the existing services. Done.

It wasn't until you tried to install the new system on an actual vehicle, that you saw the problem. After getting the new hardware set up and having it push some sensor data through, you were happy - the rest of the team was, as well. The next morning when you came in to the office, though, that sense of happy quickly deteriorated.

When you came in that morning, your software was taking far too long to connect to the vehicle and retrieve data. After spending 30 minutes dismantling the hardware and plugging the hub in to your workstation, you found out why.

The new "sensor" queue that you put in to the hub's RabbitMQ configuration had more than 250,000 messages in it.

Shocked at the number of messages, you asked for the hardware team to look in to this with you. After a few hours of talking, reviewing code and looking at hardware configuration, you determined that this was the correct number of message for the last 12 hours. The RabbitMQ server, as it turns out, was destroying this little micro-computer. The large number of messages was eating up all available memory. The continued onslaught of messages was thrashing the flash based storage of the device as Linux was forced to page memory to the drive to keep itself running.

Correcting The Queue

The hardware team was concerned that this wasn't going to work, after seeing the initial catastrophe on this prototype. Would you be able to fix this? How long would it take?

Fortunately you realized the mistake fairly quickly after diagnosing the problem. The problem was not RabbitMQ, it was your configuration of the exchange and queue.

The first change you need to make was to put a limit of messages on the queue. Instead of storing everything and waiting for the queue to drain, a cap of a few hundred messages should be sufficient.

There aren't that many sensors in the device, and the mesh network is smart enough to round-robin reading the sensors. This will guarantee an appropriate sampling size of every sensor, for every 100 messages in the system. Putting a limit of a few times that size means that you will have far more data than you need for any given sampling that you read from the queue.

Table 2. Adding A Queue Limit

Exchange	Ex. Type	Routing Key	Queue	Limit
sensor	topic	sensor.data	sensor	300

To test this limit, you let the sensors run all night on your workstation. They produce a known number of messages based on real examples from the vehicles, so this should be a sufficient test for your change.

The next morning, you're happy to see that the number of messages in the queue is sitting at exactly 300. You turn on your application and a few minutes later, all 300 messages have been processed. The rate of incoming messages is now slower than the application is consuming them, as well. All seems well, so far!

Where'd They Go?!

A few days later, you're feeling good about the system. Things have been running smooth in your tests and in the demo vehicle.

Just as you sit down after lunch, one of the QA mechanics stops by your office. He's got a puzzled look on his face and asks if you have a moment. On the way to the workstation in the shop, he explains the situation to you.

After having a successful few days of running the sensors on the demo vehicle, they decided to stand up a second set of sensors. They wanted to see how the system would work with multiple mesh networks in proximity of each other.

Happily, the mesh networks were intelligently configured by the hardware team. Each network has it's own ID that is associated with a specific vehicle. Sensors won't communicate sensors from other vehicles, this way. The problem they are having, it seems, is in the hub computers on each vehicle.

When they first launched the laptop software for the system on a second computer, it was still configured with the network setup from the first vehicle. They expected this, as the second laptop was a clone of the first. What they didn't expect, however, was for half of the messages from the first vehicle to seemingly disappear.

It took a while to notice the problem. They were running the software on the original vehicle and things were looking slightly off but within range. An hour later when they started running the software on the new laptop, they were seeing a duplicate of the original vehicle's readings. This was odd, as the second set of sensors were coded with different responses to simulate a second vehicle.

After looking at the log files for the first system, the noticed that it only pulled in 150 message. This had them thoroughly confused, which is when they came to get you. By the time you got back to the shop floor, though, the other QA mechanic had an answer for where the messages went.

She was looking at the software on the second laptop and noticed that it had pulled in 450 message. That was the 300 she expected to see, plus 150 more.

Fortunately, the problem is immediately obvious to you, and you sigh at the realization.

When the second laptop connected to the first sensor system, it used the same queue on the RabbitMQ server of the hub. When multiple message consumers connect like this, RabbitMQ round-robins the messages to the consumers. With two consumers, each of them got half of the messages.

Before heading back to your cube, you've got an idea. But you need to verify something with the QA mechanics, first.

You're wondering if it would be ok for the laptop system to have a small time delay when obtaining the diagnostics from the vehicle. In other words, if the laptop has to "run diagnostics" for about a minute or two, would that be a disaster? Considering the existing delay of the background processing, the QA engineers don't see that as an issue. As long as each laptop gets a copy of all the messages, they don't really care. They just need to have both laptops produce accurate reports. Happy with that, you head back to your office to fix the problem.

Waiting For The Messages

The solution seems simple enough. Rather than using a durable queue with a message limit, you'll switch the system to use exclusive queues that auto-delete when the message consumer disconnects.

Each laptop having it's own exclusive queue on the hub's RabbitMQ. This will ensure that only that one laptop can consume messages from that queue. Each queue will be bound to the sensor exchange as well. This allows each queue to receive a copy of every message. There won't any opportunity for messages to get "lost" or mixed up between laptops with this configuration.

Since you're talking about different laptops connecting at different times, using exclusive queues that prevent other consumers, you decide to modify the exchange type to a more suitable option.

Table 3. Modified Exchange Type

Exchange	Ex. Type	Routing Key	Queue	Limit
sensor	fanout	(n/a)	(exclusive)	300

While you're looking at the new layout, you realize the queue limit size it useless now. With each consumer creating a queue when it connects and deleting the queue when it disconnects, there won't be any messages to keep around for a backlog.

Table 3. Removing The Queue Limit

Exchange	Ex. Type	Routing Key	Queue
sensor	fanout	(n/a)	(exclusive)

It looks like a good layout. Now you only need to update your software to have a "running diagnostics" message show on the screen. This screen will display until a certain threshold of data has been reached for each sensor, ensuring you have the most up to date and accurate readings for each vehicle, every time you run the diagnostics.

Epilogue

A month has passed since you switched everything to a fanout queue on the hub's RabbitMQ configuration. The software has been running rock-solid in the QA lab, and you've just deployed the new system to the first production vehicle.

The hardware and sensors are still based on the prototyping boards, but that's ok. The team was able to pull together the needed parts, configuration and design in near record time.

With your help, the flexible architecture of your application and your moment of brilliance with RabbitMQ, everything looks to be running smoothly. This has everyone smiling, including The Boss. He has been smiling ear to ear for the last week, and has been not-so-subtly hinting at something about a bonus.

Appendix A: Additional Resources

This book is not a technical resource for people that are wanting to learn how to implement RabbitMQ in their systems. I wrote this in a manner that is more conducive to learning the "why" of RabbitMQ's various options, rather than the "how" - and I did that very intentionally.

However, there are a lot of great resources available for the "how" of RabbitMQ. And I would like to share some of my personal favorites with you.

Over the next few pages, you'll find information about other resources that I've used in the last few years. From RabbitMQ basics, to advanced topics and techniques for system integration, these resources will help you continue your journey.

WatchMeCode: JavaScript Screencasts

Link bit.ly/db-rmq-wmc[4]

WatchMeCode is my own series of JavaScript screencasts, aimed at teaching you everything you need to know to about working with JavaScript on a daily basis.

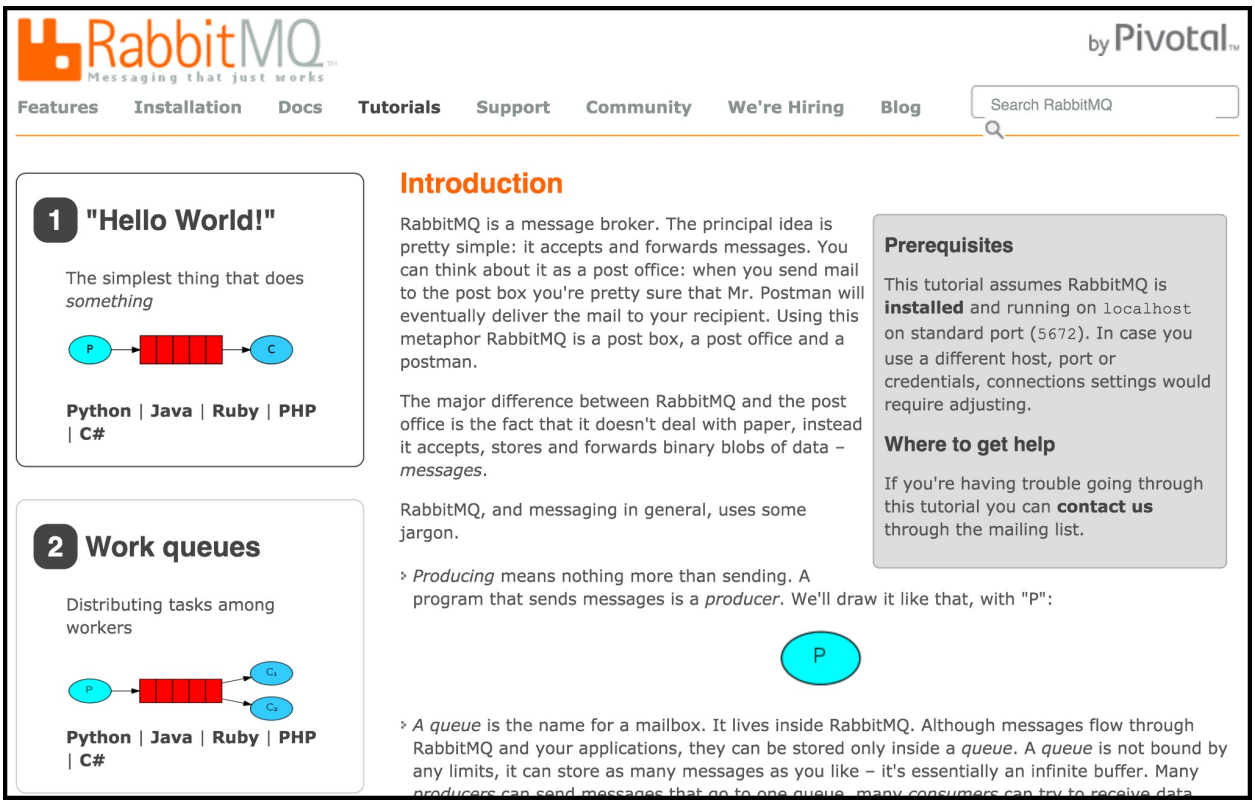
Included in this extensive library of videos, is a series on using RabbitMQ with NodeJS. If you have not yet viewed any of these videos and are looking for the fastest possible way to get started and become productive with RabbitMQ in a NodeJS environment, then these videos are for you.

From installation, to the basics of management and configuration, to the in-depth and real-world scenarios that you need for using RabbitMQ with NodeJS, this series of screencasts has it all covered.

[4] http://bit.ly/db-rmq-wmc

The RabbitMQ Tutorials

Link: bit.ly/db-rmq-tutorials[5]

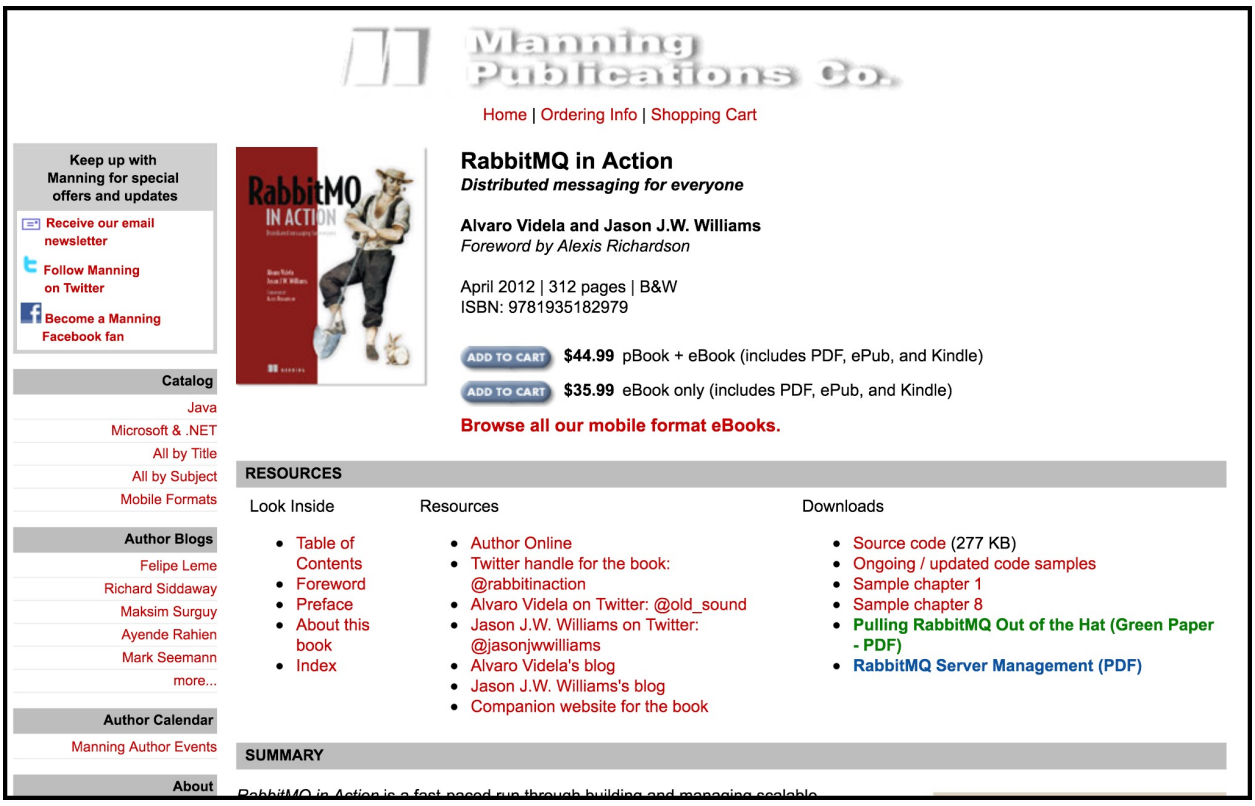


One of the best places to start for the how-to of basic uses cases in RabbitMQ is the tutorials that they provide on the RabbitMQ website. With examples in Python, Ruby, Java, C# and PHP, they have most languages covered. And if your language isn't covered, there's a good chance that the examples they do provide will at least give you the right direction to head with your language and driver of choice.

[5] http://bit.ly/db-rmq-tutorials

RabbitMQ In Action

Link: bit.ly/db-rmq-inaction[6] Discount Code: **rabbitpod15** Discount: **40% off**



This is the book that taught me most of what I know about RabbitMQ, and it continues to be a resource for more information on how to work with and effectively use this system.

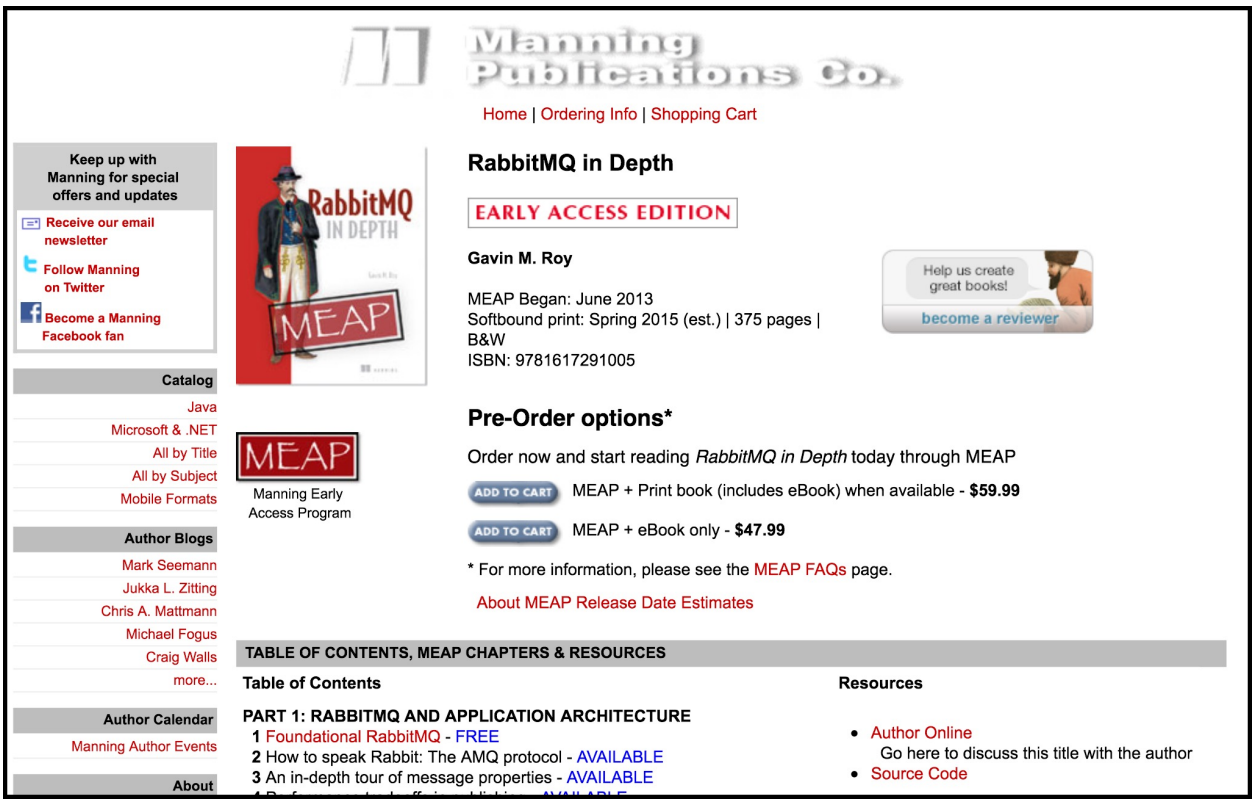
If youâ€™re doing any work with RabbitMQ and you need to understand the core of this messaging system, then I highly recommend picking up this book as soon as possible.

And at 40% off, RabbitMQ In Action is a book that you can't afford not to have in your library! Be sure to use the link and discount code from above to take advantage of this 40% off deal.

[6] http://bit.ly/db-rmq-inaction

RabbitMQ In Depth

Link: bit.ly/db-rmq-indepth[7] Discount Code: **rabbitpod15** Discount: **40% off**



Beyond the core use of RabbitMQ, there is a world of knowledge for optimizing, distributing across a cluster, understanding the AMQP protocol, and so much more.

RabbitMQ In Depth has proven to be an invaluable resource for me, answering questions about the AMQP specification and showing me just how deep the rabbit hole goes. With this book in hand, I have increased my own understanding of the tools and technology around RabbitMQ immensely.

If you're moving beyond the simple, every day use of RabbitMQ and in to a production environment where you need high availability with failover clustering, need to understand the AMQP specification, or want to look at how you can take advantage of this "programmable" protocol, then this is the book for you. It is an essential part of any RabbitMQ developer's library, for moving past the initial development stages and in to the production needs of real-world applications.

Once again, you will receive 40% off this book using the above link and discount code!

[7] http://bit.ly/db-rmq-indepth

Enterprise Integration Patterns

The Book: bit.ly/db-rmq-eip[8]

The Website: EnterpriseIntegrationPatterns.com[9]

This is *the* book on messaging patterns in software development. With extensive coverage of nearly every aspect of both the how and why of messaging, you will not find a better resource for understanding the intent of using messages in software development.

I have personally used this book as a constant source of knowledge in the last 7 years. From IBM WebsphereMQ to Microsoft's MSMQ, and even today with my RabbitMQ deployments, this book has provided a depth of knowledge on how to structure and work with messages that is second to none.

While this book does not cover RabbitMQ specifically, it is an absolute must have on your library bookshelf. You will learn every imaginable pattern, technique and tool for working with a message oriented architecture.

[8] http://bit.ly/db-rmq-eip

[9] http://www.enterpriseintegrationpatterns.com/

About Derick

Hello, my name is Derick Bailey.

I'm an entrepreneur and software developer, a consultant, screencaster, blogger, speaker, trainer and more. I've been working professionally in software development since the late 90's and have been writing code since the late 80's.

My career has spanned many different tools, technologies, platforms and languages. But my most recent love-hate relationship has been with JavaScript. From the browser to the server, to the database engine and even controlling hardware - I've dipped my toes in just about every aspect of JavaScript there.

I've spent several years working with messaging systems of various sizes and complexities, including WebsphereMQ, MSMQ and most recently RabbitMQ. Through the experience of building large systems from small pieces that must communicate with each other, I've found a few patterns of message groupings to be useful - and I want to share those patterns with you, in this book.

You can find me writing about these and many other subjects at my blog. I also produce screencasts, provide information about my consulting services, and more, through the following websites:

- My Blog: DerickBailey.com[10]
- Screencasts (free and paid): WatchMeCode.net[11]
- Open Source Projects: GitHub.com/DerickBailey[12]
- Entrepreneurial Podcast: The Entreprogrammers[13]

[10] http://derickbailey.com
[11] http://watchmecode.net
[12] http://github.com/derickbailey
[13] http://entreprogrammers.com

If you have any questions, comments or concerns, you can contact me using the following:

- Email: derick@mutedsolutions.com[14]
- Twitter: @derickbailey[15]

[14] mailto:derick@mutedsolutions.com
[15] http://twitter.com/derickbailey