Table of contents

The RabbitMQ plugin provides integration with the RabbitMQ Messaging System.

RabbitMQ Plugin - Reference Documentation

Authors: Jeff Brown, Peter Ledbrook

Version: 1.0.0

Table of Contents

1 Introduction To The RabbitMQ Plugin

The RabbitMQ plugin provides integration with the RabbitMQ highly reliable enterprise messaging system. The plugin relies on Spring AMQP as an implementation detail, which provides a high level abstraction for sending and receiving messages.

This guide documents configuration details and usage details for the plugin. More information on RabbitMQ itself is available at rabbitmq.com.

1.1 Change log

Version 1.0.0 - 28 Nov 2012

- First GA Release

Version 0.3.2 - 16 Mar 2011

- Upgrade to Spring AMQP 1.0.0.M3

Version 0.3.1 - 14 Feb 2011

- spring-core dependency now excluded
- Corrected the SCM and documentation URLs in the plugin descriptor

Version 0.3 - 7 Feb 2011

- Upgraded to Spring AMQP 1.0.0 M2
- Added transaction support
- Added support for pub/sub model
- Applications can now configure exchanges and bindings in addition to queues
- Applications can now connect to any virtual host, not just '/'
- Bug fixes:
  - GRAILSPLUGINS-2496 - Messages are now picked up when the application starts

2 Configuration

The plugin supports a number of configuration options which all may be expressed in `grails-app/conf/Config.groovy`. A basic configuration might look something like this:

```
// grails-app/conf/Config.groovy
rabbitmq {
    connectionfactory {
        username = 'guest'
        password = 'guest'
        hostname = 'localhost'
    }
}
```

Those are settings which are necessary in order for the plugin to be able to connect to and communicate with a RabbitMQ server.

Following is a list of other configuration settings supported by the plugin.

Configuration Property	Description	Default
rabbitmq.connectionfactory.username	The user name for connection to the server	(none)
rabbitmq.connectionfactory.password	The password for connection to the server	(none)
rabbitmq.connectionfactory.hostname	The host name of the server	(none)
rabbitmq.connectionfactory.virtualHost	The name of the virtual host to connect to	'/'
rabbitmq.connectionfactory.channelCacheSize	The connection channel cache size	10
rabbitmq.concurrentConsumers	The number of concurrent consumers to create per message handler. Raising the number is recommended in order to scale the consumption of messages coming in from a queue. Note that ordering guarantees are lost when multiple consumers are registered.	1
rabbitmq.disableListening	Disables all service listeners so that they won't receive any messages.	false
rabbitmq.retryPolicy.maxAttempts	Sets the maximum number of retries for failed message deliveries	0

2.1 Configuring Queues

Queues must be declared in the RabbitMQ server before consumers can be associated with those queues and before messages may be sent to those queues. If the Grails application will be sending messages to or receiving messages from queues that may not already be declared in the RabbitMQ server, the application needs to declare those queues up front. One way to do that is to add beans to the Spring application context of type `org.springframework.amqp.core.Queue`. That might look something like this:

```
// grails-app/conf/spring/resources.groovy
beans = {
    myQueue(org.springframework.amqp.core.Queue, 'myQueueName')
    myOtherQueue(org.springframework.amqp.core.Queue, 'myOtherQueueName') {
        autoDelete = false
        durable = true
        exclusive = false
        arguments = [arg1: 'val1', arg2: 'val2']
    }
}
```

The plugin also supports a DSL for describing these queues. This DSL is expressed in `grails-app/conf/Config.groovy`. The code below configures the same queues as the previous code sample.

```
// grails-app/conf/Config.groovy
rabbitmq {
    connectionfactory {
        username = 'guest'
        password = 'guest'
        hostname = 'localhost'
    }
    queues = {
        myQueueName()
        myOtherQueueName autoDelete: false, durable: true, exclusive: false,
arguments: [arg1: 'val1', arg2: 'val2']
    }
}
```

With both techniques, the `autoDelete`, `durable` and `exclusive` attributes default to `false` and the `arguments` attribute defaults to null.

So what do those queue options mean?

Option	Description
autoDelete	If `true`, the queue will be removed from the broker when there are no more clients attached to it. Note that this doesn't take effect until after at least one client connects to the queue.
durable	If `true`, the queue will survive a restart of the broker.
exclusive	Only the client that created the queue can connect to it.

One final thing: when you declare a standalone queue like this, it automatically gets bound to the broker's default exchange, which has an implicit name of '', i.e. the empty string. You can easily send messages to this queue via the rabbitSend method.

2.2 Configuring Exchanges

Queues are the foundation of consuming messages, but what if you want to send messages? In AMQP, you send messages to an exchange and the exchange then routes those messages to the appropriate queues based on something called a binding. The key to setting up complex messaging systems lies in configuring these exchanges and queues appropriately.

Declaring an exchange

Let's start with an example of how to set up a simple exchange (with no queues):

```
rabbitmq {
    connectionFactory {
        …
    }
    queues = {
        exchange name: 'my.topic', type: topic
    }
}
```

As you can probably guess, this will create a topic exchange with the name 'my.topic'. There are two things to note at this point:

1. the name and type are required
2. the type value is *not* a string literal

So what types are available to you?

Type	Description
direct	An exchange that only routes messages that are bound to it with a key that matches the routing key of the message exactly. Typically this exchange is used for point-to-point messaging and the routing key is the queue name.
fanout	Sends messages to all queues bound to it. It basically does a broadcast.
topic	Similar to the `fanout` exchange, this routes messages to the queues bound to it, but only queues whose binding matches a message's routing key will receive that message. Wildcards are supported in the binding.
headers	Similar to topic, but messages can be filtered by other any message header, not just the routing key.

The exchange declaration also supports a couple of extra options that should be familiar from the queue declarations:

Option	Description
autoDelete	If `true`, the exchange will be removed from the broker when there are no more queues bound to it. Note that this doesn't take effect until at least one queue is bound to the exchange.
durable	If `true`, the exchange will survive a restart of the broker.

With the above syntax, it is up to you to bind queues to the exchange via another AMQP client or via the RabbitMQ management interface. In other words, this is most suitable if your Grails application is purely a publisher of messages and not a consumer (or at least not a consumer of 'my.topic' messages).

What if you want to create queues and automatically bind them to the exchange? Don't worry, that's supported by the configuration DSL too.

Binding queues to exchanges

An exchange on its own isn't particularly useful, but we can easily bind queues to it by declaring them as nested entries:

```
rabbitmq {
    connectionFactory {
        …
    }
    queues = {
        exchange name: 'my.topic', type: topic, durable: false, {
            foo durable: true, binding: 'shares.#'
            bar durable: false, autoDelete: true, binding: 'shares.nyse.?'
        }
    }
}
```

In the example above, we bind two queues ('foo' and 'bar') to the exchange 'my.topic'. Since this is a topic exchange, we can use a binding key to filter which messages go from 'my.topic' to each queue. So in this case, only messages with a routing key beginning with 'shares.' will end up on the 'foo' queue. 'bar' will only receive messages whose routing key begins with 'shares.nyse.'.

This approach isn't limited to topic exchanges: you can automatically bind queues to any exchange type. There are a few things to bear in mind though:

1. the default binding for direct exchanges is the queue name (unless this is explicitly overridden by a 'binding' option);
2. the 'binding' is ignored for fanout exchanges; and
3. the headers exchange requires a map of message header names and values for its binding.

⚠ RabbitMQ has several built-in exchanges with names of the form 'amq.*', for example 'amq.direct'. If you want to bind to these, you currently have to declare them with the correct attributes, i.e.

```
exchange name: "amq.direct", type: direct, durable: true,
autoDelete: false
```

As you can imagine, these few building blocks allow you to configure some pretty complex messaging systems with very little effort. You can tailor the messaging system to your needs rather than tailor your applications to the messaging system.

2.3 Advanced Configuration

When you need fine-grained control over your service listeners, you can tap into the power of Spring. Since each service listener is implemented as a set of Spring beans, you can use Grails' bean property override mechanism to provide your own low-level settings.

So how are these beans set up? If a service has either a `rabbitQueue` or `rabbitSubscribe` property, then you will have these beans:

- `<serviceName>_MessageListenerContainer` of type SimpleMessageListenerContainer
- `<serviceName>RabbitAdapter` of type MessageListenerAdapter

As an example, let's say you have a `MessageStoreService` like so:

```
class MessageStoreService {
    static rabbitSubscribe = [exchange: "amq.topic", routingKey: "logs.#"]
    …
}
```

You can then customise things like the number of concurrent consumers, whether the channel is transacted, what the prefetch count should be, and more! Simply add code like this to your runtime configuration (Config.groovy):

```
beans {
    messageStoreService_MessageListenerContainer {
        channelTransacted = false
        concurrentConsumers = 10
        prefetchCount = 5
        queueNames = ["q1", "q2"] as String[]
    }

messageStoreServiceRabbitAdapter {
        encoding = "UTF-8"
        responseRoutingKey = "replyQueue"
    }
}
```

This approach works for any property that accepts a basic type. But what about bean references? In this case, you can't use the bean property overrides. Fortunately, the most common bean reference you are likely to want to override, the message converter, has a dedicated configuration option:

```
rabbitmq.messageConverterBean = "myCustomMessageConverter"
```

This is a global setting that accepts the name of a message converter bean. For the rare occasions that you need to override other bean references, you can declare your own `<serviceName>_MessageListenerContainer` or `<serviceName>_RabbitAdapter` beans in resources.groovy.

Finally, you can override some of the global config options on a per-service basis:

```
rabbitmq {
    services {
        messageStoreService {
            concurrentConsumers = 50
            disableListening = true
        }
    }
}
```

There are many options for customisation and we hope the above will get you started.

3 Sending Messages

The plugin adds a method named `rabbitSend` to all Grails artefacts (Controllers, Services, TagLibs, etc...). The `rabbitSend` method accepts 2 parameters. The first parameter is a queue name and the second parameter is the message being sent.

```
class MessageController {

def sendMessage = {
        rabbitSend 'someQueueName', 'someMessage'
        …
    }
}
```

Messages may also be sent by interacting with the `RabbitTemplate` Spring bean directly. See the Using The RabbitTemplate Directly section for more information.

4 Consuming Messages

The plugin provides two simple ways of consuming messages:

1. from a named Queue
2. by subscribing to an exchange (the traditional pub/sub model)

Which approach you take depends on whether you want to implement the pub/sub messaging model and how much control you need.

4.1 Pub-Sub

One of the most common messaging models people use involves a producer broadcasting messages to all registered listeners (or more accurately, consumers). This is known as the publish/subscribe model, or pub/sub for short. There are two steps to getting this set up in Grails:

1. create the exchange you're going to publish messages to
2. create some consumers that subscribe to that exchange

The first step can be done either outside of the Grails application or in the plugin's configuration. If the Grails application is the publisher, then it makes sense to declare the exchange in `grails-app/conf/Config.groovy`.

The second step is dead easy with the plugin: create a service with a static `rabbitSubscribe` property and a `handleMessage()` method. Here's an example:

```
package org.example

class SharesService {
    static rabbitSubscribe = 'shares'

void handleMessage(message) {
        // handle message…
    }
}
```

As long as the broker contains an exchange with the name `shares`, the `SharesService` will receive all messages sent to that exchange. Every time a message is received from the broker, the service's `handleMessage()` method is called with the message as its argument. We'll talk more about messages shortly.

⚠ The `rabbitSubscribe` option only makes sense when applied to fanout and topic exchanges.

In the case of a topic exchange, you can filter messages based on the routing key. By default your service will receive all messages, but you can override this with an alternative syntax for `rabbitSubscribe`:

```
package org.example

class SharesService {
    static rabbitSubscribe = [ name: 'shares', routingKey: 'NYSE.GE' ]
    …
}
```

In this example, the service will only receive messages that have a routing key of 'GE'. Of course, you can use standard AMQP wildcards too like 'NYSE.#', which will match all messages with a routing key that starts with 'NYSE.'.

Under the hood, the plugin creates a temporary, exclusive queue for your service which is removed from the broker when your application shuts down. There is no way for you to control the name of the queue or attach another listener to it, but then that's the point in this case. If you do need more control, then you must manage the queues and their bindings yourself.

The map syntax also allows you to customise the properties of the Spring message listener container and the corresponding listener adapter (see the section on advanced configuration for more details on these). For example,

```
static rabbitSubscribe = [
        name: 'shares',
        routingKey: 'NYSE.GE',
        encoding: "ISO-8859-1",
        prefetchCount: 1]
```

will set the encoding and prefetch count for just this service listener. This technique is also possible for straight queue listeners as well.

4.2 Manual queue management

The plugin provides a convention based mechanism for associating a listener with a queue. Any Grails Service may express that it wants to receive messages on a specific queue by defining a static property named `rabbitQueue` and assigning the property a string which represents the name of a queue.

```
package org.grails.rabbitmq.test

class DemoService {
    static rabbitQueue = 'someQueueName'

void handleMessage(message) {
        // handle message…
    }
}
```

As with the pub/sub model, messages are delivered to the service by invoking the `handleMessage()` method. That's all there is to it! The real trick is to configure your exchanges and queues with appropriate bindings, as we described in the configuration section.

If you want more say in the configuration of the underlying listener, then you can also specify a map:

```
static rabbitQueue = [queues: "someQueueName", channelTransacted: true]
```

The "queues" option can either be a simple queue name or a list of queue names. Again, have a look at the advanced configuration section for information about the extra properties you can set here.

One last subject to discuss is the form that the messages take.

4.3 Messages

What is a message? In the examples you've seen in this section, the message has been some arbitrary object but we haven't discussed what the type of that object might be. That's because, it can be pretty much anything! Within the messaging system, the content of a message is simply a byte array - it's up to the producer can consumer to interpret/convert that raw data.

Fortunately the plugin (via Spring AMQP) automatically handles messages whose content is in familiar forms, including:

- strings
- byte arrays
- maps
- other serializable types

One manifestation of this support is that different message types may be handled with overloaded versions of `handleMessage()`:

```
package org.grails.rabbitmq.test

class DemoService {
    static rabbitQueue = 'someQueueName'

void handleMessage(String textMessage) {
        // handle String message…
    }

void handleMessage(Map mapMessage) {
        // handle Map message…
    }

void handleMessage(byte[] byteMessage) {
        // handle byte array message…
    }
}
```

This is a great convenience, but be aware that using serializable Java objects limits the types of client you can interact with. If all the clients you're interested in are using Spring AMQP, then you should be fine, but don't expect Ruby or Python clients to handle `Map` messages! For production systems, we recommend you use strings and byte arrays.

Sometimes you want access to the raw message, particularly if you want to look at the message headers. If so, just change the signature of the `handleMessage()` method and add an extra option to your `rabbitQueue` or `rabbitSubscribe` property:

```
package org.grails.rabbitmq.test

import org.springframework.amqp.core.Message

class DemoService {
    static rabbitQueue = [queues: 'someQueueName', messageConverterBean: '']

void handleMessage(Message msg) {
        // Do something with the message headers
        println "Received message with content type
${msg.contentType};${msg.encoding}"
        …
    }
}
```

As you can see, all you have to do is accept an argument of type `Message` and add the `messageConverterBean` option with an empty string as its value. This disables the automatic message conversion, allowing you to interrogate the raw message as required.

5 Using The RabbitTemplate Directly

Most of the interaction with the RabbitMQ server is being handled by an instance of RabbitTemplate. For many applications this is happening at a lower level than the application needs to be concerned with. The plugin does provide a Spring bean to the application context that is an instance of the `RabbitTemplate` class which may be used directly. The bean name is `rabbitTemplate`.

```
class MessageController {

def rabbitTemplate

def sendMessage = {
        rabbitTemplate.convertAndSend('someQueueName', 'someMessage)
        …
    }
}
```