12 Understanding messaging and scheduling

In this chapter

- What messaging is and how it works
- How messaging aids scalable architectures
- Points in your Grails application that benefit from messaging
- How to implement queues and topics with a popular JMS server
- How scheduling is implemented in Grails
- Scheduling jobs with cron-style expressions
- Programmatically creating and managing scheduled jobs
- Advanced scheduling use cases for persistent and clustered jobs

In the last chapter, we spent some time investigating remoting options for Grails applications and looked at generating and consuming interapplication messages. In this chapter, we'll keep you in that enterprise headspace, but we'll look at sending intra-application messages. In particular, we'll examine how different components in an application can communicate *internally* while different events

Licensed to Manning Marketing <stho@manning.com>

in the application's lifecycle unfold. One of the most popular ways of doing that is via messaging queues, an architecture sometimes referred to as *message-oriented middleware* (MOM).

If you've been around enterprise circles for a while, you've probably used or heard of MOM architectures. You might be thinking it's some kind of heavyweight old-school technology that just won't die. Nothing could be further from the truth. In fact, with the birth of service-oriented architecture's (SOA's) Enterprise Service Bus (ESB), and the rise of massive Web 2.0 social networking sites, we're experiencing an explosion of interest in messaging architectures.

You might be wondering why these styles of architecture have had such a resurgence in recent years. From Twitter to Digg to LinkedIn, if you look behind any of today's big Web 2.0 applications, you'll find that they're backed by an extensive messaging infrastructure. These messaging architectures are so prevalent at high-volume sites for three reasons:

- They lead to loosely coupled architectures, which means you can replace parts of your infrastructure without any client downtime.
- They have high scalability—you can add more components to process work on your queue.
- They offer a reliable transport that ensures your messages and transactions don't get lost in the system.

In this chapter, we'll add a messaging system to Hubbub so we can create a link between Hubbub and Jabber, a popular instant messaging (IM) system. By the time we're done, you'll be able to post messages to your Hubbub account via your IM client, and we'll also bridge the other way so you can be notified of your friends' Hubbub posts in your IM client. Along the way, you'll learn the ins and outs of all the common messaging scenarios and get some ideas on how to apply them to your current projects.

But messaging isn't the only asynchronous game in town. In many situations, a lightweight scheduling solution is all you need. Kicking off a daily backup? Sending out daily digest emails? Regenerating your full text index? Every developer needs to deal with these kinds of scheduled events occasionally, and Grails offers a robust and easily configurable scheduler based on the popular Quartz framework. We'll look at the different ways you can schedule jobs—how to write daily-digest type jobs, how to turn them off and on while your application is running, and how scheduling works in clustered environments.

We'll get into the details of scheduling a little later. For now, let's sink our teeth into some messaging.

12.1 A hitchhiker's guide to messaging

Messaging has been around for ages, but its predominant use has been in large enterprise scenarios, so you may never have been exposed to how this style of architecture works. In this section, we'll discuss the basics of how messaging works and get you sending and receiving Java Message Service (JMS) messages. Buckle up!

Licensed to Manning Marketing <stho@manning.com>

12.1.1 Learning to think in async: what are good messaging candidates?

Often, parts of your application are time- and resource-intensive and don't need to be done immediately. One example is generating a PDF flight itinerary and emailing it to the user. When the user books the flight, you tell them you're emailing the PDF to them, but the work doesn't have to be done that instant. Generating the PDF is likely to be CPU-intensive, and you don't want to hold up all users' web experience while the server is bogged down generating one user's PDF. Realistically, the PDF can be generated and emailed anytime in the next minute or so. It needs to be done soonish, and it needs to be done reliably.

This is a classic example of a candidate for messaging, and this "do it soon" approach is known as *asynchronous processing*. Here's how it might work behind the scenes. When the user requests a flight itinerary, a message is placed on an itinerary message queue. That can be done immediately, and you can report to the user that the PDF is in the mail. Another process, perhaps even on a different server (inside a firewall, with access to a mail server), retrieves the itinerary request off the queue, generates the PDF, and emails it to the user. Figure 12.1 shows the PDF request flowing through the queue to the target process.

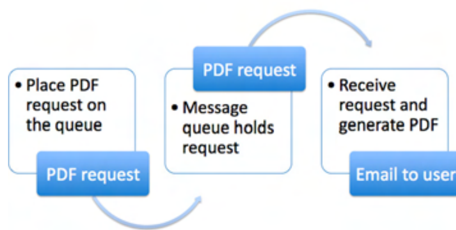

Figure 12.1 A PDF request flows through a message queue to a target process.

One of the cool parts of this asynchronous approach is that the messaging server persists the messages on the queue, which means that the messages will remain until a client is available to service them. If generating PDFs is a bottleneck, you can have many clients listening on the queue to partition the work of generating and mailing PDFs, and the messaging server will preserve the transactional semantics, making sure requests are removed from the queue once they've been serviced.

Now that you understand where asynchronous systems can make sense, it's time to get acquainted with some of the implementation terminology you need to know. Let's implement our first queue-based feature for Hubbub.

12.1.2 Messaging terminology: of producers, consumers, topics, and queues

Before you implement messaging, you need to understand some of the basic JMS terminology. All of the plugin documentation and sample articles will assume you know what topics, queues, and producers are, so we'll first cover that and give you a feel for which situations lend themselves to which messaging techniques.

Licensed to Manning Marketing <stho@manning.com>

First, there are two types of actors in the JMS market:

- *Producers* produce and place messages on the queue.
- *Consumers* pull entries off the queue.

In our PDF example, the web application (the producer) posts new PDF requests to the queue, and the PDF-emailing application (the consumer) pulls them off.

How do consumers and producers communicate? JMS offers two main communication models:

- *Queues*—Queues operate on a FIFO (first in, first out) principle, where each message that a producer places on a queue is processed by one (and only one) consumer. This is sometimes known as *point-to-point* messaging, and it's demonstrated in figure 12.2.

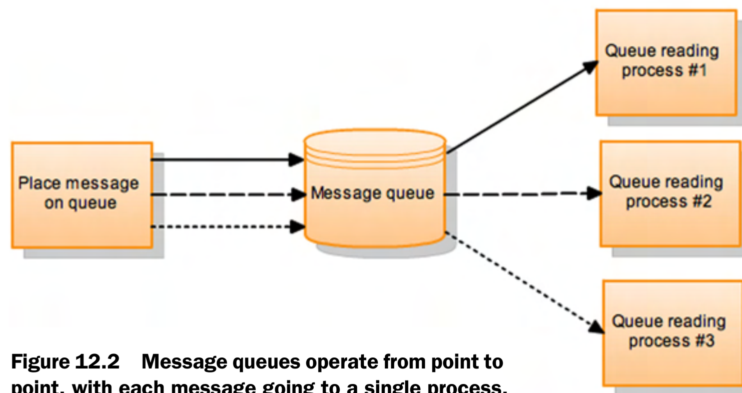

Figure 12.2 Message queues operate from point to point, with each message going to a single process.

- *Topics*—Topics use a *broadcast* model where all listeners on the topic get a copy of the message. The producer places one message on the queue, but that message is duplicated and shuffled off to many consumers simultaneously. This design is shown in figure 12.3.

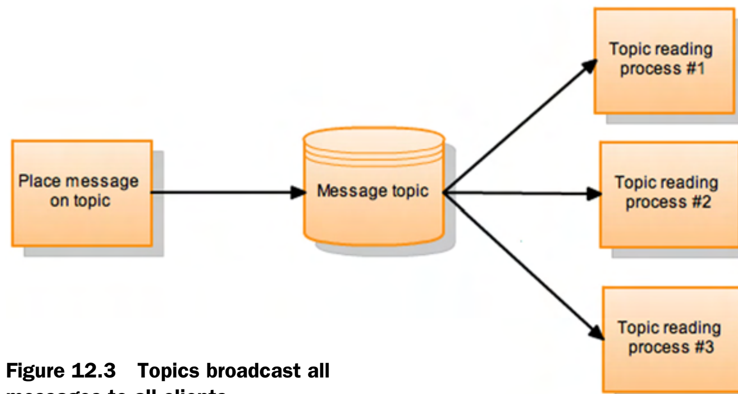

Figure 12.3 Topics broadcast all messages to all clients.

Licensed to Manning Marketing <stho@manning.com>

One example that may work well with a topic-style architecture is a network-monitoring application. For example, when a device in the system experiences an outage, a monitoring application can broadcast on a topic to notify other system components to use an alternative device. In this scenario, all listeners on the topic process the incoming message. In our PDF example, you only want your PDF-generation messages processed once, so you should use a queue.

12.1.3 Installing and configuring the JMS plugin

With the theory out of the way, let's take a look at how to implement a basic messaging system for Hubbub. The first step is installing the JMS plugin:

```
grails install-plugin jms
```

Next, you need to make some decisions about choosing a messaging provider. There are currently a few dominant ones in the industry—some are open source (ActiveMQ, Open MQ, JBoss Messaging) and some are commercial (IBM's WebSphere MQ). It's a requirement of the Java EE specification that an application server ship with a JMS container, so the decision may have been made for you (Open MQ ships with Glassfish, for example, and it's a great JMS server).

But if you're planning to deploy to a servlet container (like Tomcat or Jetty), then you're free to choose any provider you like.

Only small differences in configuration exist between the vendors, so we'll use ActiveMQ, a popular open source messaging server, in this chapter.

NOTE If you get a message saying "Compilation error: java.lang.NoClassDefFoundError: javax/jms/MessageListener" when installing the JMS plugin or when running the first time after installing, it means you need a J2EE JAR file in the lib directory of your Grails application. This JAR file defines the JMS interfaces and supporting classes. If you're using Open MQ, you can use jms.jar. If you're using ActiveMQ, use activemq-all-5.1.0.jar.

INSTALLING ACTIVEMQ

ActiveMQ is open source, free, and popular—it's currently the messaging stack used by LinkedIn, for instance (http://hurvitz.org/blog/2008/06/linkedin-architecture). It's the messaging provider we'll be using in this chapter.

To install it, download a copy of ActiveMQ from http://activemq.apache.org/ and unzip it into your preferred installation location. No configuration is required, so start /activemq/bin/activemq from a new command prompt. Once the startup process is complete, you can access the ActiveMQ console via the browser at http://localhost:8161/admin/. Figure 12.4 shows the interface in action.

The ActiveMQ console lets you browse your queues and topics to make sure your messages are getting through—we'll explore that later. Now that the messaging server is running, it's time to configure Hubbub to point to it.

Licensed to Manning Marketing <stho@manning.com>

Figure 12.4 The ActiveMQ console is available via a browser interface.

CONFIGURING YOUR PROVIDER

After installing the JMS plugin and starting your messaging server, you may need to do two further things to set it up:

- Configure your messaging service in /grails-app/conf/spring /resources.groovy.
- Add your JMS provider's JAR file(s) to your project's /lib directory.

Let's tackle the message-service configuration first. Each JMS provider supplies a connection factory class that is responsible for establishing connections to your JMS provider. For ActiveMQ, the connection factory needs the hostname and port of the messaging server, so let's update resources.groovy to give the plugin the information it needs. Listing 12.1 shows the required changes.

Listing 12.1 Updating resources.groovy to connect to ActiveMQ

```
import org.apache.activemq.ActiveMQConnectionFactory        // Imports broker factory

beans = {

   connectionFactory(ActiveMQConnectionFactory) {           // Defines broker connection
      brokerURL = "tcp://localhost:61616"                   // Configures broker endpoint
   }

}
```

Next up, you need to copy the ActiveMQ JAR files into the application's lib directory. For the current version of ActiveMQ, there's just one JAR file, activemq-all-5.1.0.jar. This provides the `ActiveMQConnectionFactory` class.

Licensed to Manning Marketing <stho@manning.com>

We're now configured and ready to go. It's time to harness the power of the Grails JMS plugin to send some JMS messages.

12.2 Using the Grails JMS plugin

The Grails JMS plugin gives you a simple way to both send to and receive from JMS topics and queues. Like most Grails plugins, it uses a sensible Convention over Configuration approach to make sure you spend your time sending messages, not configuring queues (although there are overrides for all the conventions if you want to set up your own queue names).

In this section, we'll cover the basics of getting messages onto a queue and reading them off. We'll also beef up Hubbub with an instant messaging gateway.

12.2.1 Our killer Hubbub feature: IM integration with Jabber

Let's consider what messaging features we should implement for Hubbub. One that would be cool is instant messaging (IM) integration, so for Hubbub we'll write a simple IM gateway to bridge to the popular IM networks. For this example, we'll write a messaging gateway for Jabber, a popular open source IM system that can gateway to other clients (AIM, Yahoo! IM, and so on).

If a Hubbub user registers their IM account, we can let them post from their favorite IM client; and if they're online, we can even send them back posts from users on their timeline to keep them up to date in real time. When the user adds a new post, we'll put it on a messaging queue (to be broadcast to any followers that are IM active). Similarly, if the user sends an IM message to the Hubbub bot, we'll put it on an incoming queue to be posted on the Hubbub website. We'll use the IM transport to read and send.

Figure 12.5 shows our basic architecture with messages flowing between Hubbub and the gateway through the queue.

But before we can implement our gateway, we need to look at what's involved in putting outgoing messages on a JMS queue.

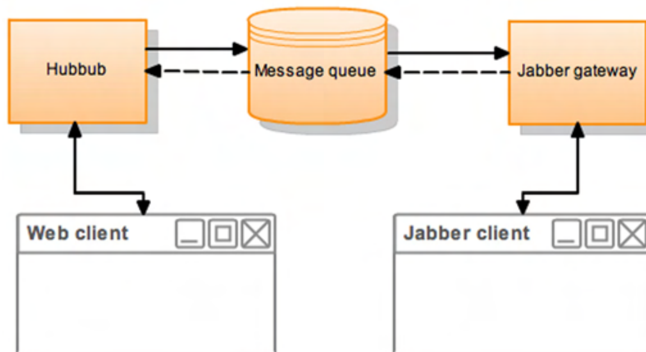

Figure 12.5 Our Jabber gateway architecture

Licensed to Manning Marketing <stho@manning.com>

12.2.2 Sending JMS messages

The JMS plugin works by empowering all your service and controller classes with new JMS-related methods. Which method you invoke depends on whether you're sending your message to a queue or a topic. Table 12.1 lists the methods for each destination type.

Table 12.1 Method names for each destination type

Destination	Method
Queue	`sendJMSMessage()`
	`sendQueueJMSMessage()`
Topic	`sendPubSubJMSMessage()`
	`sendTopicJMSMessage()`

There are two methods each for queues and topics, but they're aliases to one another, so feel free to use whichever makes more sense to you. For our examples, we'll use `sendQueueJMSMessage()` for queues and `sendTopicJMSMessage()` for topics because they make things explicit (which is a good thing for other developers).

Whether you're dealing with queues or topics, the method parameters are the same. The first parameter is the destination name (that is, the name of the queue or topic in your messaging server), and the second parameter is the payload of the message. ActiveMQ doesn't require that you precreate queue names, but your provider may differ.

In listing 12.2, we add a `JabberService` class that handles sending our messages. We'll place a `Map` holding all of our relevant message data on the queue.

Listing 12.2 Implementing a Jabber service

```
class JabberService {

   void sendMessage(post, jabberIds) {

      log.debug "Sending jabber message for ${post.user.userId}..."
      sendQueueJMSMessage("jabberOutQ",
        [ userId: post.user.userId
          content: post.content,
          to: jabberIds.join(",") ] )

   }

}
```

Places Map on the queue

All the infrastructure is in place, but nothing is available yet to read off the queue. Let's write a test harness to generate some traffic. Listing 12.3 shows a basic test case to exercise the service.

Licensed to Manning Marketing <stho@manning.com>

Listing 12.3 Exercising our Jabber service with an integration test

```
class JabberServiceTests extends GroovyTestCase {

   def jabberService

   void testWriteToQueue() {

      def post = [user: [userId: 'chuck_norris'],
         content: 'is backstroking across the atlantic']
      def jabberIds = ["glen@grailsinaction.com",
         "peter@grailsinaction.com" ]
      jabberService.sendMessage(post, jabberIds)

   }

}
```

Make sure you've started up ActiveMQ, and then give the test case a run with `grails test-app JabberService`. This test will show that you can put elements on a queue.

After the test is finished, point your browser to the ActiveMQ console at http://localhost:8161/admin. In figure 12.6, we have selected the Queues menu to see our new jabberOutQ queue.

Browse jabberOutQ

Message ID	Correlation ID	Persistence	Priority	Redelivered	Reply To	Timestamp	Type	Operations
ID:decaf.local-52961-1222939837053-0:1:1:1:1		Persistent	4	false		2008-10-02 19:30:37:906 EST		Delete

Figure 12.6 Browsing the Jabber queue using ActiveMQ's web interface

You can click individual messages to see that everything has arrived safely. Figure 12.7 shows what you'll see when you inspect the contents of an individual message.

All our message details look in order, and our `Map` of data is persisted on the queue awaiting a listener to retrieve it. But before we look at how to read messages, let's take a detour into what types of payload you can put on a queue.

You've just seen map messages being put on a JMS queue, and you might be wondering what sorts of things are queueable. You can send several basic JMS data types to a destination, and they're listed in table 12.2.

Table 12.2 The basic data types for JMS messages

Type	Example
`String`	`"Message from ${user.name}"`
`Map`	`[ name: "glen", age: "36", job: "stunt programmer" ]`
`byte[]`	`image.getBytes()`
`Object`	Any object that implements `Serializable`

Licensed to Manning Marketing <stho@manning.com>

Headers		Properties
Message ID	ID:decaf.local-52961-1222939837053-0:1:1:1:1	
Destination	queue://jabberOutQ	
Correlation ID		
Group		
Sequence	0	
Expiration	0	
Persistence	Persistent	
Priority	4	
Redelivered	false	
Reply To		
Timestamp	1222939837906	
Type		

Message Details

{to=glen@grailsinaction.com,peter@grailsinaction.com, userId=chuck_norris, content=is backstroking across the atlantic}

Figure 12.7 Inspecting a message on the queue

Although some people prefer to use XML payloads in the string data type, we've found the `Map` style message to be the most flexible. Using a `Map` means you can easily add new properties to objects you're sending to a destination without having to worry about breaking any parsing code elsewhere in the system.

> **How does type conversion work?**
>
> If you've worked with JMS before, you know that JMS supports its own type system (`TextMessage`, `MapMessage`, `BytesMessage`, and so on). The JMS plugin does the conversion of payload data to the appropriate type for you, leaving you to get on with building your application.
>
> Behind the scenes, the JMS plugin uses the standard Spring JMS `Template` class. By default, this class delegates all type conversion to Spring's `SimpleMessageConverter`, which handles marshaling the basic types listed in table 12.2.

We've done the hard work of getting everything on our queue, so it's time to implement the queue-reading side of things so we can get our work done.

12.2.3 Reading the queue

By taking advantage of a convention-based model, the JMS plugin makes the reading process straightforward. For the basic implementation, you need to do three things:

- Add an entry to your service class to expose it as a queue or topic listener.
- Provide an `onMessage()` method to handle incoming messages.

Licensed to Manning Marketing <stho@manning.com>

- Override conventions (when required) to match your queue names and the quantity of listener threads.

Let's cover each of those steps to get you up and running.

IMPLEMENTING YOUR FIRST ONMESSAGE()

First, only services can be exposed as JMS endpoints. To let the plugin know that a service is a JMS endpoint, you need to include the following line in your class definition:

```
static expose = ['jms']
```

The `expose` property is used by a number of remoting plugins (XFire, Remoting, Jabber), and you can happily mix and match SOAP and JMS endpoints in the same service class.

Next, we need to add an `onMessage()` method that the plugin can call when new messages arrive. That gives us a complete messaging class. Listing 12.4 implements the new feature.

Listing 12.4 Handling an incoming message in the service

```
class JabberService {

   static expose = ['jms']

   static destination = "jabberInQ"          // 1 Names queue to listen on

   static listenerCount = 5                  // 2 Specifies number of listening threads

   void onMessage(msg) {
      log.debug "Got Incoming Jabber Response from: ${msg.jabberId}"
      try {
         def profile = Profile.findByJabberAddress(msg.jabberId)
         if (profile) {
            profile.user.addToPosts(new Post(content: msg.content))
         }
      } catch (t) {                          // 3 Catches error conditions
        log.error "Error adding post for ${msg.jabberId}", t
      }
   }

   void sendMessage(post, jabberIds) {

      log.debug "Sending jabber message for ${post.user.userId}..."
      def msg = [userId: post.user.userId,
            content: post.content, to: jabberIds.join(",")]
      sendQueueJMSMessage("jabberOutQ", msg)

   }
}
```

Notice that we've specified the `destination` property of the queue (1). Following convention, the JMS plugin takes the queue name from the service, so our `JabberService` defaults to a queue name of "Jabber". In this example, we want our incoming and outgoing queue names to follow a different standard, so we overwrite the `destination` property to tell the plugin what name we want.

Licensed to Manning Marketing <stho@manning.com>

In addition to customizing the queue name, we've also customized the number of threads that will be listening on the queue (2). The default is 1, but we increased that to 5 because we're expecting a lot of messages on the queue.

Finally, we're particular about handling exception cases (3). Some messaging servers get upset if clients don't behave well when reading from open connections, so we make sure that we terminate nicely when experiencing stray or malformed messages.

PULLING OUT THE STOPS: IMPLEMENTING A JABBER GATEWAY APPLICATION

Now that our messaging interface is up and running in the web-facing portions of Hubbub, it's time to write an application to interface with the Jabber protocol. To make things simple, we'll write our gateway application as a Grails application and use the JMS and Jabber plugins to interface with the rest of the internet.

You can install the Jabber plugin using the normal Grails plugin installation mechanism:

```
grails install-plugin jabber
```

The Jabber plugin works much like the JMS plugin you're already familiar with. The Jabber plugin identifies any service class marked with an `expose = ['jabber']` property and automatically adds a `sendJabberMessage()` method. If the service offers an `onJabberMessage()` closure, the plugin will call it when any Jabber message arrives on the configured queue.

After installing the JMS and Jabber plugins, the whole application is implemented in a single service class as shown in listing 12.5.

Listing 12.5 A gateway service that reads and writes Hubbub messages to Jabber

```
class GatewayService {

   static expose = ['jabber', 'jms']

   static destination = "jabberOutQ"

   void onMessage(msg) {

      log.debug "Incoming Queue Request from:
        ${msg.userId} to: ${msg.to} content: ${msg.content}"

      def addrs = msg.content.split(",")
      addrs.each {addr ->
         log.debug "Sending to: ${addr}"
         sendJabberMessage(addr, msg.content)
      }

   }

   void onJabberMessage(jabber) {

      log.debug "Incoming Jabber Message Received
        from ${jabber.from()} with body ${jabber.body}"
      def msg = [jabberId: jabber.from, content: jabber.body]
      sendQueueJMSMessage ("jabberInQ", msg)

   }

}
```

(1) Marks service as JMS and Jabber-aware
(2) Sets JMS queue name
(3) Receives incoming JMS messages
(4) Sends message to Jabber queue
(5) Receives incoming Jabber messages
(6) Sends message to JMS queue

Licensed to Manning Marketing <stho@manning.com>

NOTE The complete source for the application is included with the source code for this chapter under the name jabber-gateway.

Our `GatewayService` starts with the configuration for receiving both JMS and Jabber messages ❶ and for setting up the name of the JMS queue (`destination`) it will be listening on ❷. It then implements `onMessage()` for JMS messages ❸ and takes incoming JMS messages and sends them to a Jabber destination that it pulls from the message ❹.

Finally, the service implements `onJabberMessage()` ❺, which receives Jabber messages and places them on the message queue for Hubbub to process and add to users' timelines ❻.

With those 20 or so lines of code, we've implemented a two-way gateway from Jabber to JMS! As you can see, harnessing the power of plugins can lead to massive reductions in the code you'll need to maintain.

> **When to topic, when to queue?**
>
> We've now had a good look at sending and receiving JMS messages via a queue. Using a queue made a lot of sense in this case, because we only wanted our messages to be processed once.
>
> Topics, on the other hand, are ideal for broadcast scenarios—where you want all listeners to be updated about a particular event. Imagine writing a network-monitoring system to keep track of when your services go up or down. The node responsible for probing servers might want to let everything else in the system know when a server crashes. Topics are ideal for this kind of broadcast scenario.

That covers the basics of messaging. It's now time to explore another, more lightweight, alternative for our asynchronous needs: Grails scheduling.

12.3 Grails scheduling

We've looked at messaging architectures and seen some easy ways to take advantage of their asynchronous approach to making systems simpler, more scalable, and more flexible. But quite a lot of infrastructure is involved in getting such a reliable and performant architecture.

Sometimes, you want a simple asynchronous solution to run some function at a scheduled time (for example, a daily report, an index update, or a daily backup). For those scenarios, Grails offers a fantastic, easy-to-use scheduling capability, and it's time to explore it in some depth.

12.3.1 Writing a daily digest job

Grails' support for scheduling operations is handled by the Quartz plugin. Quartz is a popular Java library with robust and powerful scheduling capabilities, and the Quartz plugin gives you a simple Grails-style way to access all that power. Let's use it to send a

Licensed to Manning Marketing <stho@manning.com>

daily digest email to each Hubbub user, outlining all the activity on their followers' timelines for the past day.

We'll start by installing the plugin:

```
grails install-plugin quartz
```

With the plugin installed, you'll notice that two new commands are available:

```
grails create-job
grails install-quartz-config
```

The first is used to create new job templates (much like `grails create-service`), and the second installs a custom Quartz configuration file (which is only needed for advanced use cases like clustering; we'll talk more about it later).

To create our daily digest email, we need to create a new job that will run each night:

```
grails create-job DailyDigest
```

This newly created job class is located in grails-app/jobs/DailyDigestJob.groovy.

The simplest way to use jobs is to specify a `timeout` value in milliseconds. Every time an interval of `timeout` passes, the plugin invokes your job. Listing 12.6 shows the shell for our daily digest job.

Listing 12.6 A basic daily digest job (using `timeout` style)

```
class DailyDigestJob {

   def timeout = 24 * 60 * 60 * 1000        // Runs job once per day
   def startDelay = 60 * 1000               // Delays first run for one minute

   def execute() {
      log.debug "Starting the Daily Digest job."
      // ... do the daily digest
      log.debug "Finished the Daily Digest job."
   }

}
```

Notice that we've also added a `startDelay` field, which is the initial wait period before the plugin invokes your job. This is if handy you have tight timeouts (a few seconds) but you want to make sure the rest of your Grails application has finished bootstrapping before the first job fires.

At this stage, you might be tempted to implement your business logic in the job class. This is supported, but it's almost never a good idea. Because jobs support the same injection-based conventions as other artifact classes, it's much better to call an injected service rather than implement the process inline. Using an injection-based approach makes things much more testable, and it also fosters code reuse. Our newly created job is refactored in listing 12.7 to tidy things up.

Licensed to Manning Marketing <stho@manning.com>

Listing 12.7 A basic daily digest job

```
class DailyDigestJob {

   def timeout = 24 * 60 * 60 * 1000
   def startDelay = 60 * 1000

   def dailyDigestService

   def execute() {
      log.debug "Starting the Daily Digest job."
      dailyDigestService.sendDailyDigests()      <-- Encapsulates logic in service class
      log.debug "Finished the Daily Digest job."
   }

}
```

By defining our `dailyDigestService` field, the Quartz plugin will make sure everything is nicely wired together before any jobs are started.

Now that our daily digest is up and running, it's time to rethink our scheduling mechanism. So far, we've been using simple Quartz scheduling, which is fine for jobs that need to fire every so many seconds. But we'd prefer our daily digest to be sent out at the same time each day: perhaps 1 A.M., when things are quiet on the servers. To get that kind of calendar-based flexibility, we'll need to get acquainted with the cron scheduler.

12.3.2 Fine-grained scheduling with cron

If you have any kind of UNIX background, you're probably familiar with the cron service. Cron is a UNIX facility that allows you to schedule jobs to run on certain days at certain times, or on a particular day of the week or month, with all kinds of flexibility. With that flexibility comes a rather arcane syntax that only a hardcore command-line fiend could love. Figure 12.8 shows the basic components of a cron expression.

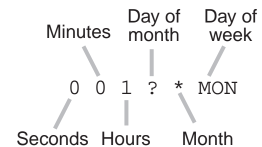

Figure 12.8 The basic components of a cron expression

As shown in figure 12.8, each field of the cron expression refers to a different time period. This example tells cron to run the job at 1 A.M. every Monday.

Cron expressions give you incredible scheduling power, but the syntax is certainly something to wrestle with. All fields in a cron expression can take numbers, wildcards (*), ranges (5-15), sets (5,10,15), or increments (10/15). There are some special cases for the month and day-of-week fields where you can use special literals. For months, you can use expressions like JAN-MAR; and for the days of the week, you can use expressions like MON-FRI.

It's much easier to understand cron expressions when you see a few in action. Table 12.3 lists some common expressions.

Licensed to Manning Marketing <stho@manning.com>

Table 12.3 A series of basic cron expressions

Expression	Description
0 0,45 1 ? * MON-FRI	Every weekday at 1 A.M. and 1:45 A.M.
0 0/15 1 ? * MON	Every 15 minutes from 1 A.M. to 1:45 A.M. on a Monday
0 0 10-12 1 * ?	10 A.M., 11 A.M., and 12 P.M. on the first of the month
0 0 0 1 1 ?	Midnight on New Year's Eve

TIP The Quartz website has a comprehensive reference to cron expressions and more examples. Check out the tutorials section of the website for a comprehensive walkthrough: http://www.opensymphony.com/quartz/wikidocs/CronTriggers%20Tutorial.html.

With that little bit of dangerous knowledge under our belt, let's re-implement our daily digest service so that it runs at 1 A.M. each weekday. Listing 12.8 shows the new version of our job.

Listing 12.8 A basic daily digest job with custom cron settings

```
class DailyDigestJob {

   def cronExpression = "0 0 1 ? * MON-FRI"

   def dailyDigestService

   def execute() {
      log.debug "Starting the Daily Digest job."
      dailyDigestService.sendDailyDigests()
      log.debug "Finished the Daily Digest job."
   }

}
```

Supplies cron-style expression to Quartz job

That covers the basic scheduling operations available in Grails. It's time now to explore some of the more advanced options for putting the scheduler to work.

12.4 Advanced scheduling

We've covered a lot of the common scenarios for Grails scheduling, and you're probably full of ideas for adding these kinds of jobs to your next Grails application. But there's still plenty to explore. In this section, we'll create, trigger, and control jobs programmatically, and we'll add an administrative UI so we can control them directly from our application. We'll also look at sharing data between job runs, or sharing jobs in a cluster. By the time we're finished, you'll know the fine points (and gotchas) of all these scenarios.

Let's start by getting acquainted with how the scheduling plugin handles stateful and re-entrant jobs.

Licensed to Manning Marketing <stho@manning.com>

12.4.1 Dealing with re-entrance and stateful jobs

By default, the Quartz plugin creates a new instance of your job class and calls it each time your job runs. But there may be situations when you don't want two instances of your job to fire at the same time.

Imagine you have an SMS notifier for Hubbub. A timer job fires every 10 seconds to see if there are any unsent SMS messages; if there are, it shuffles them off to an SMS web service that sends them. But what happens if the SMS service takes 60 seconds to time out? Your job might fire again, and again, and again within the same minute, resulting in multiple (annoying) message sends. You could work around this by keeping a `processed` field on the message itself, or use a JMS queue for the sending; but assuming you've ruled those out, you'll want a way to make sure your job is never run concurrently.

The Quartz plugin protects against concurrency via the `concurrent` property. Listing 12.9 demonstrates this feature.

Listing 12.9 Using the `concurrent` property to stop re-entrance

```
class SmsSenderJob {

   def timeout = 10000 // execute job every 10 seconds
   def concurrent = false

   def execute() {
      log.error "Sending SMS Job at ${new Date()}"
      Thread.sleep(20000)                                  <- Simulates web service delay
      log.error "Finished SMS Job at ${new Date()}"
   }
}
```

If you run this application, you'll see that even though the timeout is specified to run every 10 seconds, the simulated delay means it runs only when the job isn't already running. Here's the output of this sample SMS job:

```
[22642] task.SmsSenderJob Sending SMS Job at Tue Oct 14 13:40:38 EST 2008
[42643] task.SmsSenderJob Finished SMS Job at Tue Oct 14 13:40:58 EST 2008
[42646] task.SmsSenderJob Sending SMS Job at Tue Oct 14 13:40:58 EST 2008
[62649] task.SmsSenderJob Finished SMS Job at Tue Oct 14 13:41:18 EST 2008
[62653] task.SmsSenderJob Sending SMS Job at Tue Oct 14 13:41:18 EST 2008
[82654] task.SmsSenderJob Finished SMS Job at Tue Oct 14 13:41:38 EST 2008
```

It's important to understand that if a job is scheduled to run, but another instance is already running, the new job is skipped rather than batched up to run later.

Another consequence of marking a job as not concurrent is that the plugin creates the job as a Quartz `StatefulJob`. That means a shared state area, called a `jobDataMap`, is available for you to share information with subsequent jobs. In our SMS gateway example, we might use a counter to keep track of the number of failed sends, and raise a warning when a large number of jobs have timed out. Listing 12.10 shows how we might implement this.

Licensed to Manning Marketing <stho@manning.com>

Listing 12.10 A stateful job gets a persistent context to work with

```
class SmsSenderWithTimeoutJob {
   def timeout = 10000 // execute job every 10 seconds
   def concurrent = false

   def execute(context) {                          <-- 1
      log.debug "Sending SMS Job at ${new Date()}"

      def failCounter = context.jobDetail.jobDataMap['failCounter'] ?: 0
      log.debug "Failed Counter is ${failCounter}"
      try {
         // invoke service class to send SMS here
         failCounter = 0
      } catch (te) {
         log.error "Failed invoking SMS Service"
         failCounter++
         if (failCounter == 5) {
            log.fatal "SMS has not left the building."
         }
      }
      context.jobDetail.jobDataMap['failCounter'] = failCounter       <-- 2
      log.debug "Finished SMS Job at ${new Date()}"
   }
}
```

In this example, you'll notice that we changed our `execute()` method to take a Quartz `jobContext` argument (1).

The second thing to note is that you can store any kind of `Serializable` object in the context (2): numbers, dates, strings, collections, and so on. Try it out to see the counter value being passed into subsequent executions.

12.4.2 *Pausing and resuming stateful jobs programmatically*

So far, we've explored stateful jobs and looked at how we can handle re-entrance. But what if we want to take control of scheduling programmatically?

The Quartz scheduler lets you pause and resume individual jobs, groups of jobs, or the entire scheduler. In order for your job to be easily controllable, you need to place it in a `group`. Listing 12.11 shows our first crack at a pausable job.

Listing 12.11 The `group` property makes it easy to control jobs programmatically

```
class ControllableJob {

   def timeout = 5000 // execute job once in 5 seconds
   def concurrent = false

   def group = "myServices"                 <-- Sets group property
                                                to control job
   def execute() {                              programmatically
      println "Controllable Job running..."
   }
}
```

Licensed to Manning Marketing <stho@manning.com>

Notice that we've specified a `group` attribute on the job. Later, we'll use the scheduler to gain access to this job via its `group` name.

For now, though, we need a way of getting a handle to the scheduler itself. It will come as no surprise that this can be done via the standard Grails injection pattern. In listing 12.12, we create a controller to tinker with our jobs programmatically.

Listing 12.12 A controller for pausing and resuming jobs programmatically

```
class JobAdminController {

   def quartzScheduler

   def index = { redirect(action:'show') }

   def show = {

      def status = ""

      switch(params.operation) {
         case 'pause':
            quartzScheduler.pauseJob("ControllableJob", "myServices")
            status = "Paused Single Job"
            break
         case 'resume':
            quartzScheduler.resumeJob("ControllableJob", "myServices")
            status = "Resumed Single Job"
            break
         case 'pauseGroup':
            quartzScheduler.pauseJobGroup("myServices")
            status = "Paused Job Group"
            break
         case 'resumeGroup':
            quartzScheduler.resumeJobGroup("myServices")
            status = "Resumed Job Group"
            break
         case 'pauseAll':
            quartzScheduler.pauseAll()
            status = "Paused All Jobs"
            break
         case 'resumeAll':
            quartzScheduler.resumeAll()
            status = "Resumed All Jobs"
            break
      }

      return [ status: status ]

   }
}
```

1 Obtains handle to Quartz scheduler object

2 Determines which operation the user selected

Our `JobAdminController` introduces a few important aspects of job control. First, we define the `quartzScheduler` property to inject the scheduler **1**. The `switch` statement demonstrates the different ways you can pause and resume jobs—by name, by group, or globally **2**.

In listing 12.13, we add a basic UI so we can drive the scheduler.

Licensed to Manning Marketing <stho@manning.com>

Listing 12.13 A basic web UI for interacting with our jobs

```
<html>
   <head>
      <title>Job Admin</title>
      <style>
         div#status {
            margin: 1em;
            padding: 1em;
            border: 1px solid blue;
            background: lightblue;
         }
         body {
            font-family: "Trebuchet MS",Helvetica;
         }
      </style>
   </head>

   <body>
      <h1>Job Admin</h1>

      <g:if test="${status}">
         <div id="status">
            ${status}
         </div>
      </g:if>

      <g:form action="show">
         <fieldset>
            <legend>Job Admin Operations</legend>
            <label for="operation">Select an operation:</label>
            <g:select id="operation" name="operation"
                  from="${ [
                  'pause', 'resume',
                  'pauseGroup', 'resumeGroup',
                  'pauseAll', 'resumeAll'
                  ] }" />
            <g:submitButton name="go" value="Go"/>
         </fieldset>
      </g:form>

   </body>
</html>
```

Open http://localhost:8080/quartz/jobAdmin/show to see this admin UI in action, as shown in figure 12.9.

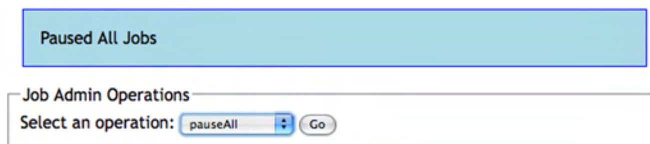

Figure 12.9 Pausing a job via our new admin UI

Licensed to Manning Marketing <stho@manning.com>

Looking at the Grails output to the command line, log messages show that `ControllableJob` (and our other jobs) can be started and stopped via the admin UI. This kind of control comes in handy for the administrative section of your applications, where you want the ability to pause scheduled jobs when dealing with emergency situations (like your SMS service provider going down).

We've now looked at handling stateful and re-entrant jobs. But what happens to our scheduled jobs and their `jobDataMaps` when the application server gets restarted? And what happens when we want to run our jobs in a cluster? For these sorts of cases, you need to learn a little about how Quartz handles persistence.

12.4.3 Persistence and clustering

The default storage mechanism for Quartz is the `RAMJobStore` class, and as you can probably guess from the name, it's fast, but it isn't persistent. If you restart your server, all of your jobs will terminate, and any persistent data in your `jobDataMaps` will be lost. If you'd like your stateful jobs to be persistent, you need to swap out that `RAMJobStore` for something permanent, like a `JDBCJobStore`.

To do that, you need to create a Quartz plugin config file:

```
grails install-quartz-config
```

The preceding command will write a new file in /grails-app/conf/QuartzConfig.groovy that lets you enable JDBC storage. It looks like this:

```
quartz {
   autoStartup = true
   jdbcStore = false
}
```

When `jdbcStore` is set to `true`, your job state will be persisted in the database. But before that can happen, you need to create the required tables.

The SQL to create the tables is found in /plugins/quartz-{version}/src/templates/sql/tables/ inside the Quartz plugin. SQL scripts are available for all the common databases, so use your favorite database admin tool to import the scripts and create the tables.

Once your database has the required tables, you can modify QuartzConfig.groovy to turn on Quartz persistence:

```
quartz {
   autoStartup = true
   jdbcStore = true
}
```

Job persistence is now enabled, but one final change is required before we can rerun an update of listing 12.10 and see if the job state of our counter job survives a restart of the application.

By default, all Quartz jobs are marked as volatile, which means their state won't be persisted to the database. Let's set that right now by marking one of our jobs as non-volatile, as shown in listing 12.14.

Licensed to Manning Marketing <stho@manning.com>

Listing 12.14 Marking jobs as nonvolatile

```
class SmsSenderWithTimeoutJob {
   def timeout = 10000 // execute job every 10 seconds
   def concurrent = false
   def volatility = false
   def execute(context) { ... }
}
```

We're in business. Let's restart our application and see our job write its counters out.

A special note for HSQLDB users

HSQLDB persistence is broken in Quartz 1.6. It may be fixed by the time you read this, but if not, you'll need to create a /src/java/quartz.properties file to customize the Quartz query strings. Place the following line in that file, and you're in business:

```
org.quartz.jobStore.selectWithLockSQL=
➥ SELECT * FROM {0}LOCKS WHERE LOCK_NAME = ?
```

If you want to see persistence in action on HSQLDB, you'll need to change your data source `url` property to a persistent version, such as:

`jdbc:hsqldb:file:devDB;shutdown=true`.

And with our exploration of persistent jobs complete, we've finished our tour of Grails messaging and scheduling features. Let's wrap things up with some best practices to take away.

12.5 Summary and best practices

We covered a lot of asynchronous territory in this chapter. We started by introducing you to the nature of asynchronous technologies and the sorts of applications they're well suited to. We then took you on a tour of basic messaging terminology and jumped in the deep end with the JMS plugin.

After applying our JMS plugin skills to build a Jabber gateway for Hubbub, we moved on to explore the lightweight asynchronous options of Grails scheduling. We looked at using cron-style expressions to implement a daily digest email and then discussed programmatic uses of scheduling.

It's time to review some best practices:

- *Know your application server.* Your application server probably already ships with a JMS provider, and it's time to try using it. If you're running on a servlet container, Open MQ and ActiveMQ are the best options.
- *Choose queues or topics.* Use queues when you want a message to be processed by one listener, and use topics for broadcast scenarios.
- *Favor maps.* Favor map-style messages on queues and topics—they give you greater flexibility to evolve your API over time.

Licensed to Manning Marketing <stho@manning.com>

- *Override convention when needed.* Don't be afraid to override default settings (such as destination names) if it makes your system easier to maintain. It's nice to know what a destination is used for by looking at its name (such as `smsIncomingQueue` and `smsOutgoingQueue`).
- *Know your throughput.* Set the number of queue listeners to match your expected throughput. Don't guess—do some basic profiling to see what your system is capable of under load.
- *Use your console.* The ActiveMQ admin console gives you good insight into what's happening on your queues—take advantage of it. For other servers, Hermes is a good open source queue-browser alternative that works with any JMS container.
- *Separate business logic.* Don't put business logic in Quartz job classes. Make use of service injection to wrap your logic in service classes that are more testable and more reusable.
- *Favor cron.* Cron-style expressions are concise and expressive and give you more consistency than straight `timeout` values.
- *Expose jobs programmatically.* Always give your jobs a `group` attribute so you can get easy programmatic access to them later on.
- *Be cluster-aware.* If you're running in a cluster, you can take advantage of Quartz database persistence to share state and partition the work between nodes.

In the next chapter, we'll move into some of the more advanced use cases for GORM. We'll explore caching, performance optimizations, and integrating legacy databases.

Licensed to Manning Marketing <stho@manning.com>