Extracted from:

Enterprise Recipes with Ruby and Rails

Connect to Message Queues with ActiveMessaging

Problem

Most of your company's infrastructure is based on asynchronous messaging; in other words, vital components can be used only by exchanging messages with them. One of them is a central order handler.

It's your task to build a Rails application for placing orders by sending messages to the company's central order handler. Orders will be stored in a local database, and the application will listen for order status messages emitted by the order handler. This way, the front end can provide a nice and responsive user experience while it can still keep track of the current status of the orders.

Ingredients

- Perform all installation steps described in Recipe 37, *Create a Messaging Infrastructure*, on page 233.
- From your Rails application's root directory, install the *ActiveMessaging*[15] plug-in:

```
mschmidt> script/plugin install \
> http://activemessaging.googlecode.com/svn/trunk/plugins/\
> activemessaging
```

Solution

This scenario is pretty common: a time-consuming task is handed to a back-end service that sends back a result asynchronously when it has finished the task (see a simplified view of our architecture in Figure 8.3, on page 250).

In Recipe 37, *Create a Messaging Infrastructure*, on page 233, you can see how to integrate ordinary Ruby code with message-oriented middleware. This time Rails gets added to the game, and it does not support

15. http://code.google.com/p/activemessaging/

access to messaging architectures natively. But *ActiveMessaging* is a plug-in that makes messaging with Rails a piece of cake.

Before we send and receive messages, we'll build a model for orders in the database:

Download messaging/activemessaging/msgdemo/db/migrate/001_create_orders.rb

```
create_table :orders do |t|
  t.column :customer, :string
  t.column :product,  :string
  t.column :quantity, :int
  t.column :status,   :string, :default => 'OPEN'
  t.timestamps
end
```

Admittedly, this is a rather lightweight order model, but for our purposes it's sufficient. It stores the customer's name, the order's status, and the name and quantity of the product that has been ordered (for an order entry form, see Figure 8.4, on page 251). We could already implement a controller for manipulating it, but our controller does not need to store only orders; it also has to send them to a message queue. We have to edit some configuration files first that have been installed together with the *ActiveMessaging* plug-in.

One of them, broker.yml, defines all connection parameters for the message broker. We'll use ActiveMQ with the STOMP protocol, so our configuration looks as follows (*ActiveMessaging* supports more message brokers, but for the rest of the recipe I assume you're running ActiveMQ in its standard configuration):

Download messaging/activemessaging/msgdemo/config/broker.yml

```
development:
    adapter: stomp
    login: ""
    passcode: ""
    host: localhost
    port: 61613
    reliable: true
    reconnectDelay: 5
```

The next configuration file is messaging.rb. It defines symbolic names for all message queues that we are going to use:

Download messaging/activemessaging/msgdemo/config/messaging.rb

```
ActiveMessaging::Gateway.define do |s|
  s.destination :order,        '/queue/orders.input'
  s.destination :order_status, '/queue/orders.status'
end
```

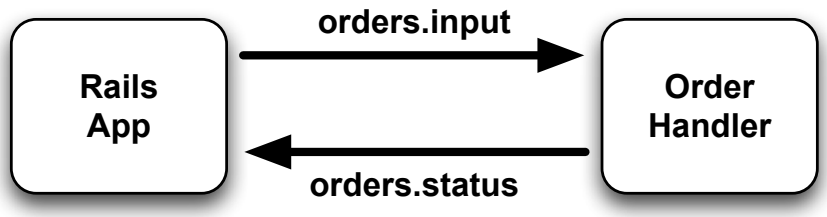

Figure 8.3: High-level architecture

In our application we need two messages queues: one for sending orders (:order) and one for receiving order status messages (:order_status). The symbolic :order queue is mapped to a physical message queue named /queue/orders.input. It's used in the OrderController class to send incoming orders to the central order handler where they get processed asynchronously:

Download messaging/activemessaging/msgdemo/app/controllers/order_controller.rb

```
Line 1  require 'activemessaging/processor'
-
-       class OrderController < ApplicationController
-         include ActiveMessaging::MessageSender
5
-         publishes_to :order
-
-         def add
-           order = Order.new(params[:order])
10          if request.post? and order.save
-             flash.now[:notice] = 'Order has been submitted.'
-             publish :order, order.to_xml
-             redirect_to :action => 'show_status', :id => order.id
-           end
15        end
-
-         def show_status
-           @order = Order.find(params[:id])
-         end
20      end
```

Our first Rails controller with *ActiveMessaging* support does not differ much from an ordinary controller.

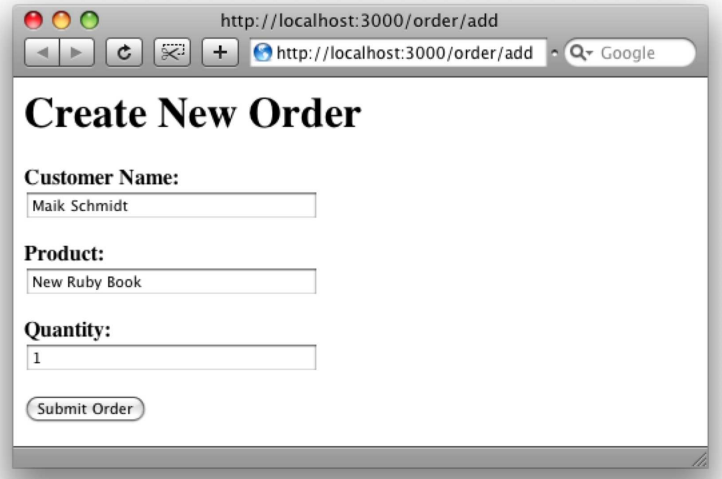

Figure 8.4: Create a new order.

We mix in ActiveMessaging::MessageSender, and in line 6, we tell Rails that this controller will send messages to the order queue we defined earlier in messaging.rb.

The add() method works like an ordinary Rails action; it takes the form parameters from a view, creates a new Order instance, and stores it in the database. Then, in line 12, we use the publish() method to send an XML representation of the newly created order to the order handler.

After the order has been placed, it will have the default status OPEN, as you can see in Figure 8.5, on page 253. This status will not change no matter how often you click the refresh button, because at the moment we do not process the status messages published by the order handler. To change this, we have to add a *processor* to our Rails application. The corresponding generator is part of the *ActiveMessaging* plug-in, and you can run it like this:

```
mschmidt> ruby script/generate processor OrderStatus
```

This creates a skeleton file named order_status_processor.rb that looks as follows after we have added all functionality we need:

Download messaging/activemessaging/msgdemo/app/processors/order_status_processor.rb

```
Line 1  require 'rexml/document'
-
-   class OrderStatusProcessor < ApplicationProcessor
-     subscribes_to :order_status
5
-     def on_message(message)
-       doc = REXML::Document.new(message)
-       order_id = doc.root.attributes['id']
-       order_status = doc.root.text
10      order = Order.find(order_id)
-       order.status = order_status
-       order.save
-       logger.debug "Status of order #{order_id} is #{order_status}."
-     end
15  end
```

Similar to the OrderController, we have to declare that we are using messaging facilities. In line 4, we tell Rails that our OrderStatusProcessor listens for new messages in the :order_status queue. That's all we have to do, because the rest of the messaging mechanism is more or less passive: whenever a new message arrives in the order status queue, the on_message() action gets invoked automatically by *ActiveMessaging*. In the action, we parse the XML document contained in the message, extract its order ID and the order status, and store it in the database. The incoming XML documents are very simple and typically look like this:

```
<order-status id="47110815">SHIPPED</order-status>
```

To be concise, on_message() is not invoked completely automatically, because that would mean the listener is running within the Rails framework itself. To circumvent this, the *ActiveMessaging* developers have created a *poller daemon* that waits for messages and invokes the appropriate Rails actions whenever it receives something new. The poller script is part of the *ActiveMessaging* plug-in, and when you start it like this:

```
mschmidt> ruby script/poller run
```

you'll see the following in your application's log file:

```
ActiveMessaging: Loading ... app/processors/application.rb
ActiveMessaging: Loading ... app/processors/order_status_processor.rb
=> Subscribing to /queue/orders.status (processed by \
   OrderStatusProcessor)
```

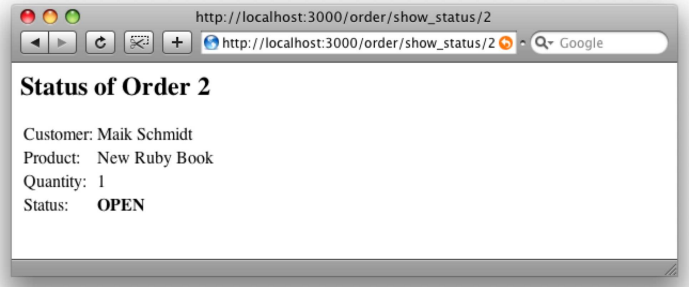

Figure 8.5: The order has been submitted.

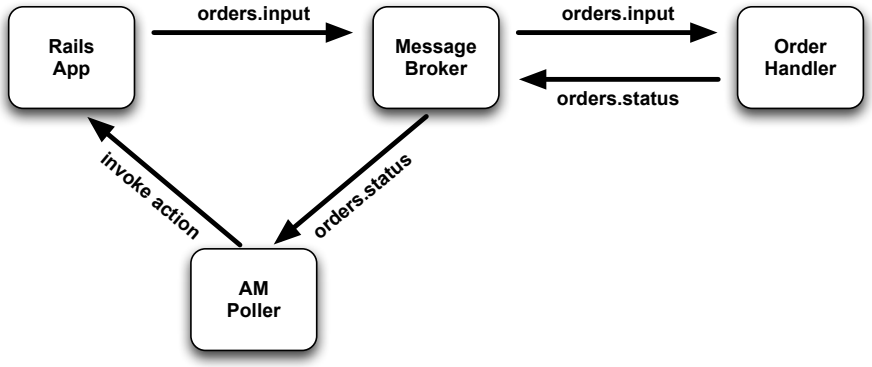

Figure 8.6: System design

For a more detailed view of the architecture we have developed in this recipe so far, see Figure 8.6. The Rails application puts messages into a queue named orders.input, which is managed by the ActiveMQ message broker. The broker passes the message to the order handler, which actually processes the order. When the order has been processed, the order handler sends the result to another message queue named orders.status, which is also managed by ActiveMQ. Afterward, the status message is transmitted to the poller daemon, and the daemon turns it into a call to the right on_message() action.

Only one component of the overall architecture is missing in our test environment: the order handler. Perhaps we could use a copy of the production system, but for testing purposes it's always better to have your own simulator at hand:

Download messaging/activemessaging/order_handler.rb

```
Line 1  require 'stomp'
        require 'rexml/document'
        
        class OrderHandler
          attr_accessor :user, :password, :host, :port
        
          def initialize
            @user, @password = '', ''
            @host, @port = 'localhost', 61613
          end
        
          def handle_orders(in_queue, out_queue)
            connection = Stomp::Connection.open @user, @password, @host, @port
            connection.subscribe in_queue, { :ack => 'client' }
            puts "Waiting for messages in #{in_queue}."
            while true
              message = connection.receive
              body = message.body
              message_id = message.headers['message-id']
              puts "Got a message: #{body} (#{message_id})"
              order_status = get_order_status(body)
              options = { 'persistent' => 'false' }
              connection.send out_queue, order_status, options
              connection.ack message_id
            end
            connection.disconnect
          end
        
          private
        
          def get_order_status(body)
            doc = REXML::Document.new(body)
            order_id = doc.root.attributes['id']
            "<order-status id='#{order_id}'>SHIPPED</order-status>"
          end
        end
```

Our OrderHandler's complete business logic can be found in the handle_orders() method. Basically, it takes order documents from an input queue, parses them, and creates output documents that have the same order ID and a constant status (SHIPPED). That might not be very sophisticated, but for testing the other components it's good not to have too many variable parts.

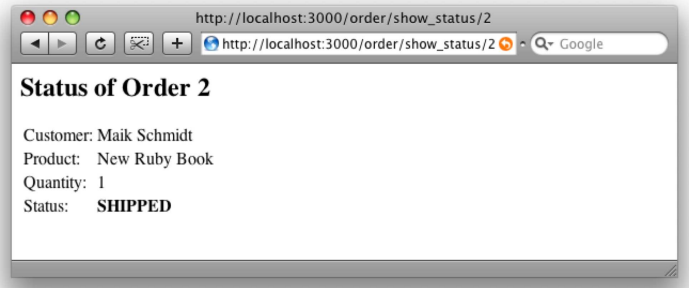

Figure 8.7: The order has been shipped.

As usual, we start a STOMP connection, subscribe to a destination, and start an event loop. This time we chose to use the client acknowledge mechanism in line 14; in other words, we have to explicitly acknowledge incoming messages in line 24. Otherwise, the message would be delivered again by the message broker.

After you have started the order handler like this:

Download messaging/activemessaging/order_handler.rb

```
order_handler = OrderHandler.new
order_handler.handle_orders(
  '/queue/orders.input',
  '/queue/orders.status'
)
```

you can refresh your browser window a few times and eventually see a picture similar to Figure 8.7.

We already knew that messaging with Ruby is easy, but *ActiveMessaging* makes it even more comfortable. Using only a minimal set of configuration parameters and three methods (publishes_to(), subscribes_to(), and publish()), we've been able to combine an existing messaging architecture and a Rails application in record time.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Enterprise Recipes with Ruby and Rail's Home Page
http://pragprog.com//titles/msenr
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com//titles/msenr.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com