MQ Light on Rails

SHead
Published on 14/08/2015 / *Updated on 18/08/2015*

In this article I will discuss how to connect a web application developed and running on Mac OS to an MQ queue manager. Ruby on Rails will be used to develop the web application and the AMQP beta for MQ will be used to allow connections from AMQP clients. MQ Light will provide the AMQP client required to connect the system.

The sample Rails project can be downloaded from GitHub.

Ruby on Rails

Ruby on Rails is a model-view-controller (MVC) framework designed to allow developers to create websites rapidly. Rails is designed to create websites using software engineering principles such as “don’t repeat yourself” (DRY) and “convention over configuration” (CoC). Rails is usually deployed with a database operating under it, providing access and storage of model objects created within the framework.

In a standard MVC applications, Models define objects used within the application, Controllers handle requests and process information, and Views display content created within the application.

By using Ruby as the request handler, Rails provides an object-oriented programming language to implement web services.

AMQP tech preview

In February, it was announced that AMQP support would be added to MQ v8.0.0.3. This will allow MQ Light applications connect to MQ networks and applications within that network.

Scenario

Now you know about the technologies involved, let’s see where this might be useful. If you are creating a blogging website, you may want to be notified when a blog gets posted about a specific topic. By setting up a message sender when new blogs are posted and a receiver subscribed to the topic, you would be notified when there are new posts.

In this article, I will demonstrate how to set up a message sender on a topic with MQ Light. A subscriber could be set up separately using one of the various MQ Light clients. Rails is designed to quickly create web pages with static content so embedding a receiver to listen on a topic would require languages outside the scope of this article.

Setting up AMQP on MQ

For the remainder of this article, MQ will be used as the queue manager. However, MQ Light could also be used to develop the system if MQ is not available. Where MQ running on a separate machine is discussed you can replace the details with address *amqp://localhost* and *port 5672* if you are using a local installation of MQ Light.

A queue manager needs to be created on a platform supported by the AMQP beta. The instructions below demonstrate how to get and AMQP channel running on Linux.

```
ctrmqm AMQP_RAILS_SAMPLE
strmqm -e CMDLEVEL=801 AMQP_RAILS_SAMPLE
strmqm AMQP_RAILS_SAMPLE
setmqaut -m AMQP_RAILS_SAMPLE -t qmgr -p nobody -all +connect
setmqaut -m AMQP_RAILS_SAMPLE -t topic -n SYSTEM.BASE.TOPIC -p nobody -all +pub
+sub
runmqsc AMQP_RAILS_SAMPLE

START SERVICE(SYSTEM.AMQP.SERVICE)
DEFINE CHANNEL(AMQP_RAILS_CHANNEL) CHLTYPE(AMQP) PORT(5672) MCAUSER('nobody')
START CHANNEL(AMQP_RAILS_CHANNEL)
```

Now you've got an MQ queue manager running and you're ready to start developing your Rails application.

Getting a server running

The first step is to ensure you have everything installed correctly. In order to develop in Rails on Mac OS you will need Ruby 1.9.3 or higher, sqlite3 and MQ Light Ruby client.

In order to create your new application and get the server running you will need to open a new terminal to work in. Navigate to the directory you want the application stored in and enter the following commands:

```
rails new mqapp
cd mqapp
rails server
```

This will first build a new package with everything necessary to run a Rails server then start the server. Once this is complete you will be able to access the server on http://localhost:3000. To start with there will be no content and a default page is displayed. You will now need to put in the front page of website. To kill the server at any time issue ctrl+c to kill the server process running in the terminal.

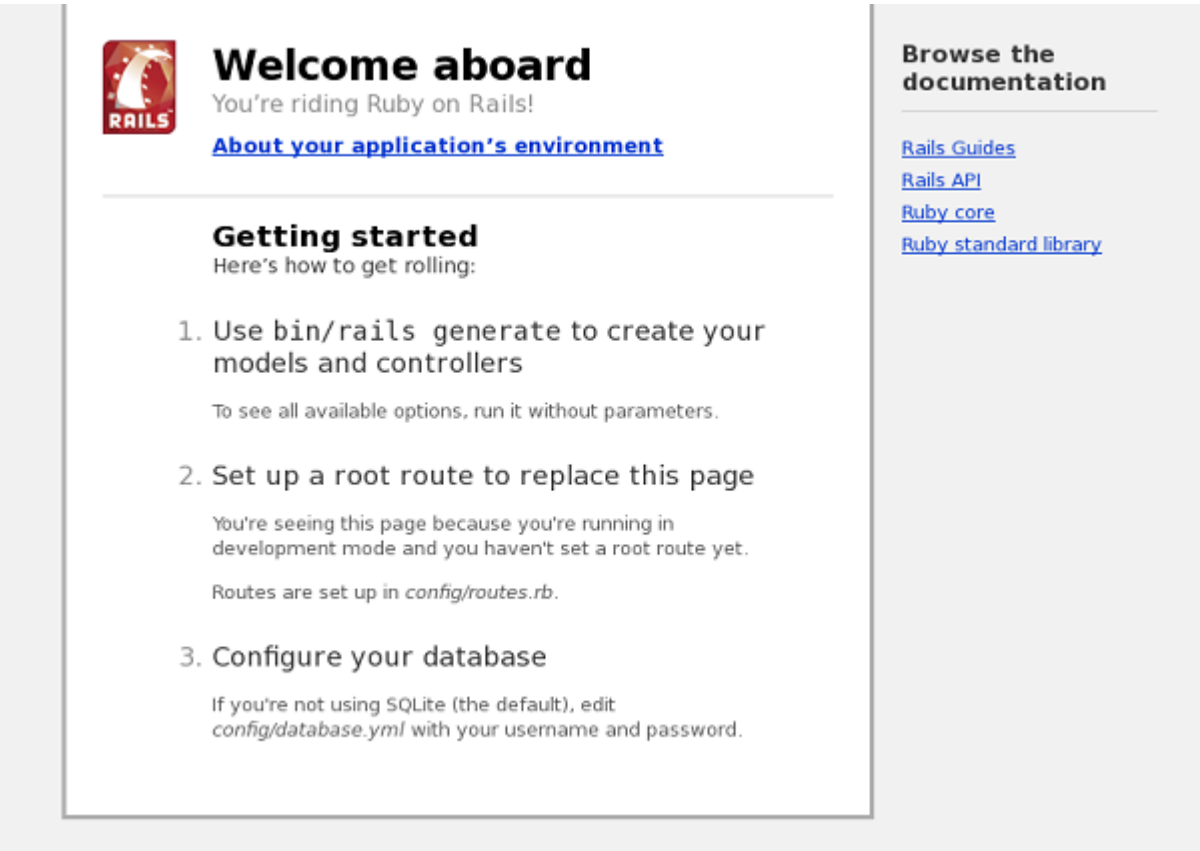
Following the MVC framework, you will need to create a new controller to handle requests to the home page and a view to display the front page content. Open a new terminal and navigate to the mqapp directory.

```
rails generate controller welcome index
```

Now the necessary files have been created for the front page. The next step is to tell Rails to load the welcome index page. In the mqapp directory, open config/routes.rb. This is the main routing file for the application and will specify how to connect requests to controllers. As Rails encourages

CoC, most routes will be handled automatically by specifying the name of a controller and the action required. Route are only manually specified when the name of the controller is not being used as the request name.

The comments in routes.rb provide examples of various routes. The one you are interested in is root *'welcome#index'*. Uncomment this line and reload the webpage. You will now be redirected to the generic Rails index page.



Looking in the app directory in *mqapp*, you will see there is are multiple folders including controllers and views. These are the two main folders that are needed for development.

Let's start by personalising the home page. Open *view/welcome/index.html.erb*. Clear the current content and replace it with:

```
<h1>MQ Light on Rails</h1>
<h2>Publish message</h2>
```

Save this file and the new content will be there when you reload the page. The second line is a placeholder for the moment. You now have a work Rails application running. The next step is to add a message sender.

Building a publisher

A new controller is needed to handle the message requests.

```
rails generate controller publishers
```

You will need to inform the application that there are new routes to be added. In *mqapp/config/routes.rb* add the publishers resource:

```
Rails.application.routes.draw do
  get 'welcome/index'

  resources :publishers
…
```

Now that the resource has been added, a view is needed to allow the user to enter a message and connection details. Navigate to views/publishers and create a new file *new.html.erb*. This file will be loaded when http://localhost:3000/publishers/new is called. By convention, Rails will look for a view file called new as that is the action being requested. Sticking to this convention means you do not need to define routes manually.

The *new.html.erb* will contain a form that allows users to enter a message with connection details and publish it to a specified topic. The user will need to be able to enter the queue manager address, port, topic and message. Rails provides a form builder to quickly add forms to web pages.

```
<h1>Publish message</h1>
<%= form_for :publisher, url: publishers_path do |f| %>

    <%= f.label :address, "IP Address"%>

    <%= f.text_field :address %>


    <%= f.label :port, "Port"%>

    <%= f.text_field :port %>


    <%= f.label :topic, "Publication Topic"%>

    <%= f.text_field :topic %>


    <%= f.label :content, "Message Text" %>

    <%= f.text_area :content %>


    <%= f.submit "Publish" %>

<% end %>
```

When the form is submitted, the create function will need to be called in the controller so *url: publishers_path* is added at the top of the form. You can see how this path is defined by running rake routes in the terminal. This displays the paths and actions that are currently defined within the application.

The view has now been created for publishing messages. Next you need to handle the create request in the publishers controller. Open *controllers/publishers_controller.rb*. The first step is to add the create action to handle sending messages.

```
class PublishersController < ApplicationController
  def create
  end
end
```

You can now see the form added to the view at */publishers/new*. However, if you submit the form you will receive a template error. This is because you have defined the create action but have not yet implemented the action. A simple display of the contents of the form can be seen by adding the following to the create action defined earlier in the publishers controller.

```
render plain: params[:publisher].inspect
```

Now let's do something more useful than just displaying the content of the form we've just submitted. Using the information entered into the form and the MQ Light Ruby sample client you can publish the message to the desired topic.

```
require 'mqlight'

class PublishersController < ApplicationController
  def create
    address = params[:publisher][:address]
    port = params[:publisher][:port]
    topic = params[:publisher][:topic]
    content = params[:publisher][:content]

    amqpservice = 'amqp://' + address + ':' + port

    client = Mqlight::BlockingClient.new(amqpservice)
    client.send(topic, content)

    render plain: "Published"
  end
```

Add the address of your queue manager, if this is with running on a separate machine you will need the IP address from that machine. Earlier we defined a channel listening on port 5672 so enter that into the form. Now submit the form again and a message should be displayed if the queue manager is running and the details were correct.

Error handling

If the queue manager is not running or the client is unable to publish the message an exception will be thrown and the application will break. The next step is to add some error handling to catch these exceptions.

```
    …
    amqpservice = 'amqp://' + address + ':' + port

    result = "Message published"
    begin
      client = Mqlight::BlockingClient.new(amqpservice)
      client.send(topic, content)
      rescue Exception => e
```

```
        result = e.message
    end

    render plain: result
  end
```

Now exceptions will be caught and returned to the user. The current display shows message result so we need to redirect the user. The follow code demonstrates how to return the user to the publish message page, including the result of the publication.

```
class PublishersController < ApplicationController
  @@result = ""

  helper_method :getResult
  helper_method :setResult

  def getResult
    @@result
  end

  def setResult
    @@result = ""
  end

  def create
    address = params[:publisher][:address]
    port = params[:publisher][:port]
    topic = params[:publisher][:topic]
    content = params[:publisher][:content]

    amqpservice = 'amqp://' + address + ':' + port

    @@result = "Message published"
    begin
      client = Mqlight::BlockingClient.new(amqpservice)
      client.send(topic, content)

      rescue Exception => e
        @@result = e.message
    end

    redirect_to "/publishers/new"
  end
end
```

A number of changes have been made to the publisher controller. The result variable has been changed into a class variable so it can be used within the view. Helper methods have been added to access it. A redirect has been added to return the user to publish message page.

The view now needs to be updated to display the message returned from the MQ Light client. The following changes are made in */views/publishers/new.html.erb*.

```
<h1>Publish message</h1>
<%= getResult %>
<%= setResult %>
<%= form_for :publisher, url: publishers_path do |f| %>
```

Finishing the application

The final steps are to add links from the welcome page to the publisher. The *index.html.erb* file needs to be updated with the following:

```
<h1>MQ Light on Rails</h1>
<h2><%= link_to 'Message Publisher', new_publisher_path %></h2>
```

You now have an MQ Light running on Rails publishing messages at a MQ queue manager running on a separate system (or maybe an MQ Light installation running locally).