A 10-minute Guide to Creating your own Mirth-Rest-Adapter

If you don't have rvm, install it and ruby 1.9.2 using:

```
$  \curl -L https://get.rvm.io | bash -s stable --rails --autolibs=enabled –ruby=1.9.2
```

Next create a default gemset:

```
$ rvm use 1.9.2@hquery --create -default
```

Should you ever want to remove rvm and your entire ruby/rubygems collection, use:

```
$rvm implode
```

Now, install Rails with

```
$ gem install rails -v 3.2.5
```

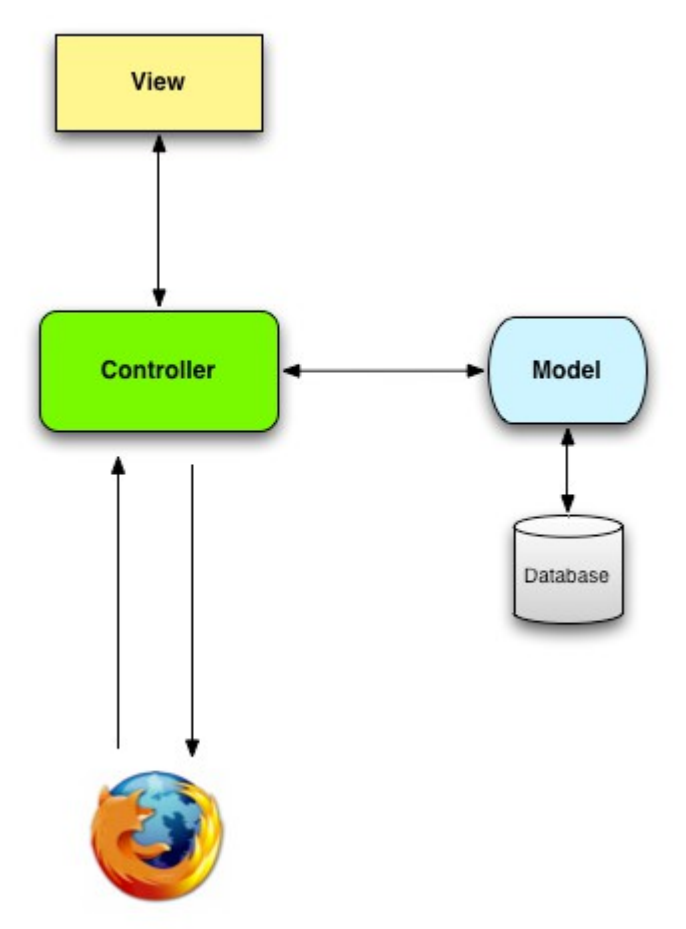
Next create a Rails project

```
$ rails new mirth-rest-adapter
$ cd mirth-rest-adapter
$ ls
app     config.ru  doc      Gemfile.lock  log     Rakefile     script  tmp
config  db         Gemfile  lib           public  README.rdoc  test    vendor
```

(Optional) Create a markdown file named README.md with content specific to this project. Also, put the project under source control management:

```
$ vi README.md
$ git init                    # edit .gitignore; suggest adding .DS_Store
$ git add .
$ git commit -m”Good start”
```

Next poke around in the app directory. Notice the models, views and controllers subdirectories. The Rails framework autogenerates an MVC scaffold.



The scaffolding generated by Rails is enough to provide basic web services.

$ rails server

Open http://localhost:3000 to investigate the default page. Follow the “Getting started” instructions presented there.

First generate the models and controllers. We only need to relay records from Mirth Connect to the query-gateway so no models are needed but we need to add a controller:

$ rails generate controller Records relay

Next we set a default route and (optionally) remove public/index.html. The routes are set up in config/routes.rb. All we need to do is change 'get relay' → 'post relay'.

Because we are merely relaying files, we do not need to concern ourself with database creation. However, we need to add some business logic to /app/controllers/records_controller.rb.

We will need to do a multipart-post to relay the XML files on to the hQuery query-gateway server. Nick Sieger provides a Ruby gem for doing this. See his usage example at https://github.com/nicksieger/multipart-post/blob/master/README.txt

First install the gem:

```
$ gem install multipart-post
```

Next cut-and-paste Nick's usage example into /app/controllers/records_controller.rb

```
class RecordsController < ApplicationController
  require 'net/http/post/multipart'

  def relay

    #store incoming content into a temporary file

    #then we load temporary file and relay it on
    url = URI.parse('http://localhost:3001/records/create')
    File.open("./tmp_record.xml") do |xml_file|
      req = Net::HTTP::Post::Multipart.new url.path,
        "file" => UploadIO.new(xml_file, "text/xml", "tmp_record.xml")
      res = Net::HTTP.start(url.host, url.port) do |http|
        http.request(req)
      end
    end
  end
end
```

At this point we need to know how to grab content from the HTTP request object within the Rails environment.

The hQuery query-gateway's app/controllers/records_controller.rb is an obvious place to look. It has:

```
xml_file = params[:content].read
```

So we will need to add something like this to app/controllers/records_controller.rb:

```
file = File.new('tmp_records.xml', 'w+')
file.write(params[:content].read)
file.close()
```

Will also need to tell Mirth Connect what happened. Again, looking at the query-gateway controller and query-composer/lib/gateway_utils.rb we see that we will need to add something like:

```
render :text => res.message , :status => res.code
```

At this point we are nearly done. For privacy reasons we need to remove the temporary file with:

```
File.delete('tmp_records.xml')
```

So let's compare our new Mirth-Rest-Adapter with the version on Github:

```
$ mkdir scoophealth
$ cd scoophealth
$ git clone git@github.com:scoophealth/mirth-rest-adapter.git
$ cd ../
$ diffmerge scoophealth/mirth-rest-adapter mirth-rest-adapter
```

We will conclude by examining the business logic in query-gateway (quite simple) as well as the business logic in health-data-standards (more involved).

But first Ruby Off Rails...

```
#!/usr/bin/env ruby
# A simple WEBrick web server
#
require 'webrick'
require 'net/http/post/multipart'

include WEBrick # import WEBrick namespace

config={}
config.update(:Port => 3000)
config.update(:DocumentRoot => './')
server = HTTPServer.new(config)

# Mount servlets
server.mount_proc('/') { |req, resp|
  resp.body = '<a href="/records/destroy">Delete</a> test patient records<br><a href="/records/relay">Create</a> test patient records'
}
server.mount_proc('/records/destroy') { |req, resp|
  uri = URI.parse("http://localhost:3001/records/destroy")
  http = Net::HTTP.new(uri.host, uri.port)
  request = Net::HTTP::Delete.new(uri.request_uri)
  response = http.request(request)
  resp.body = response.body
}
class RecordRelayServlet < HTTPServlet::AbstractServlet
  def do_GET(request, response)
    Dir.glob('/vagrant/files/*.xml') do |xml_file|
      url = URI.parse('http://localhost:3001/records/create')
      res = nil
      File.open(xml_file) do |xml|
        req = Net::HTTP::Post::Multipart.new url.path, "content" => UploadIO.new(xml, "text/xml", "temp_scoop_document.xml")
        res = Net::HTTP.start(url.host, url.port) do |http|
          http.request(req)
        end
      end
      response.body = res.body
    end
    raise HTTPStatus::OK
  end
  alias :do_POST :do_GET # accept POST request
end
server.mount('/records/relay', RecordRelayServlet)

# Trap signals to shutdown cleanly.
['INT', 'TERM'].each do |signal|
  trap(signal) {server.shutdown}
end

# Start the server
server.start
```

Future talks?

Rails Testing

There are many available technologies to choose from:

Hypervolemia on the Tracks

- Test-driven development starting with the official guide at http://guides.rubyonrails.org/testing.html and then moving on to the more exotic?
  - more bottom-up than BDD
  - maybe better for testing controllers and models
  - already part of Rails and used by hQuery
- Behaviour-driven testing with Rspec, Cucumber, etc. (Rails 3 in Action, railstutorial.org, etc.
  - more top-down than TDD
  - maybe better for views
  - Rspec uses a domain-specific language to present user stories
    - Spork is a helper utility that keeps the rails environment in memory so that there is less load time when a test is ran.
    - Guard is another helper which limits the testing to just those tests associated with the code that is being changed.

Rails IDEs – for the most part seem a bit crude. The main site at http://rubyonrails.org/ecosystem recommends VIM for Rails and Emacs for Rails as well as TextMate on OS X. Sublime 2 is an option. RubyMine looks interesting.