RAGI: Ruby Asterisk Gateway Interface

"Delivery Phone" Tutorial

Revision:
January 19, 2006

snapvine
WWW.SNAPVINE.COM

Overview

This tutorial illustrates how to use RAGI to build interactive web and phone services with Asterisk and Ruby on Rails.

About RAGI

Ruby Asterisk Gateway Interface (RAGI) is a useful open-source framework for bridging the Ruby on Rails web application server environment and Asterisk, the open-source PBX.

RAGI eases the development of interactive automated telephony applications such as IVR, call routing, automated call center, voice mail systems and extended IP-PBX functionality by leveraging the productivity of the Ruby on Rails framework. RAGI simplifies the process of creating rich telephony and web apps with a single common web application framework, object model and database backend.

RAGI was created by SnapVine, Inc and is available under the BSD license.

For more information on RAGI, including forums, demos and tutorials, please see:
http://www.snapvine.com/code/ragi

The Tutorial Application

The application we'll be creating in this tutorial is called "Delivery Phone", and is a simple but useful service for monitoring UPS delivery status. The user experience includes a web site where users can sign in and register packages and a phone interface for listen to tracking status by dialing in over a phone.

The system is built using the following components:

- **Asterisk** – the open source PBX which is used to interface to VOIP/PSTN provider over SIP to place real telephone calls
- **Ruby on Rails** – the super-productive web application framework built in the Ruby programming language
- **MySQL** – database is used to store the records associated with the application data model. All database access is handles by Ruby on Rails.
- **RAGI** – Ruby Asterisk Gateway Interface, an API for building Asterisk telephony applications in Ruby (and Ruby on Rails).

Things You Should Already Know

Although the tutorial will step you through everything needed to create the app, some things are not emphasized or explained in much detail. It is assumed that the reader is already familiar with:

- Asterisk
  - How to install
  - How to setup
  - Basics of what AGI is
- Ruby on Rails
  - How to install
  - Rails app structure (controllers, views, models, etc)
- MySQL
  - How to install
  - Basic SQL operations and queries

To Learn More

For more information on any of these technologies, please refer to the following resources.

- Asterisk
  - http://www.voip-info.org/wiki-Asterisk
  - http://www.asterisk.org
- Ruby on Rails
  - http://www.rubyonrails.org
  - http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html
- RAGI
  - http://www.snapvine.com/code/ragi

"Delivery Phone" Tutorial

The user experience of our tutorial application is very simple:

- A user accesses the website via a web browser and creates an account by entering their registration phone number and 1 or more UPS package tracking numbers.
- The user dials a phone number to listen to the delivery status of their packages. The system should attempt to authenticate the user by looking at their caller ID.

Design

In our system there are two object types, a "user" and a "delivery". User's are registered by their phone number. Each user may have many deliveries they are tracking.

USERS:
id, phone_number, updated_on, created_on

DELIVERIES:
id, user_id, tracking_number, status, updated_on, created_on

Creating the Web UI in Ruby on Rails

Step 1. Create your Rails directory "ragi_tutorial" and generate rails app "tutorial" inside of it.

```
mkdir c:\ragi_tutorial
cd c:\ragi_tutorial
rails tutorial
```

Step 2. Create the database and tables for our application. I recommend using "My SQL Front" for this. We're calling our database "tutorial".

```
CREATE TABLE `deliveries` (
  `id` int(11) NOT NULL auto_increment,
  `user_id` int(11) NOT NULL default '0',
  `tracking_number` varchar(64) NOT NULL default '',
  `delivery_status` varchar(255) default NULL,
  `updated_on` datetime default NULL,
  `created_on` datetime default NULL,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL auto_increment,
  `phone_number` varchar(10) NOT NULL default '',
  `updated_on` datetime default NULL,
  `created_on` datetime default NULL,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Step 3. Edit the database.yml to tell Rails where this database is located

```
notepad C:\ragi_tutorial\tutorial\config\database.yml
```

For example,

```
development:
  adapter: mysql
  database: tutorial
  socket: /path/to/your/mysql.sock
  username: root
  password:
```

Step 4. Next, use the Rails scripts to generate "model" and "controller" classes for the users object. From the directory "C:\ragi_tutorial\tutorial", run:

```
ruby script\generate model User
ruby script\generate model Delivery
ruby script\generate controller User
```

Step 5. Add the scaffold methods (CRUD) to the UserController class

```
class UserController < ApplicationController
  scaffold :user
end
```

Step 6. Start the server to look at the "skeleton" app:

```
ruby script\server
```

Check out the list of users (currently empty):

```
http://localhost:3000/user/list
```

Create a new user:

```
http://localhost:3000/user/new
```

Step 7. Each user may have multiple "deliveries". The database model we created already permits that relationship. To let Rails know about it, we add the following line to user.rb:

```
  has_many :deliveries
```

Now each user object will have a ".deliveries" method to retrieve the list of Delivery objects associated with that user instance.

Step 8. Now that our database is "alive" thanks to Rails, we can create a basic website to fulfill our website requirements, if you recall:

> *A user accesses the website via a web browser and enters their registration phone number and 1 or more UPS package tracking numbers.*

Okay, here's our "detailed engineering spec" for that:

- Off the root of the website, you enter your phone number.
- That brings up a list of your "packages", if any.
- You can add and delete packages to track.
- There is a "logout" button.

Note: It is left as an exercise for the reader to actually code up the web parts of this app. You'll find a finished implementation of this web UI included with the tutorial materials.

Now, on to the telephony part! First we going to install and configure RAGI, then we'll code up the call handler IVR aspect of the application.

Installing and Configuring RAGI

Step 9. Install RAGI. This is easy, because RAGI is packaged as a "gem". Just type the following from your command prompt:

```
gem install ragi
```

Step 10. Next, we'll configure RAGI to work with Rails and Asterisk. Most of this is a one-time setup. For this tutorial, don't worry if you don't understand all the confirmation steps. Later, you can read the full documentation, "Configuring RAGI for Rails and Asterisk".

First, edit your environment.rb in the config directory to tell Rails about RAGI. Technically speaking, RAGI is both an API and a server. Asterisk sends calls from your linux server to your Rails server and talks to RAGI on a special port. This code in environment.rb will make sure a RAGI server is running when your app server boots up.

```
# Include your application configuration below


# The following code tells Rails to start a Ragi server
# as a separate thread.

Dependencies.mechanism = :require

# Simple server that spawns a new thread for the server
class SimpleThreadServer < WEBrick::SimpleServer
  def SimpleThreadServer.start(&block)
```

```
    Thread.new do block.call
    end
  end
end

require 'ragi/call_server'

RAGI::CallServer.new(
:ServerType => SimpleThreadServer )
```

Step 11. Next, we turn our attention to Asterisk for a minute. In this step, we'll tell Asterisk where are RAGI server is so that the appropriate calls can be routed into our new application.

Specifically, we'll set up an extension "998" that when dialed, connects to our UPS Delivery Tracking application.

Edit the Asterisk extensions.conf as follows are reload or restart Asterisk:

```
RAGI_SERVER = yourservername

exten => 998,1,Answer()
exten => 998,2,deadagi(agi://${RAGI_SERVER}/tutorial/dialup)
exten => 998,3,Hangup
```

You can now easily configure Asterisk to map outside lines to directly patch through to extension 998 in order to allow direct dial access to your application.

Step 12. A finally setup step that you may want to do is to set up a softphone (PC SIP Phone) to dial into your Asterisk server to extension 998 so that you can easily test your new phone line while it is in development. Here's how:

There are many softphones that will work, but the one we're including instructions for is Firefly (https://www.virbiage.com/download.php)

1. Modify your iax.conf to set up an account for a new user.
2. Install and launch the "firefly" softphone client.
3. Set up a new network from the "options" menu using IAX protocol for the user and password you set up in step 2.
4. Dial extension 998 from the softphone. Your default handler running on your Rails server will answer the call.

Building your Dial-in Phone Interface

Step 13. Create a directory called "handlers" in your Rails app directory at the same level in your directory structure as controllers, models and views. This is where you will put the class that implements

the call logic. We call these “RAGI Call Handlers”. A Rails controller is to a web page as a RAGI Call Handler is to a phone call.

Step 14. Create a new file called “delivery_status_handler.rb” and open that file to edit.

Step 15. Type or paste the following into your new file:

```
require 'ragi/call_handler'

class DeliveryStatusHandler < RAGI::CallHandler
  def dialup
    say_digits('12345')
  end
end
```

This is basically your “hello world” call handler. The class is “DeliveryStatusHandler” and is an implementation of the RAGI “CallHandler” class. The method “dialup” is automatically called when a phone call is routed by Asterisk through RAGI to this handler.

The API you can use to operate on the call (play sounds, take input, etc) is documented in call_connection.rb in the gem (located in your ruby directory, e.g. C:\ruby\lib\ruby\gems\1.8\gems\ragi-1.0.1\ragi)

If you have set up firefly as per step 12, you should now be able to run your Rails server and then dial 998 to try things out now.

Go head, try it!! You should hear “one two three four five” before being hung up on.

Step 16. Okay, now you are ready to implement your call logic! All of the objects you created in your Rails application are available to your new call handler. To give this a demonstration, try replacing `say_digits("12345")` with the following:

```
say_digits(User.count.to_s)
```

Now when you dial up to the line, you’ll hear the number of users you have in the Users table in your database.

The nice thing about using Rails with Asterisk in this way is that you can program super-productive web sites and have access to all of the objects and methods you create directly when creating your phone user interfaces!

Step 17. Let’s familiarize with the other RAGI API for controlling Asterisk. One of the most important API is get_data, which plays a sound and listens for user key presses.

```
keypresses = get_data("beep", 5000, 3)
say_digits(keypresses)
```

The parameters are sound file, milliseconds to wait for user input, and maximum number of key presses to listen for.

By default, Asterisk stores sounds in "/var/lib/asterisk/sounds". That's also the root where RAGI will play sounds from. One of the sounds that should be in there is "beep.gsm".

Please create a sound file called "ragi_tutorial_please.gsm" in this directory with the sound "Please enter your phone number"[1].

Now when you dial up the call handler you'll here a beep sound, then you can press buttons on the phone and you'll hear what you pressed read back to you. With just a few RAGI API such as get_data and playSound, you'll be able to create all kinds of IVRs easily.

Step 18. With our application, the IVR logic is fairly simple. The user dials up, the callerID is read from the connection and used to authenticate. Text to speech is used to read the status of each delivery object associated with the phone number.

First, pull the caller ID out.

```
    def dialup
      answer()

      # read the caller id from the connection
      user_phone_number = @params[RAGI::CALLERID]
```

If caller ID is not available (for example, if you are calling from a softphone), you'll get back "0" as the caller ID. Check for that and ask the user to manually enter their phone number.

```
      if (user_phone_number == "0")
        user_phone_number = getPhoneNumber()
      end
```

The method "getPhoneNumber" is a new method we'll add to the delivery_status_handler class:

```
    def getPhoneNumber()
      #say "please enter your phone number"
      phonenumber = get_data("ragi_tutorial_please", 5000, 10)
      return phonenumber
    end
```

Now that we have a phone number, look up the user instance associated with the phone number and hang up if none is found.

```
      user = User.find_by_phone_number(user_phone_number)
```

[1] We recommend Audacity for recording audio and sox to convert the sound to gsm format. Both of these programs are open source and are available on Windows and Linux.

```
    if (user == nil)
      speak_text("We could not locate any accounts.  Please try again later")
      hangUp()
      return
    end
```

Finally, get the array of delivery objects associated with this user, update their status and read it back to the user:

```
    deliverylist = user.deliveries

    speak_text("You have #{deliverylist.size} deliveries to track")

    if deliverylist.size > 0
      updateAll(deliverylist)
      readDeliveryStatus(deliverylist)
    else
      speak_text("Please go to the website and register some deliveries.")
    end

    speak_text("Goodbye!")
    hangUp()
  end
```

The methods “updateAll” and “readDeliveryStatus” are new methods to be added to the handler class. Here they are:

```
  def updateAll(deliverylist)
    speak_text("Now updating information")
    deliverylist.each do |delivery|
      delivery.updateStatus
    end
  end

  def readDeliveryStatus(deliverylist)
    i = 0
    deliverylist.size.times do
      delivery = deliverylist[i]
      speak_text("delivery number #{i+1}")
      if delivery.delivery_status == nil
        speak_text("The status is unknown")
      else
        speak_text(delivery.delivery_status)
      end
      i = i + 1
    end
  end
```

Step 19. If you try to run the code now, it will not work, because there are two methods we’ll need to add to the Delivery class.

- updateStatus – fetches the delivery status from the UPS website and parses the HTML to extract status information, formats to be human readable and updates that status in the database for this object.
- parseUPSString – helper method which takes a comma-delimited string extracted from the HTML result page and formats a human-readable status text.

We're in luck because web requests to track packages at UPS look like this http://wwwapps.ups.com/WebTracking/processRequest?TypeOfInquiryNumber=T&InquiryNumber1=***tracking-number-here*** and quite miraculously, the resulting HTML page contains a very nice element called "line1" whose value is a comma-delimited set of fields that give us everything we need to interpret the latest delivery status!

```
<INPUT TYPE="HIDDEN" NAME="line1"
VALUE="TrackDetailInfo,1Z4A14T9PP11305735,313190c4a38ebb0000e5f4827602991d9c\
,313190c4a38ebb0000e5f482760299089c\,\,,,D,,20051117,10:58:00,SEATTLE,WA,US,,
FRONT
DOOR,,,,20051114,SEATTLE,WA,US,BASIC,3.50,LBS,1,,,,,tdts,FS,,,,,,,,,,4A14T9,
,,4A14T9,,,21b428be,,,,,D,,,      USGD,20051114,220627,10010">
```

This text in the "value" field is a comma-delimited string which can be easily parsed to extract useful information such as:

- Delivery status code – index 7
- City – index 11
- Time – index 10
- Detailed location – 15

Please see "delivery.rb" in the tutorial code distribution for a completed implementation.

Note: UPS does in fact have a XML web API to retrieve package status information. Using that would make for a much more robust implementation, but for this tutorial "prototype" we decided that the extra steps of getting a developer key and registering with UPS were not worth the trouble and did help illustrate RAGI per say.

Step 20. Congratulations, you're finished!

APPENDIX: Configuring RAGI for Rails and Asterisk

The way RAGI ties together Rails with Asterisk is simple, but the configuration is a bit "detail oriented". The beauty of all this is that once you've set things up, you can write as many different call handlers for outbound and inbound calls as you want, all in Ruby on Rails, and never have to touch your Asterisk server again.

How To Configure Asterisk to Send Calls to RAGI

For inbound calls, RAGI can be used for calls connecting from your Asterisk server to your Ruby on Rails server.

To configure calls to route properly, you'll need to set your Asterisk extensions.conf file for each phone number (or extension) you want handled by your Ruby on Rails server.

Be sure to replace the values as you see fit for your own application (for example, "hermes:4574" should be your Ruby on Rails server name or IP and port running RAGI). In this example, "`/appname/dialup`" is the handler that will be used when (212)555-1212 is called.

```
RAGI_SERVER = hermes:4574

;;; Map the phone number or extension to the app extension "tutorial"
exten => 2125551212,1,Goto(tutorial,entry-point,1)

[tutorial]
exten => entry-point,1,Answer
exten => entry-point,2,Wait(1)
exten => entry-point,3,deadagi(agi://${RAGI_SERVER}/appname/dialup)
exten => entry-point,4,Hangup
```

Notes:

- In the example above, all calls to (212)555-1212 will be routed to the RAGI process running on port 4574 on a server called "hermes". Furthermore, all of these calls will be answered by the *Call Handler* corresponding to "tutorial/dialup" (see part C below)
- The string "tutorial/dialup" corresponds to the URI of your *Call Handler* implementation.
- RAGI can be used for all of your calls or just some – it just depends on how you set up extensions on your Asterisk server.

How To Configure RAGI to Send Calls to Asterisk

RAGI can also be used to place outbound phone calls whose call logic is handled in a RAGI CallHandler. The object provided for this is callInitiate.rb.

It's simple to use; in your applications simple make a method call like this:

```
CallInitiate.place_call("2125551212", "8002001111, "tutorial/dialout", hashData)
```

In the above example, a call would be placed to (212)555-1212, showing as caller ID (800)200-1111, and when answered would be handled by the RAGI Call Handler mapped to “tutorial/dialout”. Furthermore, the contents of the hashtable “hashData” will be available to the callHandler by calling `@params['key']` in your routine.

The callInitiate object works by writing call files to Asterisk’s “outgoing” directory. Asterisk automatically scans for new call files and when it finds any, it places the call and connects it through the extension, before routing back to the RAGI server and appropriate CallHandler.

The one-time setup to enable this is as follows:

1. Add the following to your Asterisk extensions.conf

```
[dialout]
exten => outbound,1,Answer  ; switches to outbound-handler
exten => outbound,2,Wait(60)
exten => outbound,3,Hangup

exten => outbound-handler,1,Dial(${CallInitiate_phonenumber}|50|gM(outbound-
connect^${AGI_SERVER}${AGI_URL}^${CallInitiate_hashdata}^${MACHINE_STATUS_UNKNOWN}))
exten => outbound-handler,2,GotoIf($["${DIALSTATUS}" = "ANSWER"]?104)
exten => outbound-handler,3,NoOp(status=${DIALSTATUS}, DIALEDTIME=${DIALEDTIME},
ANSWEREDTIME=${ANSWEREDTIME})
exten => outbound-handler,4,SetVar(CallInitiate_hashdata=${CallInitiate_hashdata})
exten => outbound-handler,5,deadagi(agi://${AGI_SERVER}${AGI_URL}) ;DIAL_STATUS is busy, etc.
exten => outbound-handler,6,Goto(104)
exten => outbound-handler,102,SetVar(CallInitiate_hashdata=${CallInitiate_hashdata})
exten => outbound-handler,103,deadagi(agi://${AGI_SERVER}${AGI_URL}) ;DIAL_STATUS is busy, etc.
exten => outbound-handler,104,Hangup()

[macro-outbound-connect]
exten => s,1,Answer()
exten => s,2,SetVar(CallInitiate_hashdata=${ARG2})
exten => s,3,SetVar(machinestatus=${ARG3})
exten => s,4,deadagi(agi://${ARG1})
exten => s,5,Hangup
```

2. Make sure the server running the process that uses callInitiate has file write access to the server running your Asterisk process. This means mapping a drive from your Asterisk server to your Rails server and making sure the wakeup directory and outgoing directories are writeable to it.

How to Configure a Softphone for Testing

It is helpful to configure a softphone to use for testing so you don’t have to use real phone to call your handler while in development.

There are many softphones that will work, but the one we’re including instructions for is Firefly (https://www.virbiage.com/download.php)

1. Modify your Asterisk extensions.conf to include a dedicated test extension for your app.

```
exten => 102,1,Answer()
exten => 102,2,deadagi(agi://192.168.2.202)
exten => 102,3,Hangup
```

This will send all calls to extension “102” to the RAGI server running on the server at 192.168.2.202, and the call will be handled by the call handler defined as “:DefaultHandler” in your environment.rb.

2. Modify your iax.conf to set up an account for a new user.

3. Install and launch the “firefly” softphone client.

4. Set up a new network from the “options” menu using IAX protocol for the user and password you set up in step 2.

5. Dial extension 102 from the softphone. Your default handler running on your Rails server will answer the call.