Building an STRSQL utility with PowerRuby

PowerRuby tutorial

Cairns Tony

March 20, 2014

This tutorial creates a Rails version of old green screen Start SQL Interactive Session (STRSQL) utility. Most IBM i professionals use STRSQL on IBM i, but if you have not, simply sign on to your IBM i 5250 and enter STRSQL on the command line. Other products duplicate the STRSQL support as a client/server application, but this Rails version stays completely on the IBM i server and use any device with a browser as the interface.

Introduction

This tutorial creates a Rails version of the old green screen STRSQL utility. Other products duplicate the STRSQL support as a client/server application, but this Rails version stays completely on the IBM i server and can use any device with a browser as interface.

This tutorial uses a new IBM i technology, PowerRuby, to fast path your way through the web-based STRSQL application. If you are unfamiliar with Ruby on Rails, see the following PowerRuby tutorials for instruction on PowerRuby installation and PowerRuby Rails fast path Command Language (CL) utilities:

- PowerRuby tutorial
- iProDeveloper tutorial

What is PowerRuby? PowerRuby is a privately held business that has teamed with IBM to provide a formal port of the Ruby language and the Rails framework including a native database driver that communicates directly with IBM® DB2® on i without the need for any other proxies (such as MySQL, JDBC, ODBC).

PowerRuby is currently available in a free community version. Additional features under development can take the Ruby and Rails environments to the next level and be included in a licensed commercial version of PowerRuby.

Rails web version STRSQL

This tutorial uses the IBM i command line to edit, test, and deploy a step-by-step Rails web version of the IBM i STRSQL application instead of using the PowerRuby CL-based fast Rails helpers (SETPOWRBY, RAILSNEW, and RAILSSVR). This manual approach can provide a better understanding of the Rails plumbing, which should make it easier for you to develop PowerRuby applications.

A Ruby gem is simply a community contribution to the Ruby project. In this case, IBM authored both ibm_db gem for DB2 access and xmlservice gem for IBM i access to native objects (*CMD, *PGM, *SRVPGM, DB2 objects, and so on). You will be creating your web STRSQL using PowerRuby included IBM gem technologies:

- xmlservice gem - enables IBM i call of CMD, PGM, and so on.
- ibm_db gem - enables DB2 for i

Rails commands used:

```
rails new strsql --skip-sprockets --skip-bundle
   bundle install --local
rails generate model
   rake db:migrate
rails generate controller
rails server -p 4242
   Ctrl-C to shutdown server
```

Rails STRSQL step-by-step

This tutorial uses the IBM Portable Application Solutions Environment (PASE) for i. The PASE command line is used to edit, test, and deploy a step-by-step Rails web version of the IBM i STRSQL application. A simplified interface application will be created without using asset pipeline (rails sprockets or CoffeeScript), thereby the tutorial remains focused on IBM i specific topics.

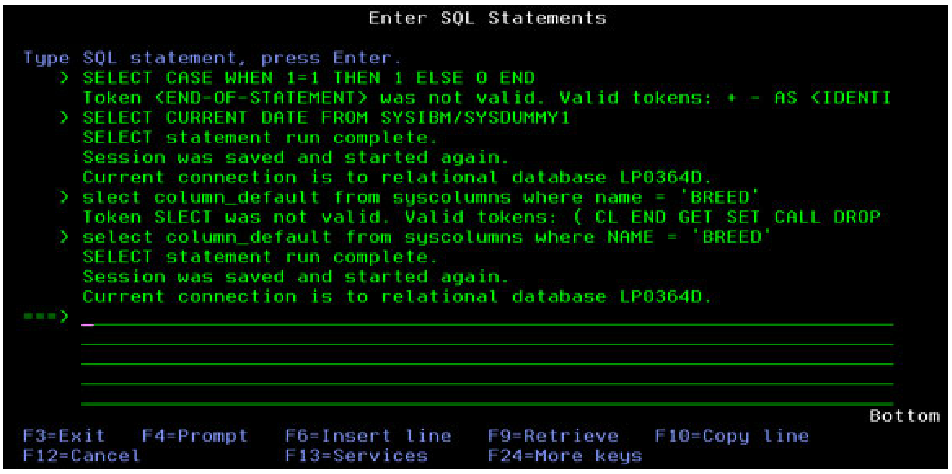
Web GUI designers and Rails asset pipeline development requires a native JavaScript engine to process and package CoffeeScript or JavaScript™ code. As of this writing, IBM i does not provide a native JavaScript engine, and therefore, Rails commands dealing with asset pipeline or sprockets might fail. However, with a simple web search, you can find web pages that explain the concept of precompiling assets, developing JavaScript assets on a notebook and deploying them on an IBM i system.

Start now …

Step 0. Design web STRSQL

A quick tour of the STRSQL interface will define our web STRSQL requirements. We will simplify the interface of web STRSQL to use HTML, but we will keep the basic functions found in the green screen version.

Figure 1. STRSQL requirements match green screen interface



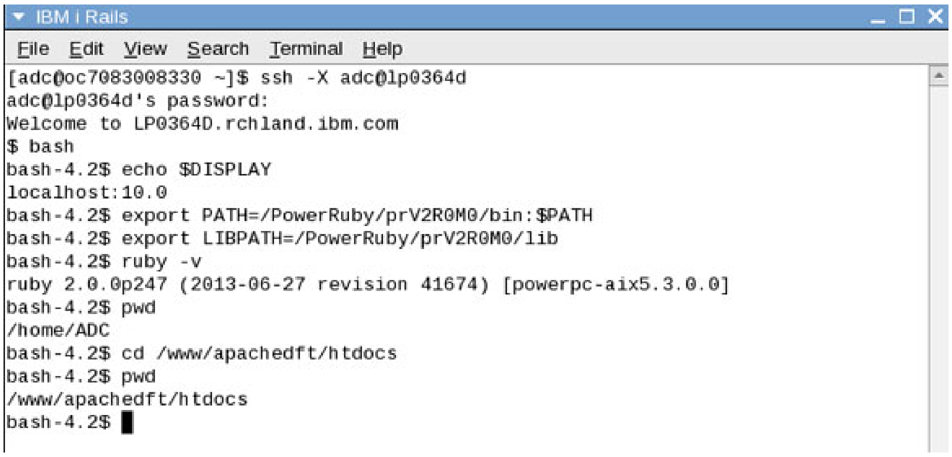
```
                           Enter SQL Statements

 Type SQL statement, press Enter.
    > SELECT CASE WHEN 1=1 THEN 1 ELSE 0 END
      Token <END-OF-STATEMENT> was not valid. Valid tokens: + - AS <IDENTI
    > SELECT CURRENT DATE FROM SYSIBM/SYSDUMMY1
      SELECT statement run complete.
      Session was saved and started again.
      Current connection is to relational database LP0364D.
    > slect column_default from syscolumns where name = 'BREED'
      Token SLECT was not valid. Valid tokens: ( CL END GET SET CALL DROP
    > select column_default from syscolumns where NAME = 'BREED'
      SELECT statement run complete.
      Session was saved and started again.
      Current connection is to relational database LP0364D.
 ===> _

                                                                    Bottom
 F3=Exit    F4=Prompt    F6=Insert line    F9=Retrieve    F10=Copy line
 F12=Cancel              F13=Services      F24=More keys
```

- STRSQL enables SQL statements to create tables, insert data, and delete data.
- STRSQL SQL Select statements also fetch data rows for display.
- STRSQL records user history, allowing recall of previous SQL statements.

Step 1. Command line and editor

As a Linux® user, I prefer a Secure Shell (SSH) terminal connected to my IBM i server (lp0364d). This tutorial is not about SSH. However, note that the `ssh -X` option allows me to run a graphical IBM AIX® editor on IBM i (SSH setup). YiPs site provides a copy of nedit binary on wiki page Fun with QSH, call qp2term and RPG (SHELL_use-1.0.3.zip). Feel free to substitute your favorite remote attached editor, but I am going with native IBM i PASE technology. If you choose not to use the SSH terminal, you can enter Rails commands from 5250 command line as follows: `CALL qp2term`.

Figure 2. Start SSH Terminal - ssh -X me@myibmi



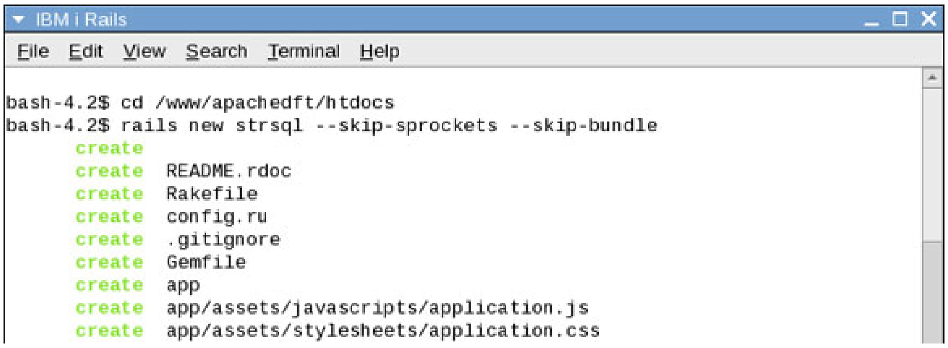
```
[adc@oc7083008330 ~]$ ssh -X adc@lp0364d
adc@lp0364d's password:
Welcome to LP0364D.rchland.ibm.com
$ bash
bash-4.2$ echo $DISPLAY
localhost:10.0
bash-4.2$ export PATH=/PowerRuby/prV2R0M0/bin:$PATH
bash-4.2$ export LIBPATH=/PowerRuby/prV2R0M0/lib
bash-4.2$ ruby -v
ruby 2.0.0p247 (2013-06-27 revision 41674) [powerpc-aix5.3.0.0]
bash-4.2$ pwd
/home/ADC
bash-4.2$ cd /www/apachedft/htdocs
bash-4.2$ pwd
/www/apachedft/htdocs
bash-4.2$ █
```

ssh -X me@myibmi
The SSH terminal session in Figure 2 is launched from the bash shell. The bash shell is the preferred shell for most Linux users – it's available as a binary download from the YiPS website. The two export commands are required to select the PowerRuby version of Ruby on Rails. The working directory is changed to create the application in DocumentRoot.

Step 2. Create Rails application

We need to create our Rails skeleton application. We are using a few options with `rails new` to help IBM i stay within the basic PowerRuby distribution. Executing the PowerRuby Rails utility, `rails new`, in the /www/apachedft/htdocs directory creates a new sub-directory `strsql/` with the full Rails skeleton application sub-directories and "get started" code.

Figure 3. The rails new strsql command



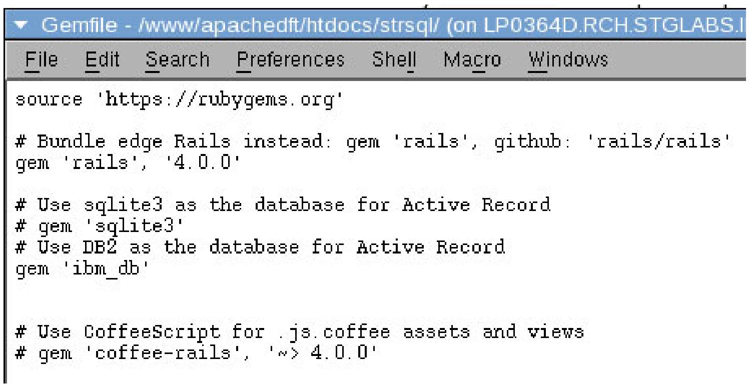
```
bash-4.2$ cd /www/apachedft/htdocs
bash-4.2$ rails new strsql --skip-sprockets --skip-bundle
      create
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      create  Gemfile
      create  app
      create  app/assets/javascripts/application.js
      create  app/assets/stylesheets/application.css
```

As you can see in Figure 3, the rails new `strsql` command invocation specifies the `--skip-sprockets` option to ignore asset pipeline and the `--skip-bundle` option to avoid bundle processing.

Step 3. Edit Gemfile

We wish to use DB2 for i, not sqlite3 (default). So, we need to edit Gemfile created by `rails new` and replace sqlite3 with ibm_db (DB2 for i) as shown in Figure 4. Also, we are not using the many options for Rails asset pipeline (JavaScript, and so on). So, comment out these directives to avoid unwanted processing errors later in the tutorial and to simplify this example.

Figure 4. Edit Gemfile to add ibm_db

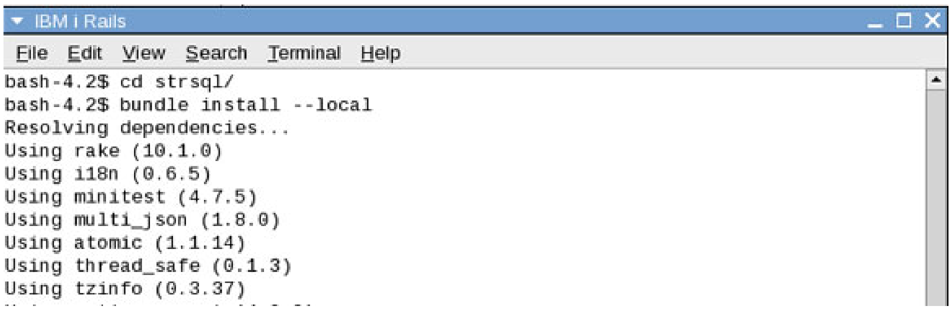

Gemfile - /www/apachedft/htdocs/strsql/ (on LP0364D.RCH.STGLABS.I
File Edit Search Preferences Shell Macro Windows

```
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.0.0'

# Use sqlite3 as the database for Active Record
# gem 'sqlite3'
# Use DB2 as the database for Active Record
gem 'ibm_db'


# Use CoffeeScript for .js.coffee assets and views
# gem 'coffee-rails', '~> 4.0.0'
```

Step 4. Bundle install

We wish to avoid unwanted automatic gem version updates from Rails http://rubygems.org. So, we will use the local option (`--local`) when running the `bundle install` command in the /www/apachedft/htdocs/strsql directory as demonstrated in Figure 5. Examining the Gemfile.lock file after `bundle install`, we see that our application gem dependencies are PowerRuby locked for any deployments. Of course PowerRuby is our IBM i team guardian with a PowerRuby installation, so our Rails applications are likely to run out of box on other PowerRuby systems.

Figure 5. Command bundle install

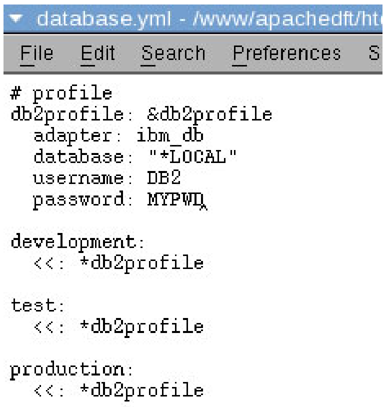


```
bash-4.2$ cd strsql/
bash-4.2$ bundle install --local
Resolving dependencies...
Using rake (10.1.0)
Using i18n (0.6.5)
Using minitest (4.7.5)
Using multi_json (1.8.0)
Using atomic (1.1.14)
Using thread_safe (0.1.3)
Using tzinfo (0.3.37)
```

Step 5. Edit database.yml

We wish to use DB2 for i for our Rails database. So, we must remove all sqlite3 (default), in favor of DB2 for i profile information as shown in Figure 6. This tutorial uses an IBM i user profile named DB2. A user profile name of `DB2` is not a requirement for Rails applications. Your profile name could be any user profile on your IBM i system. The PowerRuby `RAILSNEW` command will ask you for a profile you wish to use and update `database.yml` on create, but as promised, this is what is happening under the covers. The `database.yml` file is located in the /www/apachedft/htdocs/strsql/config directory.

Figure 6. Edit database.yml user ID, password, and options


database.yml - /www/apachedft/ht
File Edit Search Preferences S

```
# profile
db2profile: &db2profile
  adapter: ibm_db
  database: "*LOCAL"
  username: DB2
  password: MYPWD

development:
  <<: *db2profile

test:
  <<: *db2profile

production:
  <<: *db2profile
```

There are many ibm_db options for the `database.yml` file, and thanks to IBM DB2 Connect™ 10.5, most will work remotely from your workstation connected to a DB2 for i server. DB2 Connect is a licensed program product that needs to be purchased. For production usage, the DB2 Connect Unlimited Edition for System i packaging typically offers the best terms for IBM i customers. Contact your local IBM representative or IBM Business Partner for pricing information. For more information on this product, see the DB2 Connect website. A trial DB2 Connect license file for evaluation purposes can be obtained by sending an email to: rmahendr@us.ibm.com.

Additionally, DB2 for i system naming mode, starting with IBM i 7.1, accepts both SQL naming `schema.table` and system naming `LIB/FILE`. So, open source software such as Ruby/Rails has a higher probability of working with IBM i traditional applications dependent on the `*LIBL` (library list) behavior. However, the PowerRuby team has not tested Rails under all possible IBM i options. So, it is perhaps best to simply use the default values for your Rails application.

The follow is a list of possible DB2 values in your `database.yml`, IBM i specific values labeled 'ibm_i_' are intended for DB2 for i, only one value is allowed on a single line.

List of possible `database.yml` options

```
development:
  schema: BOB
  app_user: bob
  account: bob
  application: bob1
  workstation: bobws
  ibm_i_naming: system, sql (default sql)
  ibm_i_libl: BOB QTEMP (CHGLIBL)
  ibm_i_curlib: BOB (CHGCURLIB)
  ibm_i_sort_seq: job, system (default system)
  ibm_i_isolation: none, ur, cs, rs, rr
  ibm_i_date_fmt: iso, usa, eur, jis, dmy, mdy, ymd, jul, job
  ibm_i_time_fmt: iso, usa, eur, jis, hms
  ibm_i_date_sep: slash, dash, period, comma, blank, job
  ibm_i_time_sep: colon, period, comma, blank, job
  ibm_i_decimal_sep: period, comma, job
  ibm_i_query_goal: first, all
  <<: *db2profile
```

YiPs ibm_db versions include `database.yml` password encryption ability, details of encrypt support are not fully tested, and therefore, support might change in ibm_db gem. However, if you chose to use encrypt support today, here are the new ibm_db APIs.

- `pwd_key = ActiveRecord::Base.ibm_db_generate_key()`
- `pwd_enc = ActiveRecord::Base.ibm_db_generate_password(password,pwd_key)`
  - Sample program ibm_db download `test/IBMi/samples/genpassword.rb`

You have multiple choices for database.yml:

- Open passwords one database.yml

```
db2profile: &db2profile
  adapter: ibm_db
  database: "*LOCAL"
  username: DB2
  password: MYPWD
development:
  <<: *db2profile
test:
  <<: *db2profile
production:
  <<: *db2profile

Notes:
- typical Rails yaml file
- password open/clear password
```

- Everything in one database.yml

```
db2profile: &db2profile
  adapter: ibm_db
  database: "*LOCAL"
  username: DB2
  pwd_enc: "alkqap/Ao7ACWwizSQ2JvZ86+s0yR5FdDmIU68JuQv4=%0A"
  pwd_key: "YIPS4321AAAAAAAAAAAAAAA132424245"
development:
  <<: *db2profile
test:
  <<: *db2profile
production:
  <<: *db2profile

Notes:
- encrypted password, but key in the
same file
- pwd_key =
ActiveRecord::Base.ibm_db_generate_key() (or your 32 characters)
- pwd_enc =
ActiveRecord::Base.ibm_db_generate_password(password,pwd_key)
```

- Separate yaml files
  - database.yml

    ```
    db2profile: &db2profile
      adapter: ibm_db
      database: "*LOCAL"
      username: DB2
      pwd_yaml: /my/safe/path/password.yml
    development:
      <<: *db2profile
    test:
      <<: *db2profile
    production:
      <<: *db2profile

    Notes
    - encrypted password and key in separate files (limit access)
    - pwd_yaml path to password yaml file
    ```
  - password.yml

    ```
    key_yaml: /my/really/safe/path/key.yml
    DB2:
    pwd_enc: "alkqap/Ao7ACWwizSQ2JvZ86+s0yR5FdDmIU68JuQv4=%0A"
    FLINROCK:
    pwd_enc: "alkqap/Ao7ACWwizSQ2Jvrdt+s0yR5FdDmIU68JuQv4=%0A"
    SLATER:
    pwd_enc: "alkqap/Ao7A123izSQ2Jvrdt+s0yR5FdDmIU68JuQv4=%0A"

    Notes:
    - admin many profiles, where key is username from database.yml (DB2 this case)
    - key_yaml path to key yaml file
    - pwd_enc = ActiveRecord::Base.ibm_db_generate_password(password,pwd_key)
    ```
  - key.yml

    ```
    pwd_key: "YIPS4321AAAAAAAAAAAAAAA132424245"

    Notes:
    - pwd_key used both generation/runtime
    - pwd_key = ActiveRecord::Base.ibm_db_generate_key() (or your 32 characters)
    ```

Step 6. Try web server

So far, we completed `rails new` house keeping administration, edited Gemfile, and database.yml. Therefore, we now have enough Rails application to check our web environment. For the web server component, we have many options in Rails living within IBM i Apache.

Most Rails applications are deployed in proxy/reverse configuration, meaning, you start/stop a Rails application independently of your main website. Non-Apache web servers exist to start/stop/mange Rails applications as child jobs, but these web servers largely hide proxy/reverse configuration (nice for administrators, challenging for developers to understand). This tutorial is using the IBM i Apache web server, and therefore, it will follow traditional Rails reverse proxy configuration and start the Rails web server independent of the Apache web server.

IBM i Apache server start.

- `STRTCPSVR SERVER(*HTTP) HTTPSVR(APACHEDFT)`

Rails webrick server start.

```
 cd /www/apachedft/htdocs/strsql
rails server -p 4242
(Ctrl-C to shutdown server)
```

Note: We can choose to deploy the Rails application as Apache FastCGI similar to PHP on i (Zend Server), but FastCGI is not the Rails way. Not to mention you might have trouble finding information on using Rails and FastCGI together.

PowerRuby RAILSNEW and RAILSSVR commands

Rails issues dealing with the IBM i Apache server largely disappear using PowerRuby smart IBM i wizards (RAILSNEW and RAILSSVR). Essentially, the `RAILSNEW` and `RAILSSVR` commands are Apache DocumentRoot agnostic, allowing you to create your Rails application anywhere in the IFS and `RAILSNEW` creates an Apache configuration without the need of DocumentRoot. `RAILSNEW` creates an Apache configuration (httpd.conf), which passes all web traffic directly to your independently started Rails application (RewriteRule/ProxyPassReverse). PowerRuby `RAILSSVR` of course, manages day-to-day administration aspects such as starting and stopping your Rails applications (rails server -p 4242). And the `RAILSSVR` command can be used as a stand-alone interface to batch-submit the application in this tutorial.

RAILSNEW no DocumentRoot issues

```
httpd.conf:
Listen *:10022
RewriteEngine On
RewriteRule ^(.*) http://127.0.0.1:4242/$1 [P]
ProxyPassReverse / http://127.0.0.1:4242/
```

The RAILSNEW and RAILSVR wizards are a great way to get online quickly with Rails applications. However, when you try to fit a Rails application into an existing website, or want other ways of configuring your site, you might need a better understanding of the Rails experience.

Step 6.1. Try web server (simple site detour)

Our tutorial goal is to add a Rails application to an exiting site (next step), but we will also take a quick look at a simple one Rails application site.

In previous steps, we created our Rails application in the Apache subdirectory, `/www/apachedft/htdocs/strsql`, to avoid issues with Apache subdirectories, we must modify our Apache DocumentRoot to fit our one Rails application site.

```
/www/apachedft/htdocs/strsql
/www/apachedft/httpd.conf:
Listen *:10022

# DocumentRoot /www/apachedft/htdocs
DocumentRoot /www/apachedft/htdocs/strsql

# Simple configuration (like RAILSNEW)
RewriteEngine On
RewriteRule ^(.*) http://127.0.0.1:4242/$1 [P]
ProxyPassReverse / http://127.0.0.1:4242/
```

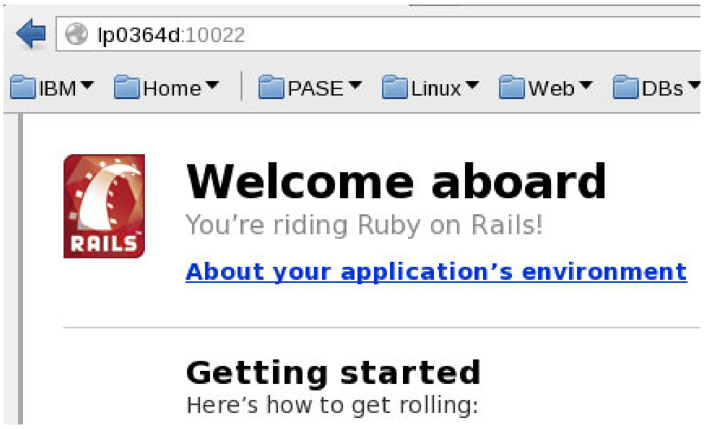
Now, we start our Rails application.

```
cd /www/apachedft/htdocs/strsql
rails server -p 4242
(Ctrl-C to shutdown server)
```

And we start our Apache server — order of start is not important.

```
STRTCPSVR SERVER(*HTTP) HTTPSVR(APACHEDFT)
```

Figure 7. First STRSQL welcome screen



Welcome aboard

You're riding Ruby on Rails!

About your application's environment

Getting started

Here's how to get rolling:

`http://lp0364d:10022`

Technically speaking, as Apache reverse proxy sends a request `http://lp0364d:10022`, HTTP "get route" seen by your Rails application contains "/". There are no Apache subdirectory issues, because we changed DocumentRoot to `/www/apachedft/htdocs/strsql`, so Rails reverse proxy just works.

Note: We are essentially mimicking RAILSNEW and RAILSVR wizards by removing all Apache DocumentRoot subdirectory issues. In fact, we could move the Rails application out of `/www/apachedft/htdocs` and start in another directory and all would continue to work.

Step 6.2. Try web server (existing site)

When you create an IBM i Apache instance, DocumentRoot is set to `/www/instance/htdocs`. This enables a '/' root directory container for your many web application running the websites: htdocs/index.html, htdocs/wiki, htdocs/strsql, and so on to wit, keeps web traffic contained to `/www/instance/htdocs` child subdirectories to avoid hacker anarchy access to non-web parts of your system. So, let's add the new application we have been building in the subdirectory, `htdocs/strsql`.

Fail first attempt

We are adding a Rails application to an existing website and we want to use the traditional Rails proxy/reverse method, therefore, a simple modification to the previous example Apache configuration adding '/strsql' seems logical, but does not work.

```
/www/apachedft/httpd.conf:
Listen *:10022

DocumentRoot /www/apachedft/htdocs
# DocumentRoot /www/apachedft/htdocs/strsql

# Simple configuration (like RAILSNEW)
# RewriteEngine On
# RewriteRule ^(.*) http://127.0.0.1:4242/$1 [P]
# ProxyPassReverse / http://127.0.0.1:4242/

# Existing web site (change config/routes.rb)
RewriteEngine on
RewriteRule ^/strsql(.*) http://127.0.0.1:4242/strsql/$1 [P]
ProxyPassReverse /strsql/ http://127.0.0.1:4242/strsql/
```

Now we start our Rails application.

```
cd /www/apachedft/htdocs/strsql
rails server -p 4242
(Ctrl-C to shutdown server)
```

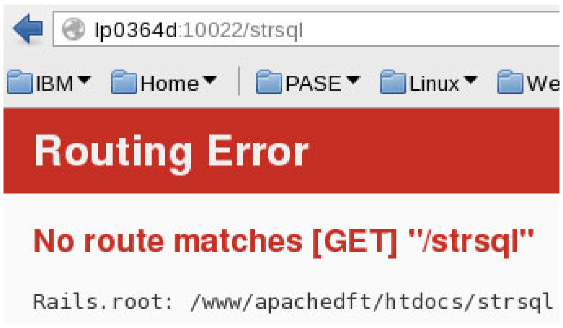
And, we start our Apache server — the starting order is not important

```
STRTCPSVR SERVER(*HTTP) HTTPSVR(APACHEDFT)
```

Rails and the Apache subdirectory problem

This tutorial is about lifting the curtain on the PowerRuby CL-based helpers and exposing Rails plumbing. There are many Ruby folks who face difficulty in deploying Rails applications in an existing website using Apache. The good news is that Rails can work within existing IBM i Apache websites, but you must understand the true multiple application website landscape. As we see from the browser error, our first issue, unfortunately, Rails does not work well as an Apache subdirectory to DocumentRoot `'/strsql'`.

Figure 8. Error STRSQL welcome screen



Routing Error

No route matches [GET] "/strsql"

Rails.root: /www/apachedft/htdocs/strsql

`http://lp0364d:10022/strsql`
Technically speaking, as the Apache reverse proxy sends a subdirectory request `http://lp0364d:10022/strsql`, HTTP "get route" seen by your Rails application contains the subdirectory, "/strsql". Unfortunately, your Rails application was started under subdirectory and assumes the DocumentRoot location "/", which results in a route mismatch Apache sent "/strsql" and you see an error in your browser.

Rails subdirectory scope fix

Rails provides an easy work around by changing the Rails application configuration file, config/routes.rb, to understand the subdirectory scope "/strsql" sent by Apache.

```
config/routes.rb (use scope):
scope "/strsql" do
  get "welcome/index"
  root :to => 'welcome#index'
end
```

Simple enough fix, however, our tutorial in not yet ready to test out this fix. A STRSQL welcome page to replace the default Rails welcome page is explained in the next step.

Later versions of Rails

This tutorial is written generically for any version of Rails. As you approach later versions of Rails edge documentation, you find alternative ways dealing with Rails applications in a subdirectory. Feel free to try the following recommendation.

Add this line to the config/environments/development.rb file.

```
config.relative_url_root = "/strsql"
```

Then modify the config.ru file.

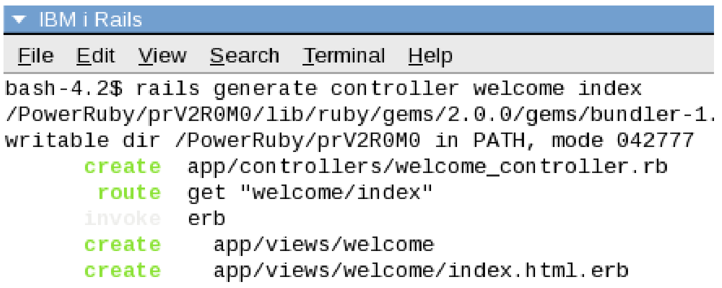
```
require ::File.expand_path('../config/environment',  __FILE__)
map Rails.application.config.relative_url_root || "/" do
  run Rails.application
end
```

You should be able to remove most manually entered `/strsql` in the tutorial.

Step 7. Create welcome page

We wish to replace the default Rails welcome screen (simple site detour), and actually begin the process of building our application. The first task is to create a welcome page. We will be viewing the default index page created by `rails generate controller welcome index` command in the /www/apachedft/htdocs/strsql directory as shown in Figure 9. So, you see exactly what this Rails command generates. However, don't worry: we will modify our page to mimic STRSQL later in the tutorial.

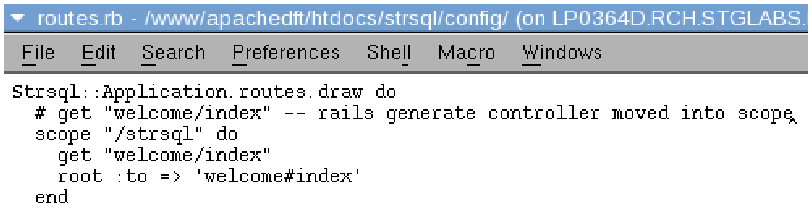
Figure 9. Rails generate controller command



```
bash-4.2$ rails generate controller welcome index
/PowerRuby/prV2R0M0/lib/ruby/gems/2.0.0/gems/bundler-1.
writable dir /PowerRuby/prV2R0M0 in PATH, mode 042777
      create  app/controllers/welcome_controller.rb
       route  get "welcome/index"
      invoke  erb
      create    app/views/welcome
      create    app/views/welcome/index.html.erb
```

We will find our new welcome page at `app/views/welcome/index.html.erb`. However, as mentioned previously, before we can try our web server, we need to adjust the `config/routes.rb` file in the in the /www/apachedft/htdocs/strsql/ directory to understand our Apache subdirectory, "/strsql" by adding the scope command as shown in Figure 10 .

Figure 10. Edit routes.rb to add subdirectory scope


routes.rb - /www/apachedft/htdocs/strsql/config/ (on LP0364D.RCH.STGLABS.
File Edit Search Preferences Shell Macro Windows

```
Strsql::Application.routes.draw do
  # get "welcome/index" -- rails generate controller moved into scope
  scope "/strsql" do
    get "welcome/index"
    root :to => 'welcome#index'
  end
```

Step 7.1. Try welcome web server

We added our welcome page, `app/views/welcome/index.html.erb,` and we fixed the scope '/strsql' in the `config/routes.rb` file. So our existing site Rails application will now work (repeat steps).

```
# rails /strsql (subdir affects rails)
RewriteEngine on
RewriteRule ^/strsql(.*) http://127.0.0.1:4242/strsql/$1 [P]
ProxyPassReverse /strsql/ http://127.0.0.1:4242/strsql/
```

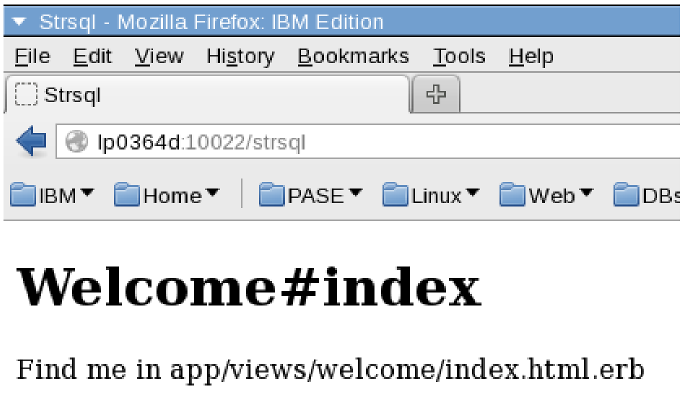
Now, we start our Rails application.

```
cd /www/apachedft/htdocs/strsql
rails server -p 4242
(Ctrl-C to shutdown server)
```

And, we start our Apache server — the starting order is not important.

```
STRTCPSVR SERVER(*HTTP) HTTPSVR(APACHEDFT)
```

Figure 11. Basic STRSQL command screen


Strsql - Mozilla Firefox: IBM Edition
File Edit View History Bookmarks Tools Help
Strsql
lp0364d:10022/strsql
IBM Home PASE Linux Web DBs

Welcome#index

Find me in app/views/welcome/index.html.erb

As Figure 11 displays, the default welcome page is nothing fancy, but it works!

So far …

Time to take a breath and recap our progress:

- rails new strsql —skip-sprockets —skip-bundle
    - Created our rails application in `/www/apachedft/htdocs/strsql`
    - Option: —skip-sprockets used to avoid asset pipeline (IBM i does not have native JavaScript)
    - Option: —skip-bundle used to avoid updates from http://rubygems.org
- Edit `strsql/Gemfile`
    - change from SQLite3 to ibm_db (DB2)
- Bundle install —local
    - bundle locked our Gemfile.lock to PowerRuby
    - option: —local avoids unwanted updates into PowerRuby (leave alone)
- Edit `/strsql/db/database.yml`
    - Open passwords one database.yml (traditional Rails)
    - Encrypted passwords/key everything in one database.yml
    - Encrypted passwords/key separate .yml files
- Start the web server
    - Tutorial following traditional dual server Rails model proxy/reverse
    - IBM i Apache start.

      ```
      STRTCPSVR SERVER(*HTTP) HTTPSVR(APACHEDFT)
      ```
    - Rails webrick server start.

      ```
      cd /www/apachedft/htdocs/strsql
      rails server -p 4242
      Ctrl-C to shutdown server)
      ```
    - Multiple configurations possible.

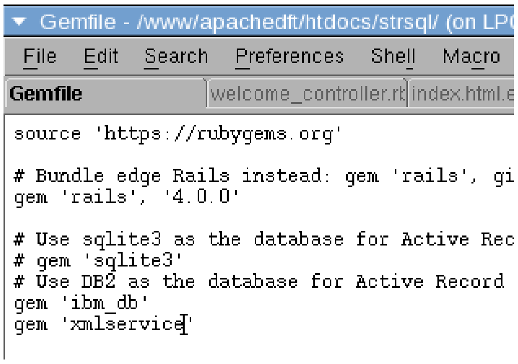
```
# Simple configuration (like RAILSNEW)
RewriteEngine On
# RewriteRule ^(.*) http://127.0.0.1:4242/$1 [P]
# ProxyPassReverse / http://127.0.0.1:4242/

# Existing web site (change config/routes.rb)
RewriteEngine on
RewriteRule ^/strsql(.*) http://127.0.0.1:4242/strsql/$1 [P]
ProxyPassReverse /strsql/ http://127.0.0.1:4242/strsql/
```

- Next steps
  - An example of adding the Rails application to the existing website
  - No longer have to start/stop Apache server, only the Rails application server needs to start/stop for the remainder of the tutorial.

Step 8. Add xmlservice (Gemfile)

We wish to use the gem xmlservice to process our STRSQL commands. Therefore, we must alter Gemfile in the /www/apachedft/htdocs/strsql directory as shown in Figure 12 and run the bundle command again. As mentioned earlier, the bundle install `'--local'` command invocation will lock our Gemfile.lock to PowerRuby gems, and includes PowerRuby xmlservice gem.

Figure 12. Edit Gemfile to add xmlservice

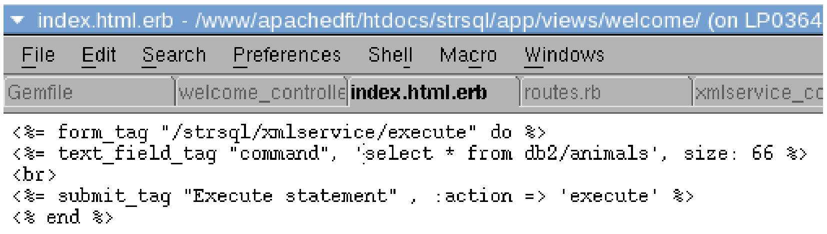


```
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', gi
gem 'rails', '4.0.0'

# Use sqlite3 as the database for Active Rec
# gem 'sqlite3'
# Use DB2 as the database for Active Record
gem 'ibm_db'
gem 'xmlservice'
```

Step 8.1. STRSQL welcome page (view)

We need our welcome page to look like STRSQL, so that SQL statements can be submitted to the xmlservice gem. For now, let's ignore recalling previous SQL requests and add a very simple form with a text input to act as our STRSQL command line. This is done by editing the index.html.erb file in the /www/apachedft/htdocs/strsql/app/views/welcome/ directory.

Figure 13. Edit index.html.erb file to add simple STRSQL command line



```
<%= form_tag "/strsql/xmlservice/execute" do %>
<%= text_field_tag "command", 'select * from db2/animals', size: 66 %>
<br>
<%= submit_tag "Execute statement" , :action => 'execute' %>
<% end %>
```

Any `html.erb` file might look intimidating at first glance, but similar to HTML, you simply have to play around with the syntax to see how it works and consult the Ruby erb manual .

- erb is a mix of pseudo-HTML and pure HTML.
- erb is processed by Ruby, and therefore, Ruby statements and variables can be used (not this form).

Specifically, this form, we are hard coding REST target `/strsql/xmlservice/execute`, and defaulting to the 'post' action. Rails has many other ways to accomplish the form task, but true to this tutorial, we are staying close to the plumbing to aid understanding (and to keep it simple). The `submit_tag "Execute statement"` is a button on the welcome form.

The `:action => 'execute'` directive routes to the `def execute` method in our xmlservice controller.

```
app/controllers/xmlservice_controller.rb:
class XmlserviceController  < ApplicationController
  def execute
  end
end
```

If you read the manual, you will see `text_field_tag "command"` will render as `params[:command]` in our xmlservice controller. So, now we know how to transfer the

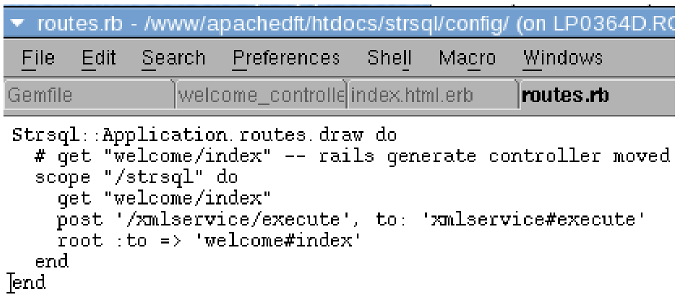
STRSQL request from the welcome form text input field.

```
app/controllers/xmlservice_controller.rb:
class XmlserviceController < ApplicationController
  def execute
  @command =  params[:command]
  end
end
```

Step 8.2. STRSQL execute route (route)

We need to modify `routes.rb` file in the /www/apachedft/htdocs/strsql/config directory to route our new STRSQL welcome form 'post' request `/strsql/xmlservice/execute`.

Figure 14. Edit routes.rb file to new STRSQL command screen


routes.rb - /www/apachedft/htdocs/strsql/config/ (on LP0364D.R
File Edit Search Preferences Shell Macro Windows
Gemfile
welcome_controlle
index.html.erb
routes.rb

```
Strsql::Application.routes.draw do
  # get "welcome/index" -- rails generate controller moved
  scope "/strsql" do
    get "welcome/index"
    post '/xmlservice/execute', to: 'xmlservice#execute'
    root :to => 'welcome#index'
  end
end
```

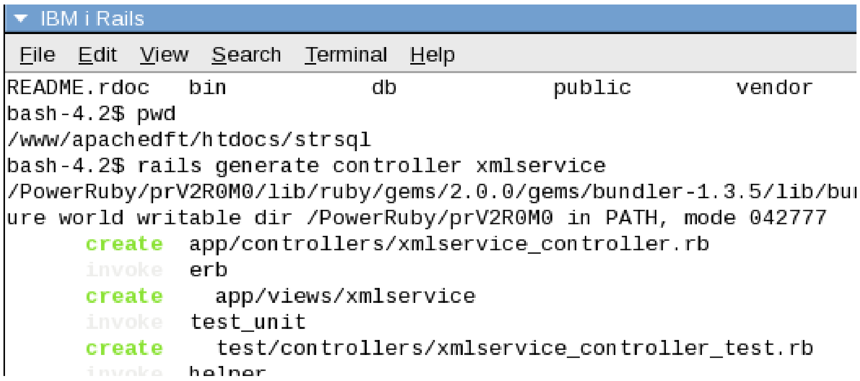
As we learned, life in a the '/strsql/ Apache subdirectory requires our routes to be scoped, therefore we place the target of `/strsql/xmlservice/execute` inside the scoped block. We know action is post using the welcome form and that '/strsql' will be part of the scope, leaving `/xmlservice/execute` in routes, which we route as `controller#action` to our xmlservice controller `xmlservice#execute` on form submit.

```
app/views/welcome/index.html.erb:
<%= form_tag "/strsql/xmlservice/execute" do %>
<%= text_field_tag "command", 'select * from db2/animals', size: 66 %>
<br>
<%= submit_tag "Execute statement" , :action => 'execute' %>
<% end %>


config/routes.rb:
Strsql::Application.routes.draw do
  scope "/strsql" do
    get "welcome/index"
    post '/xmlservice/execute', to: 'xmlservice#execute'
    root :to => 'welcome#index'
    end
  end

app/controllers/xmlservice_controller.rb:`
class XmlserviceController < ApplicationController
  def execute
  @command =  params[:command]
  end
end
```

Step 9. Add xmlservice (create)

We need a controller to handle `xmlservice` requests for STRSQL. This is done by executing the rails generate controller command as shown in Figure 15 in the /www/apachedft/htdocs/strsql directory.

Figure 15. Rails command to generate controller xmlservice

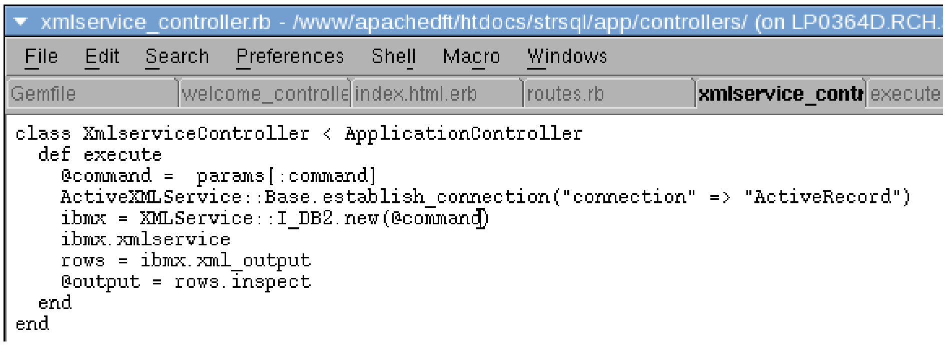


```
README.rdoc   bin           db            public        vendor
bash-4.2$ pwd
/www/apachedft/htdocs/strsql
bash-4.2$ rails generate controller xmlservice
/PowerRuby/prV2R0M0/lib/ruby/gems/2.0.0/gems/bundler-1.3.5/lib/bu
ure world writable dir /PowerRuby/prV2R0M0 in PATH, mode 042777
      create  app/controllers/xmlservice_controller.rb
      invoke  erb
      create    app/views/xmlservice
      invoke  test_unit
      create    test/controllers/xmlservice_controller_test.rb
      invoke  helper
```

Step 9.1. Add xmlservice (controller)

We need some code to handle the `strsql/xmlservice/execute` request from the welcome form. This code is added by editing the xmlservice_controller.rb file in the /www/apachedft/htdocs/strsql/app/controllers directory.

Figure 16. Edit xmlservice_conroller.rb file to handle execute SQL requests


xmlservice_controller.rb - /www/apachedft/htdocs/strsql/app/controllers/ (on LP0364D.RCH.
File Edit Search Preferences Shell Macro Windows
Gemfile welcome_controlle index.html.erb routes.rb xmlservice_contr execute

```
class XmlserviceController < ApplicationController
  def execute
    @command =  params[:command]
    ActiveXMLService::Base.establish_connection("connection" => "ActiveRecord")
    ibmx = XMLService::I_DB2.new(@command)
    ibmx.xmlservice
    rows = ibmx.xml_output
    @output = rows.inspect
  end
end
```

Again, our welcome form will route as `controller#action` to our xmlservice controller `xmlservice#execute` on the form submit action.

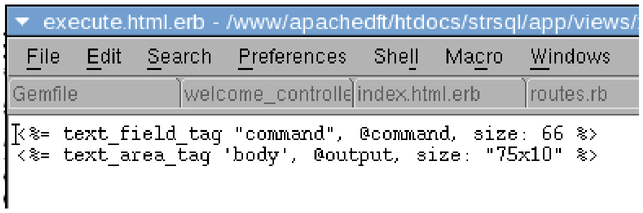
```
config/routes.rb:
Strsql::Application.routes.draw do
  scope "/strsql" do
   get "welcome/index"
   post '/xmlservice/execute', to: 'xmlservice#execute'
   root :to => 'welcome#index'
  end
end

app/controllers/xmlservice_controller.rb:
class XmlserviceController < ApplicationController
  def execute
    @command =  params[:command]
    ibmx = XMLService::I_DB2.new(@command)
    ibmx.xmlservice
    rows = ibmx.response.output
    @output = rows.inspect
  end
end
```

ActiveXMLService provides default `ActiveXMLService::Base.establish_connection("connection" => "ActiveRecord")`, thereby default XMLSERVICE requests flow on your current DB2 ActiveRecord connection (I_DB2, I_PGM, I_SRVPGM, and so on). A connection to DB2 for i started when our Rails application started `rails server -p 4242`. In addition, XMLSERVICE provides a stored procedure interface (iPLUG4K - iPLUG15MB), therefore, `def execute` only need to use `ibmx = XMLService::I_DB2.new(@command)` and it all works (simple Rails model).

Step 9.2. Add xmlservice (view)

We need a view for xmlservice execute. We did not ask `rails generate controller xmlservice` for a view component, so we just add a view manually by editing the execute.html.erb file in the /www/apachedft/htdocs/strsql`/app/views/xmlservice/execute.html.erb` directory.

Figure 17. Add view for xmlservice result execute to execute.html.erb


execute.html.erb - /www/apachedft/htdocs/strsql/app/views/
File Edit Search Preferences Shell Macro Windows
Gemfile
welcome_controlle
index.html.erb
routes.rb

```
<%= text_field_tag "command", @command, size: 66 %>
<%= text_area_tag 'body', @output, size: "75x10" %>
```

Nothing fancy on the xmlservice execute view, simply dump the output of xmlservice data provided by xmlservice controller in `text_area_tag`. Displayed information is `@command = params[:command]` command passed through from welcome form, and of course data returned from xmlservice `@output = rows.inspect` . XMLSERVICE on IBM i only deals with XML documents, so XML document to array output was handled by xmlservice gem, `XMLService::I_DB2.new(@command)`.

```
app/views/xmlservice/execute.html.erb:
<%= text_field_tag "command", @command, size: 66 %>
<%= text_area_tag 'body', @output, size: "75x10" %>

app/controllers/xmlservice_controller.rb:
class XmlserviceController < ApplicationController
  def execute
    @command =  params[:command]
    ibmx = XMLService::I_DB2.new(@command)
    ibmx.xmlservice
    rows = ibmx.response.output
    @output = rows.inspect
  end
end
```

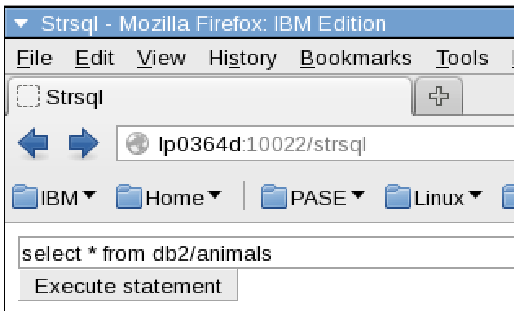
Step 9.3. STRSQL query animals

We now have a fully functional STRSQL, and now, you can restart the Rails application with the following commands.

```
 cd /www/apachedft/htdocs/strsql rails server -p 4242 (Ctrl-C to
          shutdown server)
```

STRSQL welcome page now appears as shown in Figure 18.

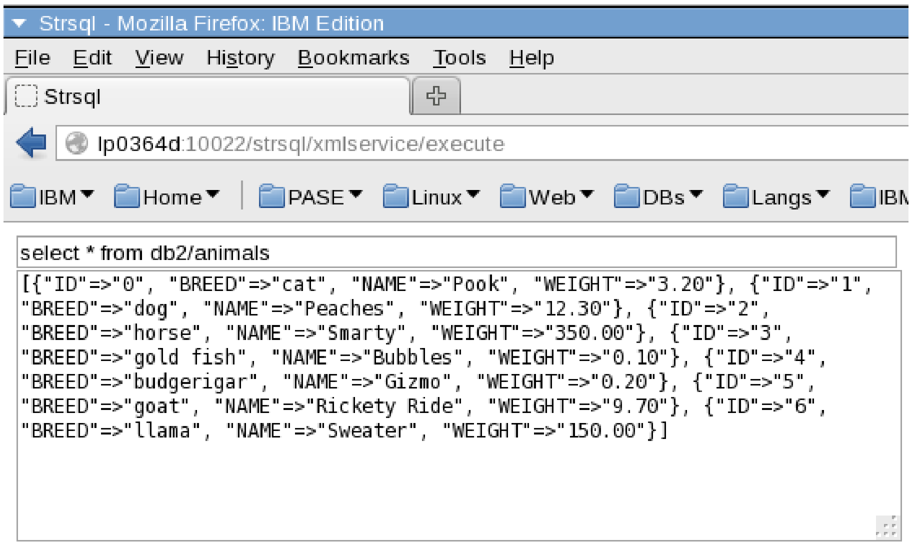
Figure 18. STRSQL command line welcome page


Strsql - Mozilla Firefox: IBM Edition
File Edit View History Bookmarks Tools
Strsql
lp0364d:10022/strsql
IBM
Home
PASE
Linux
Execute statement

select * from db2/animals

Click **Execute statement** to run the SQL `select` statement found in the text box: `select * from db2/animals`. The output returned by the executed `select` statement is displayed in Figure 19.

Figure 19. STRSQL execute simple SQL query output


Strsql - Mozilla Firefox: IBM Edition
File Edit View History Bookmarks Tools Help
Strsql
lp0364d:10022/strsql/xmlservice/execute
IBM Home PASE Linux Web DBs Langs IBM

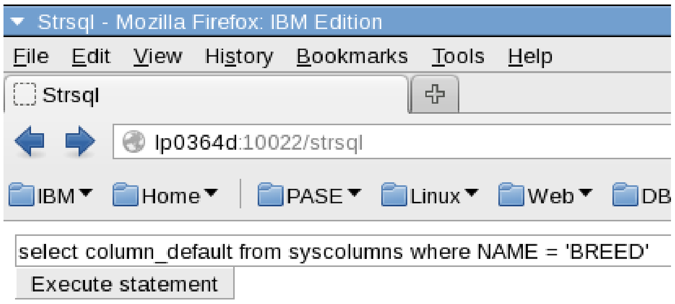
select * from db2/animals

```
[{"ID"=>"0", "BREED"=>"cat", "NAME"=>"Pook", "WEIGHT"=>"3.20"}, {"ID"=>"1",
"BREED"=>"dog", "NAME"=>"Peaches", "WEIGHT"=>"12.30"}, {"ID"=>"2",
"BREED"=>"horse", "NAME"=>"Smarty", "WEIGHT"=>"350.00"}, {"ID"=>"3",
"BREED"=>"gold fish", "NAME"=>"Bubbles", "WEIGHT"=>"0.10"}, {"ID"=>"4",
"BREED"=>"budgerigar", "NAME"=>"Gizmo", "WEIGHT"=>"0.20"}, {"ID"=>"5",
"BREED"=>"goat", "NAME"=>"Rickety Ride", "WEIGHT"=>"9.70"}, {"ID"=>"6",
"BREED"=>"llama", "NAME"=>"Sweater", "WEIGHT"=>"150.00"}]
```

Step 9.4. STRSQL query against Syscolumns catalog view

We can do almost anything with STRSQL at this point, so we will try running a query against the Syscolumns catalog view to return the default column value for any columns named BREED in the database.

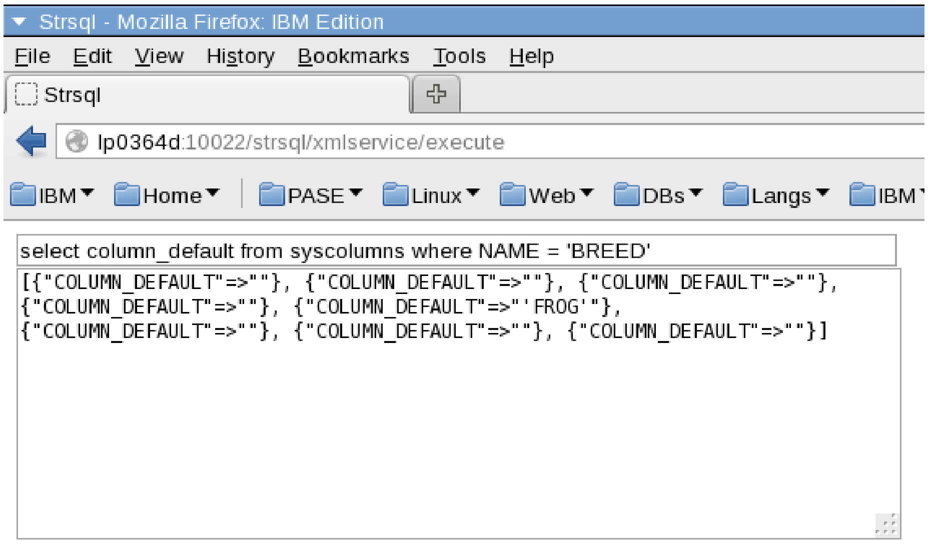
Figure 20. STRSQL command line welcome page


Strsql - Mozilla Firefox: IBM Edition
File Edit View History Bookmarks Tools Help
Strsql
lp0364d:10022/strsql
IBM Home PASE Linux Web DB
Execute statement

select column_default from syscolumns where NAME = 'BREED'

Simply type in the following query, `select column_default from syscolumns where NAME = 'BREED'` into the text box and click the **Execute statement** button. The output of this query is displayed in Figure 21.

Figure 21. STRSQL output for Syscolumns query


Strsql - Mozilla Firefox: IBM Edition
File Edit View History Bookmarks Tools Help
Strsql
lp0364d:10022/strsql/xmlservice/execute
IBM Home PASE Linux Web DBs Langs IBM

```
select column_default from syscolumns where NAME = 'BREED'
```

```
[{"COLUMN_DEFAULT"=>""}, {"COLUMN_DEFAULT"=>""}, {"COLUMN_DEFAULT"=>""},
{"COLUMN_DEFAULT"=>""}, {"COLUMN_DEFAULT"=>"'FROG'"},
{"COLUMN_DEFAULT"=>""}, {"COLUMN_DEFAULT"=>""}, {"COLUMN_DEFAULT"=>""}]
```

So far …

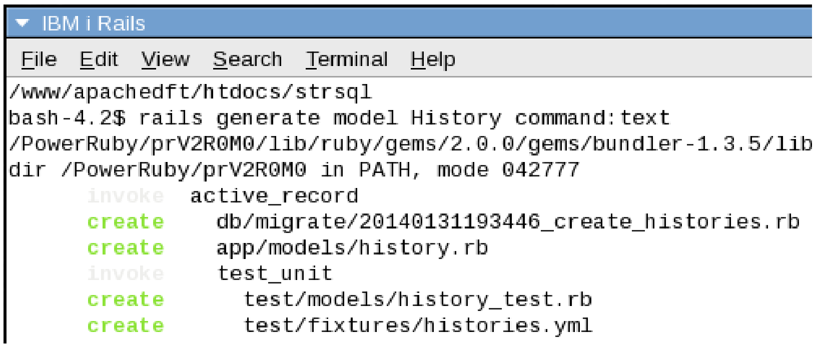
Time to take a breath and recap our progress:

- Edit Gemfile
  - Add xmlservicegem
  - Bundle install —local
    - Locked Gemfile.lock to PowerRuby (with `xmlservice`)
- Edit welcome page `app/views/welcome/index.html.erb`
  - Made a crude text entry to run STRSQL requests (like the green screen)
  - Added a form that will post requests to `/strsql/xmlservice/execute`
- Edit routes `config/routes.rb`
  - Added a 'post' route to handle `/xmlservice/execute`
- Rails generate controller `xmlservice`
  - Generated a controller to handle `/xmlservice/execute`
- Edit xmlservice controller to add the `execute` method. `app/controllers/xmlservice_controller.rb`
  - After calling the `xmlservice` class variable @output has result
- Edit the `xmlservice` view for execute to dump processed output
  - Crude text area to display provided @output variable
- Next steps?
  - Cleanup the STRSQL views
  - Create a recall model database table to emulate the ability of STRSQL to retrieve previously run SQL statements

Step 10. Create history (model)

STRSQL needs a history database to enable 'recall previous command' similar to to the retrieve capability provided by the STRSQL command F9 function key. The Rails command, `rails generate model History command:text,` executed in the /www/apachedft/htdocs/strsql directory will build a database migration model template file. The template is found in the `db/migrate/20140131193446_create_histories.rb` file. Our History table will contain an integer ID key column (automatic), and we added a command of type text `command:text`, where `text` maps to the DB2 CLOB data type.

Figure 22. Generate model database history



```
/www/apachedft/htdocs/strsql
bash-4.2$ rails generate model History command:text
/PowerRuby/prV2R0M0/lib/ruby/gems/2.0.0/gems/bundler-1.3.5/lib
dir /PowerRuby/prV2R0M0 in PATH, mode 042777
      invoke  active_record
      create    db/migrate/20140131193446_create_histories.rb
      create    app/models/history.rb
      invoke    test_unit
      create      test/models/history_test.rb
      create      test/fixtures/histories.yml
```

Rails plurals

Note that the `rails generate model` command specified a name of `History`, whereby, Rails immediately asserted the actual physical table name 'Histories', `db/migrate/20140131193446_create_histories.rb`. The intent of pluralization is to make your code more readable and transparent, but this convention drives some developers crazy. The following list contains the default Rails rules for plural and singular. Becoming familiar with these three conventions will go a long way toward getting comfortable with Rails.

- Database table names: plural
  - Database table names are expected to be pluralized. For example, a table containing employee records should be named Employees.
- Model class names: singular
  - Model class names are the singular form of the database table that they are modeling. For example, an employee model is created based on a table named Employees.
- Controller class names: plural
  - Controller class names are pluralized, such as EmployeesController or AccountsController.

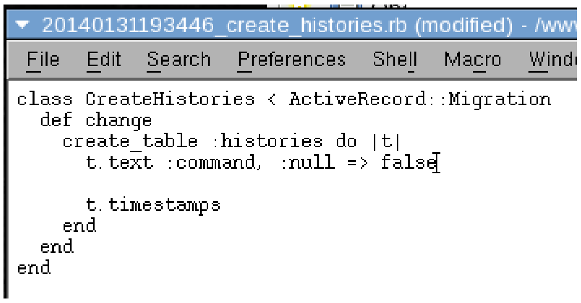
Where is my table?

The physical DB2 table, Histories, was not created using the command `rails generate model`, only a migration template was created in the `db/migrate/20140131193446_create_histories.rb file.` Therefore, thereby allowing us to edit custom attributes before the actual migration and creation of the DB2 table object. You can verify this by using the schema value specified in the database.yml file on the WRKLIB (work with libraries) system command. Using a traditional Rails database.yml file, the DB2 default schema will be the same as the user name directive (username: DB2). Other schema options are possible by making configuration changes to the database.yml file.

Step 10.1. Edit history migration template file

Edit the generated migrate file `db/migrate/20140131193446_create_histories.rb` to add any attributes required before the actual migration and creation of the DB2 histories table. The Histories table definition was customized to not allow NULL values by adding `:null => false` as shown in Figure 23.

Figure 23. Edit database migration template for Histories table

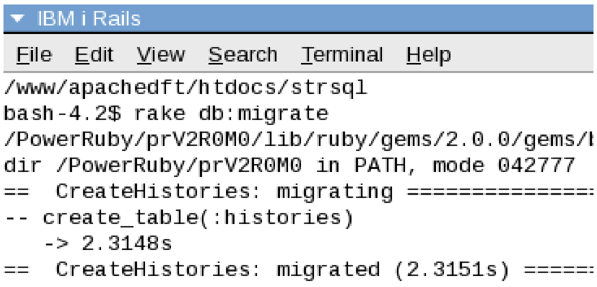

20140131193446_create_histories.rb (modified) - /www
File Edit Search Preferences Shell Macro Wind

```
class CreateHistories < ActiveRecord::Migration
  def change
    create_table :histories do |t|
      t.text :command, :null => false

      t.timestamps
    end
  end
end
```

Step 10.2. Create/Migrate History model

After editing the custom attributes in Histories migration, we will generate the actual physical DB2 table using the `rake db:migrate` command in the /www/apachedft/htdocs/strsql directory as shown in Figure 24.

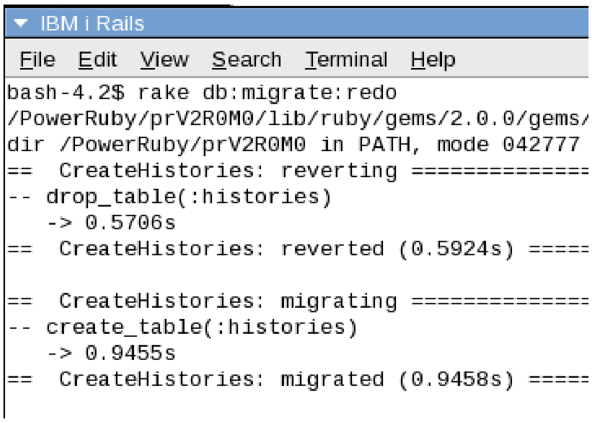
Figure 24. Create database table histories



```
/www/apachedft/htdocs/strsql
bash-4.2$ rake db:migrate
/PowerRuby/prV2R0M0/lib/ruby/gems/2.0.0/gems/
dir /PowerRuby/prV2R0M0 in PATH, mode 042777
==  CreateHistories: migrating ==============
-- create_table(:histories)
   -> 2.3148s
==  CreateHistories: migrated (2.3151s) =====
```

If we made a mistake in our schema design, we can re-createthe tables using the `rake db:migrate:redo` command as shown in Figure 25

Figure 25. Re-create histories table



```
bash-4.2$ rake db:migrate:redo
/PowerRuby/prV2R0M0/lib/ruby/gems/2.0.0/gems/
dir /PowerRuby/prV2R0M0 in PATH, mode 042777
==  CreateHistories: reverting ==============
-- drop_table(:histories)
   -> 0.5706s
==  CreateHistories: reverted (0.5924s) =====

==  CreateHistories: migrating ==============
-- create_table(:histories)
   -> 0.9455s
==  CreateHistories: migrated (0.9458s) =====
```

There are many `rake` options, see command line `rake -T`. Here's a list of some more popular commands below. I didn't have time to test all options with DB2 for i.

```
rake db:create            - creates the database for the current env
rake db:create:all        - creates the databases for all envs
rake db:drop              - drops the database for the current env
rake db:drop:all          - drops the databases for all envs
rake db:migrate           - runs migrations for the current env that have not run yet
rake db:migrate:up        - runs one specific migration
rake db:migrate:down      - rolls back one specific migration
rake db:migrate:status    - shows current migration status
rake db:migrate:rollback  - rolls back the last migration
rake db:forward           - advances the current schema version to the next one
rake db:seed              - (only) runs the db/seed.rb file
rake db:schema:load       - loads the schema into the current env's database
rake db:schema:dump       - dumps the current env's schema
                           (and seems to create the db as well)
rake db:setup             - runs db:schema:load, db:seed
rake db:reset             - runs db:drop db:setup
rake db:migrate:redo      - runs (db:migrate:down db:migrate:up)
                            or (db:migrate:rollback db:migrate:migrate)
                            depending on the specified migration
rake db:migrate:reset     - runs db:drop db:create db:migrate

Migration ID establishes up/down:

> cd /www/apachedft/htdocs/strsql
> rake db:migrate:status
database: *LOCAL
Status   Migration ID    Migration Name
--------------------------------------------------
up     000             ********** NO FILE **********
up     20131017214312  ********** NO FILE **********
up     20131017214541  ********** NO FILE **********
up     20131118211937  ********** NO FILE **********
up     20140131193446  Create histories
```

After migration, you can verify whether the table exists on the server by running the `DSPFD` (display file description) command on a 5250 Emulator command line.

Step 11. Save history commands (xmlservice)

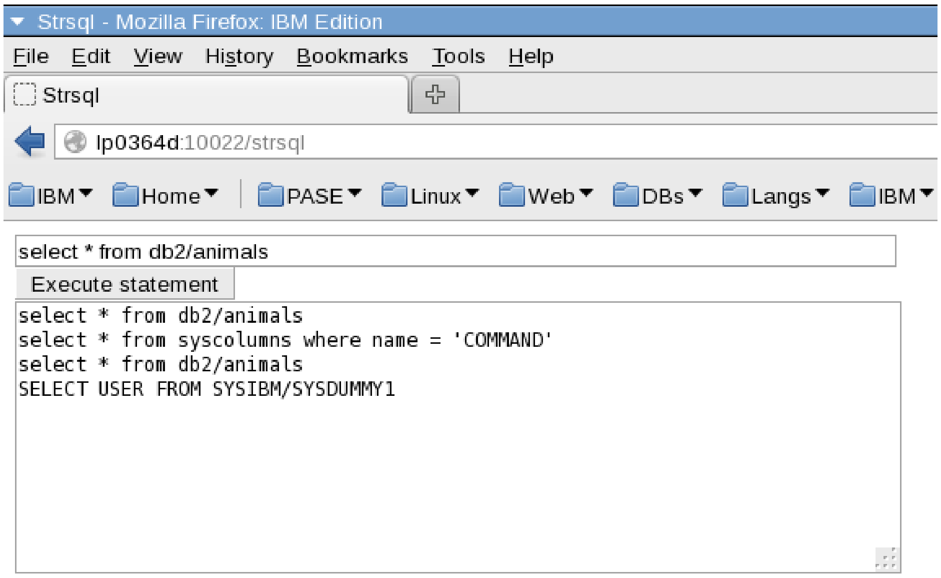
Our xmlservice controller already has the functional capability to run SQL statements. So, all we need do is track History records with `History.create(:command => @command)`.

```
app/controllers/xmlservice_controller.rb:
class XmlserviceController < ApplicationController
  def execute
    @command =  params[:command]
    ibmx = XMLService::I_DB2.new(@command)
    ibmx.xmlservice
    rows = ibmx.response.output
    @output = rows.inspect
    # add to history
    History.create(:command => @command)
  end
end
```

Step 11.1. Display STRSQL history (welcome)

Initially, we will simply display the STRSQL statement history on the STRSQL welcome page as displayed in Figure 26.

Figure 26. STRSQL welcome screen with a simple command history


Strsql - Mozilla Firefox: IBM Edition
File Edit View History Bookmarks Tools Help
Strsql
lp0364d:10022/strsql
IBM Home PASE Linux Web DBs Langs IBM
Execute statement

select * from db2/animals

```
select * from db2/animals
select * from syscolumns where name = 'COMMAND'
select * from db2/animals
SELECT USER FROM SYSIBM/SYSDUMMY1
```

Here is our welcome view and controller code, augmented with history of SQL statements. This uses a default list, so it's nothing fancy – but it works!

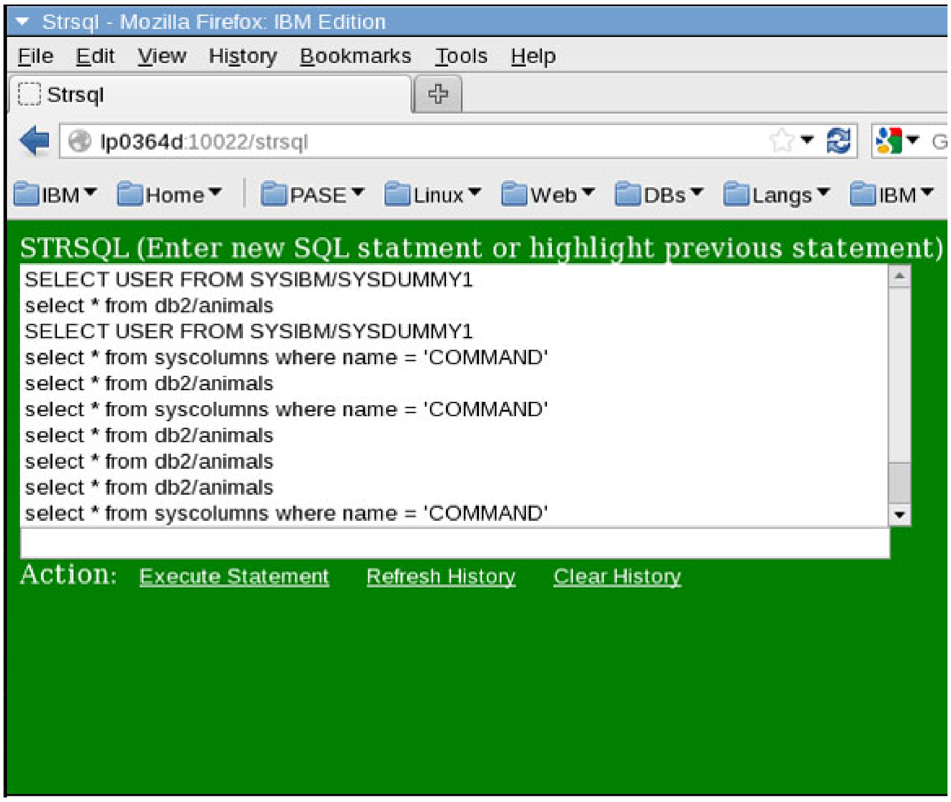
```
app/views/welcome/index.html.erb:
<%= form_tag "/strsql/xmlservice/execute" do %>
<%= text_field_tag "command", 'select * from db2/animals', size: 66 %>
<br>
<%= submit_tag "Execute statement" , :action => 'execute' %>
<br>
<%= text_area_tag 'body', @output, size: "75x10" %>
<% end %>

app/controllers/welcome_controller.rb:
class WelcomeController < ApplicationController
  def index
    # display history
    @output = ""
    History.find_each do |row|
    @output += row.command + "\n"
    end
  end
end
```

Step 12. Fancy history recall

At this point, we have covered the basics of view and controller logic. So, we will run quickly through the STRSQL history recall.

CSS was added to the welcome form to look more like the STRSQL green screen interface, but we took some design liberty in Figure 27 to modernize the task of recalling a previous statement to be a simple mouse click on the history line for a better user experience.

Figure 27. Fancy STRSQL welcome form with SQL history


Strsql - Mozilla Firefox: IBM Edition
File Edit View History Bookmarks Tools Help
Strsql
lp0364d:10022/strsql
IBM Home PASE Linux Web DBs Langs IBM

STRSQL (Enter new SQL statment or highlight previous statement)

```
SELECT USER FROM SYSIBM/SYSDUMMY1
select * from db2/animals
SELECT USER FROM SYSIBM/SYSDUMMY1
select * from syscolumns where name = 'COMMAND'
select * from db2/animals
select * from syscolumns where name = 'COMMAND'
select * from db2/animals
select * from db2/animals
select * from db2/animals
select * from syscolumns where name = 'COMMAND'
```

Action: Execute Statement Refresh History Clear History

We replaced text_area_tag tag with an easy-to-use, line-by-line select_tag and added a small bit of JavaScript in welcome STRSQL `:onchange =>'changeValue("previous","command");` to move any history item selected with a mouse click into the text filed for Execute Statement.

Ruby erb has a very nice select_tag feature, `options_for_select(@output)`, which allows us to pass an array to populate options for our select, and therefore we can alter the welcome controller to pass an array instead of a string used for removed text_area_tag.

```
app/controllers/welcome_controller.rb:
class WelcomeController < ApplicationController
  def index
    # display history
    @output = Array.new
    History.find_each do |row|
    @output << row.command
    end
  end
end
```

We added the submit_tag buttons to our form_tag `/strsql/xmlservice/execute`:

- Execute Statement - run statement (original)
- Refresh History - refresh history list
- Clear History - clear history database

Welcome erb also changed `:action => 'execute'` to `:name => 'execute'`, allowing much simpler coding of multiple button logic in the execute controller.

```
app/views/welcome/index.html.erb:
<head>
<style>
body { background-color:green; color:white; }
select {width:550px; border-style:none;}
option {border-style:none; }
input[type="text"] {width:535px;}
input[type="submit"] {background-color:green; color:white; border-style:none;
 text-decoration:underline; }
</style>
<script type="text/javascript">
function changeValue(id1,id2)
{
  here = document.getElementById(id1);
  there = document.getElementById(id2);
  if (here.selectedIndex >= 0) {
    there.value = here.options[here.selectedIndex].value;
}
}
</script>
</head>
<body>
<%= form_tag "/strsql/xmlservice/execute" do %>
STRSQL (Enter new SQL statment or highlight previous statement)
<br>
<%= select_tag 'previous', options_for_select(@output), size: 10, :onchange =>'
changeValue("previous","command");' %>
<br>
<%= text_field_tag "command", ''%>
<br>
Action: <%= submit_tag "Execute Statement" , :name => 'execute' %>
<%= submit_tag "Refresh History", :name => 'refresh' %>
<%= submit_tag "Clear History", :name => 'clear' %>
<% end %>
</body>
```

We updated our xmlservice controller to handle multiple button names: execute, clear, and refresh.

```
app/controllers/xmlservice_controller.rb:
class XmlserviceController < ApplicationController
  def execute
    if !params[:execute].nil?
      if params[:command].strip == ""
        refresh
      else
        run
      end
    elsif !params[:clear].nil?
      clear
      refresh
    elsif !params[:refresh].nil?
      refresh
    end
  end
def run
  @command =  params[:command]
  ibmx = XMLService::I_DB2.new(@command)
  ibmx.xmlservice
  rows = ibmx.response.output
  @output = rows.inspect
  # add to history
  History.create(:command => @command)
end
def clear
  History.delete_all
end
```

```
def refresh
  redirect_to :back
end
end
```

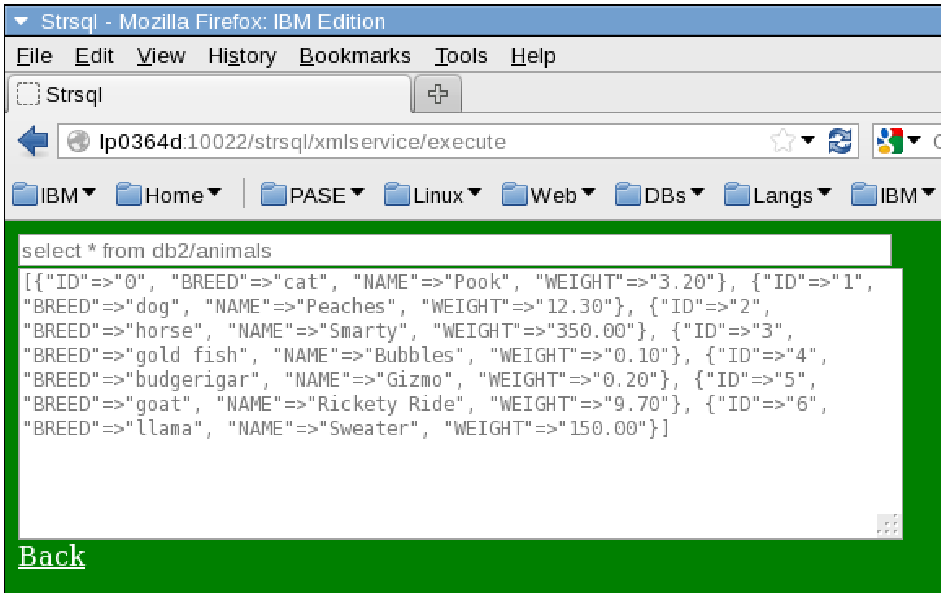
Our `config/routes.rb` remains the same for `post '/xmlservice/execute', to: 'xmlservice#execute'`, simply included for completeness.

```
config/routes.rb:
Strsql::Application.routes.draw do
  # get "welcome/index" -- rails generate controller moved into scope
  scope "/strsql" do
    get "welcome/index"
    post '/xmlservice/execute', to: 'xmlservice#execute'
    root :to => 'welcome#index'
  end
end
```

Step 13. Fancy output display

The tutorial leaves the task of creating a fancier STRSQL output display to the reader, but at least we did go green.

Figure 28. STRSQL simple output display screen


Strsql - Mozilla Firefox: IBM Edition
File Edit View History Bookmarks Tools Help
Strsql
lp0364d:10022/strsql/xmlservice/execute
IBM Home PASE Linux Web DBs Langs IBM

select * from db2/animals

```
[{"ID"=>"0", "BREED"=>"cat", "NAME"=>"Pook", "WEIGHT"=>"3.20"}, {"ID"=>"1",
"BREED"=>"dog", "NAME"=>"Peaches", "WEIGHT"=>"12.30"}, {"ID"=>"2",
"BREED"=>"horse", "NAME"=>"Smarty", "WEIGHT"=>"350.00"}, {"ID"=>"3",
"BREED"=>"gold fish", "NAME"=>"Bubbles", "WEIGHT"=>"0.10"}, {"ID"=>"4",
"BREED"=>"budgerigar", "NAME"=>"Gizmo", "WEIGHT"=>"0.20"}, {"ID"=>"5",
"BREED"=>"goat", "NAME"=>"Rickety Ride", "WEIGHT"=>"9.70"}, {"ID"=>"6",
"BREED"=>"llama", "NAME"=>"Sweater", "WEIGHT"=>"150.00"}]
```

Back

To give you a head start, we changed view execute (erb), adding `:disabled` for no edit allowed, and `link_to "Back"` to return to the welcome screen after running the statement.

```
app/views/xmlservice/execute.html.erb:
<head>
<style>
body { background-color:green; color:white; }
input[type="text"] {width:535px;}
a{color:white;}
</style>
</head>
<body>
<%= text_field_tag "command", @command, disabled:true %>
<br>
<%= text_area_tag 'body', @output, size: "75x10", disabled:true %>
<br>
<%= link_to "Back", "/strsql" %>
</body>
```

Summary

This tutorial used the IBM i command line to edit, test, and deploy a step-by-step Rails web version of IBM i STRSQL using Ruby gems ibm_db gem for DB2 access and xmlservice gem for IBM i access to native objects.

Hopefully, this manual approach provided you a deeper understanding of developing PowerRuby applications. Happy Rails to your IBM i from PowerRuby!

Resources

- PowerRuby
- Ruby on Rails site
- IBM i developerWorks forum