MarkLogic™

USER CONFERENCE 2011

SAN FRANCISCO | APRIL 26 – 29

MarkLogic and Ruby with ActiveDocument

DEVELOPER LOUNGE LAB

Table of Contents

Step 1: Preparation

Unzip the ActiveDocumentLab.zip file to the folder from which you will run the lab exercises. Once you unzip the file you will have two top level directories, DataLoading and MlucActiveDocumentLab along with a single file, dynamic_dispatch.xqy. We will explore the contents of these directories shortly but first let's make sure that your environment is ready to go.

Ruby

These samples were written using Ruby 1.9.2. While they may work with Ruby 1.9.0 I recommend running on the latest build of Ruby. If you need to install Ruby, please follow the instructions specific to your operating system (http://www.ruby-lang.org/en/downloads/).

Gems

In order to run the Rails portion of the lab you will, of course, need to install Rails. The application requires Rails 3.0.7 or greater. If you don't have Rails installed you can get the necessary version by executing the following command:

gem install rails

If you have an older version of Rails installed use this command to update Rails:

gem update rails

Lastly, you will need the ActiveDocument gem. Install it via:

gem install activedocument

That's it! At this point you have Ruby, Rails and ActiveDocument installed and you are ready to go.

Step 2: Install and configure MarkLogic

We're going to need a MarkLogic server installed and a database configured in order to start working with ActiveDocument. Please download and install the latest version of MarkLogic server for your operating system (http://developer.marklogic.com/products).

Now that we have MarkLogic installed and running we need to create a database in which to store our information. We are going to be primarily working with Twitter data, so let's call our database "Tweets". We don't need to configure any special settings or indexes at this point, we just need a default configuration for a database. If you are unsure of how to do that, I recommend the first section of this excellent tutorial: http://developer.marklogic.com/learn/2009-01-get-started-apps.

Lastly, we need to create an HTTP server that points to our newly created Tweets database. For the purposes of this tutorial I recommend you use port 8020, just to be consistent with the examples but you can use any port you like. Again, if you unfamiliar with how to create an HTTP server, the tutorial referenced in the above paragraph will walk you through the necessary steps. Lastly, once you have configured the http server you will need to move the dynamic_dispatch.xqy file that is in the directory where you unzipped MlucActiveDocumentlab.zip into the root directory of your HTTP server.

Step 3: Load Some Tweets

Now that we have everything setup and configured its time for the fun to begin! The first thing we need to do is to get some data into MarkLogic so that we can create a Rails application to search and display that information. We will use ActiveDocument to perform that initial load of data.

Look inside the DataLoading directory that was created when you performed the unzip. You will see two files, TwitterSample.rb and TwitterTracker.rb. These are two stand alone Ruby scripts that leverage ActiveDocument to load Tweets into MarkLogic. TwitterSample.rb uses the Twitter streaming API to grab new Tweets as they are created and then load them into MarkLogic via ActiveDocument. TwitterTracker.rb allows you to capture Tweets that match specific search criteria, such as all Tweets mentioning #MLUC2011.

DISCLAIMER: There is no filtering of Tweets for possible objectionable content!

Before we run both or either of those scripts, we need to edit the config.yml file that is in the same directory as TwitterSample.rb and TwitterTracker.rb. This file contains the configuration information that tells ActiveDocument how to connect to your MarkLogic server. Open the file and replace the default value for user_name and password with the correct values for your MarkLogic server and edit the uri so that it has the same port number you used when you created your HTTP server. Save and close the file when you are done.

You can now run either TwitterTracker.rb or TwitterSample.rb for some period of time to load your database. How long you want to let them is up to you. When you are done loading (or while you are loading) you should feel free to examine these files. You will notice that in order to load data all we have to do is to create a class that extends *ActiveDocument::Base*, create a new instance of that class passing in XML as text and then calling save.

Step 4: Rails

Lets begin by starting the application server. Go to a command prompt and change to a the MlucActiveDocumentLab directory that was in the zip file you extracted. From there start the Rails server with the command “script/rails server”. You can now open a web browser to http://localhost:3000/tweets. Go

ahead and explore the search and delete functions of this very simple web application. When you're done come back and we will look at how it works.

Inside the Application - Configuration

Welcome back! I hope you enjoyed our simple application. Now, lets look at how it works. (Note: in this section all file paths will be relative to the root of the Rails application, MlucActiveDocumentLab.) Look at config/finder_config.yaml. This is the configuration file ActiveDocument uses to connect to the server. How does ActiveDocument know to look for that particular file? Simple. We have to tell it. Look at config/environement/environment.rb where you will see this line:

ActiveDocument::Finder.config "#{Rails.root}/config/finder_config.yml"

That is where we tell ActiveDocument where to find its configuration information. You could initialize ActiveDocument from another location in your application; the important thing is that it must be initialized before you can use any of its functionality.

Search

Now let's look at app/controllers/tweets_controller.rb to see how search works. The *index* function is where search happens and the @*query* variable is where we are storing the text from out search text box. In order to execute the search and get results we just have to call the *search* method on *ActiveDocument::Finder* as shown below:

@results = ActiveDocument::Finder.search(@query, start, 10, @search_options)

Results

Now open app/views/tweets/index.html.haml in order to see how we display the results. Near the top of the file you can easily see how we are displaying some metadata about the search by accessing the @*results* object that we got from calling our *search* method. The @*results* object is a wrapper for all of the search results returned from our search and it contains metadata about our search such as the number of results and how long the search took.

Just slightly further down in index.html.haml you can see where we loop through the @*results* object, extracting the contained result objects into *tweet*s. We use the tweet's uri to create a link to the *show* method of our *TweetController* in order to show the details of each Tweet.

Tweet

Before we look at how we display the details of each tweet, lets take a quick moment to examine our Tweet model. Open app/models/tweet.rb. Here you see that to model a Tweet document stored in MarkLogic all we had to do was to create a class that extends *ActiveDocument::Base*. That's really all there is to it. However, in this class you can see that we added a couple of helper methods around the tweet's uri. We did this because a document's uri is its unique identifier but it doesn't work well as a url parameter (as Rails would like it to)

because it can have the "/" character in it. In order to work around that issue in this case I added the helper methods you see in *tweet.rb*.

Tweet Details

Back in *tweets_controller.rb* examine the *show* method. Here you can see that we are able to load a Tweet from MarkLogic by simply calling the *load* method on the *Tweet* class, passing in the uri of the document that we want to load. We then store that document in the @*tweet* variable for use in app/views/tweets/show.html.haml, which you should open up now.

In this file you will see that it is very easy to access the data contained in a Tweet. All we have to do is to access the elements in the tweet as (potentially nested) properties. Unfortunately, Ruby doesn't allow for the hyphen (-) character in method calls so we are forced into a bit of syntactic ugliness for elements that have hyphens in the name, as seen in this call where we are accessint the screen-name element that is a child to the user element:

```
@tweet.user.screenHYPHENname.text
```

One more note on this subject. You can see that we are also calling a *text* method on these properties, as shown above. If you don't call *text* then you get the actual XML element. In this case we want the text of the element so we call *text* on the element.

Delete

Deleting documents really couldn't be much simpler. All we have to do is to call the delete method on the Tweet class, passing in the uri of the document we want to delete. You can see this in action inside of the TweetController in the *delete* method.

Step 5: Add Facets

Let's add a facet to our application. The screen-name associated with the tweet is a good candidate. The first thing we need to do is to add an element-range index of type string for the local name "screen-name" (no namespace uri) in the MarkLogic admin console for our Tweets database.

Next, we need to configure to facets we want using search_options. Inside of the *set_options* method of the TweetController add the following line at the end of the method:

```
@search_options.range_constraints["User"] = {"element" => "screen-name", "type" => "xs:string", "collation" => "http://marklogic.com/collation/"}
```

That line tells ActiveDocument that there is a range constraint of type xs:string on the screen-name element and that it can be accessed via the User constraint. All constraints are also available as facets and are accessed in our application via this line in the *index* method:

```
@facets = @results.facets
```

We can now view the facets in our index page by adding this code to the top of app/views/tweets/index.html.haml:

```
- content_for(:sidebar) do
 - if @facets
  - @facets.keys.each do |key|
   %h2
    =key.gsub('_', ' ')
   - @facets[key].sort {|a, b| Integer(a[1])<=>Integer(b[1])}.reverse.first(10).each do |entry|
    %p
     = link_to("#{entry[0]} (#{entry[1]})", {:action => "index", :query => @query + " #{key}:\"#{entry[0]}\""})
```

That code displays the facet information and renders them as clickable links. Go ahead and try it. I'll wait… Glad you're back. Let's examine that code for a minute. If the *@facets* object is present the it loops through that object to pull out the facet information. The facets are stored as a hash containing hashes. In the first hash the key is the name of the facet, which we use to display a heading. The value is a hash of the facet value (in our case screen-name) to its frequency count. We just do a quick sort on the count in order to show them in frequency order. That's it!

Step 5: Additional Resources

You can find additional information at the following locations:

- Download MarkLogic Server at http://developer.marklogic.com/products
- Find documentation, tutorials and more at http://developer.marklogic.com/
- ActiveDocument: https://github.com/crichey/ActiveDocument/
- Follow me on Twitter @crichey or contact me directly at Clark.Richey@marklogic.com