Couch DB importing data using ruby

16 September 2014

09:47

Importing Data Into CouchDB Using Ruby

Importing data is a recurring problem that you’ll face no matter what database you end up using.

CouchDB is no exception here.

In this section, we’ll use Ruby to import structured data into our music database.

Through this you’ll see how to perform bulk imports into CouchDB, and it’ll also give us a nice pool of data to work with when we create more advanced views.

We’ll use music data from Jamendo.com, a site devoted to hosting freely licensed music

<https://developer.jamendo.com/en/wiki/NewDatabaseDumps>

Jamendo provides all their artist, album, and track data in a structured XML format, making it ideal for importing into a document- oriented database like CouchDB.

Head over to Jamendo’s NewDatabaseDumps page and download dbdump\_ artistalbumtrack.xml.gz.

The zipped file is only about 15MB.

To parse Jamendo’s XML file, we’ll use the libxml-ruby gem.

Rather than writing our own Ruby-CouchDB driver or issuing HTTP requests directly, we’ll use a popular Ruby gem called couchrest that wraps these calls into a convenient Ruby API.

We’ll be using only a few methods from the API, but if you want to continue using this driver for your own projects, the documentation is quite good.

On the command line, install the necessary gems:

**$ gem install libxml-ruby couchrest**

we’ll use a SAX-style parser to process documents sequentially for insert as they’re streamed in through standard input

See code <http://media.pragprog.com/titles/rwdata/code/couchdb/import_from_jamendo.rb>

1 To kick things off, we bring in the rubygems module and the specific gems that we need.

2 The standard way to use LibXML is by defining a callbacks class. Here we define a JamendoCallbacks class to encapsulate our SAX handlers for various events.

3 The first thing our class does during initialization is connect to our local CouchDB server using the CouchRest API and then create the music database (if it doesn’t exist already). After that, it sets up some instance variables for storing state information during the parse. Note that if you set the @max parameter to nil, all documents will be imported, not just the first 100.

4 Once parsing has started, the on\_start\_element() method will handle any opening tags. Here we watch for certain especially interesting tags like <artist>, <album>, <track>, and <tag>. We specifically ignore certain container elements—<Artists>, <Albums>, <Tracks>, and <Tags>—and treat all others as properties to be set on the nearest container items.

5 Whenever the parser encounters character data, we buffer it to be added as a property to the current container element (the end of @stack).

6 Much of the interesting stuff happens in the on\_end\_element() method. Here, we close out the current container element by popping it off the stack. If the tag closes an <artist> element, we take the opportunity to save off the document in CouchDB with the @db.save\_doc() method. For any container element, we also add a random property containing a freshly generated random number. We’ll use this later when selecting a random track, album, or artist.

7 Ruby’s ARGF stream combines standard input and any files specified on the command line. We feed this into LibXML and specify an instance of our JamendoCallbacks class to handle the tokens—start tags, end tags, and character data—as they’re encountered.

To run the script, pipe the unzipped XML content into the import script:

**$ zcat dbdump\_artistalbumtrack.xml.gz | ruby import\_from\_jamendo.rb** TOTAL: 100 records inserted

When the import has finished, drop back down to the command line, and we’ll see how our views look.

First let’s pull up a few artists.

The limit URL parameter specifies that we want only that number of documents in the response (or less).

**$ curl** [**http://localhost:5984/music/\_design/artists/\_view/by\_name?limit=5**](http://localhost:5984/music/_design/artists/_view/by_name?limit=5)

{"total\_rows":100,"offset":0,"rows":[

{"id":"370255","key":"\"\"ATTIC\"\"","value":"370255"}, {"id":"353262","key":"10centSunday","value":"353262"}, {"id":"367150","key":"abdielyromero","value":"367150"}, {"id":"276","key":"AdHoc","value":"276"}, {"id":"364713","key":"Adversus","value":"364713"}

]}

The previous request started at the very beginning of the list of artists.

To jump to the middle, we can use the startkey parameter:

**$ curl** [**http://localhost:5984/music/\_design/artists/\_view/by\_name?\**](http://localhost:5984/music/_design/artists/_view/by_name?\) **limit=5\&startkey=%22C%22**

{"total\_rows":100,"offset":16,"rows":[

{"id":"340296","key":"CalexB","value":"340296"}, {"id":"353888","key":"carsten may","value":"353888"}, {"id":"272","key":"Chroma","value":"272"}, {"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"}, {"id":"364714","key":"Daringer","value":"364714"}

]}

Previously, we started with artists whose names began with *C*.

Specifying an endkey provides another way to limit the returned content.

Here we specify that we want artists only between *C* and *D*:

**$ curl** [**http://localhost:5984/music/\_design/artists/\_view/by\_name?\**](http://localhost:5984/music/_design/artists/_view/by_name?\) **startkey=%22C%22\&endkey=%22D%22**

{"total\_rows":100,"offset":16,"rows":[

{"id":"340296","key":"CalexB","value":"340296"}, {"id":"353888","key":"carsten may","value":"353888"}, {"id":"272","key":"Chroma","value":"272"}, {"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"} ]}

To get the rows in reverse order, use the descending URL parameter.

Be sure to reverse your startkey and endkey as well.

**$ curl** [**http://localhost:5984/music/\_design/artists/\_view/by\_name?\**](http://localhost:5984/music/_design/artists/_view/by_name?\) **startkey=%22D%22\&endkey=%22C%22\&descending=true**

{"total\_rows":100,

"offset":16,

"rows":[

{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"},  
{"id":"272","key":"Chroma","value":"272"},  
{"id":"353888","key":"carsten may","value":"353888"},  
{"id":"340296","key":"CalexB","value":"340296"}

]}

A number of other URL parameters are available for modifying view requests, but these are the most common and are the ones you’ll reach for most often.

Some of the URL parameters have to do with grouping, which comes from the reducer part of CouchDB mapreduce views. We’ll explore these tomorrow.