Homework 6: Database Application

bearTunes

Due @ 11:59:59 PM on Wednesday, December 6th

Overview

For this assignment, you'll be implementing portions of a database-backed web application using Ruby on Rails. If you've been ignoring the buzz, Ruby on Rails is the hot new web application framework that makes development (relatively) quick and (usually) painless. After spending the first two programming assignments hacking away at the dark underbelly of PostgreSQL, you now get to take a step back and build something useful on top of it. That something is bearTunes, an application to keep track of artists, albums, and songs.

Don't worry, we're not using the same database as the SQL homework ☺

Why Ruby on Rails?

The guiding principle of Ruby on Rails is "convention over configuration." Based on some simple assumptions about naming conventions, it does a lot of the gross administrative junk for you, making application development much faster and more bug-free. It lets you focus on the application logic without having to get bogged down in configuration issues. To further simplify things, the Rails architecture draws clean lines between the different functional pieces of your application architecture: Model, View, and Controller (MVC).

Model

The Model consists of the classes representing your database tables. For each table in your database, there will be a corresponding Model class. You'll never have to directly communicate with your database: all actions are carried out through the Model classes. Here, you can also define foreign key semantics, so that you can ensure referential integrity even when the database chooses not to. Because all accesses to the database go through the Model classes, you can optimize your database purely for speed and let your application make sure the data stays consistent.

View

The View controls the display of your results. In this case, that comes in the form of HTML sprinkled with embedded Ruby code. The design and layout of your web pages will all be specified here. There will be one .rhtml file for each action in the Controller classes.

Controller

The Controller is what contains your application logic. Based on user input, the code in the Controller retrieves and manipulates data from the Model, and then gives it to the View so that it can be displayed. Here is where the necessary SQL commands will be generated, though you'll never have to write it yourself (of course, you can if you want). Each action defined in the Controller classes will compute values (or tuples) and store them in variables, and the corresponding View class can reference those variables and display them to the user.

Ruby

Ruby is the language you'll be using to write the application. The syntax is very simple and easy to read; you should have no problem picking it up. The Model classes are already written for you; your task will include adding to the Controller and View classes, using both Ruby and HTML. Here are a bunch of tutorials and references for the two languages - knowing them both will look great on your resume!

Ruby:
http://api.rubyonrails.org/
http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html?page=1
http://www.tutorialized.com/tutorial/Understanding-Your-Code/16240
http://www.tutorialized.com/tutorial/Beginning-Relationships-in-Rails/15353
http://www.tutorialized.com/tutorial/Learning-Ruby/11956

HTML:
http://www.w3schools.com/html/html_intro.asp
http://www.utexas.edu/learn/html/
http://www.davesite.com/webstation/html/
http://www.mcli.dist.maricopa.edu/tut/lessons.html

Setup

Before you get to work, you have to set up your environment. You should be able to do this project on any of the Solaris machines (the "shapes": rhombus, torus, sphere, etc). For this project you'll be running your own copy of both the database and the webserver (yay!). First, call the following command:

```
setuphw6
```

This will create and fill the directory `~/hw6`, so if you already have something by that name then move it first! When that finishes, call the following newly created script:

```
~/hw6/beartunes/runServer
```

This script will start up the database and the webserver. *Don't run this script in the background!* If you need access to the terminal, open another one. When you're finished, simply press CTRL-C to close down the webserver.

Accessing bearTunes

When you start the webserver using `runServer`, you will notice the following block of text telling you which port the webserver is running on:

```
---------------------------------
+ RUNNING SERVER ON PORT: ##### +
---------------------------------
```

If you're sitting at the machine on which the server is running, you can simply point your favorite browser to `http://localhost:#####` and see bearTunes in action. However, most of you will be working remotely, so you'll have to set up a secure tunnel to view the application. If you're logging in through SSH Secure Shell (which you can download for free from http://software.berkeley.edu/), here are the steps to setting up a secure tunnel:

1. Select File > Profiles > Add Profile…
2. Give it a name and click Add to Profiles
3. Select File > Profiles > Edit Profiles…
4. Find the name on the left and select it
5. Click on the Tunneling tab
6. Select Outgoing tunnels, and click Add…
7. Set Display Name to anything you want
8. Set Type to TCP
9. Set Listen Port to some random number in the 2000-4000 range
10. Check the box by Allow Local Connections Only
11. Set Destination Host to the machine where the webserver is running (i.e. rhombus.cs.berkeley.edu)
12. Set Destination Port to the number that was displayed when you started the webserver

Now close down the webserver with `CTRL-C` and log out. Now, to log back in again, select File > Profiles and find the profile you just created. Log in and start up the webserver again by running the script `bearTunes/runServer`. On your local machine, open up a web browser and point to the following URL:

```
http://localhost:{PORT}
```

Where `{PORT}` is the number you filled in under Listen Port when setting up the tunnel. If everything went according to plan, you should see the front page of bearTunes!

Database schema

The following is the schema of the underlying database. In all relations, `id` is the primary key. No foreign key constraints are specified by the database management system (though the intended relationships are obvious by the naming convention). Instead, the constraints are specified in the application (look at the files in `beartunes/app/model` to see how that's done).

```
artists(id, name, city, bio)
albums (id, artist_id, name, year, genre_id)
tracks (id, album_id, name, number, length)
genres (id, name)
```

Your assignment

First things first: populate your database! The more you have in there, the easier it will be to test your changes and the better your analysis will be at the end! There's no link to enter new songs (that is, until you finish task 1), but there's no shortage of albums, artists, and genres to put in for now.

After taking some time to play around with bearTunes, you'll now get to add some functionality to it. The following are the classes you'll need to either create or extend, which potentially includes changes to both the Controller and View. You'll find all of the code you need in the `beartunes/app` directory (use the existing files for reference). You should only make changes to the files in the `controllers` and `views` directories. For example, to modify the artist/list action, you would have to make changes to the files controllers/artist and views/artist/Unless otherwise specified, all orderings should be in *ascending order*. Also, any pages we ask you to provide a link to already exist – unless otherwise specified, you should not be creating any additional files!

1. track/list

You may have noticed the absence of anything useful on the page when you try to browse by songs. You're going to fix that. For each Track in the database, list the relevant Artist name and Track name. They should be ordered by (Track name, Artist name). When clicked on, the Artist name should lead to the page showing the details of that particular Artist. Similarly, the Track name should lead to the page showing the details of *the relevant Album*. Finally, somewhere at the bottom, put a link to the page where you can add a new Track to the database.

2. genre/show

The individual Genre pages, as you can tell, could use a little work. For all Albums *under the relevant genre*, list the same information in the same way as in albums/list. The same ordering should apply, and the same links should exist. Also, at the bottom, provide a link to delete the Genre.

3. artist/show

Currently, the individual Artist pages display only the information contained in the Artists table. You will add information regarding the Albums recorded by that particular Artist. For each Album they've recorded, display the Album's name and the year it was released. When clicked on, the Album name should lead to the page showing the details of that Album. The Albums should be ordered by year.

4. album/show

The individual Album pages currently display information from multiple tables, but they're still missing some crucial information: Track listings. For all Tracks on the relevant Album, print the Track number, name, and length, and a link to the page where you can edit the individual Track. These should be ordered by Track number.

5. track/new_by_album (new action)

Create a new action track/new_by_album, which is the same as track/new except the Album is decided by a passed-in argument instead of a drop-down list. Provide a link to this new action at the bottom of the album/show page, passing it the Album id so that a Track can be added to that particular Album. This will require the creation of a new file, `views/track/new_by_album.rhtml`.

6. album/new_by_artist (new action)

Create a new action album/new_by_artist, which is the same as album/new except the Artist is decided by a passed-in argument instead of a drop-down list. Provide a link to this new action at the bottom of the artist/show page, passing it the Artist id so that an Album can be added to that particular Artist. This will require the creation of a new file, `views/album/new_by_artist.rhtml`.

Analysis

Because Ruby on Rails hides any real work from the user and auto-generates SQL queries, it doesn't always come up with the most optimal ones. If you look at the Rails log, you can see the SQL queries it's issuing. After spending some time playing with bearTunes, examine the following file:

```
~/hw6/beartunes/log/development.log
```

You'll see a chunk of text for every action that was carried out, starting with a line that looks like this:

```
Processing TableController#action …
```

For each action that required a trip to the database, you can see the actual SQL query that was generated. There are likely to be some queries that could have been written to be more efficient. There may even be some cases where multiple queries were issued for the

same action, even though it could have been accomplished with just one! Fortunately, Ruby gives you the option of explicitly writing SQL queries to be issued from the Controller class. For all of the actions in the list below, find one example of a SELECT query (or set of SELECT queries) that was issued. Record these queries in a file called `ANALYSIS`. Did any actions cause multiple queries to be called?

Actions to analyze:
artist/list
artist/show
album/list
album/edit
album/show
album/new
genre/list
genre/show
track/list
track/edit
track/new

The log will also tell you the total time spent on each request, breaking it down by time spent getting tuples from the database and time spent rendering the results. In your `ANALYSIS` file, along with a sample query for each specified action, provide the total time spent on a request and the percent spent in database retrieval and rendering. Try to look at a few requests per action, to get average times – the more samples you look at, the more realistic the numbers will be.

Once you've recorded the types of queries and time spent for each action, answer the following questions in your `ANAYSIS` file:

1. Which action was the fastest? Why?

2. Which action took the longest? Why? Could this have been improved if you had hand-picked the query?

3. Given what you've seen, would you recommend any changes to the database schema that would make things more efficient?

Remember, the time differences may be pretty small, but you're working with an extremely small dataset. When you're running queries over terabytes of data, even the tiniest improvements can make a huge difference!

Submission

Submit all files in the following directories:

```
beartunes/app/controllers/
beartunes/app/views/
```

This includes the files that were originally in the two directories and the two new files that you created in tasks 5 and 6. Also, submit a file `ANALYSIS` with a sample query and latency times for each action, along with your answers to the three questions.

As usual, submit a `README` file that contains a line of text for each member of your team, in the following format:

```
full name, class account
```

Also include a line to tell us how many slip days (if any) you are using for this assignment, and how many each team member has left.