HW 2: more Ruby, and some Rails

In this homework you will clone a GitHub repo containing an existing simple Rails app, add a feature to the app, and deploy the result publicly on the Heroku platform. We will run live integration tests against your deployed version.

General advice: This homework involves modifying RottenPotatoes in various ways. Git is your friend: ***commit frequently*** in case you inadvertently break something that was working before! That way you can always back up to an earlier revision, or just visually compare what changed in each file since your last "good" commit.

Remember: Commit early and often!

1. deploy & enhance RottenPotatoes

> You may work in pairs on parts 1-3 as long as ***each student deploys the result separately on her own Heroku account.*** The changes you must make in parts 1-3 are cumulative, so when all done, you should deploy your working version to Heroku. We will test the features in each part separately, so you WILL get partial credit if you have a subset of the features working.

a) Add some movies to RottenPotatoes, and deploy it to the world (sort of)

We have a version of RottenPotatoes that has had some slight modifications for successful deployment on Heroku, and to which we've added a handful more movies to make things interesting.

Check it out to your VM by doing:

```
git clone git://github.com/saasbook/hw2_rottenpotatoes.git
cd hw2_rottenpotatoes
```

You should install gems with `bundle install --without production` to ignore the PostgreSQL gem for your local installation, since that gem will cause problems if you're using a development environment without PostgreSQL installed. Verify that you can successfully run the app using `rails server` and interact with it via a Web browser, as described in the book and in lecture.

Note that we provided a migration that adds a bunch more movies to the database. Take a look at the code in db/migrate/ to review how these work.

Once the above is working on your development computer, it's time to deploy. (Appendix A in

the textbook has more information about this procedure, and Heroku has detailed help pages as well.) Create a free Heroku.com account if you haven't already, and deploy RottenPotatoes there. (Note that your app's name, *something*.herokuapp.com, must be unique among Heroku apps; therefore it's unlikely that the name "rottenpotatoes" will be available, so you can either choose a different name or just keep the default name Heroku chooses for you whenever you create a new app.)

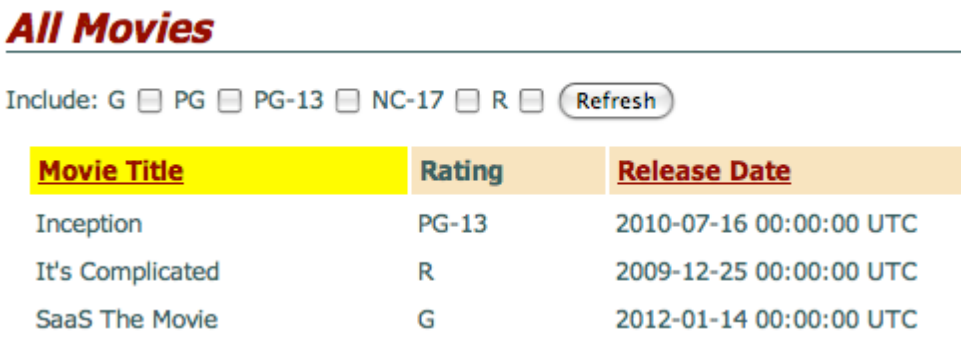
Since this is the first deployment of this app on Heroku, its database will be empty. To fix this, **after** you've pushed your app to Heroku, run **`heroku run rake db:migrate`** to apply ***all*** RottenPotatoes migrations--the initial one that creates the Movies table (which had already been applied when you got your VM) and the one we provided above that adds more movies.

Visit your site at *something*.herokuapp.com/movies to verify that it's working.

b) RottenPotatoes enhancement #1: Sorting the list of all movies.

Enhance RP in the following way:

- On the list of all movies page, the column headings for 'title' and 'release date' for a movie should be clickable links. Clicking one of them should cause the list to be reloaded but sorted in ascending order on that column. For example, clicking the 'release date' column heading should redisplay the list of movies with the earliest-released movies first; clicking the 'title' field should list the movies alphabetically by title. (For movies whose names begin with non-letters, the sort order should match the behavior of `String#<=>`.)
- **IMPORTANT for grading purposes**:
  - The link (that is, the `<a>` tag) for sorting by 'title' should have the HTML element id `title_header`.
  - The link for sorting by 'release date' should have the HTML element id `release_date_header`.
  - The table containing the list of movies should have the HTML element id `movies` (this has already been set for you by existing code).
- When the listing page is redisplayed with sorting-on-a-column enabled, the column header that was selected for sorting should appear with a yellow background, as shown below. You should do this by setting controller variables that are used to conditionally set the CSS class of the appropriate table heading to hilite, and pasting this simple CSS into RottenPotatoes' **`app/assets/stylesheets/application.css`** file.



Here are some hints and caveats as you do this part:

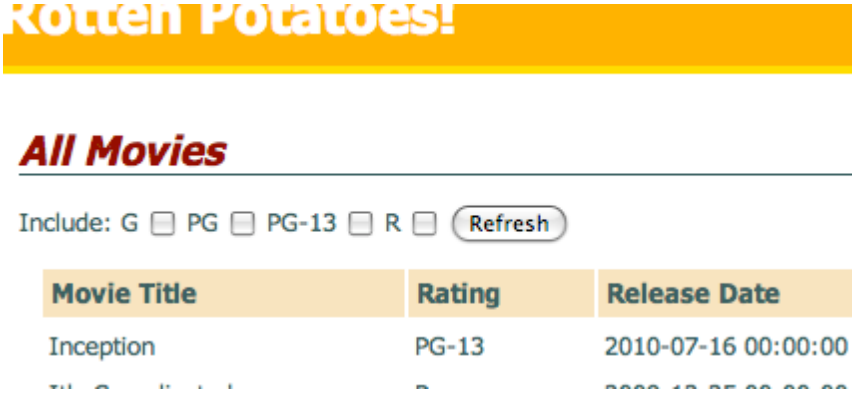
- The current RottenPotatoes views use the Rails-provided "resourceful routes" helper

`movies_path` to generate the correct URI for the movies index page. You may find it helpful to know that if you pass this helper method a hash of additional parameters, those parameters will be parsed by Rails and available in the `params[]` hash.

- Databases are pretty good at returning collections of rows in sorted order according to one or more attributes. Before you rush to sort the collection returned from the database, look at the documentation for the ActiveRecord `find` and `all` methods and see if you can get the database to do the work for you.
- Don't put code in your views! The view shouldn't have to sort the collection itself—its job is just to show stuff. The controller should spoon-feed the view exactly what is to be displayed.

2. RottenPotatoes enhancement #2: filter the list of movies

Enhance RottenPotatoes as follows. At the top of the All Movies listing, add some checkboxes that allow the user to filter the list to show only movies with certain MPAA ratings:



When the Refresh button is pressed, the list of movies is redisplayed showing only those movies whose ratings were checked.

This will require a couple of pieces of code. We have provided the code that generates the checkboxes form, which you can include in the index.html.haml template, here on Pastebin. **BUT**, you have to do a bit of work to use it: our code expects the variable `@all_ratings` to be an enumerable collection of all possible values of a movie rating, such as `['G','PG','PG-13','R']`. The controller method needs to set up this variable. And since the possible values of movie ratings are really the responsibility of the Movie model, it's best if the controller sets this variable by consulting the Model. Create a class method of Movie that returns an appropriate value for this collection.

You will also need code that figures out (i) how to figure out which boxes the user checked and (ii) how to restrict the database query based on that result.

Regarding (i), try viewing the source of the movie listings with the checkbox form, and you'll see

that the checkboxes have field names like **`ratings[G], ratings[PG],`** etc. This trick will cause Rails to aggregate the values into a single hash called **`ratings,`** whose keys will be the names of the ***checked boxes only*** and whose values will be the value attribute of the checkbox (which is "1" by default, since we didn't specify another value when calling the **`check_box_tag`** helper). That is, if the user checks the 'G' and 'R' boxes, **`params`** will include as one if its values **`:ratings=>{"G"=>"1", "R"=>"1"}`** . Check out the **`Hash`** documentation for an easy way to grab just the keys of a hash, since we don't care about the values in this case.

Regarding (ii), you'll probably end up replacing **`Movie.all`** with **`Movie.find`**, which has various options to help you restrict the database query.

Two caveats:

- Make sure that you don't break the sorted-column functionality you added in part 3b! That is, sorting by column headers should still work, and if the user then clicks the "Movie Title" column header to sort by movie title, the displayed results should both be sorted and be limited by the Ratings checkboxes.
- If the user checks (say) 'G' and 'PG' and then redisplays the list, the checkboxes that were used to filter the output should appear checked when the list is redisplayed. This will require you to modify the checkbox form slightly from the version we provided above.
- Don't put code in your views! Set up some kind of instance variable in the controller that remembers which ratings were actually used to do the filtering, and make that variable available to the view so that the appropriate boxes can be pre-checked when the index view is reloaded.
- Update: Make sure that your form elements have the following ids. The submit button for filtering by ratings should have an HTML element id of `ratings_submit`. Each checkbox should have an HTML element id of `ratings_#{rating}`, where the interpolated `rating` should be the rating itself, such as “PG-13”, “G”. An example of an id for the checkbox for PG-13 is `ratings_PG-13`.

3. RottenPotatoes enhancement #3: remember the settings

OK, so the user can now click on the Title or Release Date headings and see movies sorted by those columns, and can additionally use the checkboxes to restrict the listing to movies with certain ratings only. And we have preserved RESTfulness, because the URI itself always contains the parameters that will control sorting and filtering.

The last step is to remember these settings. That is, if the user has selected any combination of column sorting and restrict-by-rating constraints, and then the user clicks to see the details of one of the movies (for example), when she clicks the Back to Movie List on the detail page, the movie listing should "remember" the user's sorting and filtering settings from before.

(Clicking away from the list to see the details of a movie is only one example; the settings

should be remembered regardless what actions the user takes, so that any time she visits the index page, the settings are correctly reinstated.)

The best way to do the "remembering" will be to use the `session[]` hash. The session is like the `flash[]`, except that once you set something in the `session[]` it is remembered "forever" until you nuke the session with `session.clear` or selectively delete things from it with `session.delete(:some_key)`. That way, in the index method, you can selectively apply the settings from the `session[]` even if the incoming URI doesn't have the appropriate `params[]` set.

However, there are two caveats!

- If the user explicitly includes new sorting/filtering settings in `params[]`, the session should not override them. On the contrary, the new settings should be remembered in the session.
- To be RESTful, we want to preserve the property that a URI that results in a sorted/filtered view always contains the corresponding sorting/filtering parameters. Therefore, if you find that the incoming URI is lacking the right `params[]` and you're forced to fill them in from the `session[]`, the RESTful thing to do is to `redirect_to` the new URI containing the appropriate parameters. There is an important corner case to keep in mind here, though: if the previous action had placed a message in the `flash[]` to display after a redirect to the movies page, your additional redirect will delete that message and it will never appear, since the `flash[]` only survives across a single redirect. To fix this, use `flash.keep` right before your additional redirect.

DON'T FORGET TO DEPLOY:
Deploying your finished app to Heroku by the homework deadline is part of the grading process. Even if you have code checked in that works properly, you still need to also deploy it to Heroku to get full credit.

There may ALSO be a requirement to make your code available on GitHub by adding a TA as a collaborator on a private account; stay tuned.