CSCI 3308
Fall 2013
Assignment 3
Adapted from Assignment 2 from CS 169.1 on EdX

Assignment 3: Introduction to Rails

GOALS:

In this assignment you will become comfortable with the develop-deploy cycle that is available to you through working in rails with Heroku as a deployment environment. You will also become more familiar with some of the most commonly used features of RubyonRails and learn to use them to make your Rotten Potatoes app function more like a real web site. You will gain a deeper understanding of the MVC architecture that underpins the rails environment and appreciate how learning the conventions and respecting "convention over configuration" will work in your favor when developing most RubyonRails apps.

HOW TO PROCEED:

In this homework you will clone a GitHub repository ("repo") containing an existing Rails app called "RottenPotatoes", add a feature to the app, and deploy the result publicly on the Heroku platform. During interview grading you will demonstrate your app running on Heroku.

General advice

Parts of this assignment are pretty tricky but none involve a significant amount of coding by you so if you find yourself writing many lines of code, assume there is a better approach and start thinking about what it might be. Make sure to have read through Chapter 4 of the text and watched the lectures on CS169.1 on EdX. Then think about how the part of rails work together This homework involves modifying RottenPotatoes in various ways. Git (version control) is your friend. *Commit frequently* in case you inadvertently break something that was working before! This way, you can always back up to an earlier revision and/or easily compare what changed in each file since your last "good" commit.

If you have access to the SaaS textbook and have never used Rails before we strongly recommend that you work step by step through the code in chapter 4 of the textbook. We also have a series of free screencasts that walk through the code in that chapter, which you can view even if you don't have the textbook.

Important for grading purposes

- Do not use JavaScript to implement any of the functionality required for this homework.

SUBMISSION:

For each part of this homework, cut and paste the url of your Heroku deployment into the submission textbox on moodle. Each url will look like this: `http://your-app-name.herokuapp.com`

Also submit the URL to your public repository on GitHub which contains your source code.

GRADING:

The assignment has three parts. Part 1 is worth 25 points, Part 2 is worth 15 points, and Part 3 is worth 10 points. Your score for each part will be a combination of your submission and your responses during interview grading. Your submitted code is worth 50 points. During interview grading you will earn points toward these scores based on your Rotten Potatoes demonstration and your ability to discuss how your code works on a line-by-line basis. Deductions will be taken for tardiness. Any interview that is missed without prior approval by one of the instructors will be given a zero.

Don't forgot to include at the top of your text box an references you used to come up with your solution – that would be any websites, book, or people you spent significant time working with.

Feel free to watch this helpful video (you do not need to tell us about having used this or any other book/course resource as a reference):

https://www.youtube.com/watch?feature=player_embedded&v=B7aiR7Ph3tk

Deploy RottenPotatoes and Enhancement #1

(25 points possible)

Part A —Add some movies to RottenPotatoes, and deploy it to the world **(10 pts)**

We have a version of RottenPotatoes that has had some slight modifications for successful deployment on Heroku, and to which we've added an additional handful of movies to make things more interesting. Check out the app on your VM (or other development environment) by running the following commands:

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

The first command will pull down the latest version of the RottenPotatoes starter code, and the second will move you into the directory which now contains the code.

The next step is to install the necessary gems for the app (listed in the Gemfile). Do this by running

```
bundle install --without production
```

The `--without production` part of the command causes the installer 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).

Note that we provided a seed file that seeds the database with a bunch of movies. Take a look at the code in `db/seeds.rb` to review how these work. When you're ready, run

```
bundle exec rake db:migrate
```

to apply all RottenPotatoes migrations to your local database. We use 'bundle exec' to ensure that rake uses the gems specified in the Gemfile. Then run

```
bundle exec rake db:seed
```

to seed your local database with the movies from the seed file.

Verify that you can successfully run the app by using the command

```
rails server
```

If this works properly, you should be able to interact with RottenPotatoes via a web browser as described in the book and in lecture. Check that you can see a list of all the movies when you access 'localhost:3000/movies'.

Once the above steps have been verified on your development computer, it's time to deploy on Heroku. Create a free Heroku.com account if you haven't already, and deploy RottenPotatoes there. Appendix A in the textbook has more information about this procedure, and Heroku has detailed help pages as well. (Note that your app's name, *something.herokuapp.com*, must be unique among Heroku apps. It's therfore unlikely that the name "rottenpotatoes" will be available. You must either choose a different name or you may just keep the default name Heroku chooses for you when 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. In this case, we only have one migration file that creates the `Movies` table (which had already been applied when you got your VM). You will also need to run

```
heroku run rake db:seed
```

to seed your Heroku app's database with the movies we've created in `seeds.rb`.

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

***Part B — RottenPotatoes Enhancement #1: Sorting the list of all movies* (15 pts)**

You will enhance RottenPotatoes in the following ways:

- On the "All Movies" page, the column headings for "Movie 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. Namely,
    - clicking the "Movie Title" column should list the movies alphabetically by title (for movies whose names begin with non-letters, the sort order should match the behavior of `String#<=>`), and
    - clicking the "Release Date" column heading should redisplay the list of movies with the earliest-released movies first.
- 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 snippet into RottenPotatoes's `app/assets/stylesheets/application.css` file.

Hints and advice

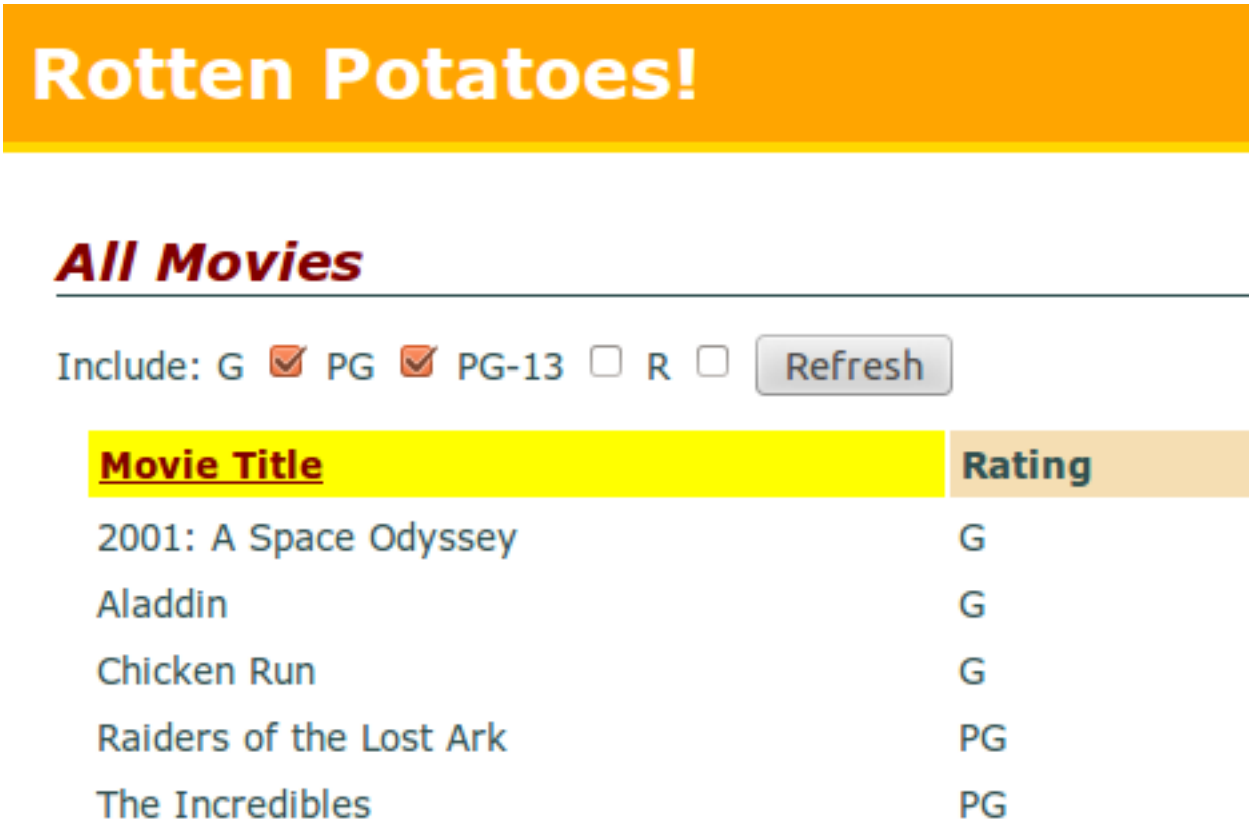
- The current RottenPotatoes views use the Rails-provided "resourceful routes" helper `movies_path` to generate the correct URI for the movies index page. You may find it helpful to

know that if you pass this helper method a hash of additional parameters, those parameters will be parsed by Rails and become 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 instead.
- 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.

Submission Directions

As noted above, you must submit add a single line with the URL of your heroku deployment. The url will look like this: `http://your-app-name.herokuapp.com` `to your moodle submit box.`

- NOTE ON THE DIAGRAM BELOW: You will add the "Include" selector to your app in the next part of the assignment.



Important for grading purposes

- The link (that is, the `<a>` tag) for sorting by "Movie Title" should have the HTML element id `title_header`.

RottenPotatoes Enhancement #2: Filter the List of Movies

(15 points possible)

In this portion of the assignment, you will creat functionality within RottenPotatoes that allows the user to filter movies by MPAA rating. The filter will be controlled by checkboxes at the top of the "All Movies" listing:

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. Code has been provided that generates the checkboxes form, which you can include in the `index.html.haml` template, here on Pastebin. *However*, you have to do a bit of work to use it: the 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 to:

(i) figure out which boxes the user checked and
(ii) 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.

Specific Requirements

- Make sure that you don't break the sorted-column functionality you added in part 1B! 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 provided above.
- The first time the user visits the page, all checkboxes should be checked by default (so the user will see all movies). For now, ignore the case when the user unchecks all checkboxes—you will get to this in the next part.

Hints and advice

- 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.

As described above, you must submit a text file, containing a single line with the url of your heroku deployment. The url will look like this: `http://your-app-name.herokuapp.com`

RottenPotatoes Enhancement #3: Remember the Settings

(10 points possible)

Now that you have implemented parts 1 and 2, the user can click on the "Movie 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. Moreover, 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 clicks to see the details of one of the movies (for example), then when she clicks "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 of 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.

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.

Specific Requirements:

- If the user explicitly includes new sorting/filtering settings in `params[]`, the values stored in the `session` should not override them. On the contrary, the new settings should be remembered in the session.
- If a user unchecks all checkboxes, use the settings stored in the `session[]` hash (it doesn't make sense for a user to uncheck all the boxes).

As noted above, you must add a single line to your moodle submission box with the Heroku URL that will look like this: `http://your-app-name.herokuapp.com`