In Homework 2 for Ruby on Rails SaaS, we were introduced to Rails. We cloned a GitHub repository (“repo”) containing an existing Rails app called “RottenPotatoes”. We added a feature to an app and deployed the result publicly on the Heroku platform. The professors ran live integration tests against our deployed version.

They recommended that we use Git version control and to commit frequently in case we inadvertently break something that was working before. In this way, we could always back up to an earlier revision and/or easily compare what changed in each file since your last “good” commit.

The professors said that for those in the class who know Javascript, it was not allowed for implementing any of the functionality required for this homework.

For each part of this homework, we submitted a text file containing a single line with the URL of your Heroku deployment. The URL's looked something like: [http://your-app-name.herokuapp.com](http://your-app-name.herokuapp.com/).

We first had to deploy RottenPotatoes and add enhancement #1 which was part A, adding some movies to RottenPotatoes. The professors had a version of RottenPotatoes that had some slight modifications for successful deployment on Heroku, and to which they added a handful of movies to make things more interesting. They asked that we check out the app first on our SaasBook Virtual Machine, a Ubuntu Linux Virtual Machine with the Saas class software installed already on it, running on Oracle's Virtual Box. We used the following commands:

git clone git://github.com/saasbook/hw2\_rottenpotatoes.git

cd hw2\_rottenpotatoes

The first command pulled down the latest version of the RottenPotatoes starter code, and the second moved us to the directory that contained the code.

The next step installs the gems for the app. We did 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 they provided a seed file that seeds the database with a bunch of movies. We took a look at the code in: db/seed.rb to review how these work. We then ran:

rake <db:migrate>

to apply all RottenPotatoes migrations to our local database. We then ran:

rake <db:seed>

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

We verified that we could 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 which is running the client part of the app. We checked that we could see a list of all movies when we accessed 'localhost:300/movies'.

Once the above steps have been verified on our development computer, it was time to deploy on Heroku. We created a free Heroku account and delployed RottenPotatoes there. We had to choose a new unique name and the procedures to do this were in an appendix in the class textbook.

Since this was our first deployment of this app on Heroku, its database will be empty. To fix this after you've pushed your app to Heroku, we ran:

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 we received our SaasBook VM). We had to run:

heroku run rake <db:seed>

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

We then visited our app's website: your-app-name.herokuapp.com/movies to verify it was working.

Part B was where we applied the first enhancement which was to sort the list of movies.

We enhanced 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.

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. We did this by setting controller variables that are used to conditionally set the CSS class of the appropriate table heading to “hilite”, and pasting a simple CSS snippet given to us into RottenPotatoe's app/assets/stylesheets/application.css file. For grading purposes, it was important that the link (<a> tag) for sorting by “Movie Title” should have the HTML element ID “title\_header”, that the link for sorting by “Release Date” should have the HTML element id “release\_date\_header”, and the table containing the list of movies should have the HTML element id “movies”.

Hints and advice given to us by the professors: The current RottenPotatoes views use the Rails-provided “resourceful routes” helper “movies\_path” to generate the correct URI for the movies index page. They said we may find it helpful to know that if we pass this helper method a hash of additional parameters, those parameters will be parsed by Rails and become available in the params[] hash. They also said that databases are pretty good at returning collections of rows in sorted order according to one or more attributes. They recommended that before we rush to sort the collection returned from the database, to 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. They asked that we did not put code in our views. The view should not have to sort the collection itself; it's job is just to show stuff. The controller should spoon-feed the view what is to be displayed.

RottenPotatoes Enhancement #2 was to filter the list of movies. In this part of the assignment, we were creating functionality within RottenPotatoes that allows the user to filter movies by MPAA rating. The filter will be controlled by checkboxes at the top of “All Movies” listing. When the “Refresh” button is pressed, the list of movies is redisplayed showing only those movies whose ratings were checked. This required a couple pieces of code. We were provided with the code that generates the checkboxes form, which you can include in the index.html.haml template.

However, we had to do a bit of work to use it. Their code expects @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.

(Recall that since models deal with the app's data, they contain code that communicates with the storage tier or database.) We had to create a method of Movie that returns an appropriate value for this collection.

We also needed code that figured out (i) how to figure out which boxes the user checked and (ii) how to restrict the database query based on the result.

Regarding (i), we were recommended to view the movie listing with the checkbox form, and see that the checkboxes have field names like ratings[G], ratings[PG], etc. This trick caused Rails to aggregate the values into a single hash called ratings, whose keys are the names of the *checked boxes only* and whose values will be the value attribute of the check box which is “1” by default, since we did not specify another value when calling the check\_box\_tag helper function. That is, if we check the boxes 'G' and 'R', params will include as one of its values :ratings => {“G” => 1, “R” => 1}. For part (ii), we were told that Movie.find may be more useful than Movie.all, since Movie.find had various ways to restrict the query.

We had to make sure that we did not break the sorted column functionality we had added in part 1B. So sorting by column headers had to still work, and if the user clicks the “Movie Title” column header to sort by movie title, the displayed results had to be both sorted and be limited by the Ratings check boxes.

If the user checks say G and PG and then redisplays the list, the check boxes that were used to filter the output should appear checked when the list is redisplayed. We had to modify the check box form slightly from the version provided.

The first time a user visits a page, all check boxes should be checked by default, so the user will see all movies. We were told that for now, we should ignore the case when the user unchecks all check boxes, and we will get to this later.

For grading purposes, our “form tag” had to have the id “ratings\_form”. The form submit button for filtering by ratings should have an HTML element id of “ratings\_submit”. Each check box had to have an HTML element id of ratings\_#{rating}, where interpolated “rating” should be the rating itself, such as “PG-13”, “G”. For instance, the id for the check box for “PG-13” should be “ratings\_PG-13”.

We were given hints and advice to: Don't put code in our views! Set up some kind of instance variable in the controller that remembers which ratings were actually used to do filtering, and make that variable available to the view so that the appropriate boxes can be pre-checked when the index view is reloaded.

RottenPotatoes Enhancement #3: Remember the settings

Now that parts 1 and 2 have been implemented, the user can click on the “Movie Title” or “Release Date” headings and see movies sorted by those columns, and can additionally use check boxes to restrict the listing to movies with certain ratings only. We also had to preserve RESTfulness, because the URI itself always contained the parameters that control sorting and filtering. So the last step is to remember the 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, then when she clicks “Back to the 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” or 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 had to preserve the property that a URI that results in a sorted/filtered view always contains the corresponding sorting/filtering parameters. Therefore, if we found 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. Redirect\_to causes a controller action to end not by rendering a view, but by restarting a whole new request to a different action, in this case, a new request with the parameters filled in with what was remembered by the session[]. The important corner case to keep in mind here was: if the previous action had placed a message in 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 flash[] only survives across a single redirect. To fix this we had to use:

flash.keep right before your additional redirect.

Our specific requirements were: If the user explicitly includes new sorting/filtering settings in params[], the values stored in 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, since it does not make sense for a user to uncheck all the boxes.