**Modifying RottenPotatoes**

**Rails Intro: add features to RottenPotatoes**

In this homework you will add a feature to an existing simple Rails app and deploy the result publicly on the Heroku cloud hosting service. 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!**

**Assignment Contents**

* **Part 0 (A):**[**Preparation: get RottenPotatoes running locally**](https://github.com/UCCSCS3300/hw-rails-intro/blob/master/docs/part_0_A.md)

Before you start, make sure your local workspace is synched with your class private git account. From the homework directory, do a git pull origin main and confirm there are no errors.

The actual RottenPotatoes starter app you will use is in another public repo: [UCCSCS3300/rottenpotatoes-rails-intro](https://github.com/UCCSCS3300/rottenpotatoes-rails-intro). Clone that repo to your homework directory and remove the .git directory inside of the rottenpotatoes-rails-intro folder.

$ cd /root/environment/homework

$ git clone https://github.com/UCCSCS3300/rottenpotatoes-rails-intro.git

$ rm -rf /root/environment/homework/rottenpotatoes-rails-intro/.git

Whenever you start working on a Rails project, the first thing you should do is to run Bundler, to make sure all the app's gems are installed. Switch to the app's root directory (in this case rottenpotatoes-rails-intro) and run bundle install --without production (you only need to specify --without production the first time, as this setting will be remembered on future runs of Bundler for this project).

Finally, get the local database created:

$ rake db:migrate

**Self Check Question:** How does Rails decide where and how to create the development database? (Hint: check the db and config subdirectories)  
**Self Check Question:** What tables got created by the migrations?

Now insert "seed data" into the database--initial data items that the app needs to run:

$ rake db:seed

**Self Check Question:** What seed data was inserted and where was it specified? (Hint: rake -T db:seed explains the seed task; rake -T explains other available Rake tasks)

At this point you should be able to run the app locally (rails server -b 0.0.0.0 -p 8080) and navigate to http://YOURUCCSUSERNAME-18-cs4300and5300-1-uccs.devedu.io:8080/movies in your browser.

Note: If you stop the server by hitting control-C, you will no longer be able to visit the RottenPotatoes site. Start the server again by repeating the last command.

Welcome to RottenPotatoes!

Lastly, back up the working framework up in github.

cd /root/environment/homework

git add rottenpotatoes-rails-intro/

git status //Make sure all the new files are tracked AND on stage for being committed!!

git commit -m “Initial HW2"

git push origin main

* **Part 0 (B):**[**Preparation: deploy to Heroku**](https://github.com/UCCSCS3300/hw-rails-intro/blob/master/docs/part_0_B.md)

First, make sure you are in the rottenpotatoes-rails-intro directory. Run git init. Note that this .git info will ONLY be linked to Heroku- NOT your class repository.

If you have deployed to Heroku before, just create a new app container with heroku create. If this is your first time deploying to Heroku, you will need to do two things. First, sign up for a free [Heroku account](http://heroku.com/). Then set up ssh keys to securely communicate with Heroku for app deployments. Next install Heroku. The four basic commands you need are the following, but see the [Heroku page](https://devcenter.heroku.com/articles/heroku-cli) for more details.

$ ssh-keygen -t rsa

$ curl https://cli-assets.heroku.com/install.sh | sh

$ heroku login -i

$ heroku keys:add

Once your keys are set up (a one-time process), you should be able to create an "app container" on Heroku into which you'll deploy RottenPotatoes:

$ cd /root/environment/homework/rottenpotatoes-rails-intro

$ git init

$ heroku create

Heroku will assign your app a whimsical name such as luminous-coconut-237; once your app is deployed, you would access it at http://luminous-coconut-237.herokuapp.com. You can login to the Heroku website if you want to change the name of your app.

Confirm the new git associations: git remote -v You should ONLY see fetch and push associated with heroku. These links automatically are built when you ran heroku create.

$ git add \*

$ git commit -m ""

Note that running git commit will open a text editor (defaults to VI but can be changed: <https://www.oreilly.com/library/view/gitlab-cookbook/9781783986842/apas07.html>). Write your commit message at the top. You can see all changes staged for commit below (the same thing you'd see when running git status). Double check that the list of files staged for commit is correct! After saving the file, confirm that there are no errors.

Finally, we deploy our app to Heroku:

$ git push heroku master

(You may see the following warning the first time - it's fine---answer "yes", and in the future you shouldn't see it anymore:)

The authenticity of host 'heroku.com (50.19.85.132)' can't be established.

RSA key fingerprint is 8b:48:5e:67:0e:c9:16:47:32:f2:87:0c:1f:c8:60:ad.

Are you sure you want to continue connecting (yes/no)?

Please type 'yes' or 'no':

You will see some warnings, but none are critical in most cases.

Is the app running on Heroku? If you navigate to the heroku URL that is printed at the end of the results from git push heroku master you'll get a "We're sorry, but something went wrong." error in the browser.

We can get a hint as to why by running the following command:

$ heroku logs

which will show an error like:

ActionView::Template::Error (PG::UndefinedTable: ERROR: relation "movies" does not exist

Just as we ran rake db:migrate and rake db:seed to do first-time database creation locally, we must also cause a database to be created on the Heroku side:

$ heroku rake db:migrate

and

$ heroku rake db:seed

Now you should be able to navigate to your app's URL. heroku open opens your browser to that URL in case you forgot it, however this command does not work on c9, where you will need to navigate to the relevant URL.

**Note:** don't proceed past this point until you are able to complete the above successfully, or you won't be able to receive a grade for this assignment!

* **Part 1:**[**Sort the list of movies (15 points)**](https://github.com/UCCSCS3300/hw-rails-intro/blob/master/docs/part_1.md)

On the list of all movies page, make the column headings for "Movie Title" and "Release Date" into 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" header should list the movies alphabetically by title. (For movies whose names begin with non-letters, the sort order should match the behavior of String#<=>.)

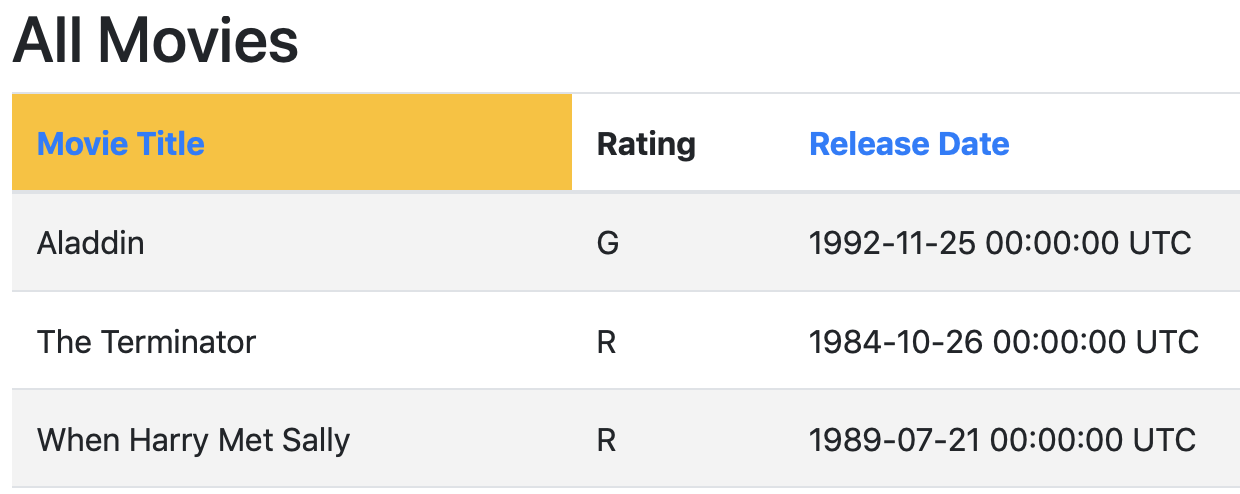
When the listing page is redisplayed with sorting-on-a-column enabled, the column header that was selected for sorting should appear with a yellow background, as shown below. You should do this by setting controller variables that are used to conditionally set the CSS class of the appropriate table heading to hilite, and pasting this simple CSS into RottenPotatoes app/assets/stylesheets/default.css file:

table#movies th.hilite {

background-color: yellow;

}

The result should look something like this:

[](https://github.com/saasbook/hw-rails-intro/blob/master/table-header-screenshot.png)

**IMPORTANT for grading purposes:**

The link (that is, the <a> tag) for sorting by "title" should have the HTML element id title\_header, and 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 the starter code).

### Hints and caveats:

* The current RottenPotatoes views use the Rails-provided "resource-based 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](http://api.rubyonrails.org/v4.2.6/) for ActiveRecord.order 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.

### Finishing Part 1

You'll submit the code for this part after you deploy on Heroku and when you supply your Heroku deployment URL in part 3.

For now, commit all the changes you have made so far (see the beginning of this part), and deploy them to check that they work on Heroku before moving on to the next section:

$ cd /root/environment/homework/rottenpotatoes-rails-intro

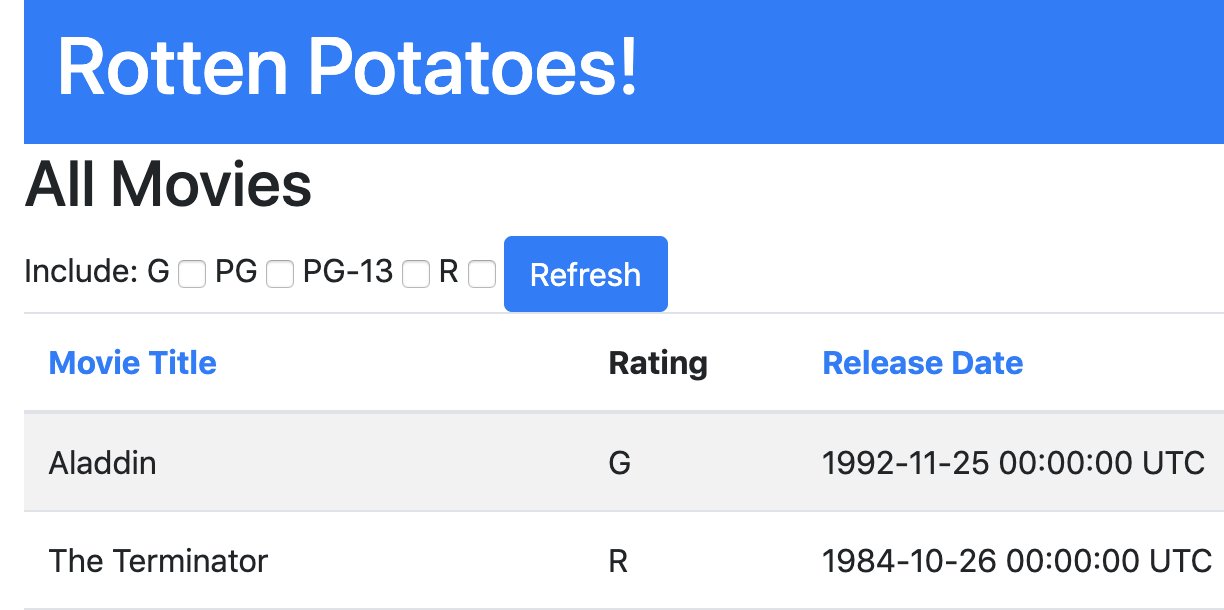
$ git commit -am "part 1 complete"

$ git push heroku master

You must be in the rottenpotatoes-rails-intro directory to deploy to Heroku because that's the home of the application (and the corresponding git configuration)

* **Part 2:**[**Filter the list of movies by rating (15 points)**](https://github.com/UCCSCS3300/hw-rails-intro/blob/master/docs/part_2.md)

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:

[](https://github.com/saasbook/hw-rails-intro/blob/master/filter-screenshot.png)

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:

= form\_tag movies\_path, :method => :get do

Include:

- @all\_ratings.each do |rating|

= rating

= check\_box\_tag "ratings[#{rating}]"

= submit\_tag 'Refresh'

BUT, you have to do a bit of work to use the above code: as you can see, it 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. Hence, you should create a class method of Movie that returns an appropriate value for this collection.

You will also need code that knows (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 (checkboxes that weren't checked don't appear in the params hash at all).

Regarding (ii), you'll probably end up replacing Movie.all in the controller method with Movie.where, which has various options to help you restrict the database query.

### IMPORTANT for grading purposes

* Your form tag should have the id ratings\_form.
* The form 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 ratings\_PG-13, ratings\_G, and so on.

### Hints and caveats

Make sure that you don't break the sorted-column functionality you added previously! 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 be sorted but do not need to be limited by the checked ratings (we'll get to that in part 3).

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.

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.

Reminder: Don't put code in your views! Set up an 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.

## Finishing Part 2

You'll submit this part after you deploy on Heroku and when you supply your Heroku deployment URL in part 3. But you can commit all the changes you have made so far to git (see instructions at the beginning of part 2), deploy them to Heroku and check that they work on Heroku before moving on to the next section:

$ cd /root/environment/homework/rottenpotatoes-rails-intro

$ git commit -am "part 2 complete"

$ git push heroku master

* **Part 3:**[**Remember the sorting and filtering settings (70 points)**](https://github.com/UCCSCS3300/hw-rails-intro/blob/master/docs/part_3.md)

OK, so the user can now 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. 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" her 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.

### Hints and caveats

If the user explicitly includes new sorting/filtering settings in params[], the session should not override them. Instead, these new settings should be remembered in the session.

If a user unchecks all checkboxes, use the settings stored in the session[] hash, since it doesn't make sense for a user to uncheck all the boxes.

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.

### Finishing Part 3

Push to git and deploy to Heroku by following the same process as before:

$ cd /root/environment/homework/rottenpotatoes-rails-intro

$ git commit -am "part 3 complete"

$ git push heroku master

**How to submit when you're all done**

There are 3 parts required:

* Push final work on github (all source code)
* Push to Heroku

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.

Once you're confident the functionality works correctly on Heroku, submit the URI of your deployed Heroku app in a text file with no other contents. **Please be careful** to use **http** and not **https**, that is, submit http://your-app.herokuapp.com **and NOT** https://your-app.herokuapp.com.

* Submit on Canvas in tar.gz format

As an extra backup/safety net, submit all code in tar.gz format on Canvas.

$ cd /root/environment/homework/rottenpotatoes-rails-intro/

$ tar -czf yourUCCSname\_hw2.tar.gz \*

Submit your tar.gz file.

Your final submission on Canvas includes 1) your tar.gz file and 2) a text file containing only the URI of your deployed Heroku App