ECE:5820/CS:5820
Fall 2018
Sixth Homework Assignment
Due: Saturday Oct. 20 by 11:59 p.m.
(Submission instructions are provided below)

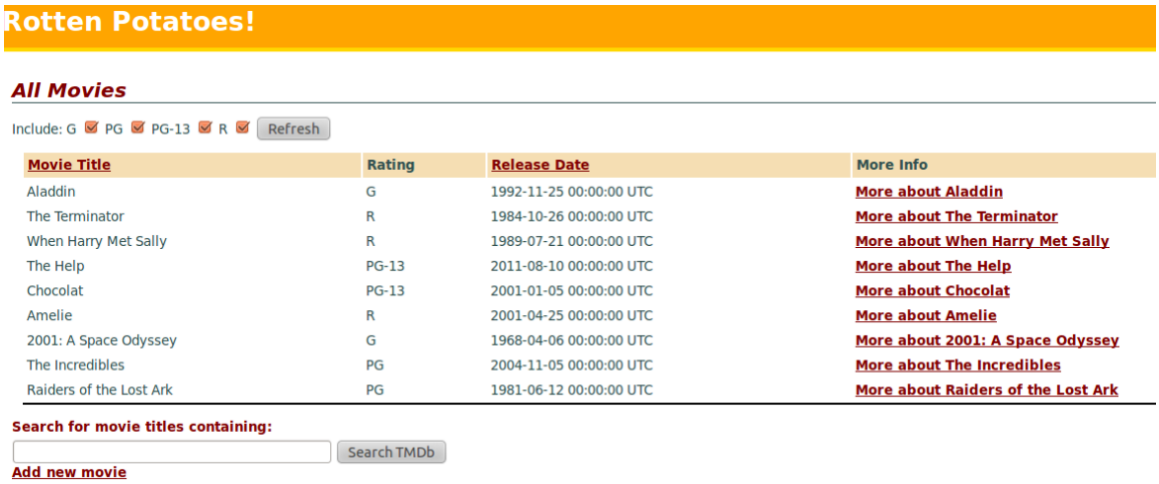
Introduction:

In this assignment, you will add a new feature to RottenPotatoes and write a set of RSpec unit tests to test the new feature. The new feature will provide the ability to search for movies in the TMDb (www.themoviedb.org) database and import selected movies from TMDb into RottenPotatoes. The new feature is similar to the example used in lecture and chapters seven and eight of the text, but differs in some important respects, as described below.

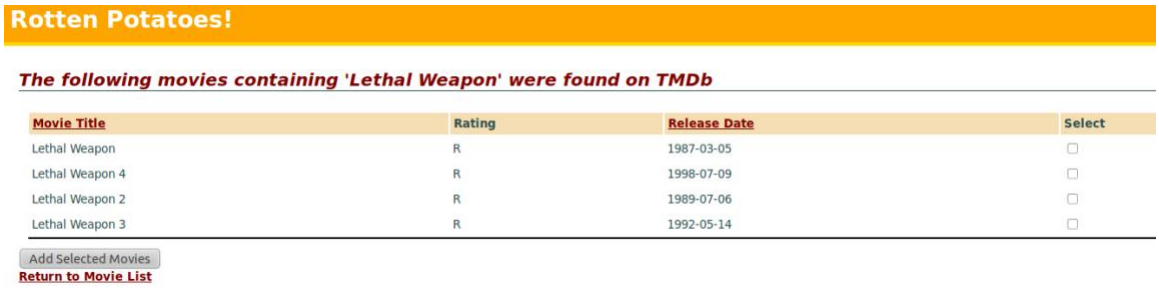
This assignment requires deployment of your completed app to Heroku in the same manner as was done for HW4. It is recommended that you do the initial deployment early on and then push updates to Heroku as you add functionality to the app. Do not leave deployment until the last minute. Your assignment will not be graded if it is not successfully deployed to Heroku.

Overview of new feature:

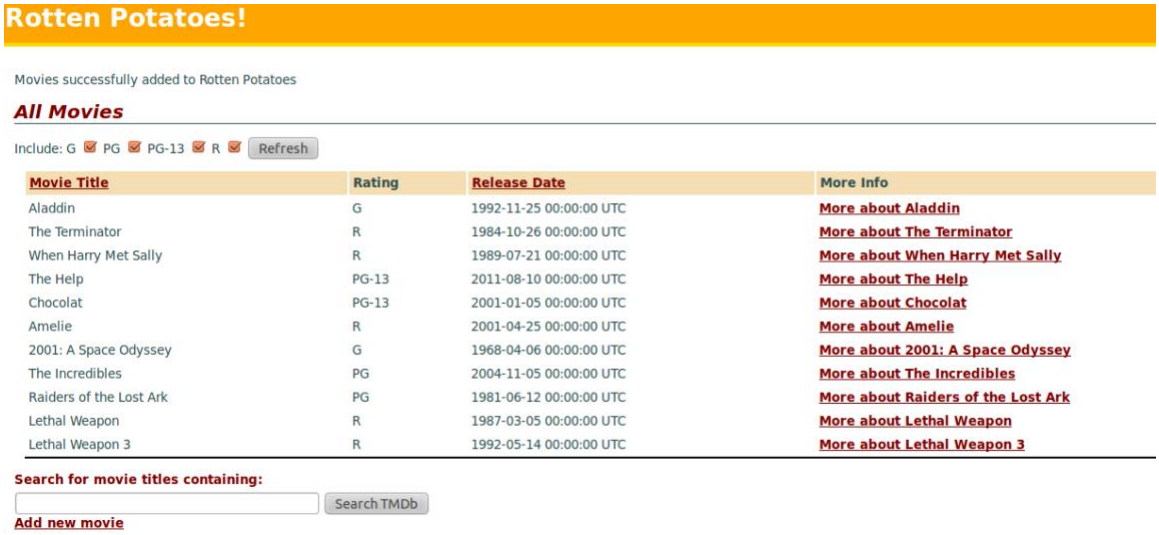
A text-box and button will be added to the RottenPotatoes homepage to allow a user to enter a search term and initiate a TMDb search, as shown below:



If a user enters a search term such as `'Lethal Weapon'` and clicks the "`Search TMDb`" button, a list of movies from TMDb whose titles include the search term should be displayed, as shown:



Using the check-boxes at the right of each item, the user can select any subset of the displayed movies. When the "`Add Selected Movies`" button is clicked, the selected movies should be added to the RottenPotatoes database and the user should be redirected to the homepage with a flash message as shown below (in the case shown, the user opted to add two of the four matching movies from TMDb):



If the user clicks "`Add Selected Movies`" without selecting any movies he/she should be redirected to the homepage with a flash "`No movies selected`"

If the TMDb search does not find any movies matching the search term, the following message should be flashed on the RP home page:

"`No matching movies were found on TMDb`"

If the user clicks "`Search TMDb`" without entering a non-blank search term, a flash of the form "`Invalid search term`" should be displayed on the homepage.

Getting started:

The starting point for this assignment will be the version of RottenPotatoes supplied on GitHub classroom. This version of RottenPotatoes already includes the basic TDD-related elements discussed in lecture and Chapter 8 of the text.

Be sure to run "`bundle install`" to insure that all needed gems are in place. You will also need to migrate and seed the database.

Next go the GitHub web page for the '`themoviedb`' gem and familiarize yourself with the basics of the gem's interface. (You can find this page by googling "themoviedb gem") You should experiment with the gem in irb. Be sure to enter the command:
require 'themoviedb'

Specifically note the following:

- The way that the TMDb API key is set for the gem:

  ```
  Tmdb::Api.key("KEY_GOES_HERE")
  ```

  You should use the following API key for this assignment:
  f4702b08c0ac6ea5b51425788bb26562

- The way that keyword searches are performed on movie titles:

  ```
  matching_movies = Tmdb::Movie.find("search term")
  ```

  Note that this method returns an array of `Tmdb::Movie` objects, one for each matching movie title. (Note that the class `Tmdb::Movie` is not the same as the Movie class `ActiveRecord::Base::Model` that implements the RP model.)

- How information can be extracted from a `Tmbd::Movie` object:

  ```
  movie = matching_movies[0]
  movie.title   #returns the title of the movie
  movie.id       # returns TMDb id of the movie
  movie.overview #returns a short synopsis of the movie
  etc.
  ```

- The manner in which the TMDb id can be used to look-up a movie by id and also find additional details, such as an overview (description) of the movie:

```
Tmdb::Movie.detail(id)  # returns details of movie with
                        # the specified Tmdb id
```

- And finally, the TMDb API does not handle rating information for movies in a straighforward manner. To see where this information can be located look at: Tmdb::Movie.releases(id)).

OK, we are finally ready to roll:

Since this is your capstone homework assignment, we will provide only *general hints and guidelines* for completing each part of the assignment. You will need to carefully read the guidelines and make inferences for some missing details along the way. For the most part, you can follow the general testing framework used in chapter 8 of the text. However, you will need to adapt some things to account for the differences between the feature being implemented in the assignment versus the slightly simpler feature used in the text.

Let's start with the controller action triggered by clicking the "Search TMDb" button on the homepage. As in the text, let's call this controller method `MoviesController#search_tmdb` and associate it with the route:

```
post  '/movies/search_tmdb'
```

You can develop the spec for this method along the same lines as the text, with the following exceptions:

- Be careful about defining the interface between the controller and the model--i.e. the nature of the object returned by the call to `Movie::find_in_tmdb`.
  There are several plausible choices--e.g. an array of `Tmdb::Movie` objects or an array of `ActiveRecord::Base::Movie` objects. However, the best choice is probably just a simple array of hashes, where each element of the array is a hash containing the information for one movie--i.e. `:tmdb_id`, `:title`, `:rating`, `:release_date`.
  The advantage of this choice is that the controller will be independent of any details of the '`themoviedb`' gem used in the model.

- The `search_tmdb` controller method should check for invalid search terms--i.e. nil or all-blank--and redirect the user as described earlier.

- The `search_tmdb controller` method will require two instance variables for communication with the view: one to pass the movie information (array of movie hashes) to the view and one to pass the search term to the view (since the search term is used in constructing the header for the view as shown in the second screenshot above.)

When you have completed the spec and implementation for the `MoviesController#search_tmdb` method, it is time to turn your attention to

the Movie model method `Movie::find_in_tmdb`. You can develop the spec for this method along the same lines as discussed in class and the text, with the following exceptions.

- As noted above, the `Tmdb::Movie.find` method returns an array of `Tmdb::Movie` objects. However, we decided earlier that the `find_in_tmdb` method should return an array of hashes. Hence, the `find_in_tmdb` method will need to convert the array of `Tmdb::Movie` objects into an array of hashes.

- If the TMDb search does not yield any results, the `find_in_tmdb` method should return an empty array, [].

- As noted earlier, you don't need to worry about possible exceptions thrown by the `Tmdb::Movie.find` method

When you are satisfied that the `MoviesController#search_tmdb` and `Movie::find_in_tmdb` methods are working correctly, you can finish the implementation of this portion by completing the associated views. Specifically, you will need to add the text-box and submit button for the search term to the `index.html.haml` view, and you will need to construct the view associated with the `MovieController#search_tmdb` method.

The `search_tmdb.html.haml` view can borrow heavily from the existing `index.html.haml` view, with the following changes:

- You will need to strip out all of the functionality associated with filtering and sorting movies, since it is not needed here.

- Keep in mind that the information passed from the controller to the view is an array of hashes, not an array of Movie objects as used for the `index.html.haml` view.

- You should replace the "`More Info`" links in the last column of the table with check-boxes. You can specify the check-box as follows:

  ```
  %td=check_box_tag "tmdb_movies[#{movie[:tmdb_id]}]"
  ```

  The effect of this will be discussed below.

- You should wrap a form-tag with submit button ("`Add Selected Movies`") around the entire table to allow the user to submit the selected movies to be added to RottenPotatoes.

Now, again using TDD, you will need to implement the controller and model actions associated with clicking the "Add Selected Movies" button. Let's start with the controller action. We will call this method `MoviesController#add_tmdb`. This method will receive a params hash containing information about which movies were selected for addition to RottenPotatoes. The manner in which the check-boxes were specified in the `search_tmdb` view will cause rails to aggregate the state of the check-boxes into a single hash called `:tmdb_movies`, whose keys will be the names of the **checked boxes only**, and whose values will be "1". In this scheme, the name of a check-box will be the `tmdb_id` of the movie it is associated with. Hence, the keys of the hash will represent the set of `tmdb_id`s of the movies that whose check_boxes are checked. You can extract these keys from the hash (using the `Array#keys` method) to construct an array of `tmdb_id`s.

Your `MoviesController#add_tmdb` method should iterate through this array of `tmdb_id`'s. For each one, it should call the Movie Model method `Movie::create_from_tmdb`, passing the `tmdb_id` as a parameter.

The `Movie::create_from_tmdb` should use the `tmdb_id` to look up the details of the movie in Tmdb (using the `Tmdb::Movie.details` method) and should then add the movie to the RottenPotatoes database.

Be sure that your controller method correctly implements the behavior specified in the *Getting Started* section above.

OK, it's all working! Now what:

Let's see how thoroughly your specs test your application. Watch Screeencast 8.7.1 from the text, which explains how to use a tool called SimpleCov to check the C0 coverage of your tests. (We will explain the concept of C0 coverage in class.) Examine the SimpleCov coverage report. If it identifies any areas of low coverage you may want to add so additional specs. Since our tests have focused only on the model and controller, you can ignore coverage results that relate to other aspects of your app.

Important: Before submitting you MUST insure that certain html elements in your views have the id attributes specified below. The ids must be exactly as specified here in order to allow grading of your submission. Be sure to check the page sources to verify that the ids are in place before submitting:

The textbox on the index page must have id: "search_box"

The "Search TMDb" button on the index page must have id: "search_button"

The check-box elements on the page that displays matching movies from TMDb must have id: "checkbox_xxx" where xxx is the Tmdb id of the corresponding movie. For example, in the scenario shown above, the element id of the checkbox in the first row of the table (i.e. the checkbox for the movie "Lethal Weapon") should be "checkbox_941", since 941 is the Tmdb id for "Lethal Weapon"

The "Add Selected Movies" button must have id: "movies_button"

All headers, labels, messages and flashes must be exactly as specified in this document.

As noted earlier, your app must be deployed and running on Heroku in order to be graded.

Submission Instructions (Be sure to follow these instructions exactly!):
Deploy your completed assignment to Heroku. We will be testing your deployed app
on Heroku so it is essential that you successfully deploy your app to Heroku. The
Heroku URI for your deployed app must be added to the GRADING.json file in your git
repository where:
GRADING.json

```
{
"heroku_url": "http://blah.blee.boo/myapp",
"instructions": "Change the string associated with heroku_url to your
heroku application url to be graded."
}
```

Where the string http://blah.blee.boo/myapp is replaced with your URL.