图片包含 场记板, 文字

已生成极高可信度的说明

reflection report

iMovie Web Application

October 23th, 2017

**Team:** **36**

team members:

Jiahui CHEN 450554021

Zhe LIU 450182813

Xuan WEI 460175351

Meng YUAN 470241734

Jichen ZHAO 460225645

Table of Contents

[Introduction 1](#_Toc496289139)

[Code structure 1](#_Toc496289140)

[Style guide 1](#_Toc496289141)

[File/folder layout change 6](#_Toc496289142)

[Data model 8](#_Toc496289143)

[Final Model style 8](#_Toc496289144)

[Function 9](#_Toc496289145)

[Design 9](#_Toc496289146)

[Change 9](#_Toc496289147)

[Highlights Tech 10](#_Toc496289148)

[Solr in search function 10](#_Toc496289149)

[Redis in ranking function 10](#_Toc496289150)

# Introduction

This document includes code structure, data model and all functions of the iMovie website. During the process of development, some functions were changed and some functions are added in the application. The developers of iMovie website implemented the application and compared the finished project to the SRS, all changes are listed in this reflective documentation. The aim of this documentation is to help audience of other developers review this application easily .

# Code structure

## Style guide

**Python style**

* Please conform to the indentation style dictated in the .editorconfig file. We recommend using a text editor with [EditorConfig](http://editorconfig.org/) support to avoid indentation and whitespace issues. The Python files use 4 spaces for indentation and the HTML files use 2 spaces.
* Unless otherwise specified, follow [**PEP 8**](https://www.python.org/dev/peps/pep-0008).
* Use [flake8](https://pypi.python.org/pypi/flake8) to check for problems in this area. Note that our setup.cfg file contains some excluded files (deprecated modules we don’t care about cleaning up and some third-party code that Django vendors) as well as some excluded errors that we don’t consider as gross violations. Remember that [**PEP 8**](https://www.python.org/dev/peps/pep-0008) is only a guide, so respect the style of the surrounding code as a primary goal.
* An exception to [**PEP 8**](https://www.python.org/dev/peps/pep-0008) is our rules on line lengths. Don’t limit lines of code to 79 characters if it means the code looks significantly uglier or is harder to read. We allow up to 119 characters as this is the width of GitHub code review; anything longer requires horizontal scrolling which makes review more difficult. This check is included when you run flake8. Documentation, comments, and docstrings should be wrapped at 79 characters, even though [**PEP 8**](https://www.python.org/dev/peps/pep-0008) suggests 72.
* Use four spaces for indentation.
* Use four space hanging indentation rather than vertical alignment:
* **raise** **AttributeError**(  
   'Here is a multine error message '  
   'shortened for clarity.'  
  )
* Instead of:
* **raise** **AttributeError**('Here is a multine error message '  
   'shortened for clarity.')
* This makes better use of space and avoids having to realign strings if the length of the first line changes.
* Use single quotes for strings, or a double quote if the the string contains a single quote. Don’t waste time doing unrelated refactoring of existing code to conform to this style.
* Avoid use of “we” in comments, e.g. “Loop over” rather than “We loop over”.
* Use underscores, not camelCase, for variable, function and method names (i.e. poll.get\_unique\_voters(), not poll.getUniqueVoters()).
* Use InitialCaps for class names (or for factory functions that return classes).
* In docstrings, follow the style of existing docstrings and [**PEP 257**](https://www.python.org/dev/peps/pep-0257).
* In tests, use [assertRaisesMessage()](https://docs.djangoproject.com/en/dev/topics/testing/tools/#django.test.SimpleTestCase.assertRaisesMessage) instead of [assertRaises()](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaises) so you can check the exception message. Use [assertRaisesRegex()](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaisesRegex) only if you need regular expression matching.
* In test docstrings, state the expected behavior that each test demonstrates. Don’t include preambles such as “Tests that” or “Ensures that”.
* Reserve ticket references for obscure issues where the ticket has additional details that can’t be easily described in docstrings or comments. Include the ticket number at the end of a sentence like this:
* **def** test\_foo():  
   *"""*  
   *A test docstring looks like this (#123456).*  
   *"""*  
   ...

**Imports**

* Use [isort](https://github.com/timothycrosley/isort#readme) to automate import sorting using the guidelines below.
* Quick start:
* **$** pip install isort  
  **$** isort -rc .
* This runs isort recursively from your current directory, modifying any files that don’t conform to the guidelines. If you need to have imports out of order (to avoid a circular import, for example) use a comment like this:
* **import** **module** *# isort:skip*
* Put imports in these groups: future, standard library, third-party libraries, other Django components, local Django component, try/excepts. Sort lines in each group alphabetically by the full module name. Place all importmodule statements before from module import objects in each section. Use absolute imports for other Django components and relative imports for local components.
* On each line, alphabetize the items with the upper case items grouped before the lower case items.
* Break long lines using parentheses and indent continuation lines by 4 spaces. Include a trailing comma after the last import and put the closing parenthesis on its own line.
* Use a single blank line between the last import and any module level code, and use two blank lines above the first function or class.
* For example (comments are for explanatory purposes only):
* *# future*  
  **from** **\_\_future\_\_** **import** unicode\_literals  
    
  *# standard library*  
  **import** **json**  
  **from** **itertools** **import** chain  
    
  *# third-party*  
  **import** **bcrypt**  
    
  *# Django*  
  **from** **django.http** **import** Http404  
  **from** **django.http.response** **import** (  
   Http404, HttpResponse, HttpResponseNotAllowed, StreamingHttpResponse,  
   cookie,  
  )  
    
  *# local Django*  
  **from** **.models** **import** LogEntry  
    
  *# try/except*  
  **try**:  
   **import** **yaml**  
  **except** **ImportError**:  
   yaml = **None**  
    
  CONSTANT = 'foo'  
    
    
  **class** **Example**:  
   *# ...*
* Use convenience imports whenever available. For example, do this:
* **from** **django.views** **import** View
* instead of:
* **from** **django.views.generic.base** **import** View

**Template style**

* In Django template code, put one (and only one) space between the curly brackets and the tag contents.
* Do this:
* {{ foo }}
* Don’t do this:
* {{foo}}

**View style**

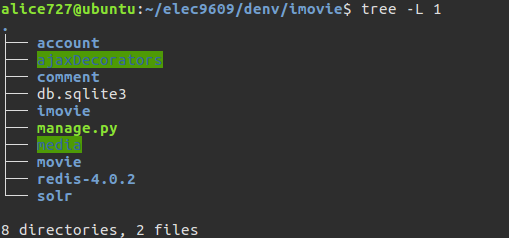
* In Django views, the first parameter in a view function should be called request.
* Do this:
* **def** my\_view(request, foo):  
   *# ...*
* Don’t do this:
* **def** my\_view(req, foo):  
   *# ...*

**Model style**

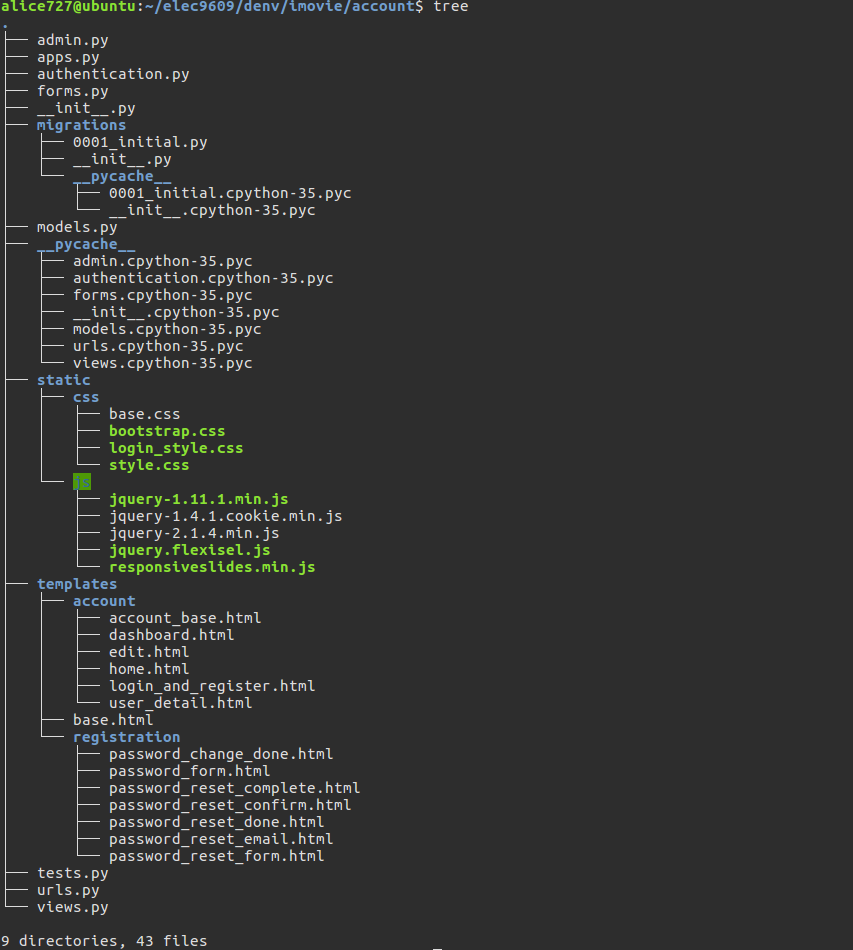
* Field names should be all lowercase, using underscores instead of camelCase.
* Do this:
* **class** **Person**(models.Model):  
   first\_name = models.CharField(max\_length=20)  
   last\_name = models.CharField(max\_length=40)
* Don’t do this:
* **class** **Person**(models.Model):  
   FirstName = models.CharField(max\_length=20)  
   Last\_Name = models.CharField(max\_length=40)
* The class Meta should appear *after* the fields are defined, with a single blank line separating the fields and the class definition.
* Do this:
* **class** **Person**(models.Model):  
   first\_name = models.CharField(max\_length=20)  
   last\_name = models.CharField(max\_length=40)  
    
   **class** **Meta**:  
   verbose\_name\_plural = 'people'
* Don’t do this:
* **class** **Person**(models.Model):  
   first\_name = models.CharField(max\_length=20)  
   last\_name = models.CharField(max\_length=40)  
   **class** **Meta**:  
   verbose\_name\_plural = 'people'
* Don’t do this, either:
* **class** **Person**(models.Model):  
   **class** **Meta**:  
   verbose\_name\_plural = 'people'  
    
   first\_name = models.CharField(max\_length=20)  
   last\_name = models.CharField(max\_length=40)
* The order of model inner classes and standard methods should be as follows (noting that these are not all required):
  + All database fields
  + Custom manager attributes
  + class Meta
  + def \_\_str\_\_()
  + def save()
  + def get\_absolute\_url()
  + Any custom methods
* If choices is defined for a given model field, define each choice as a tuple of tuples, with an all-uppercase name as a class attribute on the model. Example:
* **class** **MyModel**(models.Model):  
   DIRECTION\_UP = 'U'  
   DIRECTION\_DOWN = 'D'  
   DIRECTION\_CHOICES = (  
   (DIRECTION\_UP, 'Up'),  
   (DIRECTION\_DOWN, 'Down'),  
   )

(Django coding style, 2009)

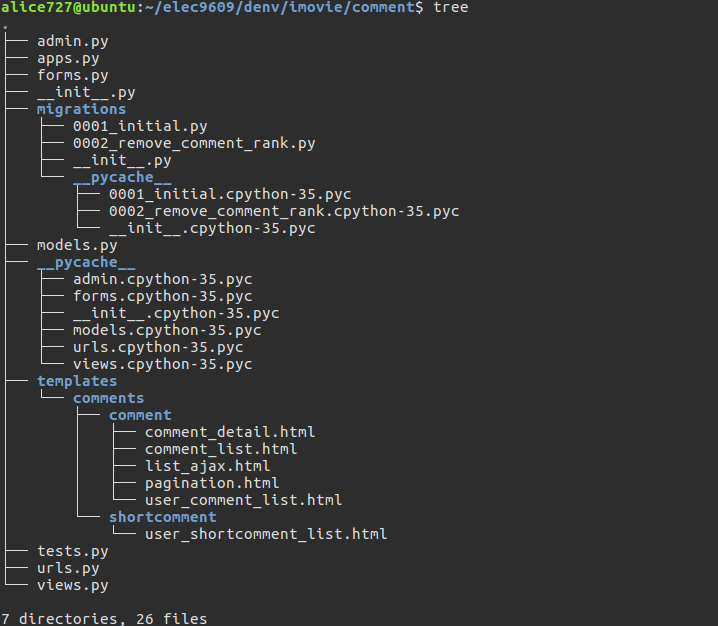
## File/folder layout change



**Figure1. Base code structure of the iMovie Web Application**



**Figure2. Account code structure of the iMovie Web Application**



**Figure1. comment code structure of the iMovie Web Application**

The folder structure of this project in system design specification documentation including account application and movie application. Each application involves different kinds of file to support the project, such as urls.py, forms.py, admin.py, models.py and templates folder of html files. The media folder in Django project in is used to save media static files, such as images , videos and sounds.

To compare the file layout in the project, the structure is much more complex than before. There are three main code structure in it which were base code, account code and comment code. The comment was added in the application during the development process, it is relating to the comment and rate function in iMovie website.

# Data model

## Final Model style

**1. Class:**

* User: User\_id Username, Firstname, Lastname, Email
* Profile: User\_id, data\_of\_birth, photo
* Book\_Activity: User\_id, Activity\_id
* Activity: Activity\_id, Body, Created, Active,Title, Slug, Date, Todate
* Category: name, slug
* Shortcomment: Comment\_id, Author, Movie, Body, Created, Rank, Active
* Comment: Comment\_id, Author, Movie, Body, Created, Active, Title, Slug
* AnotherComment: Comment\_id , Author, Comment, Body, Created, Active
* Movies: category, name, director, scriptwriter, Nation, Star, Length, year, language, description, poster, rank, created, updated, users\_like, video

**2. Relationship**

* Profile is used to expand user’s information, so that it keeps one to one relationship with user table
* Shotcomment, Comment, and AnotherComment have comment\_id as their primary key. They should link to Movies class with their Author and Movie entities as many to one’s relationship.
* Movie should link to category table, cause we need define different category. Thus, one category can have many movies, but one movie only belong to one category.
* Activity class should link to Book\_Activity class, cause during each booking time, only can book one activity. So the relationship between Activity and Book\_Activity should be one to one.
* Because User can book many activities, and each activity can be booked by many users. Meanwhile, the relathionship should be many to many.

**3. Difference**

* We have a little bit change on User, Movie, and Activity’s entities change. Cause we find the fixed entities are more useful and related to our website.
* We add AnotherComment class, cause we find that uesers may comment others’ long comments.
* We also have category class. Because we did not thought on category’s detail on movies.

# Function

Design

In SRS document, 25 use cases were designed to implement several MVP functions, including accounts related functions, comments related functions, movies related functions, and some relevant operation functions between accounts and movies. In addition, 5 none- MVP use cases are involved in to provide better user experience.

All the requirements specified in SRS document are implemented in this project. The detailed operations for admin actor and user actor are specified as follows.

First, for the admin actor, Django automatically generates an admin account for developers, and this account already involved all the necessary use cases announced in SRS. In details, for account related functions, admin could block and delete users’ accounts; for comments related functions, admin could delete users’ comment; for movie related functions, admin could add, delete and edit a movie and its information; and for optional functions, admin could create an activity.

Second, end-user actors could create, edit, login to, logout from and change password of the account, and they could view other users’ account. For comments related functions, users could add a new comment, edit and delete an exist comment, and view other users’ comments of a movie. For movie and relevant operations, users could view movies’ information, rate a movie, search a movie by name, filter movies’ list by category, and view the recommendation movies’ ranking, where ranking includes the highest rating ranking and the maximum clicked ranking. For none minimum viable products, users could respond to an activity, cancel their responding and view all the activities they responded.

Change

In addition to the use cases announced in SRS, two new functions are added in this project.

The first added function is the counter of how many users have viewed this movie. Adding this function has two advantages for admin. One advantage is that admin could know which category of movie is the most attractive category of users, and the other advantage is that admin could recommend movies to user with the highest clicked counters.

Another added function is that user could ‘like’ a movie, which means that users could click the ‘like’ button if they find a preferred movie during view movie’s information page. This function gives users the chance to save their preferred movies within accounts’ favourite. Users might acquire better user experience by adding this ‘like’ function, since this function reduces the number of clicking steps for users to find a viewed and liked movie.

# Highlights Tech

## Solr in search function

Search function means that users could search the information of a movie in search bar with the movie’s name. The simple way to implement this function is using SQL to select specific data from database. However, this method has many disadvantages. First, writing SQL will reduce the security of website, which increases the possibility of attack, such as SQL inject attacks. Second, when the scale of database increase, using SQL to search data is not very efficiency.

An alternative method to implement search function is using Solr, which is a famous open source search engine. Solr avoids those disadvantage of SQL, and also provide a failover mechanism. Overall, Solr is a good method to solve the search problem in large scale database.

## Redis in ranking function

Ranking function is that users could get recommendation of movies, based on movies’ rating or movies’ clicking times. The easiest way to implement this function is also using SQL. As talked above, SQL has many weaknesses. Therefore, in ranking function, Redis has been used to replace SQL searching. Redis is a key-value storage in cache with high performance -- responses very quickly, and support lots of data types. In addition, Redis could calculate the number of viewed times automatically, thus Redis is very suitable for implement the ranking function.