System Design Document

MusicBug

Client
Christopher Nguyen

Team 4
Tommy Brickwedde
Jeff Clouse
Michael Cohen
Joe DiMarino
Sean Ehrig
Kevin Wiggins

3/1/2012

MusicBug
System Design Document

Table of Contents

1. Introduction

1.1 Purpose of This Document

The purpose of this document is to describe the design of the MusicBug application. Key topics covered in this document include the high level system architecture, lower level class designs, and the persistent data design of MusicBug.

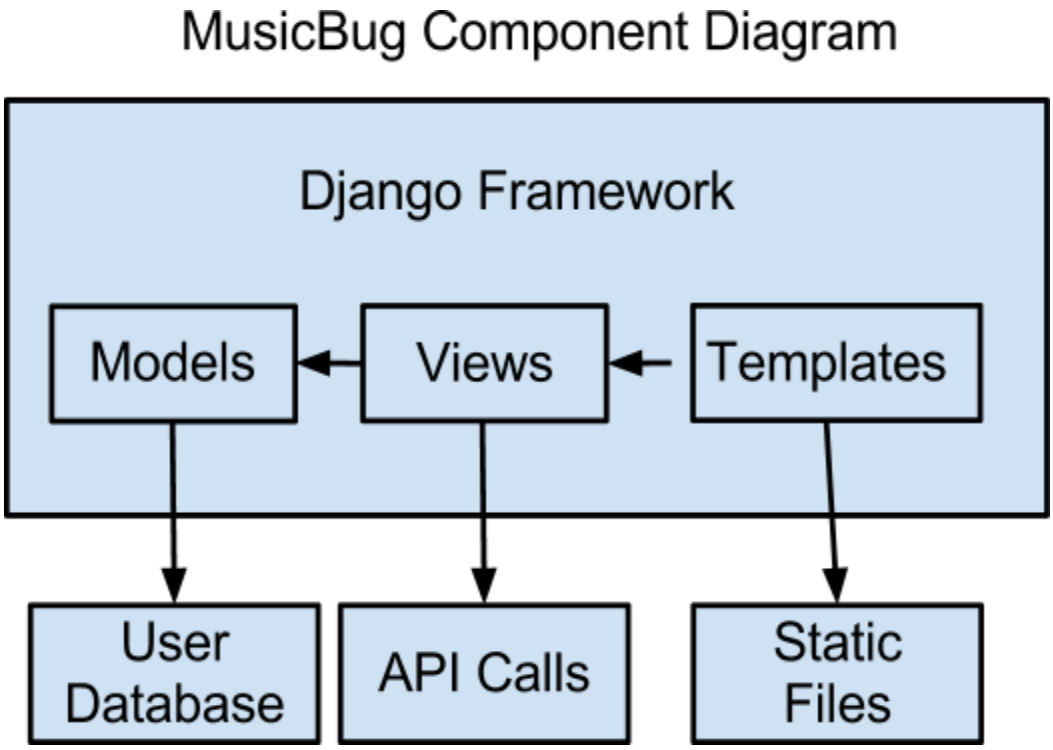
1.2 References

Throughout this document references will be made to:

1. Lukáš Lalinský (2012, Jan). CreateTables.sql. Retrieved from “https://github.com/lalinsky/mbslave/blob/master/sql/CreateTables.sql”
2. MusicBrainz Schema (2011, Nov 14). Wiki. Retrieved from “http://wiki.musicbrainz.org/-/images/5/52/ngs.png”, “http://musicbrainz.org/doc/Database_Schema”
3. Django Project (2012). The web framework for perfectionists with deadlines. Retrieved from “https://www.djangoproject.com/”
4. The MusicBug System Requirements Document

2. System Architecture

2.1 Architectural Design



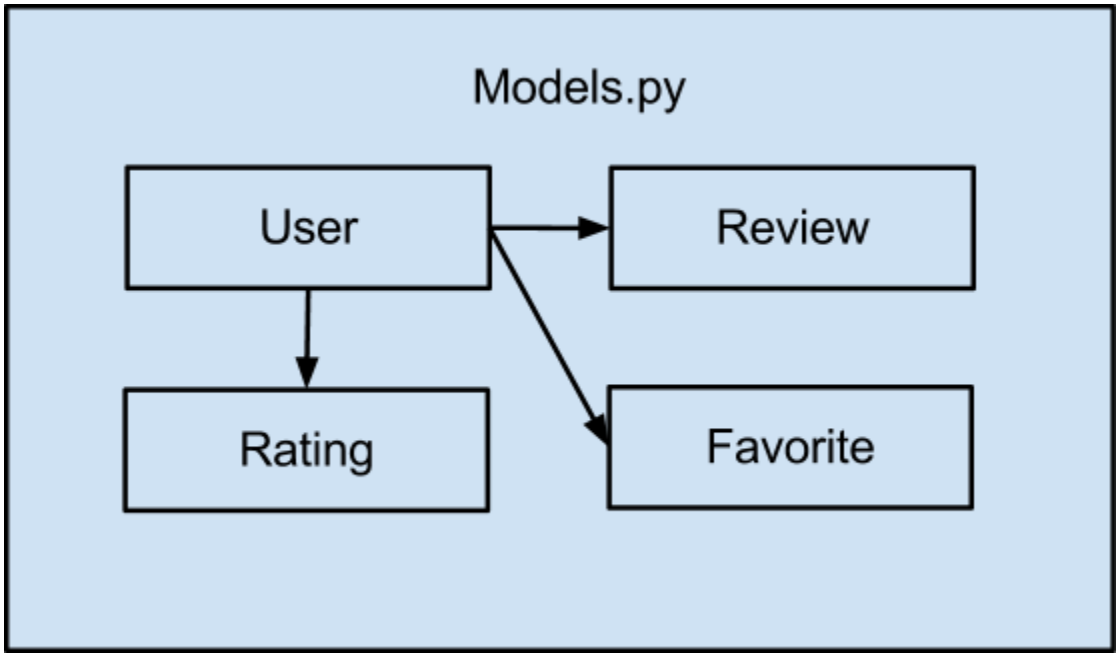
The MusicBug application will be built using the Django web framework. The basic components of the Django framework include Models, Views, and Templates, which follows a Model View Controller architecture (MVC).

The model layer consists of classes, each of which refer to a specific table in a database. All data used in our application will be represented by a model, with the actual data existing in a database. Model classes describe user data including basic user information, user credit card information, and user ratings, favorites, and reviews. These will be described in more details later in this document. Every model class in our system has a corresponding table in our database. Our persistent data design will be described later in the document.

Templates are the specific pages served to the user. Templates are made up of html and other static files. The templates used in this application will be described more fully in the User Interface Design Document.

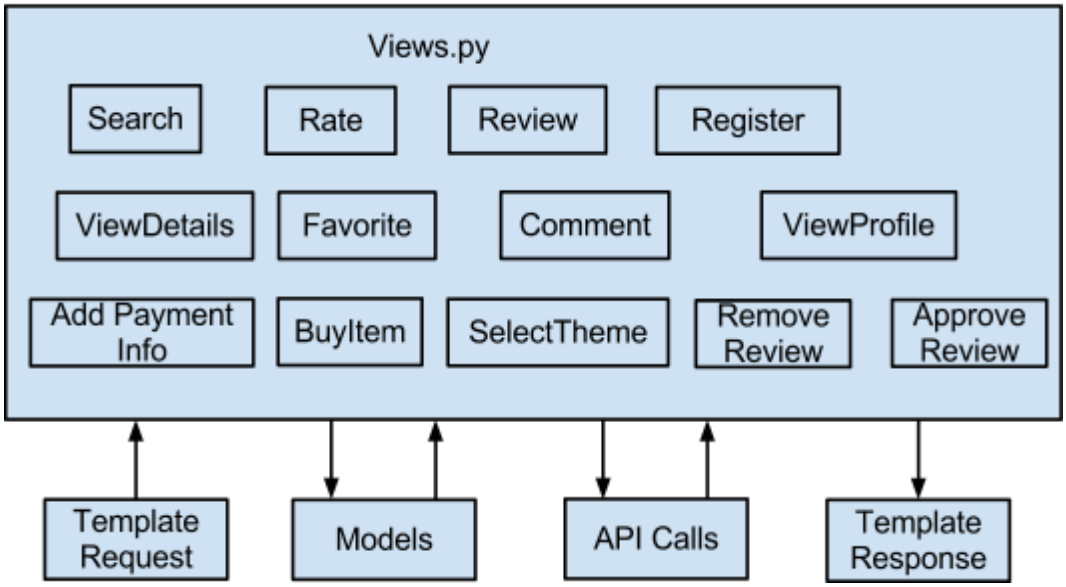
The views are a layer of abstraction between the model and template layers. A view, or the controller in MVC, is a function which takes input and returns output to the user through the template layer. Normally the view would interact with any number of model classes, then either return some subset of data, or perform operations on data and return the results. Views in this application include search, view details, and browse, among others which will be described more fully later in this document.

2.2 Decomposition Description



Each model in MusicBug is a Python class. Each class corresponds to the user database which is populated as the application is used. When a user registers with MusicBug, an entry in the user table is populated, the fields for the User class include: Name, Birthdate, RegisterDate, LastLoginDate, gender, location, moderator status, and fid (from Facebook). If the user doesn't give access to a specific field it is not saved in the system. The Review, Rating, and Favorite classes contain the User, rated or favorited, the subject being rated, reviewed, or favorited, and the rating or review itself.

MusicBug View Decompisition Diagram



Each view is a python function which takes an HTTP request and returns an HTTP response. The search function returns the query result of all artist, albums, and songs, matching a search term given by the user. The view details function returns the details of an artist, track, or album. The search and view details functions both use the MusicBrainz API, Lyrdb API, and 7digital API to access the relevant data. MusicBrainz, Lyrdb, and 7digital are all public databases of music related metadata which can be queried via webservice calls. The MusicBug application makes extensive use of all three APIs. The login view registers the user in the MusicBug system, and stores the user data in the database. The Rate view stores the user's rating of a track, album, or artist. The review view stores the user's review of a track, album or artist. The favorite view saves an artist, track, or album in the user's favorites. The comment view places a comment generated by the user in the details page for an artist, track, album or review. The view profile view shows the information we have pulled from Facebook of the user, along with the user's ratings, reviews, and favorites. The add payment Info view allows the user to store their credit card information in the MusicBug database. The buy item view directs the user to an amazon page where they can purchase the product. The select theme view changes

the color scheme of the MusicBug application. The remove review view deletes a review from the system. The remove review view is only available to users with moderator status. The register, favorite, and comment views each use the Facebook API to interact with their Facebook account, the other views only interact with the models. Each view directly implements a use case of MusicBug as described in the System Requirements Document.

Within the rate view, artist ratings works differently than track and album ratings. For track and album the user gives a rating between 1 and 5 stars, and the average of all ratings is displayed on that track or album's detail page. Artists cannot be rated directly, instead an algorithm is used to determine an artists rating based on the ratings of all tracks and albums by the artist. The artist rating algorithm is described mathematically below:

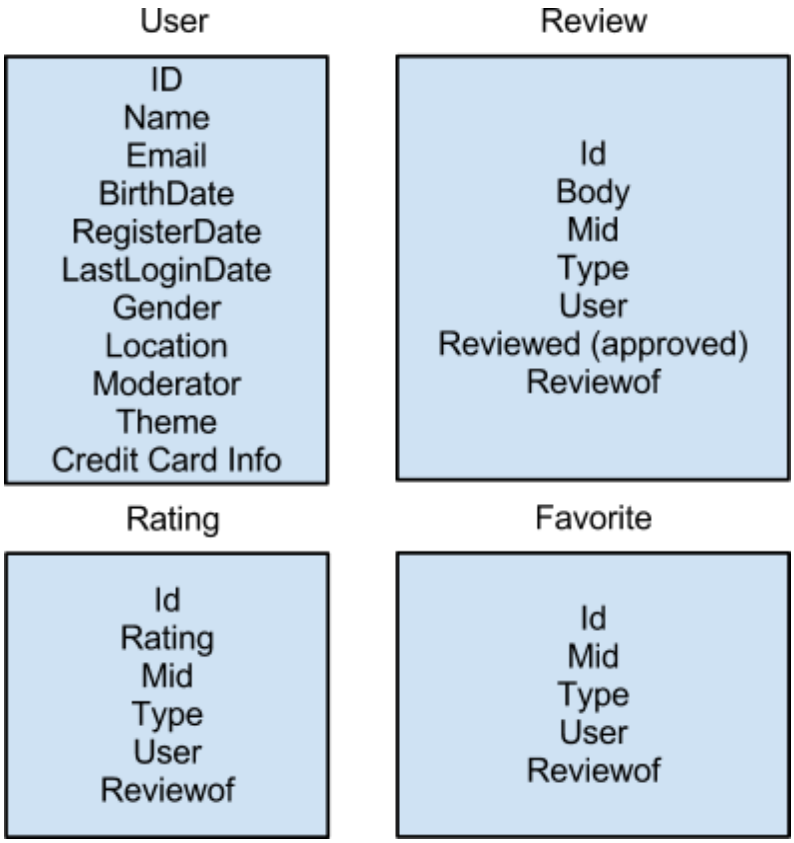
Step 1: Take the average rating of all tracks by the artist. Let this be TAvg.
Step 2: Take a special weighted average of all tracks by the artist. Let this be TAvgW.
Step 2a: If the number of ratings of a particular track exceeds 30% of the number of ratings of all tracks by this artist, only count the ratings for that track as if it had exactly 30% of the ratings.
Step 3: Take the average of TAvg and TAvgW. Let this be TRating
Step 4: Repeat steps 1 through 3 with album ratings instead of track ratings. This will give you ARating.
Step 5: Divide the number of ratings of tracks by the artist by the number of ratings of all items by the artist. Let this be TrackWeight.
Step 5a: If TrackWeight is less than 30% set TrackWeight to 30%.
Step 5b: If TrackWeight is more than 70% set TrackWeight to 70%.
Step 6a: Multiply TrackWeight by TrackRating.
Step 6b: Multiply (1 - TrackWeight) by AlbumRating.
Step 6c: Add the results of steps 6a and 6b together and divide by 2. This is the total artist rating.

The reason this algorithm is used, instead of a simpler one is to give higher ratings for artists who have a more diverse selection of music reviewed highly as opposed to an artist who has very few songs or albums rated very highly.

3. Persistent Data Design

3.1 Database Descriptions

All user database is kept in a PostgreSQL database named Users. This database was created automatically by the Django framework. After creating the user related classes in the models portion of Django, a built in tool created the database to our specification. The schema for the Users database is modeled below:

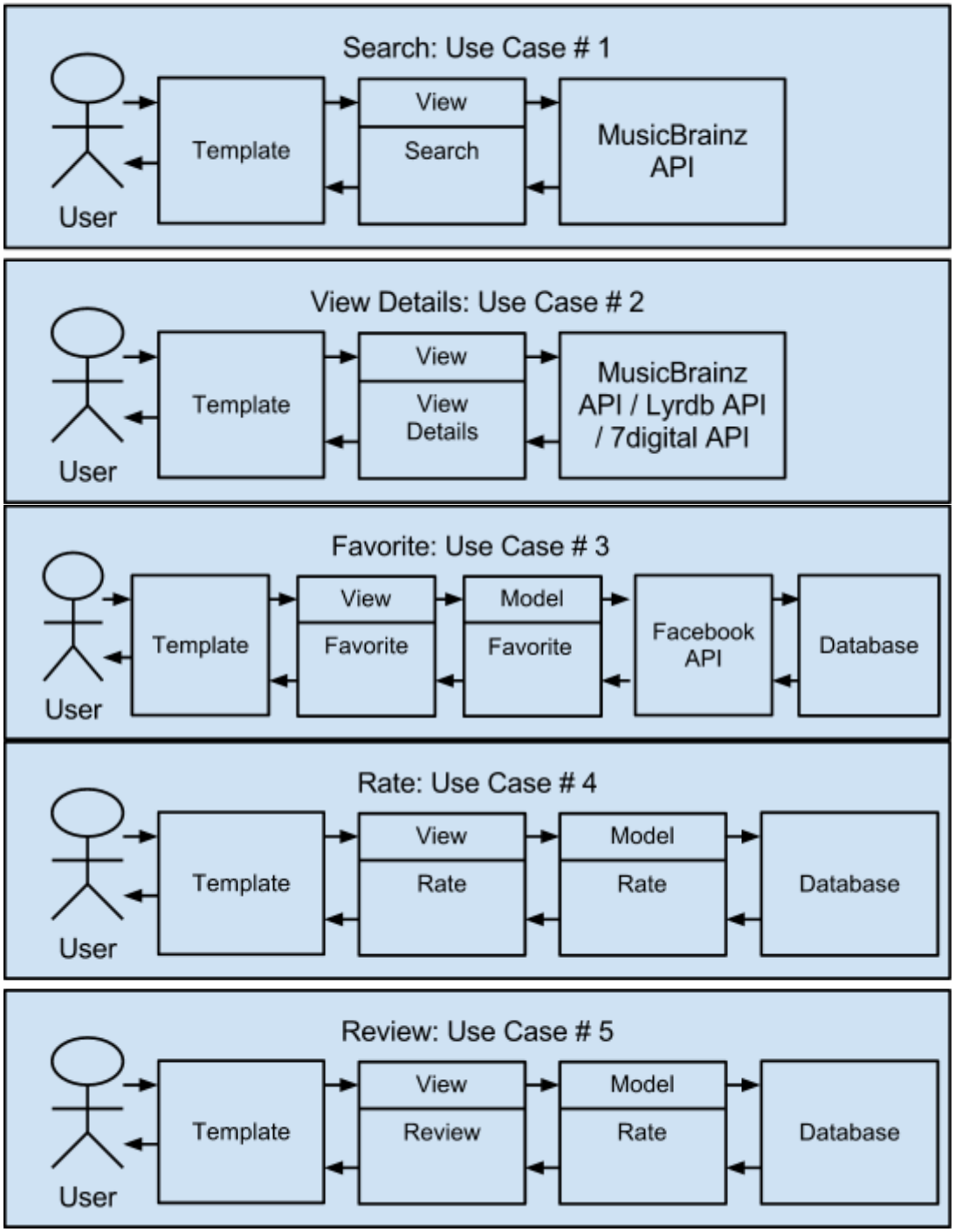


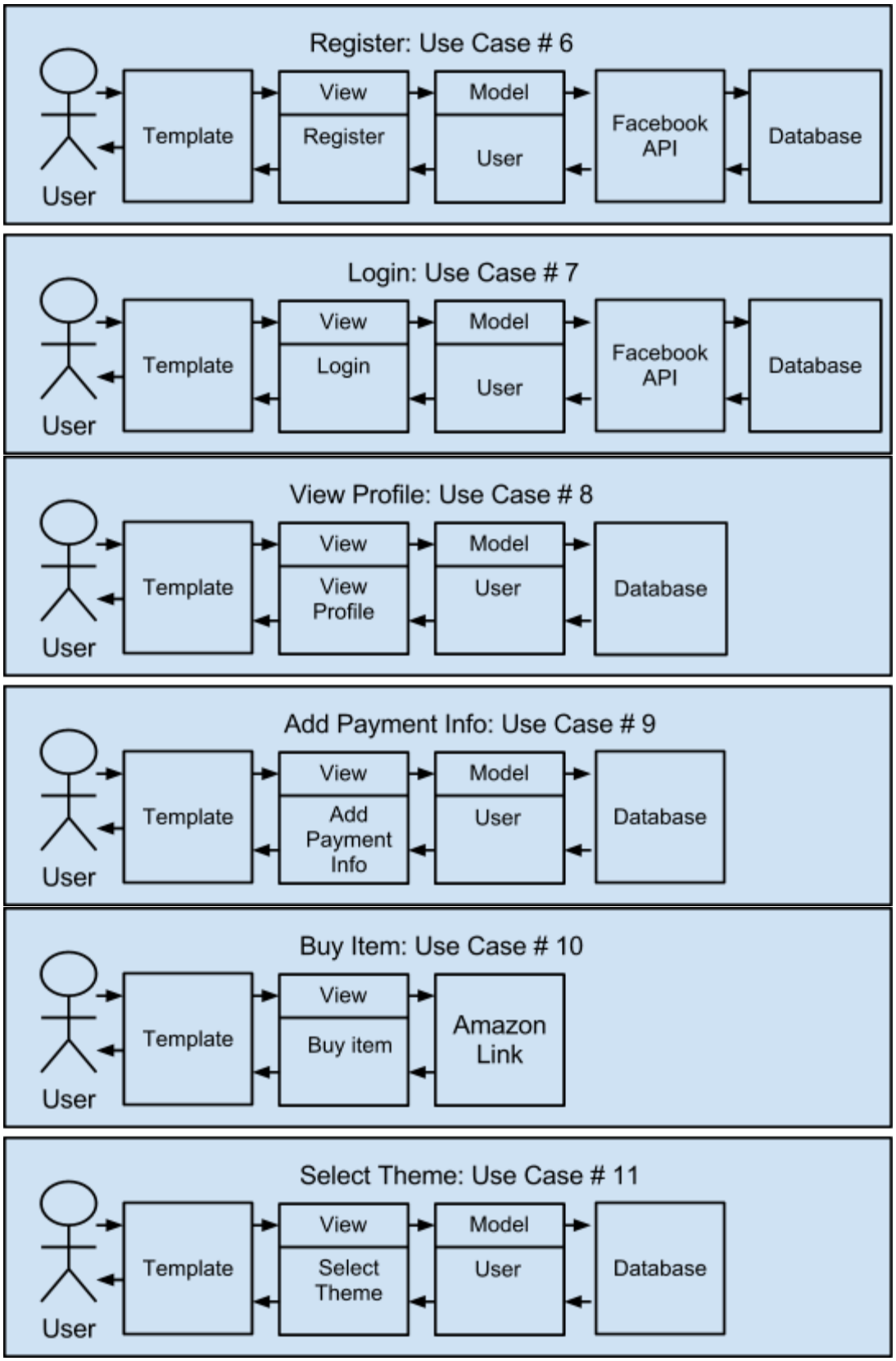
The MID fields on review, rating and favorite are the MusicBrainz ID.

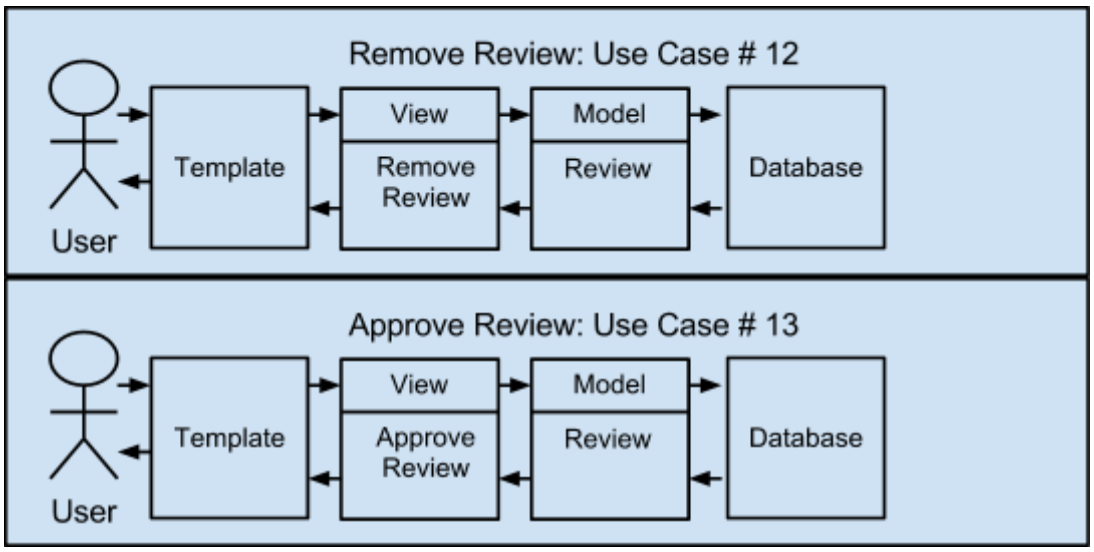
4. Requirements Matrix

Please refer to the System Requirements Specification for details regarding the corresponding use cases.

MusicBug Requirements Matrix




Register: Use Case # 6
User
Template
View
Register
Model
User
Facebook API
Database
Login: Use Case # 7
User
Template
View
Login
Model
User
Facebook API
Database
View Profile: Use Case # 8
User
Template
View
View Profile
Model
User
Database
Add Payment Info: Use Case # 9
User
Template
View
Add Payment Info
Model
User
Database
Buy Item: Use Case # 10
User
Template
View
Buy item
Amazon Link
Select Theme: Use Case # 11
User
Template
View
Select Theme
Model
User
Database


Remove Review: Use Case # 12
User
Template
View
Remove Review
Model
Review
Database
Approve Review: Use Case # 13
User
Template
View
Approve Review
Model
Review
Database

5. Appendix A – Agreement Between Customer and Contractor

The customer agrees to a *Music Social Network* system with searching, browsing and detailed meta-data capabilities. See System Requirements Specification for more information. Additional features will be provided in further development spirals.

When and if future changes to this document occur a drafted new document will be created. Both a hard and electric copy of both versions will be presented to the client for review. Upon approval, the draft will be finalized and signed off by both parties.

Client

Name _____ Date _____
Print
Name _____ Date _____
Signature

Team

Name _____ Date _____
Print
Name _____ Date _____
Signature

Name _____ Date _____
Print
Name _____ Date _____
Signature

Name _____ Date _____
Print
Name _____ Date _____
Signature

Name _____ Date _____
Print
Name _____ Date _____
Signature

Name _____ Date _____
Print
Name _____ Date

_____
Signature

Name _____ Date

_____
Print

Name _____ Date

_____
Signature

6. Appendix B – Team Review Sign-off

This document has been collaboratively written by all members the team. Additionally, all team members have reviewed this document and agree on both the content and the format. Any disagreements or concerns are addressed in team comments below.

Team

Name _____ Date

_____
Print

Name _____ Date

_____
Signature

Comments

_____
_____
__

Name _____ Date

_____
Print

Name _____ Date

_____
Signature

Comments

_____
_____
__

Name _____ Date

_____
Print

Name _____ Date

_____
Signature

Comments

_____
_____

__

Name _____ Date _____
Print
Name _____ Date _____
Signature
Comments
_____
_____
__

Name _____ Date _____
Print
Name _____ Date _____
Signature
Comments
_____
_____
__

Name _____ Date _____
Print
Name _____ Date _____
Signature
Comments
_____
_____
__

7. Appendix C – Document Contributions

Sean Ehrig took the lead on the writing of this document accounting for the completion of 100%. Michael Cohen contributed to editing of the document.