Song Recommendation System

Software Design Specification

Version 1.0

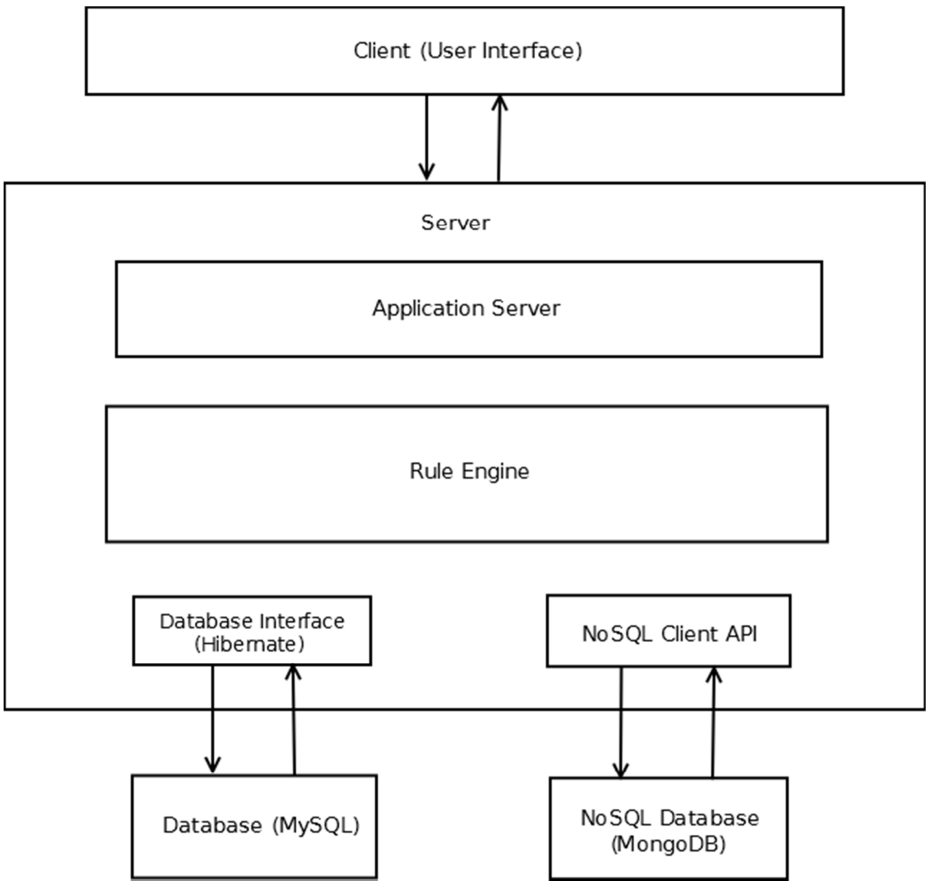
Tejas Patil
Nitin Narayana Prabhu
Pratik Lala
Ashwath Narayanan
Abhishek Kanchan

Prepared for
CS 441 – Distributed Object Programming using Middleware
Instructor: Mark Grechanik
Fall 2012

Contents

1. System Overview

1.1 Architecture Diagram



1.1.1 Architecture Description

- **Client**: This provides an interface for users to interact with the application. The client interface is used to send a request to the Application server. Requests include requesting detail pages for a song, artist, album, providing ratings, likes, shares or purchasing songs etc.
- **Application Server**: The Application Server handles the request from the client. The request is passed on to the Rule Engine for further processing. The Application Server also gets the output from the Rule Engine and sends the output to the client.

- **Rule Engine**: The Rule Engine takes the input from the application server. The Rule Engine has several rules which will be triggered based on the input data. Based on these rules, the Rule Engine will fetch data from the databases and send it to the Application Server.
- **Database Interface**: A middleware provides an interface to the Rule Engine to query and retrieve the data from a relational database. Hibernate is the preferred middleware here. It is a framework for mapping an object oriented domain model to traditional relational database. It provides functionality to query and retrieve data.
- **Database**: The database which will be used is MySQL. MySQL is a free relational database used for querying and storing data.
- **NoSQL Client API**: This constitutes a client that will use APIs to fetch data from a NoSQL database.
- **NoSQL Database**: NoSQL databases do not require any kind of relational model for data structure. It is used for working with large amount of data as it is highly optimized. Generally, data in such databases are stored as <key, value> pairs. We have decided to use MongoDB as our NoSQL database.

1.2 Data

For the recommendation system, we had to obtain a large amount of data pertaining to songs, albums and artists. While there are multiple datasets available online, none of them had the number of metrics we desired to incorporate in our system. Therefore, we decided to randomly generate all the data that was required by our system. Approximately 500000 rows of songs, albums and artists were generated. Data pertaining to users was also randomly generated resulting in about 150000 rows. Overall approximately 225MB of data was generated.

1.3 Recommendation Logic

The logic for recommending songs, artists or albums to users was based on content-based filtering, an algorithm that is used in many commercial applications for recommendation purposes. The idea of this approach is as follows

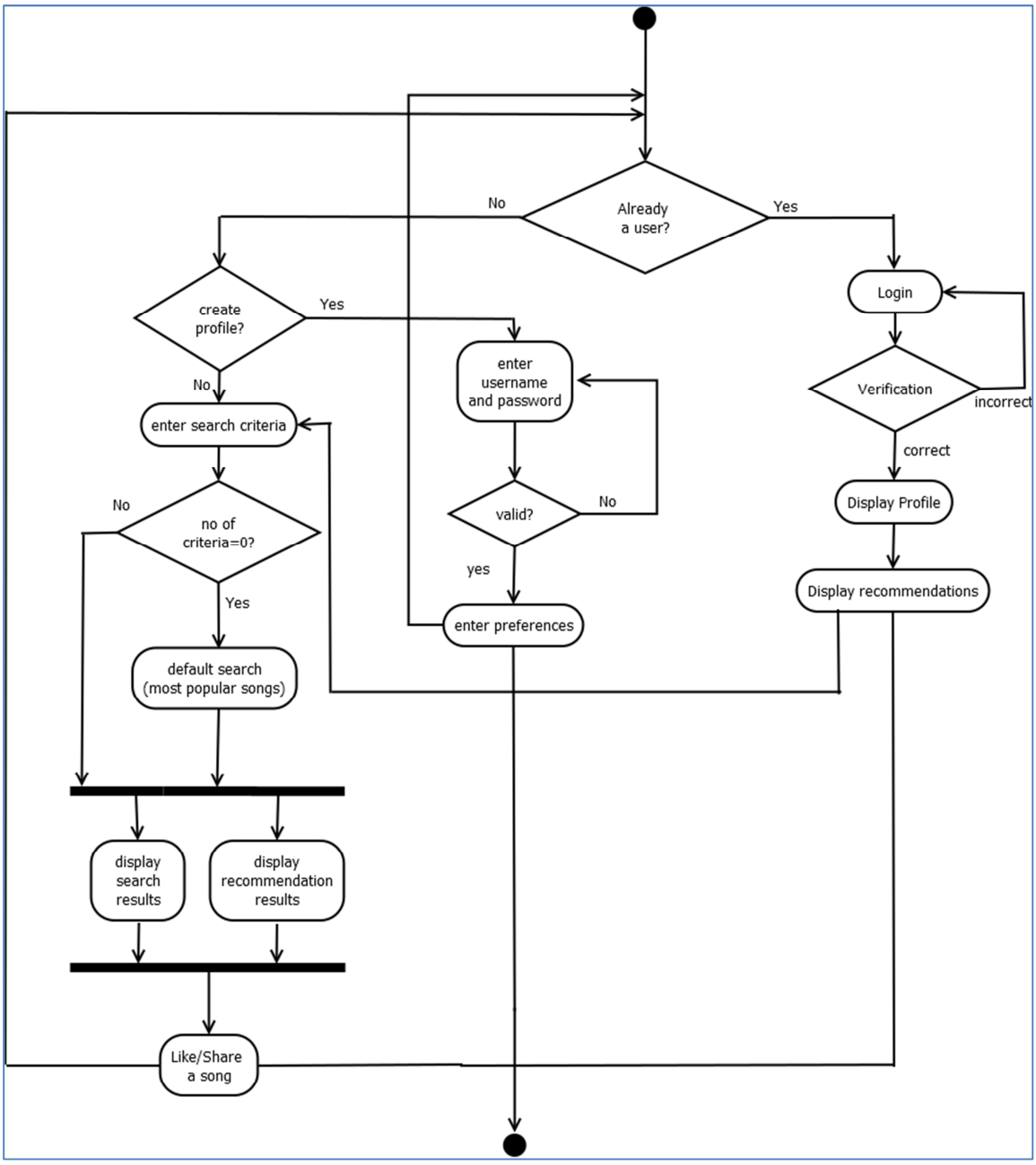
1. Determine the similarity between songs. The similarity can be computed on different metrics such as whether the songs belong to the same genre, whether the songs are from the same location, the songs have comparable hotness, danceability etc. The similarity between songs can then be obtained as a weighted average of all the metrics that were used to compare the songs. Similarly, we determine the similarity between artists and similarity between albums. After computing these similarity scores, we obtain a matrix of song-song, artist-artist and album-album similarity scores. However, for each song, album or artist, we are only considering the 20 most similar entities of the same type.
2. For every existing user in the system, we compare them with the song, album and artist data and compute an importance score by evaluating whether the user has liked, shared, played, rated or purchased the song or whether the item under consideration belongs to the user's preferred genre. Such a computation yields us a weighted score for each user and allows us to

construct user-song, user-album and user-artist importance tables with user as key and most important item as value.

3. When a user logs into the system, we can simply refer the importance tables to fetch the most important song, album or artist for the user and then use this data to fetch similar songs, albums and artists from the similarity matrices and this serves as our recommendations to the user.

2. UML Diagrams

2.1 Activity Diagram



2.1.1 Activity Diagram Symbols

- Black Circles denote initial/final state of activity
- Rectangle boxes denote an activity
- Diamond-shaped boxes denote a decision point in the activity

- Horizontal bars denote activities that would occur simultaneously

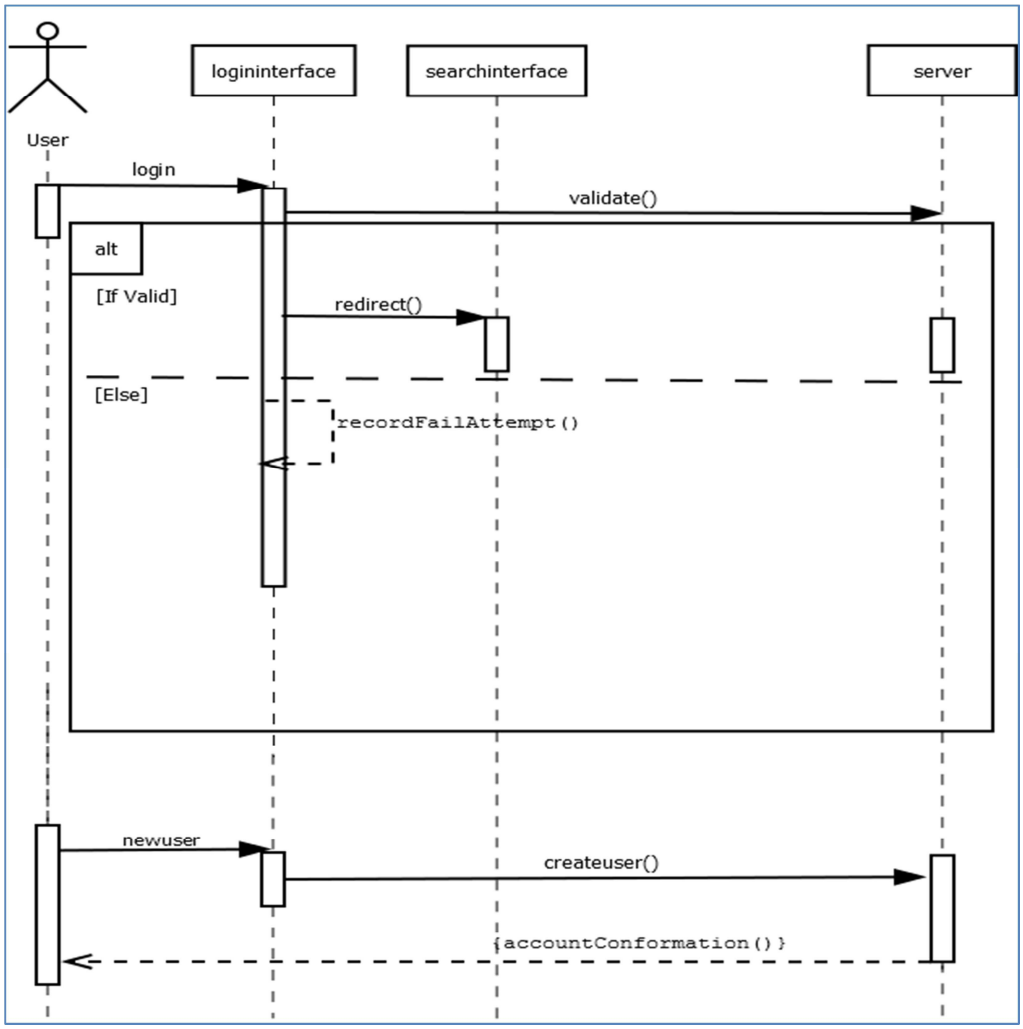
2.1.2 Activity Diagram Description

There are basically three types of paths that a user can take:

1. If the user already has a user profile, then when the user logs in, the system displays profile data and recommendations to him based on his/her preferences.
2. If the person is not a user, then the system asks the user if he/she wants to create a profile. Based on the user's answer there are 2 possibilities:
    a. If the person wants to create a profile, then the system asks the user to enter a username and password. The username and password should be valid. If they are not valid, then the system asks the user to re-enter the username and password.
    b. If the person doesn't want to create a profile, then the system asks the person to enter search criteria. If no criterion is entered, then the system will display a subset of the highest rated songs to the user. If the user already has a user profile, then the system will display a subset of the highest rated songs based on the user preference. Otherwise search and recommendations are performed based on the criteria specified by the user.
3. After any of these three paths, the user can either go to the final state or go back to the initial state from where a new path can be taken again.

2.2 Sequence Diagrams

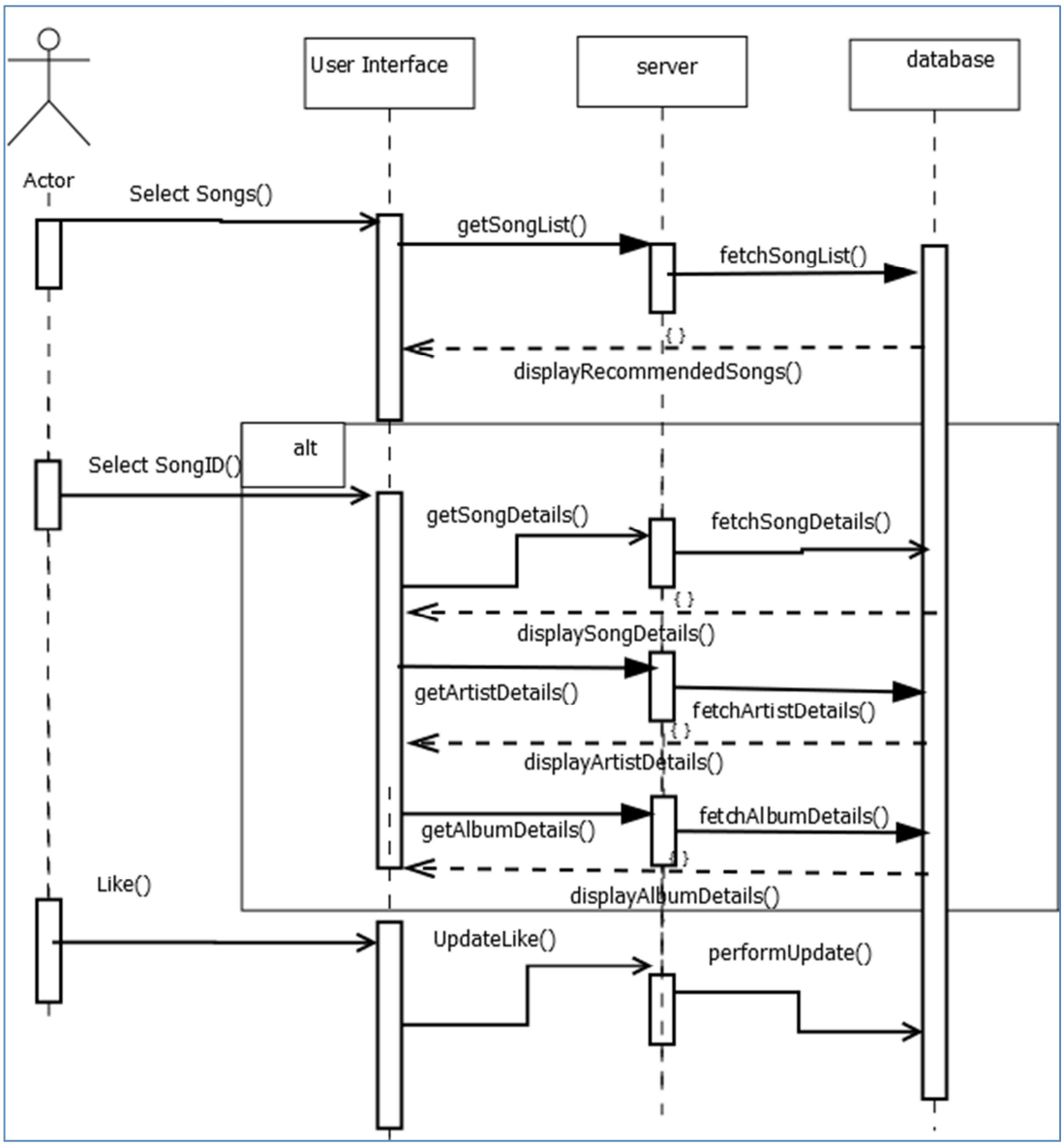
2.2.1 UserAccountLogin



Description:

1. The user logs in using the client interface. Once the credentials are provided, the details are validated by the login service.
2. If the user credentials are validated successfully, the user is recommended with songs/albums/artists based on his selection on the user interface.
3. If the credentials are invalid, then the user is asked to re-enter his/her credentials.

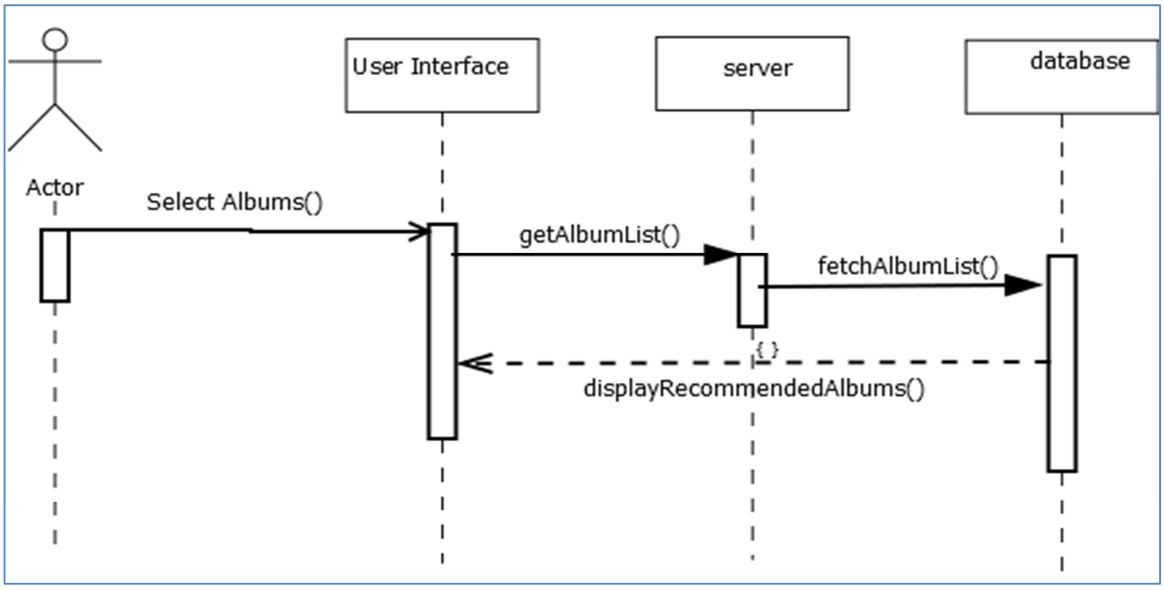
2.2.2 Song Recommendations



Description:

1. Once the user's credentials are validated, he/she can get his recommended songs.
2. The user request is sent from user interface to the server**.** The server processes the request and retrieves relevant song information from the song database.
3. The **displayRecommendedSongs()** return a list of recommended songs for the user.
4. User can select a particular songID to get its songdetails . The **displaySongDetails()** returns details of selected songID by the user.
5. The user can also get artist and album information about that particular song. The **displayArtistDetails()** and **displayAlbumDetails()** performs this operation.
6. User's **like** operation performs update of like information for that particular song.

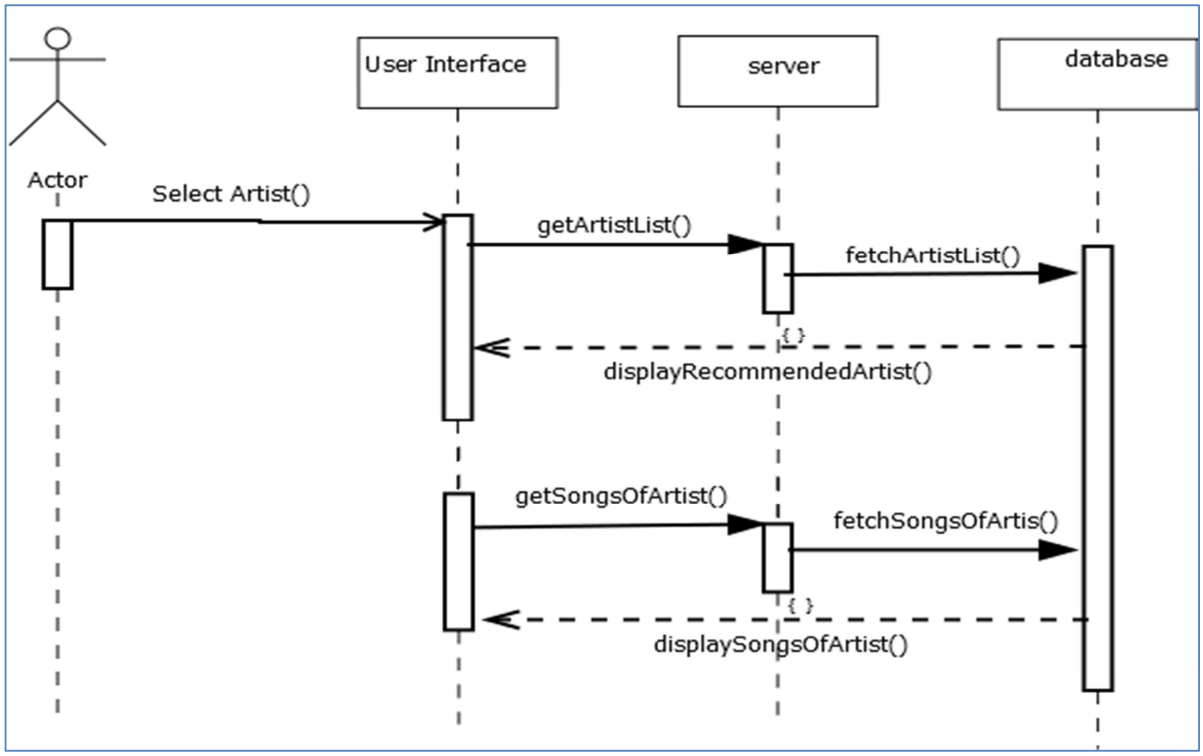
2.2.3 Album Recommendations



Description:

1. Once the user's credentials are validated, the user interface displays his profile along with option to choose from song, album, artist recommendation
2. The **displayRecommendedAlbums**() provided the user with the list of recommended songs. The recommendation provided is based on content based filtering of Album details based on his likes and preferences.

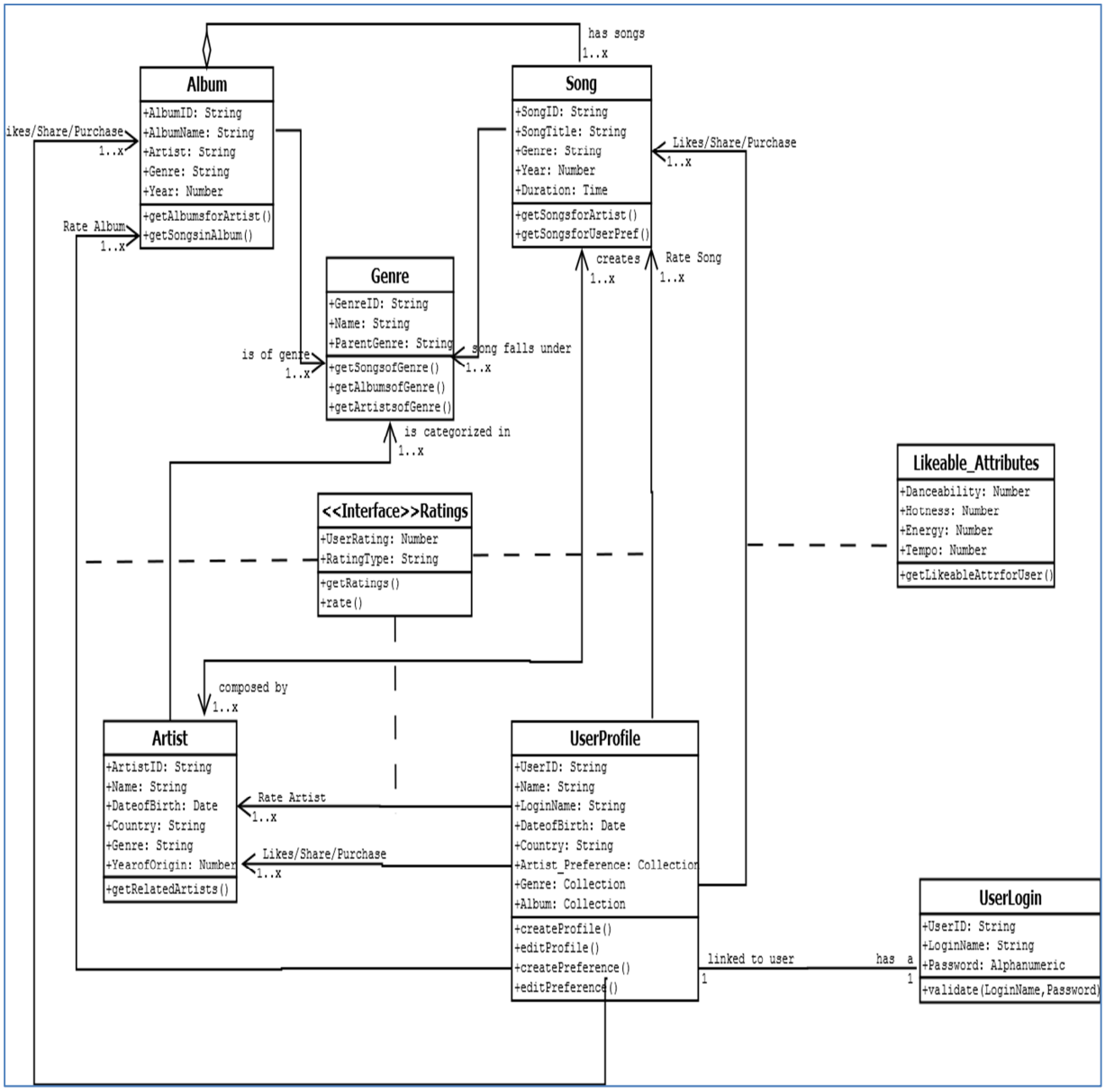
2.2.4 Artist Recommendations

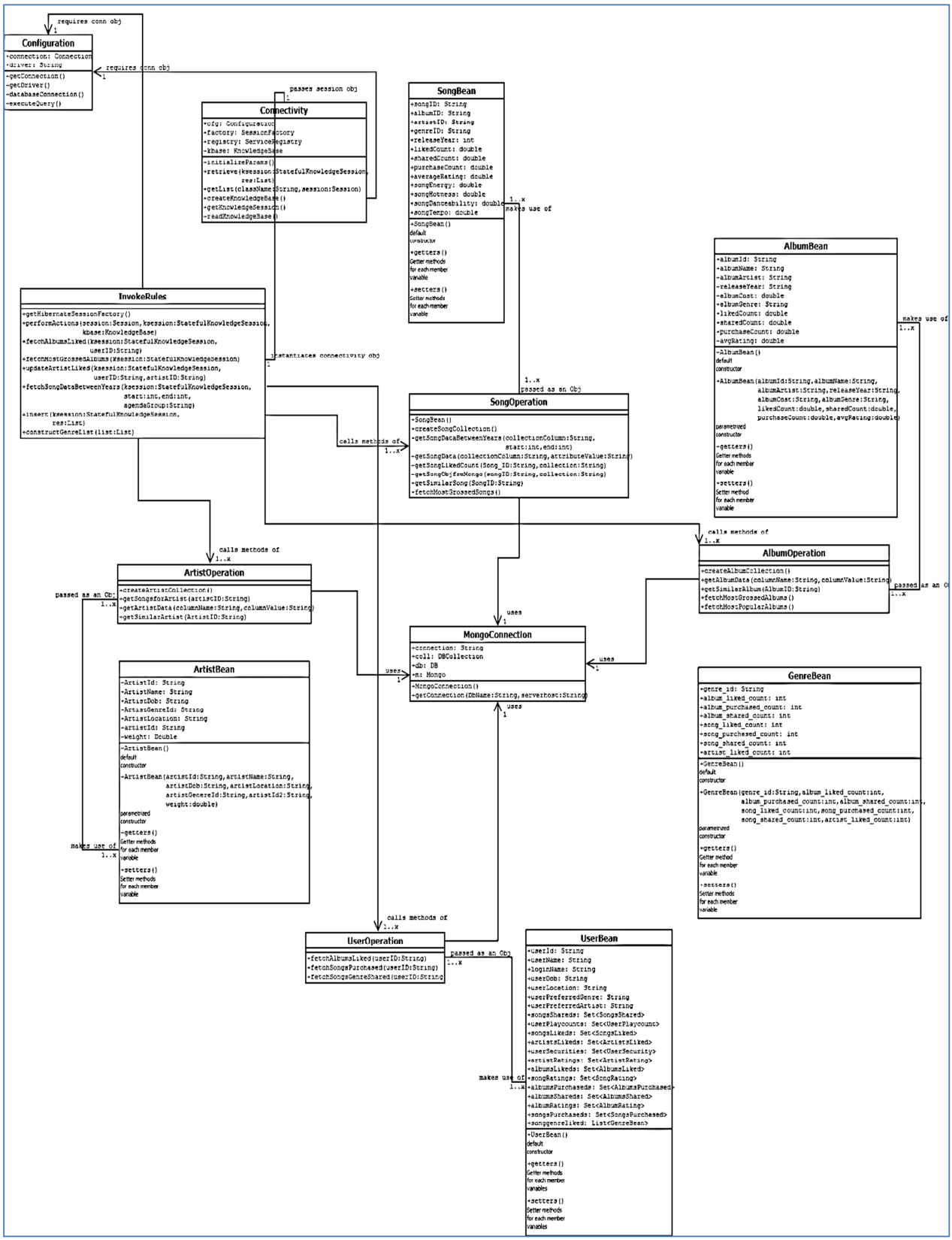


Description:

1. Once the user's credentials are validated, the is provided with option to choose from song, album, artist recommendation
2. The **displayRecommendedArtist**() provided the user with the list of recommended artists. The recommendation provided is based on content based filtering of Artist details based on his likes and preferences.
3. The user can get the list of songs by this particular Artist. The **displaySongsOfArtist()** performs this operation.

2.3 Class Diagram




Configuration
-connection: Connection
-driver: String
-getConnection()
-getDriver()
-databaseConnection()
-executeQuery()
requires conn obj
Connectivity
-cfg: Configuration
-factory: SessionFactory
-registry: ServiceRegistry
-kbase: KnowledgeBase
-initializeParams()
-retrieve(ksession:StatefulKnowledgeSession, res:List)
-getList(className:String,session:Session)
-createKnowledgeBase()
-getKnowledgeSession()
-readKnowledgeBase()
passes session obj
SongBean
+songID: String
+albumID: String
+artistID: String
+genreID: String
+releaseYear: int
+likedCount: double
+sharedCount: double
+purchaseCount: double
+averageRating: double
+songEnergy: double
+songMotness: double
+songDanceability: double
+songTempo: double
+SongBean()
default constructor
+getters()
Getter methods for each member variable
+setters()
Setter methods for each member variable
makes use of
AlbumBean
+albumId: String
+albumName: String
+albumArtist: String
+releaseYear: String
+albumCost: double
+albumGenre: String
+likedCount: double
+sharedCount: double
+purchaseCount: double
+avgRating: double
-AlbumBean()
default constructor
+AlbumBean(albumId:String,albumName:String, albumArtist:String,releaseYear:String, albumCost:String,albumGenre:String, likedCount:double,sharedCount:double, purchaseCount:double,avgRating:double)
parametrized constructor
-getters()
Getter methods for each member variable
+setters()
Setter method for each member variable
InvokeRules
+getHibernateSessionFactory()
+performActions(session:Session,ksession:StatefulKnowledgeSession, kbase:KnowledgeBase)
+fetchAlbumsLiked(ksession:StatefulKnowledgeSession, userID:String)
+fetchMostGrossedAlbums(ksession:StatefulKnowledgeSession)
+updateArtistLiked(ksession:StatefulKnowledgeSession, userID:String,artistID:String)
+fetchSongDataBetweenYears(ksession:StatefulKnowledgeSession, start:int,end:int, agendaGroup:String)
+insert(ksession:StatefulKnowledgeSession, res:List)
+constructGenreList(list:List)
instantiates connectivity obj
calls methods of
passed as an Obj
SongOperation
+SongBean()
+createSongCollection()
+getSongDataBetweenYears(collectionColumn:String, start:int,end:int)
+getSongData(collectionColumn:String,attributeValue:String)
+getSongLikedCount(Song_ID:String,collection:String)
+getSongObjfromMongo(songID:String,collection:String)
+getSimilarSong(SongID:String)
+fetchMostGrossedSongs()
AlbumOperation
+createAlbumCollection()
+getAlbumData(columnName:String,columnValue:String)
+getSimilarAlbum(AlbumID:String)
+fetchMostGrossedAlbums()
+fetchMostPopularAlbums()
ArtistOperation
+createArtistCollection()
+getSongsForArtist(artistID:String)
+getArtistData(columnName:String,columnValue:String)
+getSimilarArtist(ArtistID:String)
MongoConnection
+connection: String
+coll: DBCollection
+db: DB
+m: Mongo
+MongoConnection()
+getConnection(DBName:String,serverhost:String)
uses
ArtistBean
-ArtistId: String
+ArtistName: String
+ArtistDob: String
-ArtistGenreId: String
+ArtistLocation: String
+artistId: String
-weight: Double
-ArtistBean()
default constructor
+ArtistBean(artistId:String,artistName:String, artistDob:String,artistLocation:String, artistGenreId:String,artistId2:String, weight:double)
parametrized constructor
-getters()
Getter methods for each member variable
+setters()
Setter methods for each member variable
GenreBean
+genre_id: String
+album_liked_count: int
+album_purchased_count: int
+album_shared_count: int
+song_liked_count: int
+song_purchased_count: int
+song_shared_count: int
+artist_liked_count: int
+GenreBean()
default constructor
+GenreBean(genre_id:String,album_liked_count:int, album_purchased_count:int,album_shared_count:int, song_liked_count:int,song_purchased_count:int, song_shared_count:int,artist_liked_count:int)
parametrized constructor
+getters()
Getter method for each member variable
+setters()
Setter methods for each member variable
UserOperation
+fetchAlbumsLiked(userID:String)
+fetchSongsPurchased(userID:String)
+fetchSongsGenreShared(userID:String
UserBean
+userId: String
+userName: String
+loginName: String
+userDob: String
+userLocation: String
+userPreferredGenre: String
+userPreferredArtist: String
+songsShareds: Set<SongsShared>
+userPlaycounts: Set<UserPlaycount>
+songsLikeds: Set<SongsLiked>
+artistsLikeds: Set<ArtistsLiked>
+userSecurities: Set<UserSecurity>
+artistRatings: Set<ArtistRating>
+albumsLikeds: Set<AlbumsLiked>
+songRatings: Set<SongRating>
+albumsPurchaseds: Set<AlbumsPurchased>
+albumsShareds: Set<AlbumsShared>
+albumRatings: Set<AlbumRating>
+songsPurchaseds: Set<SongsPurchased>
+songgenreliked: List<GenreBean>
+UserBean()
default constructor
+getters()
Getter methods for each member variables
+setters()
Setter methods for each member variables

2.3.1 Class Description

The purpose of the class diagram is to model the static view of this application. The class diagram is the one which can be directly mapped with instances of the system.

1) **Song**: This class represents the basic song entity which is associated with the album and the artist class.

   Attributes:
   1) **SongID**: This attribute identifies each song uniquely and is used for linking the songs to their corresponding album, the artist as well as to link the user profile with a list of songs that he/she likes.
   2) **SongTitle**: The title of the song denotes the name of the song.
   3) **Genre**: This represents the genre to which the song belongs.
   4) **Year**: This represents the year in which the song was released.
   5) **Duration**: This represents the length of time for which the song lasts.

   Methods:
   1) **getSongsforArtist()**: This method retrieves the list of all songs for a particular artist. If the optional album argument is given then it will retrieve all the songs for that particular album and artist.
   2) **getSongsforUserPref()**: This method gives the list of songs that have been queried for various preferences saved in a user profile.

2) **Album**: This class represents the album which consists of one or more songs. An album can have one artist or more than one artist associated with it. Also the assumption of the system is that an album will have at least one song in it.

   Attributes:
   1) **AlbumID**: This attribute uniquely identifies an album.
   2) **AlbumName**: This is used for representing the name of the album.
   3) **Artist**: This attribute gives us the artist for a particular album.
   4) **Genre**: This attribute signifies the genre that the album belongs to.
   5) **Year**: This attribute tells us the year in which the album was released.

   Methods:
   1) **getAlbumsforArtist()**: This function retrieves the list of all albums released by a particular artist since their inception. It takes the artist name as a parameter.
   2) **getSongsinAlbum()**: This function will retrieve a list of all the songs in a particular album. The input parameter is the album name. Assumption here is that the same album name is not linked to more than one artist.

3) **Artist**: This class represents the artist object that composes a song and creates an album. Each of these artists can be saved to the user profile of a particular user.

   Attributes:

1) **ArtistID**: This attribute uniquely identifies the artist in our song recommendation system. The artists of the same band will have different ArtistID's.
2) **Name**: It represents the full name of the artist.
3) **DateofBirth**: Represents the date of birth of the artist and is denoted as a date type.
4) **Country**: This attribute signifies the country of origin of the artist.
5) **Genre**: This attribute tells us the category of music to which artists closely associates themselves with.

Methods:

1) **getRelatedArtists()**: This function takes in an artist as an input parameter and displays the list of closely related artists.

4) **UserProfile**: This class represents the user as an entity who has his/her individual profile and can create his/her customized preference parameter in his profile to get song recommendations that closely match his/her choice in music.

Attributes:

1) **UserID**: This attribute is used to uniquely identify a user.
2) **Name**: This attribute stores the full name of the user in his profile.
3) **LoginName**: This attribute is a unique one which is used to map a user with his login credentials.
4) **DateofBirth**: It is used to store the user's date of birth in date format.
5) **Country**: This denotes the country to which the user belongs.
6) **Artist_Preference**: This attribute stores the list of artists in a collection who all are preferred by the particular user.
7) **Genre**: This attribute stores the list of genre values that is favored by the user.
8) **Album**: This attribute stores the user's most liked album or albums which assists the recommendation engine to recommend songs and albums to the user.

Methods:

1) **createProfile()**: This method is used to create the profile of the user where he/she can input his basic information required by the system to instantiate him/her as a user.
2) **editProfile()**: This method is used to edit the profile of the user wherein he/she can modify or delete information pertaining to his/her user profile.
3) **createPreference():** This functionality is used by the user to input and save his preferences like favorite genre, album, artist etc.
4) **editPreference():** This functionality allows the user to modify the music preferences that he had saved earlier according to his choice.

5) **Ratings:** The Ratings class is the association class which creates the specific ratings instance, mapped from the user-preference to each of the Artist, Song or the Album class.

Attributes:

1) **UserRating**: This attribute represents the rating given by an individual user to any song, album or artist in our music database.
2) **RatingType**: This attribute signifies the type of the rating i.e whether it is a song rating or an album rating or an artist rating.

Methods:

1) **getRatings()**: This function takes in either one of song, album or an artist as the input parameter and gives the overall rating of the entity as calculated by the system.
2) **rate()**: This function is used by the user to rate a song/album/artist as per his/her liking.

6) **Genre**: The Genre class is used to instantiate each of the genre's to which a song, album or an artist broadly fall into.

Attribute:

1) **GenreID**: This attribute is used to uniquely represent a genre of our music recommendation system.
2) **Name**: This represents the actual name of the Genre for eg. Rock, Pop, Classical etc.
3) **ParentGenre**: This attribute points to the parent genre of the current genre.

Methods:

1) **getSongsofGenre()**: This function gives us a list of songs from the database that fall under a specified genre. It takes the genre name as an input parameter.
2) **getAlbumsofGenre()**:This function gives us a list of albums from the database that fall under a specified genre. It takes the genre name as an input parameter.
3) **getArtistsofGenre()**: This function takes in the genre name as the input and gives the list of all artists who can be classified under one genre.

7) **Likeable_Attributes**: This is an association class which stores the additional attributes of a song.

Attribute:

1) **Danceability**: It is a number attribute which is used to define the danceability of a song on a predefined scale.
2) **Hotness**: It is a number attribute which is used to define the hotness of a song on a predefined scale.
3) **Energy**: It is a number attribute which is used to define the Energy of a song on a predefined scale.
4) **Tempo**: It is a number attribute which is used to define the Tempo of a song on a predefined scale.

8) **UserLogin**: This class represents the user login entity that is associated with a particular user of the system.

Attribute:

1) **UserID**: This string attribute maps the UserLogin object to its corresponding UserProfile.
2) **LoginName**: This attribute represents the login name used by the user for logging into the Music Recommendation system.
3) **Password**: This is a password attribute which is used to authorize the user by the system during the login process.

9) **InvokeRules**: Creates knowledge sessions for firing each rule. Used for preparing the Statement objects.

10) **Configuration:** This class has methods which create a connection with the MySQL database once for each separate user. Both objects of Connectivity and InvokeRules use this class.

11) **Connectivity**: Used by the Servlets to communicate with the system. It has methods which create the KnowledgeBase and pass a session to InvokeRules object. It also connects the DROOLS system with the hibernate configuration to interact with the database.

12) **MongoConnection**: This class contains methods to establish connection with the MongoDb instance. The connection is created once and is used by all the objects.

13) **SongBean**: Represents a single Song entity of the system. Has attributes like songID, albumID, artistID, genreID, releaseYear, likedCount, sharedCount, purchasedCount, averageRating, songEnergy, songHotness, songDanceability and songTempo. Contains the default and parameterized constructor as well as Getter and Setter methods for all the member variables.

14) **SongOperation**: This class is used as a client by various other objects to interact with MongoDb song data collection. Consists basic functions like createSongCollection(), getSongDatabetweenYears(), getSongData(), getSongLikedCount() etc.

15) **ArtistBean:** Represents a single Artist entity. Has attributes like artistID, artistName, artisDOB, artistGenreID, artistLocation. Contains the default and parameterized constructor as well as Getter and Setter methods for all the member variables.

16) **ArtistOperation:** This class is used as a client by various other objects to interact with MongoDb artistdata collection. Consists basic functions like createArtistCollection(), getSongsforArtist(), getArtistData(), getSimilarArtist().

17) **AlbumBean:** Represents a single Album entity. Has attributes like albumID, albumName, albumArtist, releaseYear, albumCost, albumGenre, likedCount, sharedCount, purchasedCount, avgRating. Contains the default and parameterized constructor as well as Getter and Setter methods for all the member variables.

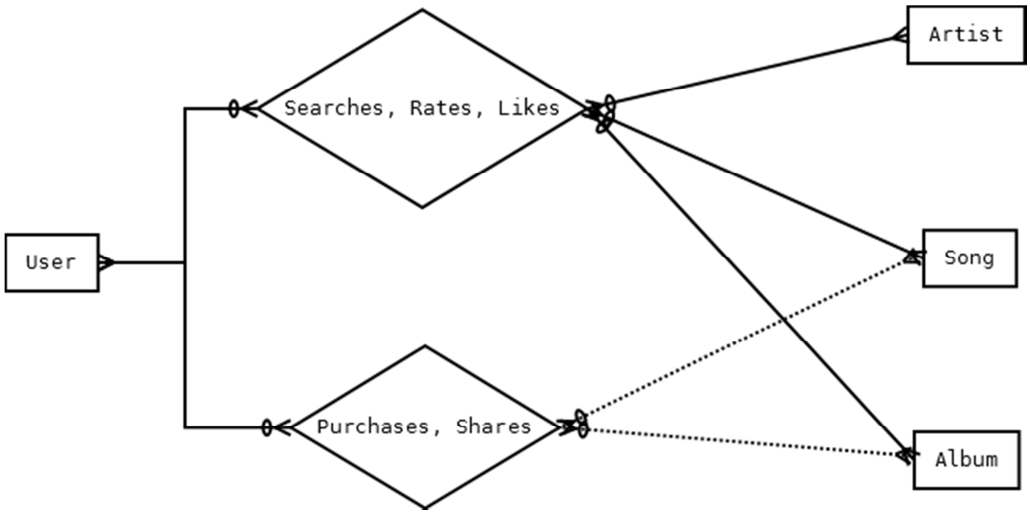
18) **AlbumOperation:** This class is used as a client by various other objects to interact with MongoDb albumdata collection. Consists basic functions like createAlbumCollection(), fetchMostPopularAlbums(), fetchMostGrossedAlbums(), getAlbumData(), getSimilarAlbum().
19) **GenreBean:** Represents a Genre entity. Has attributes like genre_id, album_liked_count, album_purchased_count, album_shared_count, song_liked_count, song_purchased_count, song_shared_count, artist_liked_count. Contains the default and parameterized constructor as well as Getter and Setter methods for all the member variables.

2.3.2 Associations between the classes

1. The Song class and the Album class has a basic aggregation association between them which tells that an album can have one or many songs associated with it but for a Song instance to exist it need not necessarily require a particular album. The assumption here is that if an album is instantiated, it must have atleast one song in it.
2. The Song, Album and the Artist class are associated in a unidirectional manner with the Genre class wherein each one of them can be categorized into one specific or more than one Genre's.
3. The Artist and the Song class have a bidirectional association between them where a song can be created by one or more artists and an Artist can create one or more songs in his career.
4. The UserProfile class is associated with each of the Song, Album or the Artist class in a uni-directional manner where the UserProfile instance can Like or Share one or more song, album or artist.
5. The UserProfile also has an association with each of the Song, Artist and Album class which allows him to rate a particular song, artist or an album.
6. The Ratings class is an association interface that is implemented whenever a user of the system wants to rate a song/album/artist.
7. The LikeableAttributes is an association class that gives more detailed attributes of a particular song when a User likes or shares a particular song.
8. The UserLogin class and the UserProfile class are associated in a bidirectional manner where each user has only one user login instance and both UserLogin instance and the User profile instance know about each other's existence.
9. The InvokeRules requires a single configuration object which is used to establish connection with the MySQL database. So it has a one to one unidirectional association with the Configuration class.
10. The Connectivity and the Configuration class has a one to one unidirectional association which implies that whenever the DROOLS system or any other system tries to link with hibernate, it creates a one-to-one mapping with the connectivity class.
11. The Connectivity and the InvokeRules class has one-to-one bidirectional association in a way that the Connectivity creates the KnowledgeBase and passes a single session object whenever any single rule is fired from InvokeRules class.
12. Each of the Bean Class with the corresponding Class Operation has a many-to-many bidirectional association where any number of operations defined in the Operation class can use any number of objects of the Bean class as a parameter or a return type.
13. Each of the Operation classes defined which uses the Bean object will make use of a single MongoConnection object where the association between each of the Operation class and MongoConnection is unidirectional.

3. Database Model

3.1 Entity – Relationship diagram (ER diagram)



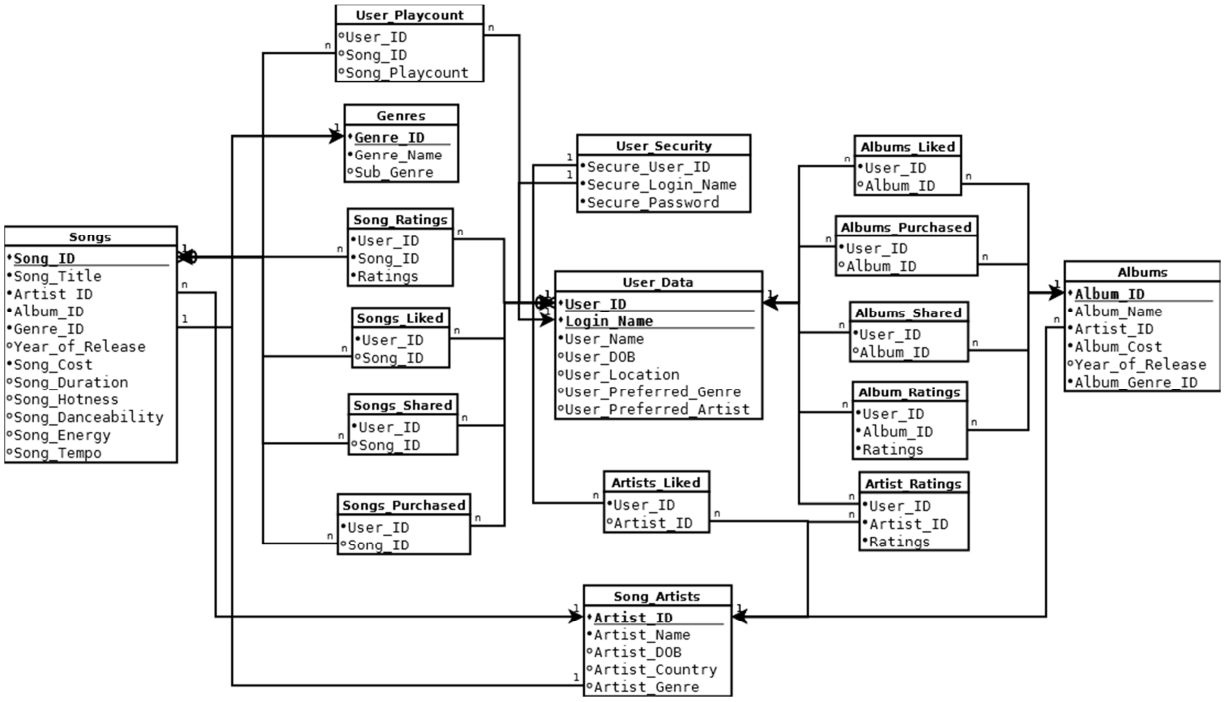
3.1.1 Diagram Notations

- Rectangle boxes denote entities in the system. An entity is an object which exists in the real world and can be differentiated from other entities.
- Diamond-shaped boxes denote the relationship that exists between entities.
- The image  is used to denote multiple instances of entities.
- The image  denotes that entity on left is related to zero or multiple instances of entity on right

3.1.2 ER Diagram Description

The entity 'User' has the relationship 'searches, likes, rates' with entities 'Artist', 'Album' and 'Song'. This implies that a User can search, like or rate an Artist, Album or Song. 'User' also has the relationship 'purchase, share' with entities 'Album' and 'Song'. This means that users have the option to purchase or share albums or songs.

3.2 Database Model Diagram



3.2.1 Database Table Description

The description for the tables shown in the diagram above is as follows:

Artists: This table stores information pertaining to song artists.

Attribute	Description
Artist_ID	Unique ID, assigned to each artist
Artist_Name	Name of the artist
Artist_DOB	Date of Birth of the artist
Artist_Country	Country in which artist is born
Artist_Genre	Genre of the songs played by the artist

Albums: This table stores information pertaining to song albums.

Attribute	Description
Album_ID	Unique ID, assigned to each album
Album_Name	Name of the album
Artist_ID	Foreign key reference to the unique ID of the artist
Year_of_Release	Year in which the song was released
Album_Cost	Cost of the album
Album_Genre_ID	Genre to which the album belongs

Genre: This table stores information pertaining to different music genre.

Attribute	Description
Genre_ID	Unique ID, assigned to each genre
Genre_Name	Name of the genre
Parent_Genre	ID of the Parent genre for this genre

Songs: This table stores all the attributes of a song.

Attribute	Description
Song_ID	Unique ID, assigned to each song
Song_Title	Title of the song
Artist_ID	Foreign key reference to the unique ID of the artist
Album_ID	Foreign key reference to the unique ID of the album
Genre_ID	Foreign key reference to the unique ID of the genre
Year_of_release	Year in which song is released
Song_Cost	Cost of the song
Song_Duration	Duration of the song in seconds
Song_Hotness	Popularity measure of the song, ranging between 0 and 1
Song_Danceability	Danceability measure of the song, ranging between 0 and 1
Song_Energy	Energy Measure of the song, ranging between 0 and 1
Song_Tempo	Beats per minute measure of the song

User_Data: This table stores information pertaining to application users.

Attribute	Description
User_ID	Unique ID, assigned to each user
Login_Name	Login Name of the user
User_Name	ID of the Parent genre for this genre
User_DOB	Date of Birth of the user
User_Location	Country to which user belongs
User_Preferred_Genre	Foreign key reference to the unique ID of the genre, preferred by user
User_Preferred_Artist	Foreign key reference to the unique ID of the artist, preferred by user

User_Security: This table stores user login information.

Attribute	Description
Secure_User_ID	Foreign key reference to the unique ID of the user
Secure_Login_Name	Foreign key reference to the unique login name of the user
Secure_Password	Password specified by user

Song_Ratings: This table stores information about the ratings given by users to songs

Attribute	Description
User_ID	Foreign key reference to the unique ID of the user
Song_ID	Foreign key reference to the unique ID of the song
Ratings	Rating given by user to song

Album_Ratings: This table stores information about the ratings given by users to albums

Attribute	Description
User_ID	Foreign key reference to the unique ID of the user
Album_ID	Foreign key reference to the unique ID of the album
Ratings	Rating given by user to album

Artist_Ratings: This table stores information about the ratings given by users to artists

Attribute	Description
User_ID	Foreign key reference to the unique ID of the user
Artist_ID	Foreign key reference to the unique ID of the artist
Ratings	Rating given by user to artist

Songs_Liked: This table stores information about songs liked by the users

Attribute	Description
User_ID	Foreign key reference to the unique ID of the user
Artist_ID	Foreign key reference to the unique ID of the artist

Albums_Liked: This table stores information about albums liked by the users

Attribute	Description
User_ID	Foreign key reference to the unique ID of the user
Artist_ID	Foreign key reference to the unique ID of the artist

Artists_Liked: This table stores information about artists liked by the users

Attribute	Description
User_ID	Foreign key reference to the unique ID of the user
Artist_ID	Foreign key reference to the unique ID of the artist

Songs_Purchased: This table stores information about songs purchased by the users

Attribute	Description
User_ID	Foreign key reference to the unique ID of the user

Artist_ID	Foreign key reference to the unique ID of the artist

Albums_Purchased: This table stores information about albums purchased by the users

Attribute	**Description**
User_ID	Foreign key reference to the unique ID of the user
Artist_ID	Foreign key reference to the unique ID of the artist

Songs_Shared: This table stores information about songs shared by the users

Attribute	**Description**
User_ID	Foreign key reference to the unique ID of the user
Artist_ID	Foreign key reference to the unique ID of the artist

Albums_Shared: This table stores information about albums shared by the users

Attribute	**Description**
User_ID	Foreign key reference to the unique ID of the user
Artist_ID	Foreign key reference to the unique ID of the artist

User_Playcount: This table stores information about the number of times the user has listened to a song

Attribute	**Description**
User_ID	Foreign key reference to the unique ID of the user
Song_ID	Foreign key reference to the unique ID of the song
Song_Playcount	Number of times the user has played the song

4. MongoDB

NoSQL DEFINITION:

These are basically Next Generation Databases addressing some of the key points such as: being **non-relational, distributed, open-source** and **horizontally scalable**.
Their most inherent characteristics being schema free, easy replication support, simple API, eventually consistent / BASE (not ACID) and ability to manage huge amount of data.

4.1 Advantages of NoSQL:

1) Elastic scaling: For years database and system administrators have relied on " scaling up" –buying bigger servers as database load increases, rather than "scaling out" – distributing the database across multiple hosts as load increases. As transaction rate increases and availability requirements increases as databases move into the cloud or other virtualized environments, the economic advantages of scaling out becomes far more superior than scaling up.
2) Big Data: Just as transaction rates have grown out of recognition over the last decade, the volumes of data that are being stored also have increased massively. RDBMS's have been optimized and refined to match these increases, but as with transaction rates the constraints of data volumes that can be practically managed by a single RDBMS are becoming intolerable for some applications.
3) Low Maintenance operations: NoSQL databases are generally designed from the ground up to require less management, automatic repair, data distribution and simpler data models leading to lower administration and tuning requirements, making it somewhat an ideal choice as a datastore for flexible lightweight web applications .
4) Flexible Data models: Change management is a big headache for large production RDBMS. Even minor changes to the data model of an RDBMS have to be carefully managed and may necessitate downtime or reduced service levels.
   NoSQL databases have far more relaxed - or even nonexistent data model restrictions. NoSQL Key Value stores and document databases allow the application to store virtually any structure it wants in a data element, allowing the applications to iterate faster over a period of time.
5) Economics: NoSQL databases typically use clusters of cheap commodity servers to manage the exploding data and transaction volumes, while RDBMS tends to rely on expensive proprietary servers and storage systems. The result is that the cost per gigabyte or transaction/second for NoSQL can be many times less than the cost for RDBMS, allowing you to store and process more data at a much lower price point.

4.2 Reasons for adopting MongoDB as the NoSQL datastore for our application:

MongoDB is a scalable, high performance, open source NoSQL database. Written in C++, MongoDb features:

1) Document oriented storage: JSON style documents with dynamic schemas offer simplicity and flexibility. Data in MongoDB is stored in collections. Collections are a way of storing related data (think relational tables, without the schema). Collections contain documents which have in turn keys, another name for attributes. Documents can embed a tree of associated data, e.g. tags, comments and the like instead of storing them in different MongoDB documents. This is not specific to MongoDB, but document databases in general.

2) Index support: Indexes provide high performance read operations for frequently used queries. Indexes are particularly useful where the total size of the documents exceeds the amount of available RAM. MongoDB defines indexes on a per-collection level. You can create indexes on a single field or on multiple fields using a compound index or embedded attributes and documents. Indexes enhance query performance however each index also incurs some overhead for every write operation. Every document gets a default index on the _id attribute, which also enforces uniqueness.

3) Sharding capability: Sharding distributes a single logical database system across a cluster of machines. Sharding uses range-based portioning to distribute *documents* based on a specific *shard key*. With sharding MongoDB automatically distributes data among a collection of mongod instances.

   You should consider deploying a sharded cluster, if:

- Your data set approaches or exceeds the storage capacity of a single node in your system.
- The size of your system's active working set will soon exceed the capacity of the maximum amount of RAM for your system.
- Your system has a large amount of write activity, a single MongoDB instance cannot write data fast enough to meet demand, and all other approaches have not reduced contention.

4) Replication and Backup: Replication is the recommended way of ensuring data durability and failover in MongoDB. A new (i.e. bare and dataless) instance can be linked onto another at any time, doing an initial cloning of all data, fetching only updates after that. Initially both instances settle on which is master and which is slave, the slave taking over should the master go down. Also tools like mongodump/mongorestore and mongoexport/mongoimport lets the user to create a backup at any instance from a slave database. For more details n data backup and restore visit: http://docs.mongodb.org/manual/reference/components/#binary-import-and-export-tools

   Feature 3) and 4) of MongoDb make it a favorable option to be used in distributed rule based applications.

5) Support: MongoDb has an extensive, full-fledged API support to be used with most of the popular programming languages. Official developer communities and forums related to MongoDb allows users to get prompt support and help for most of the bugs, hot-fixes related to deployment and maintenance, or any other general queries related to schema design, applicability etc.

4.3 General installation guidelines:

1) Download the latest stable release of the mongodb server in .zip of .tgz format for the appropriate 32 bit or 64 bit operating system from http://www.mongodb.org/downloads/ , to your local drive and extract the contents of the package to a destination on your local drive having sufficient space.
MongoDB is self-contained and does not have any other system dependencies. You can run MongoDB from any folder you choose. You may install MongoDB in any directory

2) Setting up the environment: MongoDB requires a *data folder* to store its files (collections). The default location for the MongoDB data directory is C:\data\db. If you wish to change this directory to some other location, you can create a new directory called as "data" in the location where you want your MongoDb database collections to be stored. The data folder is usually the default folder where the Db dumps created using mongodump is stored. So it's a good practice to reserve some space in the drive where mongodb instance is installed along with the 'data' folder for normal operation.

3) Starting the MongoDb server: To start MongoDb server, execute the executable file mongod.exe, from the command prompt by reaching till the path \mongodb\bin\mongod.exe

   For e.g. If your MongoDb instance is stored in D:\ drive under a folder named MongoDb, you would execute the following command in cmd:
   D:\MongoDb\mongodb\bin\mongod.exe

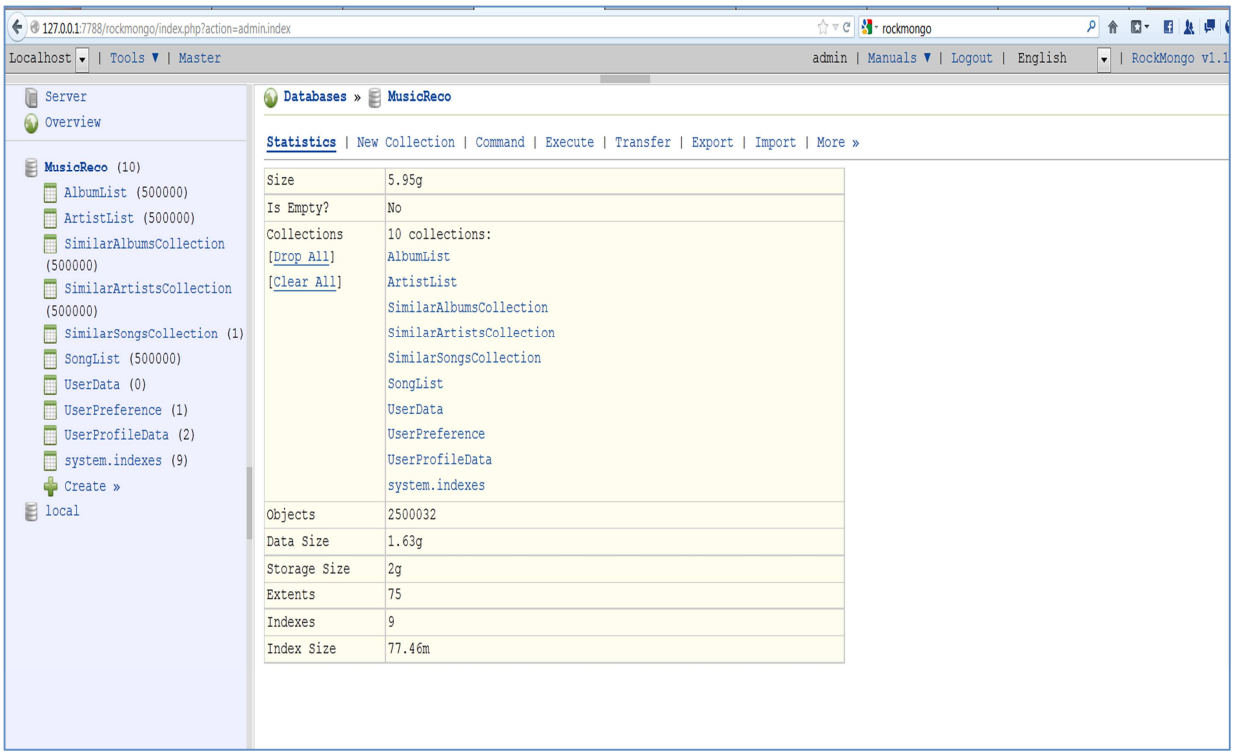
   You need not specify the --dbpath option if you have your "data" folder in the default directory else the path of this "data" folder needs to be given as a value to the --dbpath argument.
   For e.g. If the data folder is stored under D:\MongoDb\ ,then execute the command:

   D:\MongoDb\mongodb\bin\mongod.exe --dbpath D:\MongoDb\data

   You can also make MongoDb as a Windows service, the instructions of which can be found on the official link:
   http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/

   The installation guide for other operating systems can be found on the link:
   http://www.mongodb.org/display/DOCS/Quickstart

4) For accessing and updating the data stored in the MongoDb directly, we have used a PHP based MongoDb client : RockMongo. This web based client allows us to manage instances of MongoDb server, change administrator rights and create more users for a database. Also it allows efficient querying and updating documents using Javascript or JSON query.



The .zip extractable can be freely downloaded from http://rockmongo.com/ . Just extract the contents of the zip file into a local directory and run the executable rockstart.bat after your MongoDb instance is up and running. The default login name and password for the default instance of MongoDb is admin/admin.

4.4 Exporting and Importing data from MongoDb

- **mongodump** is a utility for creating a binary export of the contents of a database. Consider using this utility as part of an effective *backup strategy*. Used in conjunction with mongorestore to provide restore functionality.

  Usage:
  mongodump - -db <database name> --collection <collection name>

  This command should be executed from the command prompt by reaching till /mongodb/bin location. If the - -collection argument is not provided then all the collections which are stored in the "db" will be exported. The default path where this binary dump file will be created is mongodb/data/dump/ . For more usage options refer to:- http://docs.mongodb.org/manual/reference/mongodump/

- The **mongorestore** tool imports content from binary database dump, created by mongodump into a specific database. mongorestore can import content to an existing database or create a new one. mongorestore only performs inserts into the existing database, and does not perform updates or

upserts. If existing data with the same _id already exists on the target database, mongorestore will not replace it. mongorestore will recreate indexes from the dump.

Usage:

mongorestore --collection <collection name> --db <database name> <path>

The path argument refers to the location on the local drive where the binary dump file has been exported. For more usage options refer to :-
http://docs.mongodb.org/manual/reference/mongorestore/

- Other utilities such as “mongoexport” and “mongoimport” can also be used but it is not advisable to use these commands for full-scale backups as they may not reliably capture data type information. For exporting entire contents of a collection or a database it is recommended to use the binary import and export tools i.e. “mongodump” and ”mongorestore”.
  For more information regarding mongoexport and mongoimport and their usage refer to:-
  http://docs.mongodb.org/manual/reference/components/#data-import-and-export-tools

4.5 Some conclusions drawn after working with MongoDb for this project:

1) The concept of JOIN in MongoDB does not exist as such. Embedding of data in MongoDb is like pre-joined data. Linking of data between different collections (analogous to Tables in Relational Db) can be done by a follow up query (one query's result passed to another).

2) No restriction for a schema pattern in documents in a collection. This property can be used to create collections which can have dynamic number of attributes.

3) Mongo is great for scalability (no fixed schema design), so operations like alter table can be easily performed in Mongo by not disturbing the data in the collection.eg: Adding more attributes in “UserProfile” collection, say if some user belonging to a particular country has certain additional information that needs to be stored.

4) The schema design of MongoDb should in most cases is done after considering the use-cases of the application rather than build the application on top of a predefined schema of data in MongoDb.

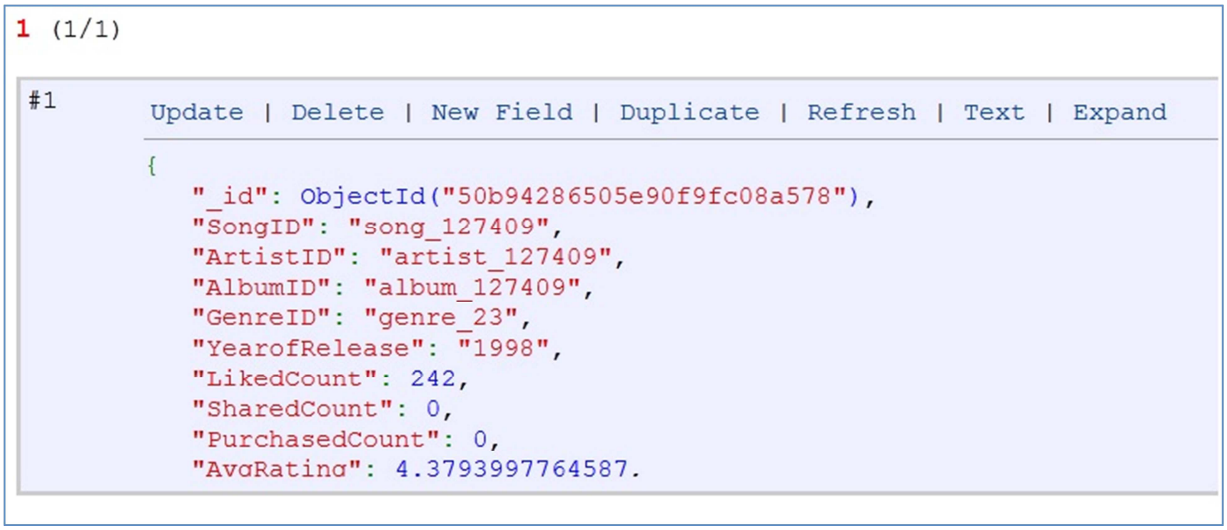
5) Performing two queries in Mongo is less efficient than a simple SQL JOIN from a relational database unless a large no. of table columns spanning across multiple tables are involved. So data should be distributed accordingly taking into consideration both performance as well as requirements of the application.

6) We used the Java driver provided by MongoDb for application programming interface, which is available at : http://www.mongodb.org/display/DOCS/Java+Language+Center#JavaLanguageCenter-Basics, as our application was primarily developed using the Java programming language. Also to parse the query result from JSON format into simple native types we made use of the “ javajson-20111122.jar” which is freely available at: http://sourceforge.net/projects/javajson/
along with the necessary documentation at : http://javajson.sourceforge.net/doc/api/ .

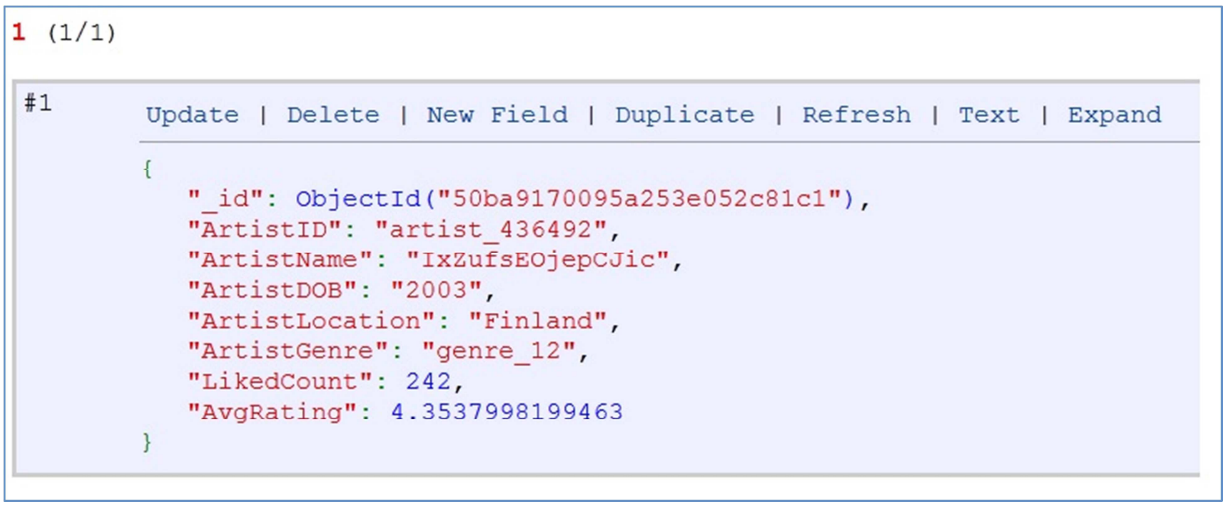
4.6 Collections implemented in MongoDb for use in our application

1) SongList: This collection consists of each Song stored in the relational database i.e MySQL ,as a document with the attribute "SongID" being the unique key for the entire document (the uniqueness of SongID is maintained by us manually in the program that populates this collection, as the default "_id" attribute is used by MongoDb internally to create a unique key constraint for each document). It also has certain additional attributes that maintains the collective information related to a particular song taken from other tables in MySQL. Some of these additional attributes apart from the ones that are already present in the song_data table are :
   - LikedCount: It is the sum total (or simply COUNT) of the number of times a particular song has been liked by Users of the system.
   - SharedCount: It is the sum total (or simply COUNT) of the number of times a particular song has been shared by Users of the system.
   - PurchasedCount: It is the sum total (or simply COUNT) of the number of times a particular song has been purchased by Users of the system.
   - AvgRating: It is the arithmetic mean (AVG) of all the ratings given by Users of the system to this particular song.

   All other attributes like SongHotness, SongDanceability, SongEnergy, SongTempo etc. are directly fetched from the song_data table in MySQL.



2) ArtistList: This collection stores all the Artists stored in MySQL database ,as a document with ArtistID being the key having the unique value. The ArtistID, ArtistName, ArtistDOB (date of birth), ArtistLocation, ArtistGenre attributes are directly fetched from artist_data in MySQL. The additional attributes stored for an Artist are:
   - LikedCount: It is the sum total (or simply COUNT) of the number of times a particular artist has been liked by Users of the system.
   - AvgRating: It is the arithmetic mean (AVG) of all the ratings given by Users of the system to a particular artist.



3) AlbumList: This collection stores all the Albums stored in the MySQL database, as a document with ArtistID being the key having the unique value. The AlbumID, AlbumName, AlbumArtistID, AlbumReleaseYear, AlbumCost, AlbumGenreID attributes are directly fetched from MySQL whereas the additional attributes are:

- LikedCount: It is the sum total (or simply COUNT) of the number of times a particular album has been liked by Users of the system.
- SharedCount: It is the sum total (or simply COUNT) of the number of times a particular album has been shared by Users of the system.
- PurchasedCount: It is the sum total (or simply COUNT) of the number of times a particular album has been purchased by Users of the system.
- AvgRating: It is the arithmetic mean (AVG) of all the ratings given by Users of the system to this particular album.

4) SimilarSongsCollection: This collection in MongoDb consists of SongIDs of 20 other songs in the system which are most similar to a given song. The SongID for which we are storing similar songs is the key having the unique value in the collection. The "SimilarSongs" is an embedded document consisting of SongID's of fifty other similar songs in the system.

#499999 Update | Delete | New Field | Duplicate | Refresh | Text | Expand

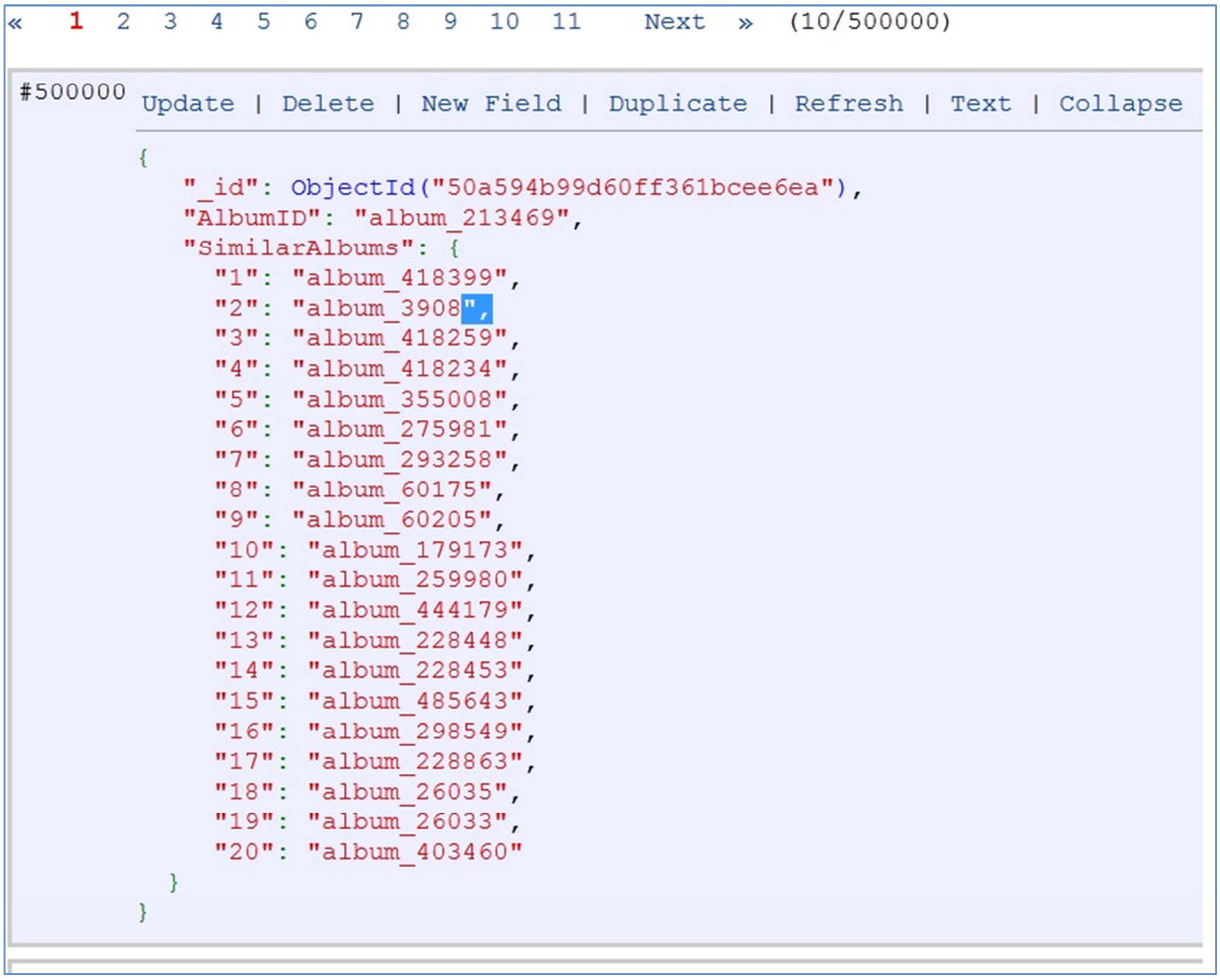
```
{
   "_id": ObjectId("50a6f8319949f3b32ddeb897"),
   "SongID": "song_125138",
   "SimilarSongs": {
     "1": "song_428997",
     "2": "song_162477",
     "3": "song_474876",
     "4": "song_185691",
     "5": "song_117819",
     "6": "song_280654",
```


5) SimilarArtistsCollection: This collection in MongoDb consists of ArtistIDs of 50 other artists in the system which are most similar to a given artist. The ArtistID for which we are storing similar artists is the key having the unique value in the collection. The "SimilarArtists" is an embedded document consisting of ArtistsID's of fifty other similar artists in the system.

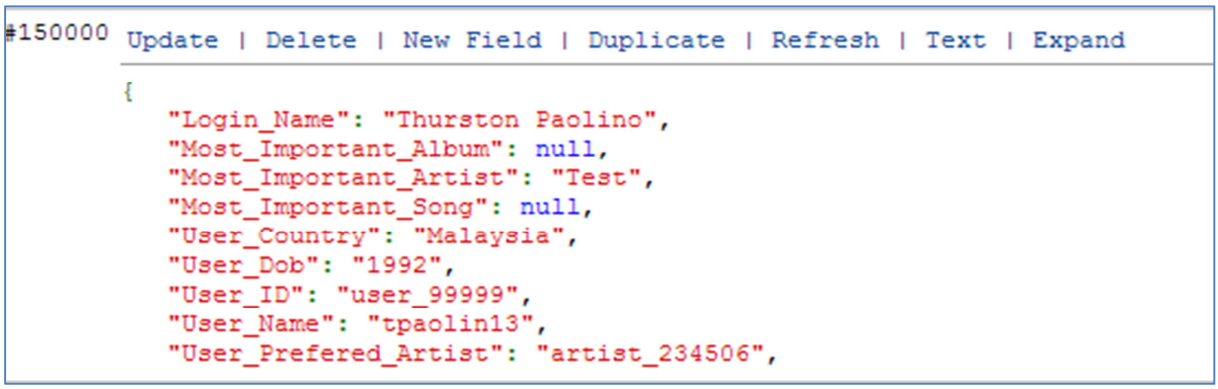
#499999 Update | Delete | New Field | Duplicate | Refresh | Text | Collapse

```
{
   "_id": ObjectId("50a1a6fe5ac7ed9a5f0a861d"),
   "ArtistID": "artist_417501",
   "SimilarArtists": {
     "1": "artist_101606",
     "2": "artist_108680",
     "3": "artist_117452",
     "4": "artist_16531",
     "5": "artist_221737",
     "6": "artist_339747",
     "7": "artist_340568",
     "8": "artist_341704",
     "9": "artist_34292",
     "10": "artist_344002",
     "11": "artist_229985",
     "12": "artist_355528",
     "13": "artist_355656",
     "14": "artist_355664",
     "15": "artist_231925",
     "16": "artist_359036",
     "17": "artist_363374",
     "18": "artist_24349",
     "19": "artist_244313",
     "20": "artist_385452",
     "21": "artist_385594",
```

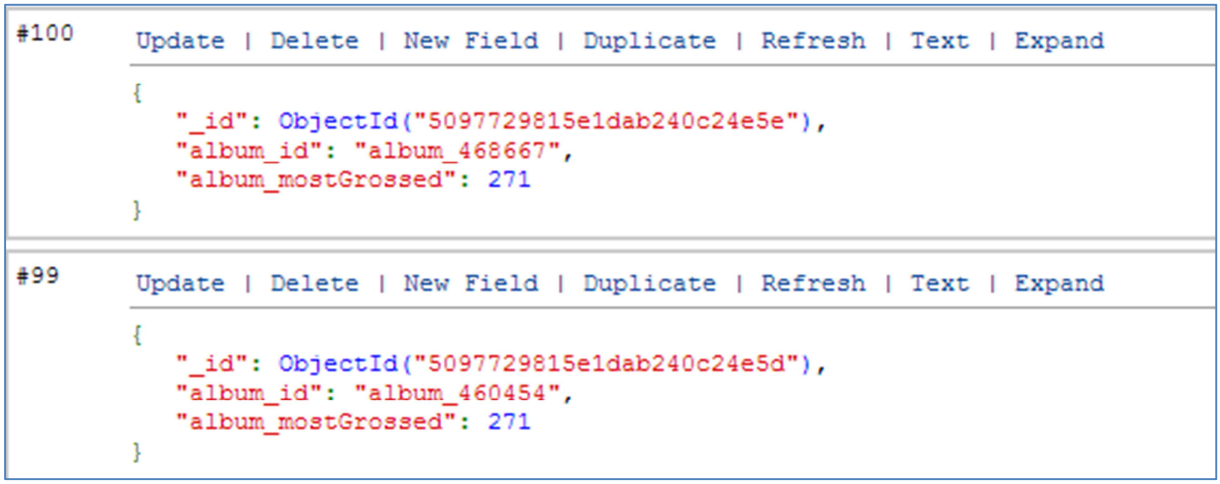
6) SimilarAlbumsCollection: This collection in MongoDb consists of AlbumIDs of 20 other albums in the system which are most similar to a given album. The AlbumID for which we are storing similar albums is the key having the unique value in the collection. The "SimilarAlbums" is an embedded document consisting of AlbumID's of fifty other similar albums in the system.



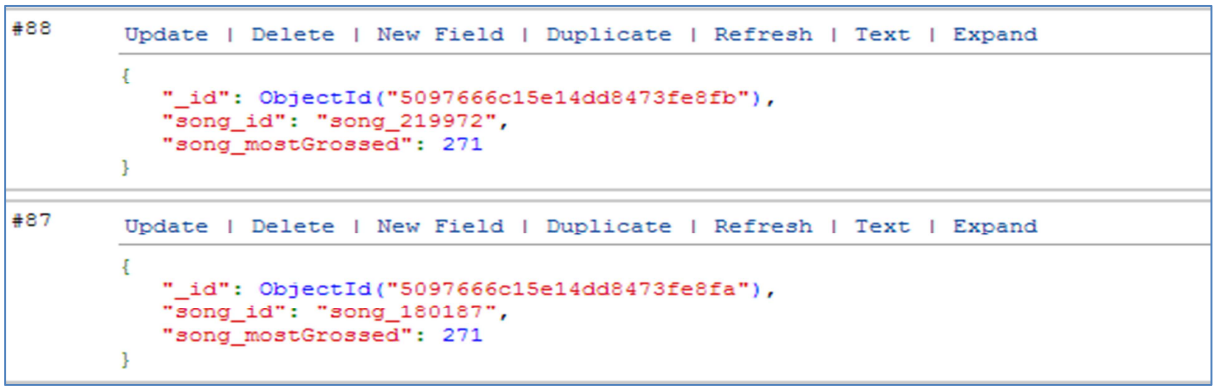
7) UserData: This collection stores user specific information as attribute keys in the MongoDb. The attributes which are fetched and stored directly from MySQL are Login_Name, User_ID, User_Name, User_Country, User_DOB, User_Preferred_Artist and User_Preferred_Genre. Other than these the Most_Important_Album, Most_Important_Artist and Most_Important_Song are stored for a particular User in this document. The values of these attributes can be updated as and when a user changes his preference of song/album/artist in the system.



8) GrossedAlbumsCollection: This collection stores most grossed or revenue generating albums. The revenue for each of the album from MySQl database is calculated and the details inserted into the GrossedAlbumCollection.



9) GrossedSongsCollection: This collection stores most grossed or revenue generating songs. The revenue for each of the album from MySQl database is calculated and the details inserted into the GrossedSongsCollection.



10) Popular Albums Collection: This collection contains the top 400 most popular songs.
The popularity value is calculated based on the users liked, users shared, users purchased.

#400 Update | Delete | New Field | Duplicate | Refresh | Text | Expand


```
{
   "_id": ObjectId("5097733b15e1dab240c24f8a"),
   "album_id": "album_75103",
   "album_popularity": 71.45
}
```


#399 Update | Delete | New Field | Duplicate | Refresh | Text | Expand


```
{
   "_id": ObjectId("5097733b15e1dab240c24f89"),
   "album_id": "album_443313",
   "album_popularity": 61.1
}
```

5. Rules

The business logic of our system is rule driven. The rules are invoked when an event occurs within the system. We have used Drools which is a part of the JBOSS Application Server to implement our Rule Engine.
As part of our application, we have implemented the following rules in our system

1. **Rule: "Songs Shared Update in NoSql"**
   The above rule belongs to the agenda group "SongsShared".
   Description: This rule will be invoked when a User will share a song. The rule will be explicitly called when its focus is set to true. It contains a SongBean class on which the rule will work on. It will then increment the number of shares of that particular song.

2. **Rule: "Songs Purchased Update in NoSql"**
   The above rule belongs to the agenda group "SongsPurchased".
   Description: This rule will be invoked when a User purchases a song. The rule will be explicitly called when its focus is set to true. It contains a SongBean class on which the rule will work on. It will then increment the number of purchases of that particular song

3. **Rule: "Album Liked Update in NoSql"**
   The above rule belongs to the agenda group "AlbumLiked".
   Description: This rule will be invoked when a User likes an album. The rule will be explicitly called when its focus is set to true. It contains an AlbumBean class on which the rule will work on. It will then increment the number of likes of that particular album

4. **Rule: "Album Shared Update in NoSql"**
   The above rule belongs to the agenda group "AlbumsShared".
   Description: This rule will be invoked when a User shares an album. The rule will be explicitly called when its focus is set to true. It contains an AlbumBean class on which the rule will work on. It will then increment the number of shares of that particular album

5. **Rule: "Album Purchased Update in NoSql"**
   The above rule belongs to the agenda group "AlbumsPurchased".
   Description: This rule will be invoked when a User purchases an album. The rule will be explicitly called when its focus is set to true. It contains an AlbumBean class on which the rule will work on. It will then increment the number of purchases of that particular album

6. **Rule: "Updating Artist Likes rule"**
   The above rule belongs to the agenda group "ArtistLiked".
   Description: This rule will be invoked when a User likes an album. The rule will be explicitly called when its focus is set to true. It contains an ArtistBean class on which the rule will work on. It will then increment the number of purchases of that particular album

7. **Rule: "Add Song Likes"**

   The above rule belongs to the agenda group "Likes".

   Description: This rule will be invoked when a User likes a song. The rule will be explicitly called when its focus is set to true. It contains SongData and UserData beans on which the rule will work on. It will then insert the User_ID and Song_Id in the Songs_Liked table.

8. **Rule: "Songs Shared Update"**

   The above rule belongs to the agenda group "SongsShared".

   Description: This rule will be invoked when a User will share a song. The rule will be explicitly called when its focus is set to true. It contains SongData and UserData beans on which the rule will work on. It will then insert the User_ID and Song_Id in the Songs_Shared table.

9. **Rule: "Album Shared Update"**

   The above rule belongs to the agenda group "AlbumsShared".

   Description: This rule will be invoked when a User will share an album. The rule will be explicitly called when its focus is set to true. It contains AlbumData and UserData beans on which the rule will work on. It will then insert the User_ID and Album_Id in the Albums_Shared table.

10. **Rule: "Album Purchased Update"**

    The above rule belongs to the agenda group "AlbumsPurchased".

    Description: This rule will be invoked when a User will purchase an album. The rule will be explicitly called when its focus is set to true. It contains a AlbumData and UserData bean on which the rule will work on. It will then insert the User_ID and Album_Id in the Albums_Purchased table.

11. **Rule: "Songs Purchased Update"**

    The above rule belongs to the agenda group "SongsPurchased".

    Description: This rule will be invoked when a User will purchase an album. The rule will be explicitly called when its focus is set to true. It contains SongData and UserData beans on which the rule will work on. It will then insert the User_ID and Song_Id in the Songs_Purchased table.

12. **Rule: "Artist Like Update"**

    The above rule belongs to the agenda group "ArtistLiked".

    Description: This rule will be invoked when a User likes an artist. The rule will be explicitly called when its focus is set to true. It contains ArtistData and UserData beans on which the rule will work on. It will then insert the User_ID and Artist_Id in the Artists_Liked table.

13. **Rule: "Album Like Update"**

    The above rule belongs to the agenda group "AlbumLiked".

    Description: This rule will be invoked when a User likes an artist. The rule will be explicitly called when its focus is set to true. It contains AlbumData and UserData beans on which the rule will work on. It will then insert the User_ID and Artist_Id in the Albums_Liked table.

14. **Rule: "Artist Rating Update"**

The above rule belongs to the agenda group "ArtistRating".

Description: This rule will be invoked when a User will rate an artist. The rule will be explicitly called when its focus is set to true. It contains ArtistData and UserData beans on which the rule will work on. It will then insert the User_ID and Artist_Id in the Artist_Rating table.

15. **Rule: "Album Rating Update"**

The above rule belongs to the agenda group "AlbumRating".

Description: This rule will be invoked when a User will rate an album. The rule will be explicitly called when its focus is set to true. It contains AlbumData and UserData beans on which the rule will work on. It will then insert the User_ID and Album_Id in the Album_Rating table.

16. **Rule: "Song Rating Update"**

The above rule belongs to the agenda group "SongRating".

Description: This rule will be invoked when a User will rate a song. The rule will be explicitly called when its focus is set to true. It contains SongData and UserData beans on which the rule will work on. It will then insert the User_ID and Song_Id in the Song_Rating table.

17. **Rule: "Getting user song liked"**

The above rule belongs to the agenda group "FetchSongsLiked"

Description: The rule will be invoked to get the list of songs liked by the user. The rule will be explicitly called when its focus is set to true. It returns the song_id of the song.

18. **Rule: "Getting user song shared"**

The above rule belongs to the agenda group "FetchSongsShared"

Description: The rule will be invoked to get the list of songs shared by the user. The rule will be explicitly called when its focus is set to true. It returns the song_id of the song.

19. **Rule: "Getting user song purchased"**

The above rule belongs to the agenda group "FetchSongsPurchased"

Description: The rule will be invoked to get the list of songs purchased by the user. The rule will be explicitly called when its focus is set to true. It returns the song_id of the song.

20. **Rule: "Getting user album liked"**

The above rule belongs to the agenda group "FetchAlbumsLiked"

Description: The rule will be invoked to get the list of albums liked by the user. The rule will be explicitly called when its focus is set to true. It returns the album_id of the album.

21. **Rule: "Getting user album shared"**

The above rule belongs to the agenda group "FetchAlbumsShared"

Description: The rule will be invoked to get the list of albums shared by the user. The rule will be explicitly called when its focus is set to true. It returns the album_id of the album.

22. Rule: "Getting user album purchased"

The above rule belongs to the agenda group "FetchAlbumsPurchased"

Description: The rule will be invoked to get the list of albums purchased by the user. The rule will be explicitly called when its focus is set to true. It returns the album_id of the album.

23. Rule: "Getting user artist liked"

The above rule belongs to the agenda group "FetchArtistsLiked"

Description: The rule will be invoked to get the list of artists liked by the user. The rule will be explicitly called when its focus is set to true. It returns the artist_id of the artist.

24. Rule: "Getting user songs genre liked"

The above rule belongs to the agenda group "FetchSongsGenreLiked"

Description: The rule will be invoked to get the count of the number of songs liked in the particular genre by the user.

25. Rule: "Getting user songs genre shared"

The above rule belongs to the agenda group "FetchSongsGenreShared"

Description: The rule will be invoked to get the count of the number of songs shared in the particular genre by the user.

26. Rule: "Getting user songs genre purchased"

The above rule belongs to the agenda group "FetchSongsGenrePurchased"

Description: The rule will be invoked to get the count of the number of songs purchased in the particular genre by the user.

27. Rule: "Getting user albums genre liked"

The above rule belongs to the agenda group "FetchAlbumsGenreLiked"

Description: The rule will be invoked to get the count of the number of albums liked in the particular genre by the user.

28. Rule: "Getting user albums genre shared"

The above rule belongs to the agenda group "FetchAlbumsGenreShared"

Description: The rule will be invoked to get the count of the number of albums shared in the particular genre by the user.

29. Rule: "Getting user albums genre purchased"

The above rule belongs to the agenda group "FetchAlbumsGenrePurchased"

Description: The rule will be invoked to get the count of the number of albums purchased in the particular genre by the user.

30. **Rule: "Similar artist rule"**

The above rule belongs to the agenda group "SimilarArtists"

Description: This rule will fetch the similar artist data from an artist. This data is computed based on content-based filtering.

31. **Rule: "Similar song rule"**

The above rule belongs to the agenda group "SimilarSongs"

Description: This rule will fetch the similar songs related to a song. This data is computed based on content-based filtering.

32. **Rule: "Similar artist rule"**

The above rule belongs to the agenda group "SimilarArtists"

Description: This rule will fetch the similar artist data from an artist. This data is computed based on content-based filtering.

33. **Rule: "Fetch Most Grossed Song"**

The above rule belongs to the agenda group "FetchMostGrossedSong"

Description: This rule will fetch the top 100 highest grossing songs

34. **Rule: "Fetch Most Grossed Album"**

The above rule belongs to the agenda group "FetchMostGrossedAlbum"

Description: This rule will fetch the top 100 highest grossing albums

35. **Rule: "Fetch Most Popular Album"**

The above rule belongs to the agenda group "FetchMostPopularAlbum"

Description: This rule will fetch the top 300 most popular albums

36. **Rule: "Fetch song data from MongoDB based on artist ID."**

The above rule belongs to the agenda group "SongFromArtist"

Description: This rule will fetch song data based on artist ID

37. **Rule: "Fetch album data from MongoDB based on artist ID."**

The above rule belongs to the agenda group "AlbumFromArtist"

Description: This rule will fetch the album data based on artist ID

38. **Rule: "Fetch artist data from MongoDB based on artist ID."**

The above rule belongs to the agenda group "ArtistDetails"

Description: This rule will fetch artist data based on artist ID

39. **Rule: "Fetch song data from MongoDB based on song ID."**
    The above rule belongs to the agenda group "SongData"
    Description: This rule will fetch song data based on song ID

40. **Rule: "Fetch album data from MongoDB based on song ID."**
    The above rule belongs to the agenda group "AlbumFromSong"
    Description: This rule will fetch album data based on song ID

41. **Rule: "Fetch artist data from MongoDB based on song ID."**
    The above rule belongs to the agenda group "ArtistFromSong"
    Description: This rule will fetch artist data based on song ID

42. **Rule: "Fetch song data from MongoDB based on genre ID."**
    The above rule belongs to the agenda group "SongFromGenre"
    Description: This rule will fetch song data based on genre ID

43. **Rule: "Fetch album data from MongoDB based on genre ID."**
    The above rule belongs to the agenda group "AlbumFromGenre"
    Description: This rule will fetch album data based on genre ID

44. **Rule: "Fetch artist data from MongoDB based on genre ID."**
    The above rule belongs to the agenda group "ArtistFromGenre"
    Description: This rule will fetch artist data based on genre ID

45. **Rule: "Fetch song data from MongoDB based on location."**
    The above rule belongs to the agenda group "SongFromArtist"
    Description: This rule will fetch artist details based on location and then invoke rules to fetch song data based on artist data.

46. **Rule: "Fetch album data from MongoDB based on location."**
    The above rule belongs to the agenda group "AlbumFromArtist"
    Description: This rule will fetch artist details based on location and then invoke rules to fetch album data based on artist data.

47. **Rule: "Fetch artist data from MongoDB based on location."**
    The above rule belongs to the agenda group "ArtistDetails"
    Description: This rule will fetch artist details based on location

48. **Rule: "Fetch song data from MongoDB based on user ID."**
    The above rule belongs to the agenda group "FetchMostPopularAlbum"

Description: This rule makes use of user's preferred genre to fetch song data

49. **Rule: "Fetch album data from MongoDB based on user ID."**

The above rule belongs to the agenda group "FetchMostPopularAlbum"

Description: This rule makes use of user's preferred genre to fetch album data

50. **Rule: "Fetch artist data from MongoDB based on user ID."**

The above rule belongs to the agenda group "FetchMostPopularAlbum"

Description: This rule makes use of user's preferred genre to fetch artist data

6. Junit Test Cases

A	testFetchSongDataForSong():	This rule was invoked from the test case . A song-id was provided and the rule was invoked. The results that are returned back are compared to assure that we get back the required result.
B	testFetchSongDataForArtist():	This rule was invoked from the test case . An artist-id was provided and the rule was invoked. The results that are returned back are compared to assure that we get back the required result.
C	testFetchSongDataForAlbum():	This rule was invoked from the test case . An album-id was provided and the rule was invoked. The results that are returned back are compared to assure that we get back the required result.
D	testFetchArtistData():	This rule was invoked from the test case . An artist-id was provided and the rule was invoked. The results that are returned back are compared to assure that we get back the required result.
E	testfetchArtistDataforSong():	This rule was invoked from the test case . A song-id was provided and the rule was invoked. The results that are returned back are compared to assure that we get back the required result.
F	testfetchArtistDataForGenre():	This rule was invoked from the test case . A genre-id was provided and the rule was invoked. The results that are returned back are compared to assure that we get back the required result.
G	testfetchAlbumDataForArtist():	This rule was invoked from the test case . An artist-id was provided and the rule was invoked. The results that are returned back are compared to assure that we get back the required result.
H	testfetchAlbumDataForAlbum():	This rule was invoked from the test case . An album-id was provided and the rule

		was invoked. The results that are returned back are compared to assure that we get back the required result.
I	testfetchAlbumDataforSong():	This rule was invoked from the test case . A song-id was provided and the rule was invoked. The results that are returned back are compared to assure that we get back the required result.
J	testfetchAlbumDataForGenre()	This rule was invoked from the test case . A song-id was provided and the rule was invoked. The results that are returned back are compared to assure that we get back the required result.

7. Traceability Matrix

	A	B	C	D	E	F	G	H	I	J
Recommendation songs, albums, artists	X	X	X					X	X	X
Profile Creation										
Rating a song, album, artist				X	X					
Liking a song,artist,album	X	X		X	X			X		
Sharing a Song/Album	X		X							X
Purchase a Song/Album	X		X	X				X		

Test Scripts that validate the Requirements

8. References

- http://en.wikipedia.org/wiki/NoSQL
- http://creately.com/blog/diagrams/uml-diagram-types-examples/
- A music recommendation system based on music data grouping and user interests,
  Hung-Chen Chen and Arbee L.P. Chen - Department of Computer Science
  National Tsing Hua University Hsinchu, Taiwan 300, R.O.C.
- An Efficient Hybrid Music Recommender System Using an Incrementally Trainable Probabilistic Generative Model
  Kazuyoshi Yoshii, Student Member, IEEE , Masataka Goto, Kazunori Komatani, Tetsuya Ogata , Member, IEEE and Hiroshi G. Okuno, Senior Member, IEEE