|  |
| --- |
| Coursera Android specialization |
| Capstone Project |
| Mutibo |
|  |
|  |
| **11/30/2014** |

|  |
| --- |
|  |

## Functional specification

### Introduction

This variant of the mutibo project aims at creating a general-purpose quiz game for android leveraging the architecture and technologies presented in the coursera android specialization path.

The quizzes follow the approach of sets consisting of 3 similar-1 different entries, and even though the final project presents only movie entries, the design is generic enough to accommodate different items with almost no modifications.

Thanks to the REST API offered by [themoviedb.org](http://www.themoviedb.org), 2404 movies and 1274 sets with various levels of difficulty have been automatically generated, only as a sample of what can be achieved. They are included on the server side of the submitted project as an H2 database. The generation mechanism is also included in the form of JUnit tests.

**Note**: for a detailed overview of how the Rubric requirements are implemented technically, please skip to the end of this document.

### Screencasts

The application screencasts are available as 4 videos on: <https://www.youtube.com/watch?v=sVU1UsI8Fcs&index=1&list=PL_5ZoQRzUDzReO9B-NKCKFTWXxmQMI3Mb>

The first 2 videos show the overall application screens and the game flow. The third demonstrates the gestures implemented, while the fourth shows the demo game. The videos are available at a resolution of 1080p; please wait a few moments and YouTube will adjust.

### Functional description

1. Users can be authenticated via username/password. Currently this authentication happens in the application and not any external service such as facebook (see further points about this).
2. If the Users choose not to authenticate they only have access to a demo game that only contains a few predefined sets (see YouTube video 4/4).
3. For authenticated users only, the initial possible actions will be 3 types of game: "*Solo*", "*Vendetta*", "Gang" as well as "*High scores*" and "Options". The actions should always be accessible in a navigation drawer. (<https://developer.android.com/design/patterns/navigation-drawer.html>)
4. "*Solo*" will start a game, where every question will pertain to a set. A set will be portrayed as four titles and (optionally in the final version) with images of the movie
5. The images of the movie are shown directly from [www.themoviedb.org](http://www.themoviedb.org), of which the REST API has been integrated with the app. This integration is used to:
   1. retrieve movie data
   2. generate questions (sets) based on similar or different movie attributes
   3. evaluate the difficulty of each set
   4. display the images of the movies. The images are not cached on the Mutibo server, but displayed directly from themoviedb.org.
6. After the user gives an answer (either correct or incorrect), he can view a text with information on the similarity of the movies.
7. Each correct answer gives one point to the accumulated score.
8. In each game, the player has the right to give two wrong answers. The third incorrect guess will mark the end of the game.
9. The total score upon the end of a game will possibly qualify for a place in the list of high scores of the corresponding screen, which is handled directly by integration with the *Amazon GameCircle* (<https://developer.amazon.com/public/apis/engage/gamecircle>).
10. After reading the explanation of each set (which movie is irrelevant), the user can either "like", or "dislike" the specific set. Sets that have at least 10 dislikes and not more than half as much likes (e.g. 30 dislikes and <=15 likes) will be automatically removed from upcoming games.
11. "*Vendetta*" is game against another mutibo user identified by their full name. The Vendetta game is a sudden death playoff, where the first player to make one mistake loses the game. The vendetta results are also stored on Amazon Gamecircle but not as a separate leaderboard (not for the needs of this capstone project)
12. "*Gang*" is a game in cooperation with another Mutibo user. The users could chat (using an external app) to agree on the correct answer. The first click on an answer by any of the two players is considered valid. The gang results are also stored on Amazon Gamecircle but not as a separate leaderboard (not for the needs of this capstone project)
13. Authorized users are able to skip a question, or ask a friend about a question. A user will be given such bonuses only after a number (e.g. 10) of consecutive correct answers.
14. The application provides the gestures that signal the immediate beginning of a new game from any screen of the app. The possible gestures are S for solo, V for Vendetta, and G for Gang. Any currently ongoing game will be abandoned with a proper warning. See Video 3/4 on YouTube for demonstration of this feature.
15. The questions vary by difficulty levels. The levels are:
    1. *Very* *Easy* (e.g. 3 movies of 2014 and 1 of 1960, easy to find the different one)
    2. *Easy (e.g. 3 movies of 2014 and one of 1990)*
    3. *Normal (e.g. 3 movies of 2014 and one of 2006)*
    4. *Hard (e.g. 3 movies of 2014 and one of 2011)*
    5. *Extreme (e.g. 3 movies of 2014 and one of 2013)*

The first questions are always very easy and the difficulty gradually increases as the player answers correctly. See the requirements mapping to code later on for details how this is implemented.

1. Authentication via facebook also *has been implemented* for this project but it was not included in the final submission because the list of facebook friends could not be retrieved, it was always empty (probably some configuration issue). But only logging in with facebook is meaningless unless the player is able to play Vendetta or Gang with a friend.

### Game types

As described above, the game offers three types. For a detailed presentation please see the Videos 1/4 and 2/4 :

1. **Solo**: Single player game. An authenticated user can start it any time by using the left menu or the S gesture. It consists of quads of movie images and she only has to click on the image that she thinks is different (which then is automatically submitted). The maximum wrong answers (lives) are 3.
2. **Vendetta**: Competition game. To begin a vendetta game the user must click on the left menu “Vendetta” option or use the V gesture. Then a list of the app users comes up, and by clicking on one name, the request is launched immediately for a Vendetta game. This requesting user then remains in waiting mode until the other responds. Now the target user receives a notification as soon as he/she logs in to the app. By clicking on this notification the game starts and both opponents receive the same questions. The game ends when one of them gives the first incorrect answer.
3. **Gang**: Cooperation game. The procedure to begin such a game is exactly the same with Vendetta (though using the “Gang” menu option or G gesture). As soon as the game starts, the co-players are answering the same questions with a maximum of 3 lives. Here the first to click on an answer is considered to give the answer of the team (the gang of two players).

### User Interface

The functionality of the application is partially demonstrated in the following screenshots. All screenshots were taken from a running instance of the app on a Sony Xperia Z1 compact, while the videos on YouTube from a Nexus 7 tablet, thus verifying that the app layout can appear without problems on different screen sizes and resolutions.

#### D:\Work\Dropbox\Camera Uploads\2014-11-30 20.26.29.png

#### Home page

The home page asks the user to login via an individual mutibo user name and password. There is an additional button to play a demo game without signing in. A full version of the demo can be seen on the YouTube video 4/4.

### D:\Work\Dropbox\Camera Uploads\2014-11-30 20.26.59.png

The first time a user opens the application, he/she can create a mutibo username and password by pressing sign up. This leads to the screen on the right where the only needed information is Full name, user name and password.

### D:\Work\Dropbox\Camera Uploads\2014-11-30 20.38.35.png

### Main menu

The navigation drawer screen menu can open by tapping the mutibo icon on the top left or scrolling from the left edge of the screen.

It contains the screen possibilities:

* Solo (new solo game),
* Vendetta (new Vendetta game),
* Gang (new Gang game),
* Hot scores (leaderboard screen),
* Options (settings of the app).

None of the 3 *game types* can be started without logging in.

#### D:\Work\Dropbox\Camera Uploads\2014-11-30 20.39.10.pngGame screen

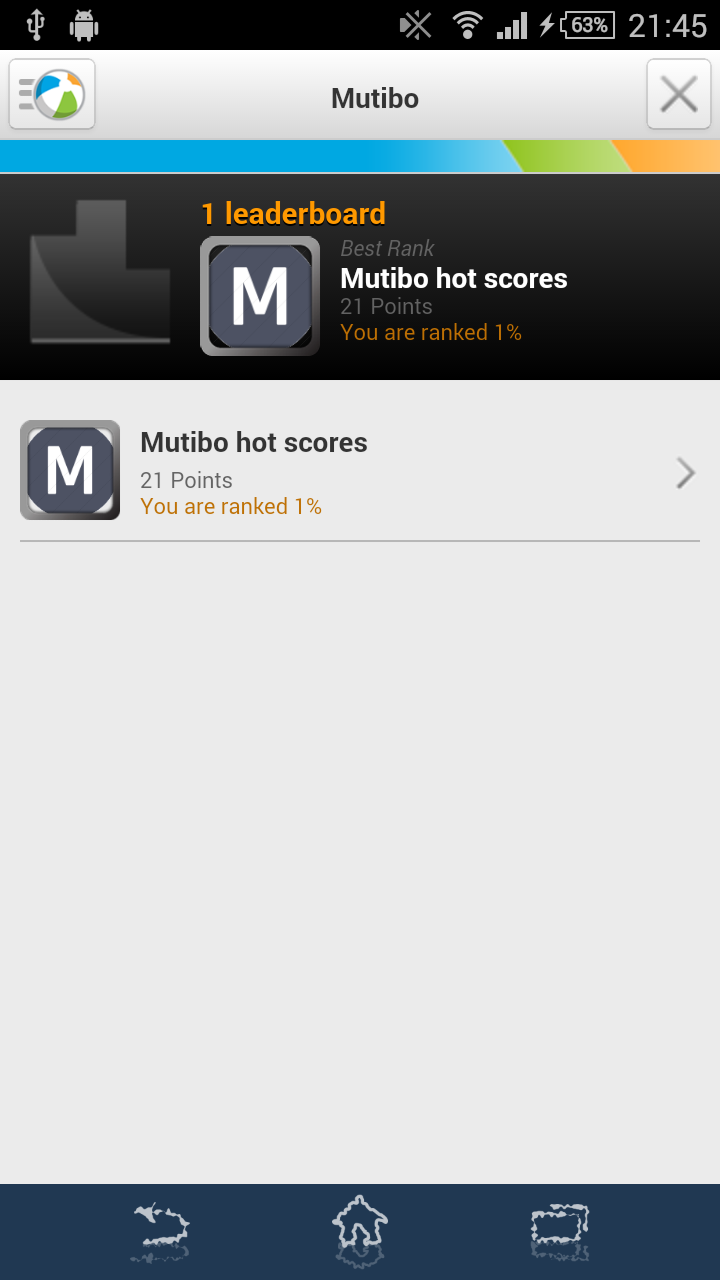
The game screen is approximately the same in all game modes. In a new solo game each set is represented on screen with 4 movie titles accompanied by their respective images. The user has to tap on one title or image to answer.

For advanced users there will be a possibility to ask a friend (another mutibo user) or skip a question, under the conditions described earlier.

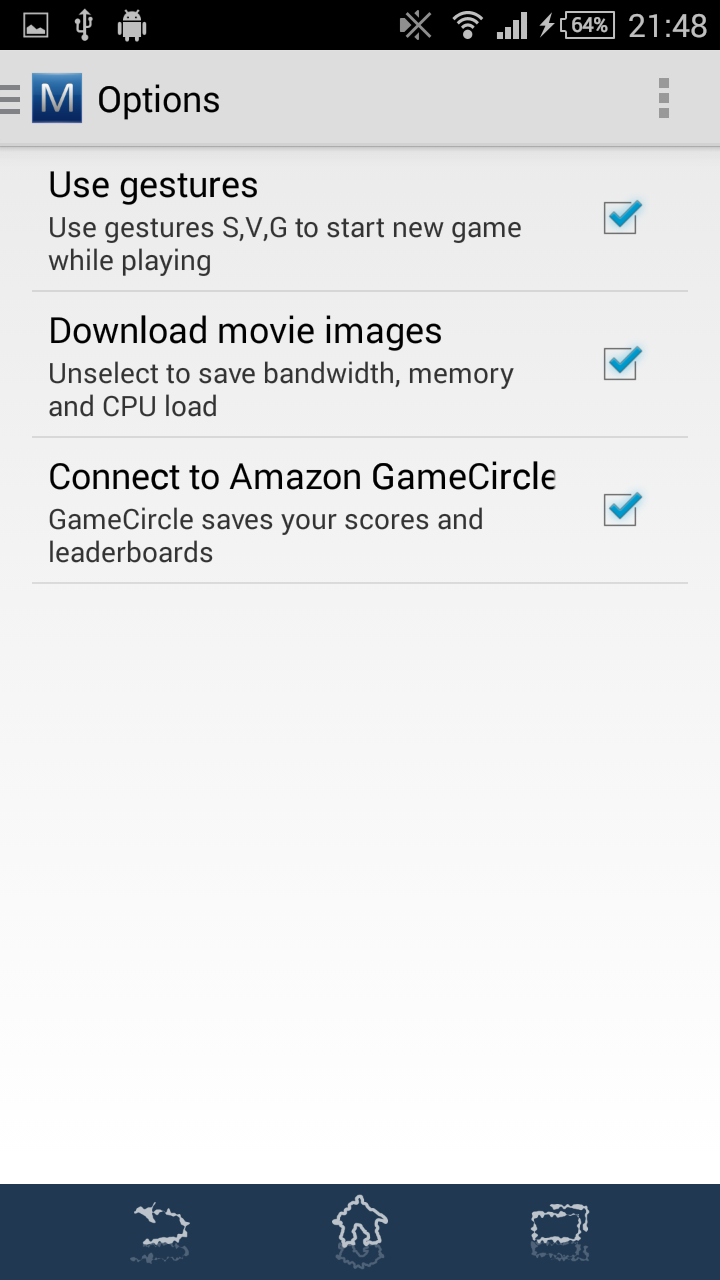
### D:\Work\Dropbox\Camera Uploads\2014-11-30 20.39.16.png

After giving an answer, either successful or unsuccessful, the user can see details about the correct answer and an explanation. Here there are the following options:

* Tapping on the movie image will lead to the next set of movies.
* Tapping on the like or dislike buttons will respectively give a like or dislike to this set. Under certain circumstances explained in the functional specification before, this set may disappear from subsequent games.

**Hot Scores screen**

As a more attractively named screen than “high scores”, it provides a list of the user’s achievements from Amazon GameCircle.



**Options screen**

The settings screen provides some basic app parameters, as seen on the screenshot.Changing an option such as Amazon GameCircle connection may require a restart of the application.

### Technical description

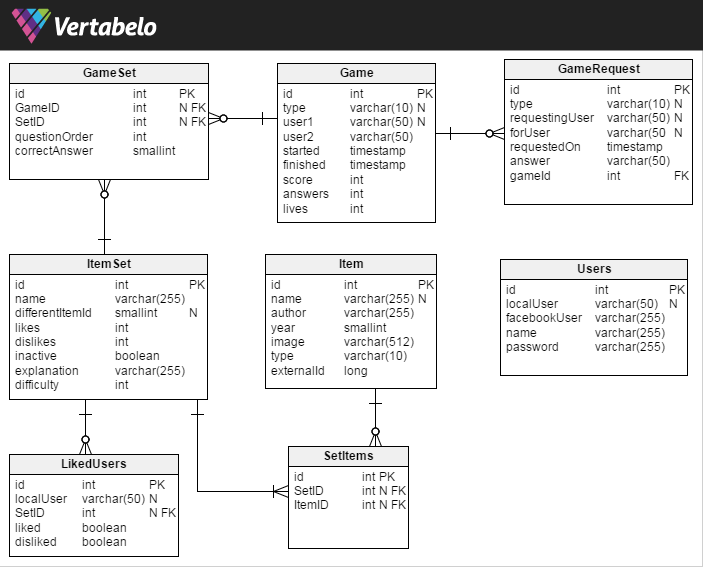
Below are some of the basic decisions on the technical aspects of the project.

1. The project consists of a server Spring based application (***Mutibo-server*** and ***Mutibo-api***) that exposes REST operations via HTTPS and an Android client application that the user can use to play the game on her phone. The spring server uses Spring Boot and Data Repositories as shown on the mobile cloud MOOC.
2. The client application (***Mutibo-android***) mainly consists of
   1. A basic android activity that exposes the user interface in the form of fragments
   2. A bound android service that will asynchronously exchange information with the server about pending game requests (for Vendetta, Gang).
3. The Client leverages a Retrofit based REST client as shown in the mobile cloud MOOC. The client uses HTTPS to communicate with the server.
4. The server uses Spring Security and Spring OAuth 2.0 to support multiple users via a custom UserDetailsService (see class *CustomUserDetailsService.java*). Users will authenticate via the Spring OAuth 2.0 token endpoint and provide bearer tokens in an authorization header to prove user identity. New users are registered and stored in the database, from where the custom UserDetailsService looks them up.
5. The server contains one controller that serves the REST HTTP requests as demonstrated in the mobilecloud MOOC, ***MutiboSvc.java***
6. The database access will be performed using JPA entities and Repositories as described and demonstrated in the mobilecloud MOOC. See Mutibo-api project, the classes of package ***org.ntk.mutibo.repository***
7. The server is stateless when it comes to the active user’s game. The “game id” has to be provided by the client via REST, but the game data (already answered questions etc.) will be kept on the server side in order to:
   1. Avoid repeated questions
   2. Aid in eventually increasing the game difficulty
   3. Provide game history in the future, especially for the collaboration modes (vendetta, gang)
8. The user preferences shown on the corresponding UI screen are stored only on the client mobile app using the built-in android preferences functionality. The preferences UI will use the Android *PreferenceFragment* to display the available options.
9. No storage of movies/sets will be performed on the client. The client will have to request every time something is needed (or for that matter also in advance). The request will be always against the REST API described in a next section.
10. The request will be performed in a thread retrieved from a Thread pool to avoid blocking user operation in the UI thread. For a detailed overview of all off-UI thread game operations, a good starting point is ***DefaultGameManager.java***
11. Along with the Mutibo source code there is also the GameCircleSDK project. This is present for the peer assessor so as not to have to download and locate the needed projects. The Mutibo-android project references it on the same level as its own, as it’s included in the submitted zip file. In case any problem occurs, the library can be referenced again from the Eclipse project properties (right click on the Mutibo-android project🡪then Properties), then *Android* category, where we can reference an Android library.

#### Database

The set and game data is maintained in a relational database. This database is an HSQLDB or H2 persisted on disk. Both options are given in the source and we can switch by enabling the @Configuration of either class *HSqlJpaConfig* or *H2JpaConfig* . The selected on on the submitted version of the project is H2. As the system scope is not to provide a production ready solution, no more attention will be given to the question what DB to use, as well as NoSQL options will not be considered for the purposes of the capstone project.

The area of focus is rather the DB schema, which has to be designed according to the project requirements. The proposed schema can be seen in the following diagram. The basic entities are:

**** - **Item**: an item of a question. In this case a movie. It contains basic information that can correspond in our case to a movie, like the author (director), year, the path to an external image file on the server, as well as creation and update timestamps.

- **Set**: a set of Items that can be used as a question. The columns contain a name (optional), an index of the different question that can receive values 0..3, the number of users that liked and disliked this set, if it's active (by default true) as well as timestamps.

- **SetItems**: realizes the many-to-many (M..N) relationship between Sets and Items.

- **Game**: every game started by the user is stored here. The type is one of the 3 types offered. Two user names, as currently there is a hard limitation of up to two users playing a game simultaneously. Also we keep the time it was started and the current (or final if finished) score. It's not yet clear if we'll need to store the status of the game (finished, ongoing etc.) and end timestamp. They will be added if needed.

- **GameSet**: realizes the many-to-many (M..N) relationship between Sets and Games. Stores additionally the order in which a specific set appeared in a game as a question, and whether the given answer was correct. The latter field is of integer type to anticipate future versions of the game where more than one attempts to answer a question might be permitted.

- **Users**: Storage of user information, particularly the mapping of a mutibo user (localUser) to her corresponding facebook and google accounts. This structure has to be reconsidered according to the interface with these systems.

- **LikedUsers**: list of the users who liked or disliked a set. It serves as a history for like/dislike display purposes to the user herself.

- **GameRequest**: a request for a cooperation or competition game. It’s requested by *requestingUser,* directed to *forUser*.

#### REST interface

Below is a summary of the most important REST operations that facilitate the communication between the Spring based HTTP server and the android application. A full list can be retrieved from the actual REST API in the source code: *MutiboApi.java.*

|  |  |  |
| --- | --- | --- |
| **Operation** | **HTTP Method** | **Description** |
| /oauth/token | POST | The access point for the OAuth 2.0 Password Grant flow. |
| /item | POST | Adds a new item (movie) to the database |
| /item/{id} | GET | Returns the information of the specific item in JSON format |
| /itemset/{id} | GET | Returns the information of the specific item set in JSON format |
| /itemset/{id}/like | POST | Sets an item as liked by the user that made the request (liking an item unsets any previous unlike of the same user for the same item) |
| /itemset/{id}/unlike | POST | Sets an item as unliked by the user that made the request (unliking an item unsets any previous like of the same user for the same item) |
| /itemset/{id}/like | GET | Returns whether the current user has liked or unliked the item indicated by {id}. Returns a composite object with both flags like/unlike. |
| /game/demo | GET | Returns the list of sets that comprise the demo game offered. |
| /game/start/{type}?user1=  {user1}& user2=  {user2} | POST | Starts a new game of the given type for the (maximum of) two users. Returns the game ID to be used by the client for subsequent calls. |
| /game/{id} | GET | Returns the game {id}. |
| /game/{id}/next | GET | Returns the next item for the current user and the game {id}. |
| /game/{id}/finish | POST | Ends the game {id}. |
| /game/{id}/answer?setid=  {setid}&answer={answer} | POST | Sets the user answer for a particular set of game {id}. |
| /game/{id}/finish | POST | Finishes the game {id}. |
| /gamerequest/{id}/{user} /request?forUser={forUser} &type={type} | POST | Request a game of type {type} as {user} by the {forUser} |
| /gamerequest/{id}/{user} /join | POST | The {user} joins a game created for the request {id} |
| /gamerequest/{user}  /pending | GET | Returns the pending game requests for the given user (as *forUser,* not the requestor) |

**Remarks**

* In all operations the root path “/mutibo” is implied.
* The operations with return objects return an HTTP response with error code 404 if the game is not found. This is true mostly for the GET requests.

### Requirements mapping

The following table summarizes the project requirements and how they’re addressed in the functional and technical specifications of this document.

|  |  |
| --- | --- |
| **Requirement** | **Technical Coverage** |
| **Basic 1**  App supports multiple users via individual user accounts | - Server side: ***DB table users*** (represented by entity class **MutiboUser.java**), user details service: ***CustomUserDetailsService.java***, that is injected in the OAuth implementation: ***OAuth2SecurityConfiguration.java***. As seen on the videos, the users can react with each other as well as have a personalized view of the game. |
| **Basic 2**  App contains at least one user facing function available only to authenticated users | All the functions described in this document are available only to authenticated users, besides the Demo game and of course user registration. More details in the Functional description. |
| **Basic 3**  App comprises at least 1 instance of each of at least 2 of the following 4 fundamental Android components:  Activity  BroadcastReceiver  Service  ContentProvider | See project ***Mutibo-android***.  ***Activity***: MainActivity.java, which is defined as singleTask (only one instance)  ***Service***: GameRequestService.java  For the needs of Amazon GameCircle integration, in the manifest there is also a ***Broadcast receiver*** of class *com.amazon.identity.auth.device.authorization.PackageIntentReceiver*  AndroidManifest.xml defines all of these elements |
| **Basic 4**  App interacts with at least one remotely-hosted Java Spring-based service | The client makes requests from   * **DefaultGameManager.java** * **LoginFragment.java** * **GameRequestService.java**   Using CallableTask instances (as per example of mobile cloud MOOC). These use a Retrofit REST client to access the server. A good approach would be also to use a SyncAdapter since there is already a bound service for this project.  The server provides the ***MutiboSvc.java*** controller that implements the REST operations defined in ***MutiboApi.java*** |
| **Basic 5**  App interacts over the network via HTTP | Similar to the previous item, the app interacts via ***HTTPS*** as described in the *Technical Description* section |
| **Basic 6**  App allows users to navigate between 3 or more user interface screens at runtime | See the screencast videos on YouTube and the screenshots. The available screens are:   * ***Login*** * ***Registration*** (sign up) * ***Game*** ***question*** * ***Answer*** ***Explanation***, results * ***Hot*** ***scores*** * ***Options*** |
| **Basic 7**  App uses at least one advanced capability or API from the following list (covered in the MoCCA Specialization): multimedia capture, multimedia playback, touch gestures, sensors, animation | The app uses gestures as seen in ***MainActivity.java*** for the initialization (method ***initializeGestures*** ) and event handling of gestures (handler **onGesturePerformed**). The gesture overlay used throughout the game is declared in the layout file ***activity\_main.xml*** and the gestures themselves are located in the file ***Mutibo-android/raw/gestures***  The app also uses the android graphics package to manipulate, translate, rotate and scale the score banner. This is located in Mutibo-android: ***BannerDrawable.java*** and used by QuestionFragment’s method ***placeOverlay()*** |
| **Basic 8**  App supports at least one operation that is performed off the UI Thread in one or more background Threads of Thread pool. | See all server communication operations in ***DefaultGameManager.java, GameRequestService,java, LoginFragment.java***, as well as the movie image loading in ***QuestionFragment.java:***   * Method ***placeOverlay*** manipulates the score diagonal banner * Method ***displayImage*** uses an asynchronous loader from the very useful library “*Android universal image loader*” that can be found on [*https://github.com/nostra13/Android-Universal-Image-Loader*](https://github.com/nostra13/Android-Universal-Image-Loader) |
| **Functional 1**  A Set is a unit of data that contains four movie titles, optional associated images for each movie, information identifying the one movie that is not like the other three, and accompanying text, explaining the relationship between the three related movies. | Please Introduction, Func. Description and DB schema. In the code you can find the JPA entity ***ItemSet.java*** under project Mutibo-api that contains these attributes and a list of the 4 movies. The image paths are stored with each Movie, in the ***Item.java*** JPA Entity.  **Warning**: please note that there are ItemSet and Item JPA entities in the Mutibo-api project, while there are also ItemSet and Item as JSon objects on the android client side. This yet remains to be improved; no duplicate information should ever be present. The role of the Mutibo-api was to provide such common ground between server and android, be included by both but such an alignment is not yet complete due to the gradle dependencies of the Mutibo-api project (like JPA jars) that should NOT be included on android. |
| **Functional 2**  A User should be able to log into the game using an authenticated user account. | Func. description 1 & UI home page, UI Registration page.  Also see Technical Description, item 4 about the custom ***CustomUserDetailsService.java*** |
| **Functional 3**  A single game presents a series of Sets and guesses, until the User has made three Incorrect Guesses. | Clarified on the introduction, Func. Description, UI screenshots and videos. The entity Game.java contains the member attribute “***lives***” for the remaining incorrect guesses, as well as the constant ***MAX\_LIVES***.  These are checked by the method ***Game.isGameOver()*** that is called directly after giving an explanation for an (in)correct answer in ***ExplanationFragment.java***. |
| **Functional 4a**  After viewing a Set, a User will be able to rate a Set based on the explanation of the link between the movies. | As seen on UI game screen for the explanation, the like/dislike buttons map to the GameManager methods likeItemSet, dislikeItemSet, that in turn call the server side REST operations with the same names. Their implementation can be seen in MutiboSvc.java, in the methods also named ***likeItemSet, dislikeItemSet***. |
| **Functional 4b**  If a Set receives a large number of poor ratings, it can be removed from the game. | Following the Functional Description, an ItemSet is characterized inactive in the methods of MutiboSvc.java: ***likeItemSet*** and ***dislikeItemSet***, which respectively call the ItemSetRepository.java methods activateLikedItemSets() and in activateDislikedItemSets(), that change the attribute “***inactive***” of the entity ***ItemSet***.  Now when a new Set is picked as next question in a game, the inactive ones are ignored from the query that picks them, that is in ItemSetRepository: method ***findByInactiveFalseAndDifficulty*** that uses the @Query ***FIND\_ID\_DISTINCT\_QUERY*** |
| **Functional 5**  For each successfully completed Set, the user will get *Points*. | Following the Functional Description, the user gets 1 point per correct answer. This is implemented in the ***Game.java*** JPA entity, method ***addCorrectAnswerToScore****()* |
| **Functional 6**  All data (questions, answers, points, etc.) are stored to and retrieved from a web-based service accessible in the cloud. | As described in Technical Description 3,9 & REST API, there is no data besides the preferences stored on the client side. All the data comes from the cloud and the following are only buffered:   * The next question(s) of the current game is buffered in ***DefaultGameManager.java***, field ***buffer*** (of type ArrayDeque<ItemSet>) * The images loaded for the movies may be cached by the current configuration of the ImageLoader component |
| **Bonus 1**  Users could be allowed to challenge a friend to a sudden death playoff. For example, friends could answer questions turn by turn, and the first person to make a mistake loses. | See Vendetta game mode in the videos and *Game Types* section. This game mode is implemented in several parts of the Android project, the basic ones being:   * ***PickUserFragment.java***: to make a request against a friend * ***DefaultGameManager***.***java***: to perform the game request and store it on the server side. * ***GameRequestService.java:*** The bound service of the app has exactly the role to retrieve the pending Vendetta, Gang requests and inform the main activity. * ***ReplyHandler.***java: receives the reply from the service and generates a Notification for the user. * ***MainActivity.java***: in the method ***onNewIntent***() we receive the notification click event. There is always one instance of the activity (as defined in the AndroidManifest.xml), so it always comes here and we then call the gameManager that the current user has joined the game. * ***DefaultGameManager.java:*** informs now the server sidethat the target user (forUser) joined and retrieves the game that’s already created for this challenge. * ***MainActivity.java***: ***onGameJoined***() listener method now handles the arrangement of the UI for the game. |
| **Bonus 2**  Users could be given progressively difficult questions (e.g., based on other users' previous success with each Set). | Following Functional Description 15, the following elements implement this logic:   * ***ItemSetDifficulty.java*** is an enumeration that defines the possible difficulty levels. * ***ItemSet.java***: attribute ***difficulty*** defines the difficulty of the particular question. * ***MutiboApi.java*** defines the operation ***getNextItemSet*** that returns the next question for the game. This requires a difficulty level that is given by the client according to its rules.   Now the most important point here is how the difficulty is assigned to the questions. This is located in the src/test/java classes of Mutibo-server project. In particular there is a mechanism there to generate movies and questions based on the data retrieved from themoviedb.org. It’s implemented in ***TheMovieDbApiTest.java*** and makes use of an abstract mechanism to implement such test data generators in the package ***org.ntk.mutibo.test.generator.*** The interface ***DifficultyEvaluator***.***java*** there defines a method to evaluate the difficulty of the newly generated set and is implemented by the two generators submitted with the project:   * ***ReleaseYearDifficultyEvaluator.java*** * ***CommonActorsDifficultyEvaluator.java*** |
| **Bonus 3**  Users could be given special "power ups" such as the ability to pass a Set, or to get help from a friend them when they are stuck. | There are the buttons to skip a question and ask a friend in the layout file ***fragment\_question.xml***, that are used in the corresponding fragment implementation, ***QuestionFragment.java***. These functions however are more likely to be present as animated moving banners in a real world app (like the score banner but animated) that appear only after 10 correct answers instead of buttons |
| **Bonus 4**  Users could be allowed to challenge Facebook friends to do various things such as to beat their high score, to help answer a question they are stuck on, etc. | Following the functional description, these are the Vendetta and Gang game type, described above in detail. The reason that the available friends are not facebook friends is like stated above, that due to some configuration issue on mutibo app configuration page (<https://developers.facebook.com/apps/597151533747401/dashboard/>), the friends list is empty, even though I’ve generated several test users, logged in with them and let them use the app. However facebook integration has been prototyped successfully for the needs of this project but not submitted. So the user can log in or log out from his/her facebook account, but do nothing more. |
| **Bonus 5**  The app could be optimized for the Amazon Appstore and could leverage Amazon’s GameCircle API to incorporate leaderboards. | The GameCircle leaderboard integration is Implemented in ***GameCircleHelper.java*** and used by ***MainActivity.java*** in methods:   * ***displayView*** * ***initGameCircle*** * ***onGameFinished*** * ***onPause*** * ***onStop***   Once logging in to the app for the first time, an icon appears on the bottom that this app can be used with Amazon appstore and gaming services.  By clicking it, the user logs in to his/her amazon account, in order to gain access to the leaderboards. Every subsequent time he/she receives a message “Welcome [user name]” from Amazon services, and does not need to log in again. This process can be seen on the video 2/4. |