Developers of the Millenium

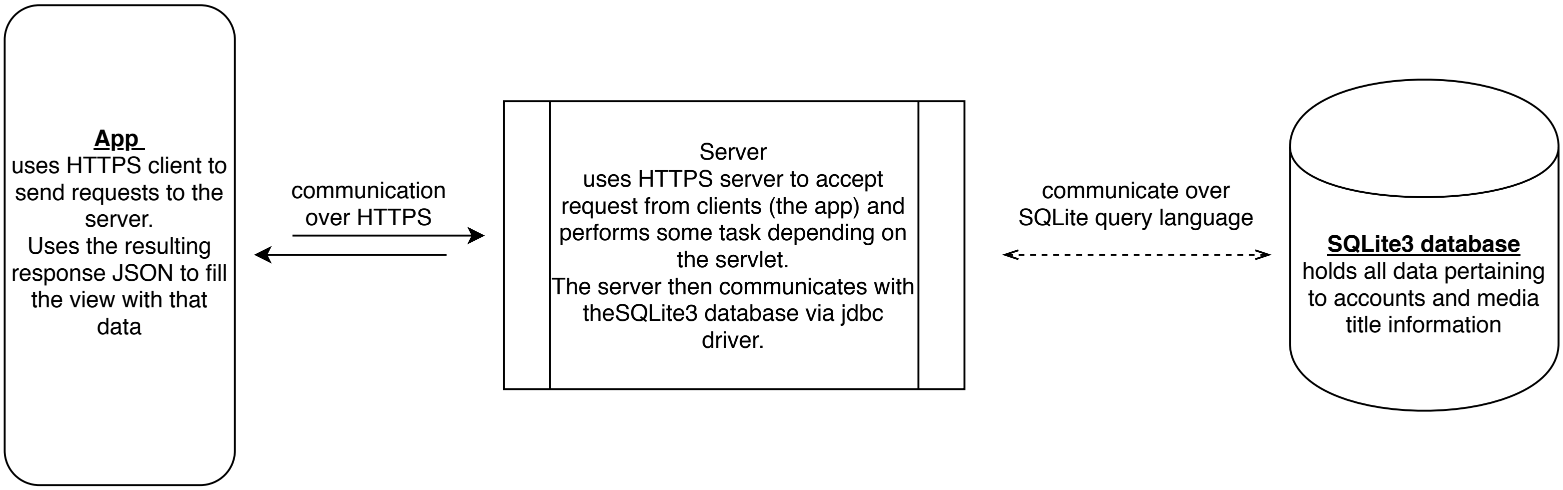
System Design

Table of Contents

Software Architecture Diagram

CRC Cards

Basics



App
uses HTTPS client to send requests to the server.
Uses the resulting response JSON to fill the view with that data

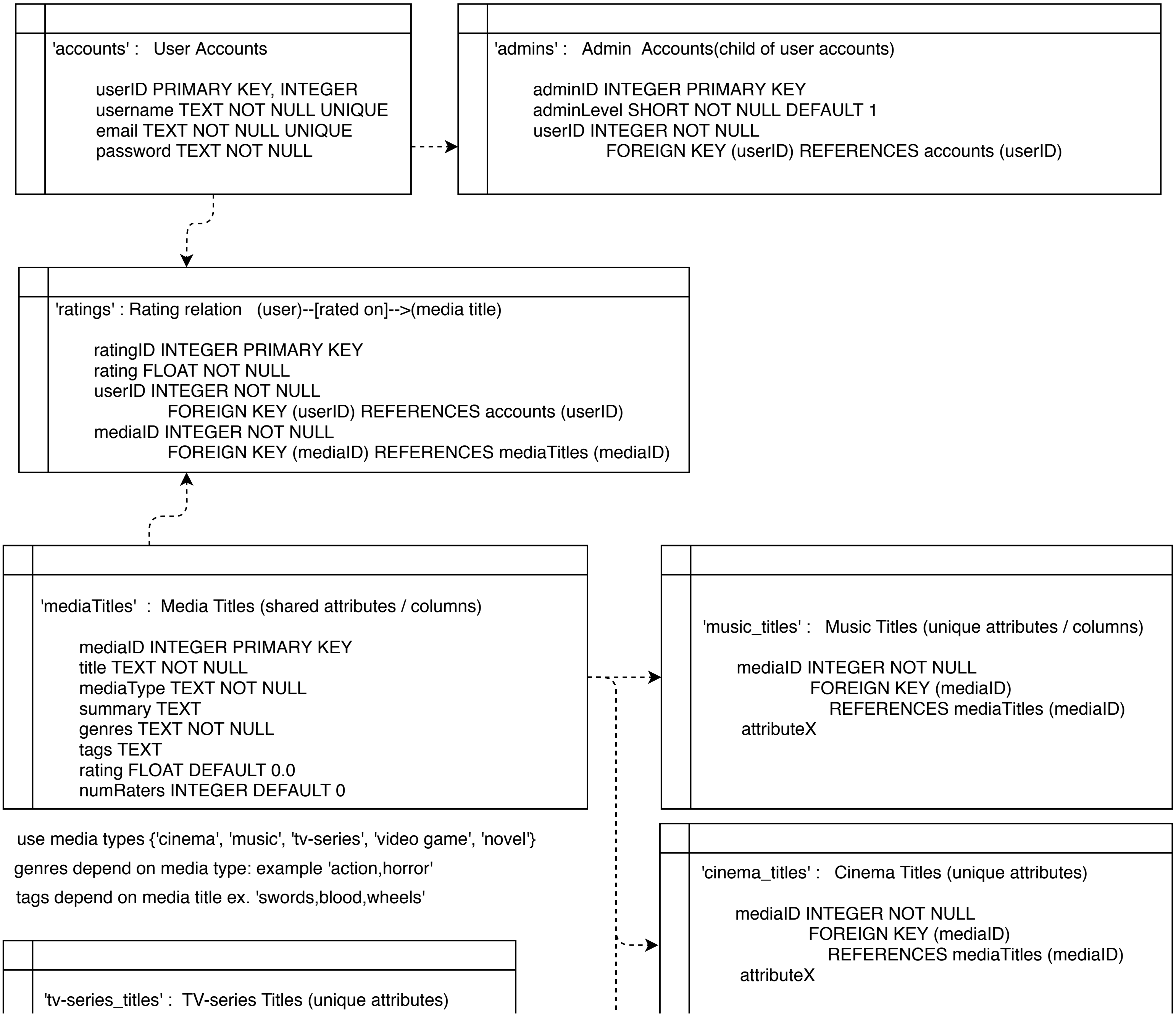
communication over HTTPS

Server
uses HTTPS server to accept request from clients (the app) and performs some task depending on the servlet.
The server then communicates with theSQLite3 database via jdbc driver.

communicate over SQLite query language

SQLite3 database
holds all data pertaining to accounts and media title information

Database (this is a work in progress)



'accounts' : User Accounts

userID PRIMARY KEY, INTEGER
username TEXT NOT NULL UNIQUE
email TEXT NOT NULL UNIQUE
password TEXT NOT NULL

'admins' : Admin Accounts(child of user accounts)

adminID INTEGER PRIMARY KEY
adminLevel SHORT NOT NULL DEFAULT 1
userID INTEGER NOT NULL
FOREIGN KEY (userID) REFERENCES accounts (userID)

'ratings' : Rating relation (user)--[rated on]-->(media title)

ratingID INTEGER PRIMARY KEY
rating FLOAT NOT NULL
userID INTEGER NOT NULL
FOREIGN KEY (userID) REFERENCES accounts (userID)
mediaID INTEGER NOT NULL
FOREIGN KEY (mediaID) REFERENCES mediaTitles (mediaID)

'mediaTitles' : Media Titles (shared attributes / columns)

mediaID INTEGER PRIMARY KEY
title TEXT NOT NULL
mediaType TEXT NOT NULL
summary TEXT
genres TEXT NOT NULL
tags TEXT
rating FLOAT DEFAULT 0.0
numRaters INTEGER DEFAULT 0

'music_titles' : Music Titles (unique attributes / columns)

mediaID INTEGER NOT NULL
FOREIGN KEY (mediaID)
REFERENCES mediaTitles (mediaID)
attributeX

use media types {'cinema', 'music', 'tv-series', 'video game', 'novel'}

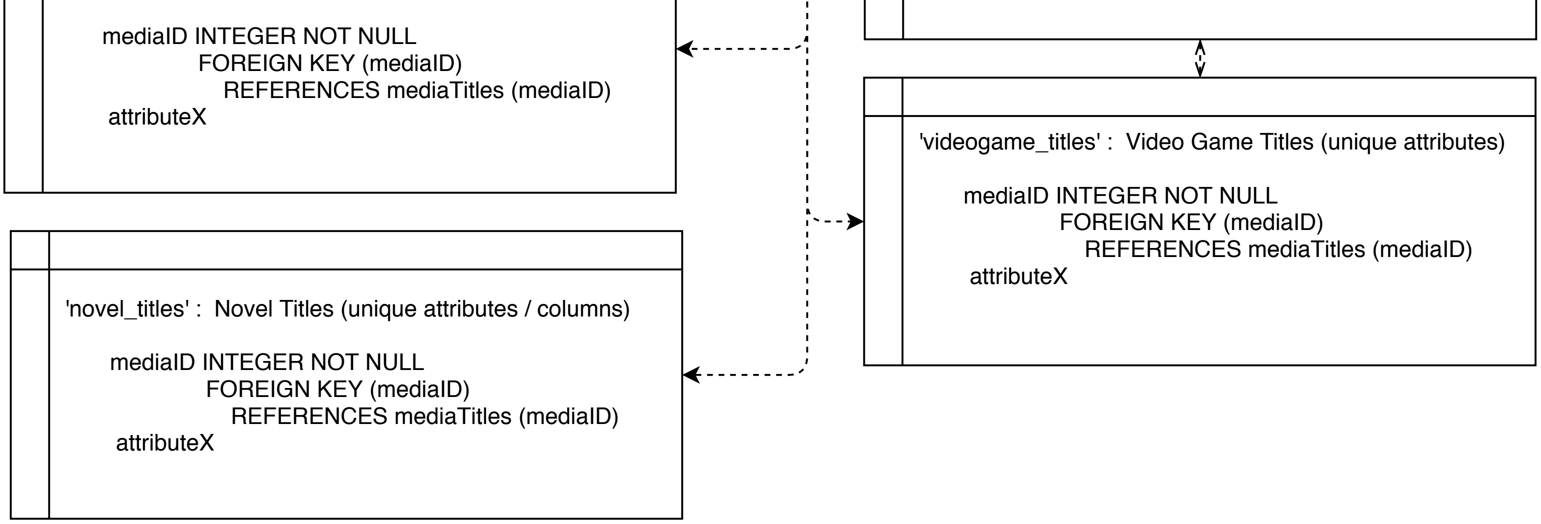
genres depend on media type: example 'action,horror'

tags depend on media title ex. 'swords,blood,wheels'

'cinema_titles' : Cinema Titles (unique attributes)

mediaID INTEGER NOT NULL
FOREIGN KEY (mediaID)
REFERENCES mediaTitles (mediaID)
attributeX

'tv-series_titles' : TV-series Titles (unique attributes)



```
mediaID INTEGER NOT NULL
            FOREIGN KEY (mediaID)
                REFERENCES mediaTitles (mediaID)
 attributeX
```

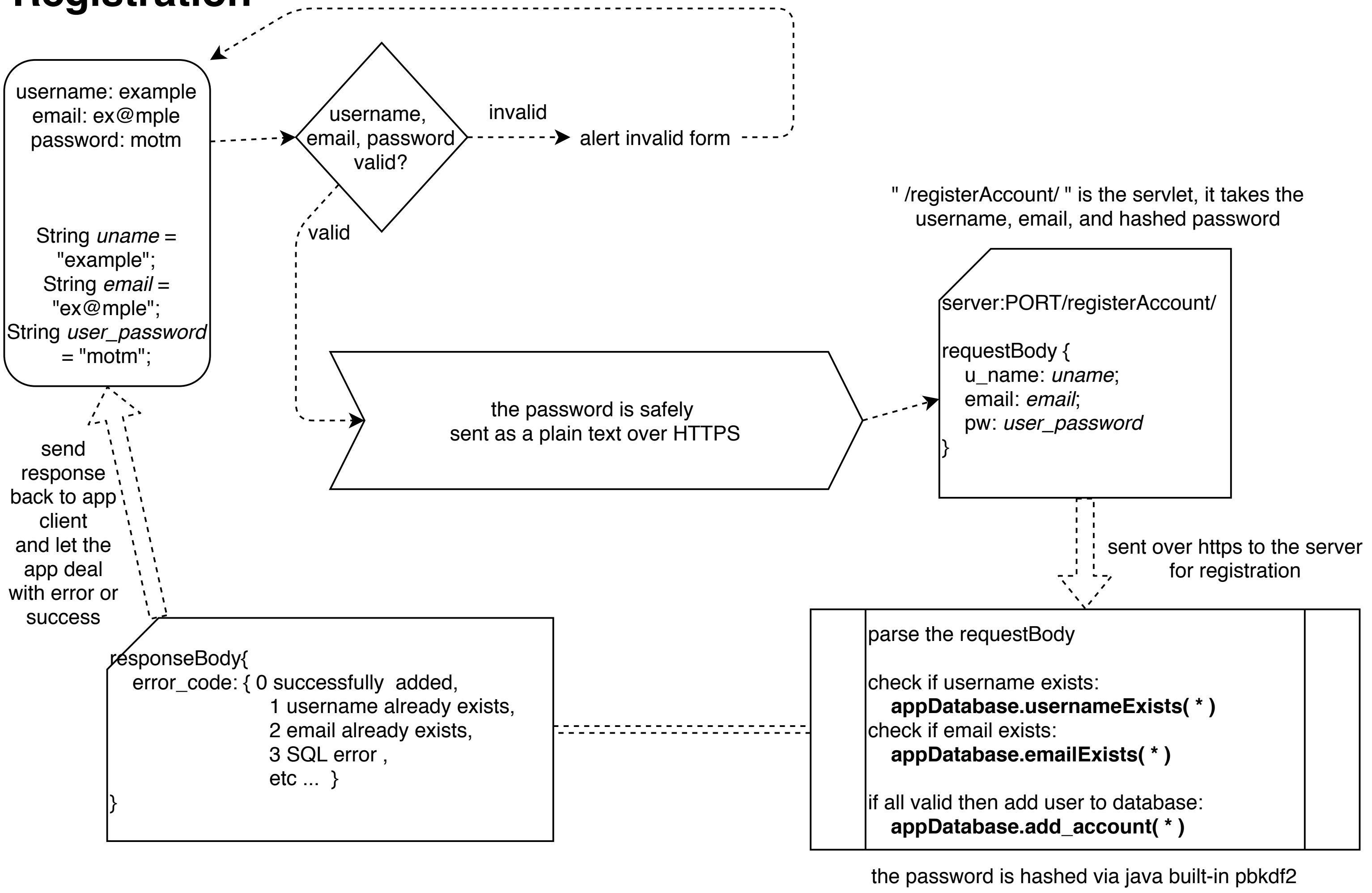
'videogame_titles' : Video Game Titles (unique attributes)

```
mediaID INTEGER NOT NULL
            FOREIGN KEY (mediaID)
                REFERENCES mediaTitles (mediaID)
 attributeX
```

'novel_titles' : Novel Titles (unique attributes / columns)

```
mediaID INTEGER NOT NULL
            FOREIGN KEY (mediaID)
                REFERENCES mediaTitles (mediaID)
 attributeX
```

Registration



username: example
email: ex@mple
password: motm

String *uname* = "example";
String *email* = "ex@mple";
String *user_password* = "motm";

" /registerAccount/ " is the servlet, it takes the username, email, and hashed password

the password is safely sent as a plain text over HTTPS

```
server:PORT/registerAccount/

requestBody {
    u_name: uname;
    email: email;
    pw: user_password
}
```

sent over https to the server for registration

parse the requestBody

check if username exists:
appDatabase.usernameExists(*)
check if email exists:
appDatabase.emailExists(*)

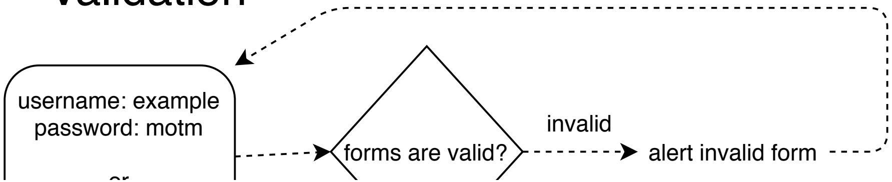
if all valid then add user to database:
appDatabase.add_account(*)

the password is hashed via java built-in pbkdf2

```
responseBody{
    error_code: { 0 successfully  added,
                        1 username already exists,
                        2 email already exists,
                        3 SQL error ,
                        etc ...  }
}
```

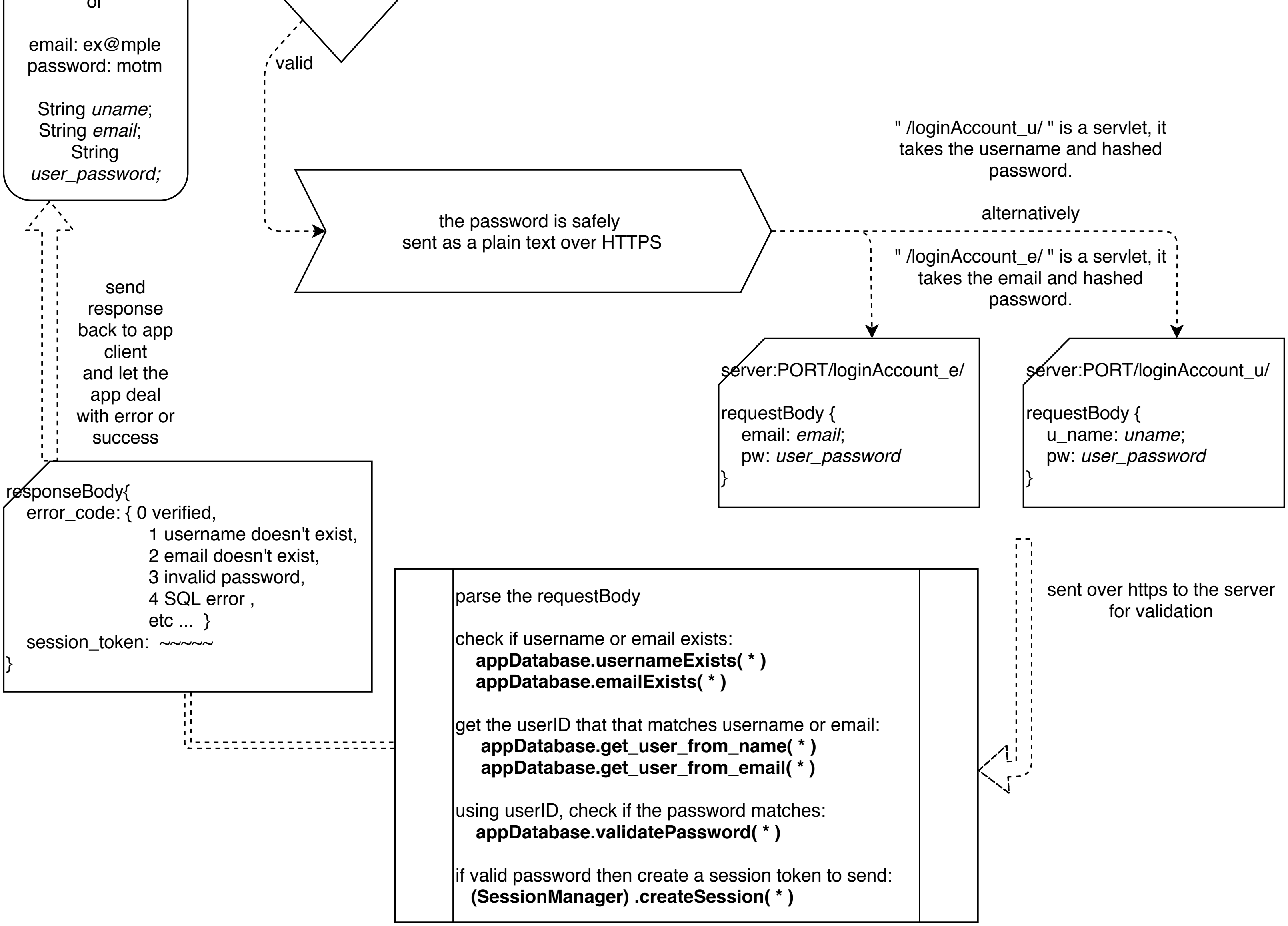
send response back to app client and let the app deal with error or success

Validation



username: example
password: motm

or



or

email: ex@mple
password: motm

String *uname*;
String *email*;
String
user_password;

the password is safely
sent as a plain text over HTTPS

" /loginAccount_u/ " is a servlet, it takes the username and hashed password.

" /loginAccount_e/ " is a servlet, it takes the email and hashed password.

```
server:PORT/loginAccount_e/

requestBody {
   email: email;
   pw: user_password
}
```

```
server:PORT/loginAccount_u/

requestBody {
   u_name: uname;
   pw: user_password
}
```

sent over https to the server for validation

parse the requestBody

check if username or email exists:
appDatabase.usernameExists(*)
appDatabase.emailExists(*)

get the userID that that matches username or email:
appDatabase.get_user_from_name(*)
appDatabase.get_user_from_email(*)

using userID, check if the password matches:
appDatabase.validatePassword(*)

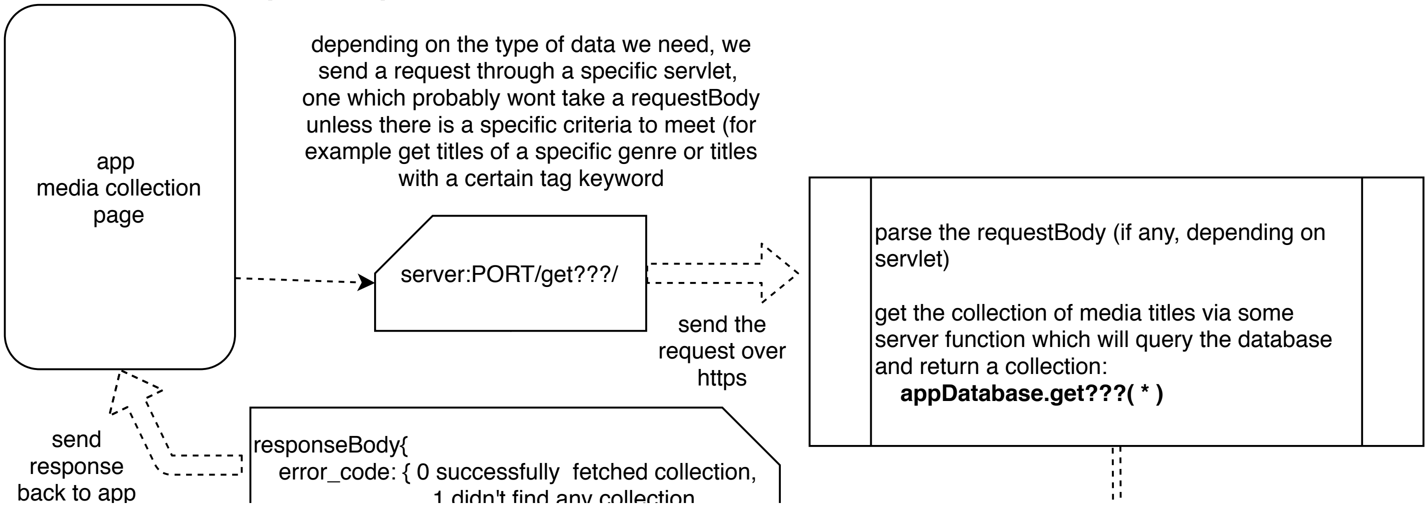
if valid password then create a session token to send:
(SessionManager) .createSession(*)

```
responseBody{
   error_code: { 0 verified,
                        1 username doesn't exist,
                        2 email doesn't exist,
                        3 invalid password,
                        4 SQL error ,
                        etc ...  }
   session_token:  ~~~~~
}
```

send response back to app client and let the app deal with error or success

The given password is is verified by applying the hashing function and comparing the differences between the stored hash and the calculated hash.
The session manager will keep track of all active sessions and remove expired sessions after a set time

get collection of Media titles (for example top 10)



depending on the type of data we need, we send a request through a specific servlet, one which probably wont take a requestBody unless there is a specific criteria to meet (for example get titles of a specific genre or titles with a certain tag keyword

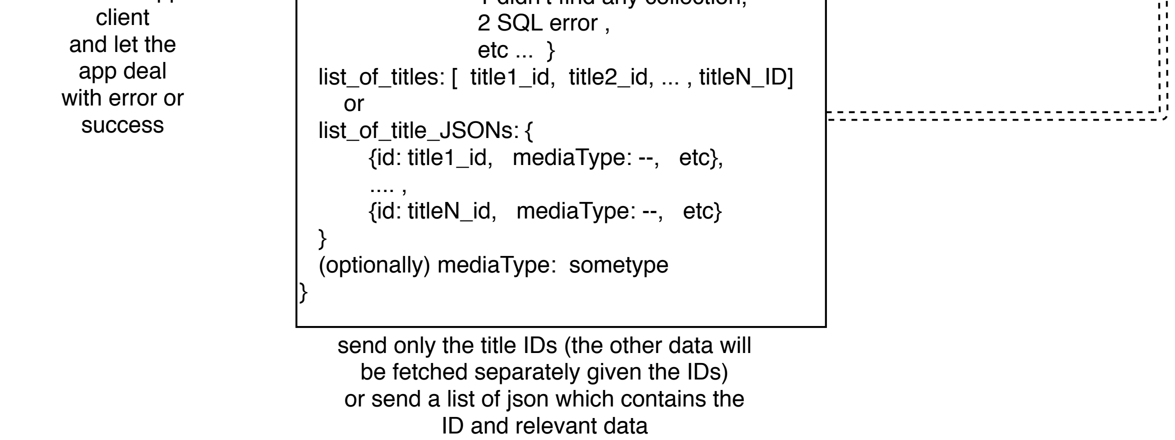
send the request over https

parse the requestBody (if any, depending on servlet)

get the collection of media titles via some server function which will query the database and return a collection:
appDatabase.get???(*)

send response back to app

```
responseBody{
   error_code: { 0 successfully  fetched collection,
                        1 didn't find any collection
```

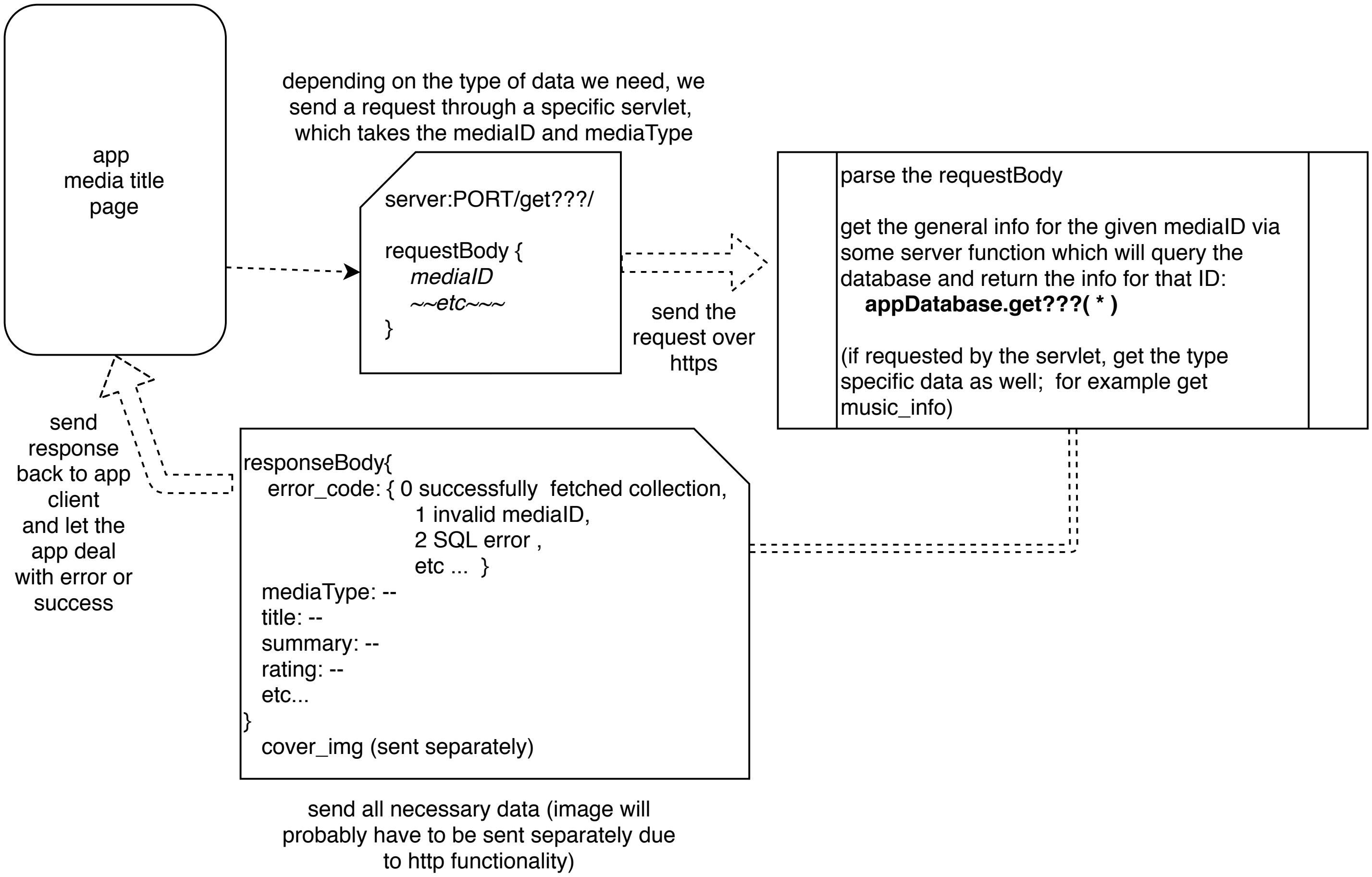


client
and let the
app deal
with error or
success

```
                    1 didn't find any collection,
                    2 SQL error ,
                    etc ...  }
  list_of_titles: [  title1_id,  title2_id, ... , titleN_ID]
    or
  list_of_title_JSONs: {
        {id: title1_id,   mediaType: --,   etc},
        .... ,
        {id: titleN_id,   mediaType: --,   etc}
  }
  (optionally) mediaType:  sometype
}
```

send only the title IDs (the other data will be fetched separately given the IDs) or send a list of json which contains the ID and relevant data

get Media title info



depending on the type of data we need, we send a request through a specific servlet, which takes the mediaID and mediaType

```
server:PORT/get???/

requestBody {
    *mediaID*
    *~~etc~~~*
}
```

send the request over https

parse the requestBody

get the general info for the given mediaID via some server function which will query the database and return the info for that ID:
appDatabase.get???(*)

(if requested by the servlet, get the type specific data as well; for example get music_info)

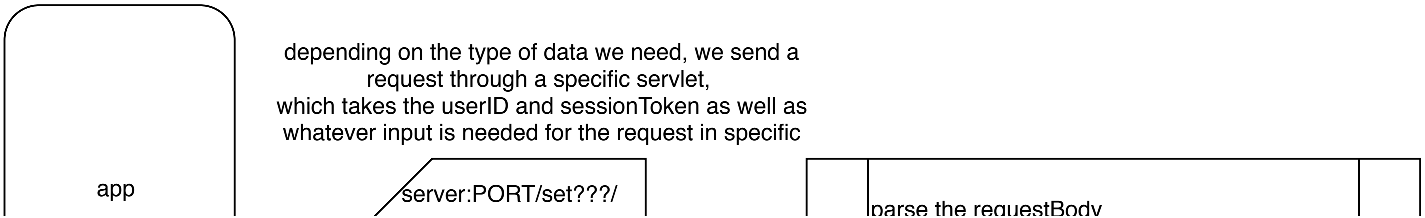
send response back to app client and let the app deal with error or success

```
responseBody{
   error_code: { 0 successfully  fetched collection,
                    1 invalid mediaID,
                    2 SQL error ,
                    etc ...  }
  mediaType: --
  title: --
  summary: --
  rating: --
  etc...
}
  cover_img (sent separately)
```

send all necessary data (image will probably have to be sent separately due to http functionality)

make account sensitive request (WIP)

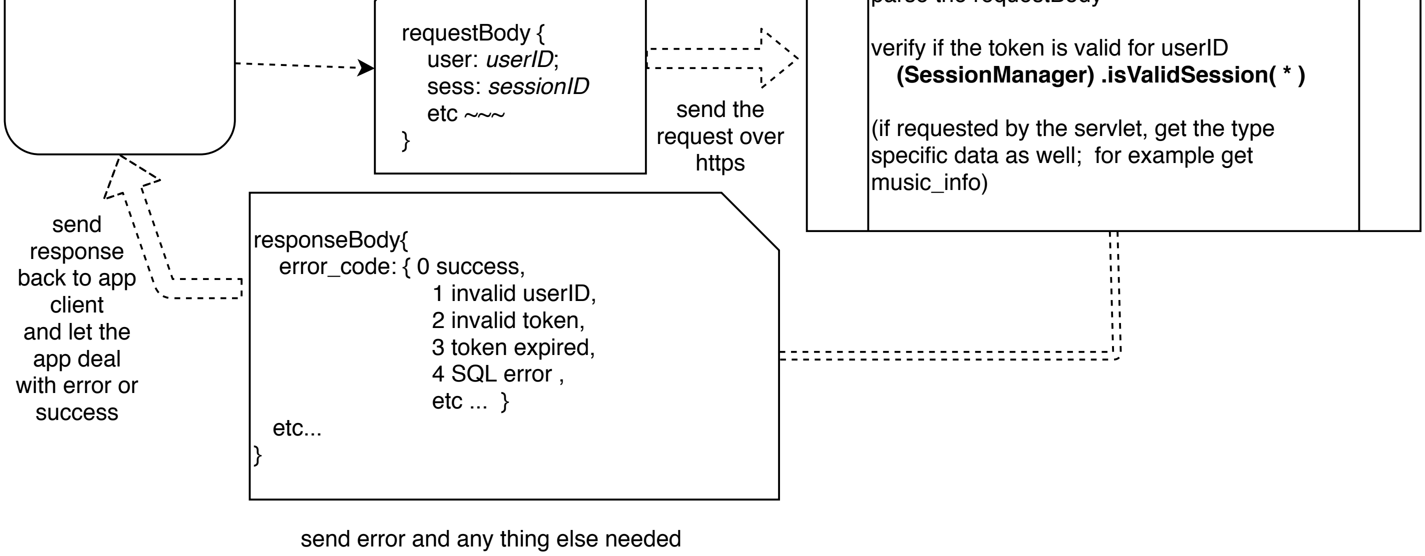
(post review, set rating, make changes to account page, etc)



depending on the type of data we need, we send a request through a specific servlet, which takes the userID and sessionToken as well as whatever input is needed for the request in specific

server:PORT/set???/

parse the requestBody



parse the requestBody
verify if the token is valid for userID
(SessionManager) .isValidSession(*)
(if requested by the servlet, get the type specific data as well; for example get music_info)
send the request over https
send response back to app client and let the app deal with error or success
send error and any thing else needed

```
requestBody {
    user: userID;
    sess: sessionID
    etc ~~~
}
```

```
responseBody{
    error_code: { 0 success,
                        1 invalid userID,
                        2 invalid token,
                        3 token expired,
                        4 SQL error ,
                        etc ...  }
   etc...
}
```

AppDatabase

MediaProfilePage

• Stores information about a media profile pages in an object • Stored information will be retrieved and displayed on the UI	

ReviewFragment

• Created functionality to the page • user is able to add,edit, and delete reviews	

reviews_page_fragment.xml

• Updated the layout to work with the functionality	

AppDatabase

Reviews

• created database for the reviews • stores information that is linked to each review made	

Server

• host the https server • main handler of session manager • main handler of database	• InetSocketAddress • SSLContext • SessionManager • AppDatabase • registerAccount • validateAccount

registerAccount

• communicate with the AppDatabase and Server to handle a request from a client, take appropriate action by interacting with the database, then returning a reply to the client • register an account with the arguments provided in the request body • (might return a sessionID on registration, allowing user to proceed after registration rather than sending back to the login page)	• AppDatabase • (SessionManager)

validateAccount

• communicate with the AppDatabase and Server to handle a request from a client, take appropriate action by interacting with the database, then returning a reply to the client • validate account credentials with the arguments provided in the request body • return a session ID on valid credentials	• AppDatabase • SessionManager

HelloWorld

• communicate with the Server to handle a request from a client and send a reply to the client • testing communication through the connection	

AsyncTask

RegisterAccount

• Handles communicating registration requests with the HTTPS client • Replaces the fragment if the request was successful	• SecureHTTPSClient

AsyncTask

ValidateAccount

• Handles communicating login requests with the HTTPS client • Replaces the fragment if the request was successful	• SecureHTTPSClient

SecureHTTPSClient	
• Sets trusted HTTPS certificates • Sends and recieves HTTPS requests	

SignupPageFragment	Fragment
• Sends signup requests to RegisterAccount • Checks if the email is in a correct format • Checks if password and passwordConfirm are the same	• RegisterAccount