G.O.A.T.

Go on And Try: A Media Aggregator and Recommender Application

By: Sean Hopkins, Peyton VanHook, Peyton White

*GitHub: https://github.com/peytonthehuman/GOAT*

# Introduction

The GOAT is a media recommender for those in a drought of media. The GOAT recommends three types of media: books, movies, and video games. It is a luxury for those conosures of media with specific tastes. A luxury for those that have the hardest time finding something new to fill their time. The GOAT is useful for anyone whether they be old, young, goth, nerdy, jockey and anything in between. Whatever the preference, the GOAT recommender is here to cater to each user's specific interests based on their watch history. Anyone who has binged a seven season series in a matter of days has dealt with the thirst of more entertainment. Sifting through the mountains of media that are at the tip of a finger can be a time consuming and arduous job and the GOAT wishes to help.

GOAT stands for “Go On And Try” and we, the developers, wish the users to do so as they please. This application has been implemented in the efforts to be a useful tool for users. What the GOAT does is it uses user specific information to give them the best recommendation its algorithms can muster. The GOAT uses multiple ways to filter media for the user and give them a myriad of tools to work with.

Users are given many useful ways to find the next media to indulge in that is just right for them. A user has their own profile with their user specific information on display as well as their Saved Media List. The Saved Media List is where media that has been saved for consumption at a later date is saved for the user.

The user also has the option to visit the Home screen. On this screen, the user is given multiple lists based off of information gained by the users ratings and saved media. These lists may be based on genres or other keywords that the user has been seen to enjoy. This tool gives the user more options to choose from while still trimming the fat of the vast mountain of media that would be otherwise cumbersome to sift through.

The next capability of the GOAT is its search capabilities. The GOAT gives the user the option to search specific media titles and recommend other media solely based off of that specific media. This tool is useful in the way that it is more specific than the Home screens recommender.

The last main function for recommending media is the Media Info window. This window gives the user a more in depth view of the media which they are looking to try out. The Profile, Home, Search windows and even the Media Info window itself have the capability to reach this screen as all of them have tables with media recommendations in them. The Media Info window also has the capability to show the user a few media titles that are related to the title being displayed, thus giving the user more options to find something they might enjoy.

What we hope this will do is give the user multiple routes to find new media to enjoy. The user may find media specific to one media title, genres and keywords. In giving users multiple ways to search for related media we hope that users are able to find suitable entertainment for their preferences.

# Technology

The technology used in the GOAT is as follows. For the GUI side of things we are building our GOAT project in Netbeans 12 using the FXML JavaFX Maven Archetype. The project consists of Java Class and FXML files that interact with one another. The Java Classes are used to create specific classes or to control actions by the user depending on how they interact with the specified FXML.

There are class files such as Media, Book, Movie, VideoGame and User. Each is used to hold the data in order to display the specified media correctly. Each holds the typical functions of getters and setters for all of its components as well as default constructors to create instances in other Java Class files. The controller Java Class files are named after the FXML file that they control. As an example, the profile.fxml file has the Java Class controller ProfileController.java to coordinate all of the options given to the user. Each controller works in this way and is needed for each FXML file as they each work differently.

The FXML files are used to display the GUI and give users options to choose from in order to traverse through the GOAT recommender. As stated previously, each FXML file is different and needs their own controllers linked to them in order to perform the correct tasks. The FXML files have been edited and connected using SceneBuilder. This is a useful tool in effectively creating an FXML without having to do all of the coding and connecting oneself.

We chose PHP as our backend language

# Software Design

## Data merging

We built the GOAT application around the idea that it could recommend related video games, books, and movies. In our research we discovered most aggregators only focused on one of those media types, and many data sources would only contain information on either books, movies, or video games. As such we had to create a procedure to merge these disparate data sources into our data schema.

Our merging procedure is a collection of PHP scripts which run on the local system. They read from locally saved source files downloaded from the internet and save their result to a collection of locally saved CSV files which match the media-related tables in our database. A separate PHP script is then used to read these CSV files and copy the information into the database.

The first script file, index.php, contains the ‘main’ function of the merging procedure. This function does a number of things. First, it includes the other source files needed for the rest of the procedure. Second, it creates files which all media processors must write to, namely the genre table (genre.csv), the keyword table, and the genre and keyword occurrence arrays. Then, for each media type, the main function runs the initialization, parsing, and deinitialization functions and prints feedback to the console. Finally, at the end the procedure merges the genre and keyword occurrence arrays by identifying which tags are shared, if any, and saving the sum of any shared tags to a single entry in the array. It then saves the resulting array to the genreOcc.csv file.

For each media type, there exists a corresponding parsing procedure file (movie\_source.php, vgame\_source.php, and book\_source.php). Each of these files contain five primary components: an initialization function, a parsing function, an output function, a scanning function, and a deinitialization function.

The initialization function is the first function called from each file by the main function. It opens the file resource to the appropriate data source file and saves those resources into an array with an appropriate key. It saves the resources in this way so that multiple sources can easily be used. This becomes particularly useful for our book source data, which is split into 42 separate files. The following code is used to establish a connection to each of the files:

for($i = 1; $i < 42; $i++) {

$iterString = '';

if($i < 10) {

$iterString = '0' . (string)$i;

} else {

$iterString = (string)$i;

}

$path = $dataSourcePath . 'Books\\book.p' . $iterString . '.xml';

print($path);

$bookSourcesConnArray[$iterString] = connectToDB($path, 'r');

}

In addition to this, the initialization function establishes a file resource to the media output file, which is shared by all media types, and the individual media type output file (movie.csv, book.csv, vgame.csv) which stores data unique to each media type. This structure reflects the structure of our database, which uses separate tables to store media type specific information.

The main function then calls the scanning function, which initiates the procedure used to scan through the data sources. For video games and movies, these functions are called parseVgameData and parseMovieData respectfully. Because the book sources are split into different files, the main function first calls parseAllBookCollections, which then individually calls the parseBookData function for each input file. Each data source is different, so the functions diverge at this point. However, despite this each function utilizes some kind of parsing routine to put the data in a usable format and then calls an output function to write that data to the media and media-type-specific files.

For the movie source, this procedure is relatively simple. The scanning function reads through each line in the input file, which is formatted as a CSV with the tab character as a delimiter. Once it parses the TSV line into an array of cells, it determines if cell #5, which in the source dataset stores a 1 or 0 to indicate if a film is marked as “adult” oriented, is false and if cell #1, which stores the type of film the item is as a string, stores the string ‘movie.’ The scanner only proceeds if both of these are true, if not it proceeds to the next line in the source file.

After this, the scanner moves to the ‘parsing phase’, which in the case of the movie data is not a discrete function. All it needs to do is parse the genre list stored with the media item (which is stored as a comma-seperated string) and merge the media genre list with the master movie genre occurrence array. If a genre is not yet in the list, it adds the genre as a string key with an associated value of 1. Otherwise, it increments the value saved with the appropriate genre string key. After it does this, it directly writes the appropriate values from the parsed TSV into the output files.

The video game scanner is somewhat more complicated in that the parsing function is a discrete function. This is necessary because the video game data is stored as JSON objects, with one JSON object for each line of the text file. So the scanner is able to read each line individually, pass it to the parsing function, and, if the parser does not return false, it continues similarly to the movie scanner. It checks to see if the entry is marked as having the “game” type, and if it does it merges the genres with the genre occurrence array and writes the entry to the output files.

The parsing function itself uses the built-in json\_decode function to decode the object into a PHP object (the associative parameter is set to false). The function then checks to see if the success member is set to false (if it is, this would indicate that the particular line in the data source contains no actual data). If this is the case, the function returns false. Otherwise, it directly copies each member of the JSON object to appropriate local variables. For example, the function copies $obj->data->steam\_appid to the $id local variable, which is sent back to the scanner. It does this for each variable except for the platforms variable and the genre array. For the platforms, it uses the platform flags to determine which of the three major OS’s (Windows, Mac, Linux) the game can support, and writes each supported platform to an array. The array is then converted to a CSV line and saved to a variable. For genres, the parser saves the list as an array.

The book parser is significantly more complicated than the other media sources. The book parser was written based on documentation provided on the MARC 21 bibliographic data format created by the United State Library of Congress. The MARC 21 format is meant to categorize entries on over 38 million items in the Library’s collection, and is appropriately complex. The MARC 21 format is an XML based format, and our parser utilizes the simple\_XML parser to decode it. The source file has each XML entry saved over several lines of text, so the parser first must read each line of the record and build a string to decode. If the next line of text in the file resource does not contain the text “<record>”, the function throws an error with code 42. The scanner function will then relaunch the parser, which we know is safe because in each collection file the only line which does not start with <record> is the first and last lines, which instead are <collection> tags. Relaunching the function a few times allows the scanner to move the file pointer to the first record, and the parsing function stays aligned until the end of the file.

Our parser uses the “simple XML object” exported by the decoder function to process the book data. It first checks to see that the leader string, which is the first data element of each entry in the MARC 21 file, contains either a ‘a’ or ‘t’ in position 6 of the string and either a ‘a’, ‘c’, ‘d’, or ‘m’ in position 7. The characters in position 6 indicate the record is either “language material” (i.e., written) or is a manuscript, while the characters in position 7 indicate what type of bibliographic entry the data is. The parser then checks position 35-37 of the control field to determine if the entry is English. Entries are only added if all of these conditions are met.

The parser then checks positions 24 to 27 of the control field to filter any materials that are marked with any flag in that set (which would indicate that they were a catalog, encyclopedia, legal publication, or some other reference material) unless the flag is ‘6’, which indicates a comic book or graphic novel, in which case the item is included. If the item is filtered out, the parser returns 0.

The parser then uses position 33 of the control field to determine the item’s genre, which is encoded by a single character. It also uses position 22 of the control field to determine the audience, which it saves as a genre.

After this processing has been done, the parser runs a foreach loop over each datafield in the entry and extracts information from the appropriate positions. Each data entry is marked with a “tag”, which indicates what information the entry encodes based on the MARC 21 specification. Tag 020 encodes the ISBN, so the parser looks for a data entry with tag 020 and, when it locates it, records the ISBN to the return array (it also sets the $hasISBN tag to true, so that the function can return 0 if the entry has no ISBN). The parser also uses tag 245 to extract the title using the following code:

// Title

if($dataRecord['tag'] == '245') {

$title = $dataRecord->subfield[0] . ' ' .

$dataRecord->subfield[1];

$pos = stripos($title, '/');

if($pos !== FALSE) {

$title = substr($title, 0, $pos);

}

$retObj["title"] = $title;

}

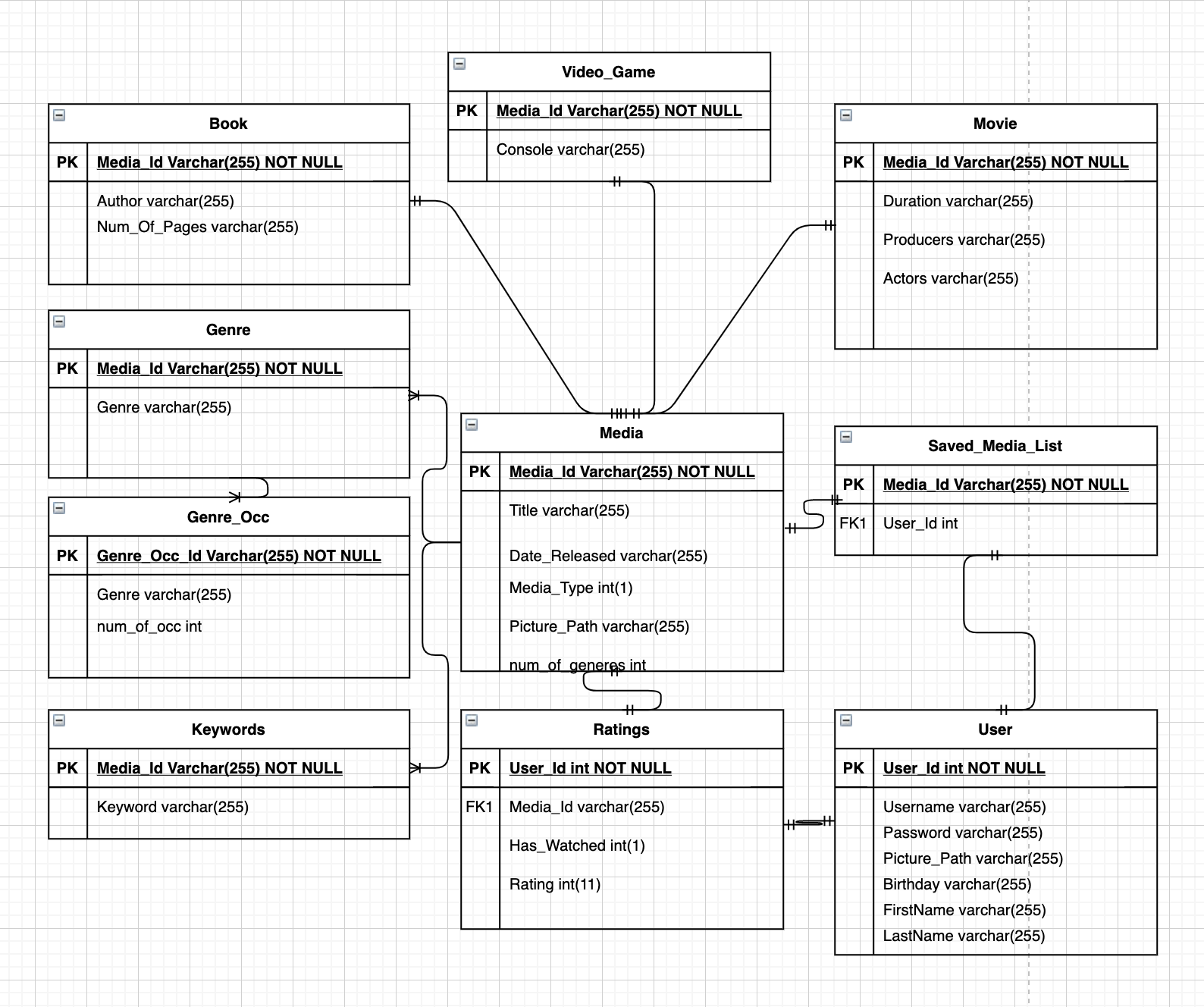
The title is encoded into a main title and subtitles, so this code recommends these elements into a single title field with a known separator. The parser then uses either tag 100 or tag 700, depending on how the data flags are set, to extract the Author’s name, and uses tag 260 to record the release date. Once all these components are gathered into the return array, the parser returns the data to the scanner function.

The scanner then checks to see if the function returned error 42, in which case it sets the return value to zero. After checking for errors, it runs an if statement. If the function returned -1, then the scanner has reached the end of the file and it returns. If the function returned 0, then the scanner continues along the file. If the function returned anything else, it merges the saved genre array into the bookGenreArray and saves the data to the output files.

In each of these parsing cases, an almost identical output function is used. The output function writes the appropriate data (passed in as individual parameters to the function) to the media file, the media type table file, and to the genre bridge file. This genre bridge file represents the genre bridge table, which encodes each genre entry as a separate line with the media ID it’s associated with.

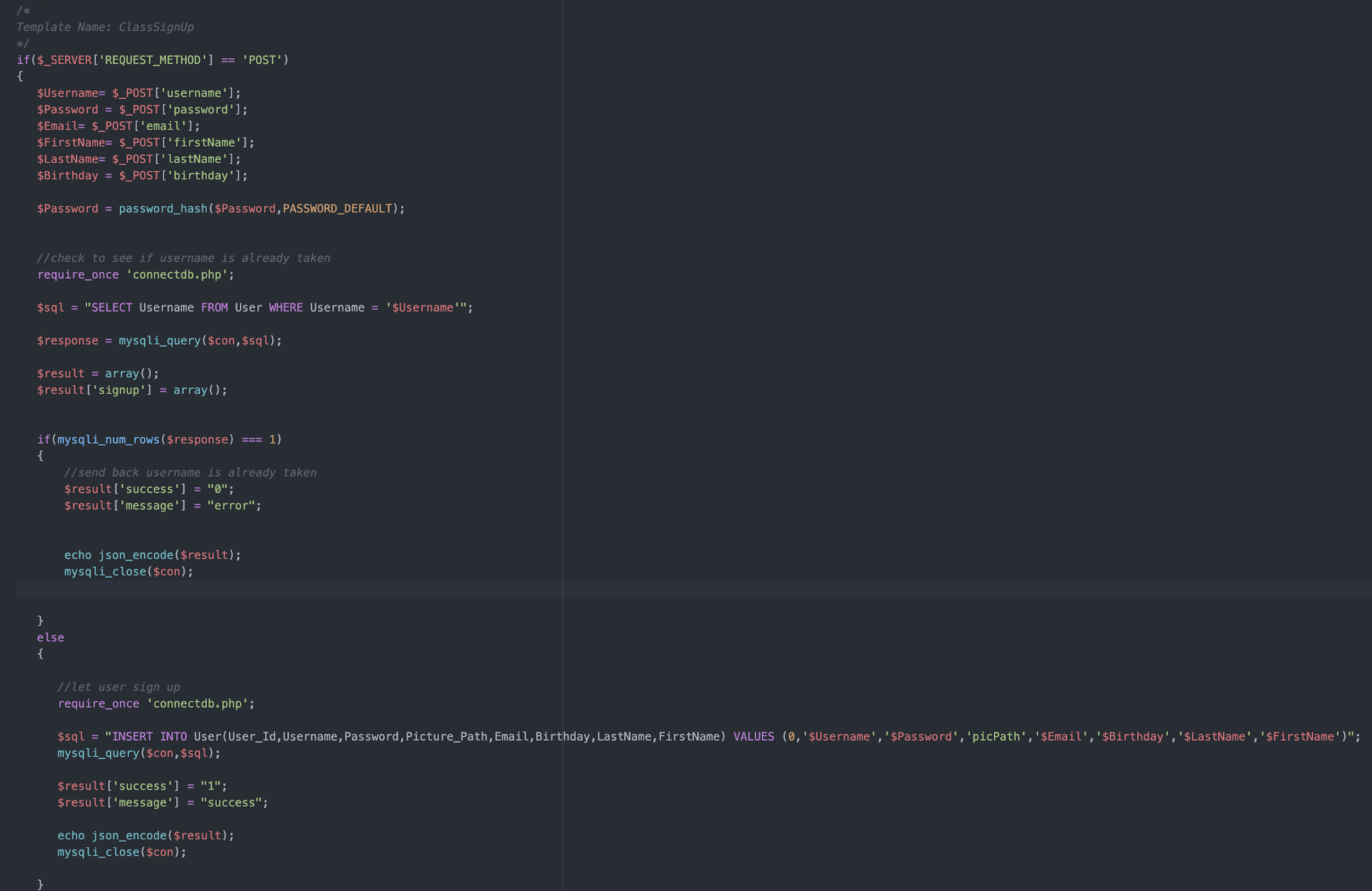
Once the scanner function finally returns, the main function calls the deinitialization function, which closes the file resources for each source file and each output file, and sets each variable to NULL.

# Database

The database consists of 10 tables that separate the data from being redundant. 

All server side scripts are PHP files with PHP and SQL. All scripts either collect data from the database or write data to the database. That is their purpose. All scripts interact with the application(client-side) through a URL. Each script has a different URL that does a specific thing. Each URL is activated through the Post method for security reasons. The scripts are as follows.

1. Connectdb.php is a simple file that connects to the database. Creating this file allows all other files to connect to the database by just calling this file.
2. Signup.php is a file that takes data from the user, username and password, and signs the user up for a profile.



1. CheckLogin.php checks if the login is correct by going through the database and finding the username and password.
2. Read.php is how the database gets populated. It reads through 5 csv files which are on the server. The PHP separates the media and populates the database accordingly.
3. GetMovies.php gets 200 movies from the database and then sends it back to the application in the form of an array.
4. Putinuser.php is how the user saves media. If a movie is double clicked the specific URL gets called and that media is put into the saved\_media\_list in the database.
5. getMoviesFromUser.php gets the saved media from the user. The application sends the user id to the database and in array form gets back all the saved movie.

# Bugs

One bug that is a constant battle is the loss of data through the passing of windows. Not the user data but when trying to save data when going from one window to the other and back to the original window some global variables will be reset, and we can not seem to 100% fix that issue. One problem is the way table views are set up. Table views get initialized before the actual window does, so when trying to use global variables inside that class you can not, because they are null. Consequently we tried another way setting up the windows. Some windows are pop ups like the profile window. This allows us to open the controller and use data before initializing the table view. A good approach that a team would solve this problem is simply knowing the answer and making every window open the same way, that way everything is repetitive and simple.

Additionally, we’ve had a number of performance issues with the dataset. Specifically, we’ve had difficulty actually importing the dataset exported by the merging function into the web database due to long response times. We believe that this is a design flaw rather than a bug, and if we had more time we would rewrite the output portions of our merging script so that it would immediately export the data to the database rather than requiring a middle-man program.

# Future work

We have great ideas for the GOAT going forward. Due to time constraints and experience we were not able to implement them now but wish to develop, fine-tune and add them on at a later date. This would include things such as: cover art display in table views, being able to add and chat with friends as well as send suggestions to each other, edit the background of the users profile to their liking, preferences and more. Since this application is a luxury we hope to make it as customizable as possible. Not only will the users have their own recommendations specific to their saved media and history of watched media, but they will be able to edit the layout of their profile to their liking and visit others as well.

Additionally, we ran out of time while we were trying to get our data into our web server and get our data to display, so we weren’t actually able to implement the recommendation algorithm portion of the application. Implementing and tuning this algorithm would be number 1 priority in the next few months of development on this project, as would incorporating more detailed keywords, a more coherent genre set, and other secondary information into the database to boost the algorithm’s predictive capabilities.

Applications have been leaning toward the social aspects lately and we plan to add such features. We hope that in adding these features that it makes the GOAT more alluring as it gives users the ability to customize their own profile as well as communicate with friends to get the best experience and recommendations they can. By creating the capability to communicate with friends with similar interests or even opposing interests gives the user recommendations that code could not, thus creating another possibility for the user to find something that they enjoy. Instead of creating another algorithm, we use what is already there to give the users more capabilities.