Final Project

ICSI 410

Team: Maksim Papenkov, Matthew Welton, Riley Wilkins

Project Introduction/Overview:

This project develops a database for movies, replicating IMDb (Internet Movie Database) - though with a simpler implementation. Movies - as well as books or albums - can be clearly represented through a relational database due to the high-dimensionality of the data. Each individual movie corresponds to one (or multiple) directors, multiple actors, and multiple genres, among other features. Similarly, most directors and actors work on multiple movies - and in general, there is not a one-to-one relationship between people and movies, requiring a complicated database structure.

This report includes two distinct components : a “prototype” of the IMDb-type application, and a plan for expansion. Of three members in the group, only one has significant programming experience using HTML, CSS, JavaScript and EJS, and therefore the work involving implementing the program could not be split evenly. Despite this challenge, this report includes an “ideal” version of the database - which is not currently implementable, though satisfies the requirements of a well-defined and efficient relational database. This “ideal” and unimplemented version is the subject of the E-R diagram (found in the Appendix), as well as much of the discussion. The current runnable database application features fewer tables, and is not normalized - though the importance normalization is discussed.

The current runnable database includes a simple user interface. Users or clients are allowed to search for and retrieve both movies and actors, as well as write reviews. To maintain the quality and integrity of the data, two types of users are defined: administrators and non-administrators. Only administrators are able to update, create, or delete entries in the database - while all types of users are able to read entries from the database using the search features. One exception to this rule is that users can write/create movie reviews. This ensures that the database will contain correct and appropriate information, which is highly desirable for any application with a high amount of regular traffic.

Requirements:

1.1 Required Data

This project develops a small-sample original dataset from scratch, to be able to implement the application and design from the ground-up. The focal point of this database is a table titled “Movies”, which links to all other tables through various relations. By using a self-built dataset, this allows the database to include a “User” table, which keeps track of all individuals who interact with the database - and perform CRUD functions. Users who are not labeled as “administrators” are allowed to Read within the database, as well as write reviews - while all other functions are blocked for them. On the other hand, users who are labeled as “administrators” may also Create, Update, or Delete records in the database. To enforce the security of the database, all administrator functions are password-protected.

1.2 Entities, attributes, relationships

In the “ideal” implementation of the IMDB database application, there would have six distinct entities: Movies, Actors, Directors, Genres, Reviews, and Users. In a fully normalized implementation of our database there would also be additional tables linking the entities together, though they would not be classified as entities in their own right. This normalized implementation is described as follows.

Each of the entities (tables) contains one primary key, and several other descriptive variables. For Movies, these variables include a unique ID (to link to other tables), along with the movie’s name, genre, and year of release. The Actors table includes a unique ID, along with the actor’s name, birthdate, and biography. Similarly, the Directors table also includes a unique ID, the director’s name, birthdate and biography. The Genres table contains a unique ID, along with the genre name, and a description of that genre. Finally, the Users table represents the individuals who interact with the database and perform CRUD operations. This table includes a unique ID, along with the user’s name, email, and administrator status.

These entities are linked together with several relationship-tables (in a normalized implementation, linking the Movies table to all other tables in the database. These tables are called Acted_In, Directed_By, Which_Genre, and Reviewed_By - which improve runtime by reducing unnecessarily duplicated data in situations where there is a one-to-many or a many-to-one relationship. For example, most actors star in multiple movies (one-to-many), while most movies have a cast of multiple actors (one-to-many) - if we look in the opposite direction, then these become many-to-one relationships. The other three relationship-tables function in an identical manner.

1.3.1 Organization of the data through formal normalization techniques (Normalized Data Model with ER Diagram)

The database that is currently implemented in our application is not normalized (due to time constraints), though the "ideal" database described in the E-R diagram (See 1.5 ER diagram) has been normalized into Third Normal Form (implying all conditions for Second Normal Form and First Normal Form have been satisfied as well.

The primary restriction of First Normal Form is that there are no repeating attributes - which is clearly satisfied, as each entity table has columns with distinctly different information. For example, in the Actors table, the columns Actor_Name and Actor_Birthdate are not dependent on one another. Further, there is no specific ordering to either the rows or columns, as each observation is independent of the others - and each column contains exactly one element of data.

The normalization process is then further extended to Second Normal Form, where the additional condition of removed partial dependencies is also satisfied. For each of the entity tables, all non-key columns are dependent on the primary keys - which are unique IDs. For example, the columns Actor_Birthdate, Actor_Name, and Actor_Bio are determined by the Actor_ID primary key. An identical argument can be made for all other entity tables.

Finally, the normalization process concludes by reducing the database into Third Normal Form, which adds the requirement that all columns must UNIQUELY depend on the primary key. The example in the preceding paragraph demonstrates this, in that each Actor_Birthday, Actor_Name, and Actor_Bio are derived from the Actor_ID. Even though there may be multiple actors born on the same day, this is not a problem - as the Actor_ID, rather than the Actor_Birthdate, is what uniquely defines the entire row.

1.3.2 Normalized data model

Ideally, this database would include tables called “Acted_In”, “Directed_By”, “Which_Genre”, and “Who_Reviewed”, to eliminate redundancy due to multiple one-to-many relationships. This effectively reduces duplicate data across multiple rows, allowing for a smaller file-size.

For example, in a single movie - there are multiple actors, each of whom star in multiple movies. Similarly, most actors star in multiple movies. By including a table titled “Acted_In”, all descriptive data for each actor and movie can be separated from the linking variables, and only unique information is copied between rows. If a user wishes to find all movies with a single actor, this can be done by finding the Actor_ID, then retrieving all Movie_IDs that match that Actor_ID in the “Acted_In” table. Once those Movie_IDs are retrieved, the full information about the movies (such as title and year of release) can be accessed from the “Movies” table. This is more efficient, in terms of storage space, than simply linking the Actors and Movies tables together - because all of the actor-specific and movie-specific information (such as actor

name and movie title) are not copies across multiple rows. The other three link-tables called "Directed_By", "Which_Genre", and "Who_Reviewed" operate in an identical manner.

1.4 Key assignments

The primary key for Movies is Movie_ID, rather than Movie_Title, because multiple movies exist under the same name. For example, two movies titled "Lion King" have been released - in 1994 and 2019, respectively. The additional columns in the Movies table are Movie_Title and Movie_Year, each of which have a one-to-one correspondence to the unique Movie_ID. In a different implementation of a similar database, a combination of Movie_Title and Movie_Year can be used as a Composite Primary Key, though this is not the most efficient normalization for the data. The primary key for Actors, Directors, Genres, Users, and Reviews is also a set of unique ID variables, that operate in a virtually identical manner.

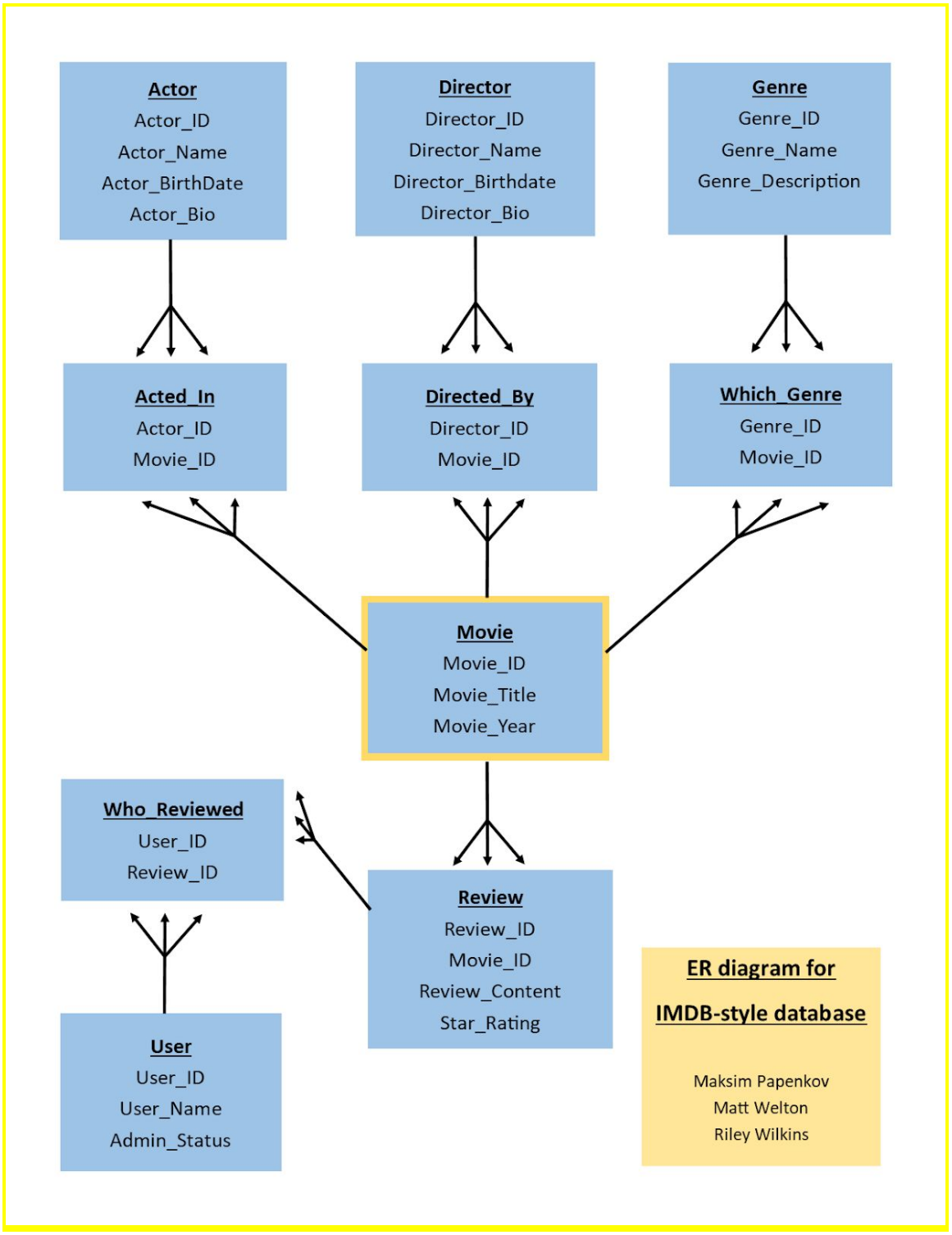
1.4.1 Data types

The four documents/tables in the Firebase database each contain a number of fields or attributes with an associated data type. These fields or attributes are primarily of type "String." This is because the majority of the data that users and admins are inputting via forms consists of fields such as name, biography, plot, table ids, and the like. These types of fields are generally associated with "String", and therefore it wouldn't make sense as a developer to assign them a different data type such as integer, or float. Although "String" is the primary data type of our

attributes, we do also utilize “Timestamp”, which was useful for logging when a user created a movie review, or when an admin created or updated a record in the movie or actor table.

With regard to the data types of variables used in the back-end logic, all functions were written in JavaScript, which is a “Loosely Typed” programming language. This means that when you declare a variable, you don’t have to specify its data type; JavaScript interprets this for you. An example of this can be seen with “var enterAnInteger, var userName;”, intuitively from the naming conventions one would assume the ‘enterAnInteger’ would be of type ‘int’ and ‘username’ variable would be a ‘String.’ However, JavaScript doesn’t care, and it is possible to initialize or change a variable’s type at any time.

1.5 E-R diagram (Third Normal Form)



Entities and Attributes (Primary Key in Bold) :

Movies (**Movie_ID**, Movie_Name, Movie_Genre, Movie_Year)

Actors (**Actor_ID**, Actor_Name, Actor_Birthdate, Actor_Bio)

Directors (**Director_ID**, Director_Name, Director_Birthdate, Director_Bio)

Genres (**Genere_ID**, Genre_Name, Genre_Description)

Reviews (**Review_ID**, User_ID, Movie_ID, Star_Rating, Review_Body)

Users (**User_ID**, User_Name, User_Email, User_Admins)

Relationship Tables :

Acted_In (Movie_ID, Actor_ID)

Directed_By (Movie_ID)

Which_Genre (Movie_ID, Genre_ID)

Reviewed_By (Movie_ID, User_ID)

1.6 Application Pages

See Appendix

Implementation:

When starting this project, the initial plan was to use Java and MySQL to create this application, but ultimately a different set of technologies was chosen. This switch was made because of issues that arose when trying to build off the Java and MySQL tutorial. One team member was having issues with setting up their environment in Eclipse for Java Developers, and there were also issues with JavaServer Pages, which would have been used to create the user and admin pages with this technology stack.

The technology stack we switched to was Express.js, Node.js, and Firebase. Express.js is a minimalist web framework that allows you to easily create a web application, while providing a nice set of built-in features. Express works on top of Node.js. Node is a JavaScript runtime environment built on Chrome's V8 JavaScript engine. The team decided to use Node because it allows for JavaScript to be executed outside of a web browser; before Node was created, JavaScript was only able to be executed inside a web browser. With Node, it is possible to code an entire application in JavaScript (other than HTML and CSS for formatting and styling). The front end is written in HTML, CSS, and embedded JavaScript, while the back-end is written purely in JavaScript.

For the database, Firebase was chosen, which is a noSQL database that is supported by Google. We chose Firebase because it has an intuitive GUI interface and its built-in support for authentication is easy to understand and set up. Firebase also excels in that it is easy to change the structure of your 'tables' (technically, documents), as well as fields and their associated data types. This was very helpful because there were several times when coding the application, that

we ran into situations where it would be easier to implement a function or method if a field was of a different data type, and Firebase allows you to easily and quickly modify fields in this way. Although Firebase is a noSQL database, the organization of our documents and their related fields is nearly identical to how a relational database is organized; this is why throughout this report we use ‘table’ and ‘document’ interchangeably.

The IDE, or integrated development environment, we used to develop the app was Visual Studio Code. We chose V.S. Code as our IDE because we are familiar with it, it provides a helpful, built-in set of features, and it also connects to GitHub seamlessly. For version control GitHub was used because it makes collaborating on a project easier, as team members can push and pull changes to the code, which makes iterating and collaborating much easier.

Current tables (non-Normalized)

Actors: Primary Key =

Actor_ID (string)

Other Columns =

Actor_Name (string)

Bio (string)

Birthday (Date)

Movies (Array of Movie_ID’s - this is also Foreign Key)

Administrative Columns (for data integrity) =

Created_By (String - corresponding to User - this is Foreign Key)

Date_Created (Date)

Date_Updated (Date)

Movies: Primary_Key =

Movie_ID (string)

Other Columns =

Movie_Name (string * note: not necessarily 1:1, because titles can repeat)

Director (string - would be a Foreign key if normalized…)

Genre (string - would be a Foreign key if normalized…)

Synopsis (string)

Administrative Columns (for data integrity) =

Created_By (String - corresponding to User - this is Foreign Key)

Date_Created (Date)

Date_Updated (Date)

Reviews: Primary_Key =

Review_ID (string)

Other Columns =

Movie_ID (string - foreign key)

Star (int between 1 and 5)

Review_Body (string)

Administrative Columns (for data integrity - would be own table if normalized)

Reviewer_Name (string)

Reviewer_Residence (string)

Users: Primary_Key =

User_ID (string)

Other Columns =

admin (boolean)

email (string -- ALSO A PRIMARY KEY)

first_name (string)

last_name (string)

Created_on (Date)

Updated_on (Date)

Functional Description:

Our IMDb application behaves similarly to the official IMDb site. The database is populated with movies, directors, genres, year of production, synopsis, and the cast. Only administrators can create, update, or delete information in the database (other than movie reviews) and must log in with a special email and password associated with these privileges. Users can access the data in the database but cannot alter or remove any entries. Users can search for their desired movie using the name of the movie and will be able to retrieve all of the information available in the database on said movie.(See Appendix “Search for Movie”) If the movie does not exist in the database an appropriate error message is displayed. Users can also search for their favorite actor and see a complete list of biographical information of said actor or

actress and what movies they are linked to in our database.(See Appendix “Search for Actor/Actress”) There is also an extensive movie review function that allows the user to enter their name, city, a rating of the movie, and text of what their opinion of the film was. (See Appendix “Movie Review”)

Administrators in our database have ultimate control of every table in the database as well as the same functionality available to the users. Administrators can sign in using the upper right log in box with their email and password and gain access to the aforementioned privileges along with all of the functionality of the users.(See Appendix “Search for a Movie” Upper right) They can create or update a movie’s fields instantly and then retrieve said movie to see it has been successfully altered or created.(See Appendix “Create/Add Movie/Actor”) They can also delete a complete movie table in one input. Admins can also create, update, and delete actor/actresses in the same way with one search for their names and a click. If an administrator reads a review a user posted that is inappropriate or unwanted they also have the ability to delete reviews.(See Appendix “Updated Movie results/ Delete a Movie Review”) For all of these functions they must enter the associated unique key for the particular movie, actor, or review and it must be a valid key for it to execute the change. Everything functions as intended and a demonstration of all functionality can be viewed via the screencast found in the appendix.

Testing:

The application was tested hundreds of times during development in order to check for bugs and other possible issues with the various functions and user interface. The four main

functions of the IMDB application are create, read, update, and delete as previously stated, and they are used to manipulate and retrieve data from the documents in the database.

With the retrieve and delete methods, there were two major test cases for each instance (i.e. Movie retrieval, Actor retrieval, and Movie Review retrieval). These two cases are 1) the record exists in the database or 2) the record cannot be found. For example, if a user searches for a movie and the movie exists, retrieve the movie from the database and display the movie information to the user. When a user enters a movie that does not exist in the database, an error message must be displayed to the user informing them that the movie was not found in the database. This can also be seen when an administrator attempts to delete a movie or movie review from the database. If the record exists, delete it and display a message indicating success. However, if the key entered doesn't match a document in the database, then display a failure message to the admin.

One issue that became apparent during testing occurred when a user searched for a movie or actor and then made additional searches; the prior search results were not disappearing from the user's page. To solve this, we created a separate function that is called at the beginning of all retrieval/search functions, which checks the location that will be written to on the user's page and replaces it with the results of the next search. This was important to solve because it prevents the user's page from becoming cluttered with search results.

A second major issue that we discovered during testing was some of the forms (i.e. the form to create a movie) were not clearing their input fields upon submission. For example, an administrator would go to create a new movie, but the information they entered wasn't disappearing from the page upon clicking the 'Create Movie' button. If the administrator

repeatedly clicked the ‘Create Movie’ button, the exact same movie would be added again and again to the database, resulting in duplicates.

A third bug that we encountered involved the drop-down menus for adding movie stars to a movie or adding movies that an actor has starred in to their record. The logic of both drop-down methods is identical, and what kept occuring was clicking one dropdown menu would cause the input fields to display within the wrong part of a user’s page. The bug was resolved by renaming the various input fields after determining the conflicts.

Conclusion:

Our IMDb database was not an easy project in the slightest but it does in fact meet all of the requirements. Our original goal was to import an enormous dataset of already existing movies from an open source site and convert it into a mySQL database. As we began this process it was clear that this would be an enormous undertaking in its entirety. The complications from importing a dataset with over 20,000 movies in it and normalizing such a database would have taken us weeks. It was at this point that changed our goal to something easier to achieve, implement a database from scratch with our own tables and supply it CRUD functionality. This was more within the constraints of our assignment and our ability as computer science students.

Once we had our goal established we began delegating tasks and we each volunteered to do something we were strong at. The overall design of the database (ER diagrams) came into place rather quickly as we were not going to make a very complex dataset to normalize. Each movie corresponds to several attributes and in general there is not a one-to-one relationship

between people and movies. This creates a need for each movie or actor to have their unique key associated with such.

Normalizing this database was much easier because of our decision to lower the bar of using an immense dataset of movies to just creating our own from scratch. This allowed us to create the tables deliberately and methodically rather than writing an algorithm to sort through thousands of entities and attributes on its own. Perhaps in the future we may go on to implement such a database, but for the purposes of this assignment we decided to keep it simple and within our bounds.

Our result after weeks of planning, work, and dedication is a fully developed and efficient version of the real IMDb with complete CRUD functionally for administrators using Express.js, Node.Js, and Firebase. I know our goal was achieved through thorough testing and debugging that was conducted. Hundreds of test cases were used for debugging and several significant bugs were found and resolved as explained earlier. Every create, read, update, and delete function works accordingly using the application layer and the changes they make to the database execute as intended. These functions have been tested thoroughly and manipulate the database accordingly. We would have liked to make an enormous database with an open source dataset with thousands of movies but due to time constraints we were not able to make that a reality. What we did achieve is a streamlined, efficient, and simplistic application with CRUD functionality that is the first step in the direction of our original goal of replicating all of IMDb.

References:

Astor, J. (2017, July 21). Retrieved May 10, 2019, from https://www.youtube.com/watch?v=8YNw8J9Edjk&list=PLJm7_t7JnSjn__VC3sK86o2YlUuwlPKkO

EJS. (2019). Retrieved from https://ejs.co/

Firebase Google. (2019). Retrieved from https://firebase.google.com/

Foundation, Node. (2019). Retrieved from https://nodejs.org/en/

Node.js web application framework. (2019). Retrieved from https://expressjs.com/

Npm. (2019). Retrieved from https://www.npmjs.com/

Otto, M., & Thornton, J. (2019). Bootstrap. Retrieved from https://getbootstrap.com/

Ratings and Reviews for New Movies and TV Shows. (2019). Retrieved from https://www.imdb.com/

Stock Images, Royalty-Free Illustrations, Vectors, & Stock Video Clips - iStock. (2019). Retrieved from https://www.istockphoto.com/

Visual Studio IDE, Code Editor, Azure DevOps, & App Center. (2019). Retrieved from https://visualstudio.microsoft.com/

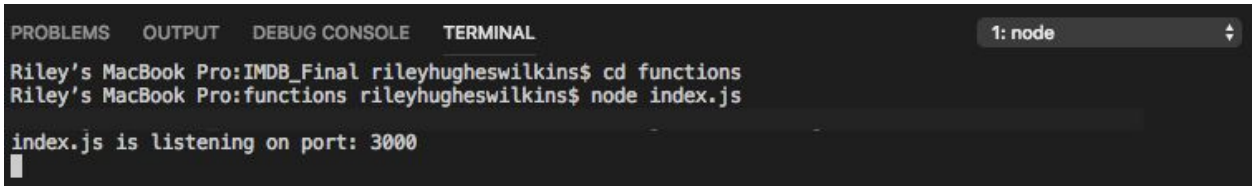
Appendix:

- **Code (too long): See GitHub repository**
- **Carolcusano's invite link to the GitHub repository:**

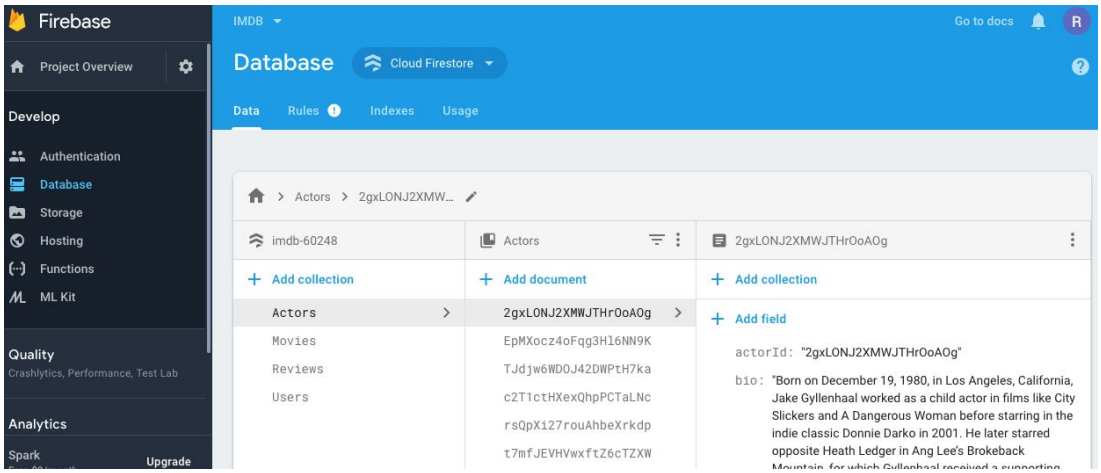
  https://github.com/rwilk19/IMDB/invitations

Screencast: https://www.screencast.com/t/f92eNoA0b

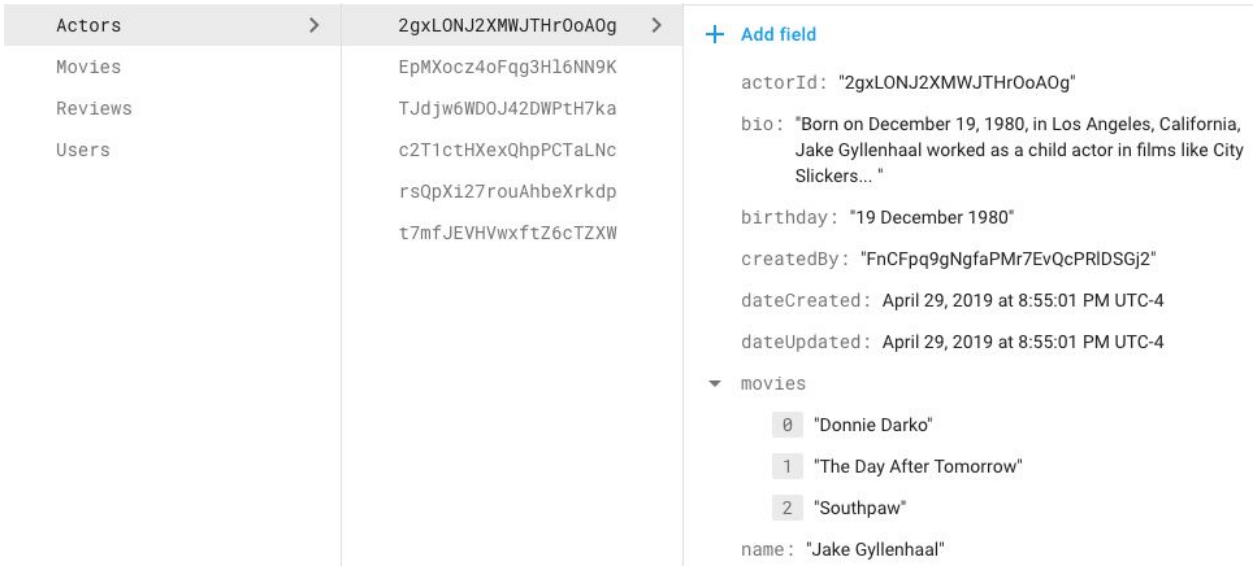
How the IMDB application's server is started:



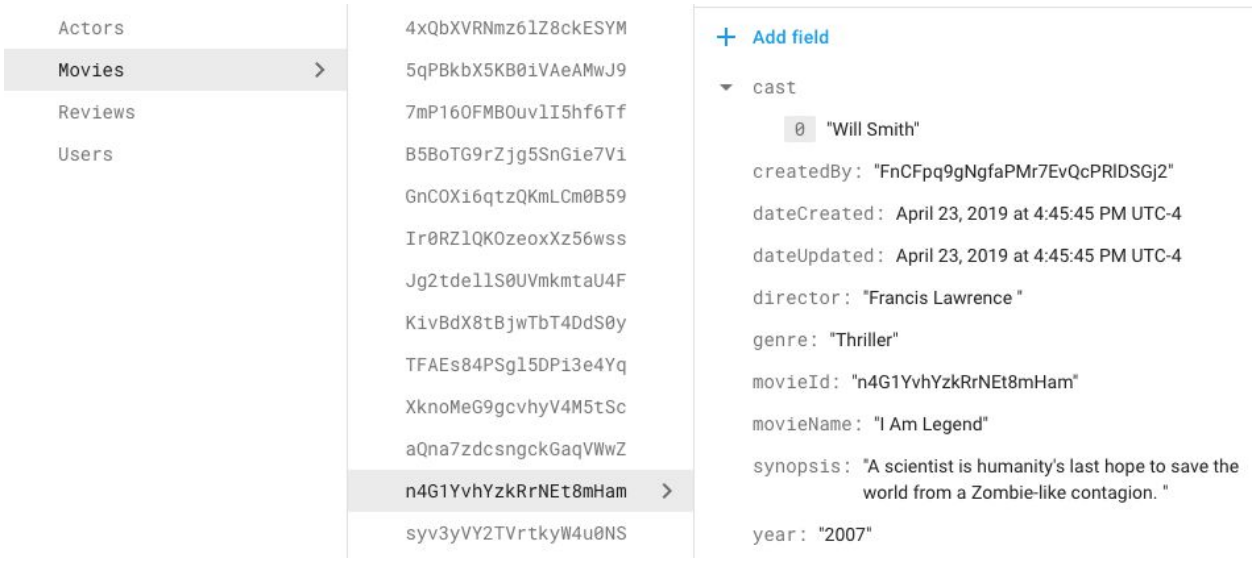
Our Firebase IMDB database GUI:



Actor “Table” (technically a document - Firebase is a noSQL database):

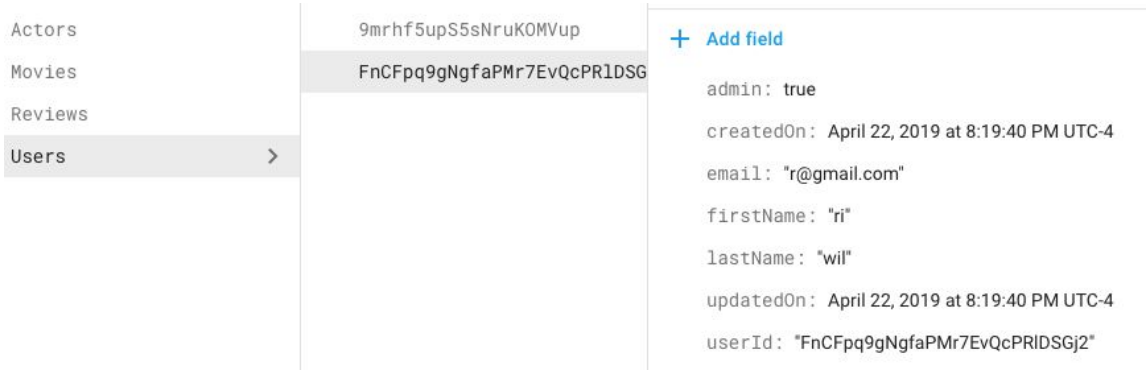


Movies “Table” (technically a document):



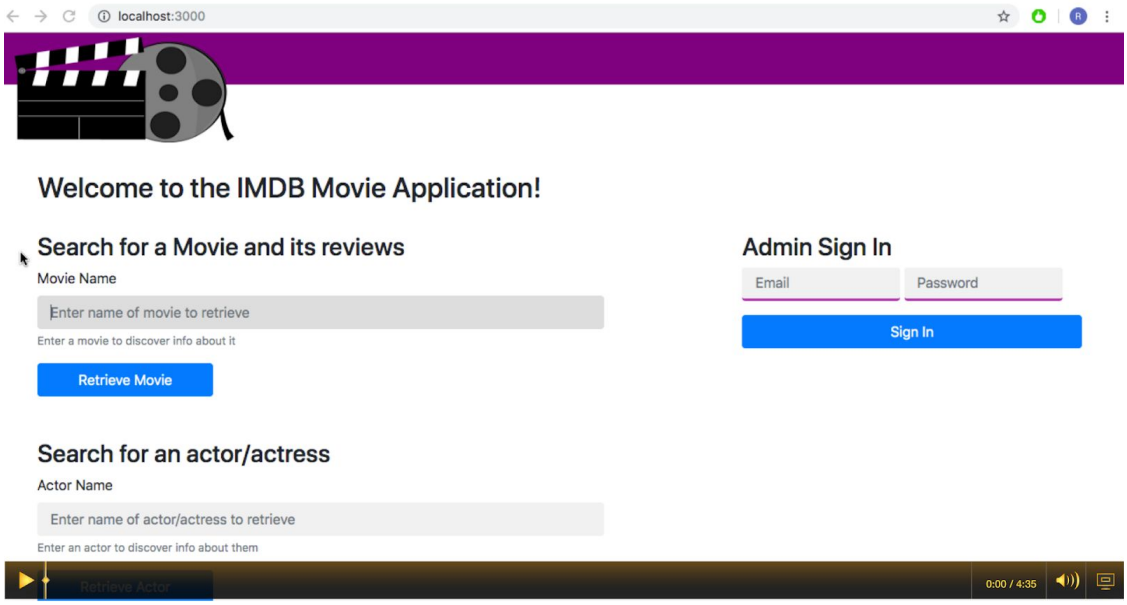
Reviews “Table”:

Users “Table”:

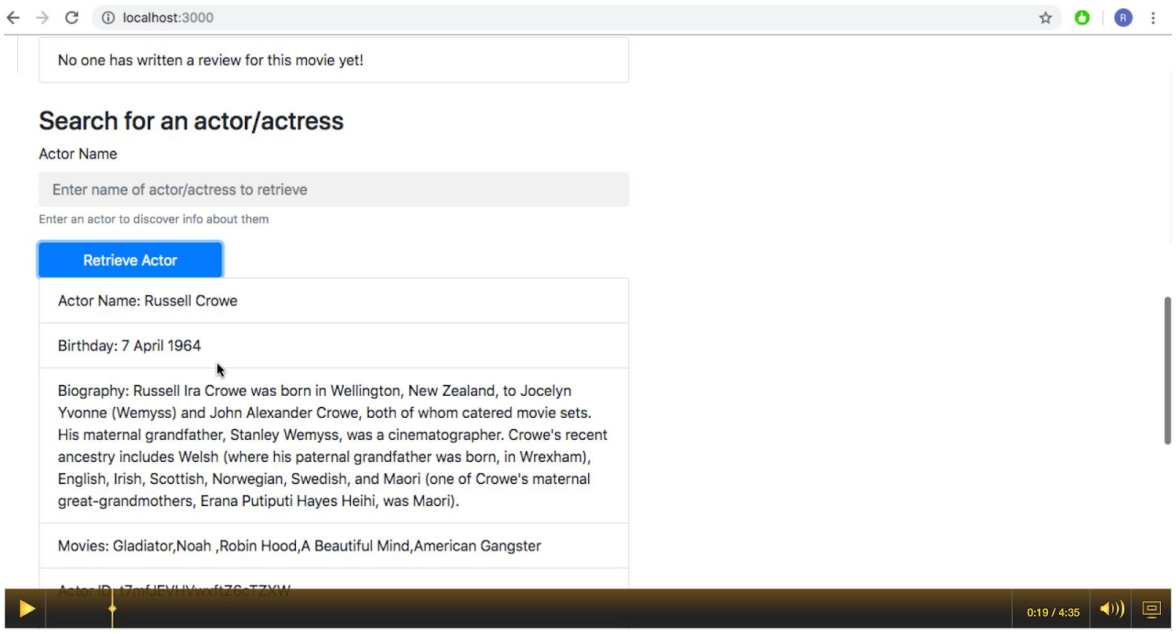


1.6 Application Pages/ Functionality Screenshots

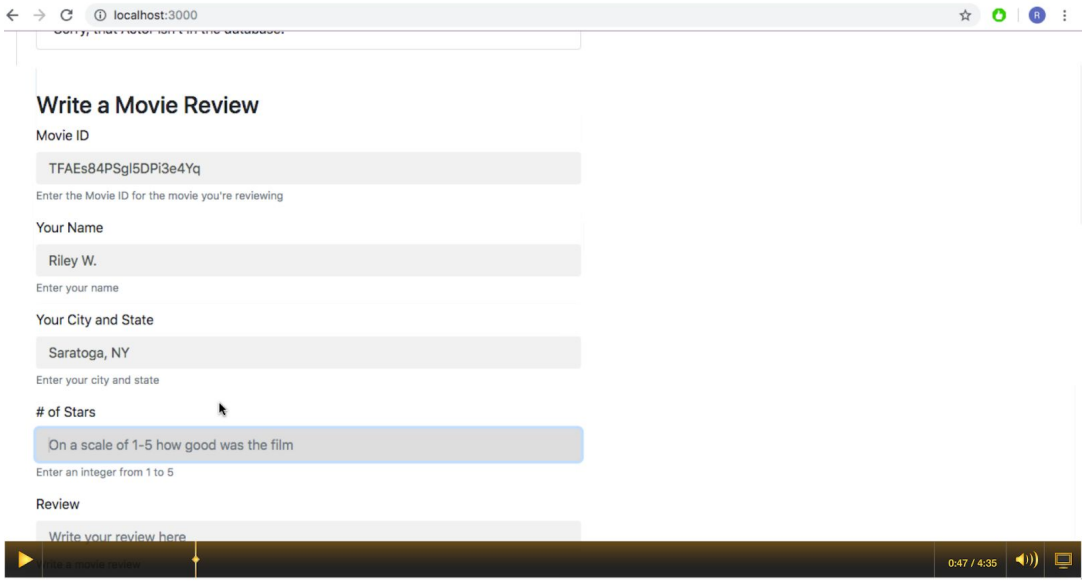
Search for a Movie and its reviews (User)



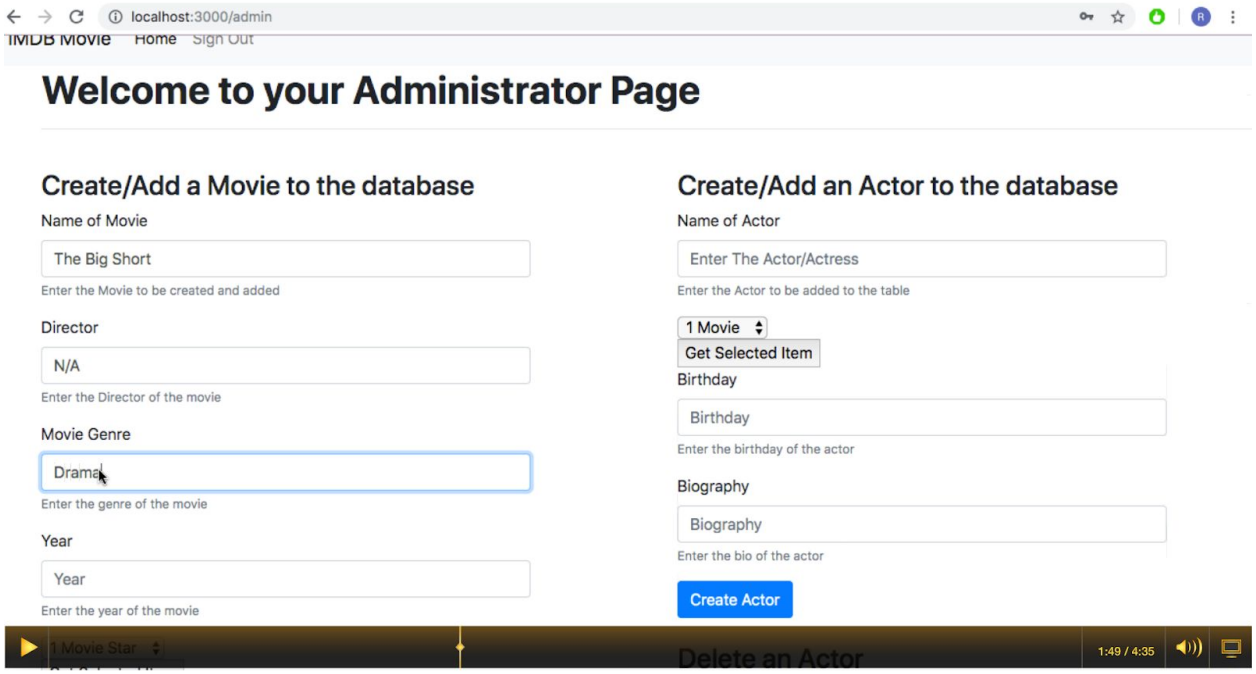
Search for an actor/actress (user)



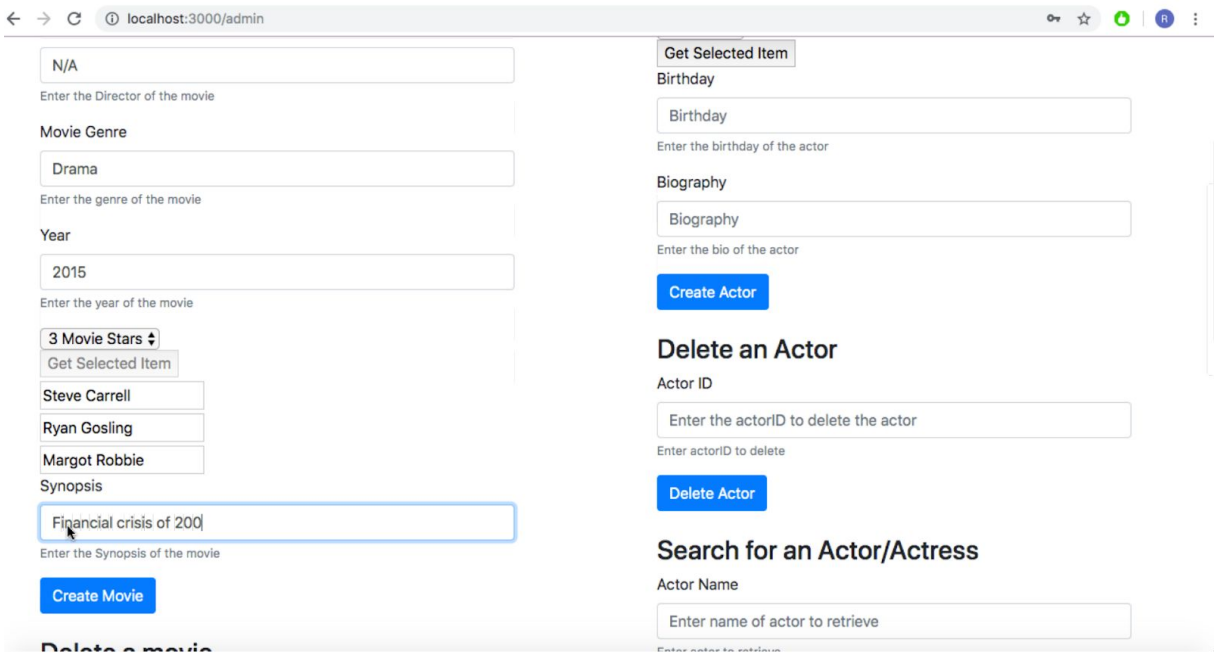
Movie Review (User)



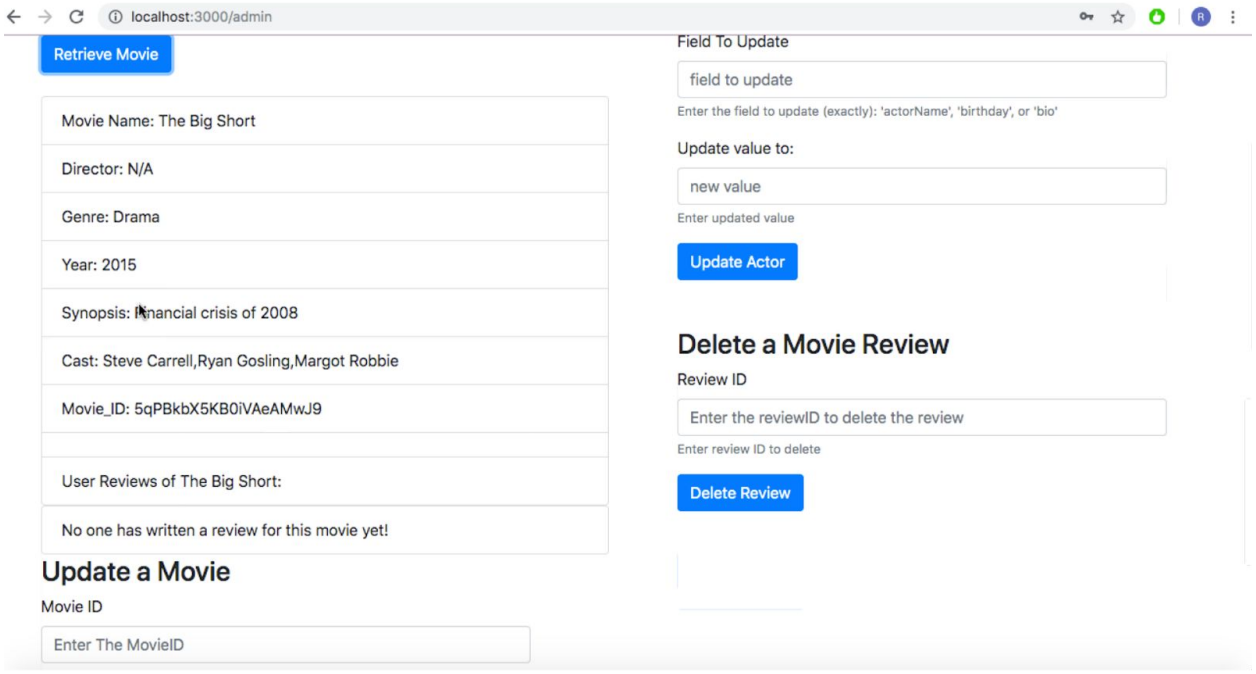
Create/Add Movie/Actor (Administrator)



Create Movie (in progress) (Administrator)



Updated Movie results/ Delete a Movie Review (Administrator)



Update a Movie (Administrator)