Project Deliverable 4 Report

Completed by: Jackson Salyards

1 Introduction

I think that a movie organizer could be a useful tool for someone with a large movie collection, especially a physical collection. I wanted to make something that could easily be used to store and filter a large dataset of movies based on multiple parameters. Anyone with a movie collection could use the program. The program uses a GUI interface for maximum ease of use.

The program uses 4 tables to store information about the movie, cast, and reviews. Reviews are broken down into a main “aggregate” review and then multiple sub reviews. The user can define as many or as few reviews as they see fit. All tables are either directly or indirectly referenced to the main movies table. This makes relating an entry in one table to another much easier.

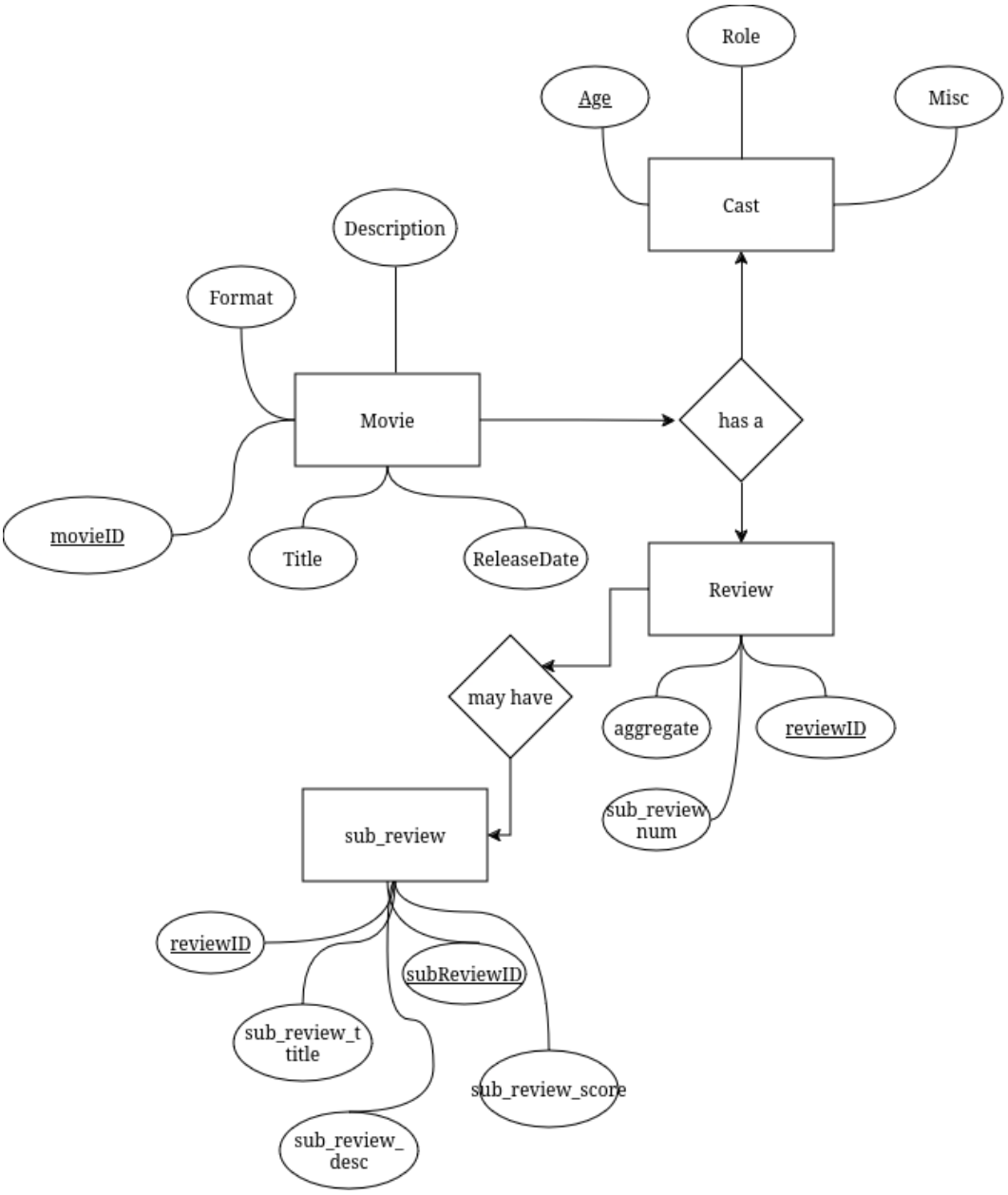
I followed an iterative design process to build this program. First, I got full database functionality then I moved on to improving/adding an GUI. This gave me time to reasearch good GUI libraries and get more familiar with rust and how it works. I would draft an idea of how I wanted a feature to function and then begin implementing it. I focused on making the code as modular as possible to make expansion easy later.

In this report, section two will go over my database design and ER diagram, section three will cover features and application information, section four will cover the conclusion and appendices one and two will cover sql code and my source code.

2 Database Design

This section presents the design milestones for the application’s database.

2.1 Final E/R Diagram



2.2 Final Relational Schema

Movie(movieId INT, title VARCHAR[50], releaseDate YEAR, format VARCHAR[10], description VARCHAR[1000]
Functional Dependencies (FDs): movieId -> title, releaseDate, format, description
Primary Key: movieId
Superkeys: movieId
The left-hand side of the FD is a superkey, so the Movie relation is in 3NF.

Cast(castID INT, moveID INT, age INT, role VARCHAR[50], mis VARCHAR[1000])
FDs: castID -> age, name, role, mis, movieId
Primary Key: castID
Superkeys: castID

The left-hand side of the FD is a superkey, so the CastMembers relation is in 3NF.

review(reviewID INT, movieID INT,aggregate INT,
sub_review_num INT)
FDs: reviewID -> aggregate, sub_review_num, movieId
Primary Key: reviewID
Superkeys: reviewID
The left-hand side of the FD is a superkey, so the Review relation is in 3NF.

Sub_Review (reviewID INT, subreviewID INT, sub_review_title VARCHAR(50), sub_review_score INT, sub_review_desc VARCHAR(1000), FOREIGN KEY (reviewID) REFERENCES Review(reviewID));
FDs: subreviewID -> reviewID, sub_review_title, sub_review_score, sub_review_desc
Primary Key: subreviewID
Superkeys: subreviewID
The left-hand side of the FD is a superkey, so the Sub_Review relation is in 3NF.

2.3 Constraints

In addition to PRIMARY keys, the following constraints are implemented:

Table	CastMembers
Constraint	FOREIGN KEY (movieId) REFERENCES Movie(movieId)
Justification	Links all cast members to one movie via ID so that searching, adding, and deleting cast members is easier.

Table	Review
Constraint	FOREIGN KEY (movieId) REFERENCES Movie(movieId)
Justification	Links reviews (and indirectly sub_reviews) to movies via an ID, so that adding, searching and deleting are easier.

Table	Sub_Review
Constraint	FOREIGN KEY (reviewID) REFERENCES Review(reviewID)
Justification	Links all sub_reviews to a single main review. Makes referencing them easier.

2.4 Database Schema in SQL

The database schema for our relational database is in Appendix I of this document.

3 Application and Database Integration

This project was completed using Rust. I wanted to use Rust for this project because I don't have much experience with it and wanted to learn more about it. I am using several libraries (crates): sqlx, chrono, dotenv, once_cell, slint,

tokio, and mariaDB 11.3. The project is using a GUI built on slint. I wanted to add more features, like editing movies but there were some time constraints, so I had to remove that feature. The only error the application has is that if the user clicks on movies (in the movie column) too fast or clicks it more than ~20 times the application hangs indefinitely. I have no idea what is causing this, but it's probably something to do with moving data into/out of the GUI. There is another UI bug where if the same actor is placed within a movie twice, and then the user searches for that actor the movie will be displayed twice.

Feature:1	Search based on Title
Description	Using the search box at the top of the program the user can enter in strings, that filter the database based on filter selected to the left of the search bar. If the user selects "Movie-Name" and inputs a title the program will return movies containing that string in their title. 
Where?	• gui_loop() -- gui_draw.rs search_by_filters() -- gui_draw.rs • filter_by_title() -- db.rs • parse_movie_list() --lib.rs • to_shared_string() -- lib.rs • App.set_MoiveList() -- gui_draw.rs
SQL Queries	SELECT title, movieId FROM Movie WHERE title LIKE ?

Feature:2	Search based on Release Date
Description	Using the search box at the top of the program the user can enter in strings, that filter the database based on filter selected to the left of the search bar. If the user selects "Release-Date" and inputs a title the program will return movies containing that release date. 

Where?	• gui_loop() -- gui_draw.rs • search_by_filters() -- gui_draw.rs • search_term.parse::<i32>() • filter_by_release() --db.rs • parse_movie_list() -- lib.rs • to_shared_string() -- lib.rs app.set_MovieList() --gui_draw
SQL Queries	SELECT title, movieId FROM Movie WHERE releaseDate = ?

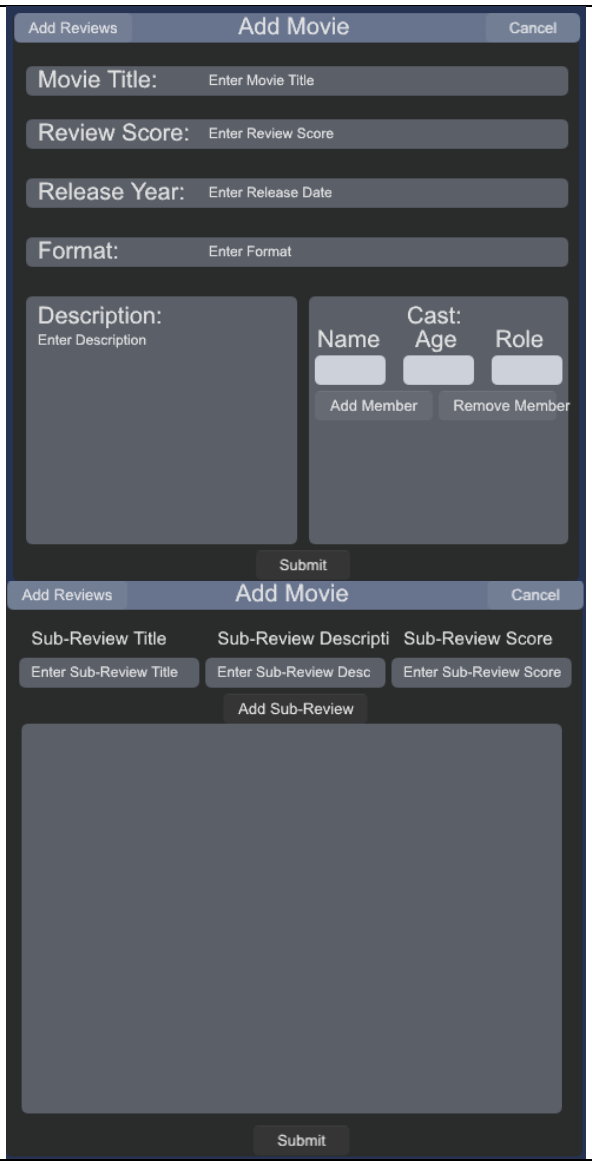
Feature:3	Search based on Review
Description	Using the search box at the top of the program the user can enter in strings, that filter the database based on filter selected to the left of the search bar. If the user selects “Rating” and inputs a title the program will return movies containing that review score. 
Where?	• gui_loop() -- gui_draw.rs • search_by_filters() -- gui_draw.rs • search_term.parse::<i32>() • filter_by_rating() --db.rs • parse_movie_list() -- lib.rs • to_shared_string() -- lib.rs app.set_MovieList() --gui_draw
SQL Queries	SELECT title, movieId FROM Review NATURAL JOIN Movie WHERE aggregate = ?

Feature:4	Search based on Cast member name
Description	Using the search box at the top of the program the user can enter in strings, that filter the database based on filter selected to the left of the search bar. If the user selects “Actor-Name” and inputs a title the program will return movies containing that cast member. 
Where?	• gui_loop() -- gui_draw.rs • search_by_filters() -- gui_draw.rs • filter_by_actor() --db.rs • parse_movie_list() -- lib.rs

	• to_shared_string() -- lib.rs app.set_MovieList() --gui_draw
SQL Queries	SELECT c.movieId, m.title AS title FROM CastMembers c JOIN Movie m ON c.movieId = m.movieId WHERE c.name LIKE ?

Feature:5	Search based on Format
Description	Using the search box at the top of the program the user can enter in strings, that filter the database based on filter selected to the left of the search bar. If the user selects "Format" and inputs a title the program will return movies containing that have that format. 
Where?	• gui_loop() -- gui_draw.rs • search_by_filters() -- gui_draw.rs • filter_by_format() --db.rs • parse_movie_list() -- lib.rs • to_shared_string() -- lib.rs app.set_MovieList() --gui_draw
SQL Queries	SELECT title, movieId FROM Movie WHERE format = ?

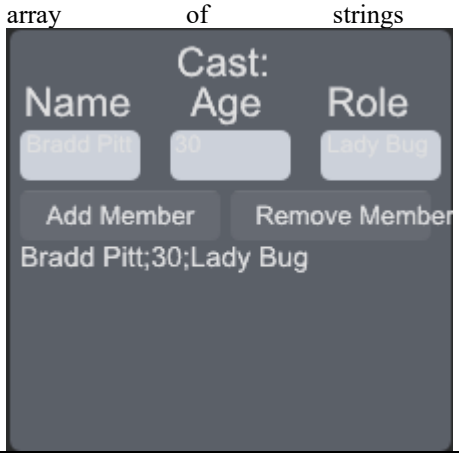
Feature:6	Add Movie Submenu
Description	The submenu contains the required text boxes and buttons to add a movie to the database. From this menu the user is able to add: Title, review aggregate, format, release year, description, cast member name, cast member age, cast member role, sub review title, sub review description and sub review scores. Both cast members and sub reviews can have as many members within them as the user wants.

	
Where?	Gui_draw.rs; lines 388 - 884
SQL Queries	None, pure UI at this stage

Feature:7	Add Movie via “Submit Button”
Description	Once the user has decided they want to add the movie, they can hit the submit button from the add movie sub menu. This triggers several function calls that add the movie and all other details enterd into the add movie sub menu.
Where?	• gui_loop() -- gui_draw.rs • Fill_from_gui() --gui_draw.rs • Add_movie() – gui_events.rs • Get_max _movie_id() – db.rs

	• Add() – db.rs • Get_max_cast_id() – db.rs • Get_max_review_id() – db.rs • Get_max_sub_review_id() – db.rs • Populate_movie_list() -- gui_draw.rs • Parse_result() -- lib.rs • Clean_result() -- lib.rs • app.set_MovieList() -- gui_draw.rs
SQL Queries	INSERT INTO Movie (movieId, title, releaseDate, format, description) VALUES ({}, '{}', {}, '{}', '{}') INSERT INTO CastMembers (castId, movieId, age, name, role, mis) VALUES ({}, {}, '{:?}', '{:?}', '{:?}', '{}') INSERT INTO Review (reviewID, aggregate, sub_review_num, movieId) VALUES ({}, {}, {}, {}) INSERT INTO Sub_Review (reviewID, subreviewID, sub_review_title, sub_review_score, sub_review_desc) VALUES ({}, {}, '{}', '{}', '{}') SELECT * FROM Movie

Feature:8	Add Cast
Description	Adding cast is part of the add movie feature, but the act of creating the array of cast member to be added is more involved. As new cast members are added the code builds an array of cast members and then the GUI displays that. This is done this way because slint lacks an easy way to get an

	array of strings out of the GUI. 
Where?	• Gui_draw.rs lines 736 – 865 • Gui_loop90 – gui_draw.rs • App.set_castListIN() - gui_draw.rs
SQL Queries	none

Feature:9	Add Sub_Reviews
Description	Adding sub reviews is also apart of the adding movies process, but like cast members creating the array of sub reviews is more involed. The user can decide to begin adding sub reviews by hitting the “Add Reviews” button. Then they can add the title, score, and description and hit the “Add Sub-Review" button. These sub reviews are displayed below.

	
Where?	• Gui_draw.rs lines 429 – 555 • Gui_loop() – gui_draw.rs • app.set_SubReviewListIN() - gui_draw.rs
SQL Queries	None

Feature:10	Remove Movie
Description	A user can decide to add a movie by pressing the “Delete” button on the right side of every movie within the “Movies” column on the left. It will tell the user via a terminal prompt if the delete was successful. 

Where?	• Gui_loop() - gui_draw.rs • Delete_data() - gui_events.rs • Filter_by_title() -- db.rs • Remove_by_id() - db.rs • Populate_move_list() - gui_draw.rs • Get_all – gui_events.rs • Get_all_records – db.rs • Parse_result() -- lib.rs • Clean_result() -- lib.rs • app.set_MovieList() -- gui_draw.rs
SQL Queries	SELECT title, movieId FROM Movie WHERE title LIKE ? DELETE FROM CastMembers WHERE movieId = ? SELECT reviewID FROM Review WHERE movieId = ? DELETE FROM Sub_Review WHERE reviewID = ? DELETE FROM Review WHERE movieId = ? DELETE FROM Movie WHERE movieId = ? SELECT * FROM Movie

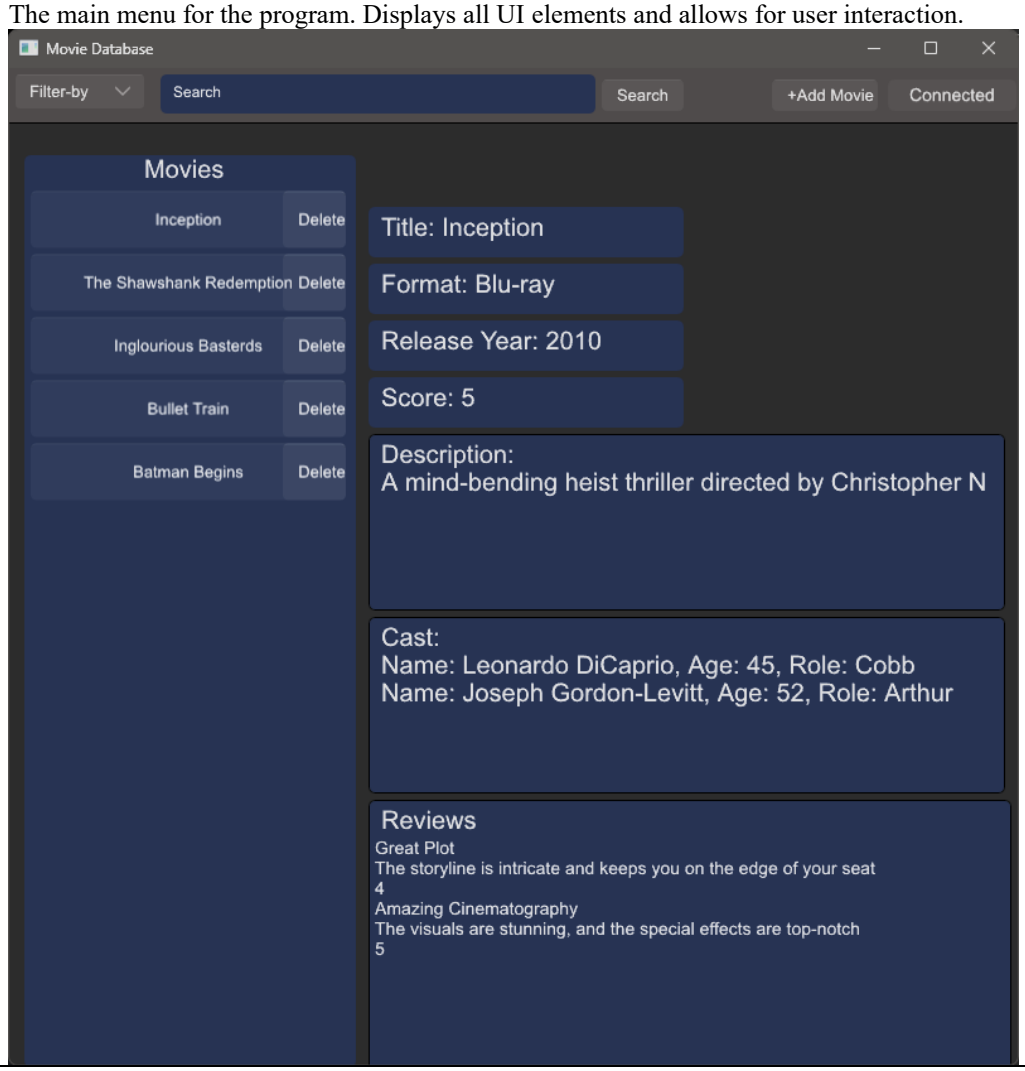
Feature:11	Click Movie for Contents
Description	To get all details about a movie the user must click on the movie from the “Movies” collum on the left of the main window. This will then display the resulting data on the right in several text boxes.

<table>
<tr><td></td><td>It will also warn the user via terminal prompts if any values are blank.
 
</td></tr>
<tr><td>Where?</td><td>• gui_draw lines 141 – 190
• gui_loop() -- gui_draw.rs
• get_movie_details() -- gui_draw.rs
• get_all_movie_details() -- gui_events.rs
• get_movie_details_from_title() – db.rs
• get_reviews_from_movieID() -- db.rs
• get_cast_from_movieID() -- db.rs
app.set_MovieTitleIN() -- gui_draw.rs
• string_to_shared_string() -- lib.rs</td></tr>
<tr><td>SQL Queries</td><td>SELECT * FROM Movie NATURAL JOIN Review WHERE movieId = ?

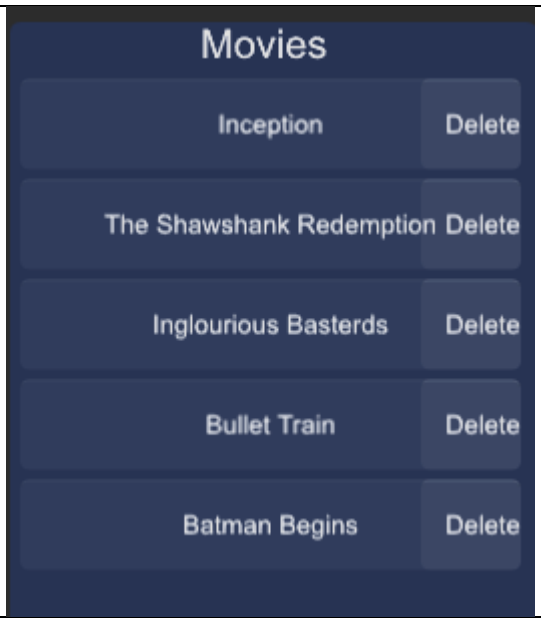
SELECT * FROM Sub_Review WHERE reviewID = ?

SELECT * FROM Movie WHERE title = ?

SELECT c.movieId, m.title AS movie_title, c.name AS actor_name, c.age AS actor_age, c.role AS actor_role
FROM CastMembers c
JOIN Movie m ON m.movieId = c.movieId
WHERE c.movieId = ?</td></tr>
</table>

Feature:12	Main menu
Description	The main menu for the program. Displays all UI elements and allows for user interaction. 
Where?	Gui_draw.rs lines 26 -892
SQL Queries	None

Feature:13	Populate Movie List
Description	A part of many functions in the program. Populates the movie list on the right side of the program window.

	
Where?	• Populate_movie_list() -- gui_draw.rs • Get_all() – gui_events.rs • Get_all_recors ()– db.rs • Parse_results() -- lib.rs • Clean_results() --lib.rs • app.set_SubReviewList() -- gui_draw.rs
SQL Queries	SELECT * FROM Movie

4 Conclusion

I am quite proud of this project. Sure, the code is probably Horrendous, but it was fun learning for me. I was able to implement everything I wanted to expect for editing movies already in the database. This is possible under the current setup but would require some reworking of the UI. Slint was also not the best choice for this type of project. I could never get it to look as good as I wanted it to look and getting data into and out of the gui was quite challenging. I might fork this project and rebuild it as an electron or web app with a rust backend. I think then I could get it looking and working how I want. It was a lot of fun to learn how to implement a database, even if this one was quite basic.

The project only has one major bug. If the user clicks on movies (in the movie column) too fast or clicks it more than ~20 times the application hangs indefinitely. I have no idea what is causing this, but it's probably something to do with moving data into/out of the GUI. I could figure this out using the built in Rust debugger, but I didn't have the time.

This project was a big learning experience for me in many ways. I learned about the language rust, SQL, and project design. In terms of code volume this is the largest single project I have ever completed at over 1800 lines. I enjoyed the project and the class.

5 References

[1] J. Ullman and J. Widom. *A First Course in Database Systems*, 4th Edition, Upper Saddle River, NJ, USA: Prentice Hall, 2007.

[2] "The rust programming language," The Rust Programming Language - The Rust Programming Language, https://doc.rust-lang.org/book/ (accessed Apr. 25, 2024).

[3] Launchbadge, "Launchbadge/SQLX: 🧰 The Rust SQL Toolkit. an async, pure rust SQL crate featuring compile-time checked queries without a DSL. supports postgresql, mysql, and sqlite.," GitHub, https://github.com/launchbadge/sqlx (accessed Apr. 25, 2024).

[4] "Once_cell - crates.io: Rust package registry," Crates.io, https://crates.io/crates/once_cell (accessed Apr. 26, 2024).

[5] "Dotenv - crates.io: Rust package registry," Crates.io, https://crates.io/crates/dotenv (accessed Apr. 26, 2024).

[6] Chrono - Crates.io: Rust package registry, https://crates.io/crates/chrono (accessed Apr. 26, 2024).

[7] "Build reliable network applications without compromising speed.," Tokio, https://tokio.rs/ (accessed Apr. 25, 2024).

[8] S. GmbH, "Declarative GUI for rust," Slint, https://slint.rs/ (accessed Apr. 25, 2024).

[9] "The rust programming language," The Rust Programming Language - The Rust Programming Language, https://doc.rust-lang.org/book/ (accessed Apr. 25, 2024).

6 Appendix I: SQL Schema and INSERT Values

```
CREATE TABLE Movie (
  movieId INT PRIMARY KEY,
  title VARCHAR(500),
  releaseDate INT,
  format VARCHAR(10),
  description VARCHAR(1000)
);

CREATE TABLE CastMembers (
  castID INT PRIMARY KEY,
  age INT,
  name VARCHAR(500),
  role VARCHAR(500),
  mis VARCHAR(1000),
  movieId INT,
  FOREIGN KEY (movieId) REFERENCES Movie(movieId)
);

CREATE TABLE Review (
  reviewID INT PRIMARY KEY,
  aggregate INT,
  sub_review_num INT,
  movieId INT,
  FOREIGN KEY (movieId) REFERENCES Movie(movieId)
);

CREATE TABLE Sub_Review (
  reviewID INT,
  subreviewID INT PRIMARY KEY,
  sub_review_title VARCHAR(50),
  sub_review_score INT,
  sub_review_desc VARCHAR(1000),
  FOREIGN KEY (reviewID) REFERENCES Review(reviewID)
);

INSERT INTO Movie (movieId, title, releaseDate, format, description)
VALUES
  (1, 'Inception', 2010, 'Blu-ray', 'A mind-bending heist thriller directed by Christopher Nolan'),
  (2, 'The Shawshank Redemption', 1994, 'DVD', 'A powerful drama based on Stephen King\'s novella'),
  (3, 'Inglourious Basterds', 2009, 'Blu-ray', 'A Quentin Tarantino film set during World War II'),
  (5, 'Bullet Train', 2022, 'Digital', 'An action thriller film directed by David Leitch');

-- Insert cast members
INSERT INTO CastMembers (castID, age, name, role, mis, movieId)
VALUES
  (1, 45, 'Leonardo DiCaprio', 'Cobb', 'An experienced thief skilled in entering the subconscious', 1),
  (2, 52, 'Joseph Gordon-Levitt', 'Arthur', 'A skilled point man and close friend of Cobb', 1),
  (3, 60, 'Morgan Freeman', 'Ellis Boyd "Red" Redding', 'A longtime inmate at Shawshank State Penitentiary', 2),
  (4, 34, 'Tim Robbins', 'Andy Dufresne', 'A banker wrongly convicted of murder', 2),
  (5, 43, 'Brad Pitt', 'Lt. Aldo Raine', 'Leader of the Basterds, a group of Jewish-American soldiers', 3),
  (6, 36, 'Diane Kruger', 'Bridget von Hammersmark', 'A German film actress and undercover agent', 3),
  (7, 55, 'Christoph Waltz', 'Col. Hans Landa', 'A cunning and ruthless SS officer known as "The Jew Hunter"', 3),
  (8, 58, 'Brad Pitt', 'Ladybug', 'A skilled assassin with a mysterious past', 5),
  (9, 42, 'Zazie Beetz', 'Lemon', 'A deadly and unpredictable assassin', 5),
  (10, 36, 'Michael Shannon', 'Tangerine', 'A seasoned hitman with a personal agenda', 5);
```

```
-- Insert reviews
INSERT INTO Review (reviewID, aggregate, sub_review_num, movieId)
VALUES
  (1, 4.5, 2, 1),
  (2, 4.8, 1, 2),
  (3, 4.2, 3, 3),
  (5, 4.6, 2, 5);

-- Insert sub-reviews
INSERT INTO Sub_Review (reviewID, subreviewID, sub_review_title, sub_review_score, sub_review_desc)
VALUES
(1, 1, 'Great Plot', 4, 'The storyline is intricate and keeps you on the edge of your seat'),
  (1, 2, 'Amazing Cinematography', 5, 'The visuals are stunning, and the special effects are top-notch'),
  (2, 3, 'Emotional Journey', 4.8, 'The emotional depth of the characters is unparalleled'),
  (3, 4, 'Outstanding Performances', 4.5, 'The cast, especially Christoph Waltz, delivered exceptional performances'),
  (3, 5, 'Unique Storytelling', 4, 'Tarantino\'s signature nonlinear storytelling adds a unique twist to the war genre'),
  (3, 6, 'Intense Action Sequences', 4.2, 'The action scenes are gripping and well-executed'),
  (5, 7, 'High-Octane Action', 4.8, 'The film delivers intense and well-choreographed action sequences'),
  (5, 8, 'Star-Studded Cast', 4.4, 'Brad Pitt and the ensemble cast contribute to the movie\'s appeal');
```

7 Appendix II: Application Source Code

Main.rs

```
use cs_317_movie_project::gui_draw;
#[tokio::main]
async fn main() {
    gui_draw::init().await;
}
```

gui_draw.rs

```
use crate::db::{
    filter_by_actor,  filter_by_format,  filter_by_rating,  filter_by_release,
filter_by_title,
};
use crate::gui_events::{
    add_movie,   delete_data,   get_all,   get_all_movie_details,   get_pool,
get_sub_review_list,
};

use crate::record::FromGui;
use crate::{
    actorlist_to_string,            parse_movie_list,            parse_result,
string_to_shared_string, vec_str_to_model,
};
use once_cell::sync::Lazy;
use slint::spawn_local;
use slint::SharedString;
use slint::{self};
use sqlx::MySqlPool;

use std::sync::Mutex;
```

```
// Shared state
static      CAST_LIST:      Lazy<Mutex<Vec<String>>>      =      Lazy::new(||
Mutex::new(Vec::new()));
static   SUB_REVIEW_SCORE_LIST:   Lazy<Mutex<Vec<String>>>   =   Lazy::new(||
Mutex::new(Vec::new()));
static   SUB_REVIEW_DESC_LIST:   Lazy<Mutex<Vec<String>>>   =   Lazy::new(||
Mutex::new(Vec::new()));
static   SUB_REVIEW_TITLE_LIST:   Lazy<Mutex<Vec<String>>>   =   Lazy::new(||
Mutex::new(Vec::new()));

slint::slint! {
    import { Button, ListView, ScrollView, GridBox, Slider, ComboBox, CheckBox,
Switch, StandardTableView, TabWidget} from "std-widgets.slint";
    export component MainGui inherits Window{
        InitButtonVisible: true;
        AllOtherVisible: true;
        ResetVisible: false;
        MovieDetailsVisible: false;
        BasicColor: #1a2646;
        DialogVisible: false;
        NumOfCastMembers: 1;
        AddRevoewVisible: false;
        //size of the window
        width: 800px;
        height: 790px;
        title: "Movie Database";
        in property <[string]> MovieList;
        in property <[string]> SubReviewList;
        in property <color> BasicColor;
        in property <bool> InitButtonVisible;
        in property <bool> AllOtherVisible;
        in property <string> MovieTitleIN;
        in property <string> MovieThumbnailPath;
        in property <string> Format;
        in property <string> Description;
        in property <string> Cast;
        in property <string> Review;
        in property <int> ReleaseDate;
        in property <[string]> CastListIN;
        in property <[string]> SubReviewListIN;
        out property <bool> ResetVisible;
        out property <bool> DialogVisible;
        out property <string> Filter;
        out property <string> SearchTerm;
        out property <string> Event;
        out property <string> MovieTitleOUT;
        out property <int> ReleaseDateOUT;
        out property <string> FormatOUT;
        out property <string> DescriptionOUT;
        out property <string> CastAgeOUT;
        out property <string> CastNameOUT;
        out property <string> CastRoleOUT;
        out property <int> ReviewOUT;
        out property <string> SubReviewScoreOUT;
        out property <string> SubReviewTitleOUT;
        out property <string> SubReviewDescOUT;
        out property <int> NumOfSubReviews;
```

```
        out property <bool> MovieDetailsVisible;
        out property <int> NumOfCastMembers;
        out property <bool> AddRevoewVisible;
        callback eventOccured();
        ComboBox {
            height: 27px;
            width: 102px;
            visible: true;
            enabled: AllOtherVisible;
            model:  ["Filter-by",  "Movie-Name",  "Release-Date",  "Format",
"Rating", "Actor-Name"];
            x: 5px;
            y: 5px;
            selected => {
                root.eventOccured();
                Event = "FilterSelected";
                Filter = self.current-value;
            }
        }
        Rectangle {
            x: 697px;
            y: 9px;
            width: 102px;
            height: 25px;
            background: #4e4a4a;
            visible: true;
            border-radius: 5px;
            Switch {
                y: 0px;
                x: 0px;
                width: 104px;
                height: 24px;
                checked: false;
                text: "Connect";
                z: 99;
                visible: InitButtonVisible;
                toggled => {
                    eventOccured();
                    Event = "InitButtonClicked";
                 }
            }
            Text {
                text: "Connected";
                x: parent.width/6;
                y: parent.height/6;
                width: 104px;
                height: 24px;
                font-size: 14px;
                visible: !InitButtonVisible;
            }
        }
        Button {
            text: "Search";
            x: 470px;
            y: 9px;
            visible: true;
            enabled: AllOtherVisible;
```

```
    height: 25px;
    clicked => {
        eventOccured();
        Event = "SearchButtonClicked";
        ResetVisible = true;
    }
}
Rectangle {
    visible: AllOtherVisible;
    x: 12px;
    y: 69px;
    width: 263px;
    height: 723px;
    border-radius: 5px;
    border-width: 1px;
    background: BasicColor;
    Text {
        text: "Movies";
        x: 95px;
        y: 0px;
        height: 41px;
        font-size: 20px;
    }
    ListView {
        height: 689px;
        x: 5px;
        y: 26px;
        width: 252px;
        for data in MovieList: Rectangle{
            width: 250px; // specify the width of the Rectangle
            height: 50px; // specify the height of the Rectangle
            Button {
                width: 250px; // specify the width of the button
                height: 45px; // specify the height of the button
                text: data;
                clicked => {
                    eventOccured();
                    Event = "MovieSelected";
                    MovieTitleOUT = data;
                    MovieDetailsVisible = true;
                }
            }
            Button{
                width: 50px;
                height: 45px;
                text: "Delete";
                x: 200px;
                clicked => {
                    eventOccured();
                    Event = "DeleteMovieClicked";
                    MovieTitleOUT = data;
                }
            }

        }
    }
}
```

```
Rectangle {
    visible: true;
    width: 345px;
    height: 31px;
    border-radius: 5px;
    border-width: 1px;
    background: BasicColor;
    x: 120px;
    y: 5px;
    TextInput {
        x: 10px;
        y: 7px;
        width: 327px;
        height: 22px;
        text: "Search";
        visible: true;
        enabled: AllOtherVisible;
        edited => {
            eventOccured();
            Event = "SearchTermEntered";
            SearchTerm = self.text;
            }
        }
    }
Rectangle {
    Text {
        text: "Title: "+MovieTitleIN;
        x: 10px;
        y: 5px;
        width: 228px;
        height: 27px;
        font-size: 20px;
    }
    visible: MovieDetailsVisible;
    border-radius: 5px;
    border-width: 1px;
    background: BasicColor;
    width: 250px;
    height: 40px;
    x: 285px;
    y: 110px;
}
Rectangle {
    Text {
        text: "Release Year: "+ReleaseDate;
        x: 10px;
        y: 5px;
        width: 236px;
        height: 32px;
        font-size: 20px;
    }
    visible: MovieDetailsVisible;
    border-radius: 5px;
    border-width: 1px;
    background: BasicColor;
    width: 250px;
    height: 40px;
```

```
    x: 285px;
    y: 200px;
}
Rectangle {
    Text {
        text: "Format: "+Format;
        x: 10px;
        y: 5px;
        width: 234px;
        height: 32px;
        font-size: 20px;
    }
    width: 250px;
    height: 40px;
    border-radius: 5px;
    border-width: 1px;
    background: BasicColor;
    x: 285px;
    y: 155px;
    visible: MovieDetailsVisible;
}
Rectangle {
    Text {
        text:"Description:\n"+ Description;
        x: 10px;
        y: 5px;
        width: 491px;
        height: 139px;
        font-size: 20px;
    }
    border-radius: 5px;
    border-width: 1px;
    border-color: #000;
    visible: MovieDetailsVisible;
    background: BasicColor;
    width: 505px;
    height: 140px;
    x: 285px;
    y: 290px;
}
Rectangle {
    Text {
        text: "Cast:\n"+Cast;
        x: 10px;
        y: 5px;
        width: 487px;
        height: 129px;
        font-size: 20px;
    }
    border-radius: 5px;
    border-width: 1px;
    border-color: #000;
    visible: MovieDetailsVisible;
    background: BasicColor;
    width: 505px;
    height: 140px;
    x: 285px;
```

```
    y: 435px;
}
Rectangle {
    border-radius: 5px;
    border-width: 1px;
    border-color: #000;
    visible: MovieDetailsVisible;
    background: BasicColor;
    x: 285px;
    y: 580px;
    width: 510px;
    height: 215px;
    Text {
        text: "Reviews";
        x: 10px;
        y: 5px;
        height: 25px;
        font-size: 20px;
    }
    ListView {
        enabled: true;
        x: 5px;
        y: 30px;
        width: 496px;
        height: 174px;
        viewport-width: 1000px;
        for data in SubReviewList: Text {
            text: data;
            font-size: 14px;
        }
    }
}
Button {
    height: 25px;
    width: 84px;
    text: "+Add Movie";
    visible: true;
    enabled: AllOtherVisible;
    x: 605px;
    y: 9px;
    clicked => {
        eventOccured();
        Event = "AddMovieClicked";
        DialogVisible = true;
    }
}
Button {
    height: 25px;
    text: "Reset";
    x: 540px;
    y: 9px;
    visible: ResetVisible;
    clicked => {
        eventOccured();
        Event = "ResetButtonClicked";
        if Filter != "Filter-by"{
            ResetVisible = false;
```

```
        }
    }
}
Rectangle {
    x: 285px;
    y: 245px;
    width: 250px;
    height: 40px;
    border-radius: 5px;
    border-width: 1px;
    background: BasicColor;
    visible: MovieDetailsVisible;
    Text {
        text: "Score: " + Review;
        x: 10px;
        y: 5px;
        height: 29px;
        font-size: 20px;

    }
}
Rectangle {
    x: -18px;
    y: -7px;
    width: 829px;
    height: 51px;
    border-color: #000;
    border-radius: 5px;
    border-width: 1px;
    background: #474343;
    z: -99;
}
Dialog {
    visible: DialogVisible;
    Rectangle {
        background: #1a2646;
        border-radius: 15px;
        border-width: 1px;
        height: 500px;
        width: 500px;
        Rectangle {
            x: 9px;
            y: 11px;
            width: 480px;
            height: 480px;
            border-radius: 5px;
            border-width: 1px;
            background: #191a1b;
        }
        Button {
            text: "Cancel";
            x: 410px;
            y: 10px;
            height: 25px;
            width: 80px;
            z:99;
            clicked => {
```

```
        DialogVisible = false;
    }
}
Button {
    text: "Add Reviews";
    x: 10px;
    y: 10px;
    height: 25px;
    width: 100px;
    z:99;
    clicked => {
        eventOccured();
        AddRevoewVisible = true;
    }
}
Rectangle {
    visible: AddRevoewVisible;
    Button {
        text: "Add Sub-Review";
        x: 190px;
        y: 105px;
        height: 25px;
        width: 120px;
        z:99;
        clicked => {
            eventOccured();
            NumOfSubReviews += 1;
            Event = "AddSubReviewClicked";
        }
    }
    Rectangle {
        x: 20px;
        y: 75px;
        height: 25px;
        width: 150px;
        border-radius: 5px;
        border-width: 1px;
        background: #53575f;
        Text {
            text: "Sub-Review Title";
            width: 150px;
            height: 32px;
            x: 10px;
            y: -25px;
            font-size: 15px;
        }
        TextInput {
            x: 10px;
            y: 5px;
            width: 140px;
            height: 25px;
            text: "Enter Sub-Review Title";
            visible: true;
            enabled: true;
            edited => {
                SubReviewTitleOUT = self.text;
            }
```

```
    }

}Rectangle {
    height: 25px;
    width: 150px;
    y: 75px;
    border-radius: 5px;
    border-width: 1px;
    background: #53575f;
    Text {
        text: "Sub-Review Description";
        width: 150px;
        height: 32px;
        x: 10px;
        y: -25px;
        font-size: 15px;
    }
    TextInput {
        x: 10px;
        y: 5px;
        width: 140px;
        height: 25px;
        text: "Enter Sub-Review Desc";
        visible: true;
        enabled: true;
        edited => {
            SubReviewDescOUT = self.text;
        }
    }
}
Rectangle {
    height: 25px;
    width: 150px;
    y: 75px;
    x: 330px;
    border-radius: 5px;
    border-width: 1px;
    background: #53575f;
    Text {
        text: "Sub-Review Score";
        width: 150px;
        height: 32px;
        x: 10px;
        y: -25px;
        font-size: 15px;
    }
    TextInput {
        x: 10px;
        y: 5px;
        width: 140px;
        height: 25px;
        text: "Enter Sub-Review Score";
        visible: true;
        enabled: true;
        edited => {
            SubReviewScoreOUT = self.text;
        }
```

```
        }


    }Rectangle {
        x: 22px;
        y: 130px;
        height: 325px;
        width: 450px;
        border-radius: 5px;
        border-width: 1px;
        background: #53575f;
        ListView {
            for data in SubReviewListIN : Rectangle {
                width: 450px;
                height: 30px;
                border-radius: 5px;
                border-width: 1px;
                Text {
                    text: data;
                    font-size: 14px;
                }
            }

        }

    }
}
Button {
    text: "Submit";
    x: 215px;
    y: 465px;
    height: 25px;
    width: 80px;
    z:99;
    clicked => {
        self.visible = true;
        DialogVisible = false;
        eventOccured();
        Event = "SubmitButtonClicked";
        // Reset the Default values
    }
}
Rectangle {
    x: 20px;
    y: 55px;
    z: 99;
    width: 460px;
    height: 25px;
    border-radius: 5px;
    border-width: 1px;
    background: #53575f;
    visible: !AddRevoewVisible;
    Text {
        text: "Movie Title: ";
        width: 105px;
        height: 32px;
        x: 10px;
```

```
        y: 0px;
        font-size: 20px;
    }
    TextInput {
        x: 155px;
        y: 5px;
        width: 298px;
        height: 25px;
        text: "Enter Movie Title";
        visible: true;
        enabled: true;
        edited => {
            eventOccured();
            Event = "MovieTitleEntered";
            MovieTitleOUT = self.text;
        }
    }
}
Rectangle {
    x: 20px;
    y: 100px;
    z: 99;
    width: 460px;
    height: 25px;
    border-radius: 5px;
    border-width: 1px;
    background: #53575f;
    visible: !AddRevoewVisible;
    Text {
        text: "Review Score: ";
        width: 138px;
        height: 32px;
        x: 10px;
        y: 0px;
        font-size: 20px;
    }
    TextInput {
        x: 155px;
        y: 5px;
        width: 299px;
        height: 25px;
        text: "Enter Review Score";
        visible: true;
        enabled: true;
        edited => {
            eventOccured();
            Event = "ReviewScoreEntered";
            ReviewOUT = self.text.to-float();
        }
    }
}
Rectangle {
    x: 20px;
    y: 150px;
    z: 99;
    width: 460px;
    height: 25px;
```

```
        border-radius: 5px;
        border-width: 1px;
        background: #53575f;
        visible: !AddRevoewVisible;
        Text {
            text: "Release Year: ";
            width: 150px;
            height: 32px;
            x: 10px;
            y: 0px;
            font-size: 20px;
        }
        TextInput {
            x: 155px;
            y: 5px;
            width: 301px;
            height: 25px;
            text: "Enter Release Date";
            visible: true;
            enabled: true;
            edited => {
                eventOccured();
                Event = "ReleaseDateEntered";
                ReleaseDateOUT = self.text.to-float();
            }
        }
    }
    Rectangle {
        x: 20px;
        y: 200px;
        z: 99;
        width: 460px;
        height: 25px;
        border-radius: 5px;
        border-width: 1px;
        background: #53575f;
        visible: !AddRevoewVisible;
        Text {
            text: "Format: ";
            width: 150px;
            height: 32px;
            x: 10px;
            y: 0px;
            font-size: 20px;
        }
        TextInput {
            x: 155px;
            y: 5px;
            width: 293px;
            height: 25px;
            text: "Enter Format";
            visible: true;
            enabled: true;
            edited => {
                eventOccured();
                Event = "FormatEntered";
                FormatOUT = self.text;
```

```
        }
    }
}
Rectangle {
    x: 20px;
    y: 250px;
    z: 99;
    width: 230px;
    height: 210px;
    border-radius: 5px;
    border-width: 1px;
    background: #53575f;
    visible: !AddRevoewVisible;
    Text {
        text: "Description: ";
        width: 150px;
        height: 200px;
        x: 10px;
        y: 5px;
        font-size: 20px;
    }
    TextInput {
        x: 10px;
        y: 30px;
        width: 217px;
        height: 175px;
        text: "Enter Description";
        visible: true;
        enabled: true;
        edited => {
            eventOccured();
            Event = "DescriptionEntered";
            DescriptionOUT = self.text;
        }
    }
}
Rectangle {
    x: 260px;
    y: 250px;
    z: 99;
    width: 220px;
    height: 210px;
    border-radius: 5px;
    border-width: 1px;
    background: #53575f;
    visible: !AddRevoewVisible;
    Button {
        text: "Add Member";
        x: 5px;
        y: 80px;
        height: 25px;
        width: 100px;
        z:99;
        clicked => {
            NumOfCastMembers += 1;
            eventOccured();
            Event = "AddMemberClicked";
```

```
    }
}
Button {
    text: "Remove Member";
    x: 110px;
    y: 80px;
    height: 25px;
    width: 100px;
    z:99;
    clicked => {
        NumOfCastMembers -= 1;
        eventOccured();
        Event = "RemoveMemberClicked";
    }
}
Text {
    text: "Cast: ";
    width: 69px;
    height: 32px;
    x: 83px;
    y: 5px;
    font-size: 20px;
    z:99;
}
Rectangle {
    background: #c8cdd6;
    border-radius: 5px;
    x: 5px;
    y: 50px;
    width: 60px;
    height: 25px;
    z:90;
    Text {
        text: "Name";
        x: 2px;
        y: -25px;
        font-size: 20px;
    }
    TextInput {
        x: 3px;
        y: 0px;
        width: 53px;
        height: 25px;
        z:90;
        edited => {
            CastNameOUT = self.text;
        }
    }

}
Rectangle {
    border-radius: 5px;
    background: #c8cdd6;
    x: 80px;
    y: 50px;
    width: 60px;
    height: 25px;
```

```
            z:90;
            Text {
                text: "Age";
                x: 10px;
                y: -25px;
                font-size: 20px;
            }
            TextInput {
                width: 56px;
                edited => {
                    CastAgeOUT = self.text;
                }

            }

        }
        Rectangle {
            y: 50px;
            x: 155px;
            width: 60px;
            height: 25px;
            z:90;
            background: #c8cdd6;
            border-radius: 5px;
            Text {
                text: "Role";
                x: 3px;
                y: -25px;
                font-size: 20px;
            }
            TextInput {
                width: 54px;
                edited => {
                    CastRoleOUT = self.text;
                }

            }

        }
        ListView {
            x: 5px;
            y: 105px;
            width: 210px;
            height: 100px;
            viewport-width: 1000px;
            for data in CastListIN: Text {
                text: data;
                font-size: 14px;
            }
        }
    }
    Rectangle {
        x: 10px;
        y: 10px;
        width: 480px;
        height: 25px;
        border-radius: 5px;
```

```
                    border-width: 1px;
                    background: #5f6a81;
                }
                Text {
                    text: "Add Movie";
                    width: 500px;
                    height: 32px;
                    x: 200px;
                    y: 10px;
                    font-size: 20px;
                }
            }
        }
    }
}

pub async fn init() {
    let app = MainGui::new().unwrap();
    let pool = get_pool().await;
    if pool.is_none() {
        //Exit on no pool found
        print!("Error: No pool found\n Exiting...");
        return;
    } else {
        gui_loop(app, pool.unwrap()).await;
    }
}
async fn gui_loop(app: MainGui, pool: MySqlPool) {
    let weak_app = app.as_weak();
    let weak_pool = pool.clone();
    app.on_eventOccured(move || {
        let app = weak_app.upgrade().unwrap();
        let pool = weak_pool.clone();
        let _ = spawn_local(async move {
            match app.get_Event().as_str() {
                "InitButtonClicked" => init_button_clicked(app, pool).await,
                "SearchButtonClicked" => {
                    let filter = &app.get_Filter();
                    let search_term = app.get_SearchTerm();
                    search_by_filters(filter.to_string(),
search_term.to_string(), app, pool).await;
                }
                "MovieSelected" => {
                    print!("Movie selected");
                    let movie_title = app.get_MovieTitleOUT();
                    get_movie_details(app, movie_title, pool).await;
                }
                "ResetButtonClicked" => {
                    let       movie_list       =       filter_by_title(&pool,
"".to_string()).await;
                    let model = parse_movie_list(movie_list);
                    app.set_MovieList(model);
                }
                "SubmitButtonClicked" => {
                    //Get movie details from GUI
                    let movie_title = app.get_MovieTitleOUT();
                    let release_date = app.get_ReleaseDateOUT();
```

```
                    let format = app.get_FormatOUT();
                    let description = app.get_DescriptionOUT();
                    let review_score = app.get_ReviewOUT();
                    let mut cast_list = CAST_LIST.lock().unwrap();
                    let           mut           sub_review_title           =
SUB_REVIEW_TITLE_LIST.lock().unwrap();
                    let           mut            sub_review_desc           =
SUB_REVIEW_DESC_LIST.lock().unwrap();
                    let           mut           sub_review_score           =
SUB_REVIEW_SCORE_LIST.lock().unwrap();
                    //fill in the struct that is sent to database
                    let from_gui = fill_from_gui(
                        movie_title.to_string(),
                        release_date.to_string().parse().unwrap(),
                        format.to_string(),
                        description.to_string(),
                        cast_list.clone(),
                        review_score,
                        sub_review_title.clone().len() as i32,
                        sub_review_title.clone(),
                        sub_review_desc.clone(),
                        sub_review_score.clone(),
                    );
                    //add movie to db
                    add_movie(from_gui, &pool).await;
                    populate_movie_list(app, pool).await;
                    cast_list.clear();
                    sub_review_title.clear();
                    sub_review_desc.clear();
                    sub_review_score.clear();
                }
                "AddMemberClicked" => {
                    let mut cast_list = CAST_LIST.lock().unwrap();
                    let cast_name = app.get_CastNameOUT();
                    let cast_age = app.get_CastAgeOUT();
                    let cast_role = app.get_CastRoleOUT();
                    // Push to the cast_list
                    cast_list.push(format!(
                        "{};{};{}",
                        cast_name.to_string(),
                        cast_age.to_string(),
                        cast_role.to_string()
                    ));
                    app.set_CastListIN(vec_str_to_model(cast_list.clone()));
                }
                "RemoveMemberClicked" => {
                    let mut cast_list = CAST_LIST.lock().unwrap();
                    cast_list.pop();
                    app.set_CastListIN(vec_str_to_model(cast_list.clone()));
                }
                "DeleteMovieClicked" => {
                    let movie_title = app.get_MovieTitleOUT().to_string();
                    delete_data(movie_title, &pool).await;
                    populate_movie_list(app, pool).await;
                }
                "AddSubReviewClicked" => {
```

```
                    let            mut            sub_review_title            =
SUB_REVIEW_TITLE_LIST.lock().unwrap();
                    let            mut             sub_review_desc            =
SUB_REVIEW_DESC_LIST.lock().unwrap();
                    let            mut            sub_review_score            =
SUB_REVIEW_SCORE_LIST.lock().unwrap();

sub_review_desc.push(app.get_SubReviewDescOUT().to_string());

sub_review_title.push(app.get_SubReviewTitleOUT().to_string());

sub_review_score.push(app.get_SubReviewScoreOUT().to_string());
                    let mut sub_review_list = vec![];
                    for i in 0..sub_review_title.len() {
                        sub_review_list.push(format!(
                            "Title: {} Desc: {} Score: {}",
                            sub_review_title[i],           sub_review_desc[i],
sub_review_score[i]
                        ));
                    }

app.set_SubReviewListIN(vec_str_to_model(sub_review_list));
                }
                _ => {}
            }
        });
    });
    app.run().unwrap();
}

async fn init_button_clicked(app: MainGui, pool: MySqlPool) {
    app.set_InitButtonVisible(false);
    app.set_AllOtherVisible(true);
    populate_movie_list(app, pool).await;
}

async fn search_by_filters(filter: String, search_term: String, app: MainGui,
pool: MySqlPool) {
    match filter.as_str() {
        "Movie-Name" => {
            let movie_list = filter_by_title(&pool, search_term).await;
            let model = parse_movie_list(movie_list);
            app.set_MovieList(model);
        }
        "Release-Date" => {
            let  search_term_int  =  search_term.parse::<i32>().unwrap();  //
Convert search_term to i32
            let movie_list = filter_by_release(&pool, search_term_int).await;
            let model = parse_movie_list(movie_list);
            app.set_MovieList(model);
        }
        "Format" => {
            let movie_list = filter_by_format(&pool, search_term).await;
            let model = parse_movie_list(movie_list);
            app.set_MovieList(model);
        }
        "Rating" => {
```

```
            let  search_term_int  =  search_term.parse::<i32>().unwrap();  //
Convert search_term to i32
            let movie_list = filter_by_rating(&pool, search_term_int).await;
            let model = parse_movie_list(movie_list);
            app.set_MovieList(model);
        }
        "Actor-Name" => {
            let search_term = search_term.to_string();
            println!("Actor-Name: {}", search_term);
            let movie_list = filter_by_actor(&pool, search_term).await;
            let model = parse_movie_list(movie_list);
            app.set_MovieList(model);
        }
        _ => {
            println!("Invalid filter, {}", filter);
        }
    }
}

pub async fn get_movie_details(app: MainGui, movie_title: SharedString, pool:
MySqlPool) {
    println!("Getting Details");
    //get movie detai
    let result = get_all_movie_details(&pool, movie_title.to_string()).await;
    //display movie details
    if !result.MovieData.is_empty() {
        app.set_MovieTitleIN(string_to_shared_string(
            result.MovieData[0].title.clone().unwrap(),
        ));
        app.set_Format(string_to_shared_string(
            result.MovieData[0].format.clone().unwrap(),
        ));
        app.set_Description(string_to_shared_string(
            result.MovieData[0].description.clone().unwrap(),
        ));
        if !result.ActorData.is_empty() {
            app.set_Cast(string_to_shared_string(actorlist_to_string(
                result.ActorData.clone(),
            )));
        } else {
            println!("Warning: ActorData is empty");
        }
        app.set_ReleaseDate(result.MovieData[0].releaseDate.unwrap());
        println!("{}", actorlist_to_string(result.ActorData.clone()));
        if !result.ReviewData.is_empty() {
            if let Some(aggregate) = &result.ReviewData[0].aggregate {

app.set_Review(string_to_shared_string(aggregate.to_string()));
                populate_sub_review_list(app,                              pool,
result.ReviewData[0].reviewID.unwrap()).await;
            } else {
                println!("Warning:    aggregate    field    is    missing    in
ReviewData[0]");
                app.set_Review(string_to_shared_string("0".to_string()));
            }
        } else {
            println!("Warning: ReviewData is empty");
```

```
        }
    } else {
        println!("Warning: MovieData is empty");
    }
}

async fn populate_movie_list(app: MainGui, pool: MySqlPool) {
    let result = get_all(&pool).await;
    let model = parse_result(result);
    app.set_MovieList(model);
}

async fn populate_sub_review_list(app: MainGui, pool: MySqlPool, review_id:
i32) {
    let result = get_sub_review_list(&pool, review_id).await;
    let model = parse_result(result);
    print!("SubReviewList: {:?}", model);
    app.set_SubReviewList(model);
}

fn fill_from_gui(
    movie_title: String,
    release_date: i32,
    format: String,
    description: String,
    cast: Vec<String>,
    aggregate: i32,
    sub_review_num: i32,
    sub_review_title: Vec<String>,
    sub_review_desc: Vec<String>,
    sub_review_score: Vec<String>,
) -> FromGui {
    //fill in the struct

    //brake cast array into elements
    let mut actor_name: Vec<String> = Vec::new();
    let mut actor_age: Vec<i32> = Vec::new();
    let mut actor_role: Vec<String> = Vec::new();

    for i in 0..cast.len() {
        let cast_member = cast.get(i).unwrap();
        let cast_member_split: Vec<&str> = cast_member.split(";").collect();

        actor_name.push(cast_member_split[0].to_string());

        let actorage: Result<i32, _> = cast_member_split[1].trim().parse();

        if let Ok(age) = actorage {
            print!("Actor Age: {}", age);
            actor_age.push(age);
        } else {
            println!("Invalid age for actor: {}", cast_member_split[1]);
            actor_age.push(0);
        }

        actor_role.push(cast_member_split[2].to_string());
    }
```

```
    //Check to make sure sub_review_title, sub_review_desc and sub_review_score
all have sub_review_num elements
    if sub_review_title.len() != sub_review_num as usize
        || sub_review_desc.len() != sub_review_num as usize
        || sub_review_score.len() != sub_review_num as usize
    {
        println!("Error: Sub-review arrays do not have the same number of
elements... Omitting sub-reviews");
        return FromGui {
            title: movie_title,
            actor_name: actor_name,
            actor_age: actor_age,
            actor_role: actor_role,
            aggregate: aggregate,
            description: description,
            format: format,
            releaseDate: release_date,
            sub_review_num: 0,
            sub_review_score: Vec::new(),
            sub_review_title: Vec::new(),
            sub_review_desc: Vec::new(),
        };
    }

    //parse sub_review_score to i32
    let mut sub_review_score_int: Vec<i32> = Vec::new();
    for i in 0..sub_review_score.len() {
        let score: Result<i32, _> = sub_review_score[i].trim().parse();
        if let Ok(s) = score {
            sub_review_score_int.push(s);
        } else {
            println!(
                "Error: Invalid score for sub-review: {}",
                sub_review_score[i]
            );
            sub_review_score_int.push(0);
        }
    }

    FromGui {
        title: movie_title,
        actor_name: actor_name,
        actor_age: actor_age,
        actor_role: actor_role,
        aggregate: aggregate,
        description: description,
        format: format,
        releaseDate: release_date,
        sub_review_num: sub_review_num,
        sub_review_score: sub_review_score_int,
        sub_review_title: sub_review_title,
        sub_review_desc: sub_review_desc,
    }
}
```

Gui_events.rs

```
use std::string;

use sqlx::MySqlPool;

use crate::db::establish_connection;
use crate::db::filter_by_title;
use crate::db::get_all as get_all_records;
use crate::db::get_cast_from_movieID;
use crate::db::get_max_movie_id;
use crate::db::get_movie_details_from_title;
use crate::db::get_reviews_from_movieID;
use crate::db::get_sub_reviews_from_reviewID;
use crate::db::remove_movie_by_id;
use crate::record;
use crate::record::FromGui;

#[derive(Clone)]

pub struct ToGui {
    pub MovieData: Vec<record::Record>,
    pub ActorData: Vec<record::CastMovieRecord>,
    pub ReviewData: Vec<record::Review>,
    pub MicroReviewData: Vec<record::MicroReview>,
    pub MovieList: Vec<record::MovieList>,
    pub MovieId: Vec<record::MovieId>,
    pub SubReview: Vec<record::SubReview>,
    pub result: Vec<String>,
    pub pool: Option<MySqlPool>,
}

pub async fn get_pool() -> Option<MySqlPool> {
    let pool = establish_connection().await;
    match pool {
        Ok(pool) => Some(pool),
        Err(_) => None,
    }
}

pub async fn handle_init() -> ToGui {
    let mut result = ToGui {
        result: Vec::new(),
        pool: None,
        MovieData: Vec::new(),
        ActorData: Vec::new(),
        ReviewData: Vec::new(),
        MovieList: Vec::new(),
        MovieId: Vec::new(),
        MicroReviewData: Vec::new(),
        SubReview: Vec::new(),
    };

    match establish_connection().await {
        Ok(pool) => {
            result.pool = Some(pool);
            result.result.push("Connection established".to_string());
```

```
        }
        Err(e) => {
            result.result.push(format!("Error: {}", e));
        }
    }
    result
}
pub async fn delete_data(title: String, pool: &MySqlPool) {
    let title_id = filter_by_title(pool, title).await;
    // handle error and get i32 from movieID
    match title_id {
        Ok(movie_id) => {
            for i in 0..movie_id.len() {
                let movie_id = movie_id[i].movieId.unwrap();
                match remove_movie_by_id(&pool, movie_id).await {
                    Ok(_) => {
                        println!("Movie deleted successfully");
                    }
                    Err(e) => {
                        println!("Error: {}", e);
                    }
                }
            }
        }
        Err(e) => {
            println!("Error: {}", e);
        }
    }
}

pub async fn add_movie(movie: FromGui, pool: &MySqlPool) {
    //Generate a new movieId
    let movieId = get_max_movie_id(pool).await.unwrap().movieId.unwrap() + 1;
    let title = movie.title;
    let releaseDate = movie.releaseDate;
    let format = movie.format;
    let description = movie.description;
    let query = format!("INSERT INTO Movie (movieId, title, releaseDate, format,
description) VALUES ({}, '{}', {}, '{}', '{}')", movieId, title, releaseDate,
format, description);
    match crate::db::add(query, pool).await {
        Ok(_) => {
            println!("Movie added successfully");
            //Add Cast Members
            let mut castId = 0;
            for i in 0..movie.actor_name.len() {
                castId = crate::db::get_max_cast_id(pool)
                    .await
                    .unwrap()
                    .castId
                    .unwrap()
                    + 1;
                let actor_name = movie.actor_name.get(i).unwrap();
                let actor_age = movie.actor_age.get(i).unwrap();
                let actor_role = movie.actor_role.get(i).unwrap();
```

```
                let query = format!("INSERT INTO CastMembers (castId, movieId,
age, name, role, mis) VALUES ({}, {}, '{:?}', '{:?}', '{:?}', '{}')", castId,
movieId, actor_age, actor_name, actor_role, "".to_string());
                match crate::db::add(query, pool).await {
                    Ok(_) => {
                        println!("Cast Member added successfully");
                    }
                    Err(e) => {
                        println!("Error: {}", e);
                    }
                }
            }
            //Add Review
            let reviewID = crate::db::get_max_review_id(pool).await.unwrap() +
1;
            let query = format!(
        "INSERT INTO Review (reviewID, aggregate, sub_review_num, movieId)
VALUES ({}, {}, {}, {})",
        reviewID, movie.aggregate, movie.sub_review_num, movieId
    );
            match crate::db::add(query, pool).await {
                Ok(_) => {
                    println!("Review added successfully");
                }
                Err(e) => {
                    println!("Error: {}", e);
                }
            }
            //Add Sub Reviews
            if movie.sub_review_num > 0 {
                let              mut              sub_review_id              =
crate::db::get_max_sub_review_id(pool).await.unwrap() + 1;
                for i in 0..movie.sub_review_num {
                    let query = format!(
                "INSERT     INTO     Sub_Review     (reviewID,     subreviewID,
sub_review_title, sub_review_score, sub_review_desc) VALUES ({}, {}, '{}',
'{}', '{}')",
                reviewID,   sub_review_id,   movie.sub_review_title.get(i   as
usize).unwrap(),     movie.sub_review_score.get(i     as     usize).unwrap(),
movie.sub_review_desc.get(i as usize).unwrap()
            );
                    sub_review_id += 1;
                    match crate::db::add(query, pool).await {
                        Ok(_) => {
                            println!("Sub Review added successfully");
                        }
                        Err(e) => {
                            println!("Error: {}", e);
                        }
                    }
                }
            }
        }
        Err(e) => {
            println!("Error: {}", e);
        }
    }
```

```
}

pub async fn get_all_movie_details(pool: &MySqlPool, movie_title: String) ->
ToGui {
    let mut result = ToGui {
        result: Vec::new(),
        pool: None,
        MovieData: Vec::new(),
        ActorData: Vec::new(),
        ReviewData: Vec::new(),
        MovieList: Vec::new(),
        MovieId: Vec::new(),
        MicroReviewData: Vec::new(),
        SubReview: Vec::new(),
    };
    match get_movie_details_from_title(pool, movie_title).await {
        Ok(records) => {
            result.MovieData = records;
        }
        Err(e) => {
            result.result.push(format!("Error: {}", e));
        }
    }
    match                                          get_reviews_from_movieID(pool,
result.MovieData.get(0).unwrap().movieId.unwrap()).await {
        Ok(records) => {
            result.ReviewData = records;
        }
        Err(e) => {
            result.result.push(format!("Error: {}", e));
        }
    }
    match                                             get_cast_from_movieID(pool,
result.MovieData.get(0).unwrap().movieId.unwrap()).await {
        Ok(records) => {
            result.ActorData = records;
        }
        Err(e) => {
            result.result.push(format!("Error: {}", e));
        }
    }
    result
}

pub async fn get_all(pool: &MySqlPool) -> ToGui {
    let mut result = ToGui {
        result: Vec::new(),
        pool: None,
        MovieData: Vec::new(),
        ActorData: Vec::new(),
        ReviewData: Vec::new(),
        MovieList: Vec::new(),
        MovieId: Vec::new(),
        MicroReviewData: Vec::new(),
        SubReview: Vec::new(),
    };
    match get_all_records(pool).await {
```

```
        Ok(records) => {
            result.MovieData = records;
            for record in &result.MovieData {
                let title = record.title.as_ref().unwrap();
                result.result.push(format!("{}", title));
            }
        }
        Err(e) => {
            result.result.push(format!("Error: {}", e));
        }
    }
    result
}

pub async fn get_sub_review_list(pool: &MySqlPool, review_id: i32) -> ToGui {
    let mut result = ToGui {
        result: Vec::new(),
        pool: None,
        MovieData: Vec::new(),
        ActorData: Vec::new(),
        ReviewData: Vec::new(),
        MovieList: Vec::new(),
        MovieId: Vec::new(),
        MicroReviewData: Vec::new(),
        SubReview: Vec::new(),
    };
    match get_sub_reviews_from_reviewID(pool, review_id).await {
        Ok(records) => {
            result.SubReview = records;
            for record in &result.SubReview {
                let title = record.sub_review_title.as_ref().unwrap();
                result.result.push(format!("{}", title));
                let desc = record.sub_review_desc.as_ref().unwrap();
                result.result.push(format!("{}", desc));
                let score = record.sub_review_score.as_ref().unwrap();
                result.result.push(format!("{}", score));
            }
        }
        Err(e) => {
            result.result.push(format!("Error: {}", e));
        }
    }
    result
}
```

Db.rs

```
use crate::record::CastId;
use crate::record::CastMovieRecord;
use crate::record::MicroReview;
use crate::record::MovieId;
use crate::record::MovieList;
use crate::record::Record;
use crate::record::Review;
use crate::record::SubReview;
use sqlx;
```

```
use sqlx::MySqlPool;
use sqlx::Row;

//Enum to contain Structs from SQL Query - Kinda clunky
pub enum QueryResults {
    Movies(Vec<Record>),
    Cast(Vec<CastMovieRecord>),
    Review(Vec<Review>),
    MicroReview(Vec<MicroReview>),
}

pub async fn establish_connection() -> Result<MySqlPool, sqlx::Error> {
    dotenv::dotenv().ok();
    let database_url = std::env::var("DATABASE_URL").expect("DATABASE_URL must
be set");
    MySqlPool::connect(&database_url).await
}
pub async fn get_max_review_id(pool: &MySqlPool) -> Result<i32, sqlx::Error> {
    let record = sqlx::query_scalar("SELECT MAX(reviewID) as reviewID FROM
Review")
        .fetch_one(pool)
        .await?;
    Ok(record)
}

pub async fn get_max_sub_review_id(pool: &MySqlPool) -> Result<i32,
sqlx::Error> {
    let record = sqlx::query_scalar("SELECT MAX(subreviewID) as subreviewID
FROM Sub_Review")
        .fetch_one(pool)
        .await?;
    Ok(record)
}

pub async fn get_all(pool: &MySqlPool) -> Result<Vec<Record>, sqlx::Error> {
    // Example implementation, adjust the query as needed

    let records: Vec<Record> = sqlx::query_as!(Record, "SELECT * FROM Movie")
        .fetch_all(pool)
        .await?;
    Ok(records)
}

pub async fn get_cast_from_movieID(
    pool: &MySqlPool,
    movie_id: i32,
) -> Result<Vec<CastMovieRecord>, sqlx::Error> {
    let records: Vec<CastMovieRecord> = sqlx::query_as!(
        CastMovieRecord,
        "SELECT c.movieId, m.title AS movie_title, c.name AS actor_name, c.age
AS actor_age, c.role AS actor_role
        FROM CastMembers c
        JOIN Movie m ON m.movieId = c.movieId
        WHERE c.movieId = ?",
        movie_id
    )
```

```
    .fetch_all(pool)
    .await?;
    Ok(records)
}

pub async fn get_reviews_from_movieID(
    pool: &MySqlPool,
    movie_id: i32,
) -> Result<Vec<Review>, sqlx::Error> {
    let records: Vec<Review> = sqlx::query_as!(
        Review,
        "SELECT * FROM Movie NATURAL JOIN Review WHERE movieId = ?",
        movie_id
    )
    .fetch_all(pool)
    .await?;
    Ok(records)
}

pub async fn filter_by_title(
    pool: &MySqlPool,
    title: String,
) -> Result<Vec<MovieList>, sqlx::Error> {
    let records: Vec<MovieList> = sqlx::query_as!(
        MovieList,
        "SELECT title, movieId FROM Movie WHERE title LIKE ?",
        format!("%{}%", title)
    )
    .fetch_all(pool)
    .await?;

    Ok(records)
}

pub async fn filter_by_actor(
    pool: &MySqlPool,
    name: String,
) -> Result<Vec<MovieList>, sqlx::Error> {
    let name = name.to_lowercase();
    let records: Vec<MovieList> = sqlx::query_as!(
        MovieList,
        "SELECT c.movieId, m.title AS title
        FROM CastMembers c
        JOIN Movie m ON c.movieId = m.movieId
        WHERE c.name LIKE ?",
        format!("%{}%", name),
    )
    .fetch_all(pool)
    .await?;

    Ok(records)
}

pub async fn remove_movie_by_id(pool: &MySqlPool, movie_id: i32) -> Result<(),
sqlx::Error> {
    // First, delete related records from the CastMembers table
    sqlx::query("DELETE FROM CastMembers WHERE movieId = ?")
```

```
        .bind(movie_id)
        .execute(pool)
        .await?;

    //Get reviewID assiocated with movieID
    let row = sqlx::query("SELECT reviewID FROM Review WHERE movieId = ?")
        .bind(movie_id)
        .fetch_optional(pool)
        .await?;

    if let Some(row) = row {
        let review_id: i32 = row.get(0);
        // Delete related records from the Sub_Review table
        sqlx::query("DELETE FROM Sub_Review WHERE reviewID = ?")
            .bind(review_id)
            .execute(pool)
            .await?;
        sqlx::query("DELETE FROM Review WHERE movieId = ?")
            .bind(movie_id)
            .execute(pool)
            .await?;
    }

    // Finally, delete the movie record from the Movie table
    sqlx::query("DELETE FROM Movie WHERE movieId = ?")
        .bind(movie_id)
        .execute(pool)
        .await?;

    Ok(())
}

pub async fn filter_by_release(
    pool: &MySqlPool,
    release: i32,
) -> Result<Vec<MovieList>, sqlx::Error> {
    let records: Vec<MovieList> = sqlx::query_as!(
        MovieList,
        "SELECT title, movieId FROM Movie WHERE releaseDate = ?",
        release
    )
    .fetch_all(pool)
    .await?;

    Ok(records)
}
pub async fn filter_by_format(
    pool: &MySqlPool,
    format: String,
) -> Result<Vec<MovieList>, sqlx::Error> {
    let records: Vec<MovieList> = sqlx::query_as!(
        MovieList,
        "SELECT title, movieId FROM Movie WHERE format = ?",
        format
    )
    .fetch_all(pool)
    .await?;
```

```
    Ok(records)
}
pub async fn get_max_movie_id(pool: &MySqlPool) -> Result<MovieId, sqlx::Error>
{
    let record = sqlx::query_as!(MovieId, "SELECT MAX(movieId) as movieId FROM
Movie")
        .fetch_one(pool)
        .await?;
    Ok(record)
}

pub async fn get_max_cast_id(pool: &MySqlPool) -> Result<CastId, sqlx::Error>
{
    let record = sqlx::query_as!(CastId, "SELECT MAX(castID) as castId FROM
CastMembers")
        .fetch_one(pool)
        .await?;
    Ok(record)
}

pub async fn add(query: String, pool: &MySqlPool) -> Result<(), sqlx::Error> {
    sqlx::query(&query).execute(pool).await?;
    Ok(())
}

pub async fn filter_by_rating(
    pool: &MySqlPool,
    rating: i32,
) -> Result<Vec<MovieList>, sqlx::Error> {
    let records: Vec<MovieList> = sqlx::query_as!(
        MovieList,
        "SELECT title, movieId FROM Review NATURAL JOIN Movie WHERE aggregate
= ?",
        rating // Unwrap the Option<i32> or use a default value
    )
    .fetch_all(pool)
    .await?;

    // Extracting movieId from each tuple
    Ok(records)
}

pub async fn get_sub_reviews_from_reviewID(
    pool: &MySqlPool,
    review_id: i32,
) -> Result<Vec<SubReview>, sqlx::Error> {
    let records: Vec<SubReview> = sqlx::query_as!(
        SubReview,
        "SELECT * FROM Sub_Review WHERE reviewID = ?",
        review_id
    )
    .fetch_all(pool)
    .await?;
    Ok(records)
}
```

```
pub async fn get_movie_details_from_title(
    pool: &MySqlPool,
    title: String,
) -> Result<Vec<Record>, sqlx::Error> {
    let records: Vec<Record> =
        sqlx::query_as!(Record, "SELECT * FROM Movie WHERE title = ?", title)
            .fetch_all(pool)
            .await?;
    Ok(records)
}
```

Record.rs

```
#[derive(sqlx::FromRow, Clone)] // This attribute is used for automatic field
mapping.
#[derive(Debug)]
pub struct Record {
    pub movieId: Option<i32>,
    pub title: Option<String>,
    pub releaseDate: Option<i32>,
    pub format: Option<String>,
    pub description: Option<String>,
}
#[derive(sqlx::FromRow, Clone)]
pub struct CastMovieRecord {
    pub movieId: Option<i32>,
    pub movie_title: Option<String>,
    pub actor_name: Option<String>,
    pub actor_age: Option<i32>,
    pub actor_role: Option<String>,
}

#[derive(sqlx::FromRow, Clone)]
pub struct Review {
    pub reviewID: Option<i32>,
    pub aggregate: Option<i32>,
    pub title: Option<String>,
    pub description: Option<String>,
    pub format: Option<String>,
    pub releaseDate: Option<i32>,
    pub sub_review_num: Option<i32>,
    pub movieId: Option<i32>,
}
#[derive(sqlx::FromRow, Clone)]
pub struct MicroReview {
    pub reviewID: Option<i32>,
    pub aggregate: Option<i32>,
    pub sub_review_num: Option<i32>,
    pub movieId: Option<i32>,
}
#[derive(sqlx::FromRow, Clone)]
pub struct SubReview {
    pub reviewID: Option<i32>,
    pub subreviewID: Option<i32>,
    pub sub_review_score: Option<i32>,
    pub sub_review_title: Option<String>,
    pub sub_review_desc: Option<String>,
```

```
}

#[derive(sqlx::FromRow, Clone)]
pub struct MovieList {
    pub movieId: Option<i32>,
    pub title: Option<String>,
}

pub struct FromGui {
    pub actor_name: Vec<String>,
    pub actor_age: Vec<i32>,
    pub actor_role: Vec<String>,
    pub aggregate: i32,
    pub title: String,
    pub description: String,
    pub format: String,
    pub releaseDate: i32,
    pub sub_review_num: i32,
    pub sub_review_score: Vec<i32>,
    pub sub_review_title: Vec<String>,
    pub sub_review_desc: Vec<String>,
}

#[derive(sqlx::FromRow, Clone)]
pub struct MovieId {
    pub movieId: Option<i32>,
}
pub struct CastId {
    pub castId: Option<i32>,
}
```

.env

```
DATABASE_URL=mariadb://root:root@localhost:3306/movies
```