This Report is submitted to

College of Engineering and Science,

Slippery Rock University

In Partial Fulfillment

of the Requirements for the Degree of Bachelor's in Computer Science.

By

Tucker Davis, Benjamin Luzier, Joseph Reiner, and Brenden Stilwell

A report submitted to the College of Engineering and Science, Slippery Rock University, in partial fulfillment of the requirements for the Degree of Bachelor's in Computer Science by:

Tucker Davis, Benjamin Luzier, Joseph Reiner, and Brenden Stilwell

The report is hereby approved by:

_____

Nitin Sukhija, Ph.D.
Assistant Professor, Department of Computer Science

Slippery Rock University of Pennsylvania

_____

Date

Table of Contents

Introduction

The aim of the Listening Study application is to provide a studying tool that mimics the listening exam environment found in university music courses. These exams quiz the student on a set of songs, where each question involves listening to a random portion of a song. In most exam settings, including ours, the student is tasked with recalling the name, composer, and year of the song. Students of these courses can utilize Listening Study as a streamlined way of study, and professors can provide songs to their students through simple means.

At its core, the application provides students with practice quizzes to prepare for their real exams. These quizzes are created with consideration for past performance so that every quiz is personalized to the student. For this reason, it is not intended as an environment to give a real exam.

This Software Requirements Specification (SRS) document is intended to be read by instructors and developers of the software. It is a comprehensive guide of the application's specifications, design, and requirements. Instructors can get an overview of their role in the application, whether teacher, moderator, or administrator, and how to use it to its fullest. Developers can get an understanding of the system's requirements, a technical overview, and the future direction of development.

This document will first give an overview of the application from a high level. Here, the principal functions of the system, different user roles, and system dependencies will be discussed. Next will be a technical overview, which includes things such as the technologies used and system architecture. There will also be a guide on how to set up the project. After this, there will be an explanation of the functional and non-functional requirements of the system. It will define everything the system should do (functional) and how well it will do it (non-functional). Next, there will be an overview of user interfaces, external software interfaces, and communication interfaces. The document will then showcase the use cases for the application along with the use case diagram, sequence diagrams, and data flow diagrams. Lastly, it will conclude and speak about future development.

High-Level Overview

Product Functions

This application was built with student experience in mind. Without a studying application, students would need to fetch all the songs themselves and have a peer quiz them. This application will circumvent this tedium, allowing students to focus solely on preparing for their listening exam. Rather than the students searching for the music, they can access playlists created by their teachers consisting of YouTube URLs to the songs. Then, they can simply choose a playlist to be quizzed on. These playlists are created for a specific class, where each class consists of one teacher and many students.

Playlists can be created, edited, and deleted by the teacher. When the playlist is formed, the student can take a quiz based upon it. There are various quiz options the student can choose from to affect how the quiz is given. They affect the timestamp the song will play from, the playback duration, and the number of questions. There is constant modification to the quiz questions based on user performance, so that every quiz is personalized to the student.

User Roles and Characteristics

This system has four user roles: student, teacher, moderator, and administrator. Every user, regardless of role, accesses their dashboard from the same log-in page. The user's dashboard presents various actions to the user depending on their role.

Students have many actions they can take. They can view the playlists from their classes and preview the songs within them. They can also design and take quizzes from the dashboard. First, the options for the quiz are chosen, then they take the quiz. After the quiz is taken, they can see the results of their quiz. Students can also see their overall performance on songs on the performance page. Here, it shows information the user may be interested in for measuring their knowledge and progress. Lastly, students can report bugs to the system administrator.

Similarly to students, teachers can view playlists and create bug reports. When viewing the playlist, however, they can also choose to modify the playlist. This includes adding, editing, or deleting songs from it. They can also choose to delete an existing playlist or create a new playlist. Additionally, they can view the students in their class or classes.

From the moderator's dashboard, they can view every class in the system. They have the option to disable or remove these classes. Disabling a class prevents users from interacting with the class; however, its information is still stored and can be enabled again. Removing a class effectively deletes the class and cannot be undone. The moderator can also create classes, which involves naming the class and designating an existing teacher account to manage the class. Lastly, moderators can create bug reports like students and teachers.

The administrator's main duties are to manage bug reports and users. Bug reports are received on their dashboard, and once dealt with, can be designated as acknowledged or resolved. The administrator may acknowledge a report if they recognize the report is valid or a fix is in progress. They will only resolve the report if the bug fix is fully integrated into production.

Statement of Need

SRU's music department offers classes that conduct music listening exams. In these exams, the student listens to a portion of a song and is tasked with recalling the song name and composer. Currently, there does not exist an application for students to prepare for these exams without the assistance of another individual. The goal of this project is to create a software tool that mimics the environment of the exam for students to study effectively. Users of this application, students and professors, can create, share, and study from study sets.

Technical Overview

This section of the document provides an overview of the technology and project setup. The following are assumptions that have been made to have the application run correctly- access to YouTube, an Internet connection, an adblocker that works with YouTube, the ability to download and install executables on local machine, and the ability to make changes to system environment variables. The application has been tested only on Windows 11 machines. The application is NOT guaranteed to run on a Linux machine, and the installation instructions below are intended to work on a Windows 11 system. Note that if these requirements cannot be fulfilled, the application may not function as intended. This application features a standard front-end, back-end, database architecture for a web application.

Front-end

The front-end of the Listening Study Application is comprised of three languages- HTML, CSS, and JavaScript. One HTML framework is used, which is Bootstrap. One library is utilized in JavaScript, which is YouTube API (for song playback). YouTube API is used to be able to access and play songs given YouTube video links. One framework is utilized in JavaScript, which is jQuery. jQuery is a framework that allows for enhanced selection of element tags over vanilla JavaScript.

Back-end

The back-end of the Listening Study Application is comprised entirely of Java 17. While no frameworks are utilized, several libraries/dependencies are included within the application. For connection to the database, MySQL Connector-J (also known as JDBC) is included. For data parsing between the frontend and backend, GSON and Jackson are utilized. For rendering reusable static file templates (HTML) within the application, Thymeleaf is utilized. For password hashing/decryption, Mindrot JBCrypt is used. For web scraping of YouTube songs, JSoup and Selenium are included. For implementation of our model, the Apache Math3 Commons dependency is included. Finally, for unit testing, both JUnit and Mockito are utilized.

Database

The database used for the Listening Study application is MySQL. The application database consists of one schema with 16 tables, and several views (which retrieve specific data from multiple tables for enhanced data retrieval).

System Architecture

The Listening Study Application can be run locally on a laptop or computer. The application has been run on various machines throughout the development process with minimal issues. The “weakest” system configuration that the application successfully ran on is included below:

- Processor Intel(R) Core(TM) i5-8365U CPU @ 1.60GHz, 1896 Mhz, 4 Core(s), 8 Logical Processor(s)
- Installed RAM: 16 GB

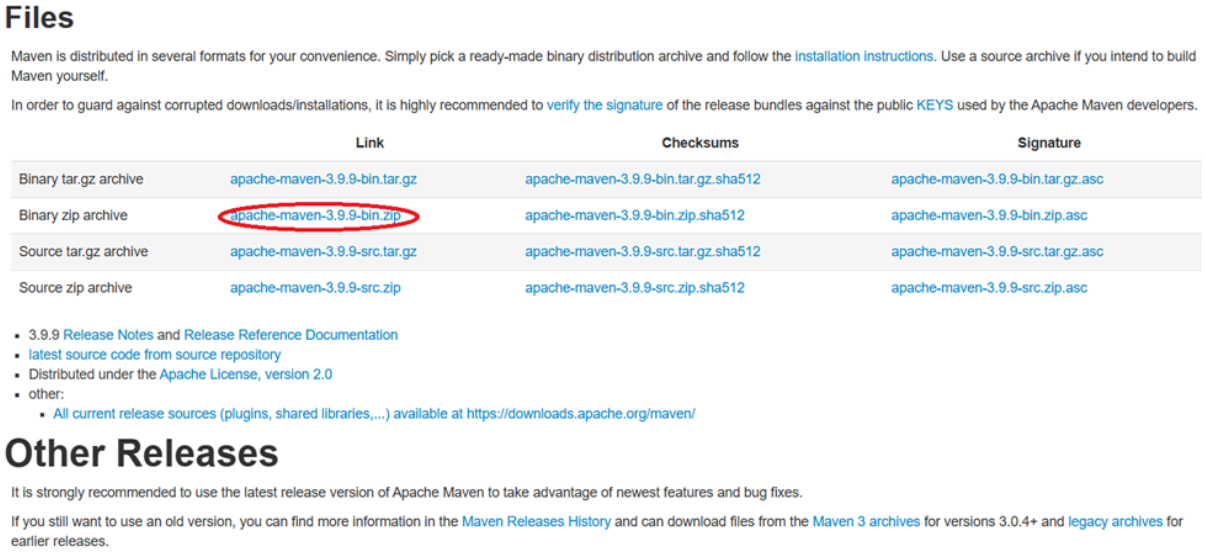
Local Setup Instructions

This section of the document outlines the entire process to be able to successfully run the Listening Study Application locally on a new machine. There are two external downloads and configurations required to be able to run the application.

Maven

Maven is a package manager for Java applications. It allows for less external downloads/configurations for projects and instead includes an XML build file to add dependencies. The instructions to download and configure Maven are shown below:

1. Navigate to this link in a web browser https://maven.apache.org/download.cgi
2. Download the binary zip circled in the screenshot below:



Files

Maven is distributed in several formats for your convenience. Simply pick a ready-made binary distribution archive and follow the installation instructions. Use a source archive if you intend to build Maven yourself.

In order to guard against corrupted downloads/installations, it is highly recommended to verify the signature of the release bundles against the public KEYS used by the Apache Maven developers.

	Link	Checksums	Signature
Binary tar.gz archive	apache-maven-3.9.9-bin.tar.gz	apache-maven-3.9.9-bin.tar.gz.sha512	apache-maven-3.9.9-bin.tar.gz.asc
Binary zip archive	apache-maven-3.9.9-bin.zip	apache-maven-3.9.9-bin.zip.sha512	apache-maven-3.9.9-bin.zip.asc
Source tar.gz archive	apache-maven-3.9.9-src.tar.gz	apache-maven-3.9.9-src.tar.gz.sha512	apache-maven-3.9.9-src.tar.gz.asc
Source zip archive	apache-maven-3.9.9-src.zip	apache-maven-3.9.9-src.zip.sha512	apache-maven-3.9.9-src.zip.asc

- 3.9.9 Release Notes and Release Reference Documentation
- latest source code from source repository
- Distributed under the Apache License, version 2.0
- other:
  - All current release sources (plugins, shared libraries,...) available at https://downloads.apache.org/maven/

Other Releases

It is strongly recommended to use the latest release version of Apache Maven to take advantage of newest features and bug fixes.

If you still want to use an old version, you can find more information in the Maven Releases History and can download files from the Maven 3 archives for versions 3.0.4+ and legacy archives for earlier releases.

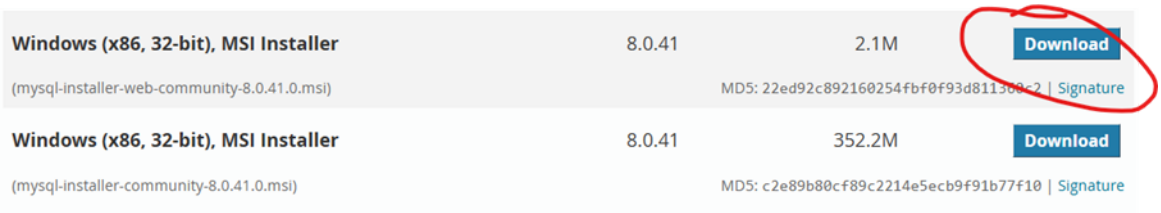
3. Extract the zip to C:\Program Files (make sure to allow any administrator prompts)
4. Press windows key, then type “edit the system environment variables”
5. Open the suggested application
6. Click “Environment Variables...” near the bottom-right of the window
7. Click “New…” under **SYSTEM** variables
    a. Name: MAVEN_HOME
    b. Value: C:\Program Files\apache-maven-3.9.9
8. Click OK, then find the PATH variable
    a. Click Edit…

   b. Click New
   c. Type “C:\Program Files\apache-maven-3.9.9\bin”
9. Click OK, then OK, then OK. You should now be out of the editor entirely
10. A system reboot may be required to complete the configuration

MySQL Database

The database used for this application is MySQL. Although any MySQL editor could work for this application, the exact details for the configuration of the MySQL editor for this application are included. If you prefer to use an already existing MySQL editor, then skip to step 19.

1. Navigate to this link in a web browser https://dev.mysql.com/downloads/installer/
2. Download the MSI Installer circled below:



3. Run the installer
4. Click through prompts until “Custom Setup” is an option
5. Click Custom Setup
6. Click the most recent MySQL Server, then click the arrow
7. Click the most recent MySQL Workbench, then click the arrow
8. Click execute
9. Click install
10. Keep Type Networking default
11. Keep Group Replication default
12. Keep Authentication Method default
13. For the default account, specify:
    a. MySQL root password: root
    b. Repeat password: root
    c. Add User (this will add another user)
        i. Username: admin

ii. DB_ADMIN

iii. Password: admin

14. Keep Windows Service default
15. Click execute
16. Finish and start MySQL workbench
17. Connect to the localhost server by adding a new connection (see arrow below)

18. Add the following credentials:
    a. User: root
    b. Password: root
19. Now, navigate to the src/main/resources/db directory of the project
20. Open schema.sql, copy its contents, and run the entire paste in SQL editor
21. Verify that schema.sql ran without issues
22. Open data.sql, copy its contents, and run the entire paste in SQL editor

Note: Changing default credentials will require changes in application.properties. Here are some of the fields that may be impacted:

```
# Server/DB Configuration
server.port=8080
db.username=root
db.password=root
db.url=jdbc:mysql://localhost:3306/listeningapp2
```

Now, with these two external entities downloaded and configured, the application can be run. From a Linux based terminal (we used Git Bash in testing), navigate to the project root (the directory with pom.xml in it). Run the following commands:

- $mvn clean install
    - Only needs run once
- $mvn clean package
    - Needs run everytime the applicaiton needs started.

By default, the project port is set to 8080. The home screen should now be visible at the following URL in a web browser:

- http://localhost:8080

Use Cases

Descriptions

Create Account:

Actors: Student, Teacher

Inputs: First name, last name, password, re-typed password, email, account type

Outputs: New account of teacher or student

Normal Operation:

A user will fill in all input fields and declare their account type. They will then be taken to the dashboard upon successful account creation.

Exceptions:

Emails must be unique values pertaining to no other users. Password and re-typed passwords must have the same value.

Log In:

Actors: Student, Teacher, Moderator, Admin

Inputs: Email and password

Outputs: Transferred to dashboard

Normal Operation:

A user will enter their email and password and then be redirected to the dashboard based on a successful login with matching email and password.

Exceptions:

Email and password values must not be empty. Both values must be valid.

View Performance:

Actors: Student

Inputs: Student and class details

Outputs: Display of student's quiz results in a given class

Normal Operation:

The student chooses to view quiz performance on their dashboard. The page will display the student's past results with statistical insights. The page will also show the student's average performance on each song that they have seen in a quiz.

Exceptions:

Page will be empty if a student has not taken any previous quizzes for the specified class.

Set Quiz:

Actors: Student

Inputs: Selected playlist, number of questions, song playback method, song playback duration

Outputs: Viable parameters for quiz generation

Normal Operation:

A student can set the settings that they want for a quiz. The settings include the playlist, number of questions, song playback method, and song playback duration. The playlist is the list of songs that the system can reference for generating quiz questions. Number of questions is the amount of questions that a student wants to appear on a single quiz. The song playback method refers to how the system will handle the playback of a song snippet during a quiz. Options include random selection, YouTube's most replayed times, and user specified timestamps. The playback duration refers to the amount of time that a song will play during a quiz.

Exceptions:

All fields for settings must be filled. If a user exits the quiz, it will become unset.

Take Quiz:

Actors: Student

Inputs: Selected playlist, settings for quiz generation, song data

Outputs: Generated quiz based on settings from the user

Normal Operation:

Students are able to take a quiz that is generated based on the specified settings that the student selected for quiz generation. Song snippets will be played for each quiz question based on the quiz settings, and the songs that are more likely to appear will be determined by the weights. The results of a quiz will be calculated, displayed, and stored once the quiz is complete. The stored data will then be used to alter the questions for following quizzes. The generated questions will come from the associated playlist that contains the song data for grading.

Exceptions:

A student cannot take a quiz if the settings have not been specified. A quiz cannot be completed if the user exits out of the quiz.

Calculate Results:

Actors: Student

Inputs: Student's quiz answers, song information from the playlist

Outputs: Calculated quiz score

Normal Operation:

Once a student completes a quiz, their score on that particular quiz and its questions will be calculated and displayed. Scores for individual songs will also be calculated and sent to storage for reference during quiz alteration.

Exceptions:

Results cannot be calculated if the user exits out of the quiz.

Store Overall Performance:

Actors: Student

Inputs: Student's performance on specific songs

Outputs: Student's average performance on specific songs

Normal Operation:

The calculated results for specific songs from quizzes are combined with the averages on specific songs from previous quizzes. These newly calculated results are then stored as the student's overall performance on specific songs. This data can then be referenced for future quiz alterations.

Exceptions:

Songs that have never appeared on a quiz will have no performance data.

Alter Questions:

Actors: Student

Inputs: Selected playlist, quiz settings, user performance

Outputs: Quiz with altered questions based on previous user performance and quiz settings

Normal Operation:

The questions that appear on a quiz will be altered based on the student's overall performance. This performance will be represented by question averages which will be used when calculating beliefs. A question that has a lower belief value will be more likely to appear on a question than a question with a higher belief value.

Exceptions:

Questions cannot be altered based on previous quiz results if there are no previous quiz results.

Calculate Question Belief:

Actors: Student

Inputs: Previous user performance on specific songs

Outputs: Beliefs for certain songs to appear in the quiz

Normal Operation:

Beliefs will be applied to certain songs in order to make them more likely to appear on a quiz. The songs that receive low beliefs are songs for which the student has a lower average performance when compared to other songs. Beliefs are calculated through Thompson sampling.

Exceptions:

Beliefs cannot be applied to songs with no previous quiz results, so alpha and beta will be set to 1 for the beta distribution sample.

Get Performance:

Actors: Student

Inputs: Student, song in playlist

Outputs: User performance during quizzes for given song in playlist

Normal Operation:

The system retrieves a student's performance data for specific songs so that it can be analyzed and prepared for weight determination. Songs with lower average performances are designated as songs of interest.

Exceptions:

Students must have previous quiz results.

Play Song Snippet:

Actors: Student, YouTube API

Inputs: Quiz being taken, songs from the associated playlist, quiz settings

Outputs: Audio snippet of a song from the playlist attached to a quiz question

Normal Operation:

The student starts a quiz, and the quiz generates questions based on songs found in the associated playlist. As part of the question, an audio snippet of the song will be played with the selected playback method and playback duration.

Exceptions:

Invalid YouTube links or blocked videos cannot be played.

Play User Defined Timestamps:

Actors: Student

Inputs: Quiz settings, timestamps, song

Outputs: Song played in quiz at user-defined timestamp

Normal Operation:

During a quiz, the system will begin the playback of a song from a timestamp defined by the user. The song will play for the amount of time that is specified in the quiz settings.

Exceptions:

A song must be selected for the playback to begin, and a quiz must be active. A song must have user defined timestamp(s) previously set by the teacher in the edit song, otherwise it will play a random section.

Play Random Timestamp:

Actors: Student

Inputs: Quiz settings, song

Outputs: Song played in quiz at random timestamp

Normal Operation:

During a quiz, the system will begin the playback of a song from a randomly selected time. The song will play for the amount of time that is specified in the quiz settings.

Exceptions:

A song must be selected for the playback to begin, and a quiz must be active.

Play Most Replayed Timestamps:

Actors: YouTube API, Student

Inputs: Quiz settings, timestamps retrieved from API, song

Outputs: Song played in quiz at most replayed timestamp

Normal Operation:

During a quiz, the system will begin the playback of a song from the region that is the most replayed according to YouTube's analytics. The song will play for the amount of time specified in the quiz settings.

Exceptions:

A song must be selected for the playback to begin, and a quiz must be active. The most replayed timestamps cannot be determined without data from YouTube. If there is no YouTube data available, it will default to random selection.

View Playlist:

Actors: Student, Teacher

Inputs: Playlist, class

Outputs: Table of songs in playlist

Normal Operation:

Teachers can view playlists they create for a specific class. Students can view playlists that are available to them from a class they are assigned to by the moderator. Both teachers and students can listen to the songs in the playlist. Teachers have buttons for editing or removing songs.

Exceptions:

The playlist is empty. There are no created playlists.

Modify Playlist:

Actors: Teacher

Inputs: Playlist, class

Outputs: Updated playlist

Normal Operation:

The teacher can select a playlist to modify. The teacher has the options of adding a song, removing a song, or editing a song's data. The teacher will be able to save the playlist once their edits are complete.

Exceptions:

A playlist must exist to be updated. An empty playlist has no songs to modify or remove.

Add Song:

Actors: Teacher

Inputs: Playlist, song name, composer, year, YouTube link

Outputs: Updated playlist

Normal Operation:

The teacher may add a song by adding its YouTube link to an existing playlist, or add a song as part of creating a new playlist. In addition to adding the song, the teacher must also add the necessary song data. This includes the song's name, composer, and the song's year of release. The teacher can also add specific timestamps for playback.

Exceptions:

Song needs to have a valid, publicly accessible YouTube link. All fields must be filled. A song link cannot already be in the playlist.

Remove Song:

Actors: Teacher

Inputs: Playlist and song

Outputs: Playlist without song

Normal Operation:

The teacher can remove a song from a playlist. Once the song is removed, it will no longer appear in the playlist, and it can no longer appear in quizzes that use that playlist.

Exceptions:

A song must exist in order to be removed. The playlist must exist.

Edit Song:

Actors: Teacher

Inputs: Title, composer, playback timestamps, year

Outputs: Data set for each individual song in the playlist

Normal Operation:

The teacher can edit the data of an existing song in an existing playlist. The teacher can alter the song's name, composer, and year of release. The teacher can also specify timestamps for playback.

Exceptions:

All required fields must be filled to be saved to a playlist. A song must exist in a playlist to edit.

Preview Song:

Actors: YouTube API, Student, Teacher

Inputs: Selected song from a playlist

Outputs: The audio for the song is played

Normal Operation:

Teachers and students are able to view playlists and preview the songs within them. The songs will be played with audio only.

Exceptions:

If the link is invalid or if the YouTube video is unavailable, the song will not be able to be played.

View Roster:

Actors: Teacher

Inputs: Students

Outputs: Display of all students in teacher's class(es)

Normal Operation:

Teachers can view a list of the students in their classes. Each class has its own separate list.

Exceptions:

A new teacher will not have any classes that can be viewed.

Create Playlist:

Actors: Teacher

Inputs: Playlist title, songs (YouTube links and song details), class

Outputs: Newly created playlist, confirmation message

Normal Operation:

The teacher can create a new playlist and add songs to it. The teacher also has to enter the song data for each song that they add to the playlist. The song data includes the song's name, composer, and year of release. The teacher will be able to save the created playlist, and it will be accessible to the class specified.

Exceptions:

If no songs are added, the playlist cannot be created and it will be discarded. A playlist can only belong to one class.

Create Class:

Actors: Moderator

Inputs: Teacher, students, class name

Outputs: Created class

Normal Operation:

A moderator can create a class and assign a teacher to it. The moderator must assign a name to the class. The moderator can also add students to the class.

Exceptions:

There can only be one teacher per class. A class must have a name.

Add User to Class:

Actors: Moderator

Inputs: Student/teacher information, class

Outputs: Student/teacher added to a class

Normal Operation:

A moderator can add a teacher and students to a class by searching for them in the system. They can be located in the system via email. Being added to a class grants the added users access to the data of the class.

Exceptions:

Missing required fields. The necessary fields for the student/teacher's information must be filled out in order for them to be added to the class. If the student/teacher is not in the system, then they cannot be added to a class.

Edit Class:

Actors: Moderator

Inputs: Student/teacher, class, class name change, option to add or remove user

Outputs: Students/Teacher added or removed, or class name changed

Normal Operation:

A moderator can select a class and make edits to it. The editing options include swapping a teacher or adding/removing a student to a class. They can also choose to change the name of the class.

Exceptions:

A class must exist, a user (teacher/student) must exist for the given class.

Remove User from Class:

Actors: Moderator

Inputs: Student/teacher, class

Outputs: Student/teacher removed from class

Normal Operation:

A moderator is able to remove students or a teacher from a class. The teacher and the students are removed by email. When a teacher is selected for removal, a replacement teacher must be selected because a teacher must always exist in a class.

Exceptions:

If the class or student/teacher does not exist, then it will prompt that it does not exist. A teacher will not be removed without a replacement.

Delete Class:

Actors: Moderator

Inputs: Class and teacher

Outputs: Class deletion confirmation message

Normal Operation:

A moderator can delete a class from the system. The deletion of a class will result in the teacher and the students associated with the class no longer having access to the class or the data within it. This includes all class playlists. The system will prompt the user for a confirmation message to confirm their choice.

Exceptions:

Class must exist to be deleted

Designate Moderators:

Actors: Admin

Inputs: Teacher

Outputs: Moderator status for chosen teacher

Normal Operation:

An admin can choose a teacher account from the system to designate as a moderator. Any classes and playlists created by that teacher will be removed.

Exceptions:

The specified teacher account must exist. Student accounts cannot be designated as moderators.

Create Report:

Actors: Student, Teacher, Moderator, Admin

Inputs: Bug details, report date, reporter

Outputs: Message sent notification

Normal Operation:

Any user can create a bug report within the system. The user fills out a form reporting the issue that is then sent to the admin. The report date is retrieved and added to the message. The user who reported the reports email and account type will also be sent.

Exceptions:

The report description must have details to be sent.

Report Date:

Actors: Time

Inputs: Time of bug report, report ID

Outputs: A date assigned to a bug report.

Normal Operation:

The system will assign the current date and time to a bug report.

Exceptions:

The report must exist to be generated.

View Reports:

Actors: Admin

Inputs: Report ID

Outputs: Display report description and date, Reporter

Normal Operation:

The admin can view a bug report from a list of reports and acknowledge it to begin addressing the reported issue.

Exceptions:

The report does not exist or has been deleted.

Resolve Report:

Actors: Admin

Inputs: Report ID

Outputs: Resolved report

Normal Operation:

After the admin has viewed and fixed a bug report, they can resolve it. The report will be archived.

Exceptions:

The report does not exist or has already been resolved.

Delete Users:

Actors: Admin

Inputs: User email to be deleted

Outputs: The specified user is deleted

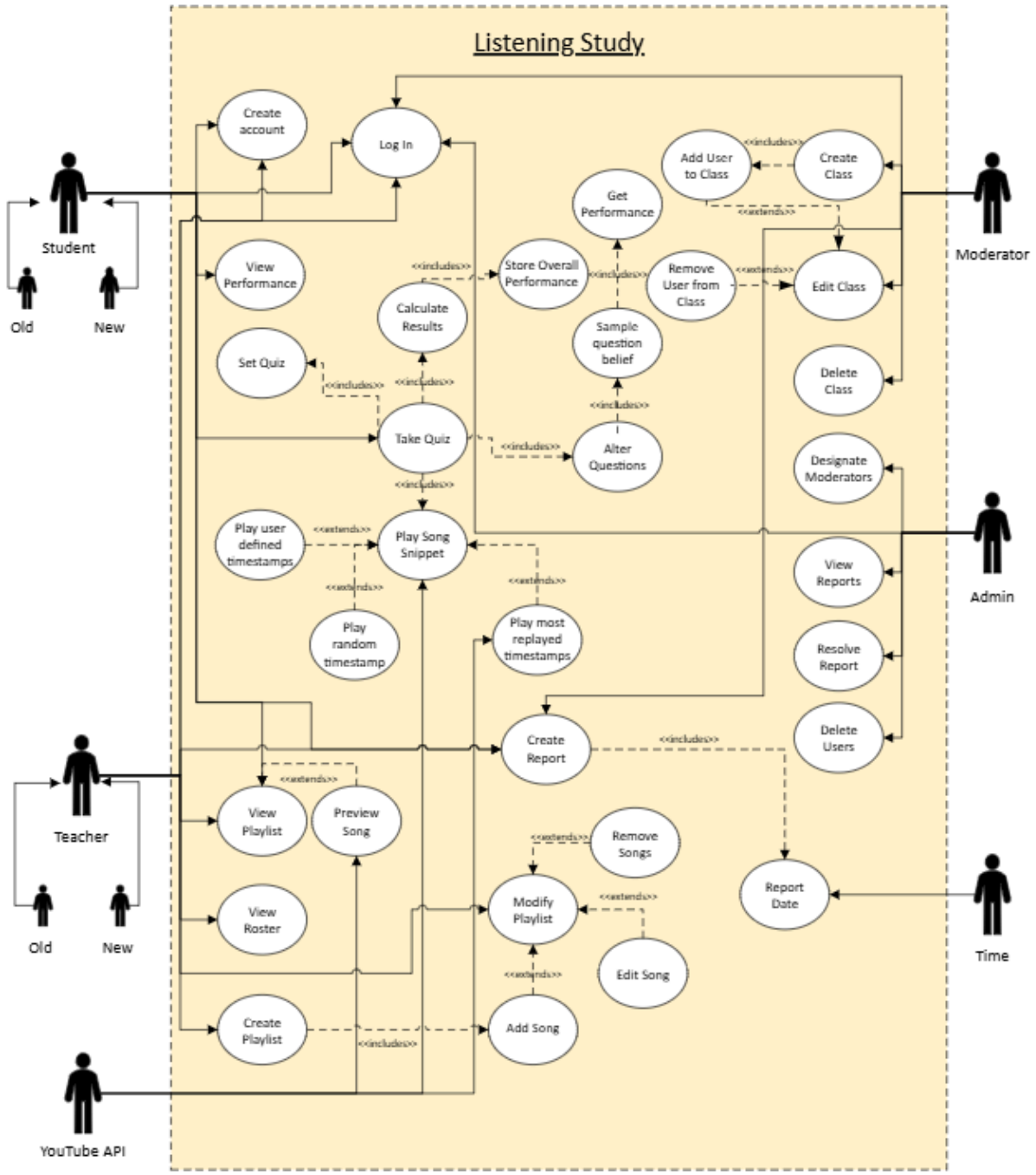
Normal Operation:

The admin can search for a user by email within the system, and if that user is found, the user will be deleted from the system. A deleted user is no longer able to login using their account.

Exceptions:

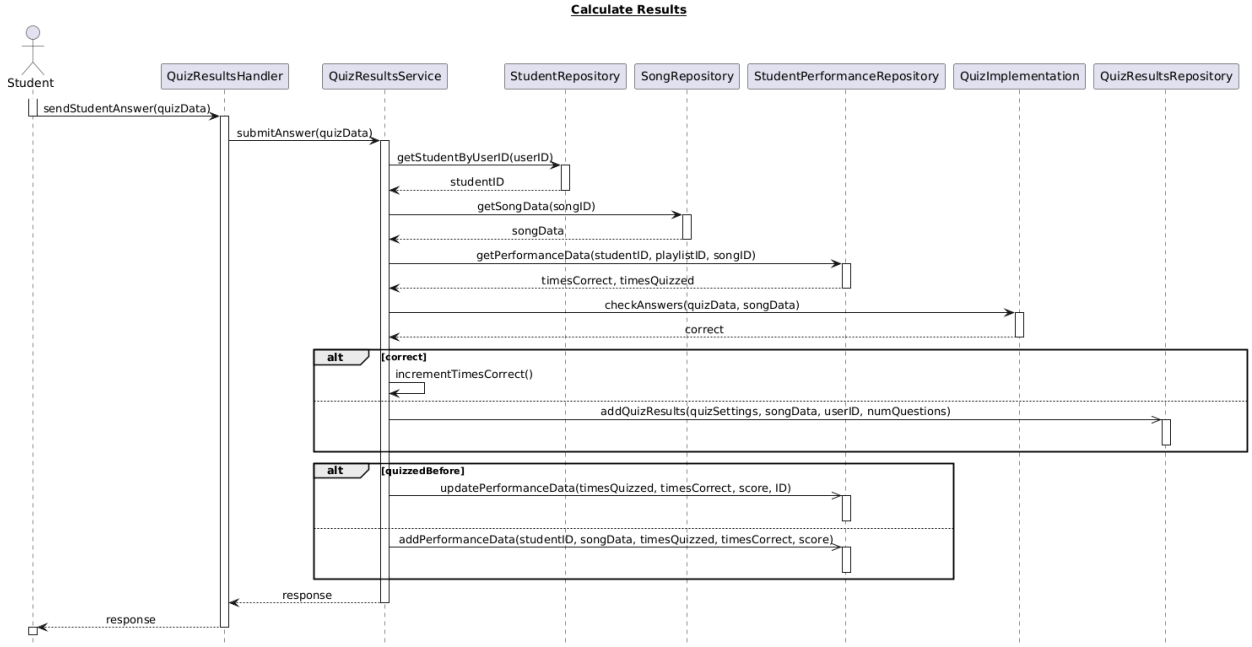
The user does not exist. If the user does not exist in the system, then they cannot be deleted.

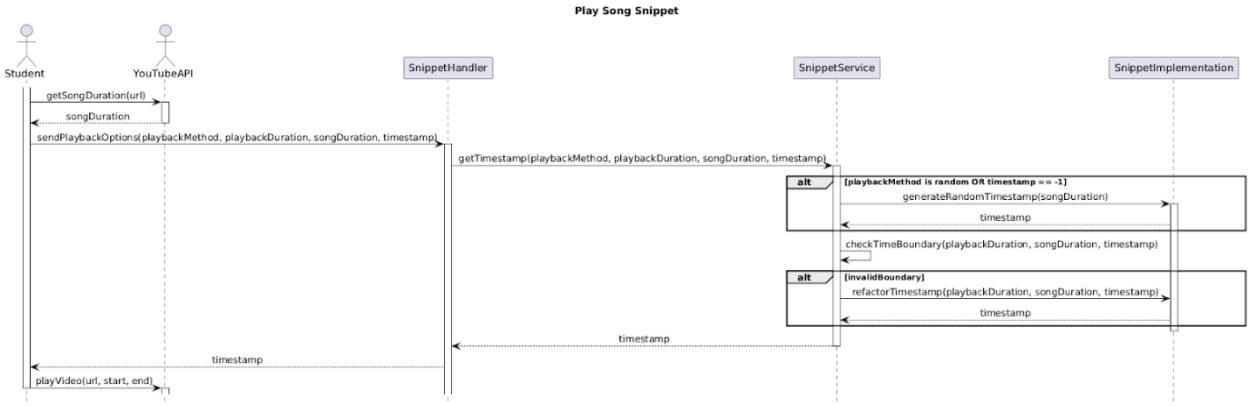
Use Case Diagram

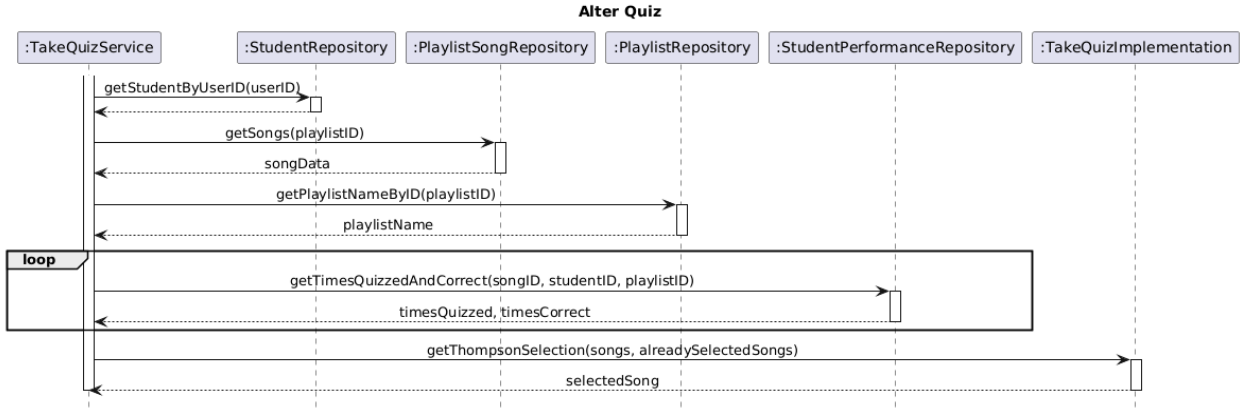


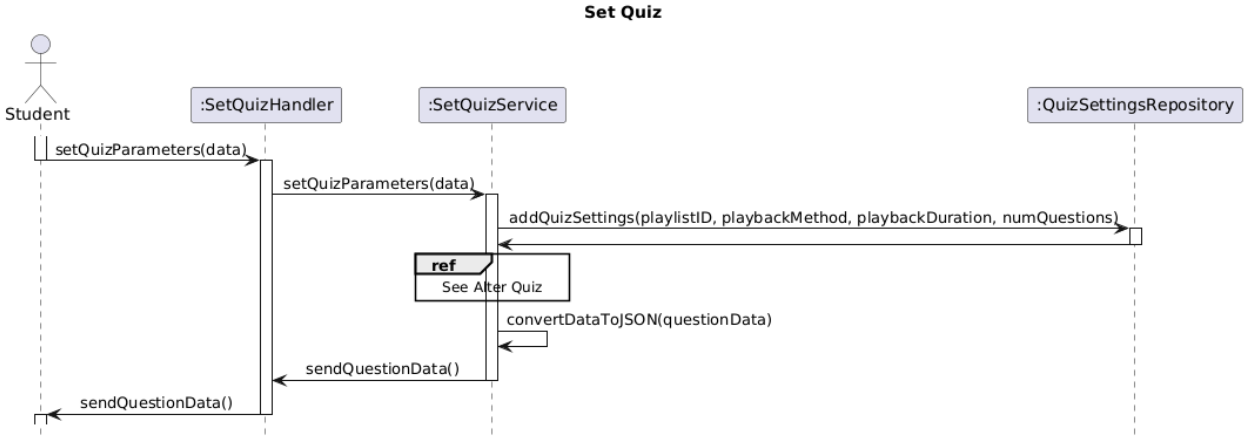
Sequence Diagrams

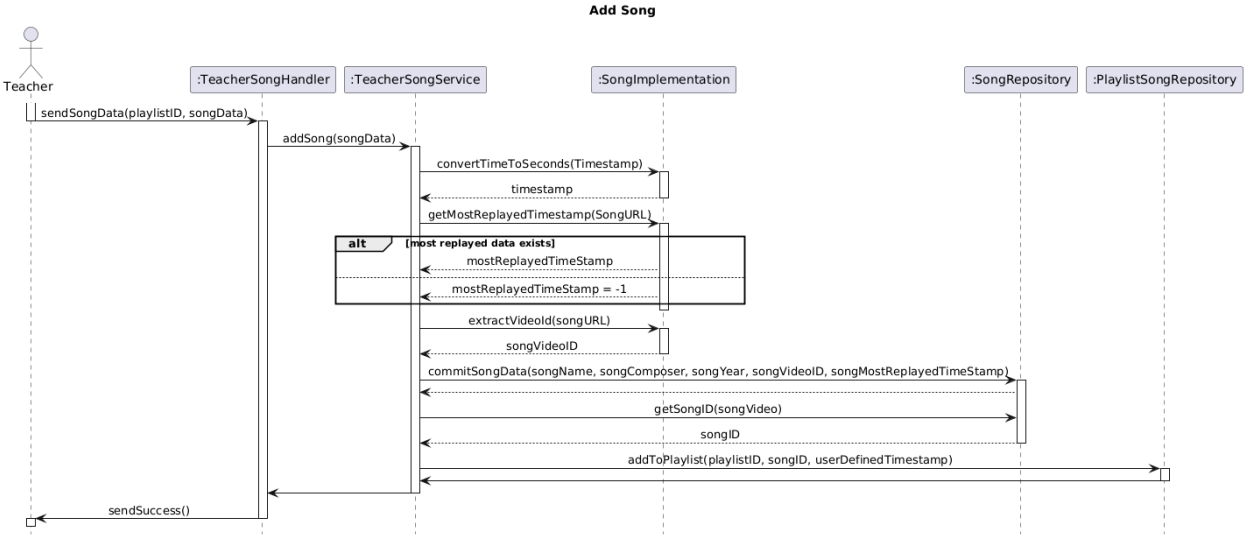
The five sequence diagrams for the Core Processing of the Listening Study Are included below:



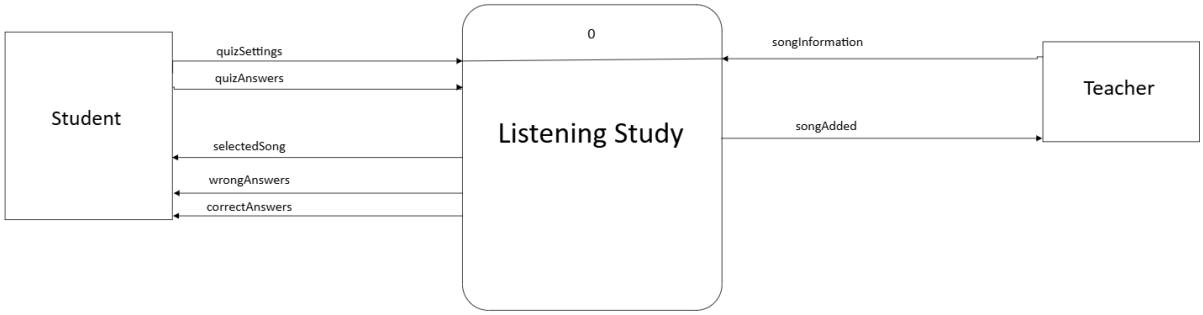


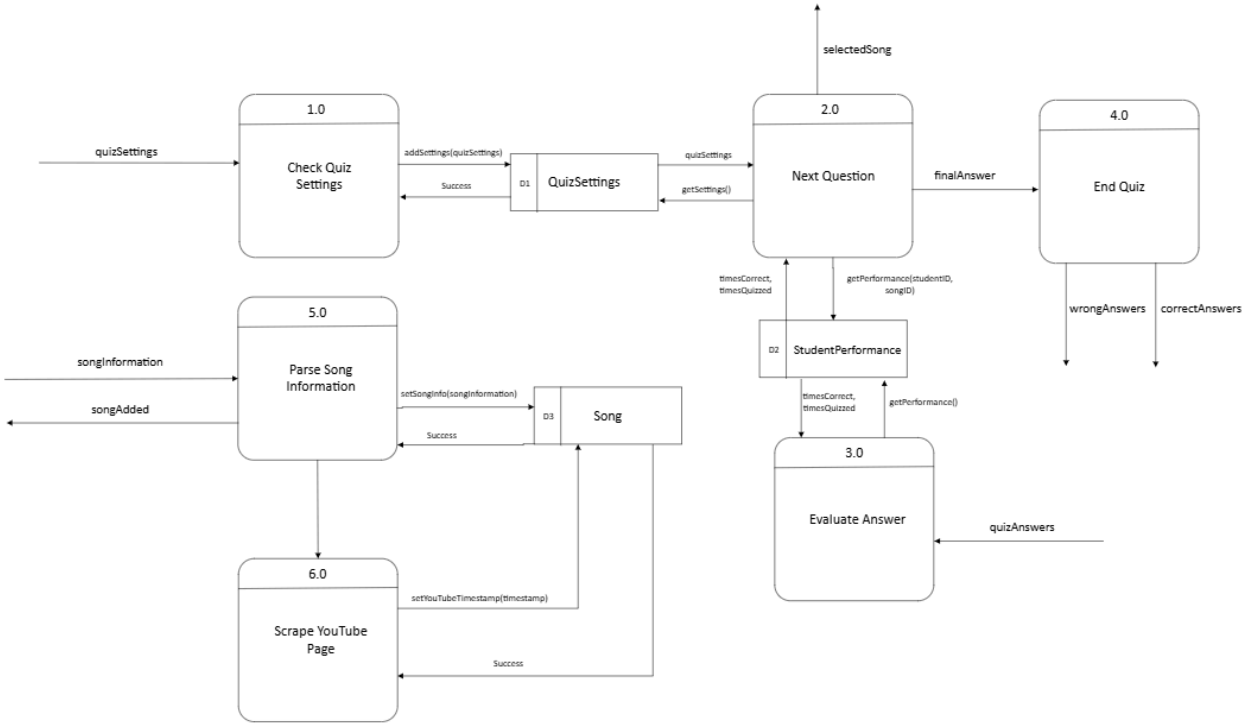

Alter Quiz
:TakeQuizService
:StudentRepository
:PlaylistSongRepository
:PlaylistRepository
:StudentPerformanceRepository
:TakeQuizImplementation
getStudentByUserID(userID)
getSongs(playlistID)
songData
getPlaylistNameByID(playlistID)
playlistName
loop
getTimesQuizzedAndCorrect(songID, studentID, playlistID)
timesQuizzed, timesCorrect
getThompsonSelection(songs, alreadySelectedSongs)
selectedSong

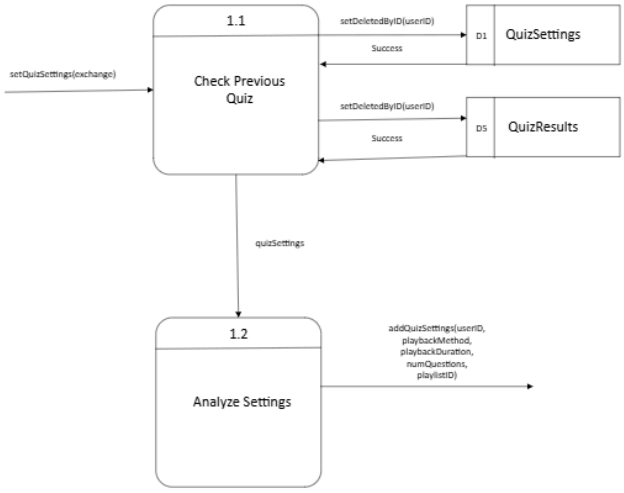

Set Quiz
Student
:SetQuizHandler
:SetQuizService
:QuizSettingsRepository
setQuizParameters(data)
setQuizParameters(data)
addQuizSettings(playlistID, playbackMethod, playbackDuration, numQuestions)
ref
See Alter Quiz
convertDataToJSON(questionData)
sendQuestionData()
sendQuestionData()

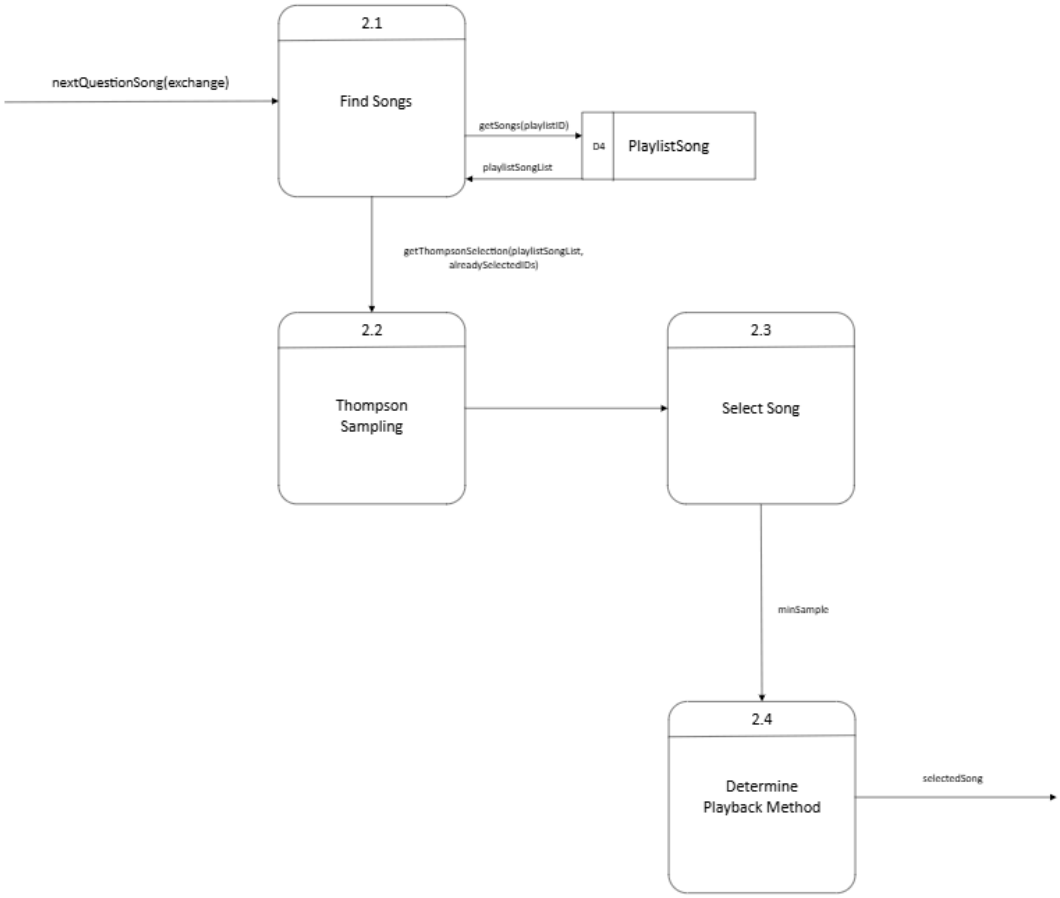

Add Song
Teacher
:TeacherSongHandler
:TeacherSongService
:SongImplementation
:SongRepository
:PlaylistSongRepository
sendSongData(playlistID, songData)
addSong(songData)
convertTimeToSeconds(Timestamp)
timestamp
getMostReplayedTimestamp(SongURL)
alt
[most replayed data exists]
mostReplayedTimeStamp
mostReplayedTimeStamp = -1
extractVideoId(songURL)
songVideoID
commitSongData(songName, songComposer, songYear, songVideoID, songMostReplayedTimeStamp)
getSongID(songVideo)
songID
addToPlaylist(playlistID, songID, userDefinedTimestamp)
sendSuccess()

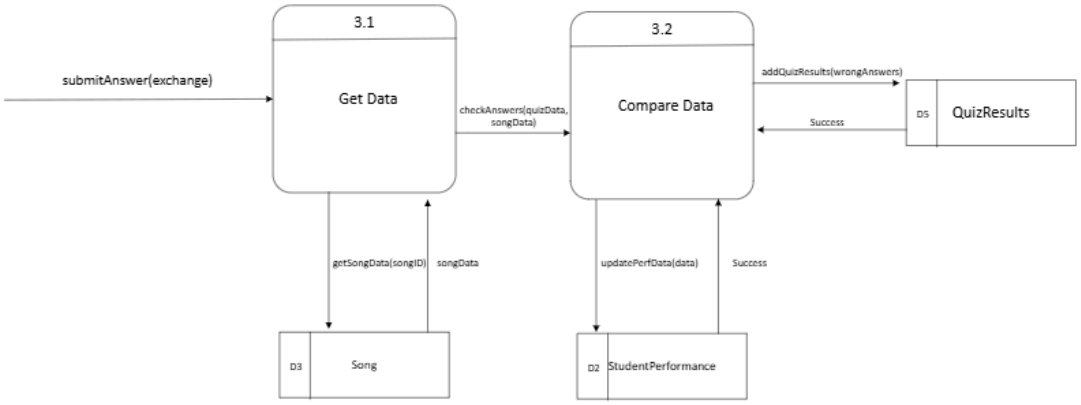
Data Flow Diagrams

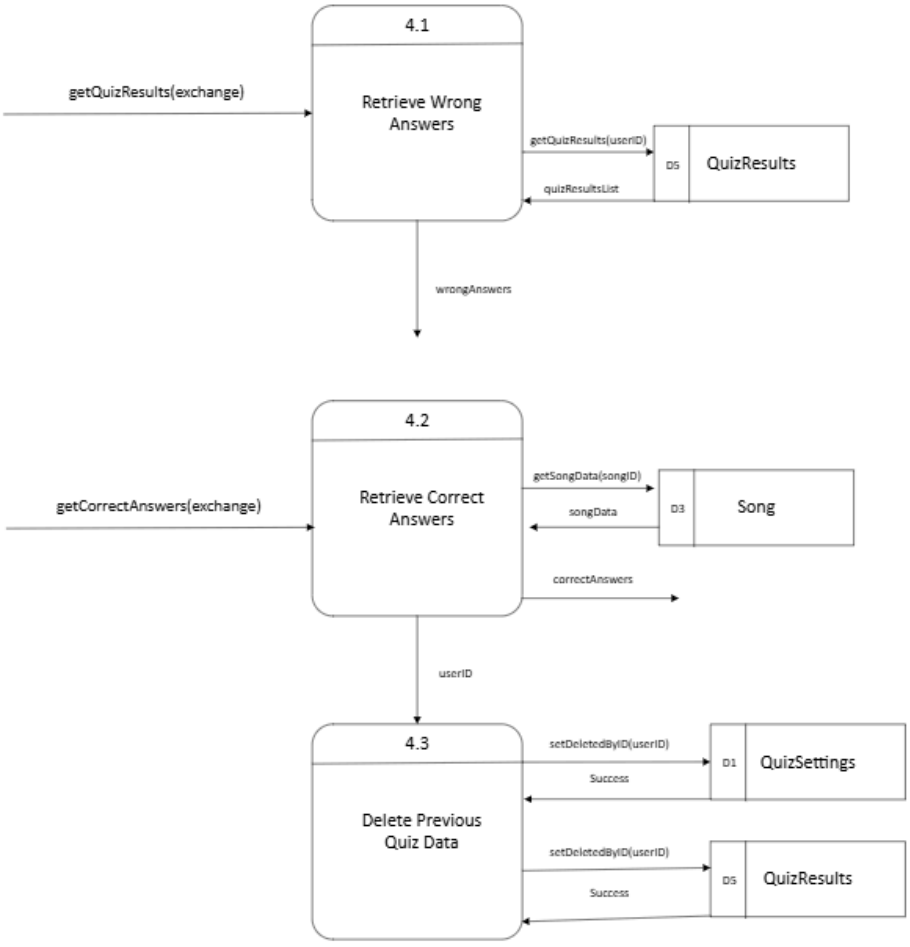


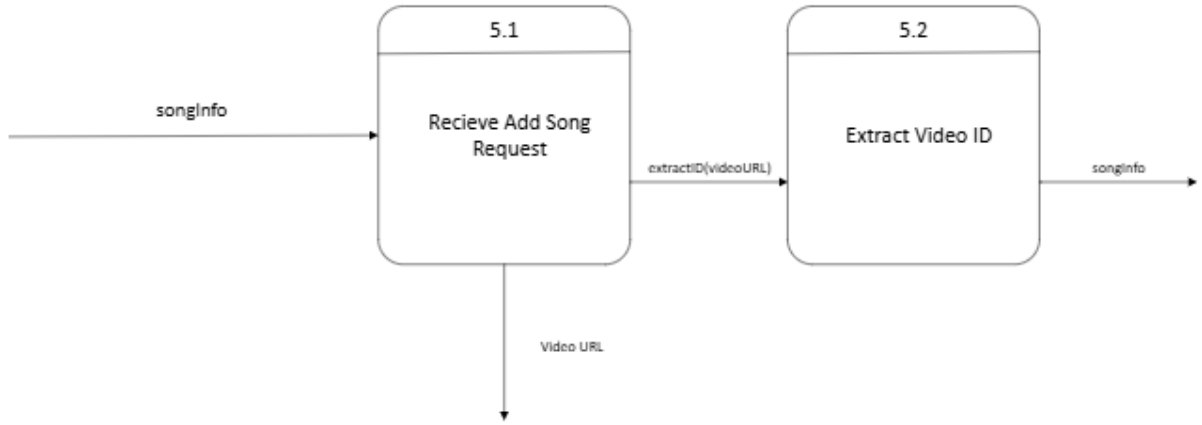


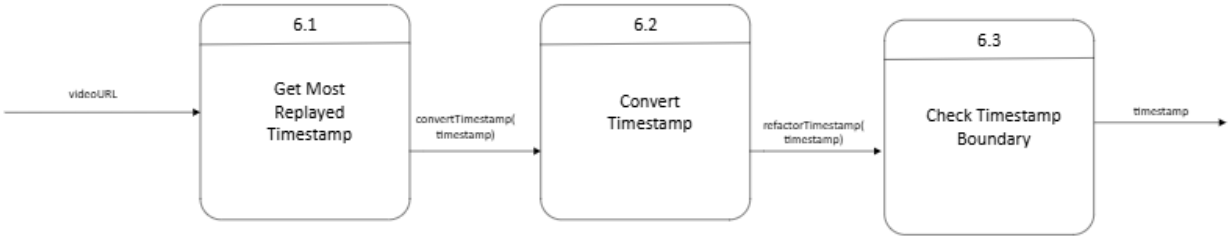

1.1
Check Previous Quiz
setQuizSettings(exchange)
setDeletedByID(userID)
Success
D1
QuizSettings
setDeletedByID(userID)
Success
D5
QuizResults
quizSettings
1.2
Analyze Settings
addQuizSettings(userID, playbackMethod, playbackDuration, numQuestions, playlistID)


2.1
Find Songs
nextQuestionSong(exchange)
getSongs(playlistID)
playlistSongList
D4
PlaylistSong
getThompsonSelection(playlistSongList, alreadySelectedIDs)
2.2
Thompson Sampling
2.3
Select Song
minSample
2.4
Determine Playback Method
selectedSong


3.1
Get Data
3.2
Compare Data
submitAnswer(exchange)
checkAnswers(quizData, songData)
addQuizResults(wrongAnswers)
Success
D5
QuizResults
getSongData(songID)
songData
updatePerfData(data)
Success
D3
Song
D2
StudentPerformance


4.1
Retrieve Wrong Answers
getQuizResults(exchange)
getQuizResults(userID)
quizResultsList
D5
QuizResults
wrongAnswers
4.2
Retrieve Correct Answers
getCorrectAnswers(exchange)
getSongData(songID)
songData
D3
Song
correctAnswers
userID
4.3
Delete Previous Quiz Data
setDeletedByID(userID)
Success
D1
QuizSettings
setDeletedByID(userID)
Success
D5
QuizResults

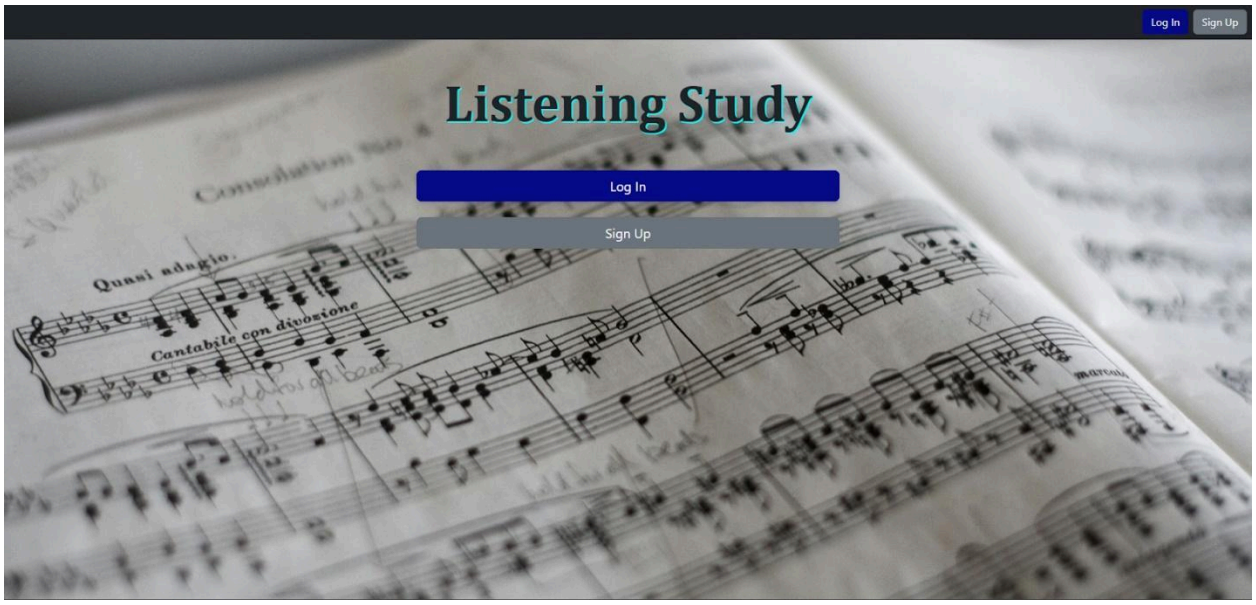

5.1
songInfo
Recieve Add Song Request
extractID(videoURL)
5.2
Extract Video ID
songInfo
Video URL


6.1
videoURL
Get Most Replayed Timestamp
convertTimestamp(timestamp)
6.2
Convert Timestamp
refactorTimestamp(timestamp)
6.3
Check Timestamp Boundary
timestamp

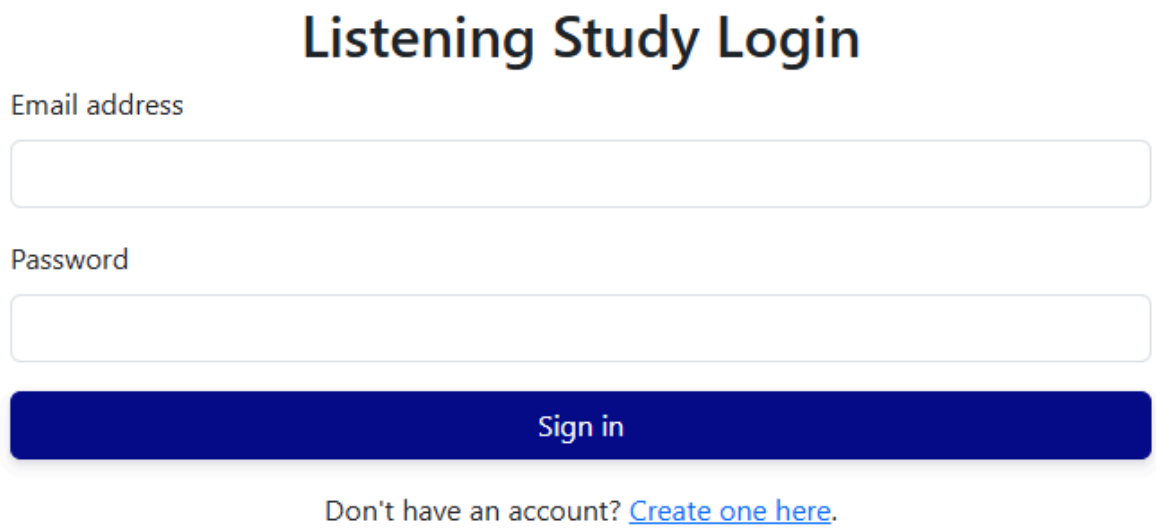
Graphical User Interface (GUI) Explanation

This section of the document details each user interface visible within the application. The section is broken down into five parts- shared views, student views, teacher views, moderator views, and administrator views.

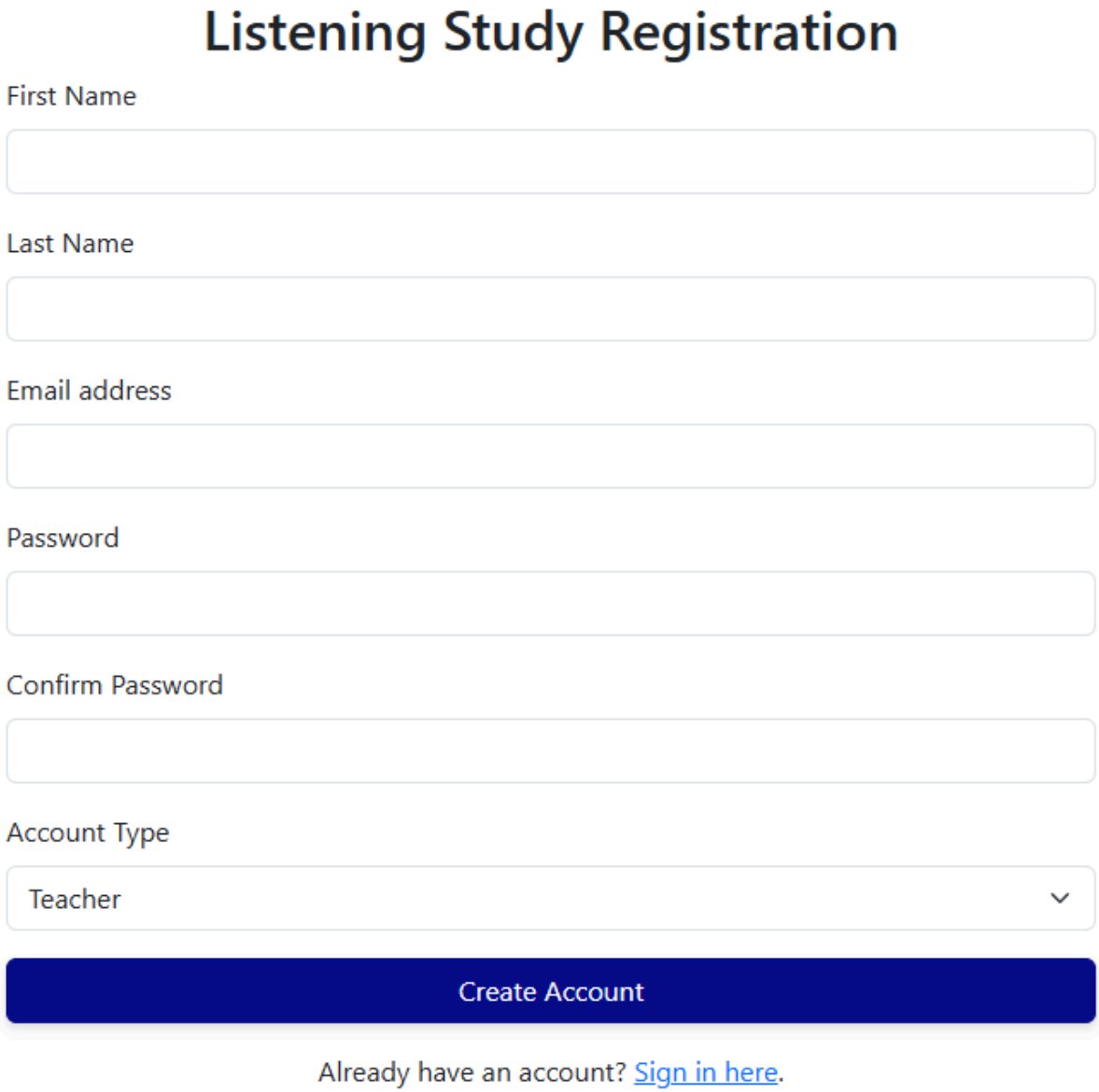
Shared Views



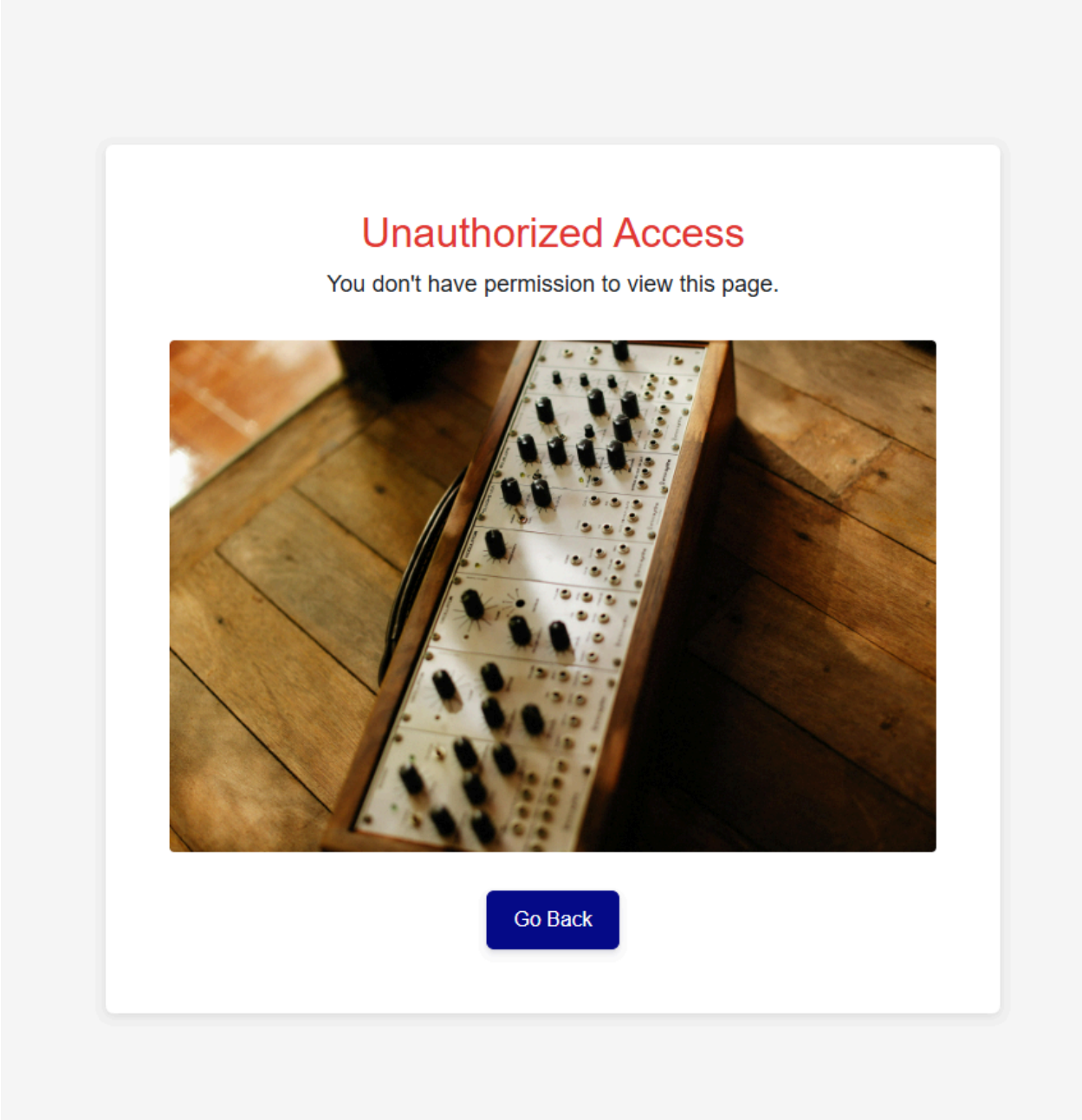
This is the home screen of the Listening Study Application. From this screen, the user has the option to click a login or sign-up button. Both pages redirect to separate screens (to be shown below). Existing users will login, while new users will sign up.



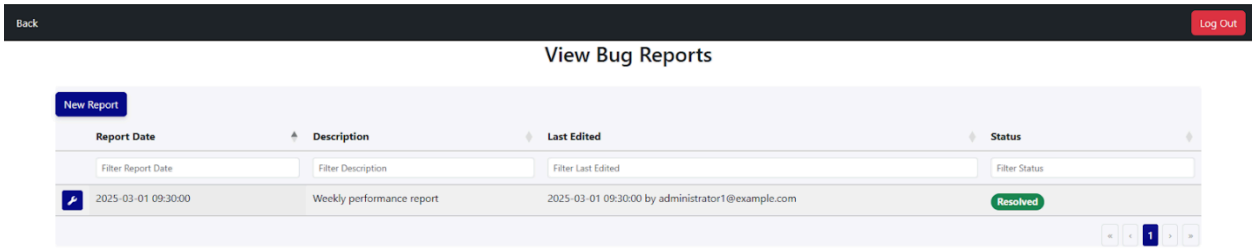
This is the login page. Existing users will navigate here to enter their account credentials. Even when directed to this page, users have the option to navigate to the register account page. If the user desires to register from this page instead, the user can click the hyperlink to “Create one here” to be redirected.



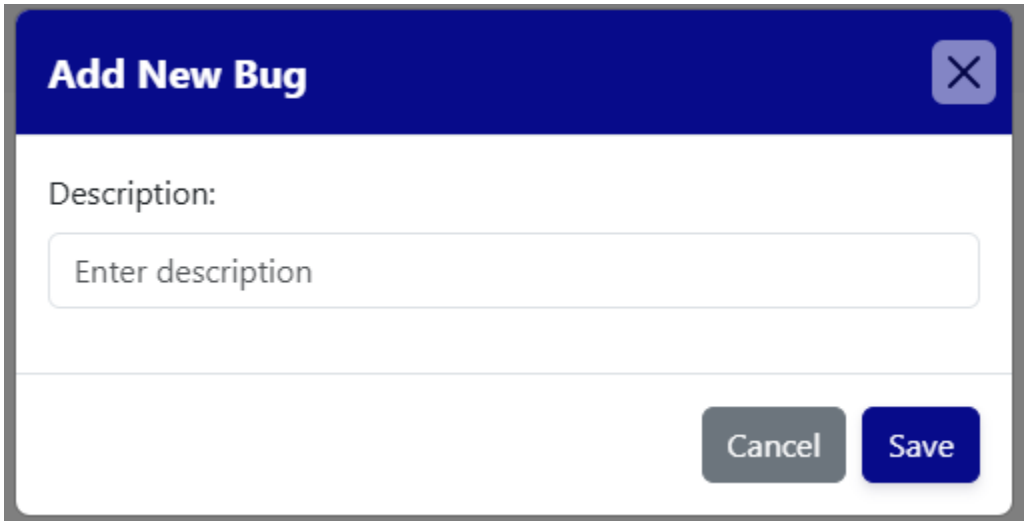
This is the registration page. New users will navigate here to create their account credentials. Even when directed to this page, users have the option to navigate to the login page. If the user desires to login from this page instead, the user can click the hyperlink to “Create one here” to be redirected.



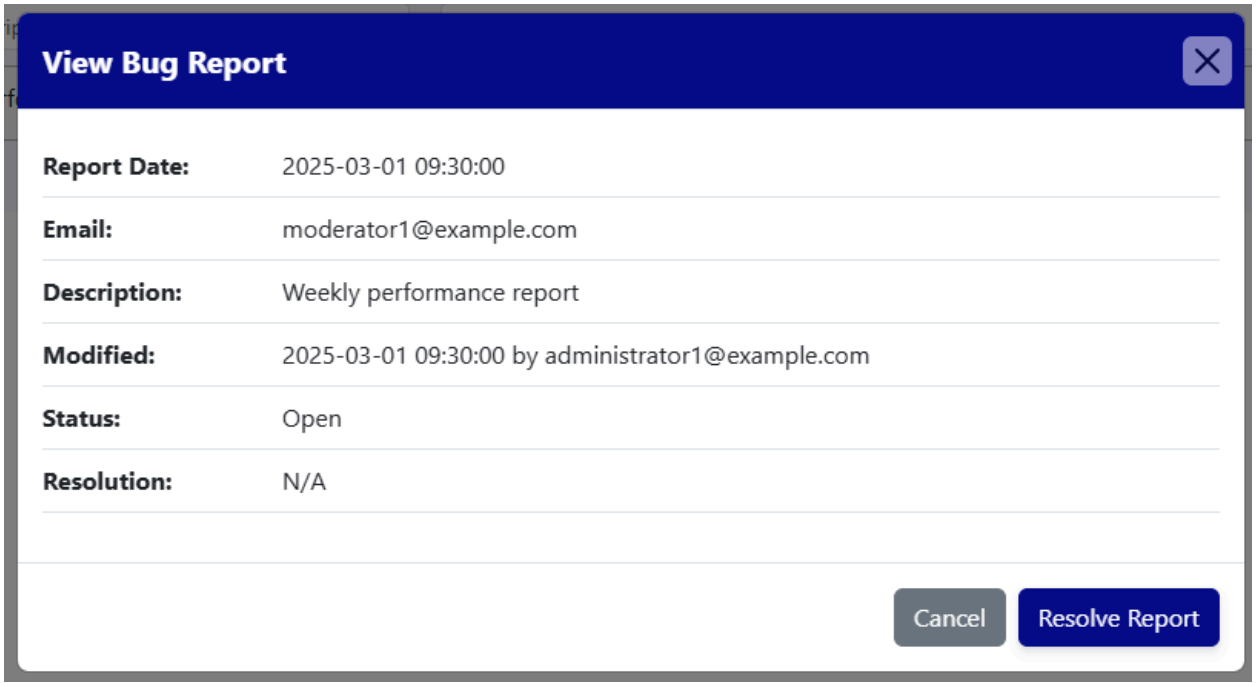
This is the page that a user sees when they try to access a URL that they are not authorized to see. The “Go Back” button redirects back to where the user entered the URL from originally (the page before trying to get to unauthorized). This is mainly to prevent URL fishing, users cannot really access this page just by clicking through.



This is the bug reports page, which is accessible to all user types. Any user can create a bug report, view their past reports, and mark a report as resolved (so that administrators no longer have to track it). The back button navigates to the previous page the user was on.

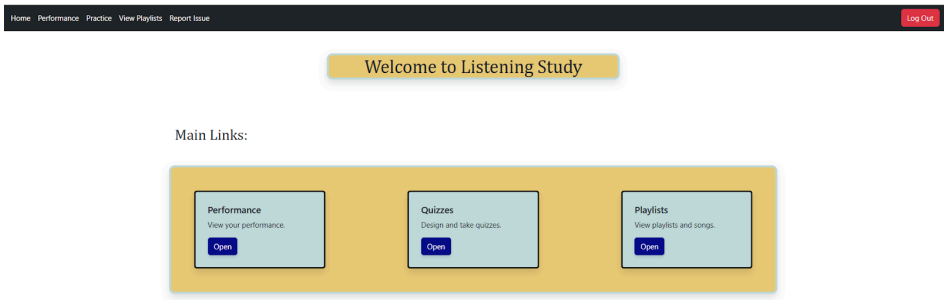


This is a popup modal that appears when the “New Report” button is clicked from the main page. Once the user enters a description and clicks the save button, the report is created.

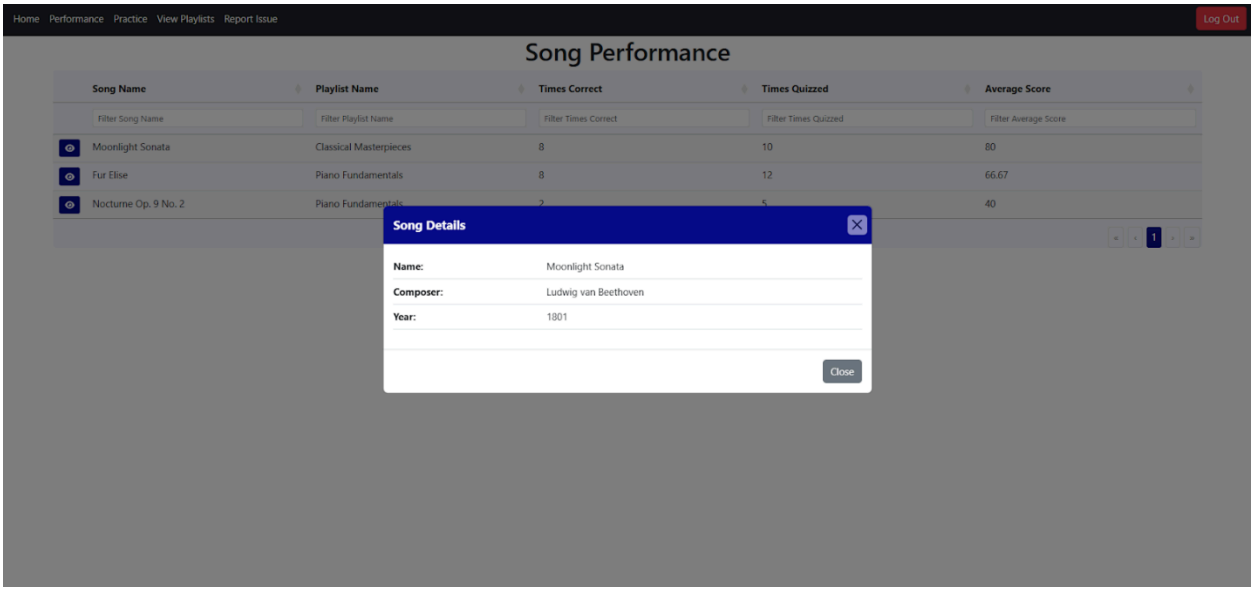


This is a popup modal that appears when the user clicks on the wrench in the row of the bug report page and then selects the option to view details. This shows all of the details pertaining to the specific bug report. If the report has not yet been resolved by the administrator or user, the "Resolve Report" button is visible at the bottom, for the user to resolve the report. Otherwise, the Resolution field will have a resolution, and the button will disappear.

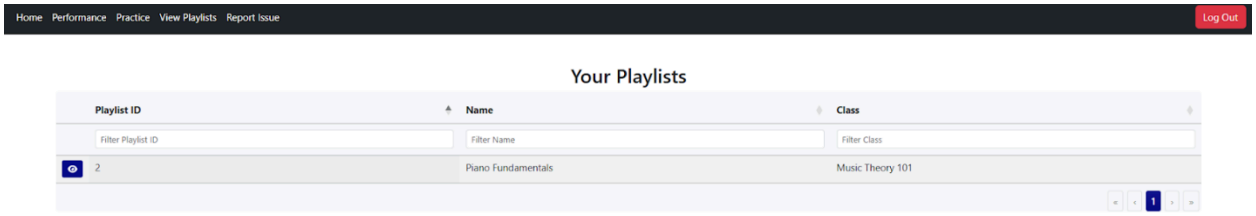
Student Views



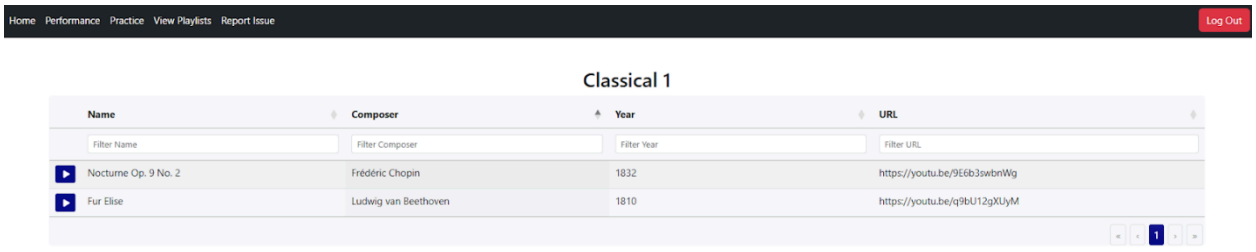
This is the student dashboard page. It highlights the main links of the student role. The links are identical to the navbar navigation.



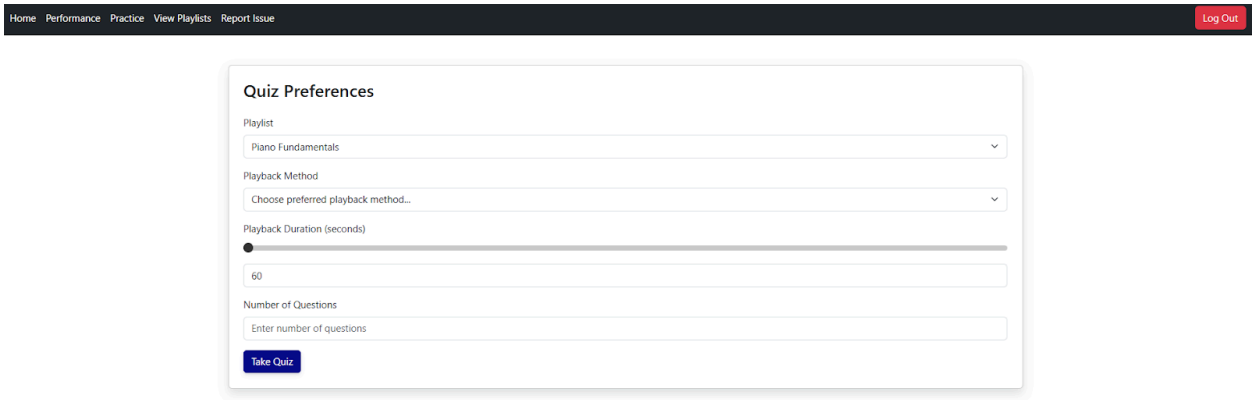
The performance page will show the student user song performance of songs they have practiced on. It will display the song name, the playlist the song is from, the times correct, the times they have been quizzed on it, and the average score of how often they are correct. When clicking the eye icon, a popup modal of song details shows up. This allows the user to study with the correct answers.



The view playlist page for the student user is a library of playlists. shows all playlists that they can practice. These playlists come from their class and the class name and name of the playlist, alongside the playlist ID will be shown. When pressing the eye icon it will go to the view of that particular playlist.

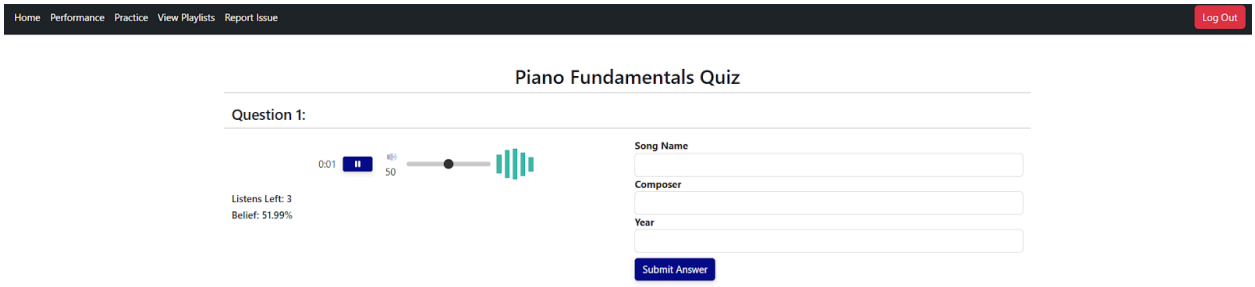


The individual playlist view page will show the contents of a playlist. The songs on a playlist will be shown. The details of a song include the name, composer, year, and the YouTube URL needed for playing the song snippet. Pressing the play icon will play the song snippet.

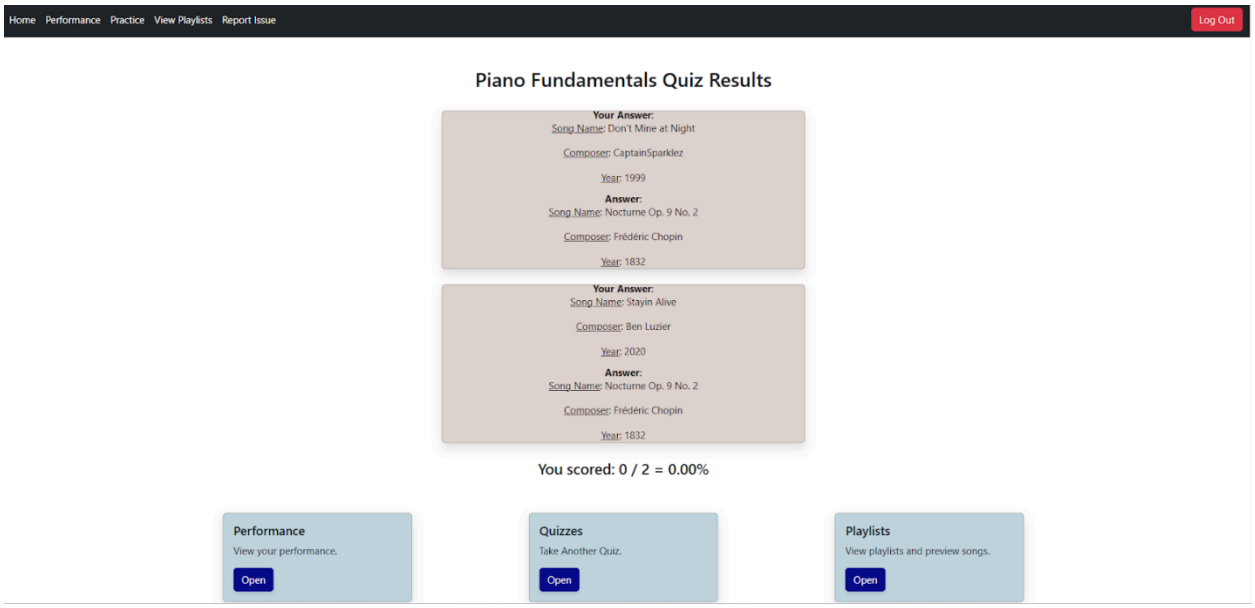


The practice page is where a student user will be able to set the parameters for a practice quiz. First they will choose a playlist from the dropdown. The playlists will be pre-populated from playlists they can view. The playback method will be set, which can be user defined, random, or the most replayed portion received when a song is added. The playback duration will be set in

seconds, and this is how long the song will play. The number of questions is also to be set. When the user is ready to start, they will press the start quiz button.

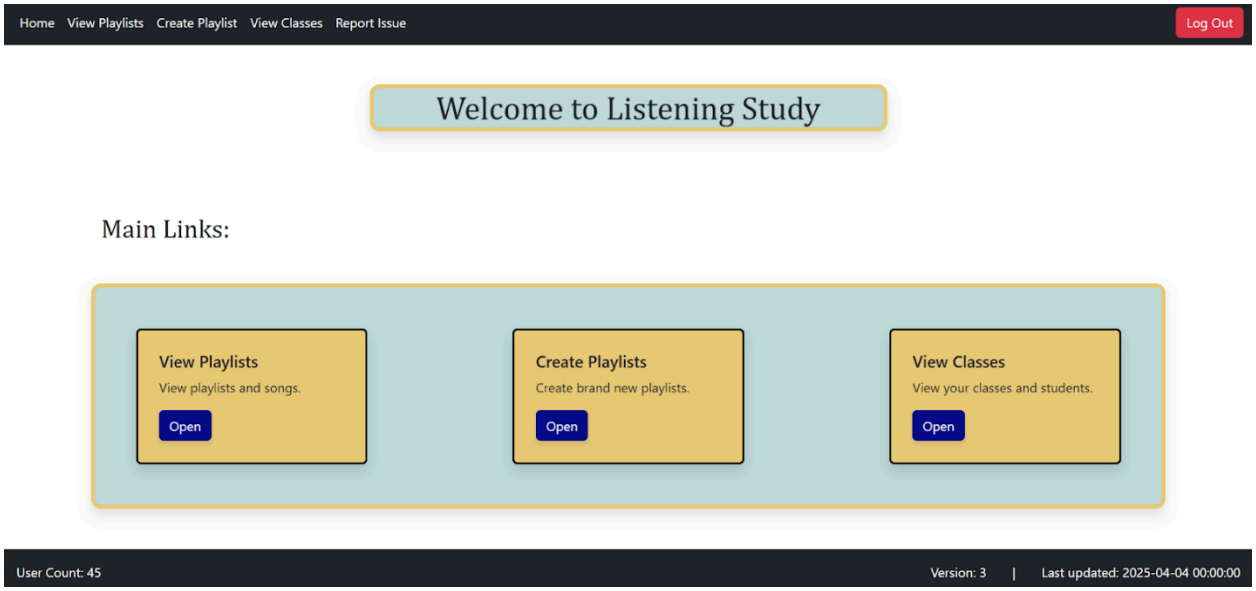


When the student user is taking the quiz they will see the question number, listens left, which is the amount of times they can listen to the song snippet, a volume meter, belief, which is the belief the system determines the user will get this question correct, and a user input area where they will need to input the song data for the question. Once they are finished, the submitted answer will move onto the next question until the quiz is finished.

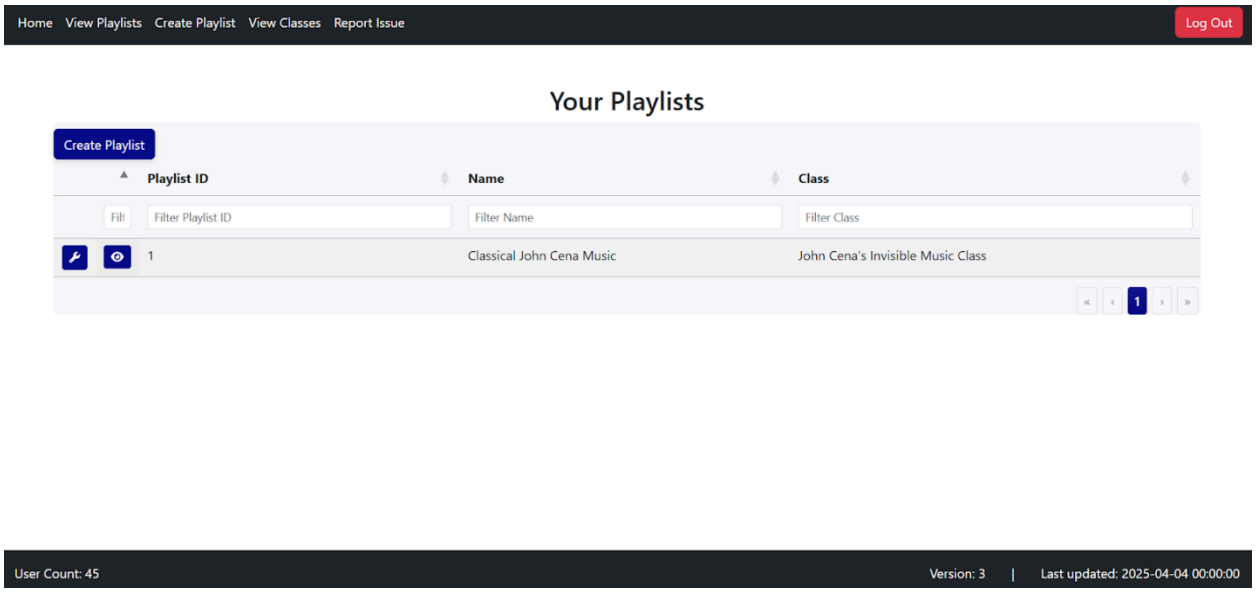


This is the quiz results page. The student user will be redirected to this page upon completion of a quiz. This page will show them cards of every answer that they got wrong. It will also show the correct answer and give them a score out of the questions they had. It will also prompt links similar to the student dashboard.

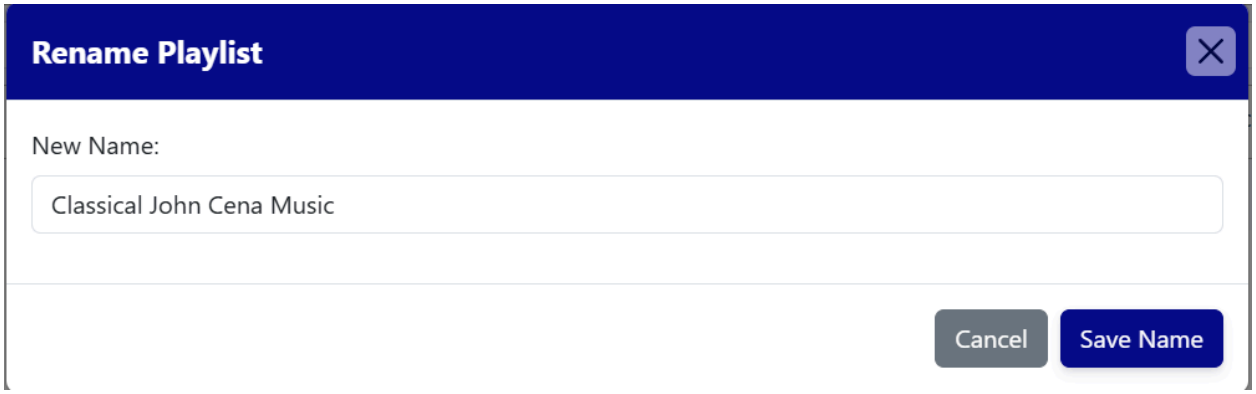
Teacher Views



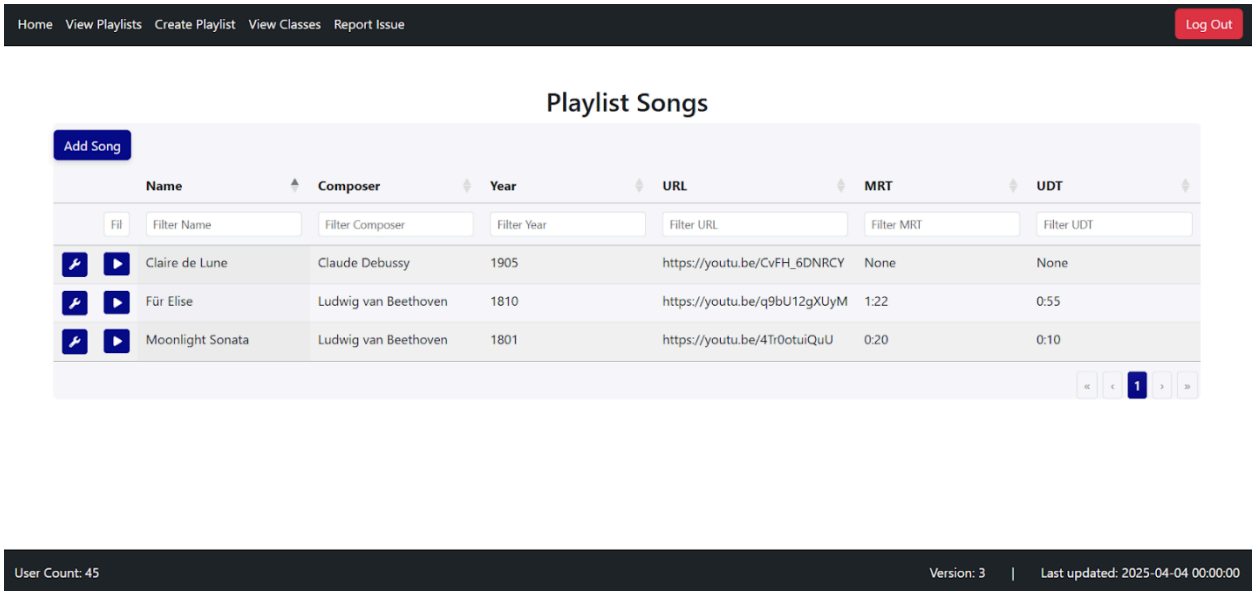
This is the teacher dashboard. It displays the main links for the teacher user. The navbar will navigate to the same links as displayed. All links will navigate to their appropriate page.



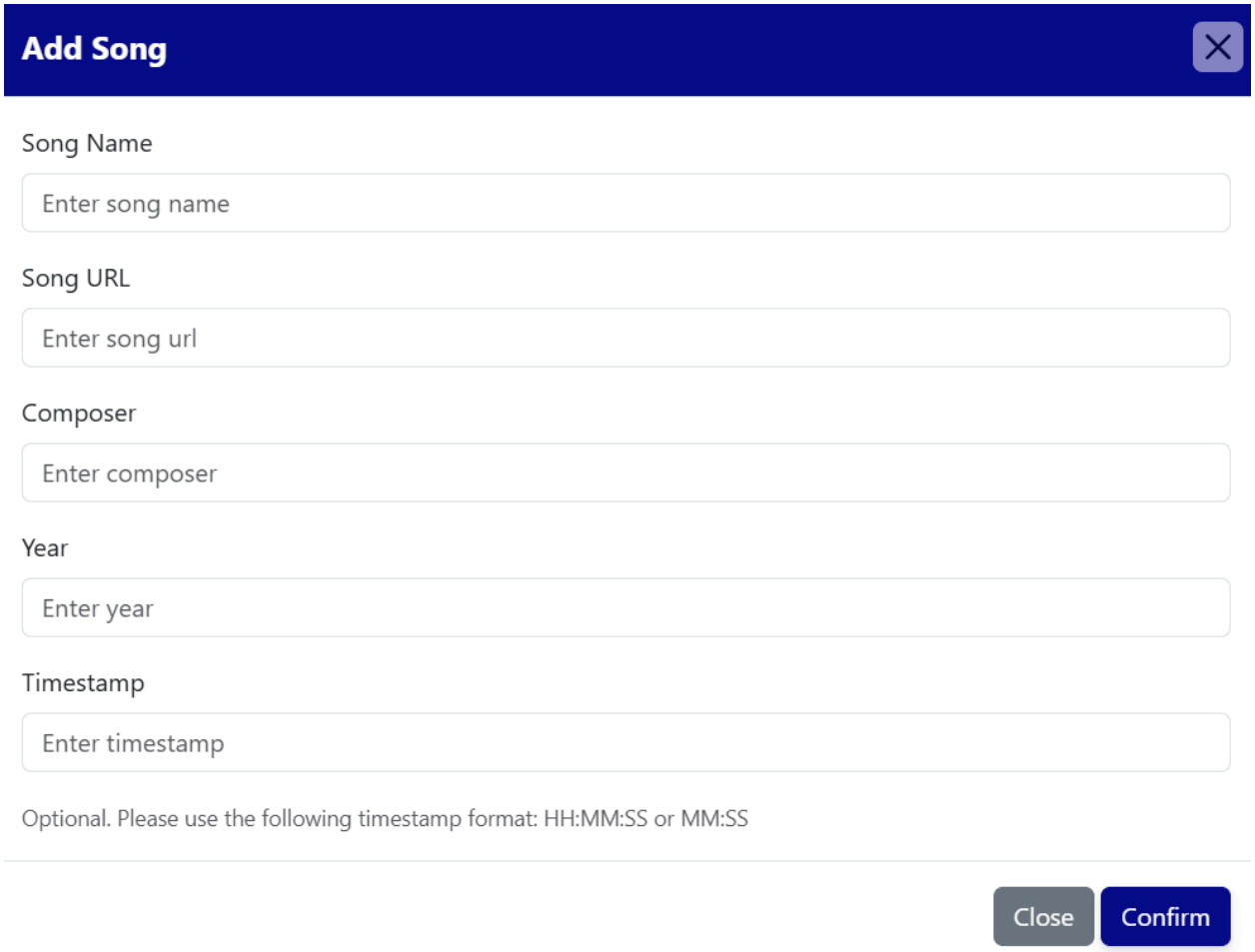
This is the view playlist page for the teacher. The create playlist button navigates to the create playlist page. It displays the all-playlist IDs, Names, and class names of all playlists the teacher has created. The wrench icon allows the user to rename a playlist or delete it. The eye icon navigates to the view of songs on a playlist.



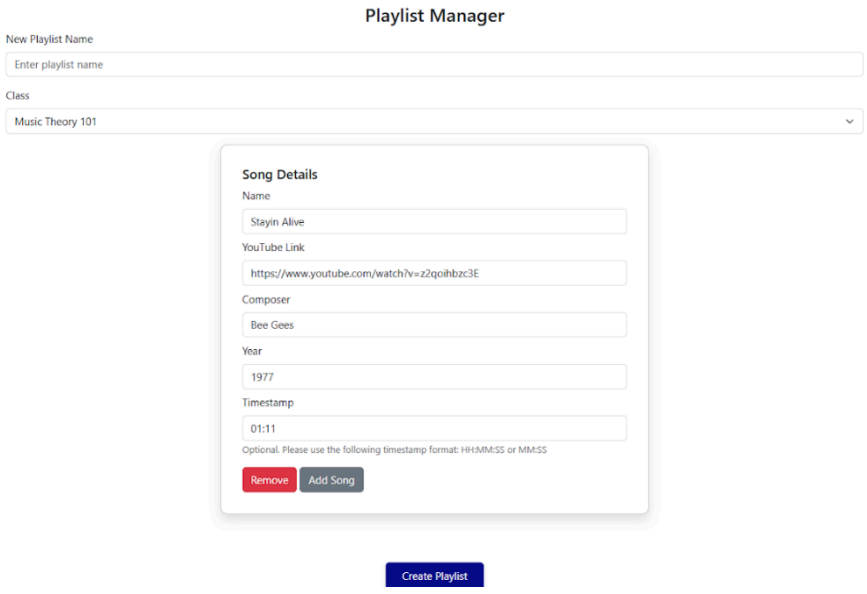
When pressing edit song, a rename playlist modal appears it allows the user to save a new name of the playlist.

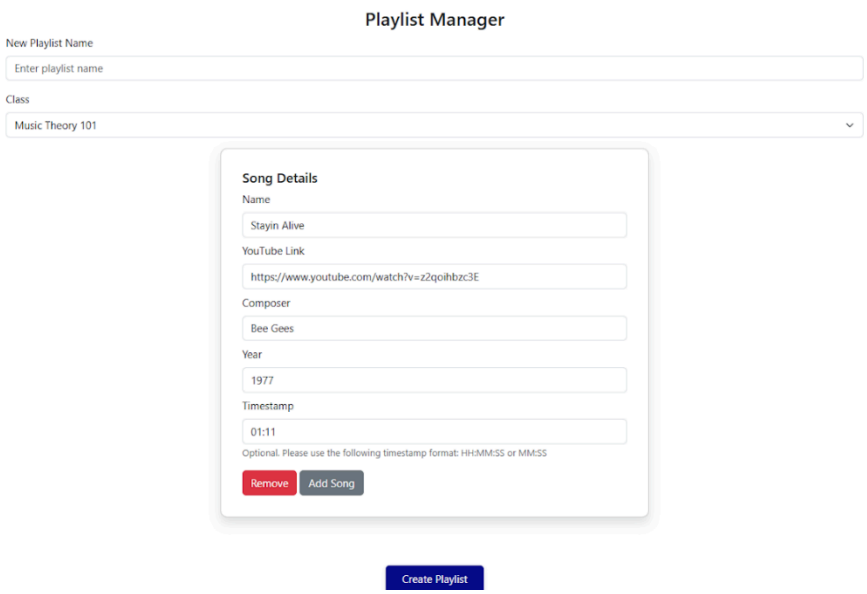


When pressing the eye on the playlist view page, the user can then view the songs on that playlist. The user can also view details of the song, those being name, composer, year, URL, MRT (most replayed timestamp), and UDT (user defined timestamp). The wrench icon prompts the user to change the details or to delete a song. The add song button prompts the add song modal.

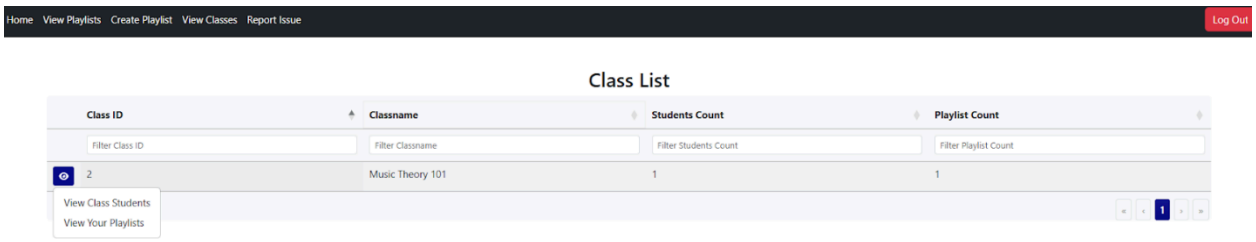


The add song modal allows the teacher user to enter in the appropriate details of a song, this is also almost identical to the edit song details modal.

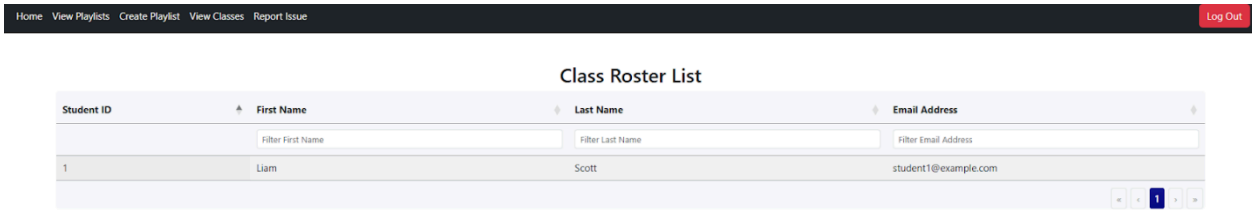




Inside the create playlist tab, the teacher user is able to add songs to a playlist. They must give a name to the playlist and choose a class for it. The user must also enter the Name of the song, the YouTube link, the composer, year, and a user-defined timestamp for quizzes. They can add a song or remove one if it is not needed. When finished, the create playlist button will create the entire playlist for that class, and any student will be able to access it.

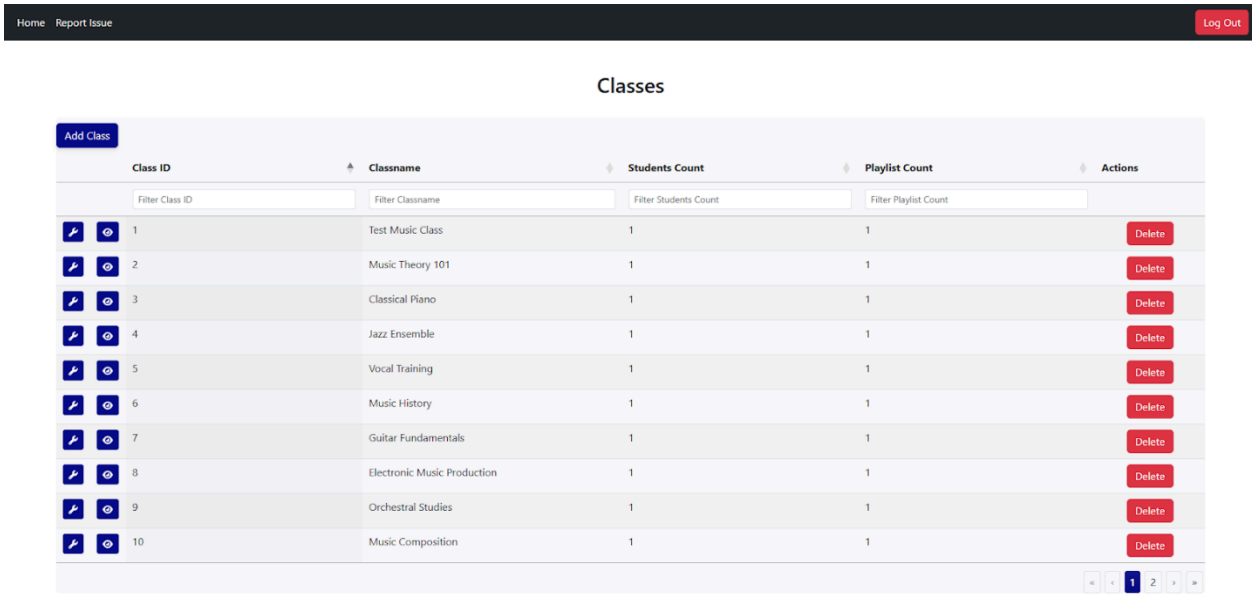


View classes will show the teacher a list of every class they are teaching. Additionally, it will show them the class ID and name of the class along with the number of students and playlists pertaining to that particular class. Pressing the eye icon next to a class will prompt two links. The view your playlists link will redirect to the playlist page and the view class student's icon will redirect to the class roster of that class.



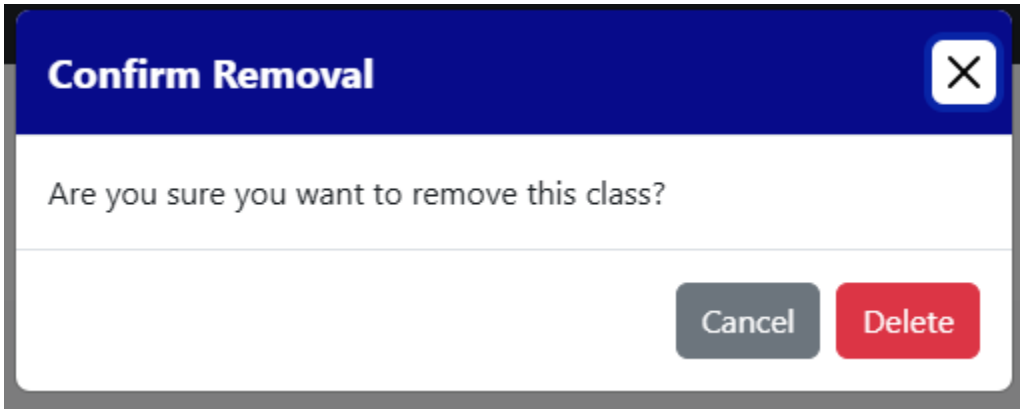
When the user clicks the view class students, it will navigate them to the class roster page. It will display every student in that class. It will also show the students ID, first name, last name, and email address.

Moderator Views



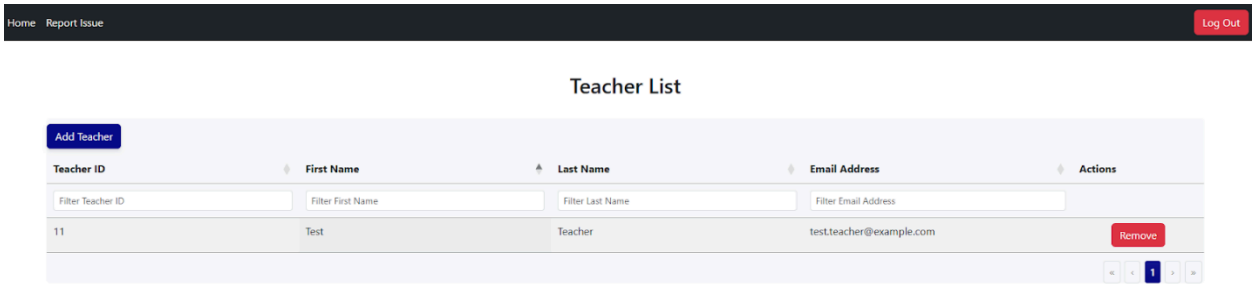
This is the landing page for a moderator account. Here, the moderator can view, sort, and filter all classes. The moderator can create and delete classes and can also use the wrench and view icons (which will be explained below).

This is the add class modal popup that can be accessed by clicking the “Add Class” button on the landing page for a moderator. Here, a moderator can create the class name and add existing teachers to the class.



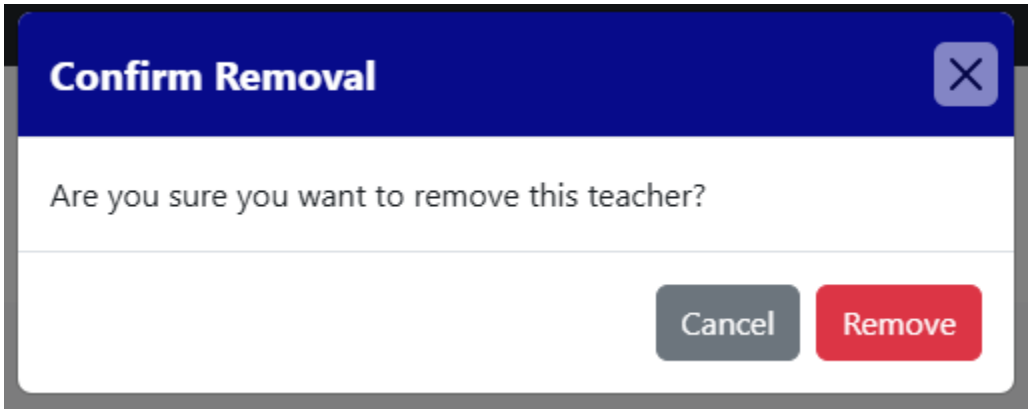
This is a modal popup from the Moderator page. This can be accessed from clicking the delete button in the action column. Once the user clicks “Delete”, the class will be removed.

This is a modal popup from the Moderator page, which allows the moderator to rename a class. Clicking Save Name will update the name.

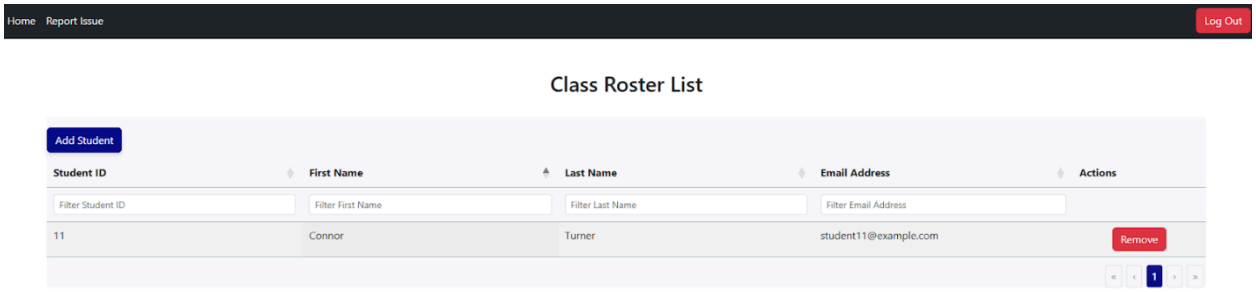


This is the Teacher List view for a class. This can be accessed by clicking the view column from the moderator dashboard, then selecting the “View Teacher” option from the dropdown menu. Here, the moderator can add or remove teachers.

This is the add teacher popup from the teacher list page. Here, a moderator can add a teacher to an already existing class by clicking the “Confirm” button.

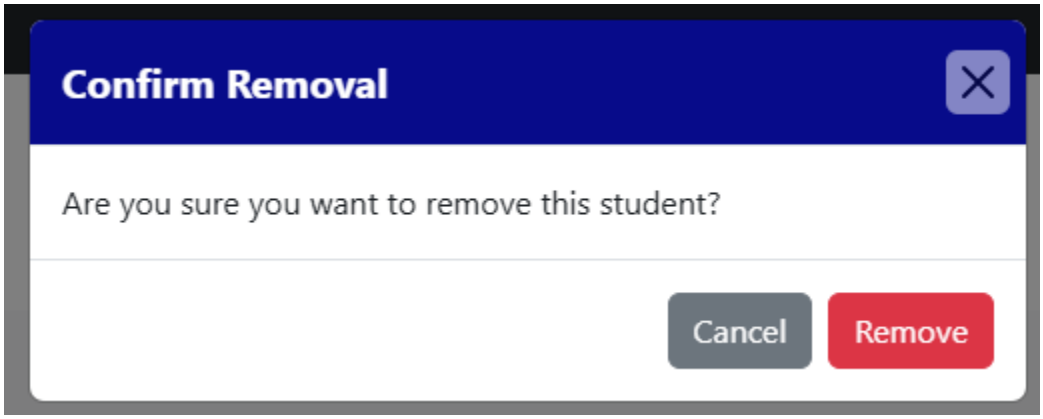


This is the remove teacher popup from the teacher list page. Here, a moderator can remove a teacher in an already existing class by clicking the “Remove” button.



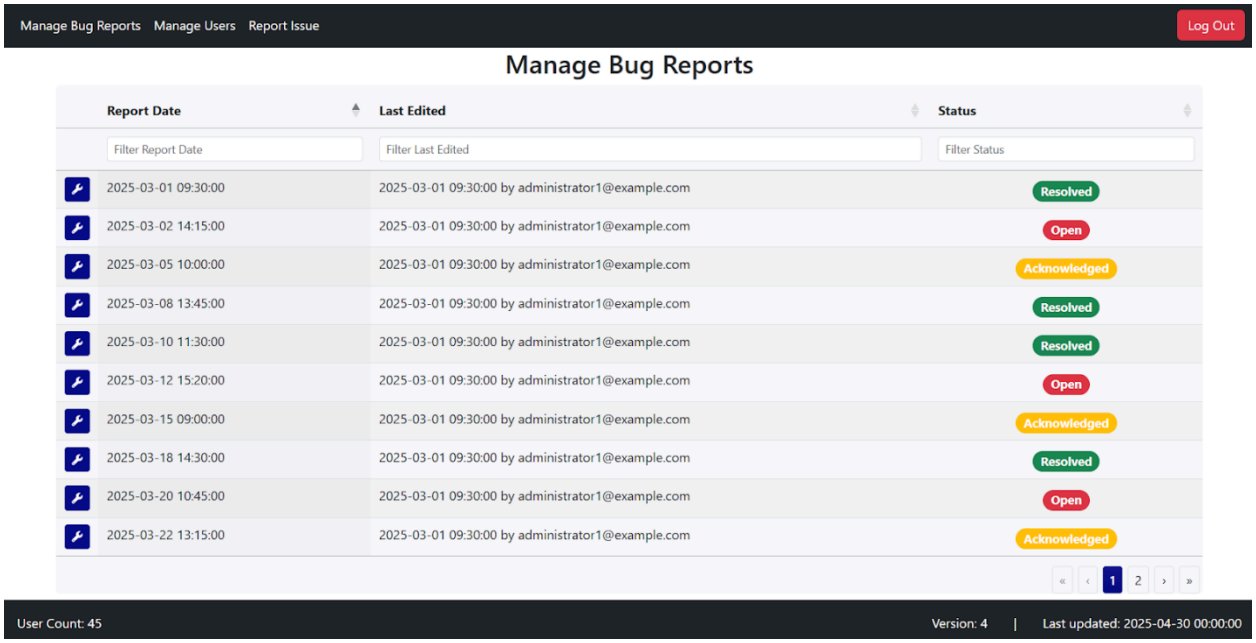
This is the Teacher List view for a class. This can be accessed by clicking the view column from the moderator dashboard, then selecting the "View Teacher" option from the dropdown menu. Here, the moderator can add or remove students from the class roster list.

This is the add student popup from the class list page. Here, a moderator can add a student to an already existing class by clicking the "Confirm" button.

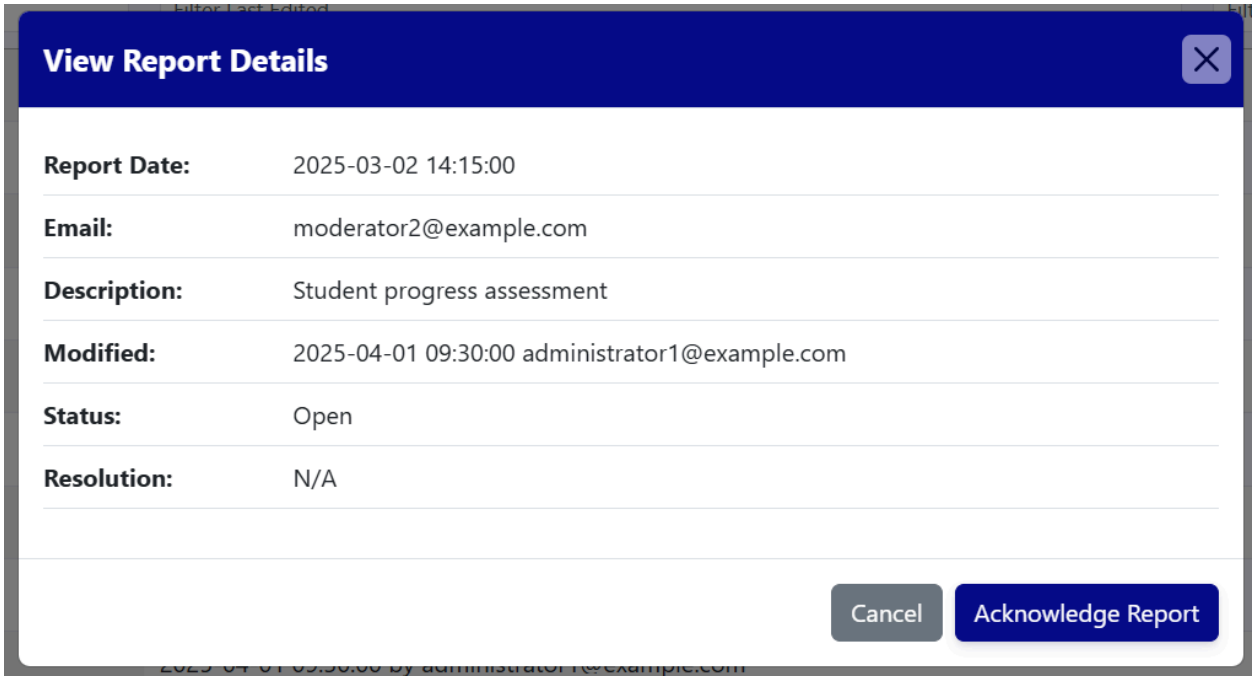


This is the remove student popup from the class list page. Here, a moderator can remove a teacher in an already existing class by clicking the “Remove” button.

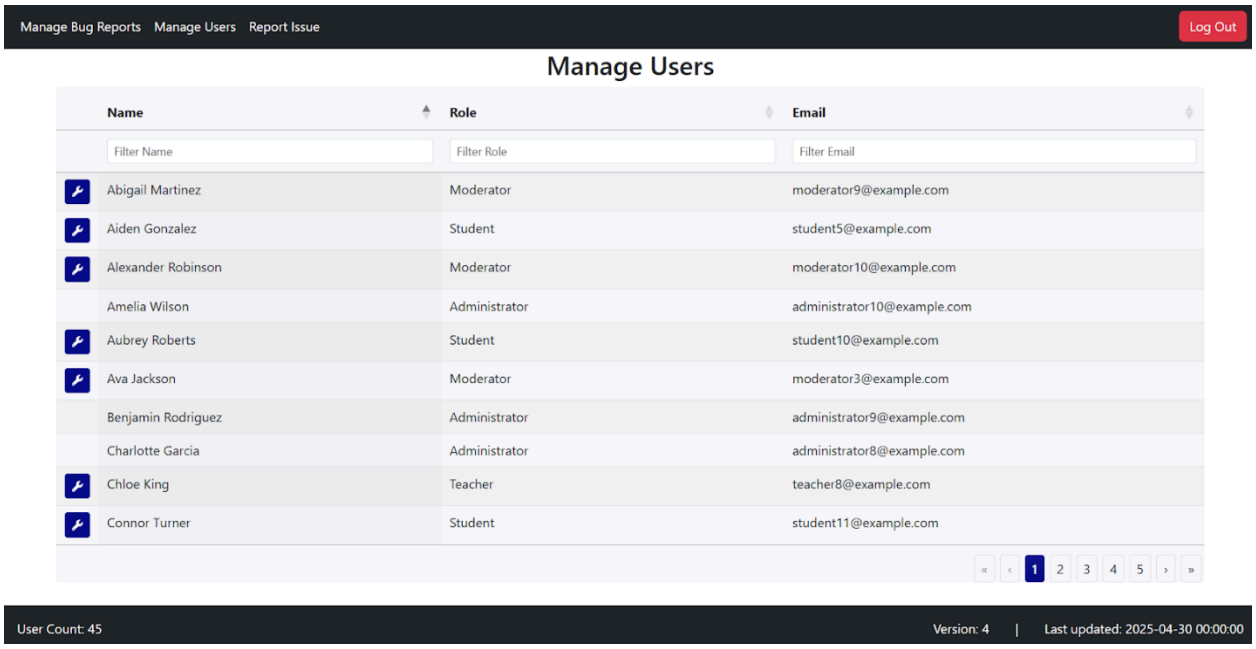
Administrator Views



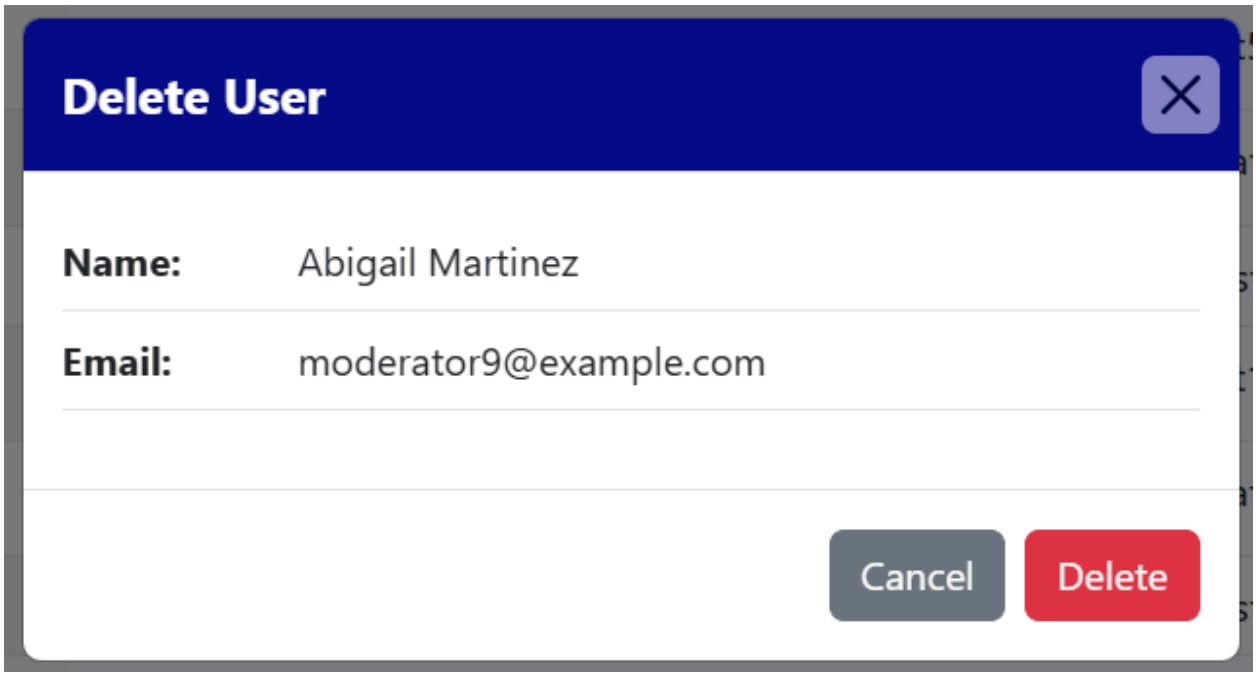
This is the Manage Bug Reports view, which is the screen that is loaded when an administrator logs in successfully. This page can also be accessed by clicking “Manage Bug Reports” from the navbar. Here, the administrator can view, sort, and filter all bug reports for the application (including ones already resolved).



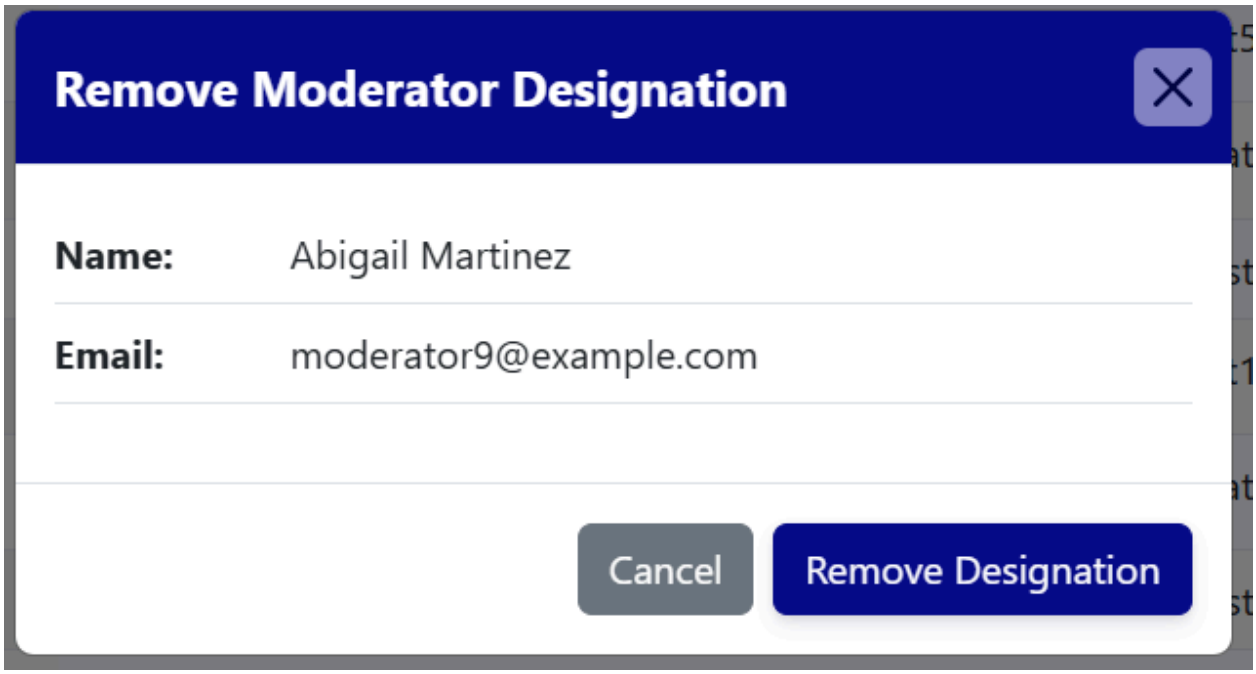
This is a popup on the Manage Bug Reports view, which allows an administrator to view all of the report details. This can be accessed by clicking the wrench column for the desired row. If the report has a status of Open, the "Acknowledge Report" button will be visible (as shown on the screen). If the status is Acknowledged, the administrator will see a "Resolve Report" button, used to resolve a bug report once it has been solved.



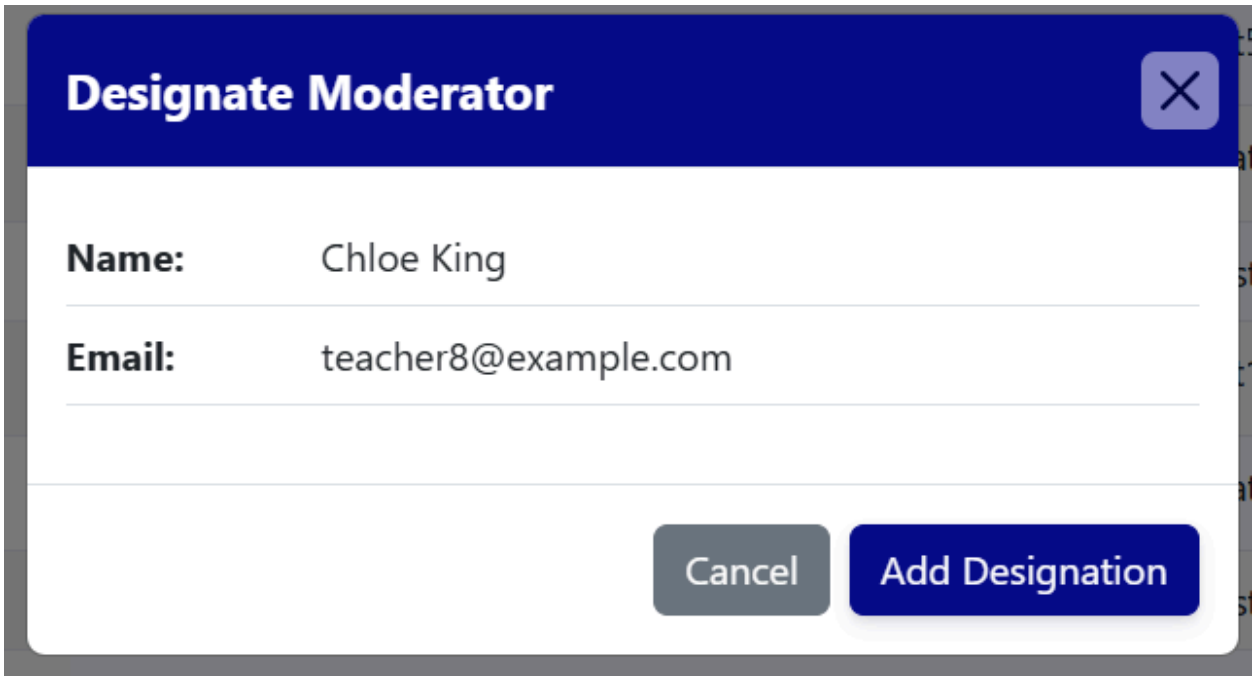
This is the Manage Users View for an administrator, which can be found by clicking the appropriate option in the navbar. This allows the administrator to view a list of users, and make edits to students, teachers, and moderators. Administrators do not have a wrench column, since they cannot make changes to each other.



This is the Delete User popup modal from the Manage Users View. This allows the administrator to delete students, teachers, and moderators. This can be accessed from the wrench column of the desired row.



This is the Remove Moderator Designation popup from the Manage Users View. Administrators can remove a moderator's designation, changing their user type to a teacher by clicking the "Remove Designation" button



This is the Designate Moderator popup modal from the Manage Users Views. Administrators can add a moderator designation to any teacher account by changing their account type when clicking the “Add Designation” button.

Backend API Explanation

The backend of the application is broken into several file types, each with their own respective package. The first is configuration, which is used for database and server connections.

Configurations

DatabaseConfiguration

Configuration class for database setup

Method Details

connect

public static void connect() throws Exception

Creates a connection to the database

Throws:

`ClassNotFoundException` - If the database driver can't be found

`Exception` - If the connection to the database fails

getConnection

public static Connection getConnection()

Creates connection access to repository classes

Returns:
Connection to the database

HttpServerConfiguration

Configuration class for HTTP server

Method Details

startServer

public static void startServer() throws IOException

Starts the HTTP server for the application

Throws:
`IOException` - If the server can't be started

createAPIEndpoints

private static void createAPIEndpoints(HttpServer server)

Creates all of the API endpoints used within the application.

Parameters:
`server` - is the created server

Utilities

The next file type is the "Utils" file type, short for utilities. These files are for utilities used in certain files in the application that are not related to setup or configuration.

ApplicationUtil

Configuration class for application properties

Method Details

getInstance

public static ApplicationUtil getInstance()

Creates a retrievable singleton instance for other files

Returns:
A retrievable instance of property configuration

getDbUsername

public String getDbUsername()

Returns the dbusername

Returns:
String containting dbusername

getDbPassword

public String getDbPassword()

Returns the dbpassword

Returns:
String containting dbpassword

getDbURL

public String getDbURL()

Returns the URL

Returns:
String containting dbURL

getServerPort

public Integer getServerPort()

Returns the server port

Returns:
String containting the server port

CookieUtil

Configuration class for cookies.

Method Details

getCookie

private static String getCookie(HttpExchange exchange, String name)

Retrieves the value of a specific cookie from the HTTP exchange.

Parameters:

`exchange` - The HTTP exchange containing the request data

`name` - The name of the cookie to retrieve

Returns:

The value of the cookie if found; otherwise, null

getCookieSessionID

public static String getCookieSessionID(HttpExchange exchange)

Retrieves the value of the "SESSIONID" cookie from the HTTP exchange.

Parameters:

`exchange` - The HTTP exchange containing the request data

Returns:

The session ID if present; otherwise, "not found"

hasValidSession

public boolean hasValidSession(HttpExchange exchange)

Checks if the request has a valid session cookie.

Parameters:

`exchange` - The data from the API request

Returns:

true if a valid session exists, false otherwise

Handlers

The next file type utilized in the Listening Study Application is a handler. Handlers are responsible for catching and sending API calls between the front-end and the rest of the backend. There are also handlers configured for static files and rendered templates. Handlres are implemented on a page/feature basis. They are one-to-one with the service files. The BaseHandler file contains files used throughout handler files.

BaseHandler

Base Handler class that contains methods to be used in other handlers.

Constructor Details

BaseHandler

public BaseHandler()

Class constructor to intialize service and util files

Method Details

isRequestAuthorized

protected boolean isRequestAuthorized(HttpExchange exchange) throws IOException

Validates if the request has a valid session and matching role.

Parameters:

`exchange` - The HTTP exchange containing the request

Returns:

true if the request is authorized; false otherwise

Throws:

`IOException` - If redirection fails

redirectToRoot

protected void redirectToRoot(HttpExchange exchange) throws IOException

Helper to redirect the request to the application root URL.

Parameters:

`exchange` - The HTTP exchange to redirect

Throws:

`IOException` - If an I/O error occurs during redirection

redirectToUnauthorized

protected void redirectToUnauthorized(HttpExchange exchange) throws IOException

Helper to redirect the request to the application unauthorized page's URL.

Parameters:

`exchange` - The HTTP exchange to redirect

Throws:

`IOException` - If an I/O error occurs during redirection

extractRoleFromUrl

protected String extractRoleFromUrl(HttpExchange exchange)

Helper to extract the role from the browser URL path. For example, from "/administrator/dashboard" or "localhost:8080/administrator/dashboard" it will extract "administrator".

Parameters:

`exchange` - The HTTP exchange containing the request

Returns:

The extracted role (student, teacher, moderator, administrator) or null if not found

sendResponse

protected void sendResponse(HttpExchange exchange, String responseContent, String responseType) throws IOException

Handles sending the response to the frontend.

Parameters:

`exchange` - The data from the API request

`responseContent` - Parameter used to determine the HTTP code to be returned

`responseType` - Checks if the request is static ("Thymeleaf") or dynamic (all other API calls).

Throws:

`IOException` - If HTTP request send or receive operations fail

AdministratorHandler

Handler class for processing API requests related to administrator actions.

Constructor Details

AdministratorHandler

public AdministratorHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

BugReportHandler

Handler class for processing API requests related to the bug reports page.

Constructor Details

BugReportHandler

public BugReportHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

CreatePlaylistHandler

Handler class for processing API requests related to the teacher's playlist creation page.

Constructor Details

CreatePlaylistHandler

public CreatePlaylistHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:
`handle` in interface `HttpHandler`
Parameters:
`exchange` - The data from the API request
Throws:
`IOException` - If HTTP request send or receive operations fail

LoginHandler

Handler class for processing API requests related to user login.

Constructor Details

LoginHandler

public LoginHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

MetadataHandler

Handler class for processing API requests related to metadata.

Constructor Details

MetadataHandler

public MetadataHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

ModeratorHandler

Handler class for processing API requests related to moderator data.

Constructor Details

ModeratorHandler

public ModeratorHandler()

Class constructor to initialize moderator service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

QuizResultsHandler

Handler class for processing API requests related to quiz results.

Constructor Details

QuizResultsHandler

public QuizResultsHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

RegistrationHandler

Handler class for processing API requests related to user registration.

Constructor Details

RegistrationHandler

public RegistrationHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

SetQuizHandler

Constructor Details

SetQuizHandler

public SetQuizHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

SnippetHandler

Handler class for processing API requests related to playing a song snippet.

Constructor Details

SnippetHandler

public SnippetHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

StaticFileHandler

Handler class for serving static files back to the frontend.

Constructor Details

StaticFileHandler

public StaticFileHandler()

Class constructor to initialize a map containing acceptable content types

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes static requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the static request

Throws:

`IOException` - If static request send or receive operations fail

setContentType

private void setContentType(HttpExchange exchange, String path)

Sets the content type of the file to be returned to the frontend.

Parameters:

`exchange` - The data from the static request

`path` - The path used for gathering the file extension

getFileExtension

private String getFileExtension(String path)

Returns the file extension type based on the path provided

Parameters:

`path` - The path used for gathering the file extension

Returns:

String containing the file extension

StudentPerformanceHandler

Handler class for processing API requests related to viewing the student performance page.

Constructor Details

StudentPerformanceHandler

public StudentPerformanceHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:
`handle` in interface `HttpHandler`
Parameters:
`exchange` - The data from the API request
Throws:
`IOException` - If HTTP request send or receive operations fail

TakeQuizHandler

Handler class for processing API requests related to the quiz page.

Constructor Details

TakeQuizHandler

public TakeQuizHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or receive operations fail

TeacherClasslistHandler

Handler class for processing API requests related to teacher classlist data.

Constructor Details

TeacherClasslistHandler

public TeacherClasslistHandler()

Class constructor to initialize service classlist file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If HTTP request send or recieve operations fail

TeacherLibraryHandler

Constructor Details

TeacherLibraryHandler

public TeacherLibraryHandler()

Class constructor to initialize service library file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:
`handle` in interface `HttpHandler`
Parameters:
`exchange` - The data from the API request
Throws:
`IOException` - If HTTP request send or receive operations fail

TeacherRosterHandler

Constructor Details

TeacherRosterHandler

public TeacherRosterHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:
`handle` in interface `HttpHandler`
Parameters:
`exchange` - The data from the API request
Throws:
`IOException` - If HTTP request send or receive operations fail

TeacherSongHandler
Handler class for processing API requests related to the song page from teacher view.

Constructor Details

TeacherSongHandler

public TeacherSongHandler()

Class constructor to initialize service file

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes HTTP requests from frontend to proper service method

Specified by:
`handle` in interface `HttpHandler`
Parameters:
`exchange` - The data from the API request
Throws:
`IOException` - If HTTP request send or receive operations fail

ThymeleafHandler

Handler class for returning dynamic thymeleaf templates and fragments.

Constructor Details

ThymeleafHandler

public ThymeleafHandler()

Class constructor to create the templating engine

Method Details

handle

public void handle(HttpExchange exchange) throws IOException

Handles/routes dynamic thymeleaf requests from frontend to proper service method

Specified by:

`handle` in interface `HttpHandler`

Parameters:

`exchange` - The data from the API request

Throws:

`IOException` - If template request send or recieve operations fail

render

private String render(String template, org.thymeleaf.context.Context context)

Helper function to render requested templates

Parameters:

`template` - The template name to be used

`context` - The data from the API request

Returns:

String The processed template, or a messsage showing failed template rendering

Services

The next file type used in the Listening Study Application is the service file type. Services act as a middleman between the Handler and Repository files. Services take in the input from the handler, do any business logic, and then call one or multiple repository files. Services are one-to-one with handler files, also being implemented on a page/feature basis. The BaseService file contains methods used throughout all of the other service files.

BaseService

Base Service class that contains methods to be used in other service classes.

Method Details

getSessionUserID

public int getSessionUserID(HttpExchange exchange) throws IOException

Returns the user ID retrieved from the session.

Parameters:

`exchange` - The HTTP exchange containing the request and cookies

Returns:

user ID retrieved using the session ID cookie

Throws:

`IOException` - If reading the session cookie fails

compareRoles

public boolean compareRoles(HttpExchange exchange, String urlRole) throws IOException

Compares the user's role (retrieved from session) with the role extracted from the URL.

Parameters:

`exchange` - The HTTP exchange containing the request and cookies

`urlRole` - The role extracted from the URL path (e.g., "administrator", "student")

Returns:

true if the user's role matches the role in the URL; false otherwise

Throws:

`IOException` - If reading the session cookie fails

getParametersList

public List<Map<String,Object>> getParametersList(HttpExchange exchange) throws IOException

Takes in JSON body (POST, PATCH, DELETE) from request and returns multiple JSON objects

Parameters:

`exchange` - The data from the API request

Returns:

List of JSON objects to be used

Throws:

`IOException` - If formatting operations fail

getParameters

public Map<String,Object> getParameters(HttpExchange exchange) throws IOException

Takes in JSON body (POST, PATCH, UPDATE) from request and returns a single JSON object

Parameters:

`exchange` - The data from the API request

Returns:

Map of singular JSON object to be used

Throws:

`IOException` - If formatting operations fail

getQueryParameters

public Map<String,Object> getQueryParameters(HttpExchange exchange)

Takes in parameters from URL (GET requests) body from request and returns a single JSON object

Parameters:

`exchange` - The data from the API request

Returns:

Map of singular JSON object to be used

readRequestBody

private String readRequestBody(HttpExchange exchange) throws IOException

Helper method used to grab the request body for POST, PATCH, DELETE calls

Parameters:

`exchange` - The data from the API request

Returns:

String formatted request body to be used in other methods

Throws:

`IOException` - If the method is unable to read the request body

formatJSON

public String formatJSON(Object data, String status) throws IOException

Returns a JSON-formatted string to frontend (GET)

Parameters:

`data` - The data to be returned to the frontend

`status` - A status used by the UI (typically "success" or "failure")

Returns:

String formatted response body

Throws:

`IOException` - If the method is unable to process and return the format

formatJSON

public String formatJSON(String status) throws IOException

Returns a successful status message to frontend (POST, PATCH, DELETE)

Parameters:

`status` - The status used by the UI

Returns:

String formatted status

Throws:

`IOException` - If the method is unable to process and return the status

formatJSON

public String formatJSON(String status, String message) throws IOException

Returns a failed status message to frontend (POST, PATCH, DELETE)

Parameters:

`status` - The status used by the UI

`message` - The message to be returned with the status

Returns:

String formatted status

Throws:

`IOException` - If the method is unable to process and return the status

AdministratorService

Administrator Service class that contains methods to be used for the admin user.

Method Details

getAllReports

public String getAllReports(HttpExchange exchange) throws IOException

Gathers the reports from the DB

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

getAllUsers

public String getAllUsers(HttpExchange exchange) throws IOException

Gathers the users from the DB

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

updateDesignation

public String updateDesignation(HttpExchange exchange) throws IOException

Updates the designation of a teacher or moderator

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

deleteUser

public String deleteUser(HttpExchange exchange) throws IOException

Deletes (marks for deletion) a user

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

updateReportStatus

public String updateReportStatus(HttpExchange exchange) throws IOException

Updates the status of a bug report

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

BugReportService

Bug Reports Service class that contains methods to be used for the bug reports functionality.

Method Details

getUserReports

public String getUserReports(HttpExchange exchange) throws IOException

Retrieves all reports associated with the currently logged-in user.

Parameters:

`exchange` - HttpExchange object containing the session information.

Returns:

A JSON-formatted string containing the user's reports or an error message.

Throws:

`IOException` - If there is an issue reading from the exchange or writing the response.

resolveReport

public String resolveReport(HttpExchange exchange) throws IOException

Resolves a report based on the parameters received from the HTTP exchange.

Parameters:

`exchange` - HttpExchange object containing the report ID to resolve.

Returns:

A JSON-formatted success message or an error message in case of failure.

Throws:

`IOException` - If there is an issue reading from the exchange or writing the response.

addReport

public String addReport(HttpExchange exchange) throws IOException

Adds a new report to the system using parameters from the HTTP exchange and session data.

Parameters:

`exchange` - HttpExchange object containing report details.

Returns:

A JSON-formatted success message or an error message in case of failure.

Throws:

`IOException` - If there is an issue reading from the exchange or writing the response.

CreatePlaylistService

Service class for taking API requests, processing, and sending queries for the creation of playlists.

Method Details

getClassOptions

public String getClassOptions(HttpExchange exchange) throws IOException

Gets class options for the create playlist page

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

createPlaylist

public String createPlaylist(HttpExchange exchange) throws IOException

Creates playlist and adds songs to it

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

LoginService

Service class for taking API requests, processing, and sending queries related to user login.

Method Details

checkExistingSession

public String checkExistingSession(HttpExchange exchange) throws IOException

Checks if a session exists and makes one if it does not.

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

authenticateLogin

public String authenticateLogin(HttpExchange exchange) throws IOException

Takes the email and password of a user and compares it to the details in the database

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

userLogout

public String userLogout(HttpExchange exchange)

Logs out the user by deleting their session from the database and expiring the session cookie.

Parameters:

`exchange` - The data from the API request

Returns:

An empty response string

MetadataService

Service class for taking API requests, processing, and sending queries related to project metadata.

Method Details

getMetadata

public String getMetadata(HttpExchange exchange) throws IOException

Returns the most recent metadata for the application

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

ModeratorService

Moderator Service class for moderator processing related to classes, students, and teachers.

Method Details

getAllClasses

public String getAllClasses(HttpExchange exchange) throws IOException

Gathers the classes from the DB

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

updateDesignation

public String updateDesignation(HttpExchange exchange) throws IOException

Updates the designation of a teacher or moderator

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

addClass

public String addClass(HttpExchange exchange) throws IOException

Adds a new class.

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

getClassStudents

public String getClassStudents(HttpExchange exchange) throws IOException

Gets the class list of students.

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

addStudent

public String addStudent(HttpExchange exchange) throws IOException

Add a student into a class.

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

getClassTeachers

public String getClassTeachers(HttpExchange exchange) throws IOException

Get teacher data from a class.

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

removeTeacher

public String removeTeacher(HttpExchange exchange) throws IOException

Remove Teacher from a class

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

addTeacher

public String addTeacher(HttpExchange exchange) throws IOException

Add a teacher to a class.

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

changeClassname

public String changeClassname(HttpExchange exchange) throws IOException

Change the name of a class.

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

removeStudent

public String removeStudent(HttpExchange exchange) throws IOException

Remove a student from the roster

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

deleteClass

public String deleteClass(HttpExchange exchange) throws IOException

Delete a class.

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of success or error message

Throws:

`IOException` - If data processing fails

QuizResultsService

Service class for taking API requests, processing, and sending queries for the quiz results page.

Method Details

submitAnswer

public String submitAnswer(HttpExchange exchange) throws IOException

Processes a submitted answer for a quiz question and updates the student performance data. If the answer is incorrect, adds the result to the quiz results database.

Parameters:

`exchange` - The HTTP exchange containing quiz answer data

Returns:

String JSON formatted response indicating success or error

Throws:

`IOException` - If there is an error processing the request data

getQuizResults

public String getQuizResults(HttpExchange exchange) throws IOException

Returns quiz results to the quiz results page for the current user

Parameters:

`exchange` - The HTTP exchange containing request data

Returns:

String JSON formatted string of quiz results data for frontend

Throws:

`IOException` - If data processing fails

getQuizSettings

public String getQuizSettings(HttpExchange exchange) throws IOException

Retrieves the quiz settings for the current user

Parameters:

`exchange` - The HTTP exchange containing request data

Returns:

String JSON formatted string of quiz settings data for frontend

Throws:

`IOException` - If data processing fails

getCorrectAnswers

public String getCorrectAnswers(HttpExchange exchange) throws IOException

Returns the correct answers for songs from a taken quiz and marks the quiz results and settings as deleted in the database

Parameters:

`exchange` - The HTTP exchange containing list of song IDs

Returns:

String JSON formatted string of correct song data for frontend

Throws:

`IOException` - If data processing fails

RegistrationService

Service class for taking API requests, processing, and sending queries related to user registration.

Method Details

registerUser

public String registerUser(HttpExchange exchange) throws IOException

Sends the data for a user to be registered to the database

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

SetQuizService

Service class for taking API requests, processing, and sending queries for the quiz settings page.

Method Details

getPlaylists

public String getPlaylists(HttpExchange exchange) throws IOException

Gathers playlist data based on classID

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

setQuizParameters

public String setQuizParameters(HttpExchange exchange) throws IOException

Sends current quiz parameters to the database for later retrieval

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

SnippetService

Service class for taking API requests, processing, and sending queries for the student song snippets.

getTimeStamp

public String getTimeStamp(HttpExchange exchange) throws IOException

Gets the timestamp for the song

Parameters:

`exchange` - The data from the API request

Returns:

String timestamp for frontend

Throws:

`IOException` - If data processing fails

checkTimeBoundary

private boolean checkTimeBoundary(int songDuration, int playbackDuration, int timestamp)

Checks the boundary of the song snippet and playback duration

Parameters:

`songDuration` - length of the song

`playbackDuration` - lenght of the song playback

`timestamp` - timestamp the playback starts at

Returns:

result of if the song plays outside of the boundary

StudentPerformanceService

Service class for taking API requests, processing, and sending queries for the student performance screen.

Method Details

getSongPerformances

public String getSongPerformances(HttpExchange exchange) throws IOException

Gets the performance of songs for a student

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

TakeQuizService

Service class for taking API requests, processing, and sending queries for the take quiz screen.

Method Details

getQuizSettings

public String getQuizSettings(HttpExchange exchange) throws IOException

Gets the quiz settings for the given playlistID

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

nextQuestionSong

public String nextQuestionSong(HttpExchange exchange) throws IOException

Gets the songs from the given playlistID

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:
`IOException` - If data processing fails

TeacherClasslistService

Teacher Classlist service for the viewing classlist.

Method Details

getClasslist

public String getClasslist(HttpExchange exchange) throws IOException

Gathers the classes from the DB

Parameters:
`exchange` - The data from the API request
Returns:
String JSON formatted string of data for frontend
Throws:
`IOException` - If data processing fails

TeacherLibraryService

Teacher library service for the viewing a teachers playlist library.

Method Details

getLibrary

public String getLibrary(HttpExchange exchange) throws IOException

Gathers the playlists and their associated class from the DB

Parameters:
`exchange` - The data from the API request
Returns:
String JSON formatted string of data for frontend
Throws:
`IOException` - If data processing fails

renamePlaylist

public String renamePlaylist(HttpExchange exchange) throws IOException

Renames a selected playlist.

Parameters:
`exchange` - The data from the API request
Returns:
String JSON formatted string of data for frontend
Throws:
`IOException` - If data processing fails

TeacherRosterService

Teacher roster service for the students in a class.

Method Details

getClassRoster

public String getClassRoster(HttpExchange exchange) throws IOException

Gathers the students in a class from the DB

Parameters:
`exchange` - The data from the API request
Returns:
String JSON formatted string of data for frontend
Throws:
`IOException` - If data processing fails

TeacherSongService

Service class for taking API requests, processing, and sending queries for the teacher songs.

Method Details

addSong

public String addSong(HttpExchange exchange) throws IOException

Adds a song to a playlist by sending its data

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

getPlaylistSongs

public String getPlaylistSongs(HttpExchange exchange) throws IOException

Gets the playlist songs to view a playlist

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend with songs

Throws:

`IOException` - If data processing fails

editSong

public String editSong(HttpExchange exchange) throws IOException

Edits data in a song (name, composer, or year)

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

deleteSong

public String deleteSong(HttpExchange exchange) throws IOException

Deletes a song from a playlist

Parameters:

`exchange` - The data from the API request

Returns:

String JSON formatted string of data for frontend

Throws:

`IOException` - If data processing fails

Implementations

The next file type for the Listening Study Application is the implementation file. Implementation files are extensions of some service files for computation. The intent of these files is to help with abstraction, readability, and maintainability of the service files.

QuizImplementation

Implementation class for extra processing of quizzes.

Method Details

checkAnswers

public boolean checkAnswers(Map<String,Object> quizData, Map<String,Object> songData)

Checks the answers to see if the student got the question correct

Parameters:

`quizData` - Map student inputted data for question

`songData` - map is actually answer of song question

Returns:

true or false if the answer matches the question

SnippetImplementation

Implementation class for extra processing of snippets.

Method Details

generateRandomTimestamp

public int generateRandomTimestamp(int songDuration)

Generates a random timestamp for the song and teturns

Parameters:

`songDuration` - duration length of song

Returns:

timestamp randomely generated

refactorTimeStamp

public int refactorTimeStamp(int songDuration, int playbackduration)

Generates a random timestamp for the song and returns

Parameters:

`songDuration` - duration length of song

`playbackduration` - duration length song playback

Returns:

refractored timestamp

SongImplementation

Implementation class for extra processing of songs.

Method Details

getMostReplayedTimestamp

public int getMostReplayedTimestamp(String url)

Generates a timestamp based on the most viewed part of the song

Parameters:

`url` - link to the youtube song

Returns:

refractored timestamp

getVideoDuration

private int getVideoDuration(org.openqa.selenium.WebDriver driver)

Retrieves the duration of the video

Parameters:

`driver` - webdriver controls browser for video information retrieval

Returns:

video duration as int

convertTimeToSeconds

public int convertTimeToSeconds(String time)

Converts timestamp string into seconds

Parameters:

`time` - timestamp string in HH:MM:SS or MM:SS format

Returns:

timestamp measured in seconds

extractMostReplayedTimestamp

```
private int extractMostReplayedTimestamp(String heatmap, int videoDuration)
```

Retrieves the most replayed timestamp of the video

Parameters:

`heatmap` - is where viewership of the video is

`videoDuration` - is the length of the video

Returns:

most replayed timestamp

extractVideoId

```
public String extractVideoId(String url)
```

Extracts the video ID if valid

Parameters:

`url` - is the link to the youtube video

Returns:

the ID returned or invalid if not a valid link

TakeQuizImplementation

Implementation class for taking processing on the take quiz screen.

Method Details

getThompsonSelection

```
public Map<String,Object> getThompsonSelection(ArrayList<Map<String,Object>>
playlistSongList, ArrayList<Integer> alreadySelectedIDs)
```

Gets the belief of success for songs using Thompson Sampling.

Parameters:

`playlistSongList` - List of songs from the playlist

`alreadySelectedIDs` - List of song IDs that have already been chosen for the quiz

Returns:

Map of song details with belief included for the frontend and the playback method to be used based on belief

Repositories

The next file type for the Listening Study Application is the repository. The repository is called from the service file and is strictly responsible for executing queries on the database. Repository files are table specific, and do not often correlate with handler/service files.

ClassRepository

Repository class to execute queries on the class table.

Method Details

getClasslist

public ResultSet getClasslist(int teacherID) throws SQLException

Retrieves classes for a specific teacher.

Parameters:

`teacherID` - the ID of the teacher

Returns:

ResultSet containing the classes taught by the specified teacher

Throws:

`SQLException` - if the SQL query encounters an error during execution

getAllClasses

public ResultSet getAllClasses() throws SQLException

Retrieves all classes from the database.

Returns:

ResultSet containing all classes

Throws:

`SQLException` - if the SQL query encounters an error during execution

removeTeacherFromClasses

public void removeTeacherFromClasses(Map<String,Object> teacher) throws SQLException

Removes a teacher from all classes by setting their teacherID to NULL.

Parameters:

`teacher` - a map containing the teacher's data, including the user ID

Throws:

`SQLException` - if the SQL query encounters an error during execution

getTeachersByClassID

public ResultSet getTeachersByClassID(int classID) throws SQLException

Retrieves teachers associated with a specific class.

Parameters:

`classID` - the ID of the class

Returns:

ResultSet containing the teachers and class information

Throws:

`SQLException` - if the SQL query encounters an error during execution

addClass

public void addClass(String className, int teacherID) throws SQLException

Adds a new class with a specified name and teacher.

Parameters:

`className` - the name of the class to add

`teacherID` - the ID of the teacher to assign to the class

Throws:

`SQLException` - if the SQL query encounters an error during execution

removeTeacherFromClass

public void removeTeacherFromClass(int classID) throws SQLException

Removes a teacher from a specific class by setting teacherID to NULL.

Parameters:

`classID` - the ID of the class to remove the teacher from

Throws:

`SQLException` - if the SQL query encounters an error during execution

renameClass

public void renameClass(Object classID, Object newClassname) throws SQLException

Updates the name of a class.

Parameters:

`classID` - the ID of the class to rename

`newClassname` - the new name for the class

Throws:

`SQLException` - if the SQL query encounters an error during execution

addTeacherToClass

public void addTeacherToClass(int classID, String teacherEmail) throws SQLException

Assigns a teacher to a class using the teacher's email.

Parameters:

`classID` - the ID of the class to assign the teacher to

`teacherEmail` - the email of the teacher to assign

Throws:

`SQLException` - if the SQL query encounters an error during execution

getStudentsByClassID

public ResultSet getStudentsByClassID(int classID) throws SQLException

Retrieves students enrolled in a specific class.

Parameters:

`classID` - the ID of the class

Returns:

ResultSet containing the students in the specified class

Throws:

`SQLException` - if the SQL query encounters an error during execution

deleteClass

public void deleteClass(int classID) throws SQLException

Deletes a class from the database.

Parameters:

`classID` - the ID of the class to delete

Throws:

`SQLException` - if the SQL query encounters an error during execution

MetadataRepository

Repository class to execute queries on the meta data view.

Method Details

getApplicationMetadata

public ResultSet getApplicationMetadata() throws SQLException

Returns the most recent meta data added

Returns:

ResultSet containing metadata information

Throws:

`SQLException` - When the query does not run properly

PlaylistRepository

Repository class to execute queries on the playlist table.

Method Details

getPlaylistByClassID

public ResultSet getPlaylistByClassID(ArrayList<Integer> classIDs) throws SQLException

Returns the playlist by Class ID.

Parameters:

`classIDs` - The list of active class IDs to search for

Returns:

ResultSet containing query results of playlists associated with the provided class IDs

Throws:

`SQLException` - When the database query does not execute properly

getPlaylistByTeacherID

public ResultSet getPlaylistByTeacherID(Object teacherID) throws SQLException

Retrieves playlists associated with a specific teacher.

Parameters:

`teacherID` - The ID of the teacher

Returns:

ResultSet containing playlists created by the specified teacher

Throws:

`SQLException` - When the database query does not execute properly

getPlaylistByStudentID

public ResultSet getPlaylistByStudentID(Object studentID) throws SQLException

Retrieves playlists accessible to a specific student based on their classes.

Parameters:

`studentID` - The ID of the student

Returns:

ResultSet containing playlists available to the specified student

Throws:

`SQLException` - When the database query does not execute properly

createPlaylist

public void createPlaylist(String playlistName, int teacherID, int classID) throws SQLException

Creates a new playlist in the database.

Parameters:

`playlistName` - The name of the playlist to create

`teacherID` - The ID of the teacher creating the playlist

`classID` - The ID of the class this playlist belongs to

Throws:

`SQLException` - When the database insertion does not execute properly

renamePlaylist

public void renamePlaylist(Object playlistID, Object newPlaylistName) throws SQLException

Updates the name of an existing playlist.

Parameters:

`playlistID` - The ID of the playlist to rename

`newPlaylistName` - The new name for the playlist

Throws:

`SQLException` - When the database update does not execute properly

getPlaylistID

public ResultSet getPlaylistID(int teacherID, int classID) throws SQLException

Retrieves the playlist ID for a specific teacher and class combination.

Parameters:

`teacherID` - The ID of the teacher

`classID` - The ID of the class

Returns:

ResultSet containing the ID of matching playlist(s)

Throws:

`SQLException` - When the database query does not execute properly

getPlaylistIDsForUser

public ResultSet getPlaylistIDsForUser(int userID) throws SQLException

Retrieves all playlist IDs accessible to a specific user, whether as a student or teacher. Uses a UNION query to combine results from both roles.

Parameters:

`userID` - The ID of the user

Returns:

ResultSet containing all playlist IDs the user has access to

Throws:

`SQLException` - When the database query does not execute properly

getPlaylistNameByID

public ResultSet getPlaylistNameByID(int playlistID) throws SQLException

Retrieves the name of a playlist by its ID.

Parameters:

`playlistID` - The ID of the playlist

Returns:

ResultSet containing the name of the specified playlist

Throws:

`SQLException` - When the database query does not execute properly

deletePlaylistByClassID

public void deletePlaylistByClassID(int classID) throws SQLException

Deletes all playlists associated with a specific class.

Parameters:

`classID` - The ID of the class whose playlists should be deleted

Throws:

`SQLException` - When the database deletion does not execute properly

PlaylistSongRepository

Repository class to execute queries on the playlistsongrepository table.

Method Details

getSongIDs

public List<Integer> getSongIDs(int playListID) throws SQLException

Returns the songIDs from a specific playlist.

Parameters:

`playListID` - The ID of the active playlist

Returns:

List of all songIDs in the tabls

Throws:

`SQLException` - When the query does not run properly

getSongTimestamps

public List<Integer> getSongTimestamps(int playListID, int playbackMethod) throws SQLException

Returns the timeStamps to be used by the quiz

Parameters:

`playListID` - The ID of the active playlist

`playbackMethod` - The playbackmethod type random, teacherdefined, or most viewed

Returns:

List of timestamps based on playback method: empty for random (1), timestamp for most viewed (2), and userDefinedtimestamp for user defined (3)

Throws:

`SQLException` - When the query does not run properly

addToPlaylist

public void addToPlaylist(int playListID, int songID, int udTimeStamp) throws SQLException

Adds a SongID and its user-defined timestamp to a playlist

Parameters:

`playListID` - The ID of the playlist being inserted into

`songID` - The ID of the song getting added to the playlist

`udTimeStamp` - The timestamp defined by the teacher added to the table

Throws:

`SQLException` - When the query does not run properly

getSongs

public ResultSet getSongs(int playlistID) throws SQLException

Returns the resultset containing the song data.

Parameters:

`playlistID` - The ID of the active playlist

Returns:

ResultSet containing query results

Throws:

`SQLException` - When the query does not run properly

getPlaylistSongs

public ResultSet getPlaylistSongs(int playlistID, int songID) throws SQLException

Returns the resultset containing the playlist song data.

Parameters:

`playlistID` - The ID of the active playlist

`songID` - The ID of the active song

Returns:

ResultSet containing query results

Throws:

`SQLException` - When the query does not run properly

updatePlaylistSong

public void updatePlaylistSong(int playlistID, int songID, int udTimestamp) throws SQLException

Updates the user-defined timestamp for a song in a playlist

Parameters:

`playlistID` - The ID of the playlist containing the song

`songID` - The ID of the song to update

`udTimestamp` - The new user-defined timestamp value

Throws:

`SQLException` - When the query does not run properly

deletePlaylistSong

public void deletePlaylistSong(int playlistID, int songID) throws SQLException

Removes a song from a playlist

Parameters:

`playlistID` - The ID of the playlist containing the song

`songID` - The ID of the song to remove

Throws:

`SQLException` - When the query does not run properly

deleteSongsByPlaylistID

public void deleteSongsByPlaylistID(int classID) throws SQLException

Deletes all songs from playlists associated with a specific class

Parameters:

`classID` - The ID of the class whose playlists' songs should be deleted

Throws:

`SQLException` - When the query does not run properly

QuizResultsRepository

Repository class to execute queries on the quiz results table.

Method Details

addQuizResults

public void addQuizResults(int quizSettingsID, String songName, String songComposer, String songYear, int songID, int userID, int numQuestions) throws SQLException

Adds active quiz's results to to the quiz results table

Parameters:

`quizSettingsID` - The ID of the active quiz

`songName` - The name of the song

`songComposer` - The composer of the song

`songYear` - The year of the song

`songID` - The ID of the song

`userID` - The year of the song

`numQuestions` - The ID of the song

Throws:

`SQLException` - When the query does not run properly

getQuizResults

public ResultSet getQuizResults(int userID) throws SQLException

Gets Quiz settings by the userID

Parameters:

`userID` - The ID of the active quiz

Returns:

ResultSet containing quiz results

Throws:

`SQLException` - When the query does not run properly

setDeletedByID

public void setDeletedByID(int userID) throws SQLException

Sets deleted field in quiz results table to 1 by userID

Parameters:

`userID` - The ID of the active quiz

Throws:

`SQLException` - When the query does not run properly

QuizSettingsRepository

Repository class to execute queries on the quiz settings table.

Method Details

addQuizSettings

public void addQuizSettings(int user_ID, String playbackMethod, int playbackDuration, int numQuestions, int playlistID) throws SQLException

Adds the most recent quiz settings to the table

Parameters:

`playlistID` - The ID of the active playlist

`user_ID` - The ID of the active user

`playbackMethod` - The current selected playback method for the quiz

`playbackDuration` - The chosen duration of playback (as a number)

`numQuestions` - The chosen number of questions for the current quiz settings

Throws:

`SQLException` - When the query does not run properly

getQuizSettingsByID

public ResultSet getQuizSettingsByID(int userID) throws SQLException

Returns the most recent quiz settings from the chosen playlistID

Parameters:

`userID` - The ID of the current user.

Returns:

ResultSet containing query results

Throws:

`SQLException` - When the query does not run properly

getQuizSettings

public ResultSet getQuizSettings(int userID) throws SQLException

gets quiz settings by the user Id.

Parameters:

`userID` - The ID of the active user

Returns:

ResultSet containing quiz settings

Throws:

`SQLException` - When the query does not run properly

setDeletedByID

public void setDeletedByID(int userID) throws SQLException

Sets deleted field in quiz settings table to 1 by quizSettingsID

Parameters:

`userID` - The ID of the active user

Throws:

`SQLException` - When the query does not run properly

ReportRepository

Repository class to execute queries on the reports table.

Method Details

getAllReports

public ResultSet getAllReports() throws SQLException

Returns all reports

Returns:

ResultSet containing query results

Throws:

`SQLException` - When the query does not run properly

getReportsByUserID

public ResultSet getReportsByUserID(Integer userID) throws SQLException

Retrieves all reports associated with a specific user based on their user ID.

Parameters:

`userID` - The unique ID of the user whose reports are to be fetched.

Returns:

ResultSet containing all matching reports from the reportTime table.

Throws:

`SQLException` - When the query does not run properly

updateReportStatus

public void updateReportStatus(Map<String,Object> report) throws SQLException

Modifies the report table by changing the status of a report

Parameters:

`report` - Map that that contains the status if report and id.

Throws:

`SQLException` - When the query does not run properly

resolveReport

public void resolveReport(Map<String,Object> report) throws SQLException

Marks a report as resolved by updating its status and resolution fields.

Parameters:

`report` - Map that contains the ID of the report to be resolved.

Throws:

`SQLException` - When the query does not run properly

addReport

public void addReport(Map<String,Object> report) throws SQLException

Adds a new report to the reportTime table with default status as "Open" and current time as time of report.

Parameters:

`report` - Map that contains email, description, and lastUpdatedBy fields for the report.

Throws:

`SQLException` - When the query does not run properly

SessionRepository

Repository class to execute queries on the sessions table.

Method Details

createSession

public void createSession(String sessionID, int userID, String userRole) throws SQLException

Creates a new session in the database for the specified user.

Parameters:

`sessionID` - Unique identifier for the session

`userID` - ID of the user associated with the session

`userRole` - Role of the user (e.g., admin, teacher, student)

Throws:

`SQLException` - If the insert query fails

deleteSession

public void deleteSession(String sessionID) throws SQLException

Deletes a session from the database based on the session ID.

Parameters:

`sessionID` - The session ID to be deleted

Throws:

`SQLException` - If the delete query fails

getUserIDBySessionID

public Integer getUserIDBySessionID(String sessionID) throws SQLException

Retrieves the user ID associated with a given session ID.

Parameters:

`sessionID` - The session ID used to look up the user

Returns:

The user ID if found; otherwise, null

Throws:

`SQLException` - If the select query fails

getUserRoleBySessionID

public ResultSet getUserRoleBySessionID(String sessionID) throws SQLException

Returns the session details including the user's role based on a valid session ID.

Parameters:

`sessionID` - The session ID of the user

Returns:

ResultSet containing session information if the session is valid and not expired

Throws:

`SQLException` - When the query fails to execute properly

SongRepository

Repository class to execute queries on the song table.

Method Details

commitSongData

public void commitSongData(String songName, String songComposer, String songYear, String youtubeLink, int mrTimestamp) throws SQLException

Adds a new song and its data to the song table.

Parameters:

`songName` - The name of the song

`songComposer` - The composer of the song

`songYear` - The year the song was released

`youtubeLink` - The YouTube link of the song

`mrTimestamp` - The most replayed timestamp of the song

Throws:

`SQLException` - When the query does not run properly

getSongID

public ResultSet getSongID(String youtubeLink) throws SQLException

Retrieves the song ID based on the YouTube link.

Parameters:

`youtubeLink` - The YouTube link of the song

Returns:

ResultSet containing the song ID if found

Throws:

`SQLException` - When the query does not run properly

getSongData

public ResultSet getSongData(int songID) throws SQLException

Retrieves all song data based on the song ID.

Parameters:

`songID` - The ID of the song

Returns:

ResultSet containing the song's data

Throws:

`SQLException` - When the query does not run properly

updateSongData

public void updateSongData(String songName, String songComposer, String songYear, String youtubeLink, int mrTimestamp, int songID) throws SQLException

Updates an existing song entry in the song table.

Parameters:

`songName` - The new name of the song

`songComposer` - The new composer of the song

`songYear` - The new year of the song

`youtubeLink` - The updated YouTube link

`mrTimestamp` - The updated most replayed timestamp

`songID` - The ID of the song to be updated

Throws:

`SQLException` - When the update query fails

StudentClassRepository

Repository class to execute queries on the studentClass table.

Method Details

addStudentToClass

public void addStudentToClass(int studentID, int classID) throws SQLException

Adds a student to a specific class.

Parameters:

`studentID` - The ID of the student to be added

`classID` - The ID of the class to which the student is added

Throws:

`SQLException` - When the insert query fails

removeStudentFromClass

public void removeStudentFromClass(int studentID, int classID) throws SQLException

Removes a student from a specific class.

Parameters:

`studentID` - The ID of the student to be removed

`classID` - The ID of the class from which the student is removed

Throws:

`SQLException` - When the delete query fails

removeAllStudentsFromClass

public void removeAllStudentsFromClass(int classID) throws SQLException

Removes all students from a specific class.

Parameters:

`classID` - The ID of the class whose student associations are to be removed

Throws:

`SQLException` - When the delete query fails

getClassIDByStudentID

public ResultSet getClassIDByStudentID(int studentID) throws SQLException

Retrieves all class IDs associated with a given student.

Parameters:

`studentID` - The ID of the student

Returns:

ResultSet containing all class associations for the student

Throws:

`SQLException` - When the select query fails

StudentPerformanceRepository

Repository class to execute queries on the student performance table.

Method Details

getPerformanceByID

public ResultSet getPerformanceByID(Object studentID) throws SQLException

gets the student performance by ID

Parameters:

`studentID` - The ID of the student

Returns:

ResultSet containing query results

Throws:

`SQLException` - When the query does not run properly

getSongWeight

public ResultSet getSongWeight(int songID, int studentID) throws SQLException

returns the weight of the question by song and studentID

Parameters:

`studentID` - The ID of the student

`songID` - The ID of the song

Returns:

ResultSet containing query results

Throws:

`SQLException` - When the query does not run properly

getTimesQuizzedAndCorrect

public ResultSet getTimesQuizzedAndCorrect(int songID, int studentID, int playlistID) throws SQLException

Gets the times quizzied and correct

Parameters:

`songID` - The ID of the song

`playlistID` - ID of the playlist

`studentID` - ID of the student

Returns:

ResultSet containing query results

Throws:

`SQLException` - When the query does not run properly

getPerformanceData

public ResultSet getPerformanceData(int studentID, int playlistID, int songID) throws SQLException

returns the performance data if a song on a playlist

Parameters:

`studentID` - The ID of the student

`playlistID` - The ID of the student

`songID` - The ID of the song

Returns:

ResultSet containing query results

Throws:

`SQLException` - When the query does not run properly

updatePerformanceData

public void updatePerformanceData(int timesQuizzed, int timesCorrect, double score, int studentPerformanceID) throws SQLException

updates the performance data of a song

Parameters:

`timesQuizzed` - number of times a question was quizzed

`timesCorrect` - number of times a student got a question correct

`score` - the score of a song

`studentPerformanceID` - The ID of performance

Throws:

`SQLException` - When the query does not run properly

addPerformanceData

public void addPerformanceData(int studentID, int songID, int playlistID, int timesQuizzed, int timesCorrect, double score) throws SQLException

Adds Performance data to the performance table

Parameters:

`studentID` - The ID of the student

`songID` - The ID of the song

`playlistID` - ID of the playlist

`timesCorrect` - number of times a student got a question correct

`timesQuizzed` - number of times a student was quizzed on a question

`score` - the score of a song

Throws:

`SQLException` - When the query does not run properly

deletePerformanceData

public void deletePerformanceData(int playlistID, int songID) throws SQLException

Deletes performance data

Parameters:

`songID` - The ID of the song

`playlistID` - ID of the playlist

Throws:

`SQLException` - When the query does not run properly

deletePerformanceByPlaylistID

public void deletePerformanceByPlaylistID(int classID) throws SQLException

Deletes performance data by the classID

Parameters:

`classID` - The ID of the class

Throws:

`SQLException` - When the query does not run properly

StudentRepository

Repository class for retrieving student-related data from the database.

Method Details

getStudentRoster

public ResultSet getStudentRoster(int classID) throws SQLException

Retrieves the roster of students in a specific class.

Parameters:

`classID` - The ID of the class

Returns:

ResultSet containing student ID, email, first name, and last name

Throws:

`SQLException` - When the query fails to execute

getStudentByEmail

public ResultSet getStudentByEmail(String studentEmail) throws SQLException

Retrieves student details by their email address.

Parameters:

`studentEmail` - The email of the student

Returns:

ResultSet containing student record details

Throws:

`SQLException` - When the query fails to execute

getStudentByUserID

public ResultSet getStudentByUserID(int userID) throws SQLException

Retrieves student details using the associated user ID.

Parameters:

`userID` - The user ID linked to the student

Returns:

ResultSet containing student record details

Throws:

`SQLException` - When the query fails to execute

TeacherRepository

Repository class to execute queries on the teacherMaster table.

Method Details

getTeacherID

public ResultSet getTeacherID(int userID) throws SQLException

Retrieves the teacher ID associated with a specific user ID.

Parameters:

`userID` - The user ID linked to the teacher

Returns:

ResultSet containing the teacher ID

Throws:

`SQLException` - When the query fails to execute

removeTeacherFromClasses

public void removeTeacherFromClasses(Map<String,Object> teacher) throws SQLException

Removes a teacher from all classes by marking them as inactive in the teacherMaster table.

Parameters:

`teacher` - A map containing the teacher's data, including the user ID (key: "id")

Throws:

`SQLException` - When the query fails to execute

getTeacherByEmail

public ResultSet getTeacherByEmail(String teacherEmail) throws SQLException

Retrieves teacher details using their email address.

Parameters:

`teacherEmail` - The email of the teacher

Returns:

ResultSet containing the teacher's record

Throws:

`SQLException` - When the query fails to execute

UserRepository

Repository class to execute queries on the user table.

Method Details

addUser

public void addUser(Map<String,Object> user) throws SQLException

Adds the user to the user table

Parameters:

`user` - map that contains the user and its data

Throws:

`SQLException` - When the query does not run properly

deleteUser

public void deleteUser(Map<String,Object> user) throws SQLException

deletes user from the user table

Parameters:

`user` - map that contains the user and its data

Throws:

`SQLException` - When the query does not run properly

getAllUsers

public ResultSet getAllUsers() throws SQLException

returns all users

Returns:

result set of the query

Throws:

`SQLException` - When the query does not run properly

getUserByEmail

public ResultSet getUserByEmail(String email) throws SQLException

returns user by email

Parameters:
`email` - of the user
Returns:
result set of the query
Throws:
`SQLException` - When the query does not run properly

getUserById

public ResultSet getUserById(Integer id) throws SQLException

returns user by id

Parameters:
`id` - of the user
Returns:
result set of the query
Throws:
`SQLException` - When the query does not run properly

getUserCountByEmail

public ResultSet getUserCountByEmail(String email) throws SQLException

returns count of a specific email

Parameters:
`email` - of the user
Returns:
result set of the query
Throws:
`SQLException` - When the query does not run properly

updateModeratorOrTeacherDesignation

public void updateModeratorOrTeacherDesignation(Map<String,Object> user) throws SQLException

updates a moderator or teacher role

Parameters:

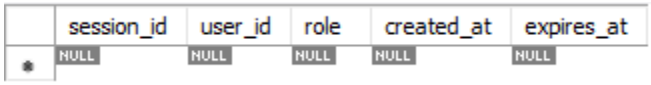
`user` - map of the user data

Throws:

`SQLException` - When the query does not run properly

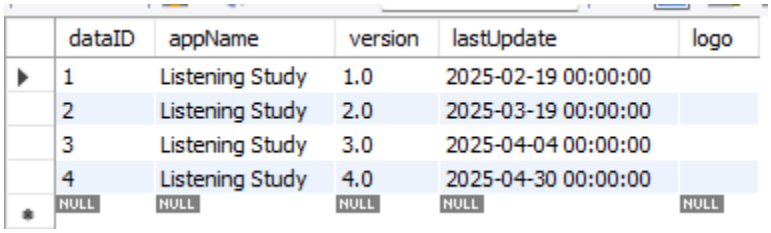
Database Explanation

This section describes each table utilized in the database created for the Listening Study application.



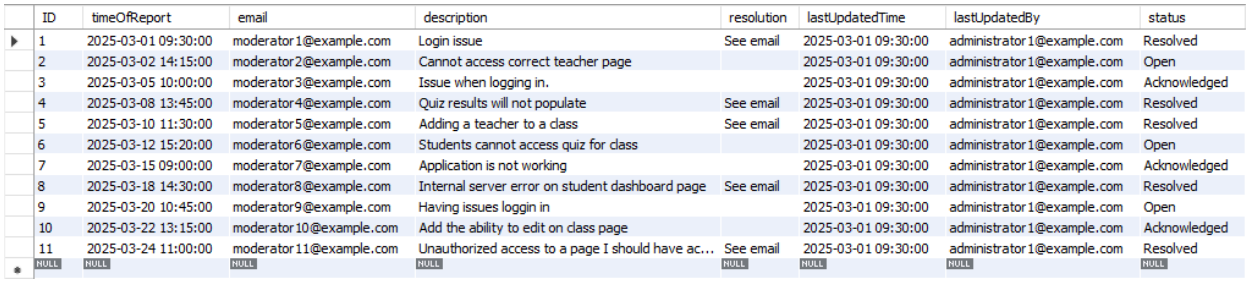
session_id	user_id	role	created_at	expires_at
NULL	NULL	NULL	NULL	NULL

This is the session table in the database. This table is used to track sessions once logged in. A session is run throughout the application, using the role field to verify that users can access what they need, and cannot access what they shouldn't. The user_id corresponds to the user who is logged in, matching the users table.



dataID	appName	version	lastUpdate	logo
1	Listening Study	1.0	2025-02-19 00:00:00	
2	Listening Study	2.0	2025-03-19 00:00:00	
3	Listening Study	3.0	2025-04-04 00:00:00	
4	Listening Study	4.0	2025-04-30 00:00:00	
NULL	NULL	NULL	NULL	NULL

This is the metadata table in the database. This serves as a repository for data used throughout footers in the application.



ID	timeOfReport	email	description	resolution	lastUpdatedTime	lastUpdatedBy	status
1	2025-03-01 09:30:00	moderator1@example.com	Login issue	See email	2025-03-01 09:30:00	administrator1@example.com	Resolved
2	2025-03-02 14:15:00	moderator2@example.com	Cannot access correct teacher page		2025-03-01 09:30:00	administrator1@example.com	Open
3	2025-03-05 10:00:00	moderator3@example.com	Issue when logging in.		2025-03-01 09:30:00	administrator1@example.com	Acknowledged
4	2025-03-08 13:45:00	moderator4@example.com	Quiz results will not populate	See email	2025-03-01 09:30:00	administrator1@example.com	Resolved
5	2025-03-10 11:30:00	moderator5@example.com	Adding a teacher to a class	See email	2025-03-01 09:30:00	administrator1@example.com	Resolved
6	2025-03-12 15:20:00	moderator6@example.com	Students cannot access quiz for class		2025-03-01 09:30:00	administrator1@example.com	Open
7	2025-03-15 09:00:00	moderator7@example.com	Application is not working		2025-03-01 09:30:00	administrator1@example.com	Acknowledged
8	2025-03-18 14:30:00	moderator8@example.com	Internal server error on student dashboard page	See email	2025-03-01 09:30:00	administrator1@example.com	Resolved
9	2025-03-20 10:45:00	moderator9@example.com	Having issues loggin in		2025-03-01 09:30:00	administrator1@example.com	Open
10	2025-03-22 13:15:00	moderator10@example.com	Add the ability to edit on class page		2025-03-01 09:30:00	administrator1@example.com	Acknowledged
11	2025-03-24 11:00:00	moderator11@example.com	Unauthorized access to a page I should have ac...	See email	2025-03-01 09:30:00	administrator1@example.com	Resolved
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

This is the reportTime table in the database. This serves as a repository for all bug reports in the application. This is used both by users to report bugs, and by administrators to view a list of bugs.

user_id	email	first_name	last_name	deleted	role	password
1	administrator1@example.com	John	Smith	0	administrator	
2	administrator2@example.com	Emma	Johnson	0	administrator	
3	administrator3@example.com	Michael	Williams	0	administrator	
4	administrator4@example.com	Olivia	Brown	0	administrator	
5	administrator5@example.com	William	Jones	0	administrator	
6	administrator6@example.com	Sophia	Miller	0	administrator	
7	administrator7@example.com	James	Davis	0	administrator	
8	administrator8@example.com	Charlotte	Garcia	0	administrator	
9	administrator9@example.com	Benjamin	Rodriguez	0	administrator	
10	administrator10@example.com	Amelia	Wilson	0	administrator	
11	administrator11@example.com	Lucas	Martinez	0	administrator	
12	moderator1@example.com	Harper	Anderson	0	moderator	
13	moderator2@example.com	Ethan	Thomas	0	moderator	
14	moderator3@example.com	Ava	Jackson	0	moderator	
15	moderator4@example.com	Noah	White	0	moderator	
16	moderator5@example.com	Isabella	Harris	0	moderator	
17	moderator6@example.com	Mason	Martin	0	moderator	
18	moderator7@example.com	Mia	Thompson	0	moderator	
19	moderator8@example.com	Jacob	Garcia	0	moderator	
20	moderator9@example.com	Abigail	Martinez	0	moderator	

This is the users table in the database. This serves as a repository for all user types to track common information, like name, email, and role. This is used throughout the application, including for administrators, registration, and login.

ID	Email	Firstname	LastName	isActive	user_id
1	student1@example.com	Liam	Scott	1	35
2	student2@example.com	Zoe	Green	1	36
3	student3@example.com	Jackson	Adams	1	37
4	student4@example.com	Lily	Baker	1	38
5	student5@example.com	Aiden	Gonzalez	1	39
6	student6@example.com	Madison	Nelson	1	40
7	student7@example.com	Owen	Carter	1	41
8	student8@example.com	Scarlett	Mitchell	1	42
9	student9@example.com	Gabriel	Perez	1	43
10	student10@example.com	Aubrey	Roberts	1	44
11	student11@example.com	Connor	Turner	1	45
NULL	NULL	NULL	NULL	NULL	NULL

This is the student table in the database. This is used to track specifically students on certain pages. The user_id corresponds to the ID provided in the users table.

ID	Email	Firstname	LastName	isActive	user_id
1	teacher1@example.com	Daniel	Lewis	1	23
2	teacher2@example.com	Sofia	Lee	1	24
3	teacher3@example.com	Matthew	Walker	1	25
4	teacher4@example.com	Ella	Hall	1	26
5	teacher5@example.com	David	Allen	1	27
6	teacher6@example.com	Grace	Young	1	28
7	teacher7@example.com	Joseph	Hernandez	1	29
8	teacher8@example.com	Chloe	King	1	30
9	teacher9@example.com	Samuel	Wright	1	31
10	teacher10@example.com	Victoria	Lopez	1	32
11	test.teacher@example.c...	Test	Teacher	1	33
12	teacher11@example.com	Henry	Hill	1	34

This is the teacherMaster table in the database. This is used to track teacher data on certain pages. The user_id corresponds to the ID provided in the users table.

ID	Email	Firstname	LastName	isActive	user_id
1	moderator1@example.com	Harper	Anderson	1	12
2	moderator2@example.com	Ethan	Thomas	1	13
3	moderator3@example.com	Ava	Jackson	1	14
4	moderator4@example.com	Noah	White	1	15
5	moderator5@example.com	Isabella	Harris	1	16
6	moderator6@example.com	Mason	Martin	1	17
7	moderator7@example.com	Mia	Thompson	1	18
8	moderator8@example.com	Jacob	Garcia	1	19
9	moderator9@example.com	Abigail	Martinez	1	20
10	moderator10@example.com	Alexander	Robinson	1	21
11	moderator11@example.com	Emily	Clark	1	22
NULL	NULL	NULL	NULL	NULL	NULL

This is the moderator table in the database. This is used to track specific moderators on certain pages. The user_id corresponds to the ID provided in the users table.

ID	Email	Firstname	LastName	isActive	user_id
1	administrator1@example.com	John	Smith	1	1
2	administrator2@example.com	Emma	Johnson	1	2
3	administrator3@example.com	Michael	Williams	1	3
4	administrator4@example.com	Olivia	Brown	1	4
5	administrator5@example.com	William	Jones	1	5
6	administrator6@example.com	Sophia	Miller	1	6
7	administrator7@example.com	James	Davis	1	7
8	administrator8@example.com	Charlotte	Garcia	1	8
9	administrator9@example.com	Benjamin	Rodriguez	1	9
10	administrator10@example.com	Amelia	Wilson	1	10
11	administrator11@example.com	Lucas	Martinez	1	11
NULL	NULL	NULL	NULL	NULL	NULL

This is the administrator table in the database. This is used to track administrators on certain pages. The user_id corresponds to the ID provided in the users table.

ID	className	teacherID
1	Test Music Class	11
2	Music Theory 101	1
3	Classical Piano	2
4	Jazz Ensemble	3
5	Vocal Training	4
6	Music History	5
7	Guitar Fundamentals	6
8	Electronic Music Production	7
9	Orchestral Studies	8
10	Music Composition	9
11	World Music	10
12	Music Appreciation	6
NULL	NULL	NULL

This is the class table in the database. This table serves as a repository for all classes throughout the application. The teacherID is used to link the teacherID where needed.

ID	studentID	classID
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	7	8
8	8	9
9	9	10
10	10	11
11	11	1
NULL	NULL	NULL

This is the studentClass table in the database. This table is used to keep track of which students are in which class, by studentID and classID.

ID	playlistName	teacherID	classID
1	Classical Masterpieces	1	1
2	Piano Fundamentals	2	2
3	Jazz Standards	3	3
4	Vocal Training Essentials	4	4
5	Historical Compositions	5	5
6	Guitar Classics	6	6
7	Electronic Music Pioneers	7	7
8	Orchestral Favorites	8	8
9	Composition Studies	9	9
10	World Music Exploration	10	10
11	Music Appreciation Sele...	11	11
NULL	NULL	NULL	NULL

This is the playlist table in the database. This is a table that holds all of the playlists to be used throughout the application. Each playlist has a teacherID and classID respectively.

ID	songName	songComposer	songYear	youtubeLink	mrTimestamp
1	Moonlight Sonata	Ludwig van Beethoven	1801	4Tr0otuiQuU	20
2	Fur Elise	Ludwig van Beethoven	1810	q9bU12gXUyM	82
3	Claire de Lune	Claude Debussy	1905	CvFH_6DNRCY	-1
4	Nocturne Op. 9 No. 2	Frédéric Chopin	1832	9E6b3swbnWg	44
5	The Four Seasons - Spring	Antonio Vivaldi	1723	mFWQgxXM_b8	15
6	Canon in D	Johann Pachelbel	1680	8Af372EQLck	30
7	Symphony No. 5	Ludwig van Beethoven	1808	fOk8Tm815lE	-1
8	The Blue Danube	Johann Strauss II	1866	cKkDMiGUbUw	20
9	Requiem in D Minor	Wolfgang Amadeus Mozart	1791	Zi8vJ_lMxQI	20
10	Gymnopédie No. 1	Erik Satie	1888	S-Xm7s9eGxU	15
11	Prelude in C Major	Johann Sebastian Bach	1722	frxT2qB1POQ	12
NULL	NULL	NULL	NULL	NULL	NULL

This is the song table in the database. This table contains information for each song used. The mrTimestamp is populated from the web scraper and denotes the most replayed part of the YouTube video in seconds.

Core Computation Explanation

The core computation of Listening Study features a Thompson sampling algorithm. Thompson sampling is a heuristic that addresses the exploration-exploitation issue in the multi-armed bandit problem. In the case of Listening Study, it chooses which songs should be chosen for a quiz based on the probability of student answering correctly on a quiz. This probability is influenced by the number of times that the student has been quizzed on a song, and the number of times that the student has gotten the song correct. The algorithm is stored in the TakeQuizImplementation.java file, and it is called from TakeQuizService.java via the getThompsonSelection() method. The Thompson selection method uses the parameters playlistSongList and alreadySelectedIDs to select the next song for a quiz. The parameter playlistSongList contains all the songs in the selected playlist along with the data for each song, as well as the student performance data for each song corresponding to the student taking the quiz. The parameter alreadySelectedIDs stores a list of song IDs that have already been selected for the current quiz. These parameters lay the foundation for the data that the Thompson sample can use to make its calculations.

In addition to those parameters, the Thompson algorithm also uses the variable minSample, which is equal to 1, and the map selectedSong, which is null. These are used later in the algorithm, but the algorithm begins with a for loop that loops through playlistSongList for

each song that can be found within playlistSongList. The algorithm will then extract the current song ID and store it in the variable songID, then it will begin to carry out various checks. The algorithm will first check for the last value contained in alreadySelectedIDs and compare it to the current value of songID. If the two values are a match, then the program will continue onto the next iteration of the for loop and attempt to select a different song. This prevents a student from getting the same song multiple times in a row on single quiz, but they can get the same song more than once. If they do not match, then the program will go to the next step and extract the values for timesCorrect and timesQuizzed from the current song. Next, alpha will be set to the value of 1 + timesCorrect and beta will be set to the value of 1 + (timesQuizzed – timesCorrect). In the case of timesQuizzed being equal to 0, alpha and beta will be set to 1.0. With alpha and beta assigned values, the Thompson algorithm is ready to determine the probability of success for the song.

To determine the probability of success on a song, the Thompson algorithm uses the values of alpha and beta to generate a beta distribution. Then, the algorithm selects a point under the curve of the data distribution as the probability of success. This gives the song selection an element of randomness, while also predicting the student's chances of success. If the probability value returned by the beta distribution sample is less than the value of minSample, then selectedSong is set equal to the current song, and the value of the probability is added to the data in selectedSong under the name of "belief." Then, the for loop at the beginning of the algorithm will continue for the remaining songs contained in playlistSongList. This process continues until the song for the question is selected.

Once the song is selected, the playback method for that song must be determined. If the Thompson algorithm determines that the student has a probability or "belief" of success greater than 0.9, then the playback method will always be random. However, if the probability is less than 0.5, the playback method will always use the most replayed part of the song according to YouTube's analytics. In the case that neither of these conditions are met, the playback method will always default to the preferred method that the student chose before starting the quiz. This ensures that songs with high success probabilities will be more difficult on a quiz than songs with low success probabilities, thus aiding the student in improving their quiz scores. Additionally, a song's probability is not static for each individual quiz. A song's probability will

change throughout the duration of the quiz as each quiz question is graded immediately. This will always give the Thompson algorithm new data to work with for the beta distribution and probability determination, while also not allowing for the same song to be selected consecutively.

Thompson sampling allows Listening Study to analyze the performance data of student users, and dynamically adapt quizzes based on the students' success. The probabilities calculated by the Thompson algorithm make it possible for students to get quiz questions that are adapted and intended to lead to improvement on their performance.

Future Development

Future development for Listening Study would revolve around making improvements to the Thompson algorithm. Thompson sampling is very effective for datasets with no data or small amounts of data. However, as the amount of data increases, Thompson sampling may not be the most accurate method. As the size of the dataset increases, it may be better to introduce a Generalized Linear Model (GLM) to handle the predictions of success. Thompson sampling aims to balance exploration and exploitation, which is perfect for songs that have little to no data within the dataset. However, a GLM aims to make accurate predictions based on variables. While a GLM may provide more accurate predictions, Thompson sampling is still necessary for Listening Study because of its ability to handle songs with no prior performance data.

Even though Thompson sampling would not be replaced, it may be possible to produce a hybrid of Thompson sampling and a GLM. The goal of this hybrid would be to have Thompson sampling handle songs that have little to no prior performance data and have the GLM handle the songs that have a sufficient amount of prior performance data. The GLM would be able to produce more accurate predictions than Thompson sampling, but it requires a dataset for training. This is where Thompson sampling could do the early work of building up the dataset through its exploration and exploitation approach. Due to Thompson sampling having an element of randomness, this would likely provide a sample dataset for the GLM that is reasonably balanced and allows for higher accuracy of predictions.

Thompson sampling works very well for Listening Study as it is now. This method provides relatively reliable probabilities of success while having an element of randomness. However, the reliability of the probabilities of success may be increased with future efforts and testing to combine the current Thompson sampling algorithm with a GLM. While Thompson sampling works well with Listening Study, it may be further improved by introducing a Thompson-GLM hybrid approach.

Conclusion

Listening Study intends to be a suitable application for students taking listening exams. Its design was based off Slippery Rock University's music history classes from the 2023-2024 school year. The main aim of the software is to replicate the testing environment, offering an effective study framework. The administrative structure involves administrators managing bug reports and designating moderators, who in turn will oversee classes in the system. Teachers can add songs to playlists for student access. Students can then generate quizzes from these playlists and view their performance.

A central feature of this software lies in its dynamic quiz generation. Students can customize quiz length, playback duration, and playback method. Playback can start from a teacher-defined timestamp, a most-replayed timestamp sourced from YouTube, or a random timestamp. The order of questions and playback method are dictated by the results of Thompson sampling. This method personalizes each quiz to the student and facilitates student progress.

In the future, the intention is to deploy this application for wide-scale use, as well as improve the prediction model. Deploying the system would require more testing, and improving the model would require more research and testing to ensure it behaves as expected. As it stands, Listening Study effectively emulates the environment for a listening exam.