**CS673 Software Engineering** 

**Team 1 - KnowItAll**

**Software Design Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Daniel Makover | Manager | *Daniel Makover* | 2021-12-08 |
| Yuwei Wu | Security Leader | *Yuwei Wu* | 2021-12-09 |
| Heli Kolambekar | Requirements Leader | *Heli Kolambekar* | 2021-12-09 |
| Gunnar Nichols | QA Leader | *Gunnar Nichols* | 2021-12-08 |
| Haoyi Zhu | Design and Implementation Leader | *Haoyi Zhu* | 2021-12-09 |
| Kun Mo | Key Algorithm | *Kun Mo* | 2021-12-08 |
| Weiye Xu | Configuration Leader | *Weiye Xu* | 2021-12-08 |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| **iteration1** |  | **2021-10-21** | **Updated to Reflact iteration 1** |
| **iteration2** |  | **2021-11-11** | **Updated to Reflact iteration 2** |
| **iteration3** |  | **2021-12-09** | **Updated to Reflact iteration 3** |

[Introduction](#_heading=h.gjdgxs)

[Software Architecture](#_heading=h.17dp8vu)

[Design Patterns](#_heading=h.3znysh7)

[Key Algorithms](#_heading=h.tyjcwt)

[Classes and Methods](#_heading=h.3dy6vkm)

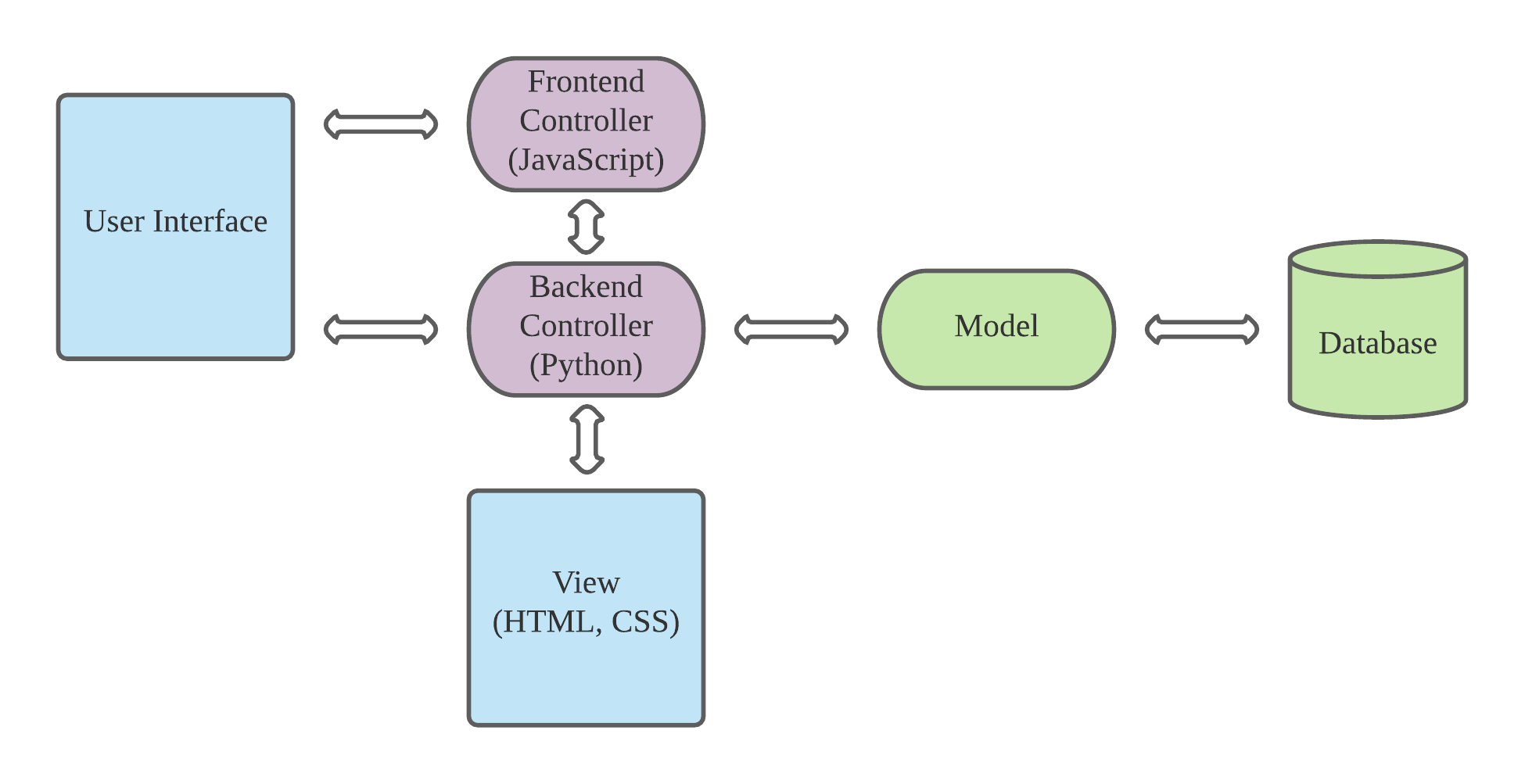
[References](#_heading=h.4d34og8)

[Glossary](#_heading=h.2s8eyo1)

# Introduction

KnowItAll is an interactive trivia web application built using Python and Javascript. It uses SQLite as a database. The following document covers the software architecture, database design, security/design plan, design patterns, key algorithms and UI Designs.

# Software Architecture



[MVC Architecture](https://lucid.app/lucidchart/c31356c7-8d51-449f-8822-82c6545c34ac/edit?invitationId=inv_c2d79705-7b1f-4310-b73d-46621a6ae158)

In our KnowItAll Web App, our design uses Flask as a framework and SQLite as a database in development environment, following the model-view-controller architecture.

For our production environment, we use PostgreSQL.

Our controller is the bridge between the user interface and the Database. Through the player's operation on the user interface, the player's input is converted into game actions, and these actions will call the data in the Database.

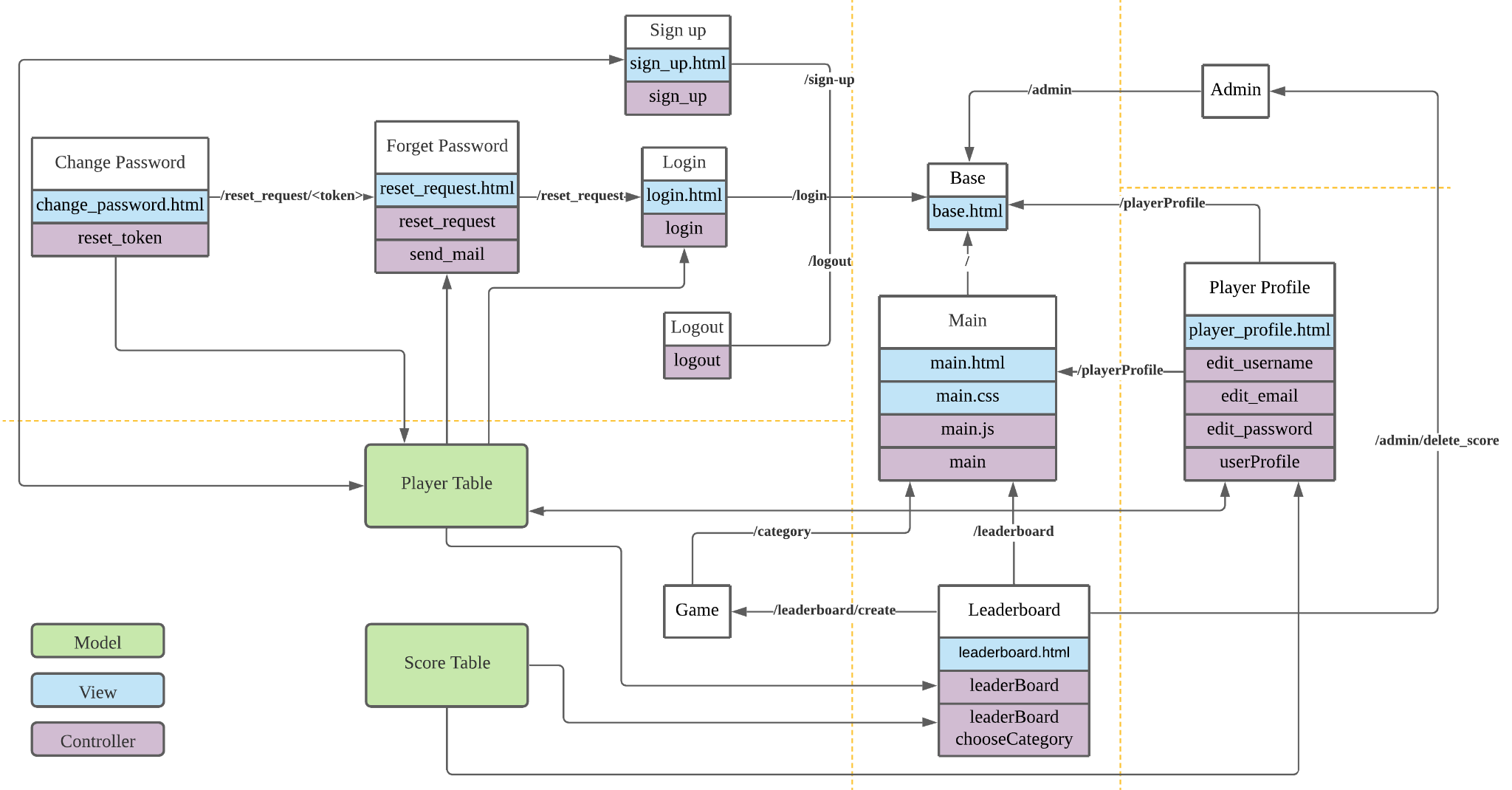
We are using the Flask BluePrint with Flask decorator to bind the backend function with our frontend function. The route is the trigger, that frontend function can use the route with ajax function to trigger backend logic.

In the controller, we have the gameSettings route to get questions from DB, then the logic inside of gameSettings would randomize the question options returned from the database and identify the correct answer after randomizing. Once the game is over we trigger logic to add the players game score to the database. The player also has the option to not add their score.

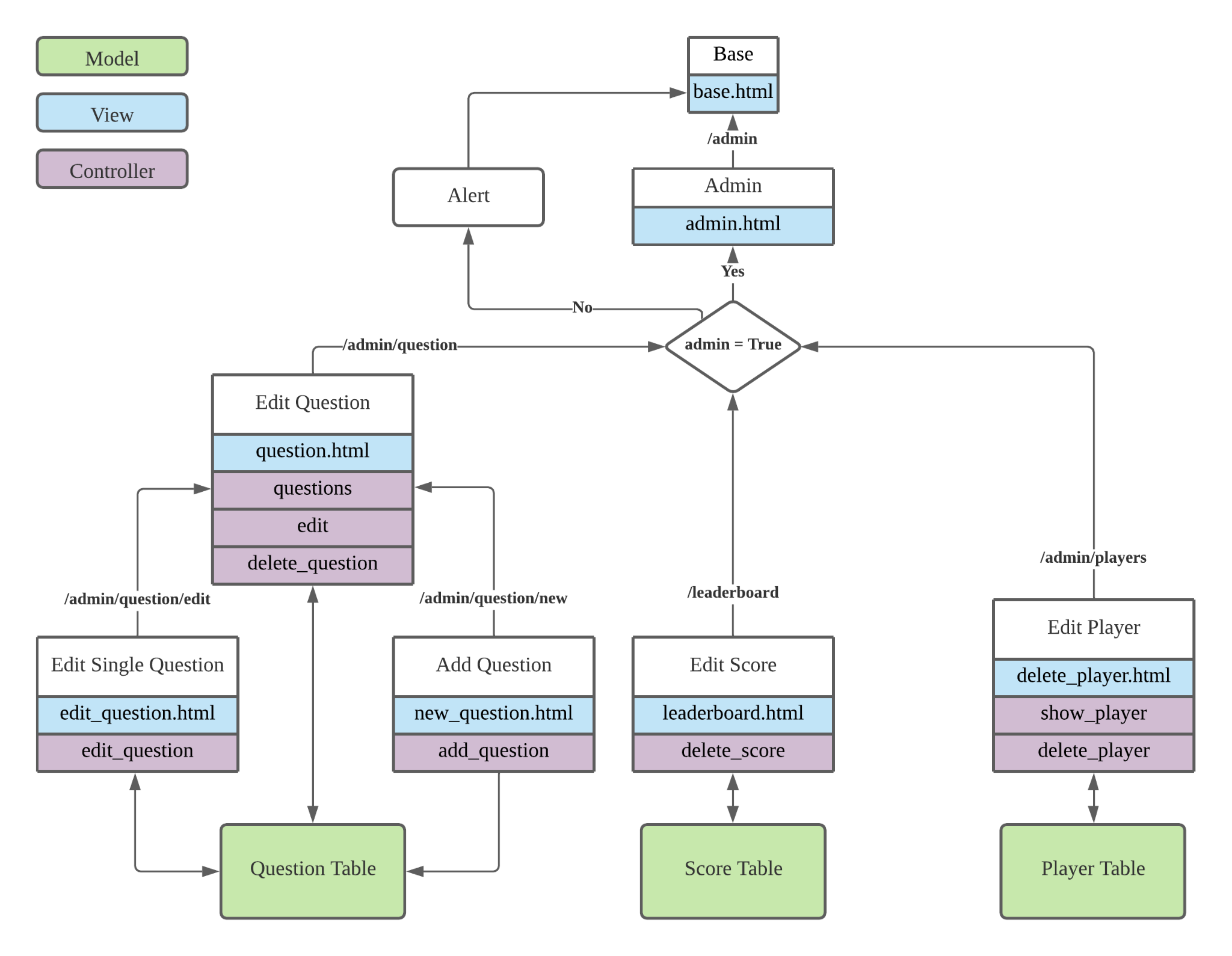
An example: When the player clicks the "Play" button on the user interface, or clicks the "Next" button on the Game interface, the Controller will convert the button corresponding to the Click into the action of "display new question." This action will provide an ajax request to the backend function which has the same route. It will return a new question. The data returned will then be displayed on the template.

We are using SQLAlchemy as SQL toolkit and object-relational mapper because of its reliability and the flexibility of SQLite. Also, SQLAlchemy has good compatibility and well written documentation for the Flask framework.

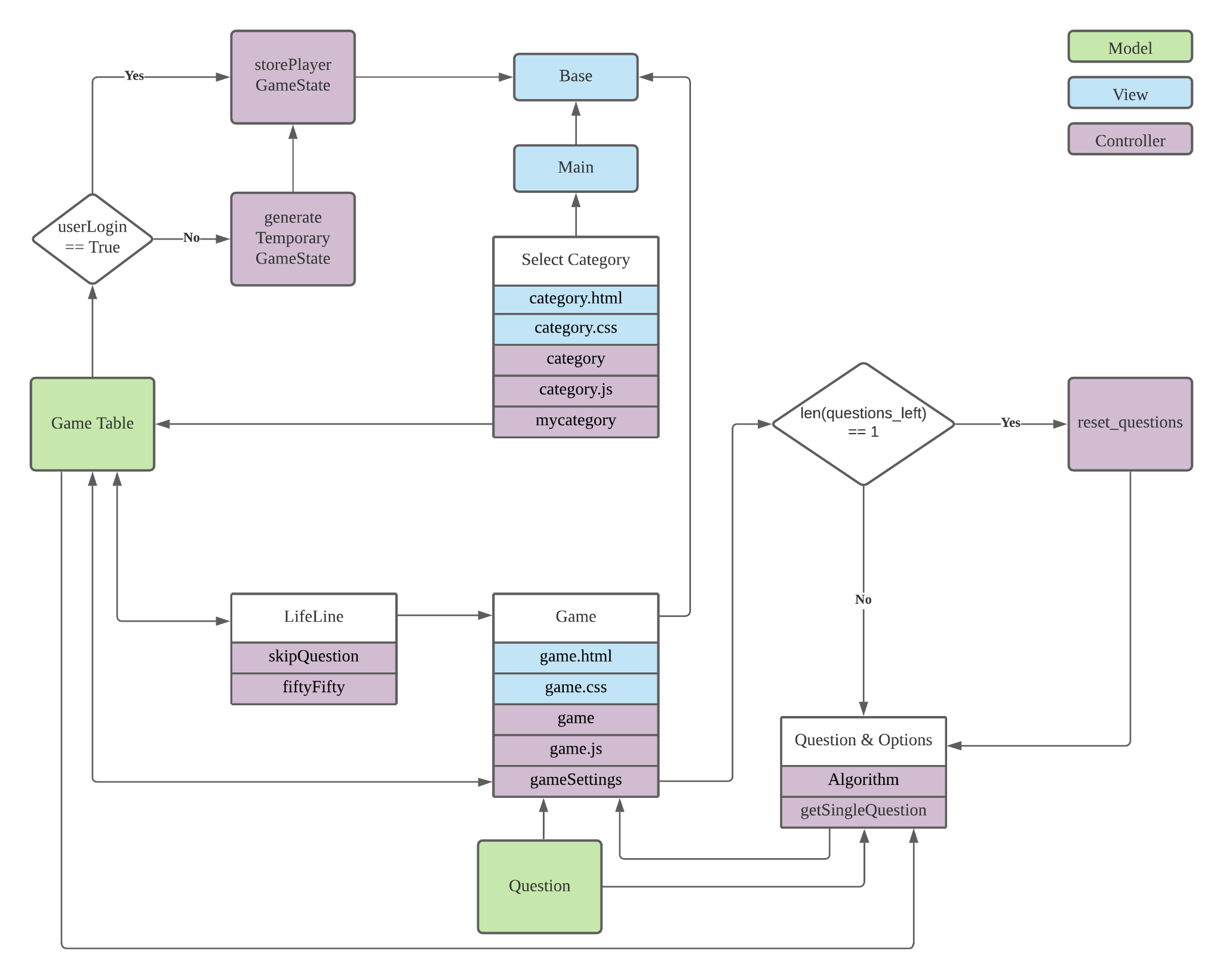
# Architecture Diagram



[Player Feature Logic](https://lucid.app/lucidchart/d9de2161-625b-490b-b8fa-5d44be8b05d1/edit?view_items=TGBy6P.5FYmP&invitationId=inv_6eba4ea2-71df-49ce-9caf-e44a6c99f21b)



[Admin Feature Logic](https://lucid.app/lucidchart/66d0e069-2c54-41a9-877c-49c4f958c6de/edit?viewport_loc=70%2C65%2C2033%2C1116%2C0_0&invitationId=inv_45250d9a-405d-4635-bf1d-353a8d0ab24a)



[Game Feature Logic](https://lucid.app/lucidchart/21bc5a90-1b5a-4162-8b10-685339c53557/edit?viewport_loc=-192%2C-137%2C2776%2C1524%2C0_0&invitationId=inv_071b5abb-4c08-4cda-a606-8832f057ec0b)

# Database Design (if applied)

The design displayed by the ERD is the design of the project we expect to be completed in the future. Our database design will continue to iterate according to the progress of the project.

In iteration1 we only need to use the database to store questions, options and answers, as well as player scores. We use SQLite to design our database, and what we got are:

Questions table

LeadeboardScores table(Score table)

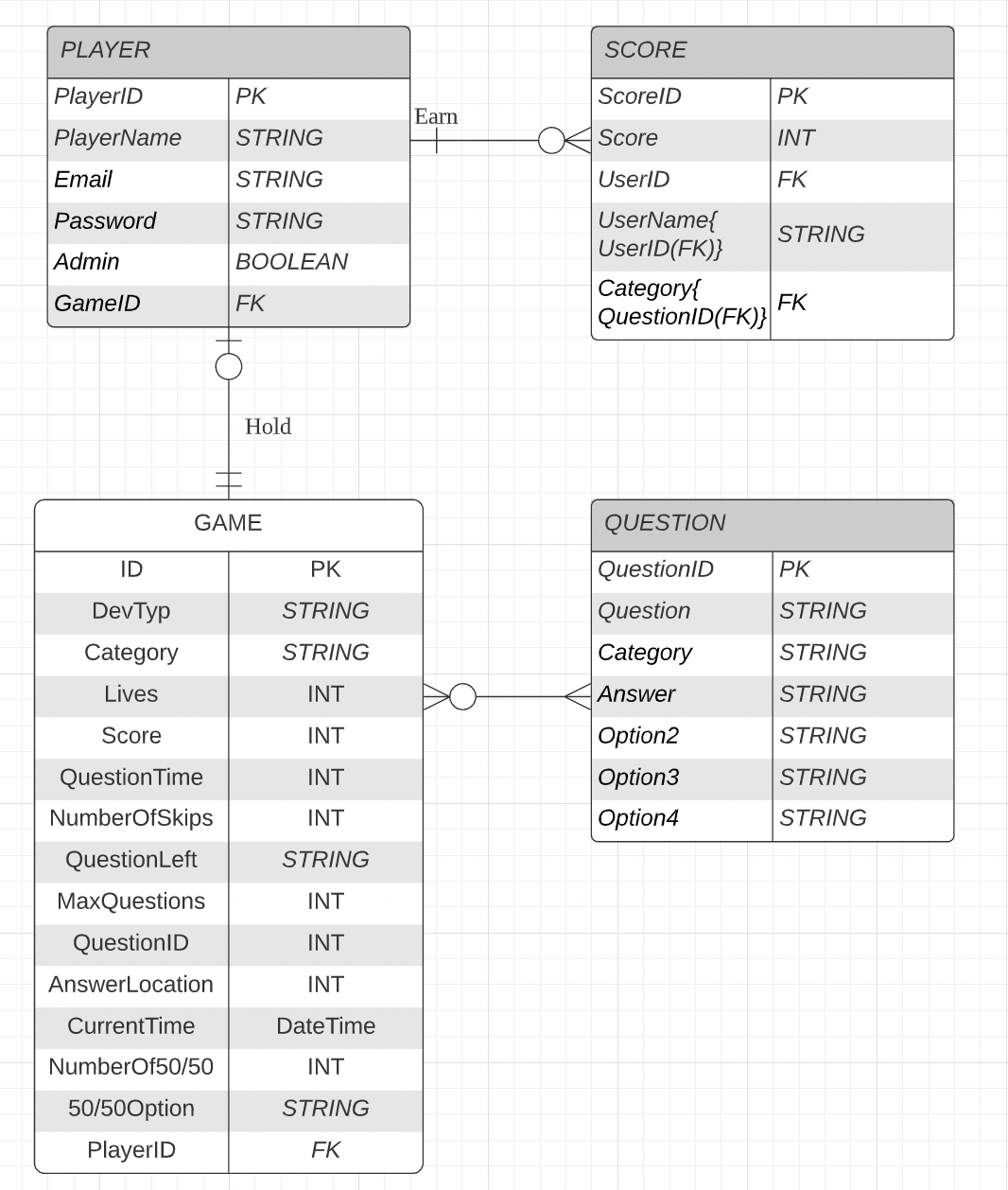
GameTable

Player Table

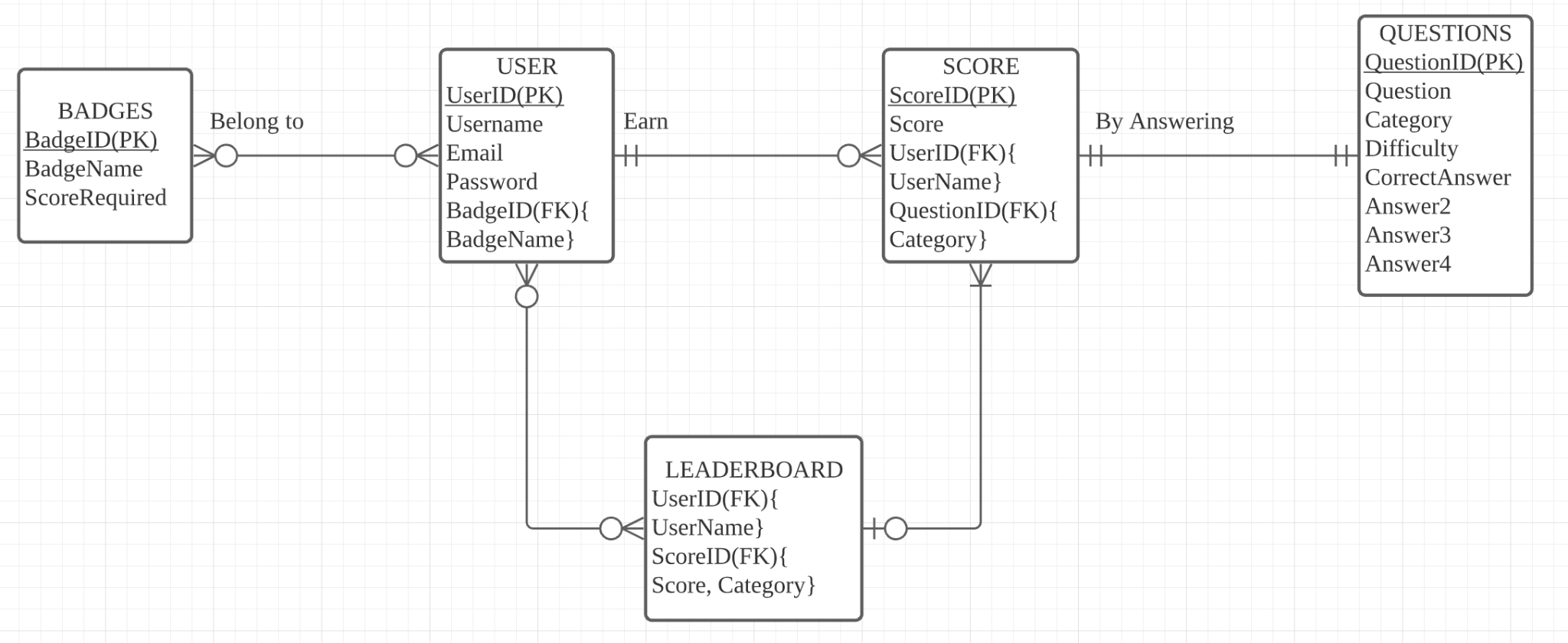
The categories in the scoring table are columns reserved for the next iteration, and we will play the game according to different categories. Currently the column has no practical usage

In iteration 2, we have Question table to store all the questions related information. Then we have Score table which has been called “LeaderboardScore” to store the score, category, userid, and username. This table only stores the information for those players who are logged in. The next table we have for this iteration is Player table to store information for those players registered, like their userID, user name, email address, and password. The last table we got is Game table. This table is used to store data that was created during the gameplay. Because we moved all the data from front end to back to make sure that no one can cheat, then these data need to have a table to store.

In iteration 3, we added admin to the Player table. Admin is a boolean value to check if the login user is superuser.



[ERD iteration 3](https://lucid.app/lucidchart/29131e2d-b1fb-448f-90d7-81942d2b2606/edit?viewport_loc=-107%2C-15%2C1729%2C949%2C0_0&invitationId=inv_6f9eb19c-7a04-4696-a354-2a6d32d07a66)



ERD plan during iteration 0:

<https://lucid.app/lucidchart/7e677350-d219-4499-b694-e9525730fa0d/edit?invitationId=inv_1ec78826-1a09-4e0e-8af6-b93c7c0a5d39&referringApp=slack&page=0_0#>

# Security Design / Plan

In this iteration, our prioritized concern is to implement an overall playable Trivia game. Therefore, the security concerns and proposed solution plans will be implemented in the next iteration accordingly.

In the iteration2 build, all game variables are stored on the backend so that they cannot impact gameplay by being directly edited on the frontend. The answer location is also hidden from the front end until the submit button is pressed, which makes it so users cannot simply look at the javascript variables to learn the answer.

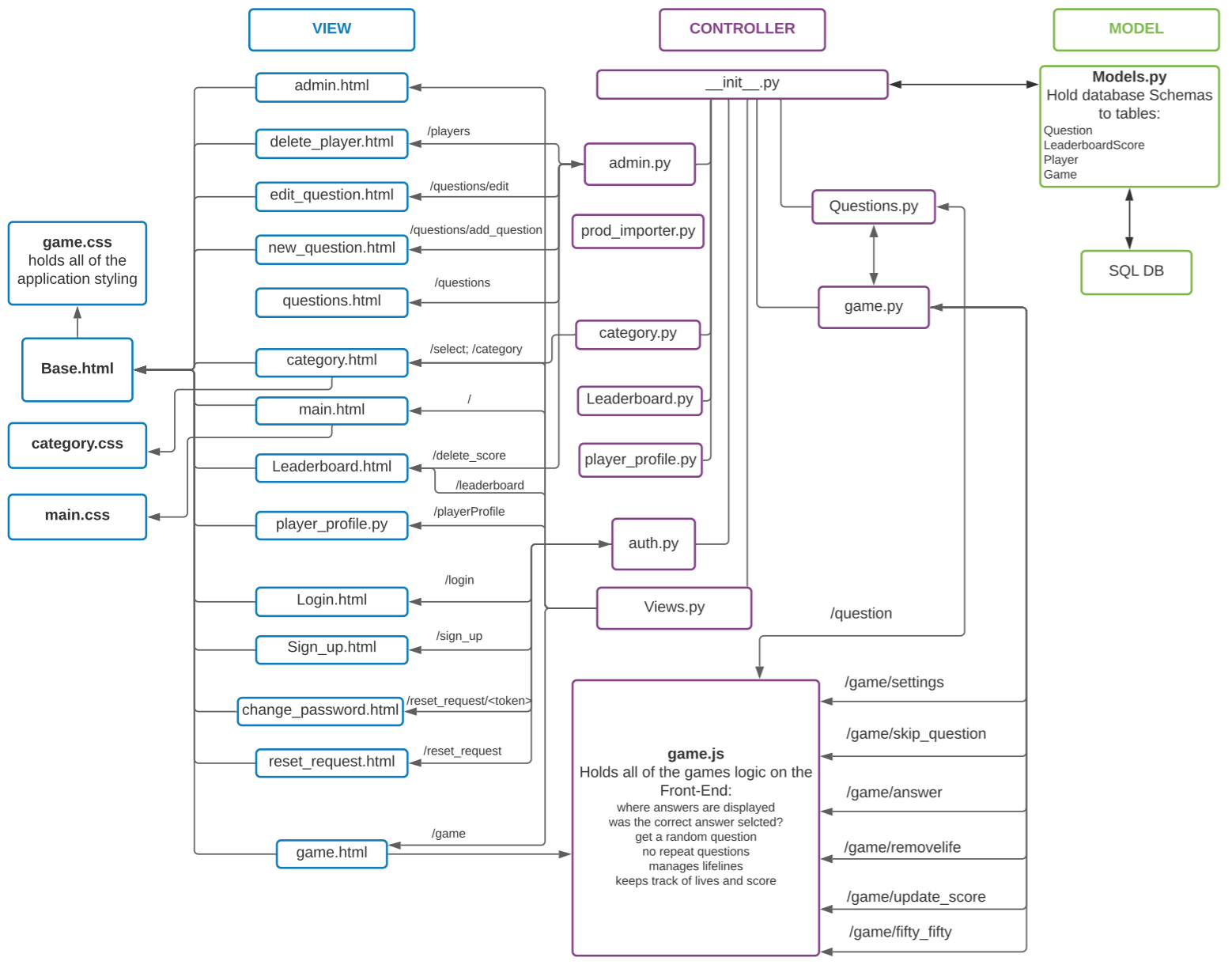
Alongside with the player profile feature being implemented in the next iteration, there will be several new possible concerns regarding the existing user stories:

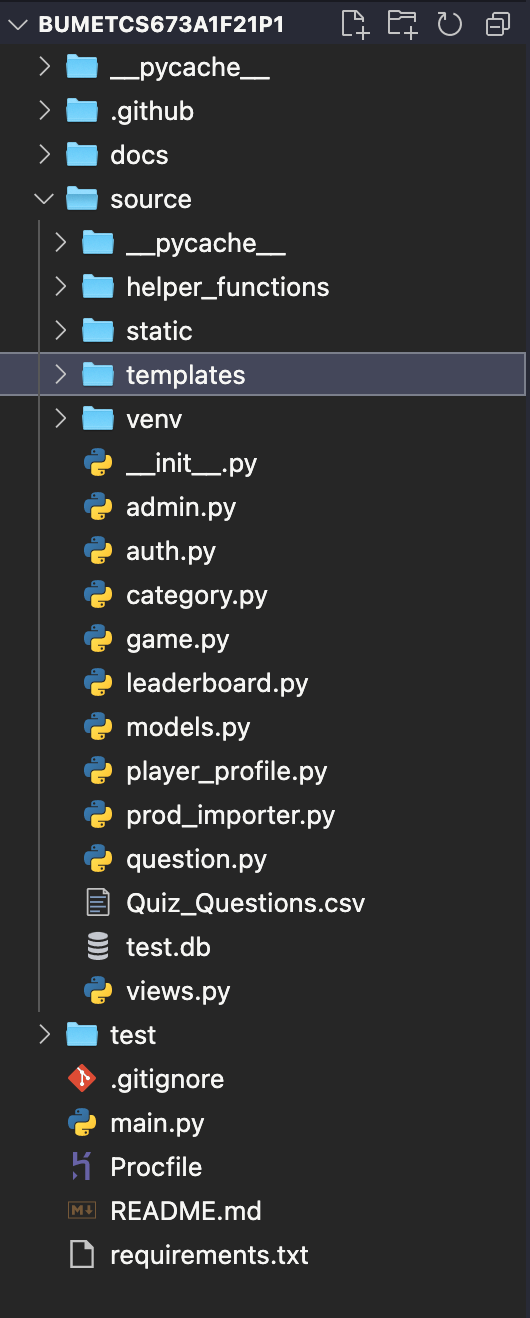
* For players who register or want to keep scores on the leaderboard:
  + If player register with username/id & password to login:
    - How to make sure passwords are secure in the backend
    - Get permission to link player’s id info with the scores and store them in the backend
  + If player use google or other login api:
    - Link the username/id and scores with the email address
* For registered/login player’s profile info:
  + Save info in the backend: the security of the backend infrastructure
* For players who do not login/register or want their scores on the leaderboard and backend database:
  + Get the player’s permission to use their ip address to record the temporary game score data and delete it when player decide not to keep score or not register/login

# Design Patterns

Our application is built with the Flask framework. Given the Flask’s loose design the application does not follow any traditional design pattern.

**Current Design:**



**File Structure:**

**/BUMET673A1F21P1:** Root directory of Application

**/docs:** contains all documents related to application

/**source**: directory where all source code for application lives

**/static:** stores all javascript and CSS files

**/templates**: stores all html files

**\_\_init\_\_.py:** intilizes and connects all back end files

**auth.py**: all backend logic for authorization

**game.py**: all backend logic associated with in-game variables

**Import\_questions.py**: inserts questions from csv file into the database

**Import\_questions\_prod.py**: inserts questions from csv file into the database for production database

**Leaderboard.py:** All backend logic for leaderboard table

**models.py**: database schemas for Questions, Leaderboard, Players, Games

**Question.py**: All backend logic for the question table

**Quiz\_Questions.csv**: All quiz questions

**Requirements.txt**: Lists all libraries needed to run application

**test.db**: the physical database

**Views.py:** controls which html templates are shown

**/testing**: contains all code related to testing

**.gitignore**: lists all files for git to not commit

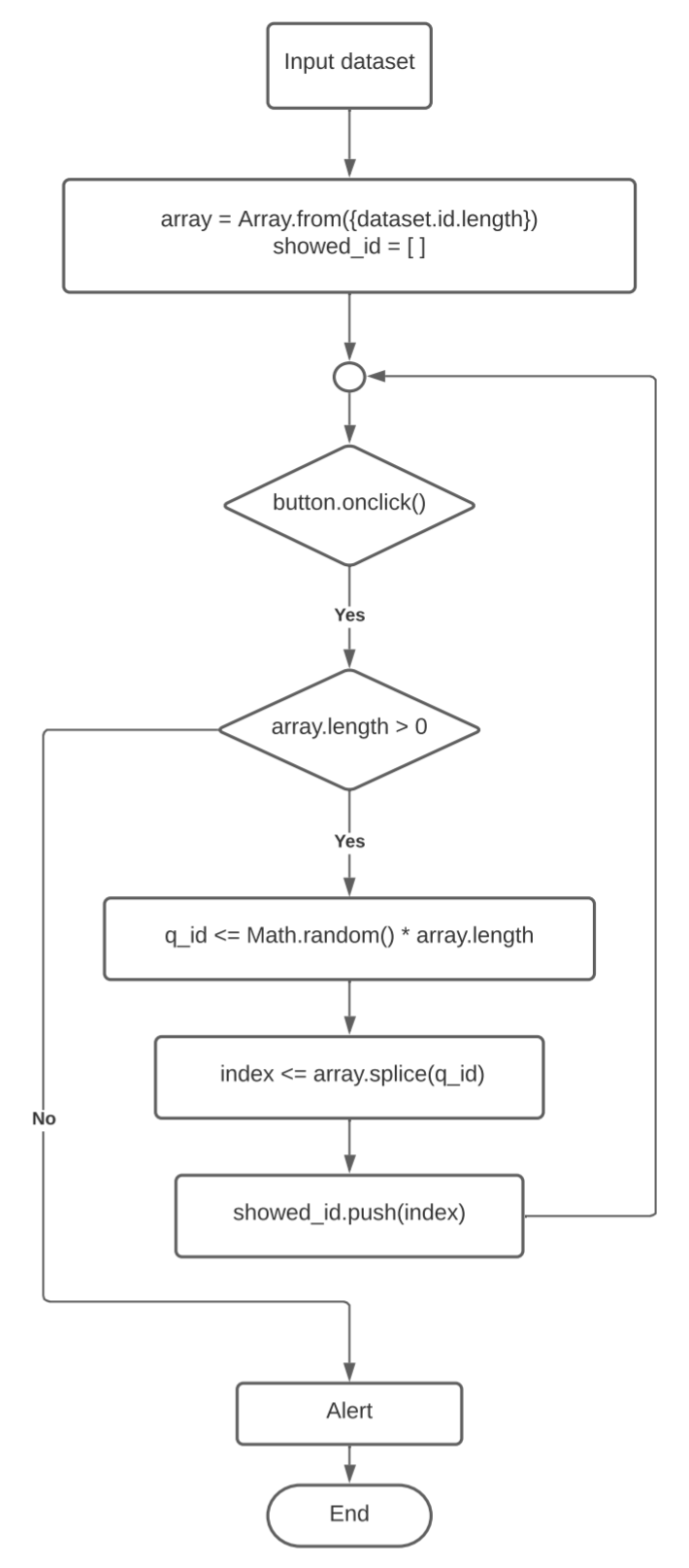
**Main.py:** the starting point for the app

**README.md**: Explains how to run application

**Team.md**: Info about team members

# Key Algorithms

1. No duplicated questions Algorithm



In this algorithm we use the flask framework to input the dataset once. Then two variables are created, one is an arraylist for storing all question\_ids, another called showed\_id for storing the question\_ids that have already been shown once during the game. The arraylist will be randomly sorted once at the beginning.

Every time the user clicks on the button for a new question, the system will determine the length of the arraylist, to know how many questions have never been shown on the page. If the length of the arraylist is greater than 0, that means there at least 1 or more questions have not been shown on the game page.

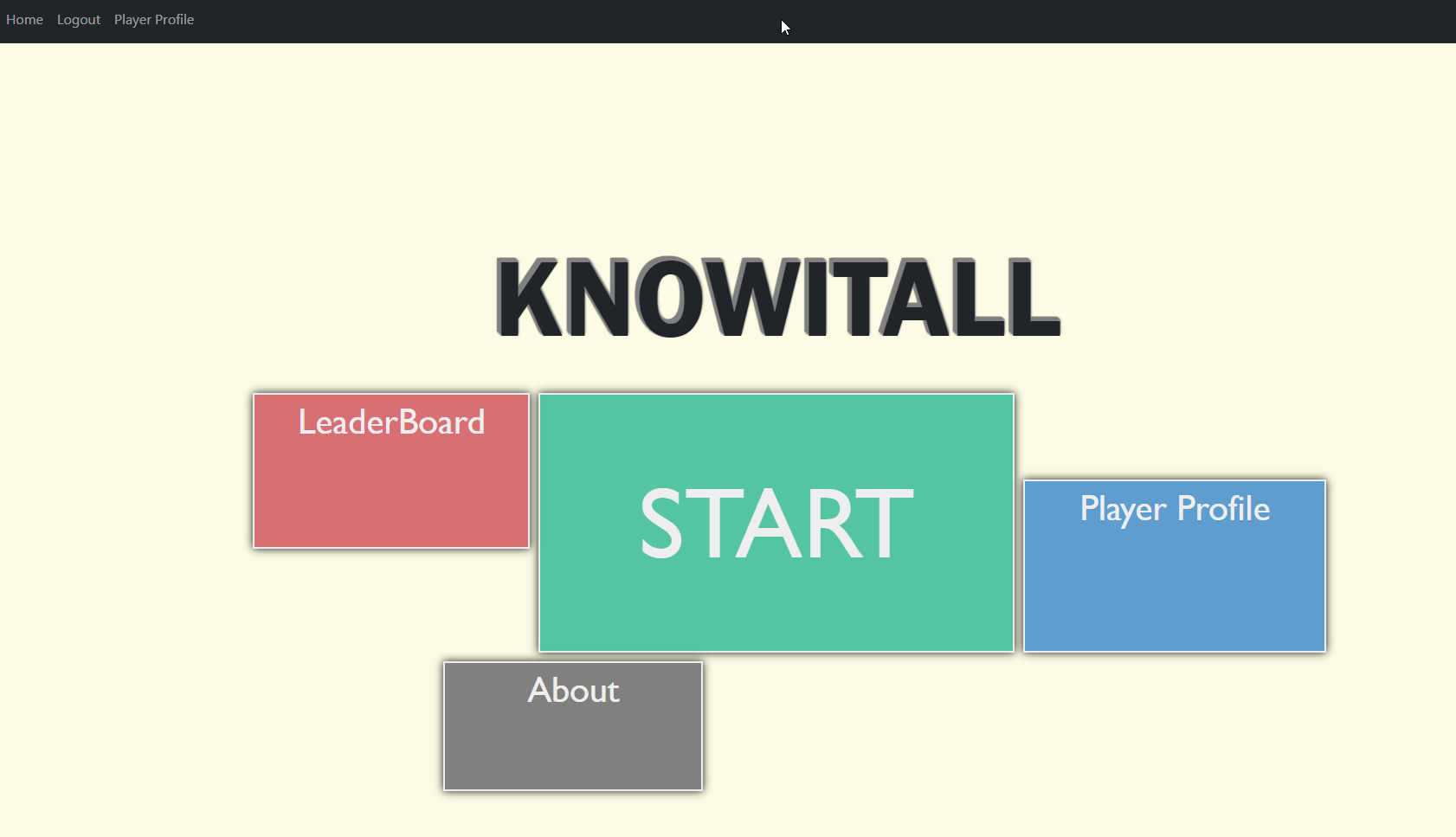
The code will choose the first number of the arraylist that has been sorted randomly. The algorithm will then delete the question\_id in the arraylist at the position of the value q\_id. Deleting the question\_id makes the arraylist one element smaller after each click.

This deleted question\_id will be the id of the next question that shows on the game page. After that, the system will store the deleted question id in the showed\_id arraylist. Data stored in the showed\_id arraylist is mainly for testing purposes.

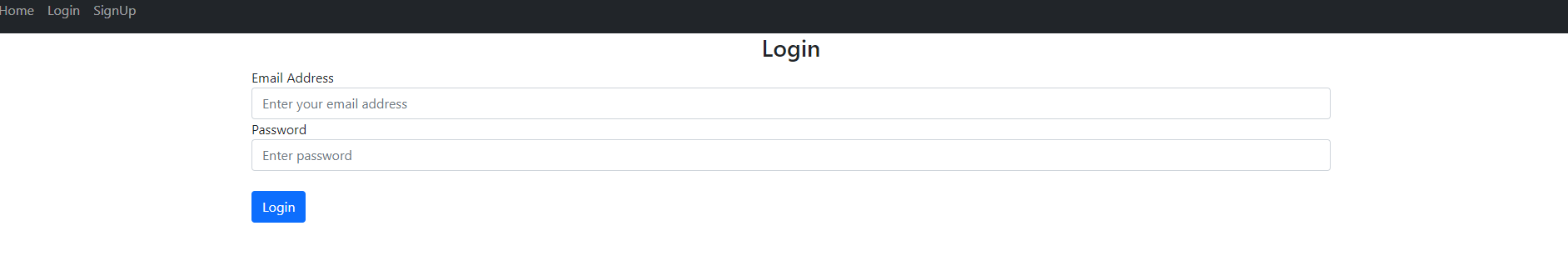
If the length of the arraylist is equal to 0, that means all the questions have been shown on the game page once, the system will pop up an alert to notice the player finished all questions, and stop.

# UI Design

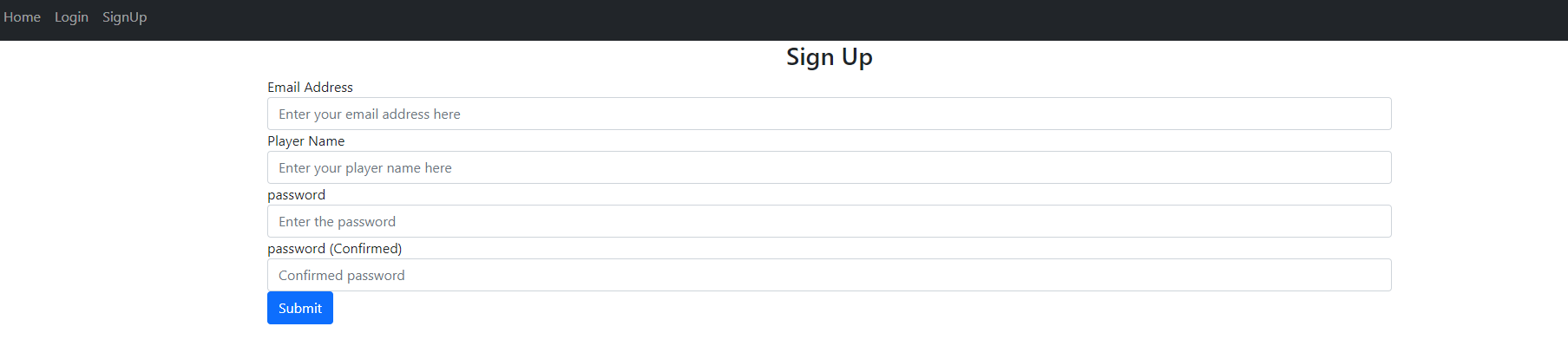
**Main.html**



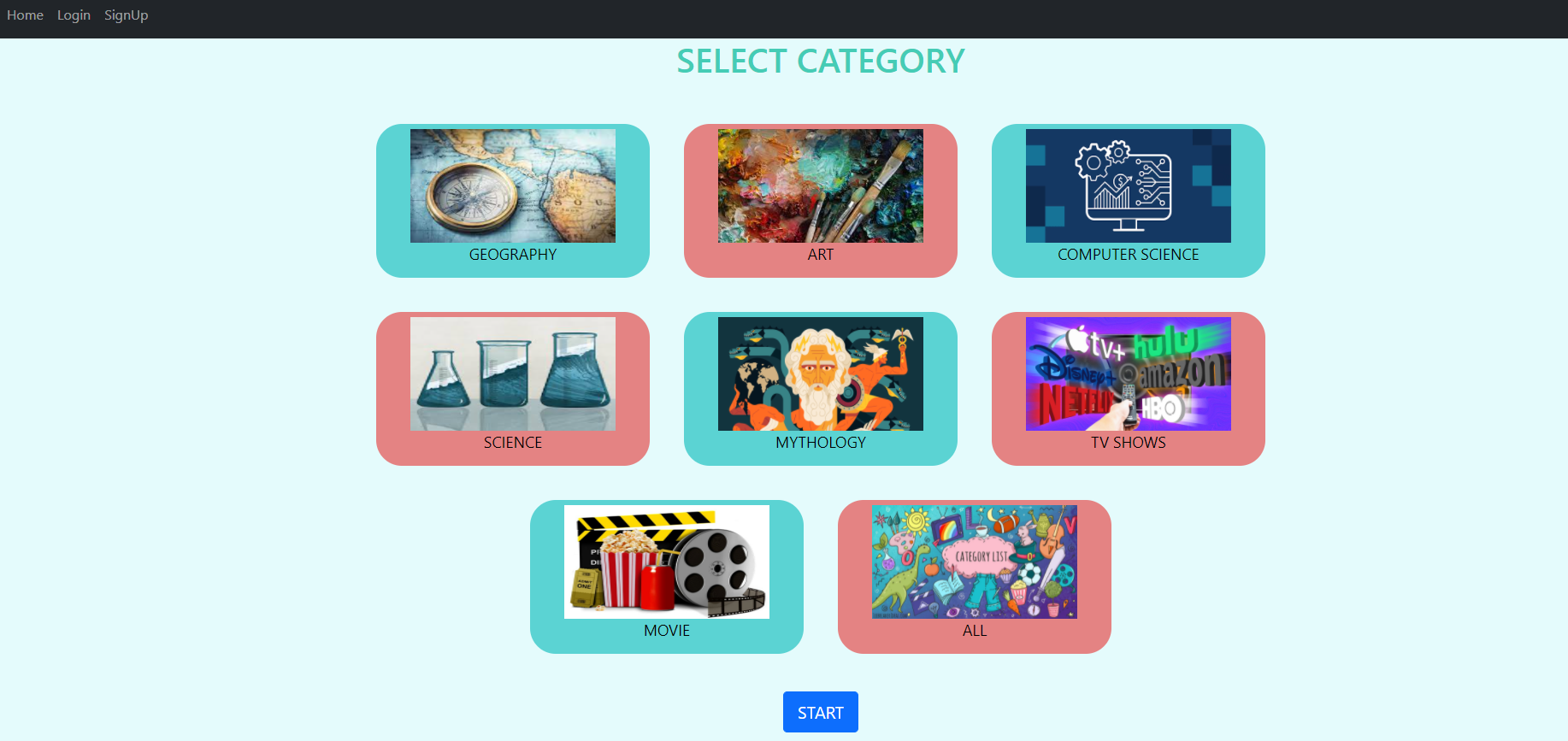
**Login.html**



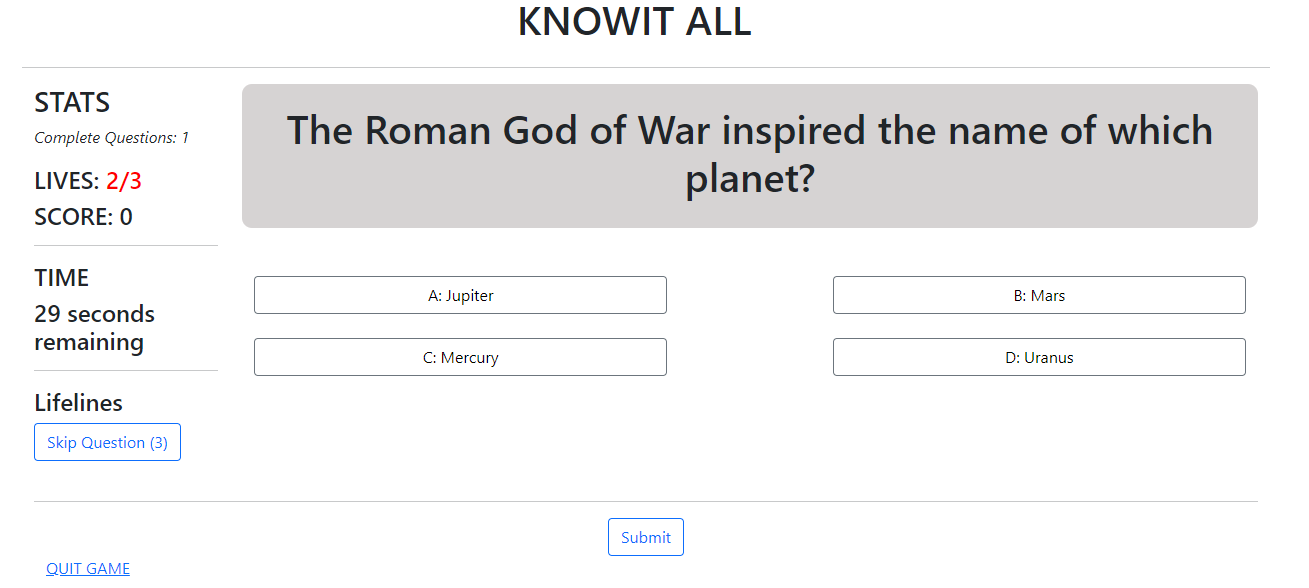
**Sing-up.html**

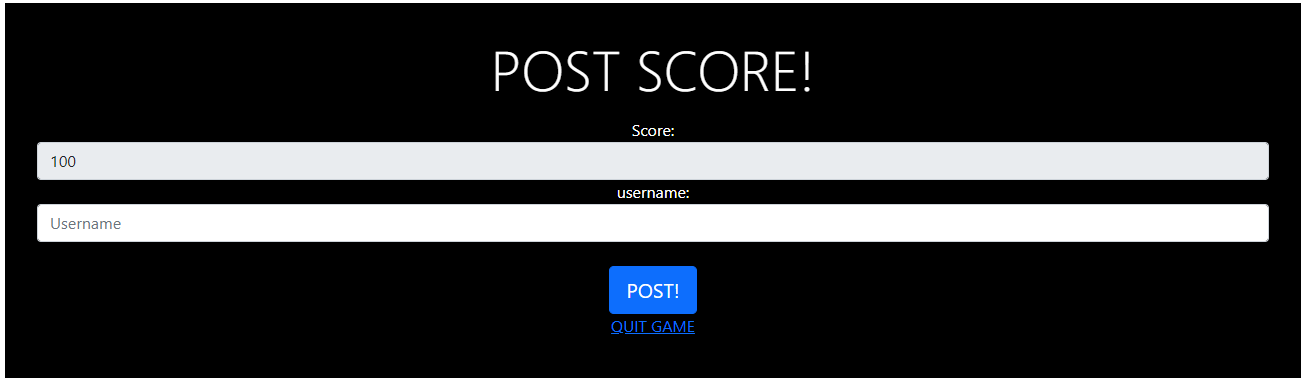


**Categories.html**

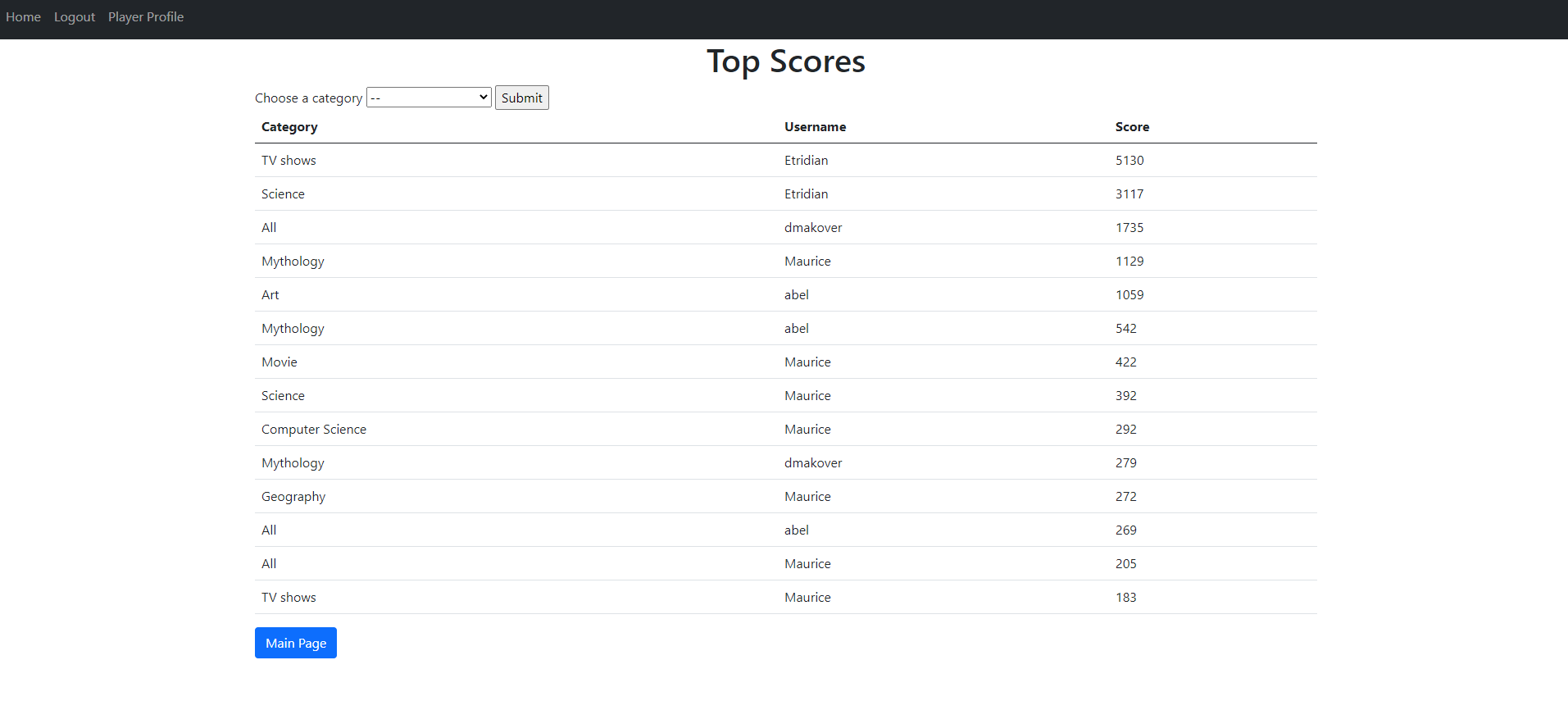


**Game.html**

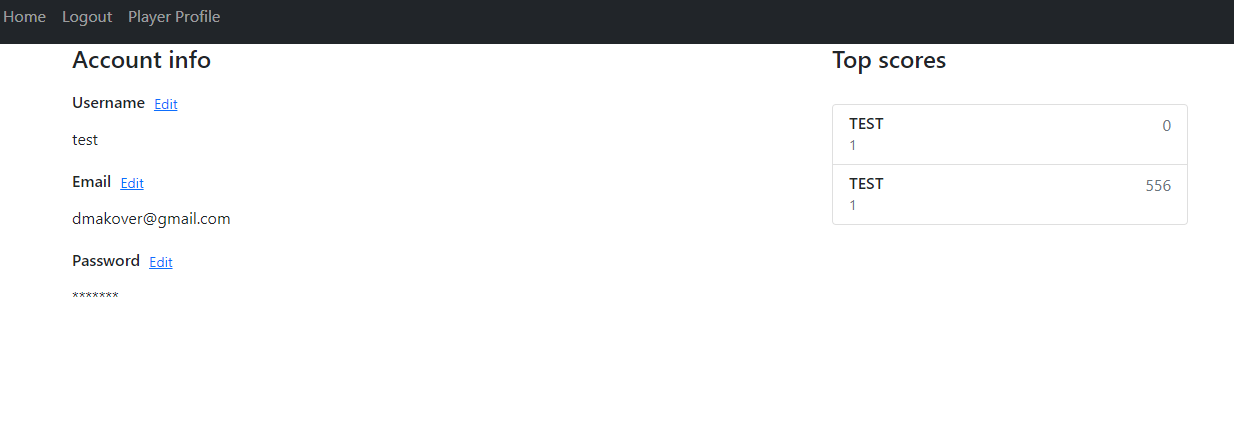




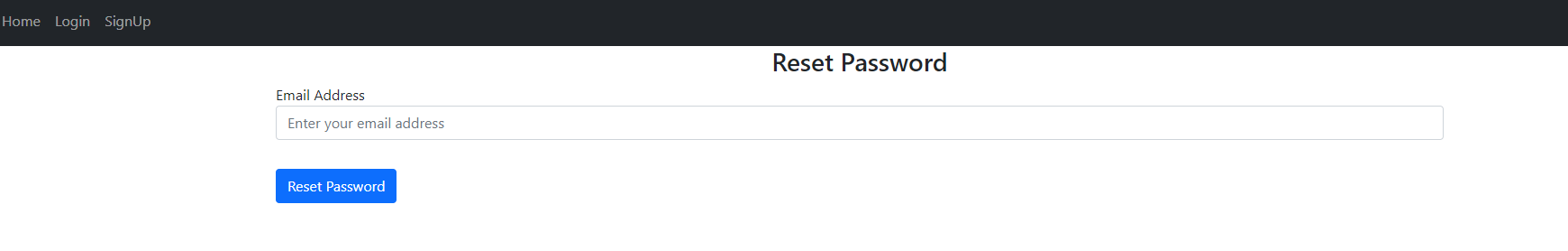
**Leaderboard.html**



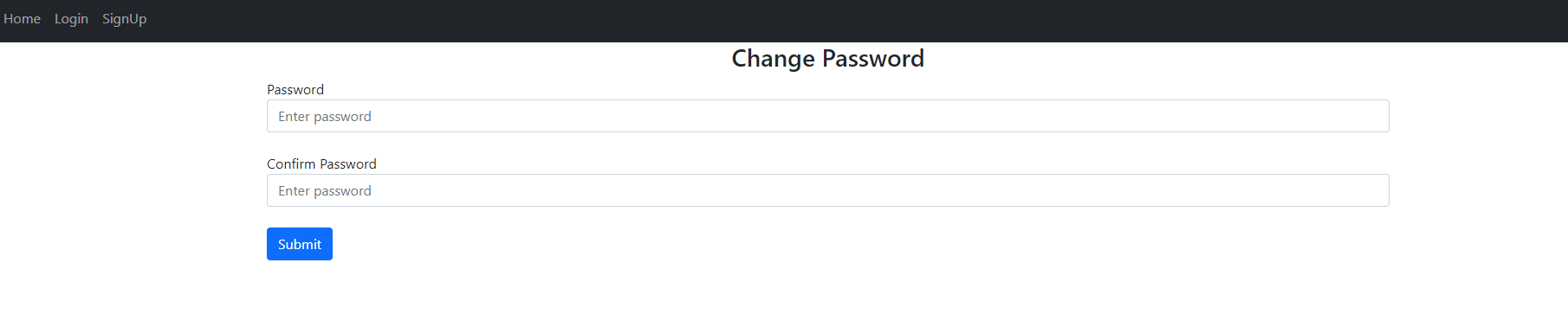
**Playerprofile.html**



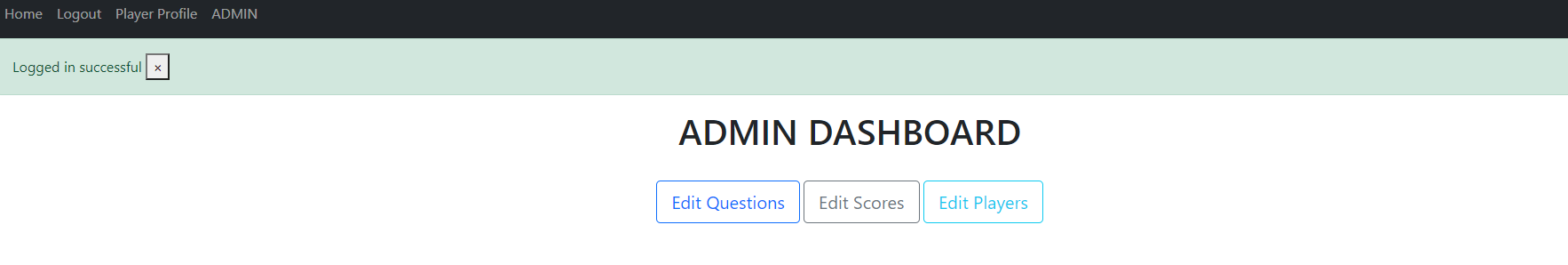
**Reset\_request.html**



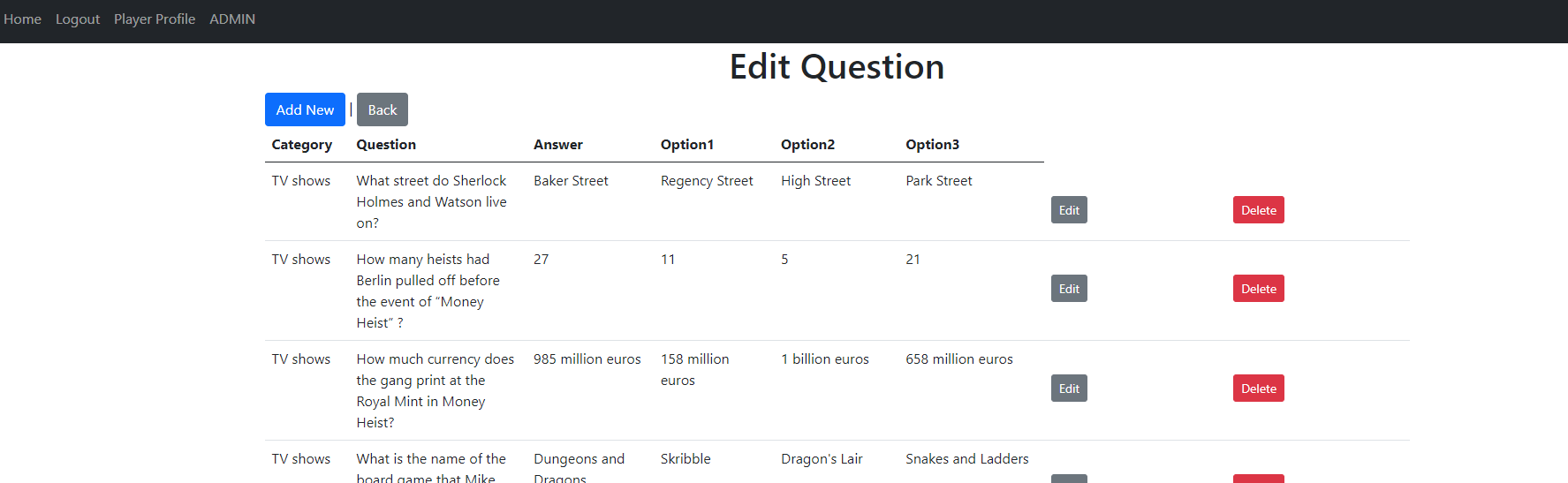
**Change\_password.html**



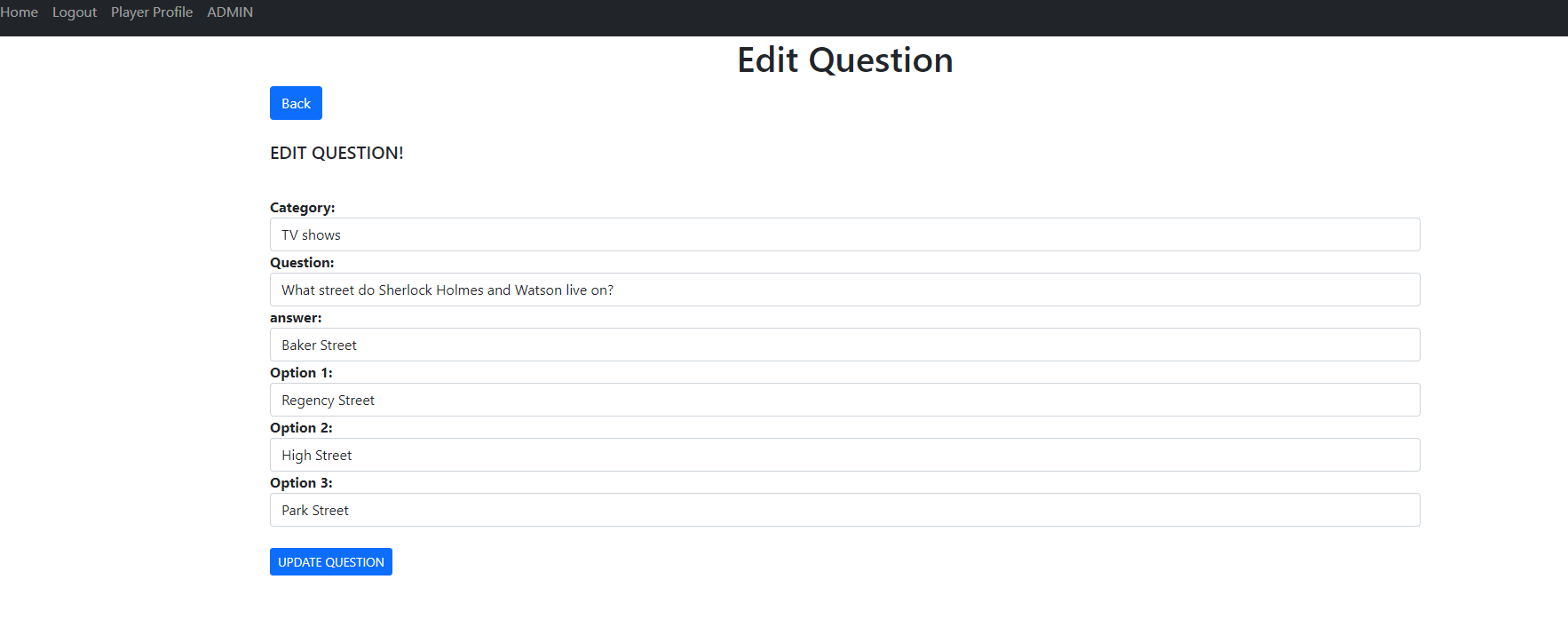
Admin.html



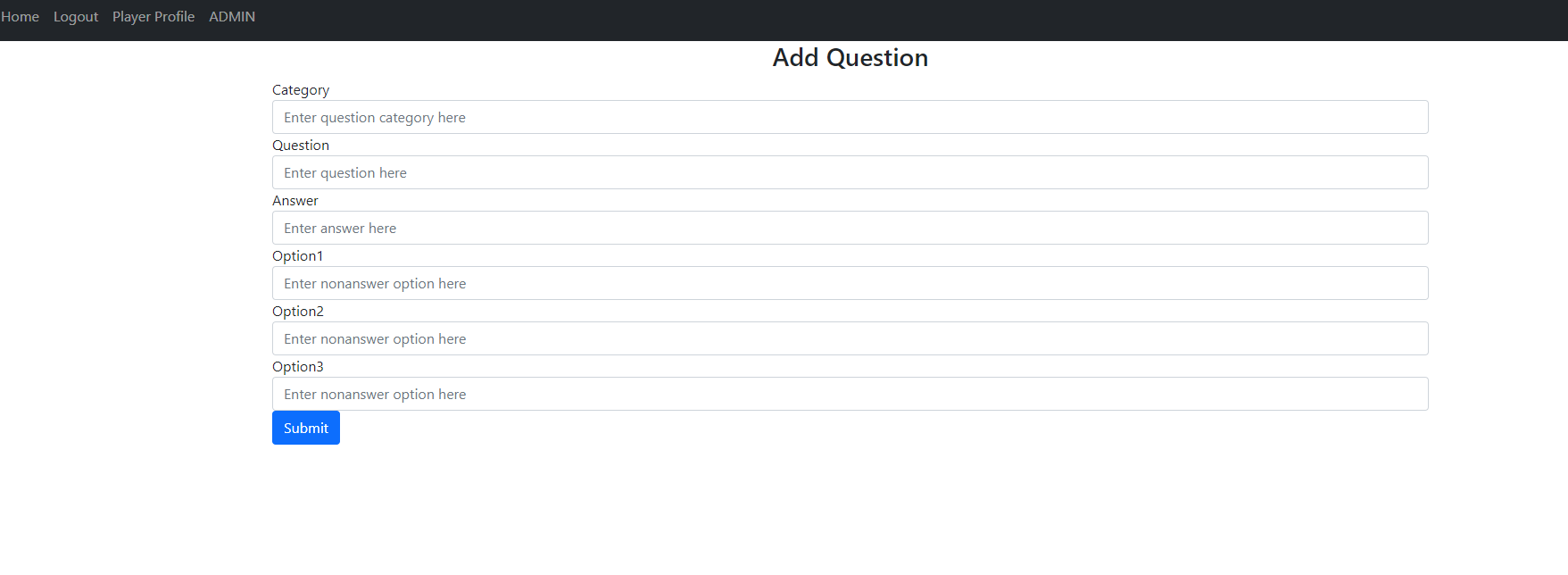
**Questions.html**



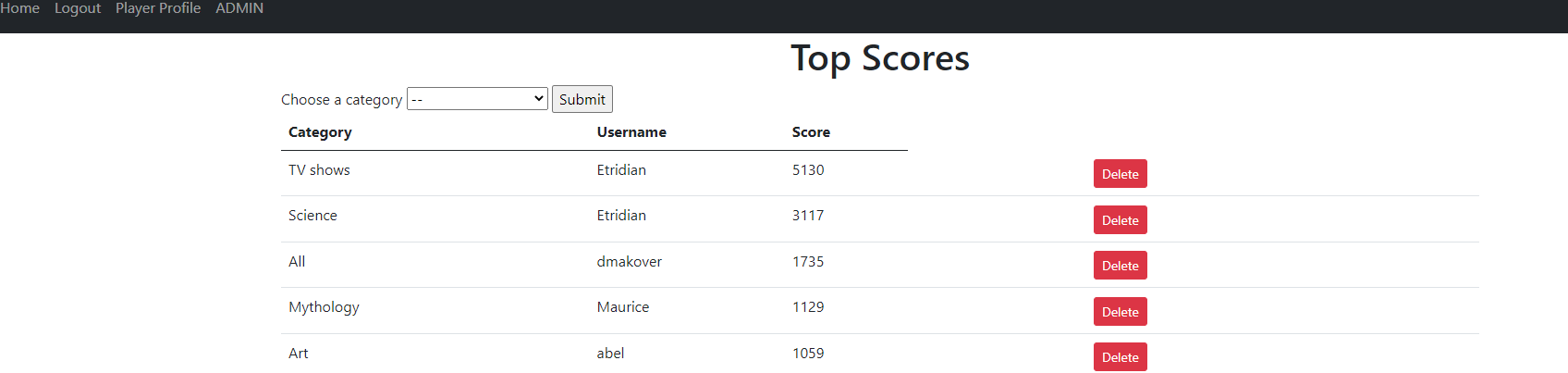
**Edit\_questions.html**



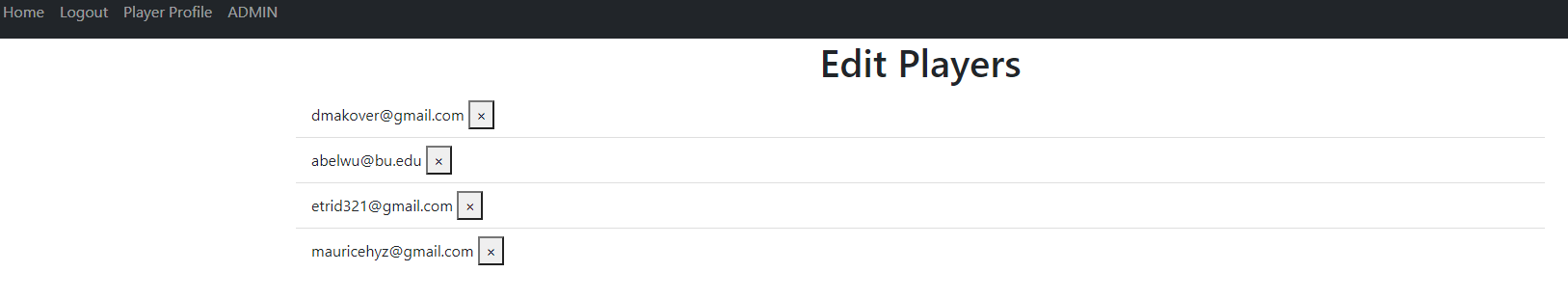
New\_question.html



Admin\_leaderbarod.html



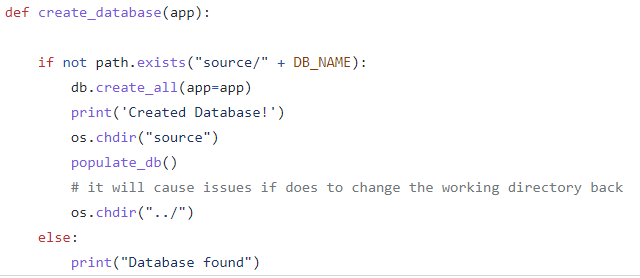
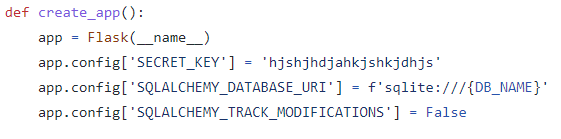
Player\_delete.html



# Classes and Methods

Our classes are contained within the init.py file

Some of the classes that we have created are as follows:-



# References

1. <https://refactoring.guru/refactoring/smells>
2. <https://docs.pytest.org/en/6.2.x/>
3. <https://www.digitalocean.com/community/tutorials/how-to-add-authentication-to-your-app-with-flask-login>
4. <https://www.selenium.dev/>
5. <https://devcenter.heroku.com/>

# 