**CS673 Software Engineering** 

**Team 1 - KnowItAll**

**Software Design Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Daniel Makover | Manager | *Daniel Makover* | 2021-10-20 |
| Yuwei Wu | Security Leader | *Yuwei Wu* | 2021-10-21 |
| Heli Kolambekar | Requirements Leader | *Heli Kolambekar* | 2021-10-20 |
| Gunnar Nichols | QA Leader | *Gunnar Nichols* | 2021-10-21 |
| Haoyi Zhu | Design and Implementation Leader | *Haoyi Zhu* | 2021-10-20 |
| Kun Mo | Key Algorithm | *Kun Mo* | 2021-10-20 |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

[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 singleplayer, multiplayer 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

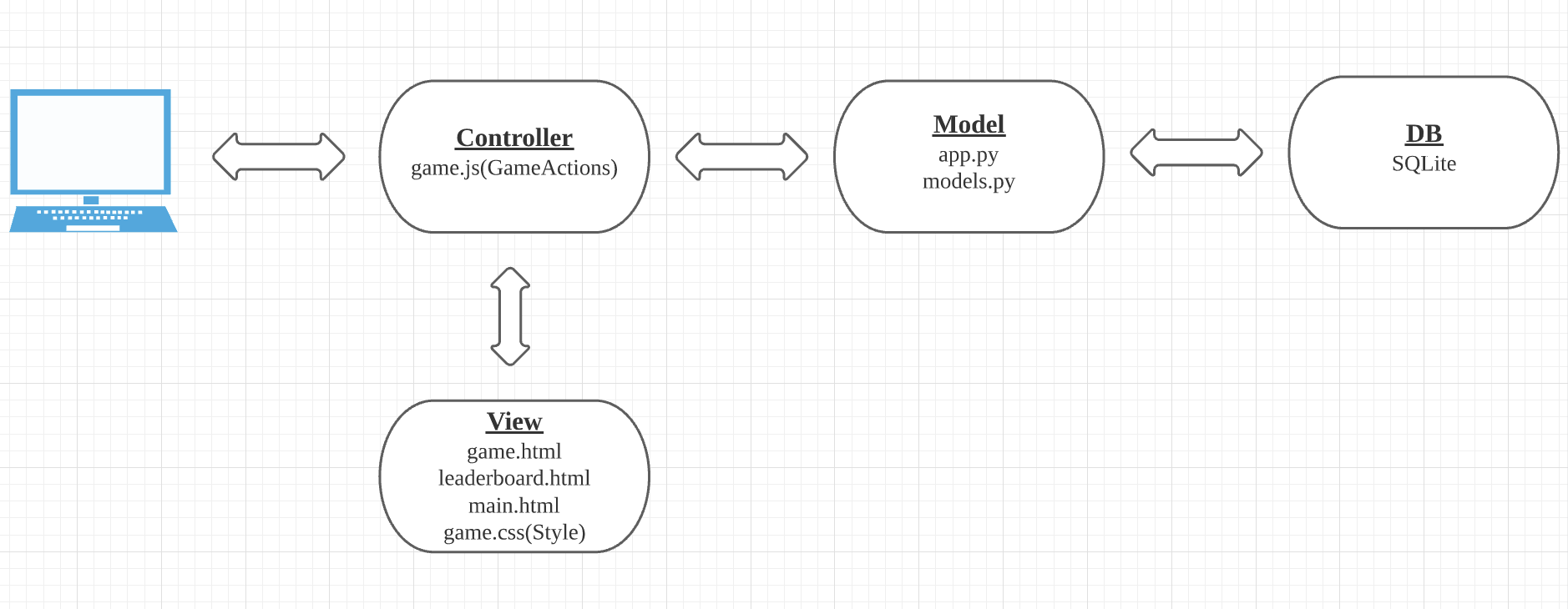
In our KnowItAll Web App, our design uses Flask as a framework and SQLite as a database, following the model-view-controller architecture.

Our controller is the bridge between the user interface and the model. 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 and logic in the model.

In the Model, we have the getQuestion function to get questions from DB, then we have logic that randomizes the question options returned from the database and identifies 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.

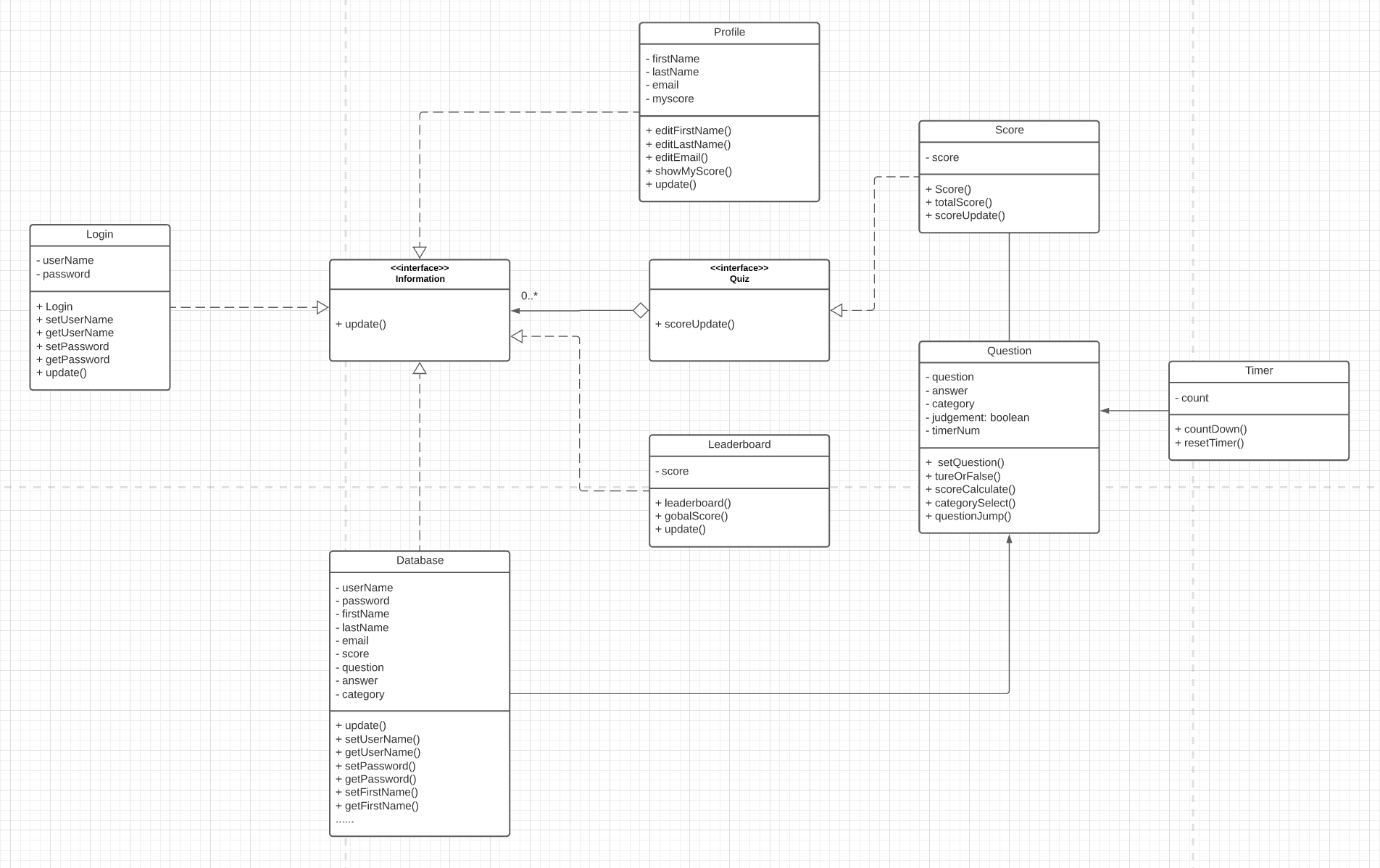
An example: When the player clicks the "Play" button on the user interface, or clicks the "Next" button on the Game interface, the Controller will convert the button corresponding to the Click into the action of "Show new question." This action will provide a request to the Model for a new question with randomized options. 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.



Flask MVC Like View:

<https://lucid.app/lucidchart/021d944e-f63e-42ea-b257-068538ff09b1/edit?viewport_loc=-63%2C-128%2C1685%2C867%2C0_0&invitationId=inv_6c63f25d-a07d-4c5c-9fca-822dfc247e7e>



Class Diagram URL:

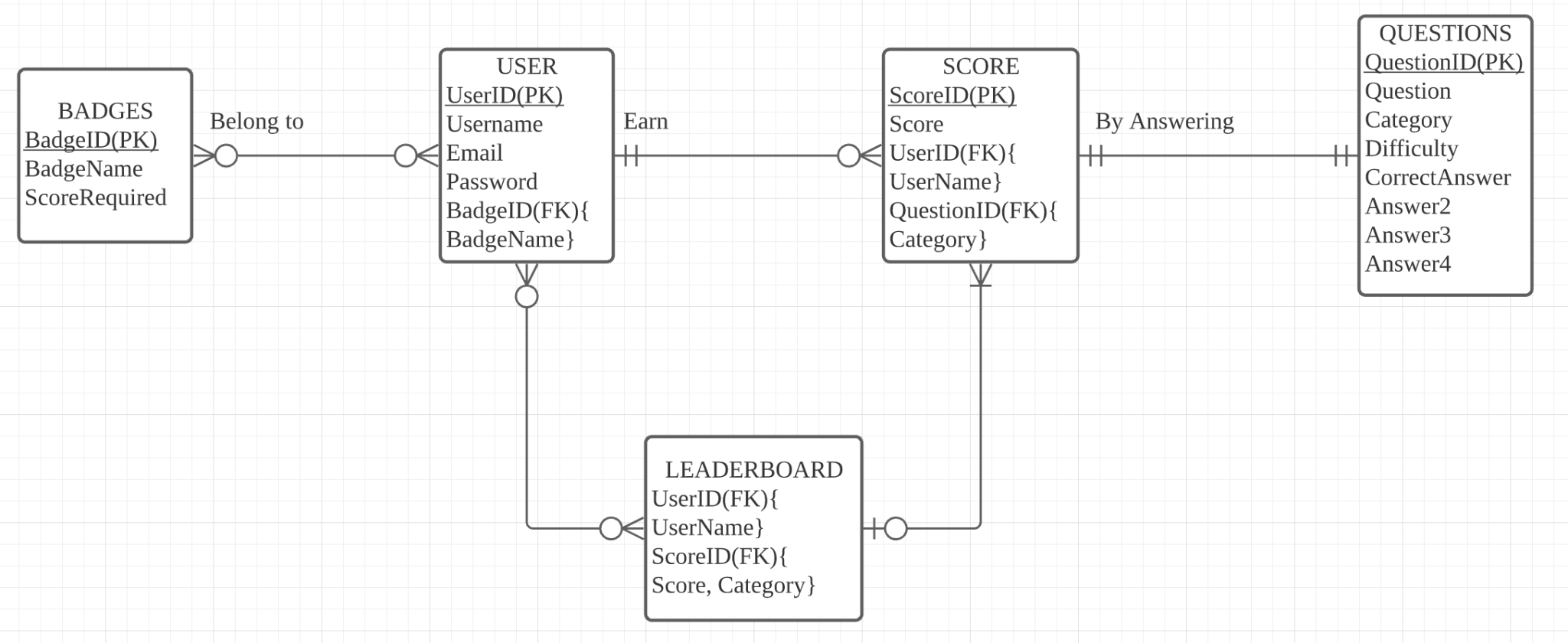
<https://lucid.app/lucidchart/862a79ff-ef9e-4105-9e1e-a6b4d19c2f5e/edit?viewport_loc=-2341%2C-1832%2C5924%2C4287%2C0_0&invitationId=inv_562760b2-f3a7-4280-8d1a-b5411efd599a>

# 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 and LeadeboardScores 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



ERD:

<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.

Based on the current build, several features will need to be implemented with security design: the remaining lives, score for correct answers, time countdown mechanism, up limits for skipping questions. For these concerns, we plan to move the relevant variables to the backend in order to prevent players from modifying the front end values. This process might need to implement a new table to store the variables in the backend.

Another concern is that, in the current building the players can easily find the correct answer in the frontend script while playing the game. We will try to find a way to use separate javascripts for server-end and client-end in the next iteration.

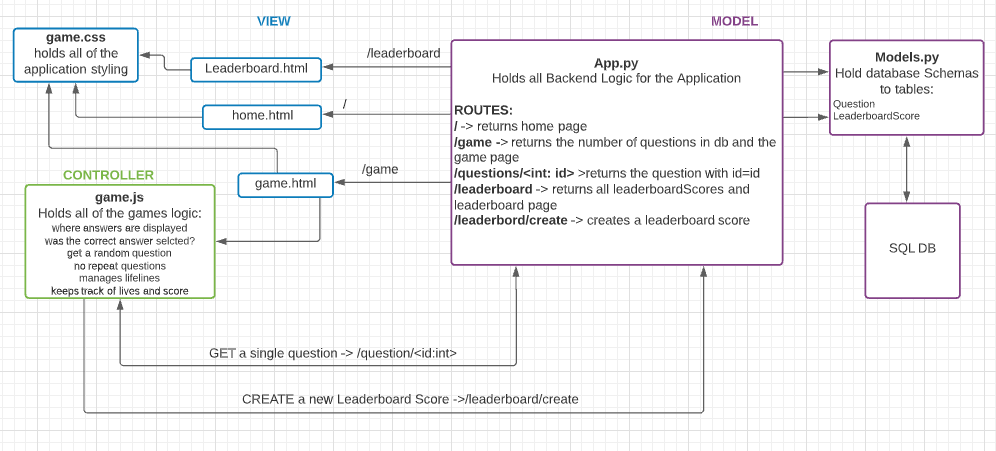
Alongside with the player profile feature being implemented in the next iteration, there will be several new possible concerns regarding the existing user stories:

* 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

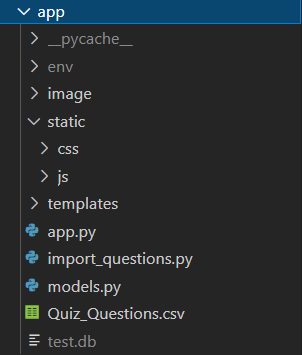
# Design Patterns

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:**

Everything is stored in the **/app** directory

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

**/templates**: stores all html files

**app.py**: all back-end logic

**models.py**: database schemas for Questions and Leaderboard

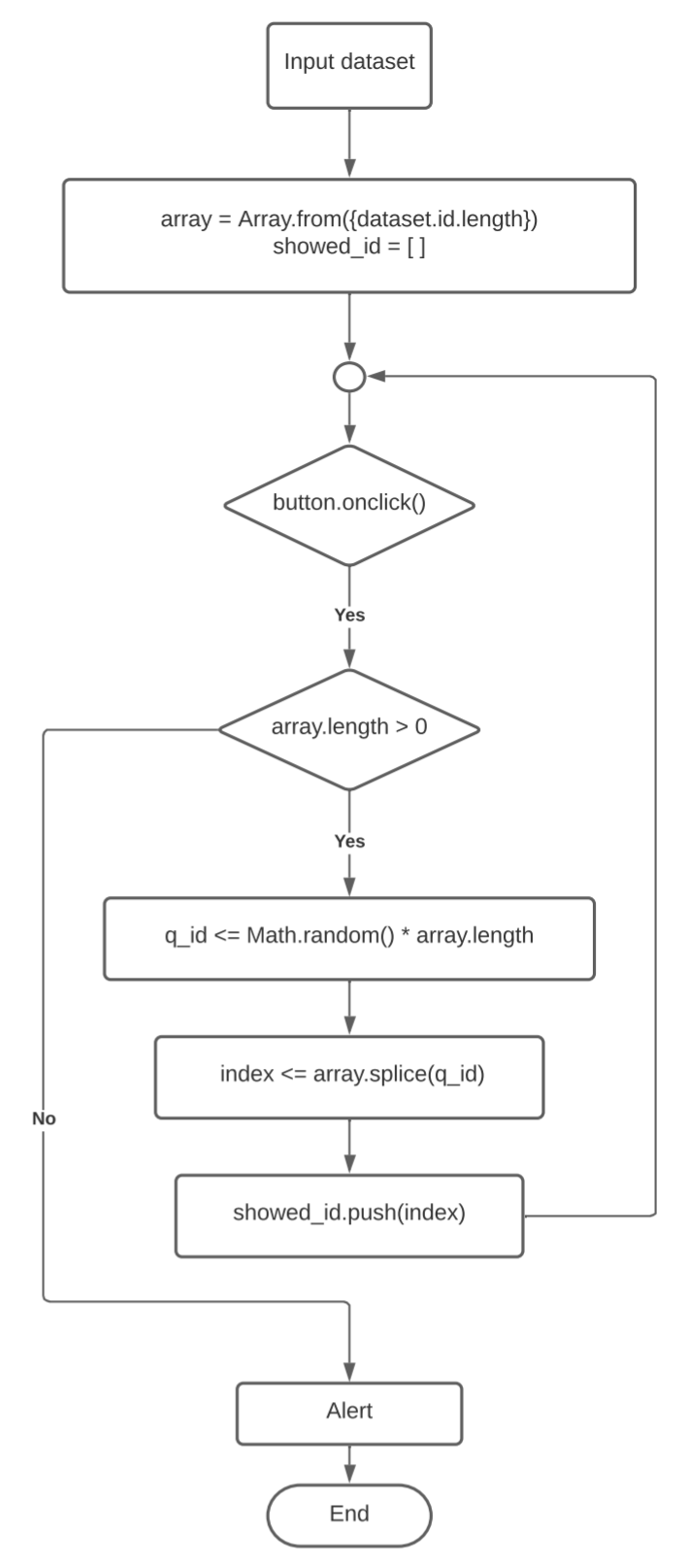
**test.db**: the physical database

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

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

# 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 called **array** for storing all question\_ids, another called **showed\_id** for storing the question\_ids that have already been shown once during the game. 

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

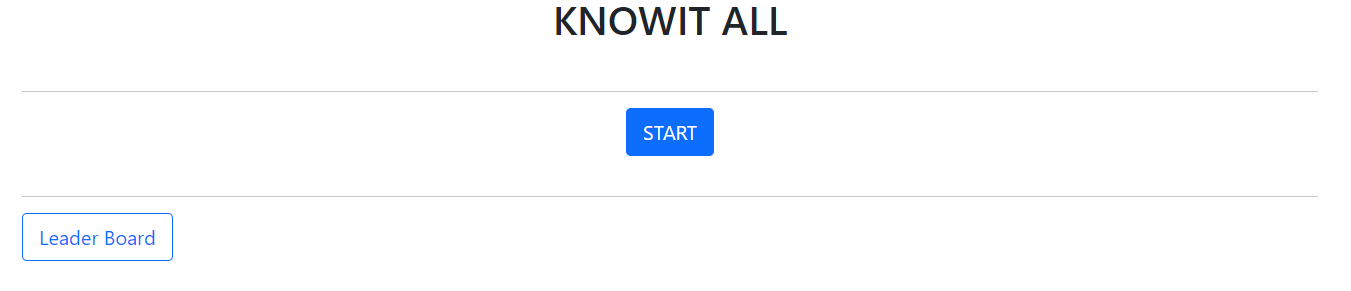
The code will randomly choose a number between the 0 and the length of the **array** and store the number in a temporary variable called **q\_id**. The algorithm will then delete the question\_id in the **array** at the position of the value **q\_id**. Deleting the question\_id makes the array one element 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** array. Data stored in the **showed\_id** array is mainly for testing purposes.

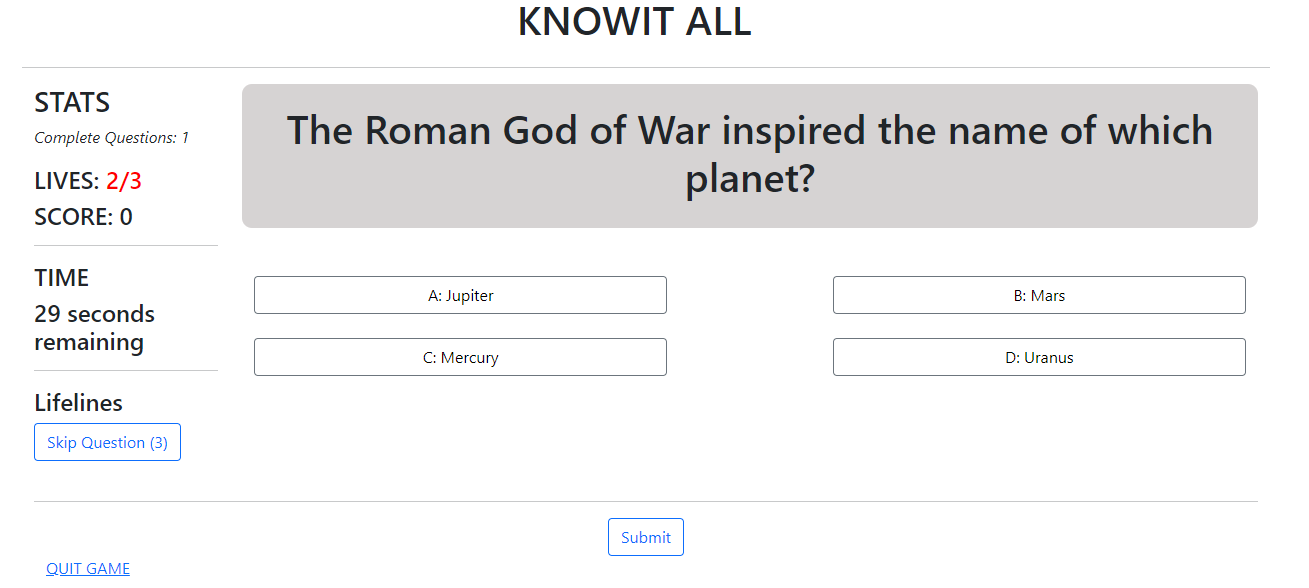
If the length of the **array** is equal to 0, that means all the questions have been shown on the game page once, the system will pop up an alert to notice the player finished all questions, and stop.

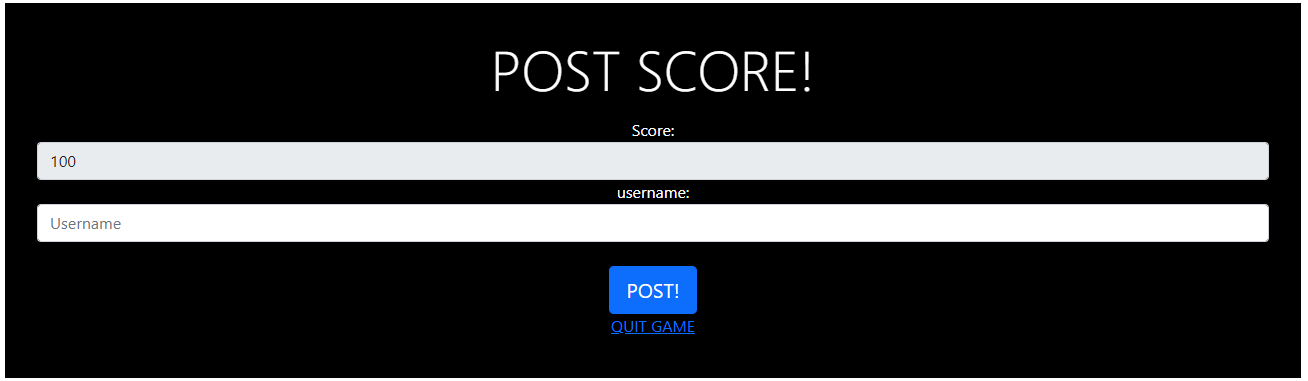
# UI Design

Main.html

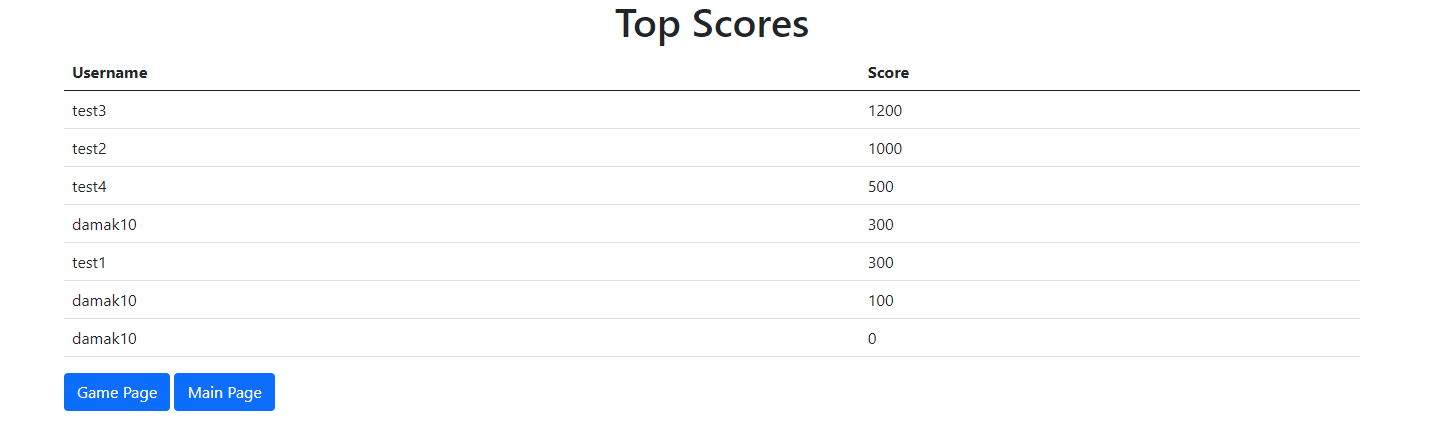


Game.html





Leaderboard.html



# Classes and Methods

[Next Iter]

# References

# Glossary