Technical Manual

Ivy Chepkwony, 2431951
Jacques Coetzee, 2302312
Mikyle Fourie, 2492832
Eden Neave, 2546984

I. Introduction

THE following document describes and details the overall system design from a developer's perspective to help other developers understand how all the systems work especially for the purposes of future development and improvement of this web application.

II. Overview of Design

This web application is structured similarly to the MVC (Model View Controller) architecture, but more specifically Django's **MVT** (Model View Template) architecture. The **Models** handle persistent data, such as the Users' profiles as well as the quizzes questions and answers. From there, the **Views** allow for some basic data manipulation functionality. Ultimately, they then send this data to, and load up the corresponding Template. The **Templates** are the html pages with all their relevant elements. For majority of the pages, this MVT structure suffices; however, for the Quiz screen itself, the html Template is connected to a **JavaScript** file, which is in turn connected via a WebSocket to a **Consumer** file. This connectivity structure [Figure 1] is handled mostly on its own by Django's backend.

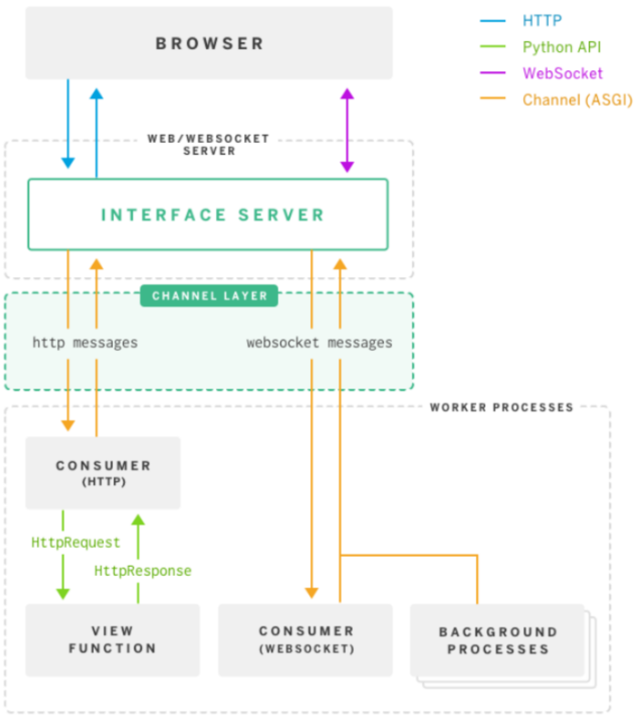


Fig. 1. Django Channels and WebSocket Connectivity

III. Model

A. Class structure

The models are the database tables that hold the actual data for the quizzes. There are six database tables for the quiz itself at the moment, and the users models are done automatically by django's all-auth functionality. The database tables in the models.py file of the Quiztest application are represented by classes. Within each class, there is a meta class that holds meta data for readability and organisation purposes, especially for the Administrator page. In Figure 9, the UML diagram for the key models is shown.

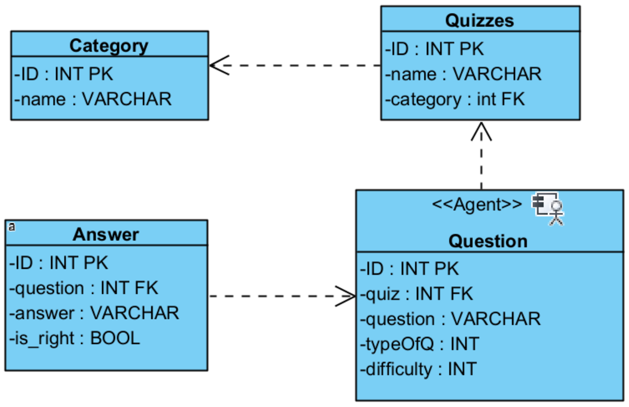
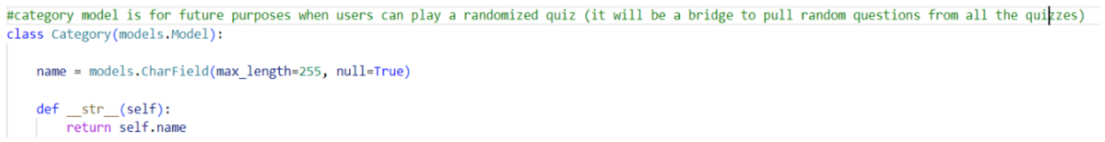


Fig. 2. Quiz Database UML Diagram

1) Category: The category class in Figure 3 defines the overall categories of quizzes. It has an id and name attribute. Django automatically creates an id attribute for every model, and so only the name attribute is defined. The name attribute is defined as a character field, which takes a maxim of 255 characters.

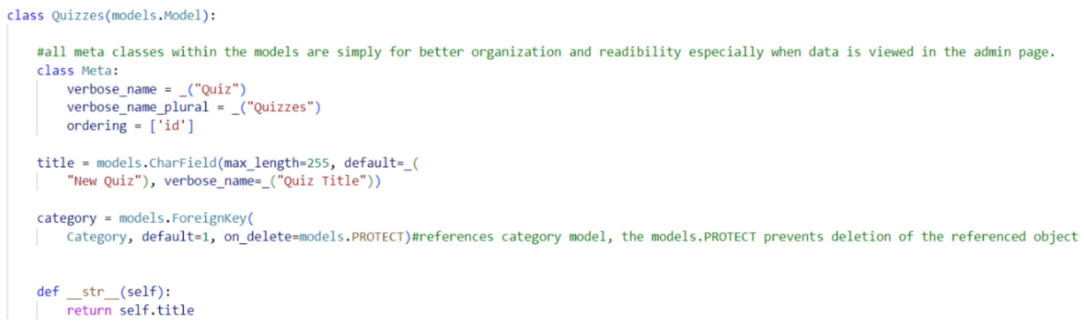


```
#category model is for future purposes when users can play a randomized quiz (it will be a bridge to pull random questions from all the quizzes)
class Category(models.Model):

    name = models.CharField(max_length=255, null=True)

    def __str__(self):
        return self.name
```

Fig. 3. Category Model Code

2) Quizzes: The Quizzes in Figure 4 class defines the quizzes itself. The reason for a category model and a quizzes model (even though they are seemingly the same thing), is for the purposes of including a Randomised/General Quiz which will pull from all the categories of quizzes. In the final implementation of the application, this will be implemented.

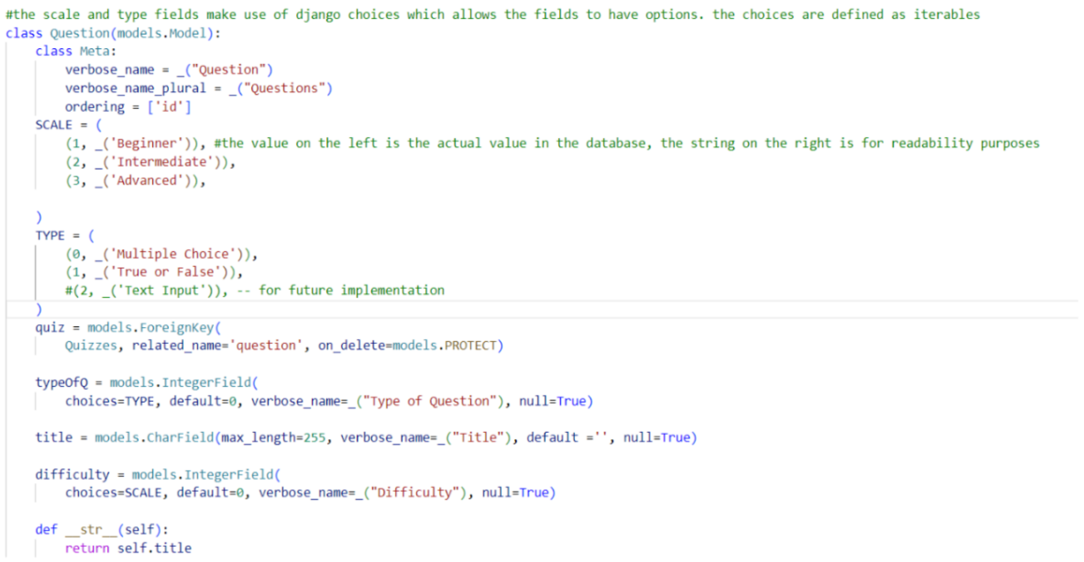
This model has the id, title (name), and category attribute. The category attribute is defined as a Foreign Key since it references the category model but the category model does not reference the quizzes model back. The foreign key's behaviour is classified as 'PROTECT' this means that if a category is deleted, the Quiz that references it won't be deleted. In the context of this QuizApp, for the purposes of preventing bulk amounts of data from being deleted (especially in the context of the questions referencing the quizzes - the questions won't be deleted if a quiz is deleted), the 'PROTECT' behaviour is ideal. .



```
class Quizzes(models.Model):

    #all meta classes within the models are simply for better organization and readibility especially when data is viewed in the admin page.
    class Meta:
        verbose_name = _("Quiz")
        verbose_name_plural = _("Quizzes")
        ordering = ['id']

    title = models.CharField(max_length=255, default=_(
        "New Quiz"), verbose_name=_("Quiz Title"))

    category = models.ForeignKey(
        Category, default=1, on_delete=models.PROTECT)#references category model, the models.PROTECT prevents deletion of the referenced object


    def __str__(self):
        return self.title
```

Fig. 4. Quizzes Model

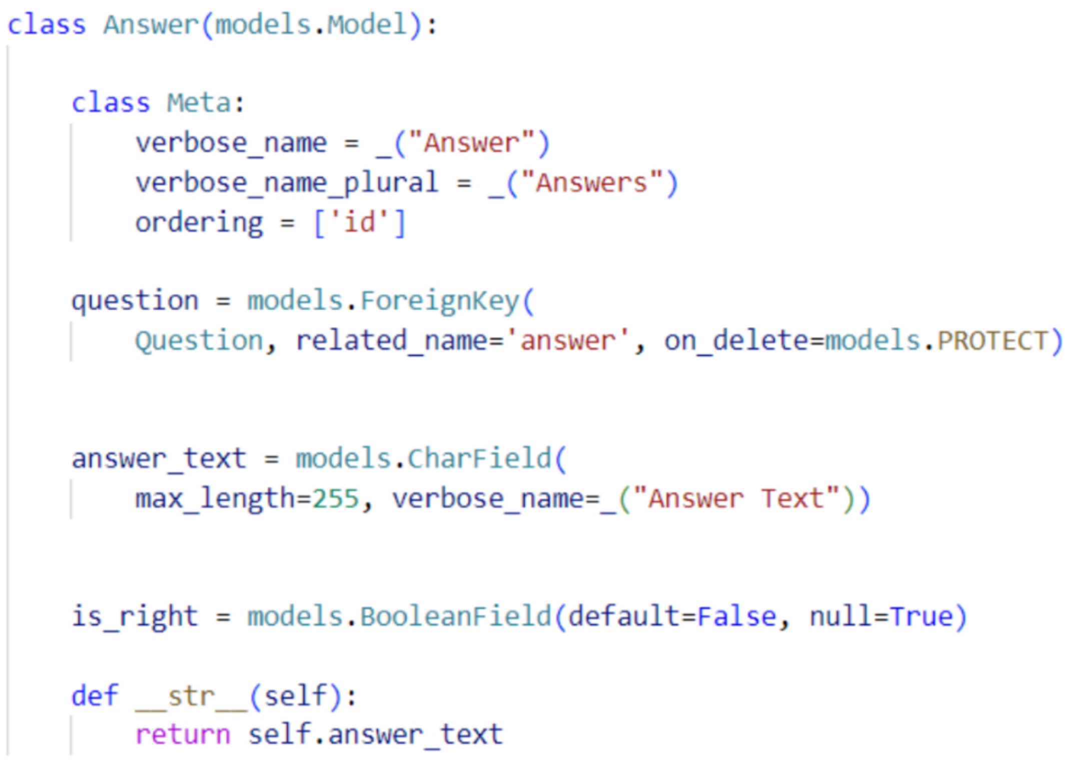
3) Questions: The question model in Figure X has a quiz, type of question, title (name), and difficulty attribute. The difficulty and type of question attribute are both integers, but represented as Django's choices to give the field predetermined options. The quiz attribute is defined as a foreign key to reference the quizzes model. The other attributes are either integer fields or character fields.



```
#the scale and type fields make use of django choices which allows the fields to have options. the choices are defined as iterables
class Question(models.Model):
    class Meta:
        verbose_name = _("Question")
        verbose_name_plural = _("Questions")
        ordering = ['id']
    SCALE = (
        (1, _('Beginner')), #the value on the left is the actual value in the database, the string on the right is for readability purposes
        (2, _('Intermediate')),
        (3, _('Advanced')),

    )
    TYPE = (
        (0, _('Multiple Choice')),
        (1, _('True or False')),
        #(2, _('Text Input')), -- for future implementation
    )
    quiz = models.ForeignKey(
        Quizzes, related_name='question', on_delete=models.PROTECT)

    typeOfQ = models.IntegerField(
        choices=TYPE, default=0, verbose_name=_("Type of Question"), null=True)

    title = models.CharField(max_length=255, verbose_name=_("Title"), default ='', null=True)

    difficulty = models.IntegerField(
        choices=SCALE, default=0, verbose_name=_("Difficulty"), null=True)

    def __str__(self):
        return self.title
```

Fig. 5. Question Model

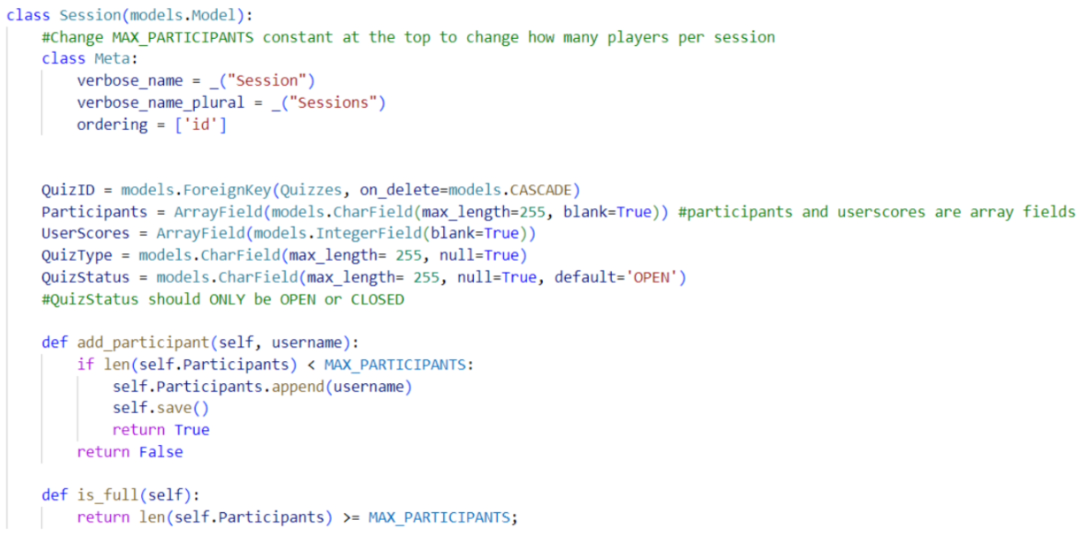
4) Answer: The answer model in Figure 6 has the attributes: question, answer, and `is_right`. The question attribute is defined as a foreign key to map each answer with a question.



```
class Answer(models.Model):

    class Meta:
        verbose_name = _("Answer")
        verbose_name_plural = _("Answers")
        ordering = ['id']

    question = models.ForeignKey(
        Question, related_name='answer', on_delete=models.PROTECT)


    answer_text = models.CharField(
        max_length=255, verbose_name=_("Answer Text"))


    is_right = models.BooleanField(default=False, null=True)

    def __str__(self):
        return self.answer_text
```

Fig. 6. Answer model

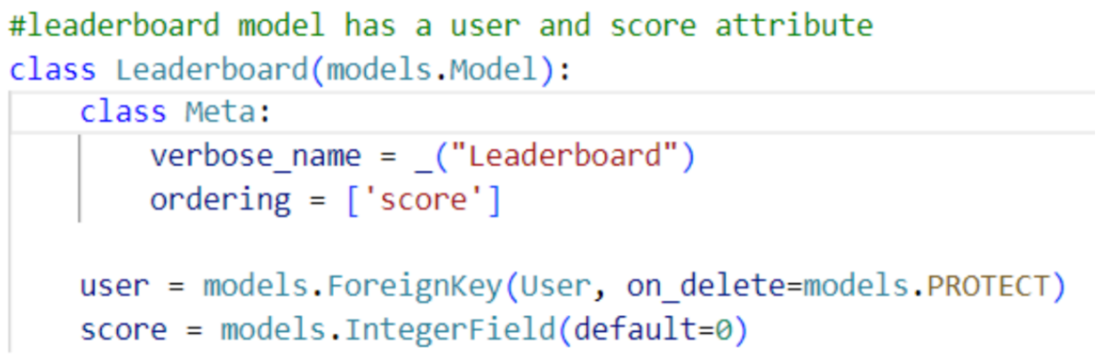
5) Session: The session model in Figure 7 has the following attributes: QuizID, QuizType, Participants, UserScores, QuizStatus. The Participants and UserScores are defined as Arrayfields so that the participants and corresponding scores for that session are accounted for in easy-to-digest representation. The QuizID is defined as a foreign key of the Quizzes class.



```
class Session(models.Model):
    #Change MAX_PARTICIPANTS constant at the top to change how many players per session
    class Meta:
        verbose_name = _("Session")
        verbose_name_plural = _("Sessions")
        ordering = ['id']


    QuizID = models.ForeignKey(Quizzes, on_delete=models.CASCADE)
    Participants = ArrayField(models.CharField(max_length=255, blank=True)) #participants and userscores are array fields
    UserScores = ArrayField(models.IntegerField(blank=True))
    QuizType = models.CharField(max_length= 255, null=True)
    QuizStatus = models.CharField(max_length= 255, null=True, default='OPEN')
    #QuizStatus should ONLY be OPEN or CLOSED

    def add_participant(self, username):
        if len(self.Participants) < MAX_PARTICIPANTS:
            self.Participants.append(username)
            self.save()
            return True
        return False

    def is_full(self):
        return len(self.Participants) >= MAX_PARTICIPANTS;
```

Fig. 7. Session Model

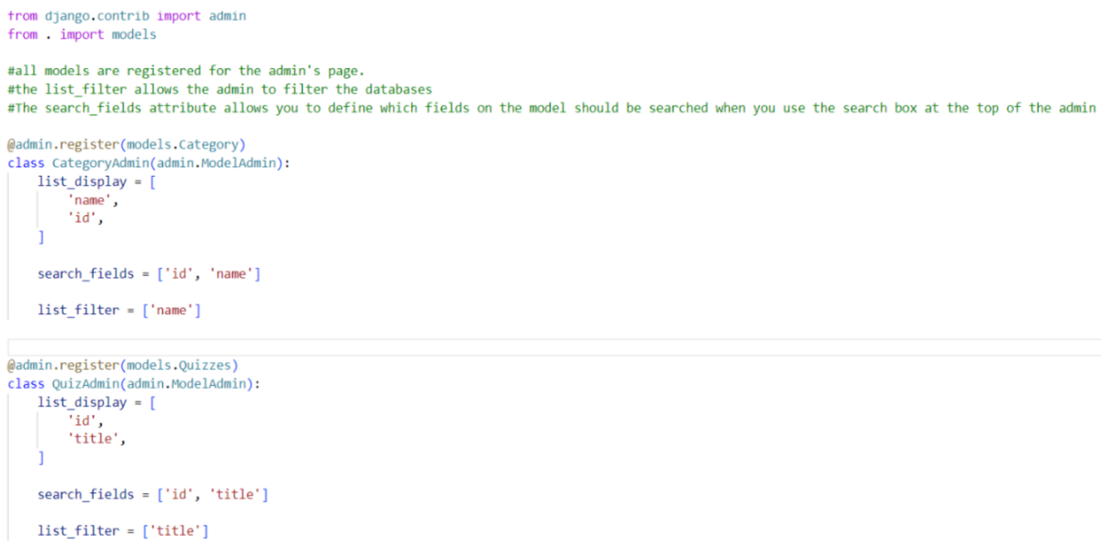
6) Leaderboard: The leaderboard model in Figure 8 has the following attributes: the User, and Score. The User is defined as a foreign key to the AllAuth User database.



```
#leaderboard model has a user and score attribute
class Leaderboard(models.Model):
    class Meta:
        verbose_name = _("Leaderboard")
        ordering = ['score']

    user = models.ForeignKey(User, on_delete=models.PROTECT)
    score = models.IntegerField(default=0)
```

Fig. 8. Leadeboard Model

7) Registering models: These models in Figure 9 are registered in the admin.py file specifically for Django's admin page, such that a staff member or superuser has access to all the data and can edit the data in the database without having to go into the database itself. The models are also called in the views.py and in the consumers.py to process the data and use the data in various functions.



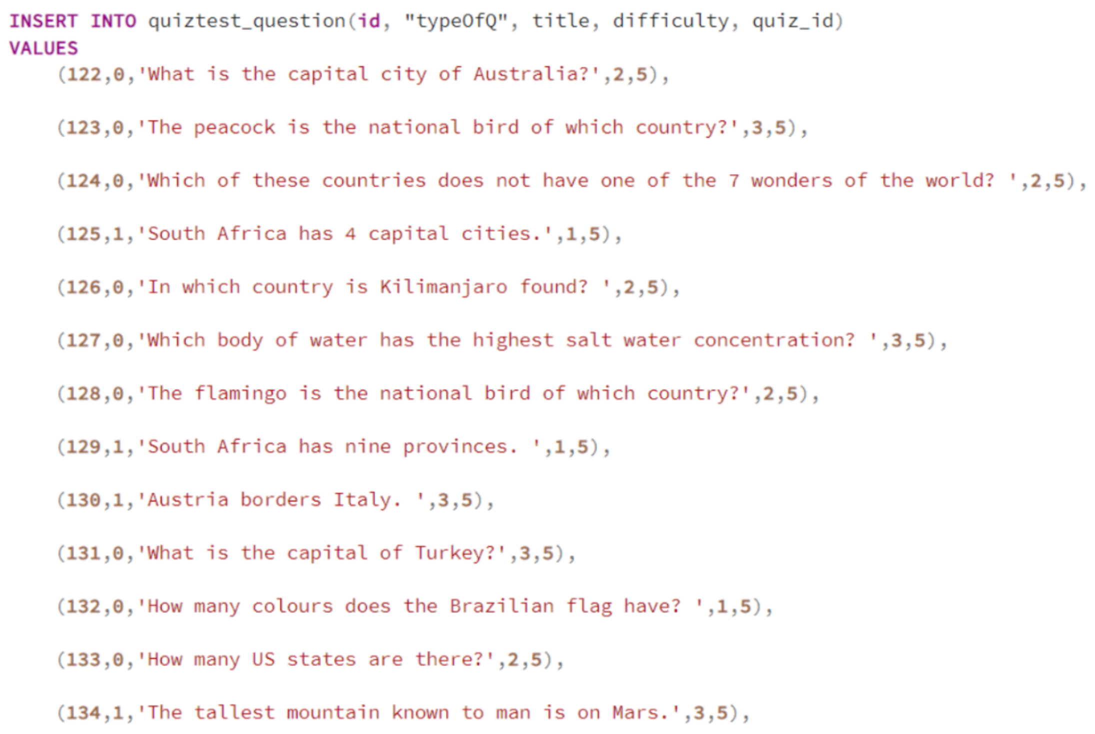
```python
from django.contrib import admin
from . import models

#all models are registered for the admin's page.
#the list_filter allows the admin to filter the databases
#The search_fields attribute allows you to define which fields on the model should be searched when you use the search box at the top of the admin

@admin.register(models.Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = [
        'name',
        'id',
    ]

    search_fields = ['id', 'name']

    list_filter = ['name']


@admin.register(models.Quizzes)
class QuizAdmin(admin.ModelAdmin):
    list_display = [
        'id',
        'title',
    ]

    search_fields = ['id', 'title']

    list_filter = ['title']
```

Fig. 9. Registering Models

8) Populating Databases using pgAdmin4 application: The populating of the models happened initially in the SQLite3 built-in database that came with Django, however this was only for testing purposes as SQLite3 has limited functionality in terms of how much data it can hold, and in terms of deploying the entire web application to a server to be available online. Thus the migration to heroku postgresql took place. The data was then put into the models using postgresql's interface, pgAdmin4.

The data, upon deployment of the website on Heroku, was pushed to Heroku Postgres - postgresql database hosted on Heroku.

Inserting of data through the pgAdmin4 Application was done using SQL INSERT statements as in in Figure 10.

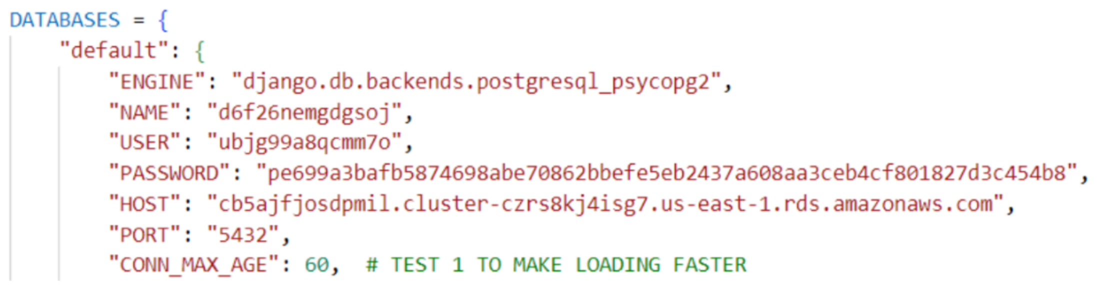


```
INSERT INTO quiztest_question(id, "typeOfQ", title, difficulty, quiz_id)
VALUES
    (122,0,'What is the capital city of Australia?',2,5),

    (123,0,'The peacock is the national bird of which country?',3,5),

    (124,0,'Which of these countries does not have one of the 7 wonders of the world? ',2,5),

    (125,1,'South Africa has 4 capital cities.',1,5),

    (126,0,'In which country is Kilimanjaro found? ',2,5),

    (127,0,'Which body of water has the highest salt water concentration? ',3,5),

    (128,0,'The flamingo is the national bird of which country?',2,5),

    (129,1,'South Africa has nine provinces. ',1,5),

    (130,1,'Austria borders Italy. ',3,5),

    (131,0,'What is the capital of Turkey?',3,5),

    (132,0,'How many colours does the Brazilian flag have? ',1,5),

    (133,0,'How many US states are there?',2,5),

    (134,1,'The tallest mountain known to man is on Mars.',3,5),
```

Fig. 10. Populating Databases using pgAdmin4

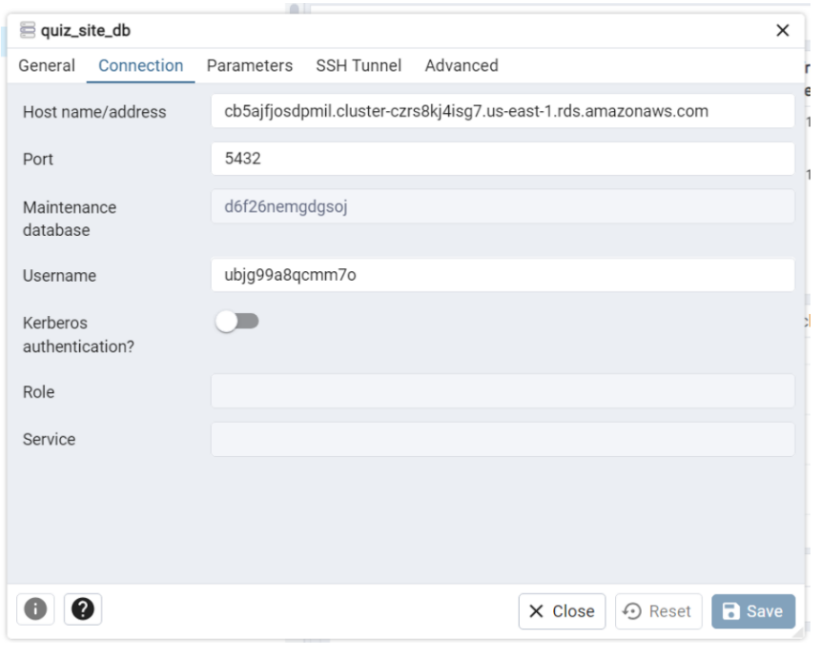
9) Important database settings: In the settings.py file shown in Figure 11, it is important to update the settings to match the credentials generated on the heroku App of the heroku postgres database. The port number is generally 5432. On the pgAdmin4 application, the same credentials will be carried over to allow for the pgAdmin4 app to connect to the heroku postgres database as shown in Figure 12. For further ease of access to the specific heroku postgres database, in the advanced settings, set a database restriction to the specific database being used as shown in Figure 13. If this is not done, all the Heroku databases that are made automatically will load (up to 100s), and so finding the specific database that is being used results in tedious scrolling. Setting the advanced restriction prevents this tedious work.

After all the important settings are set, as soon as changes are made to the database, the changes reflect automatically without having to push anything.



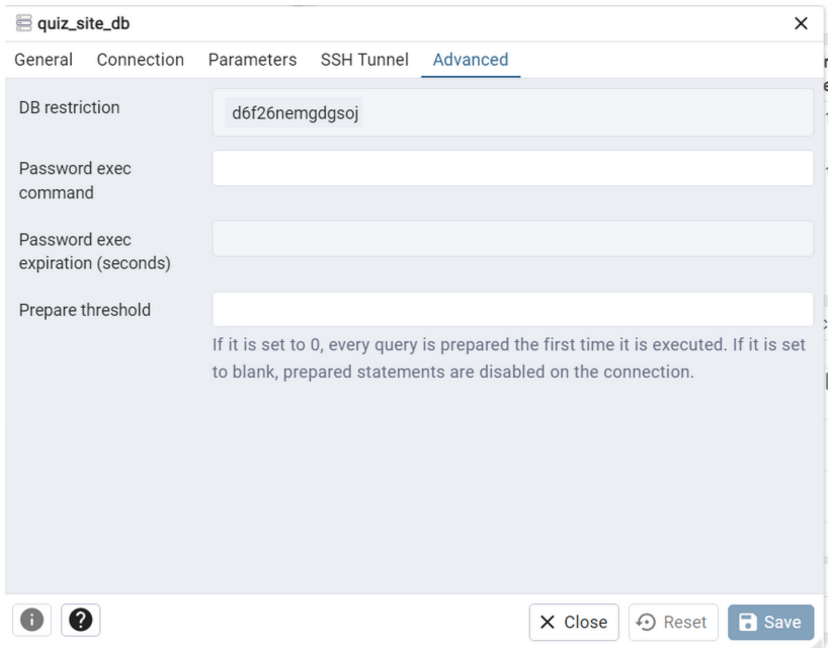
```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql_psycopg2",
        "NAME": "d6f26nemgdgsoj",
        "USER": "ubjg99a8qcmm7o",
        "PASSWORD": "pe699a3bafb5874698abe70862bbefe5eb2437a608aa3ceb4cf801827d3c454b8",
        "HOST": "cb5ajfjosdpmil.cluster-czrs8kj4isg7.us-east-1.rds.amazonaws.com",
        "PORT": "5432",
        "CONN_MAX_AGE": 60,  # TEST 1 TO MAKE LOADING FASTER
```

Fig. 11. Important database settings



Host name/address	cb5ajfjosdpmil.cluster-czrs8kj4isg7.us-east-1.rds.amazonaws.com
Port	5432
Maintenance database	d6f26nemgdgsoj
Username	ubjg99a8qcmm7o
Kerberos authentication?	
Role	
Service	

Fig. 12. DB Connection screenshot



DB restriction	d6f26nemgdgsoj
Password exec command	
Password exec expiration (seconds)	
Prepare threshold	

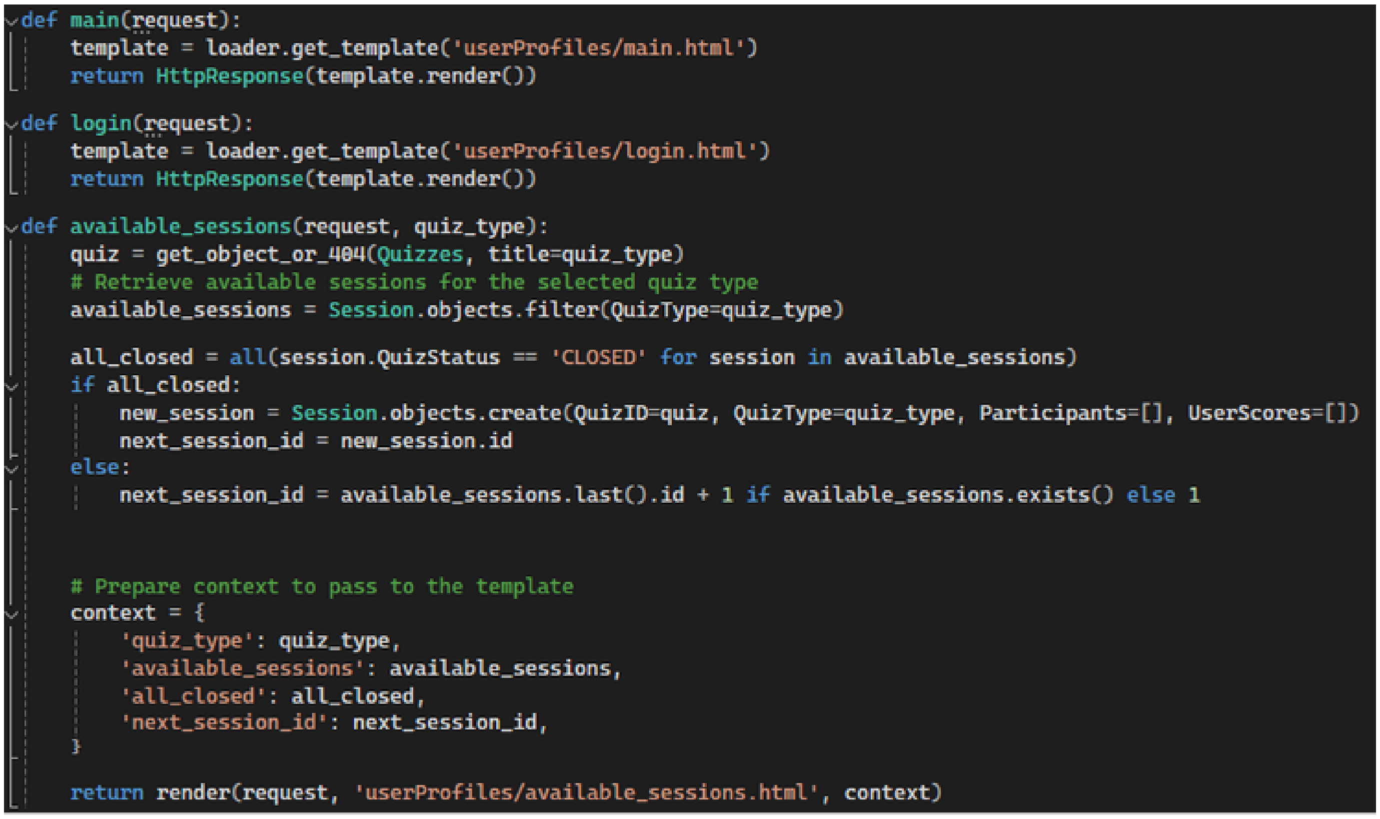
If it is set to 0, every query is prepared the first time it is executed. If it is set to blank, prepared statements are disabled on the connection.

Fig. 13. DB advanced settings screenshot

IV. Views & Templates

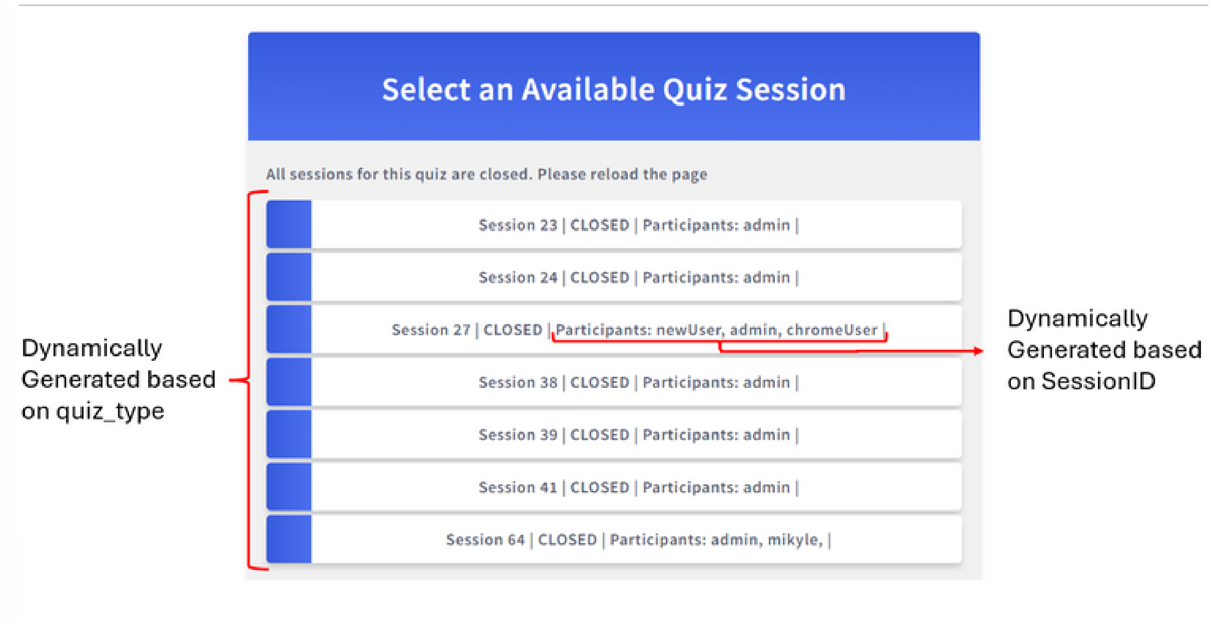
The views call up the respective html page and sends through data required as context. For instance, the login view loads the login template, and the available_sessions view loads the template and sends it data as context inf figures 14 and 15.

All the templates, save for the quiz screen, are simple html pages with elements chosen for styling. From the Quiz-Select screen, only the Art and Sports quizzes can currently be loaded.



```python
def main(request):
    template = loader.get_template('userProfiles/main.html')
    return HttpResponse(template.render())

def login(request):
    template = loader.get_template('userProfiles/login.html')
    return HttpResponse(template.render())

def available_sessions(request, quiz_type):
    quiz = get_object_or_404(Quizzes, title=quiz_type)
    # Retrieve available sessions for the selected quiz type
    available_sessions = Session.objects.filter(QuizType=quiz_type)

    all_closed = all(session.QuizStatus == 'CLOSED' for session in available_sessions)
    if all_closed:
        new_session = Session.objects.create(QuizID=quiz, QuizType=quiz_type, Participants=[], UserScores=[])
        next_session_id = new_session.id
    else:
        next_session_id = available_sessions.last().id + 1 if available_sessions.exists() else 1


    # Prepare context to pass to the template
    context = {
        'quiz_type': quiz_type,
        'available_sessions': available_sessions,
        'all_closed': all_closed,
        'next_session_id': next_session_id,
    }

    return render(request, 'userProfiles/available_sessions.html', context)
```

Fig. 14. Code that loads the template



Select an Available Quiz Session

All sessions for this quiz are closed. Please reload the page

Session 23 | CLOSED | Participants: admin |

Session 24 | CLOSED | Participants: admin |

Session 27 | CLOSED | Participants: newUser, admin, chromeUser |

Session 38 | CLOSED | Participants: admin |

Session 39 | CLOSED | Participants: admin |

Session 41 | CLOSED | Participants: admin |

Session 64 | CLOSED | Participants: admin, mikyle, |

Dynamically Generated based on quiz_type
Dynamically Generated based on SessionID

Fig. 15. Session selection screen

V. Quiz Consumers & JavaScript

When the **quiz screen** loads, it establishes a websocket connection through JavaScript. The functions connecting to this websocket are handled in the Consumers.py file. The Consumers handles the multiple-client connections, while also retrieving data from the databases. The Consumers have specific functions that run when users connect, disconnect and receive a message.. It also has other functions such as start_timer, update_score_in_db, and various broadcast functions which send data to each client. The Consumers runs from the first user connect. The associated JavaScript file catches the sent data and uses it to dynamically update the users' screens without having to reload the entire screen.

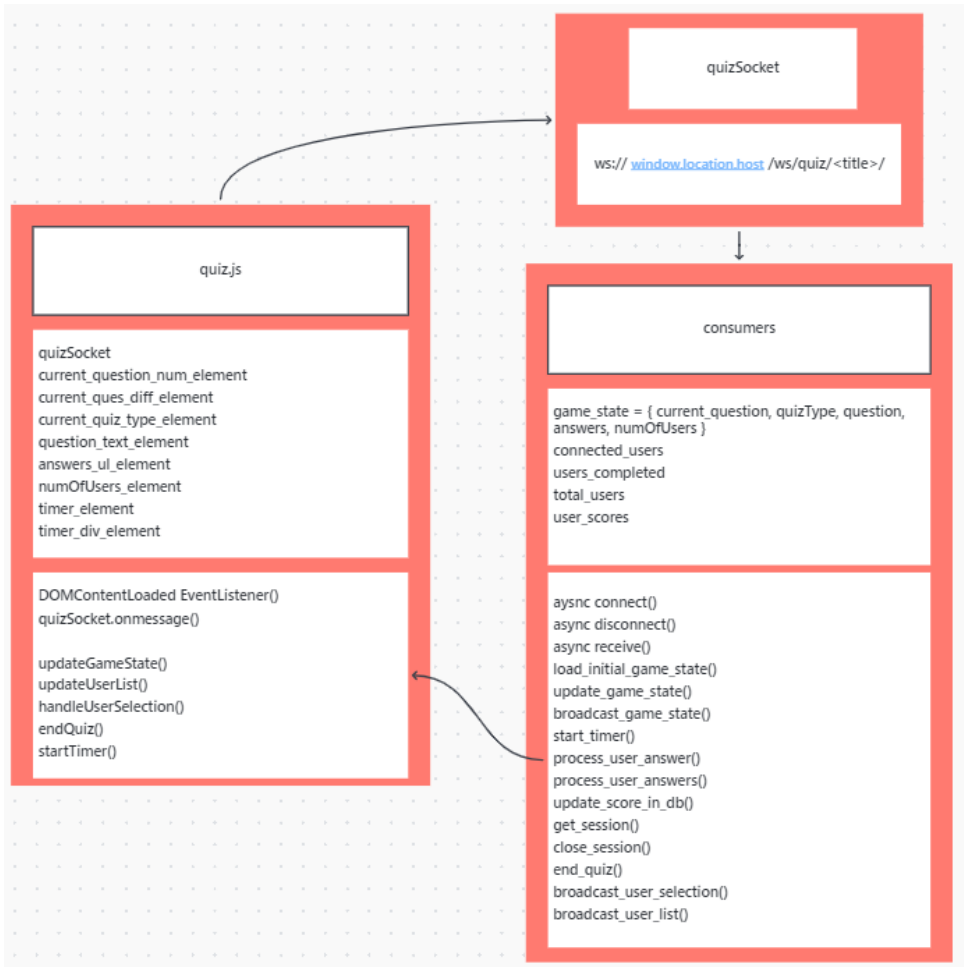


Fig. 16. Quiz Database UML Diagram

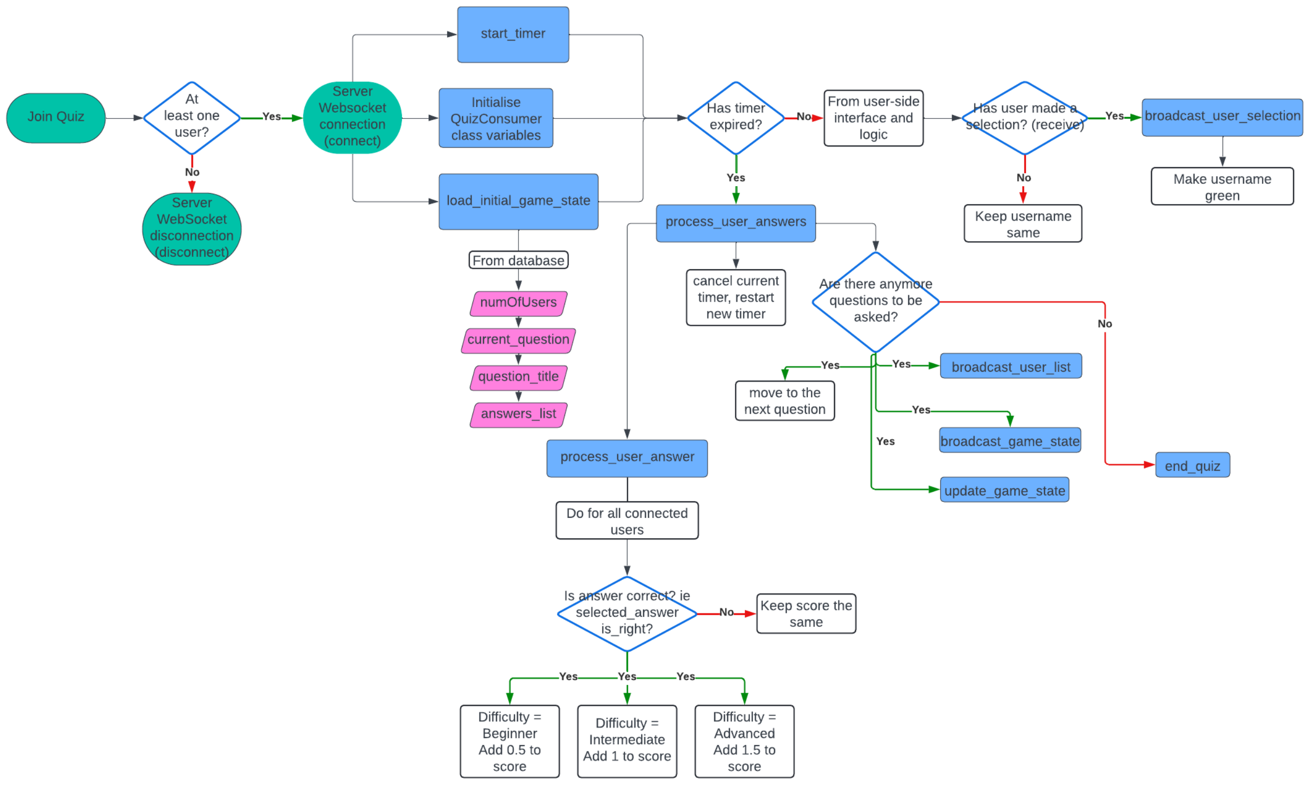
The connection between the consumers.py file and the quiz.js file also handles the validation of user answers and timer functionality. A UML diagram representing this logic can be seen in Figure 17, which represents the flow of the software functionality.

The client-side file 'quiz.js' written in JavaScript, manages user interactions and WebSocket communication with the server. Various HTML containers are retrieved using 'document.getElementById()' to dynamically update UI elements. Event listeners are attached to answer list items to handle user selection of answers, sending messages containing the selected answer's ID to the server via WebSocket. The client-side application also updates the UI based on received WebSocket messages, such as displaying the current question, available answers, and starting a timer. In the quiz.js file there is an updateGameState function that updates the game state. In this function the current question number, quiz type, question text, question difficulty and number of users is updated visually. The function populates the answers list and adds event listeners for each answer to allow users to interact and select their answer. Once an answer has been selected, a message including the selected answer ID is sent to the server as serialised JSON. The quiz.js file also includes a startTimer function that visually updates the users' screens. This function clears any existing timers, updates the timer element by decreasing the time left by one second and checks if the timer runs out. If the timer has run out a message is sent to the server notifying it that the timer has expired.

On the server side the 'consumers.py' file, written in python manages WebSocket connections and the quiz functionality. Upon connection, the server-side code loads the initial game state from the database, including quiz type, questions, and answers. It then broadcasts this initial state to all connected clients. As mentioned before, when users select answers, messages are sent from the client-side to the server-side via WebSockets. When the server receives these messages the game state is updated (function: `update_game_state`) and user answers are processed (function: `process_user_answers`). After each relevant event (e.g., user selection, timer expiration, quiz end), the server broadcasts the updated game state and user lists to all connected clients. When the quiz ends, the server sends final scores to all clients and updates user scores in the database. The 'start_timer' function begins a fifteen second timer and waits for all answers to be received or until the timer has run out. This triggers the process_user_answers function. This function checks whether the move_to_next_question bool is True and if so, checks if the current question is the last question. If it is not the last question it moves to the next question and if it is the last question it retrieves the end_quiz function. When moving to the next question the process_user_answer function is called that checks whether each user's individually selected answers were correct and increments each user's score based on the difficulty of the question and correctness. The function cancels the timer and starts a new one. When all the questions are complete, the endQuiz function is run, where it clears the timer and displays the connected users and their scores in the centre of the screen.

The leaderboard functionality was done in the consumers.py and the views.py. The function to update the leaderboard was done in the consumer.py and the function to display the leaderboard was done in the views.py. The update_score_in_db function creates a leaderboard entry and compares the user's current score in that session to the previously recorded score in the database. If the score of the session is higher than the score in the leaderboard (or if the leaderboard is empty), the score of the session is recorded in the database. In the end_quiz function, the update_score_in_db function is called to update the database at the end of the session with all the user's final scores.

In the views.py, the leaderboard function creates a list of all the high scores of the users from the updated leaderboard database and determines the rank and order of display of the scores and users. The rank is not a field in the database but simply a counter that is updated to place the user visually on the leaderboard.html page.



Join Quiz
At least one user?
Yes
No
Server WebSocket disconnection (disconnect)
Server Websocket connection (connect)
start_timer
Initialise QuizConsumer class variables
load_initial_game_state
From database
numOfUsers
current_question
question_title
answers_list
Has timer expired?
No
Yes
From user-side interface and logic
Has user made a selection? (receive)
Yes
No
broadcast_user_selection
Make username green
Keep username same
process_user_answers
cancel current timer, restart new timer
Are there anymore questions to be asked?
Yes
No
move to the next question
broadcast_user_list
broadcast_game_state
update_game_state
end_quiz
process_user_answer
Do for all connected users
Is answer correct? ie selected_answer is_right?
No
Keep score the same
Yes
Difficulty = Beginner Add 0.5 to score
Difficulty = Intermediate Add 1 to score
Difficulty = Advanced Add 1.5 to score

Fig. 17. Quiz Logic UML Flow Diagram

VI. User Authentication

A. Crucial Settings

The following settings must be present in the settings.py file in addition to the settings required for setup as set out in the AllAuth documentation:

- ACCOUNT_EMAIL_REQUIRED = True
  This enforces that a user needs to provide an email address when registering.
- ACCOUNT_AUTHENTICATION_METHOD = ”username_email”
  This allows the user to also enter a username which will be shown on the leaderboard.
- ACCOUNT_EMAIL_VERIFICATION = ”none”
  No email verification functionality is present. Change this to true to enable such functionality when implemented.
- ACCOUNT_CONFIRM_EMAIL_ON_GET = False
  This is only required when the above setting is set to “none”

B. O-Auth Apps

The two O-Auth applications namely Google and GitHub are set-up and fully functional. To add more applications:
Add the application to installed apps in settings.py: INSTALLED_APPS = [”allauth.socialaccount.providers.google”]
Follow the providers' specific instructions to create an O-Auth application. Copy the client ID and Secret to Settings.py following the following example:

```
SOCIALACCOUNT_PROVIDERS = {
”GitHub”: {
”APP”: {
”client_id”: ”codeFromProvider”,
”secret”: ”codeFromProvider”,
},
”VERIFIED_EMAIL”: True,
}
```

Finally, login to Admin on the site and add a social application filling out all the requested details.

VII. Deployment

Ensure your computer has the Heroku CLI installed and you have access to the dyno.

A. CMD Prompts

- To Enable the website dyno after it has been disabled, type: **heroku ps:scale web=1**.
- To disable the web-app dyno type: **heroku ps:scale web=0**. This is done to freeze the web-app when not in use.
- To deploy new changes to the web-app, make sure your local files are up to date with the main branch. Then type: **Git push heroku main**. This will rebuild and deploy the web-app.
- To view the live app, type **Heroku open**.

VIII. Troubleshooting

A. Software versions developed on:

To pick up and work with the applications back-end as a Developer, the correct versions of certain software and packages must be used. Newer versions may work as well, but use discretion.

- The project files should be stored within a virtual environment.
- The virtual environments DEBUG variable should be set to true (in a normal CMD terminal, this is done with set DEBUG=True. Please use appropriate command for your purposes.)
- All the software packages we used can be found with their respective versions in ”requirements.txt”

B. CMD Prompts

- To activate your virtual environment navigate to the environments Scripts folder, and simply type **activate.bat** in the CMD
- To ensure that all necessary software is installed to run locally, type **pip install -r** followed by the path to “requirements.txt”.
- To view the Heroku logs, type **Heroku logs –tail**.
- To run the local server, navigate into the *quiz_site* folder. Then in the CMD Prompt type **python manage.py runserver**
- To ensure the most recent static files (such as CSS or JS) have been uploaded to the deployed version of the site, use: **python manage.py collectstatic**

- To reset the PK sequence (especially after a large amount of data is added) use the following command in psql to avoid an IntegrityError - do this for every database table. This is an especially important command to prevent the Server 500 error. Type: **SELECT setval(pg_get_serial_sequence('app_dbtable', 'id'),coalesce(max(id), 1) + 1, false) FROM app_dbtable;**
- After every change to the Django models run both commands :
  - **python manage.py makemigrations**
  - **python manage.py migrate**