Project Documentation – Software 3 Group 5

Introduction

Our Trivia Game allows the player to challenge their knowledge across a wide range of topics, from geography and maths to programming and entertainment. The variety of topics means the questions will be varied no matter how many times it is played, without question repetition. We have also included a leaderboard element to raise the stakes and add some competition to the game, so the player can challenge themselves to improve on their top score or compete against their friends.

Aims and Objectives

The aim of our project was to create a multiple-choice trivia game using Python, that could be played via a user-friendly front-end interface, which would allow the player to use hints and keep track of top scores on a leaderboard.

The objectives were to produce a user-friendly game using a third-party API, demonstrating object-oriented programming, and using effective testing to scrutinize the code.

Background

Our game was inspired by the popular quiz show 'Who Wants to be A Millionaire'. It is a series of 15 questions, covering a wide range of topics. The questions are presented one by one, with the player being presented with 4 possible answers to choose from each time. The multiple-choice element is enhanced with the player being given the option to use four hints throughout the game, two 50/50 and two Ask the Audience hints, thereby increasing the likelihood of identifying the correct answer and scoring a point. After each question has been answered, the player is presented with the correct answer and their updated score. Once all 15 questions have been answered, the player's final score is shown, and they can choose to view the leaderboard. The leaderboard shows the highest 10 scores and the usernames of the players who scored them. The player can then choose to play again.

Specifications and design

Functional Requirements:

When developing our trivia game, we had to define the functional requirements to plan our code building and testing. We used "Who Wants to Be a Millionaire?" and other trivia games as references for our requirements. The well-known rules of these games made it easy to outline the most important requirements. We can roughly structure the requirements into groups:

Login Requirements:

The game is designed as a light, quick trivia game, making traditional authentication redundant and too complex for the player. Our approach was to allow the user to play a game with only their username, ensuring a quick and easy start. If the username is already registered, the user can play by providing the username used before or choose a new one. This approach has certain limitations, as usernames need to be unique and planned, but it allows users to play multiple times and view multiple game results on the leaderboard (if obtained). Login requirements include:

- *A player can start a game using their existing username (if the username is present in the database).*
- *A player can start a game using a new username of their choice (if the username is not present in the database).*
- *The username must not be an empty string and must not be longer than 40 characters.*

Gameplay Requirements:

Gameplay requirements are dictated by the traditional trivia game flow, ensuring that all aspects are implemented and that the code covers all possible scenarios and edge cases. The most important of them are:

- *Each game has 15 questions.*
- *A player can see the first question after logging in.*
- *Each question must be displayed with four options each time.*
- *Options should be displayed in random order.*
- *One of the four options must be correct.*
- *Only one option can be chosen, and the player can change their decision before the answer is submitted.*
- *A player must be able to see the correct answer after submitting their answer.*

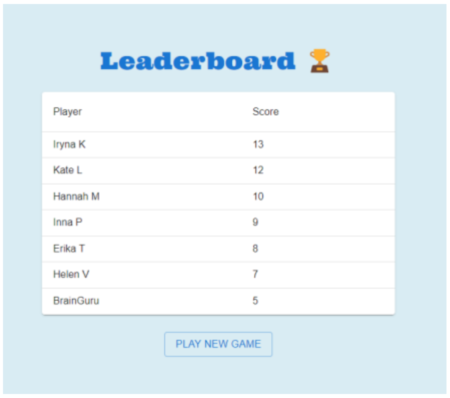
Scoring:

Scoring was a controversial part of our project with two options: to increase the numbers of points that can be scored each round, or to keep the points consistent. We opted not to increase because we wanted to let the user answer all fifteen questions without ending the game if they answer one question incorrectly. The most important score requirements are:

- *The initial score must equal 0.*
- *If the answer is incorrect, the score must remain the same.*
- *If the answer is correct, the score must increase by 1.*

Leaderboard:

- *The leaderboard must display the top 10 scores.*
- *Scores in the leaderboard must be displayed in decreasing order (Figure 1).*



Leaderboard 🏆

Player	Score
Iryna K	13
Kate L	12
Hannah M	10
Inna P	9
Erika T	8
Helen V	7
BrainGuru	5

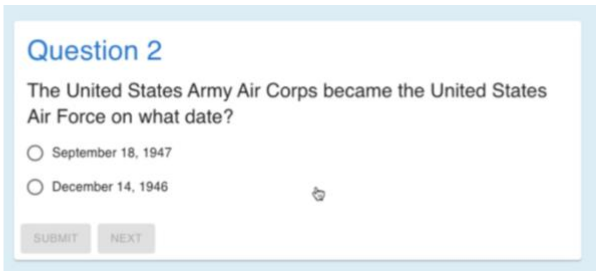
PLAY NEW GAME

Figure 1. Leaderboard.

Lifelines:

Lifeline requirements were introduced later than other requirements because they were not included in the minimal viable product concept of the game that we first aimed for. After having a stable minimal viable product, we decided to add the lifeline concept and designed its requirements:

- *Four lifeline opportunities must be offered.*
- *The number of lifelines is decreased by one after one is used.*
- *The 50/50 lifeline changes the number of multiple-choice answer options from four to two possible answers (Figure 2).*
- *One of the two remaining 50/50 options must be the correct option.*
- *The Ask the Audience lifeline generates the audience's votes for the correct answer (Figure 3)*
- *In most cases the highest percentage will be the correct answer, however in some cases the correct answer will be the second highest percentage, to reflect a real audience. The likelihood of the two answers with the lowest audience votes being correct is much lower, but not impossible.*



Question 2

The United States Army Air Corps became the United States Air Force on what date?

○ September 18, 1947

○ December 14, 1946

Figure 2. Example of 50/50 output, reducing answers from four to two.

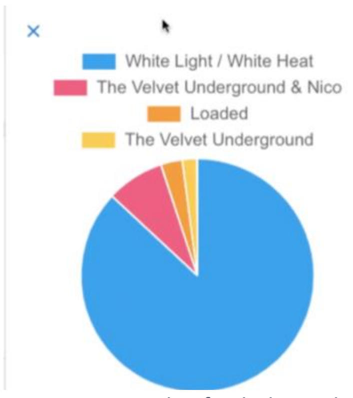


Figure 3. Example of Ask the Audience results.

Non-functional Requirements:

During the development process, our team focused mostly on the functional requirements. Among the non-functional requirements, our focus was on the usability aspect. The interface is designed to be obvious and clear to everyone who has encountered other versions of trivia games. The design is self-explanatory and doesn't require additional lengthy instructions. Future prospects for game refinement lie in the incorporation of performance, reliability, scalability, and accessibility requirements.

Design and architecture

The system functions around a backend which holds the game logic (Figure 4). A database is used to track and store key information - mainly usernames, the player's chosen answers, and scores. This is linked to the backend through a number of functions which allow the system to send and retrieve information from the database. Several endpoints in the backend have been created which are triggered by the frontend, and which retrieve questions and scores from the system, and send answers back. We used several classes to enhance code organisation and reusability (Figure 5), demonstrating object-oriented programming.

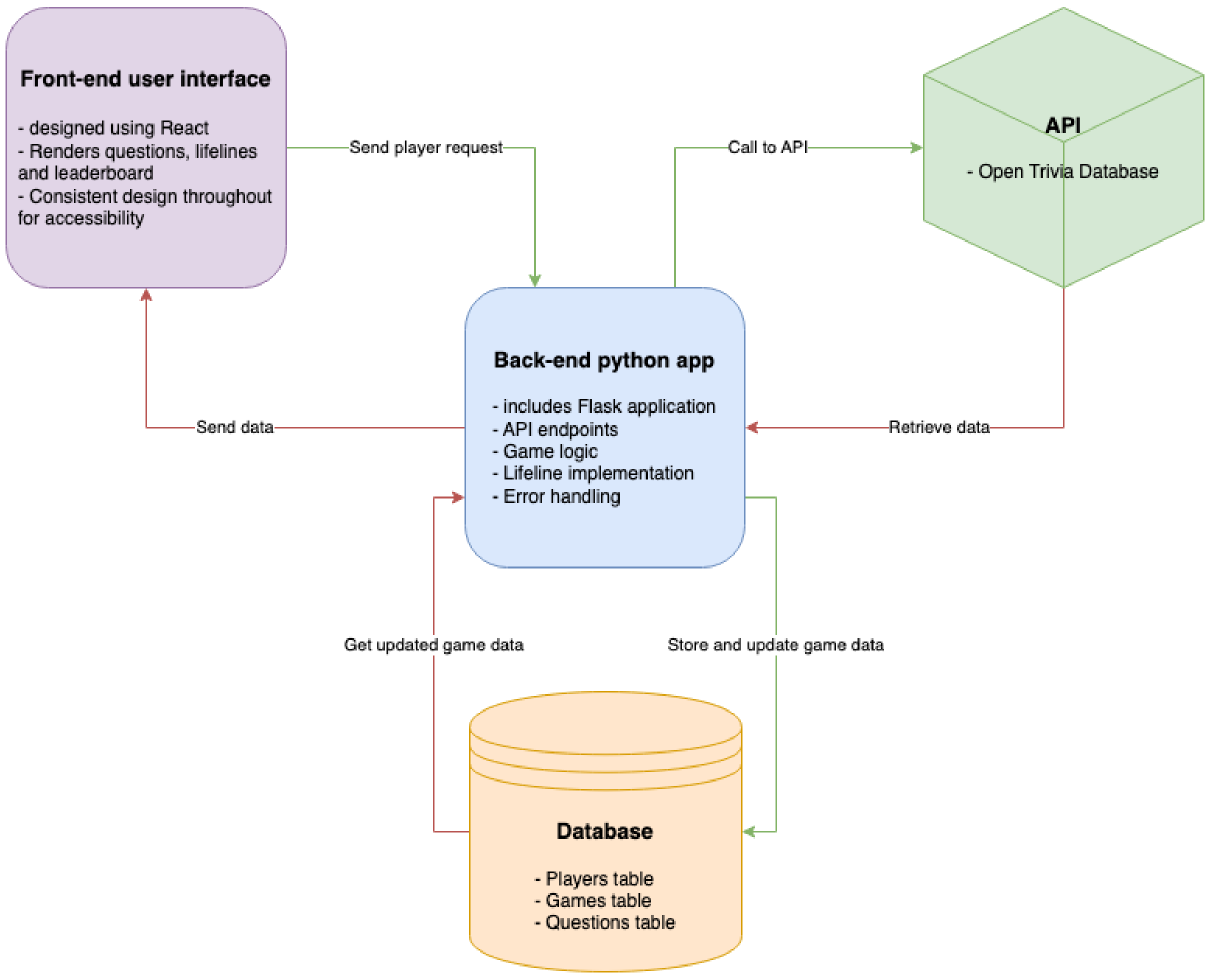


Figure 4. System architecture

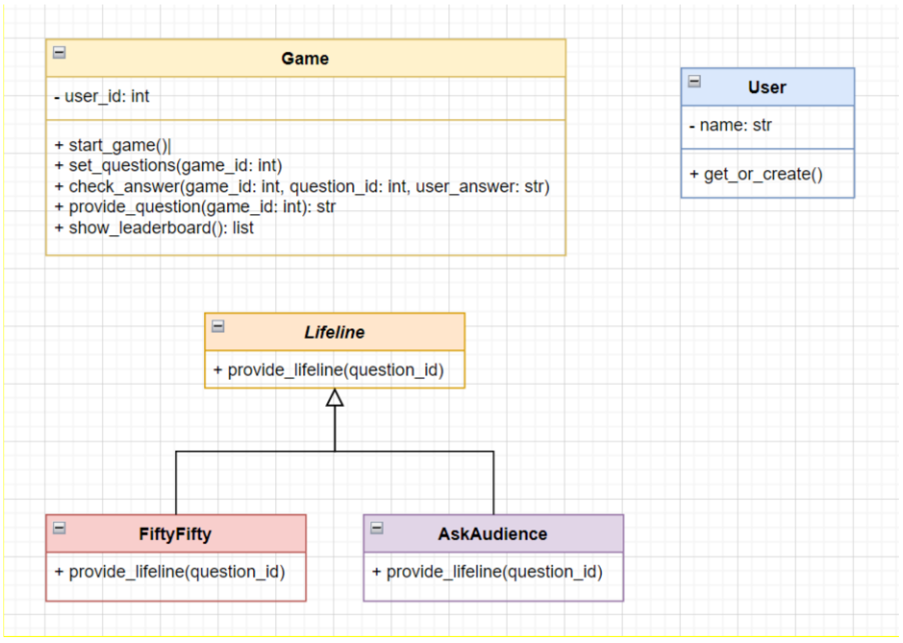

Game
- user_id: int
+ start_game()|
+ set_questions(game_id: int)
+ check_answer(game_id: int, question_id: int, user_answer: str)
+ provide_question(game_id: int): str
+ show_leaderboard(): list
User
- name: str
+ get_or_create()
Lifeline
+ provide_lifeline(question_id)
FiftyFifty
+ provide_lifeline(question_id)
AskAudience
+ provide_lifeline(question_id)

Figure 5. Classes used for improved organisation and reusability of our code.

Data Flow:

1. User interacts with the frontend, triggering API requests.
2. Flask app processes API requests, interacts with the game logic and the database.
3. Database stores and retrieves game-related data.
4. Game logic manages game state, questions, and scoring.
5. Responses are sent back to the frontend for display.

Scalability and Future Considerations:

Scalability:

- Database sharding or replication for handling a larger user base.
- Caching leaderboard data.

Enhancements:

- Adding more lifelines or features for a richer gaming experience.
- Implementing user authentication for personalized experiences.
- Introducing different difficulty levels for questions.
- Security measures such as input validation and secure database connections.

Implementation and execution

Development Approach

In our initial planning meeting, we agreed on the main areas for each team member:

- Front-end and Scrum master – Erika
- Database – Helen & Iryna
- Documentation – Hannah to write, the whole team to contribute ideas.
- Back-end – Erika, Hannah, Helen, Inna, Iryna, Kate
- Unit tests – Erika, Hannah, Helen, Inna, Iryna, Kate

To manage tasks and maintain progress we did the following:

- Group meetings every Friday and Sunday (additional meetings as needed)
- ‘Daily stand-up’ style meetings led by Erika to discuss progress, next steps, and obstacles.

- Group discussions to problem solve and brainstorm ideas.
- Tasks assigned and Trello board updated.
- Regular Slack communication.
- GitHub – branching and pull requests with code reviews.

Implementation Process

As a team, we took a dynamic approach to the project. Through regular meetings and frequent communication, we were able to work on most elements of the system at the same time, enabling adjustments to be made along the way.

We began by identifying the flow of the game and outlining the logic. There were challenges in the beginning trying to identify which information we needed to store in the database to allow the game to function, without storing unnecessary data. We decided not to ask for a password as we deemed this to be unnecessary data capture that wouldn't improve the quality of the game for the player. Once the database was created, we were able to start writing the functions to communicate between the database and the backend. We focussed on beginning with a Minimum Viable Product (MVP) and enhanced the functionality later. Erika created the front-end alongside the backend creation by the rest of the team, which enabled us to rewrite functions and endpoints to meet the needs of the frontend system. Once all of the functions were written for the MVP, we added extra functionality in the form of a leaderboard and hints to remove 2 wrong answers, or to give suggestions from the 'audience'.

Implementation Challenges

Scoping and Design:

Challenge: Agreeing on the theme, the scope and design of the game. Different team members had varying ideas and expectations.
Solution: Collaborative brainstorming sessions were held to gather ideas, clearly define the scope and features of the game, and create a shared vision in a design document.

Database Schema Design:

Challenge: Designing an efficient and scalable SQL database schema for storing both API data and game-related data.
Solution: Database schema was planned carefully and was normalized. We used foreign keys, constraints appropriately. Also, we considered the relationships between different entities such as users, questions, and game sessions.

Integration with Open Trivia Database API:

Challenge: Interacting with the Open Trivia Database API and managing the responses can be complex.
Solution: Design a module to handle API requests and responses. Implement error handling for network issues or unexpected API changes. Consider caching strategies to minimize the number of API requests.

Error Handling and Logging:

Challenge: Properly handling errors and logging issues for debugging and monitoring. Ensuring the game with multiple developers contributing code is bug-free.
Solution: Testing strategy was implemented, including unit testing and manual testing. Log errors with meaningful messages, stack traces, and relevant context.

Collaborative Development:

Challenge: Coordinating code contributions from multiple team members that might lead to conflicts and difficulties in merging changes.
Solution: We used version control systems Git, frequent communication to avoid conflicts. Consider dividing the project into modules or features to minimize code overlap.

Communication:

Challenge: Communication breakdowns can lead to misunderstandings and delays.
Solution: We established regular team meetings, used Slack for communication. Roles and responsibilities for each team member were clearly defined.

Project Management:

Challenge: Tracking progress, managing tasks, and staying on schedule.
Solution: Project management tools, Trello was used to track tasks. We used Agile development approach, breaking the project into manageable sprints with clear goals.

Testing and evaluation

Testing Strategy:

Our testing strategy predominantly involved a combination of unit tests and manual testing. The unit tests were comprehensive in that they tested all of the individual components in our code, specifically our database utility functions, and endpoint routes.

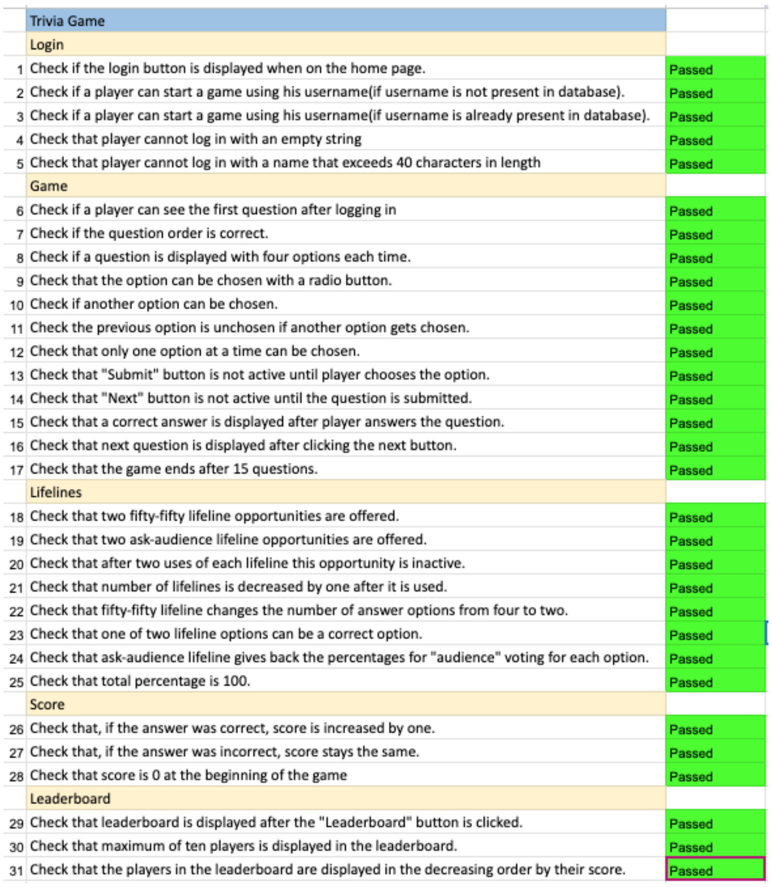
Unit Testing:

We extensively utilized mock data and connections for database utility functions, covering success, error, and edge cases. Mocking isolated these code units from the codebase and actual database, ensuring focused testing and preventing unintended data changes. This approach enhanced test reliability, execution speed, and effectiveness by providing control over the testing environment. For game endpoint routes, a similar approach was employed, involving mocking interactions between classes and methods. This simulated various game stages (e.g., player/game creation, question retrieval) and tested scenarios like empty username inputs and internal server errors during question data retrieval in the wrong content type.

Manual Testing:

While our primary emphasis was on extensive unit testing, we complemented it with manual testing to ensure comprehensive quality assurance for the game. Recognizing that unit tests focus on specific scenarios and may miss unforeseen edge cases, manual testing allowed us to ad-hocly explore the project for unexpected issues. Our manual test checklist, aligned with project requirements, covered all game scenarios (Figure 6).

Testing proved invaluable, revealing previously unnoticed bugs in our code/game and guiding agile refactoring. The result was the correct and robust behaviour of our functions and the game, aligning with our expectations and intentions.



	Trivia Game	
	Login	
1	Check if the login button is displayed when on the home page.	Passed
2	Check if a player can start a game using his username(if username is not present in database).	Passed
3	Check if a player can start a game using his username(if username is already present in database).	Passed
4	Check that player cannot log in with an empty string	Passed
5	Check that player cannot log in with a name that exceeds 40 characters in length	Passed
	Game	
6	Check if a player can see the first question after logging in	Passed
7	Check if the question order is correct.	Passed
8	Check if a question is displayed with four options each time.	Passed
9	Check that the option can be chosen with a radio button.	Passed
10	Check if another option can be chosen.	Passed
11	Check the previous option is unchosen if another option gets chosen.	Passed
12	Check that only one option at a time can be chosen.	Passed
13	Check that "Submit" button is not active until player chooses the option.	Passed
14	Check that "Next" button is not active until the question is submitted.	Passed
15	Check that a correct answer is displayed after player answers the question.	Passed
16	Check that next question is displayed after clicking the next button.	Passed
17	Check that the game ends after 15 questions.	Passed
	Lifelines	
18	Check that two fifty-fifty lifeline opportunities are offered.	Passed
19	Check that two ask-audience lifeline opportunities are offered.	Passed
20	Check that after two uses of each lifeline this opportunity is inactive.	Passed
21	Check that number of lifelines is decreased by one after it is used.	Passed
22	Check that fifty-fifty lifeline changes the number of answer options from four to two.	Passed
23	Check that one of two lifeline options can be a correct option.	Passed
24	Check that ask-audience lifeline gives back the percentages for "audience" voting for each option.	Passed
25	Check that total percentage is 100.	Passed
	Score	
26	Check that, if the answer was correct, score is increased by one.	Passed
27	Check that, if the answer was incorrect, score stays the same.	Passed
28	Check that score is 0 at the beginning of the game	Passed
	Leaderboard	
29	Check that leaderboard is displayed after the "Leaderboard" button is clicked.	Passed
30	Check that maximum of ten players is displayed in the leaderboard.	Passed
31	Check that the players in the leaderboard are displayed in the decreasing order by their score.	Passed

Figure 6. Manual testing checklist.

System limitations

- Initially, we planned to let users choose quiz difficulty levels. However, due to time constraints in developing the MVP, we opted to prioritize features like quiz lifelines, the leaderboard, and building the React Frontend. Future enhancements may include reintroducing the difficulty level feature and adjusting the points structure accordingly, with higher difficulty levels earning more points.
- Storing question data for completed games is memory-intensive. It may be beneficial to delete data for the 15 questions once a game concludes and the final score is calculated. Retaining old question data is unnecessary since each game is unique and fetches a distinct set of questions from the online API for every request.
- Due to time constraints, we prioritized a simple, user-friendly frontend interface. Future improvements should aim for a visually exciting, playful, and interactive frontend to enhance the game experience and make it more fun for the end user.
- Accessibility features for users with disabilities, such as larger text and voice controls, are important considerations for future enhancements, which would allow our game to be accessible to as many players as possible.
- Another nice-to-have is introducing a time limit for answering questions or for the overall game. This would increase difficulty and competitiveness, and the leaderboard would take into account both the highest scores and fastest playing times.
- Device compatibility was tested on a computer browser, but future efforts should ensure compatibility on all devices, such as mobile phones and tablets, to ensure a well-designed and flexible frontend interface.

Conclusion

To conclude, our aim was to develop a user-friendly, multiple choice trivia game which allowed the player to use hints and track top scores on a leaderboard. We have succeeded in achieving this by developing a programme that is centred around a python application, which interacts with a database, API, and a front-end React application. The game logic is intuitive, and the design is consistent throughout. We have implemented two hints - a 50/50, and an Ask the Audience option - and the top 10 scores are displayed on a leaderboard. Despite some limitations, mostly time related, we managed to build more than a minimal viable product and enhanced it with extra gameplay functionality. Throughout the project we worked to each other’s strengths, and through regular communication and support for one another, we were able to develop an application that we are very proud of.