Project Report

CFGTeam123: Daisy Sherwood, Daniella Tobit, Shekinah Baba-Lola, Chiara Pascucci, Ayesha Khan

INTRODUCTION:

Our project is a productivity application that allows users to time their break from work and play online games. Users will be able to login and enter the amount of time they want to play a game for. We built 4 games to play, which are: Tic Tac Toe, Blackjack, a trivia quiz, and a number guessing game.

We were inspired to build this project as throughout lockdown, many people have adjusted to working from home, which has introduced new challenges in staying productive. When you want an intermission from your work, a 5-minute break can quickly become a 30 minute break if unmonitored. We wanted to improve productivity by offering structured non-working time through engaging games. Those who struggle with time keeping can now take structured breaks and stay on track. The aim of our productivity app is to offer a choice of 4 different games which demonstrate slightly different skills that we have learnt during the course; a trivia game which connects to an external API to provide the data for the questions, as well as 3 simple games (tic-tac-toe, blackjack, and number guessing) which focus on clean Object Oriented Python code and efficient algorithms. The connection to the database and logging of user sessions allows us to integrate our knowledge of SQL and databases into the project.

BACKGROUND

Our productivity app consists of multiple component parts, as we wanted to offer a choice of games. Therefore, it was crucial that our design and implementation were planned. Below is a summary of the rules and logic for each feature.

Set Timer

Rules: The user is able to select a time length from a pre-populated list of options and a timer is set to that length. The user is then redirected to the browse game page and the timer starts. The user is free to use the site until the timer expires and the user is logged out.

Logic: The timer is implemented as a set of JavaScript functions in a dedicated file. It utilises the session object to ensure that the timer persists between requests. It generates a JavaScript alert to notify the user when the time has run out.

Browse Games

Rules: Be able to view the links to all the games and game record history on one centralised page, but only after logging in and beginning a timed session.

Logic: After the timer has been set you are directed to the browse games page. Once the timer has ended you are logged out of the site, and are no longer able to view the browse games page.

Tic Tac Toe

Rules: The rules are based on the popular 'Tic Tac Toe' game, otherwise known as x and o. The player is assigned "x " and the computer "o" and a win is recorded for the first person to have their letters align either horizontally, vertically or diagonally. If there are no wins there is a tie

Logic: There is a class that keeps track of what is going on in the game and makes the computer move. The computer makes it's moves based on a set of rules as follows: it gathers all the available spaces left on the board and store that in a "possible moves" list; it checks if it has a chance of winning for the current move and makes the move; it check if the opponent has an opportunity to win in the next move

and if so, blocks the move; if step 2 and 3 aren't possible it plays in the middle; if the middle is taken, it plays at a random available corner; otherwise it plays at a random available side.

Black Jack

Rules: We followed standard blackjack rules. The goal being to beat the dealer's hand without going over 21. Face cards are worth 10, and Aces are worth 11 or 1, whichever makes the better hand. Each player starts with two cards, but one of the dealer's cards is hidden until the end. To 'Hit' is to ask the dealer for another card, to 'Stand' is to hold your total and end your turn. If you go over 21 you go bust, and the dealer wins regardless of the dealer's hand. If you are dealt 21 with the first 2 cards, it is Blackjack and a win. No betting was included in this version of blackjack, this would be a good area for development as blackjack is traditionally used for betting.

Logic: The logic of the game is based on the creation of Deck, Card and Blackjack objects. The Deck object creates a full hand of cards to play with, the Card object improves the user readability, and the Blackjack object provides the methods which create dealer and player hands as well as determining the winner and loser of each game. The flow of the blackjack game is determined in functions which are declared outside of the object methods. The most complex areas of game logic are linked to calculating the value of a hand, and specifically the Ace card logic. The specific logic for the dealer's hand is: the dealer will continue to draw themselves another card if their hand is less than 17, upon which the dealer hand is complete.

Number Guessing Game

Rules: This is a really simple number guessing game. Where the computer chooses a number set to be between 1 and 200 and the user has 10 tries to try and guess what the number is. After each guess the computer will tell the user if their guess is right (win!), or too high, or too low.

Logic: This game is implemented as a single Python function, which is able to take in the computer number, the guess submitted by the user, the number of guesses made so far and return an appropriate response. When the game first begins, the random module is used to generate the computer's number.

Trivia

Rules: The user is presented with 10 random questions (with varying difficulty and category). For each question, they must select either a multiple choice answer, or True or False. After making their selection, the game will either print 'Correct! :)' or 'Incorrect :('. After the user has answered the final question, they will then be shown their total score, which is their correct answers out of 10.

Logic: The Trivia game connects to an external API to provide the questions and answers for the game. The game is created using a *TriviaGame* class, which will create an instance of the class for each user. This means that each user has a unique instance of a trivia game so that API calls can be differentiated. In order to iterate through the questions during each game, the *TriviaGame* class uses an iterator. Every time next() is called, it will retrieve the next question in the list.

Registration/ Login

Rules: We wanted to create registration/ login functionality so that users could see a personalized view of their break and game play history. We have coded a basic implementation of a game play history, and made the timing function and games only available if you have registered and logged in. We wanted to create our project as though we were going to deploy and make our service available to the public. In a real world scenario we would want to track who was using our app and be able to reach them for

advertising purposes and to continuously develop our program so that it remains relevant, easy to use, and desirable to our audience.

Logic: Registration and Login Forms → These forms were created as classes using the Flask-WTF package (see forms.py). This allowed us to easily apply validators to our fields (for example email format and string length) and the form would only be submitted if these conditions were met. In our applicable routes we instantiated a 'form' object from these classes which allowed us to use the data provided to then validate users and create user sessions. If all the validation checks are passed when a new user fills out the registration form the user is instantiated as an object of our CustomAuthUser class, and then we use the method save() from that class to pass that information to our database (as well as automatically generating and saving a hashed password using the werkzeug.security package, and lastly creating a user_id object which we will need at login.)

Login → A user is logged in through our login form via the login route. If the information entered into the field passes their validation checks, the user instance is retrieved with the user name, then the password of that instance is compared with the password provided in the form (through the use of a specialised function.) If this function returns true then the user is logged in. This function works in conjunction with the load_user function which is an essential part of the Flask-Login, login-manager module which we are using to handle our users' sessions. The password is passed to our verify_password() method from our CustomAuthUser class (this method contains another method which is built into the werkzeug.security package and checks that the password hash matches the password provided by the user.) If the password verification method returns True, our user is logged in with the login_user() function (again, built into the Flask-Login package), and a session is created. We have used the user_name to create our session. The session is an object which will store our user's information on the web server for the duration of their login. Sessions are temporary (they end when the user logs out), and the session information is accessible from various pages.

The login decorator (@login_required) is another aspect of the Flask-Login package which allows us to easily decorate any routes which we wish to be accessible only if the user is logged in.

SPECIFICATIONS AND DESIGN

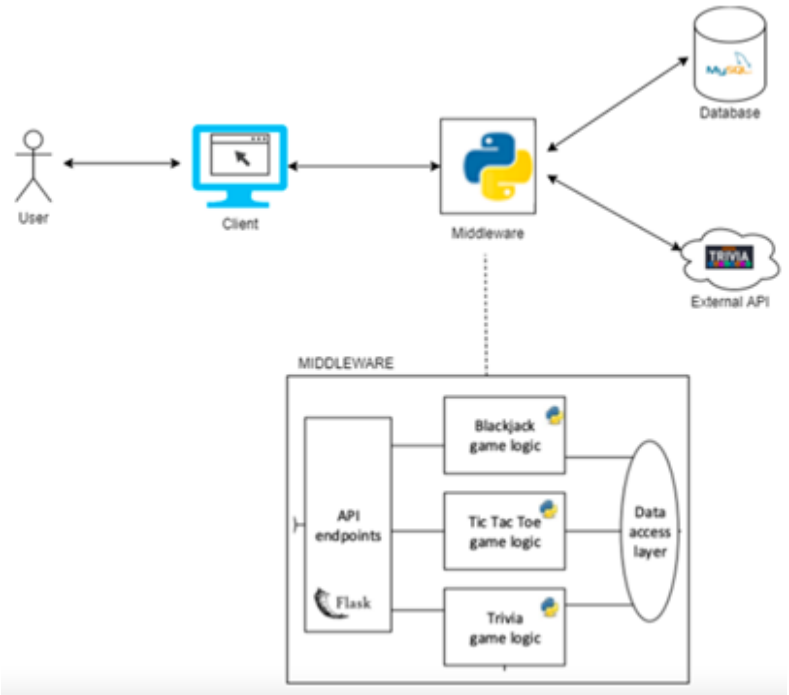

Figure 1. Diagram of productivity app basic process flow.

Figure 1 shows design of the productivity app. Users will access the productivity app via their web broswer. This interacts with the API endpoints using Flask in the middleware. Flask endpoints call on games written in Python. An external API is accessed via the middleware. To access the database, a data access layer handles all calls to the database.

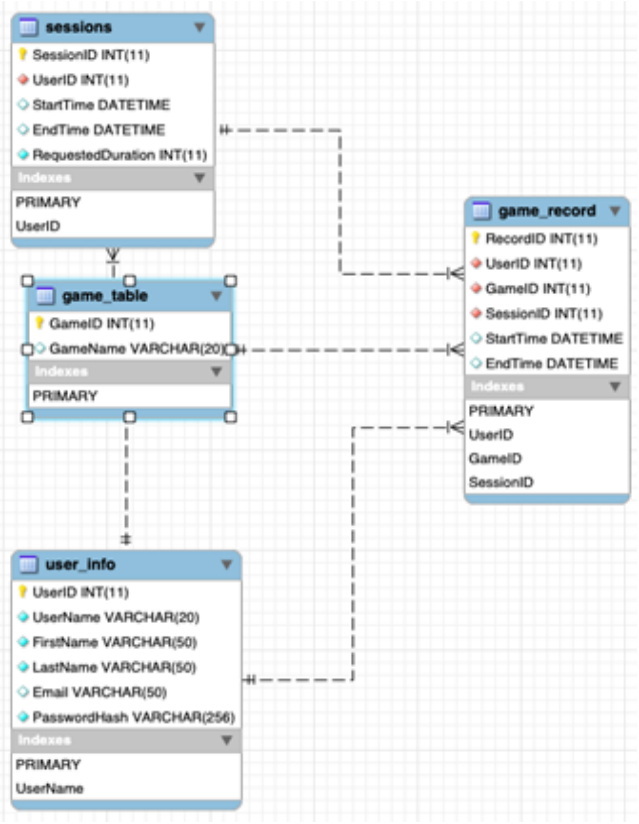

Figure 2 shows the schema for our database. User personal details are stored in the user_info table. Sessions represent the record of the user logging in and requesting their game time. Every time a user plays a game, it is stored in the game record. The game table contains a gameID for each game, and the game name.

Figure 2. Database design.

Technical requirements:

User navigation:	**Timer:**	**Games:**
- Main landing page needs to explain what the website is about. Main landing page needs to give two main central option buttons: Login or Create an account. - Users will be able to create an account on the app and log in. - After the user has logged in, they are redirected to the 'set timer' page where they can select a time. - Users will be able to choose from a small catalogue of games they can play while they are taking a break. - The user can view a basic game play history via the browse games page.	- Session start time logged into database. - Session end time logged into database. - When the timer expires the application will inform the user.	- When a user lands on any game's page, a request is sent to the server to create a new entry in the game session table in the database. - When a user leaves a game's page, a request is sent to the server to update the existing game session entry in the game session table to log end time. - User should be able to play each game until it is complete (either by finishing the quiz, or winning/losing/tieing). - A link back to the browse games page is available in every game page in case the user wants to change the game during or after their session.

Non-technical requirements:

- Code is readable, structured and maintainable
- Avoid repetition of code through focused methods and classes
- Use of simple and concise source code documentation to explain how functions/classes work, highlight any dependencies and returns

IMPLEMENTATION AND EXECUTION

Given the constraints of this project (using relatively new content and tools and working remotely) we used an Agile approach, following the Scrum framework to break up initially intimidating challenges into smaller incremental pieces of work which we could tackle as a team and work through systematically.

We assigned a scrum master to lead sprint planning meetings, maintain the Trello board, and diffuse emerging blockers. We used a Miro board to brainstorm project ideas and compare pros and cons. After agreeing on the topic, we made a list of requirements and wrote user stories. Then we came up with the architectural design of the project and broke it down into key components (backend/API, database, individual games, timer) and decided on a minimum viable product (MVP.) Our goal was to complete all the individual elements in our first sprint, integrate all of those elements in our second sprint, style, debug, and test during the third sprint, then spend the fourth and final sprint fixing any last minute problems and ensuring our code abides by the SOLID principles. We were able to stay on track and meet our goals.

We added all of the tasks in our product backlog to our Trello board as specific tickets. During our sprint planning meeting, we pulled the relevant tickets from our product backlog into the first sprint backlog, allocated tasks, and discussed potential challenges. Pair programming was used for many components as collaboration was effective for problem-solving. We repeated this process for all sprint planning.Although daily stand ups were not practical due to remote working and personal commitments, we consistently used Slack and Whatsapp to communicate daily, and any questions were shared with the team to avoid wasting additional time on one task through knowledge sharing.

At the end of a given sprint we held a sprint review/ retrospective meeting where we shared progress and ensured all team members understood feature functionality. We also had a chance to share any problems we ran into and discussed what worked from the last sprint, what didn't work, and how we could improve. Whenever we merged individual sprint branches back into the main branch (we managed our version control with git and Github) we did it together to make sure everyone was present to resolve conflicts.

We planned our project based on MoSCoW prioritisation. Our MVP was composed of the elements that fell under 'must have', and although not all aspects of the project initially planned were met, we achieved our main objectives to produce a functional productivity app with a timer and choice of games. As a team, we wanted to plan for potential deployment, and showcase our newly learnt skills using Python and SQL.

Tools:	Libraries and Packages:
● Miro: brainstorming the initial idea ● Slack: communication ● Google Meet: meetings ● Trello: product backlog ● Pycharm: IDE ● Postman ● MySQLWorkbench ● Cloud SQL: remotely access database	● Flask ● Jinja ● Bcrypt ● SQL Connector Python ● Request ● Random ● WTForms ● Flask-Login ● jQuery

Implementation challenges

- Connecting Python game logic to the frontend so users can interact with it while minimising the use of other languages (learning languages such as CSS, Ajax, JavaScript).
- Updating features that have dependencies with other classes or functions. For example, if a function makes a call to the data access layer.
- Issues with connecting and reconnecting after making a call. To overcome this, a @connect_to_db decorator used to open and close a connection every time a command is called.
- Implementing user authentication and storing hashed passwords in the database using the Flask Bcrypt package.
- We have made use of a range of features, which means that managing the dependencies was challenging. To overcome this, we created a requirements file.

TESTING AND EVALUATION

Testing strategy

We used unit testing to check that our functions behaved as we would expect by checking what they returned, and allocated testing features (such as games, timer, or login/registration) to each team member that wrote the code for that feature.

Trivia game testing**:** Using the unittest package, the tests check that the number of questions given to users is always between 0-10, the current question never exceeds the total number of questions, user answers are correctly checked (correct or incorrect)and that iterating through the trivia game instance will provide the same number of questions requested (hardcoded to 10).

Data access layer testing**:** Unit tests were completed using a test database (which used the schema for the productivity app). This prevented corrupting our production database. SetUp() and TearDown() methods were called during testing to create a user (this meant that the validate_user() function can be tested as you know they exist) then clear the table after testing. To access the test database, the @patch("FINALPROJECT.configDB_NAME", "test_db") decorator was added to all database function tests. It was difficult to test the data access layer functions, as testing could impact the production database and the majority of tests would only check that a call has been made to the database, rather than ensuring functionality.

Tic Tac Toe testing: The unit test checks if the game can identify when "o" wins and when "x" wins and also if it can identify when they don't win. It also tests if the game blocks its opponent when they are about to win. This is done by setting up the board in a way that gives the player an opportunity to win. Finally, it tests if the game will play in the middle first if not taken by the opponent.

Functional and user testing
We utilised manual functional testing, such as smoke testing whenever new code was added to our main branch, throughout the process to ensure that any individual changes made did not impact the functionality of the software. As we have developed a relatively small scale project, we used more informal user testing by inviting our friends and family to register and play on the productivity app. We were able to gain constructive feedback and make changes according to their experiences.

System limitations: *Incomplete features which fell under the 'could have' and 'would have' titles were:*
- Our project shows a basic personalised game play history, with the game, time spent playing, and date. However, the game history cannot be filtered to group games or dates together to identify trends.
- There is no displayed countdown timer visible during game play to let users know how much time they have remaining while on the website.
- Once a user is created, there is currently no option or functional method to delete that user.
- Due to the time constraints for the project, there are no accessibility controls, such as larger font sizes, screen readers, or alternative languages.

CONCLUSION
We were able to meet our goal, which was to design and implement a productivity application that is easy to navigate for users, and offers a choice of fun games to play in a structured way. We successfully coded 4 games that utilised a variety of tools and methods that are functional, and could demonstrate an understanding of more specialised engineering concepts, such OOP.

A key strength of our team's approach was our flexibility to challenges, and commitment to collaboration. Team members with additional skill sets, such as front-end development, were able to guide and advise those with limited experience using languages such as CSS, Javascript or Ajax. By using an agile approach, we were able to implement additional features, such as a wider range of game choices and improving the user experience, and by allocating tasks to each team member, code could be developed in parallel to produce an MVP quickly.

Future improvements:
A key rationale for our decision to design an online application was that we could create additional features. Therefore, we could improve our project by including a more extensive game history display, which shows historical data analytics and more detailed personalised user insights. An example of this, could be a visual peak times played per day for the individual user. In addition, to improve user experience, we could build a timer user interface component so that users can keep track of how much they have left.
The current Trivia game does not give users a choice of difficulty, categories or number of questions that they can answer. This could be extended to accept user input to further personalise the quiz.