# Analysis

## Background

For my A level project, I have decided to create a web application that allows different people, logged in on different computers, to interact with each other. I wanted to create something that is not only able to be scaled up or down depending on the number of users that access and use the program at the same time, but that is also robust in the case of the computer failing, like many modern day websites.

I thought the best way to achieve my goals was by creating a simple game which did not have many rules or intricate graphic designs. After investigating various card games and board games and after careful consideration I opted for the game Cards Against Humanity as the rules were simple, it was text based and the cards were easy to download and use legally.

Cards Against Humanity is an adult cards game in which players complete fill in the blanks in statements using preselected words or phrases. Each round a player amongst them is assigned the role of the Czar and a question card is selected from the pile. Every other player must then choose a card(s) from their hand (7 answer cards randomly assigned to the player) to fill in the blank(s) in the question card. The Czar is then assigned the task of choosing the funniest phrase between all the cards submitted. The player who submitted the phrase chosen by the Czar is the winner of the round and receives a point. The next round starts with a different Czar until someone reaches a certain number of points to win the game.

The game was released in May 2011, due to a Kickstarter campaign, and was created by a group of eight students, which were very influenced by the popular card game, at the time, Apples to Apples. The game first started out with the name “Cardenfreude” and involved a group of players writing the most humorous responses to a topic question. The name was later changed to Cards Against Humanity with the pre-written answers on the white cards, as it is known today. The title is used as it refers to the phrase “crimes against humanity”, that references the politically incorrect content.

## Who the system is made for

I first got the idea of creating this program when I was playing Cards Against Humanity with my family. The reason as to why I wanted to create a program that allowed users to play the game via the internet was because my brother was soon going to be going to university and it would be hard for us to play and have fun as a family while he was in university. That is when I had the idea of creating a website where all my family members could connect together and play CAH which would allow us to keep having fun the same way as if we were in the same room together. I want to create a program for users of any age to be able to connect and play with their family and friends as if they were in the same room together.

## Research of existing systems

To better clarify the problem at hand and to get ideas on how to handle the creation of the website I decided to research into other people’s web pages. The first website I visited was a website called <https://xyzzy.clrtd.com/zy/game.jsp> and is a place where players can join and create games to play with various other users while also being able to communicate with each other.

**Model 1:**

When I first opened the webpage, I was displayed with this interface:

Graphical user interface, website

Description automatically generated

Here the user is prompted to enter a nickname

After the user chooses a nickname, they are then allowed to join or create a game. When the game starts the interface shown is below:

Graphical user interface

Description automatically generated with low confidence

Graphical user interface, text, application

Description automatically generated

The image on the right shows what is displayed to the user. The Question card is shown on the left side of the screen in the black card with white text and the answer cards the player has been shown under the question card as white cards with black text. The answer cards that the other players submitted are shown to the right of the question card as white cards with black text.

The image on the right shows what it looks like if you open the chat log to communicate with every player in the game. By doing this it will cover up the answer cards of the player but will let the player communicate with all the other players in the game.

Criticisms:

* The main criticism about this webpage is that the design layout is plain and not very user friendly as the text is hard to read, however, the gameplay and the fact that it allows users to play together via the internet, makes this website a good example to follow for my project. I would like to use the login screen, the game table, and the chat from this website as examples to follow for my code.

Overall, this program is a very good example to follow for my code as it has most of the features I am thinking of implementing into my program

**Model 2:**

The second website I visited was a website called <https://lab.cardsagainsthumanity.com/>. When opening the program you are shown this interface:

Calendar

Description automatically generated

This website is a little peculiar as instead of playing with the usual rules of the card against humanity game the user is shown a question card and given a set of answer cards, they then must choose one of the answer cards given to them to fill in the blank, this is repeated until the user quits the website. Even though this program does not have many similarities with what I want to create I still wanted to take some inspiration for the user interface as it was very user friendly and colorful.

Criticisms:

* This website does not allow the user to play with other people together which is one of the main reasons as to why I decided to create a website instead of a normal program
* This website does not allow the user to play the version of the game I intend to code

Overall, I won’t be taking much inspiration from this program only the website design

## Project Objectives

* Phase 1:

1. Download cards for game from official website
   1. Extract question and answer cards
   2. Process answer and question cards
      1. Count number of blanks for the question cards and replace them with an identifiable string (e.g., “{}”)
      2. Strip unnecessary whitespace and unwanted characters
   3. Store cards in database
      1. Create two different tables: one for question cards and the other for answer cards
      2. Ensure each card has a unique identifier
2. Allow user to play CAH (cards against humanities)
   1. Allow user to create a player
   2. Create a game
   3. Allow user to join a game
   4. Allocate random answer cards to user
      1. Ensure only a maximum of 7 cards are allocated to user
      2. Ensure each answer card is different
      3. Ensure no more than one player has the same answer card at one time
   5. Display player’s answer cards
   6. Allocate one question card per round
      1. Ensure every player can see the question card
      2. Ensure that every player sees the same question card
   7. Display question card
   8. Allocate a player as the Czar
      1. Ensure that every player has been the Czar once before the user is allocated the Czar a second time
   9. Allow every player to choose an answer card
      1. Ensure that the Czar is not able to choose an answer card
      2. Ensure that the user must submit the same number of answer cards as the number of gaps in the question card
      3. Ensure that users cannot submit the same answer card more than once
   10. Remove chosen answer cards from player’s hand
   11. Show every player’s submitted answer cards
       1. Ensure every player’s answer cards are displayed once
   12. Allow Czar to choose a player’s answer card
       1. Ensure the Czar can only choose an answer card when all the players have submitted their answer cards
       2. Ensure the Czar can only choose one player’s answer cards
       3. Ensure the Czar does not know from which player each answer card was submitted
   13. Display winner of round
   14. Allocate a point to the player, whose answer card was chosen by the Czar
       1. Ensure only one point is allocated to the player
       2. Ensure that the answer card chosen by the Czar corresponds to the player being awarded a point
   15. If the number of points the player who won the point has is the same as the number of points needed to win the game, end the game
   16. Start a new round
   17. Display the winner of the game
3. Allow users from different computers to access and play the same game together
   1. Ensure games can be resumed and that no data is lost if the computer crashes. Store game state in a database
      1. Create table for players
         1. Store which answer cards they have in their hand
         2. Store their current choice
         3. Store the number of points each player has
      2. Create table for the games currently in progress
         1. Store the players in the game
         2. Store the question card from the current round
         3. Store the player that is the Czar of the current round
   2. Ensure that the user interface does not freeze when waiting for an event (for example another player to choose a card)
   3. Coordinate between all the users centrally as users cannot communicate directly with each other as they are on different computers

* Phase 2:

1. Create and implement a more attractive graphical design
2. Allow for more than one game to be played at once
   1. Allow users to create or join an existing game
   2. Allow users to invite other users
   3. Allow users to make the game private or public
   4. Allow users to configure the game (for example allow them to choose which version to play or let users choose how many points are need to win the game)
   5. Allow users to delete a game
   6. Display number of players and player ids in the game
3. Allow users to log in/sign up
   1. Allow user to delete account
4. Allow users to communicate with each other via a chat
5. Be able to cope with a larger amount of players using the website at the same time

## Observations

In order to meet the objective number 3 of phase 1, it is necessary to have a frontend that handles the user interface on each of the user’s computers, and a backend that communicates between the database and all of the frontends, to manage the flow of the game, as well as, stopping unnecessary data from being leaked out to the users. This type of programming is an example of client server model where each frontend is the client, and the backend is the server. As well as that, to complete the objective 3(b) the program would need to be handled in a way where the code does not wait for an event to occur before continuing to process the user interface. This is called asynchronous programing.

There are two types of programming, the first, being the usual way, is called synchronous programming where operations are carried out sequentially and is the programming style I have been coding my entire life. The second one is called asynchronous programming, which is the one I will be using for this program, and is the one where, when the code is running an event, the control is given back to the user interface and keeps going with the rest of the code until the user interface comes back with a response to the event. For example, if the code is expecting the user to input a value, rather than waiting for the response it can be asked to be called back when the user submits their answer. For this reason I will be using this type of programming as it will not freeze the interface while an event is running.

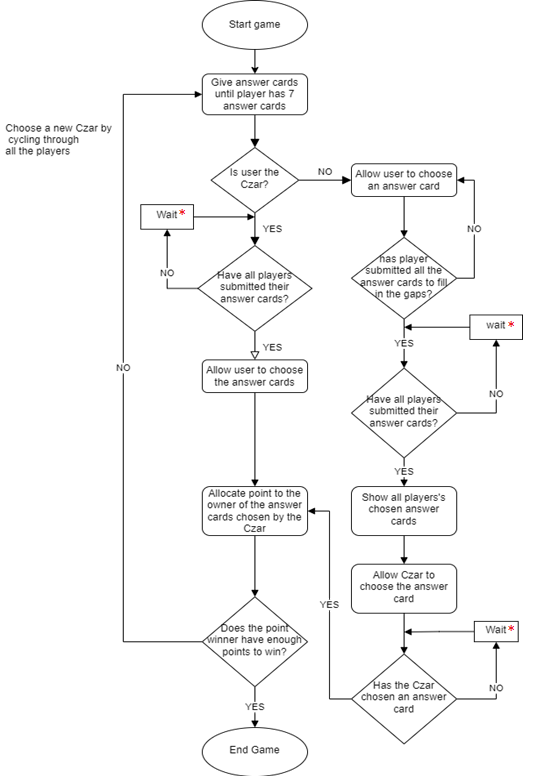
Graphical user interface, application

Description automatically generated

This diagram represents a rough representation of the main components needed for the game to function properly. There must be Players that can either be normal players or the Czar, where each player must have a score and 7 answer cards. There must be two types of cards: question cards and answer cards, where the question cards have a certain number of gaps. Each game must have a certain number of players and a question card

# Design

## Flowchart





This flowchart outlines the basic steps need to carry out to play the game. This is how I would like my code to carry out the process of playing the game. It will work by allocating different answer cards to each player and if the user is the Czar allowing them to choose a players answer card, and if the user is not the czar, they must choose an answer card. This is repeated until there is a player that has reached the maximum number of points

\*As previously mentioned, “waiting” is done asynchronously. This is covered in more detail below in the key algorithms about the function timeout in the frontend

## Data Models

I have now established the basic structure of the program, I must now establish a way of how the storing of data is going to be implemented. I have already decided that the best way of storing the data in this case is going to be the use of a database as it allows the establishment of relationships between the data. Another reason as to why I chose to store the data in a database is due to the fact that if the program were to save all the data in the database it would be storing all the state in the database, so if it were to crash it would not lose any data. This would also support the use of other backends if there were to be an increase in demand, this is called scaling.

The database is accessed in my program in three different ways. The first is for the creation of the database and is coded in python in a jupyter notebook, the second is for the accessing and maintaining of the database done in “Beekeeper-Studio” where I can forcibly change the values of the database manually. The last one is in the backend coded using python and is where the data is processed and sent to the frontend for it to display it to the user.

The database application that I will be using will be the application called “BeeKeeper-Studio” which performs the basic functions needed for the storing of data, as well as, supporting the editing and creation of the database on both python a JavaScript.

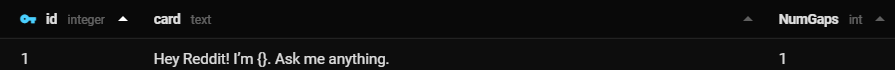
The database will have four different tables each storing four different things, however, two of those tables will be similar and closely related: the answer cards and the question cards.

The first table will be the storing of Answer Cards:



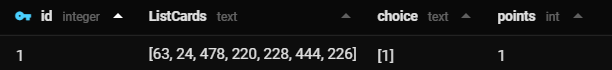
This image shows what will be storing in the table with an example. Each answer card will be store here with the unique identifier “id”, which is an integer, followed by the actual contents of the card stored in the “card” column as a string.

The Second table will be the storing of the Question Cards:



This image shows what I will be storing in the table with an example. Each question card will be stored with the unique identifier “id” as an integer used to identify each card, followed by the actual contents of the question card stored in the “card” column as a string, followed by the number of gaps stored in the “NumGaps” column.

The third table will store information about the player:



As shown in the image when a new player is created they will be stored here along with the unique identifier “id” as an integer, followed by the ids of each answer card assigned to them in a list stored in the column “ListCards” as a string. The choice of each player will also be stored, stored in the column “choice” as a string which will initially be empty, followed by the number of points stored in the column “points” as an integer that will increase by one each time the player wins a round

The last table will be storing information about the game:

Graphical user interface, text, application, website

Description automatically generated

As shown in the image when a new game is created they will be stored here with the unique identifier “id”, followed by a list of the player’s ids participating in the game stored in the column “ListPlayerID” as a string. The current question card is also stored in the column “QCard” as a string followed by the current Czar’s id stored in the column “CzarId” as an integer

For any table that contains a list of strings, since SQL does not have a data type that is similar to a list, to be able to store them, they would have to first be turned into a string and then stored. The same goes for the extraction of the lists from the database, they would have to first be turned from a string into a list to be able to then use the data.

Table

Description automatically generated with low confidence

This diagram shows a visual representation of the tables that will be stored in the database and of the information stated above

Here is another representation of the different tables I intend to create.

|  |
| --- |
| AnswerCards ( id, card ) |
| QuestionCards ( id, card, NumGaps ) |
| Players ( id, ListCards, choice, points ) |
| Games ( id, ListPlayerID, QCard, CzarID ) |

For each table the underlined parameter is the primary key used to uniquely identify each component stored in the tables. This makes it imperative that I make sure that each variable stored in the primary key is unique, fortunately, when creating a table there is an SQL command that allows for this to happen automatically.

With this information I can create an entity relationship diagram shown below:

Graphical user interface, website

Description automatically generated

Here, each Game can have various Players while each player can only appear in one Game. As well as this, one Answer Card and one Question Card can appear in various Games but one game can only have one of each question and answer card. Not only this but, as shown by the diagram, one Player can have various Answer Cards but only one specific card can appear in one player

**Creation of database:**

The reading of the cards will be written in python and the creation of the database will be written in a mix of python and SQL, by importing a python library called sqlalchemy. The frontend will be written in a mix of HTML and JavaScript and the backend will be written in python. The application I will be writing the creation of the databases will be in an open-source web application called “Jupyter Notebook” as it supports the use of SQL and can divide the code into cells in which each cell can be run separately which makes it much easier to test each separate part of the code. The rest of the code will be written in streamlined code editor called “Visual Studio Code” as it supports the development operations like debugging, task running and version control.

The reading of the cards will be one of the hardest parts of the code as I will have to download a pdf file from the internet and turn it into a text file. To overcome this I first had to find a place where I could download the cards legally, after careful consideration, I decided on downloading the normal game mode cards from the website <https://s3.amazonaws.com/cah/CAH_MainGame.pdf> and the family friendly cards from the website <https://www.cardsagainsthumanityfamilyedition.com/pdf/CAH_Family_SmallCards.pdf>. However, this only downloaded the cards into pdf files, which takes me to the second problem I had to overcome, how do I turn the pdfs into text files that I can then edit and use? To do this I decided to use a package from the site <https://pypi.org/project/pdfminer/> that was installed via the anaconda prompt and using the commands python pdf2txt.py -o CAH\_MainGame.txt CAH\_MainGame.pdf and python pdf2txt.py -o CAH\_Family\_SmallCards to convert the pdfs into text.

After converting the pdfs into text files, I will have to turn each card into a nicely formatted string with no unwanted punctuation points and weird characters. After this the tables of the database will have to be created using the commands that follow:

|  |
| --- |
| cur.execute("""CREATE TABLE AnswerCards  (id INTEGER PRIMARY KEY AUTOINCREMENT,  card text)""") |

Creates the table AnswerCards in the database

|  |
| --- |
| cur.execute("""CREATE TABLE QuestionCards  (id INTEGER PRIMARY KEY AUTOINCREMENT,  card text,  NumGaps int)""") |

Creates the table QuestionCards in the database

|  |
| --- |
| cur.execute("""CREATE TABLE Players  (id INTEGER PRIMARY KEY AUTOINCREMENT,  ListCards text,  choice text,  points int)""") |

Creates the table Players in the database

|  |
| --- |
| cur.execute("""CREATE TABLE Games  (id INTEGER PRIMARY KEY AUTOINCREMENT,  ListPlayerID text,  QCard text,  CzarID int)""") |

Creates the table Games in the database

Each card will then have to be appended into a list and inserting each card into the database using the SQL commands that follow:

|  |
| --- |
| for index in range(len(FinishedQCards) - 1):  cur.execute(f"""INSERT INTO QuestionCards (card, NumGaps) VALUES ("{FinishedQCards[index][0]}", {FinishedQCards[index][1]})""") |

This line of code loops through the question cards and saves the number of gaps and the text of the card into the table QuestionCards

|  |
| --- |
| for index in range(len(AnswerCards) - 1):  cur.execute(f"""INSERT INTO AnswerCards (card) VALUES  ("{AnswerCards[index]}")""") |

This line of code loops through the answer cards and saves the text of the card into the table AnswerCards

## Client Server model diagram

As stated in the analysis stage under the observations for the program to be able to work as a website it needs to be broken down into the backend (carries out most of processing needed to run the program and communicates with the frontend and the database) and frontend (displays the game information received from the backend).

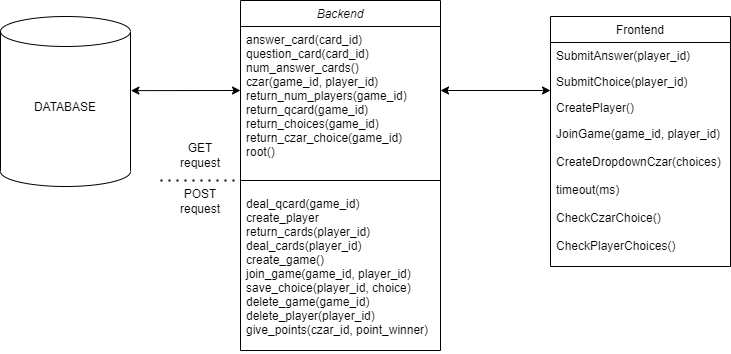
Graphical user interface

Description automatically generated

This diagram shows a visual representation on how the communication between the backend, frontend and database will work. Each user will have one frontend which will display the information to the player which it receives from the backend and gets an input from the player. The frontends can communicate with the backend but can neither communicate with the other frontends and the database. The backend acts as a kind of bridge between the frontends and the database, as well as, keeping track of the state of the game and handling queries from the frontends and serving the web page. The Backend can communicate with both the database and all the frontends

This type of programming is an example of a client server model where the client is each frontend and the server is the backend

## key algorithms



This diagram represents the key algorithms used in both the backend and the frontend. The backend is divided into two different request methods, the GET request, that just returns a value, and the POST request, changes the backend as well as being able to return a value.

* The algorithms from the backend are all explained below
* CreatePlayer: creates a player when the button “Create Player” is pressed
* JoinGame: joins the game with id game\_id after creating the player
* CreateDropdownCzar: creates a dropdown box containing the answer cards submitted by each player
* CheckPlayerChoices: calls the backend and asks it if all players have submitted their answer cards and if they have it asks for the answer cards submitted
* CheckCzarChoice: calls the backend and asks if the Czar has submitted their choice and if they have it asks for the id of the player that won the point
* SubmitAnswer: sends the backend the answer cards submitted by the player. If the question card has more than one gap it loops until there are no more answer cards needed to be submitted
* SubmitChoice: sends the backend the player id of the player that submitted the answer cards chosen by the Czar
* **timeout**: since the program is asynchronous, for the part of the program that the needs to wait for all the players to submit their answer cards or for the Czar to submit their choice, would not be possible. The reason for this is due to the program not waiting for functions to finish before continuing with the rest of the program. To combat this, this function was created, what it does is it sets up a timer and when the timer runs out it calls my program and checks if what needed to be done has been done, and if it has not the function creates a new timer, this is repeated until what needed to be done has been done. It is a way of periodically checking the condition of the program without blocking the user interface. This piece of code was taken from the website <https://stackoverflow.com/questions/33289726/combination-of-async-function-await-settimeout>
* **Explain frontend algorithms**
* **Update algorithm diagram**
* The algorithms from the backend are all explained below
  + GET Requests
* answer\_card: returns answer card text, with id card\_id, as an object
* question\_card: returns question card text and number of gaps, with id card\_id, as an object
* num\_answer\_cards: returns the total number of answer cards in the database
* czar: checks if the player with player\_id is the czar in the game game\_id and returns a Boolean value depending if it is true or not
* return\_num\_players: returns the number of players in the game with id game\_id
* return\_qcard: returns the current question card from the game with id game\_id, and all its information
* return\_choices: returns the player’s choices from the game with game\_id, with their respective ids
* return\_czar\_choice: returns the player id of the player that has been chosen by the czar
* root: returns the html code from the front end so that the user can run the code when the website is opened
* POST Requests
* deal\_qcard: changes the current question card to a different random one and stores in the database in the game with id game\_id. It returns the question cards text
* create\_player: creates a player and saves in the Players database
* return\_cards: returns the answer card ids the player with player\_id has
* deal\_cards: it gets random answer cards that have not already been dealt to other players and it saves it to the player with player\_id’s ListCards in the database Players
* create\_game: creates a new game and saves in the database Games
* join\_game: appends the player id with id player\_id into the list of ListPlayers that is found in the game with id game\_id
* save\_choice: saves the players choice into the database Players with id player\_id
* delete\_game: deletes game with id game\_id
* delete\_player: deletes player with id player\_id
* give\_points: awards a point to the player with id point\_winner and saves the player’s id into the czar’s choice

## Models of how the User Interface will look like

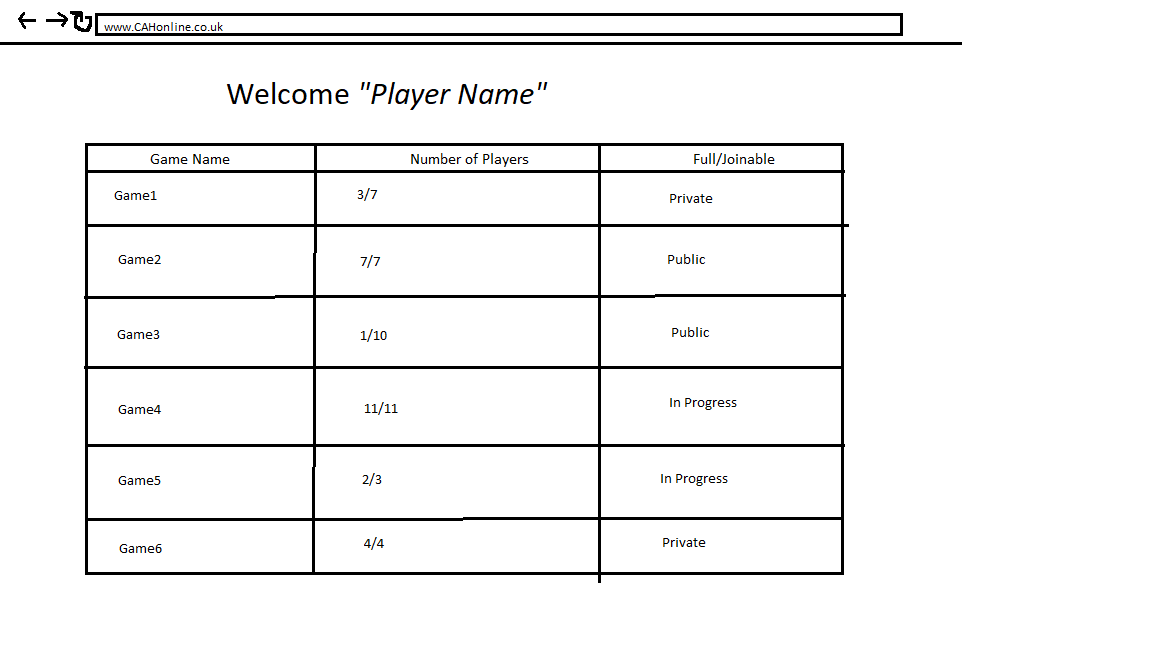
Apart from the algorithms, I must think about how the user interface is going to look like for each stage of the program. The following Illustrations show the option I have considered, as well as, what the current user interface looks like.

When the website is first opened I wish for it to look something like this:

Graphical user interface, application

Description automatically generated

The user will first be prompted for a username that they must enter into the textbox and then the user must then press the “Create Player” button to continue to the next interface. If I choose to implement this feature I must make sure that the user cannot continue onto the next interface until they have inputted a reasonable name



Properties:

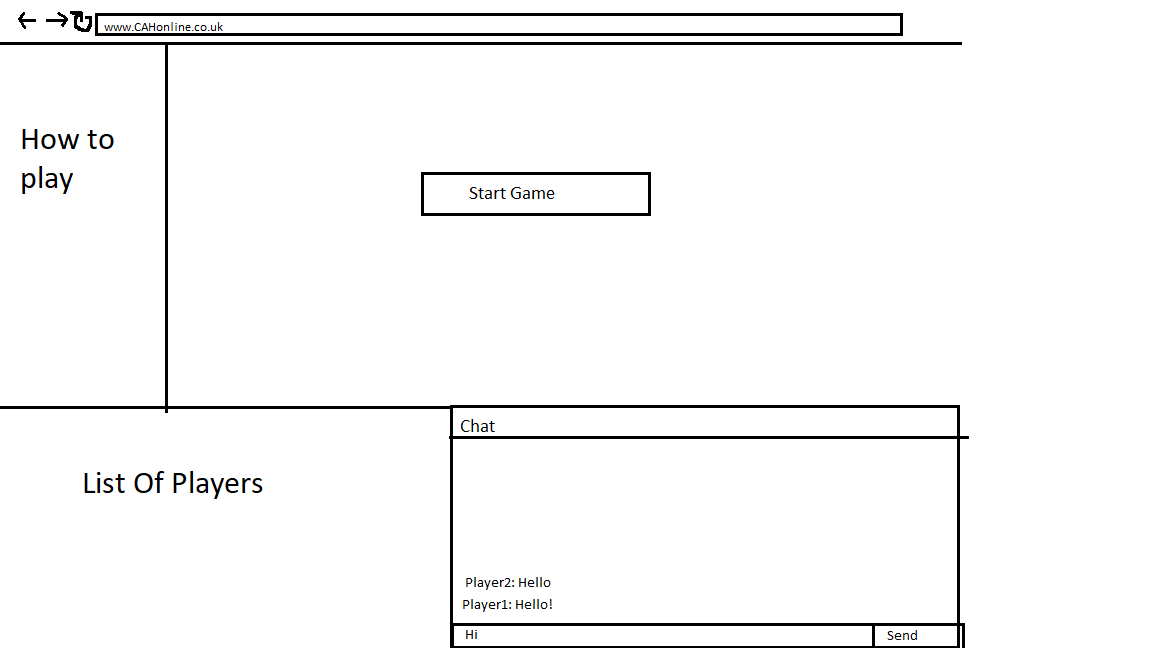
-Table with all the games and players

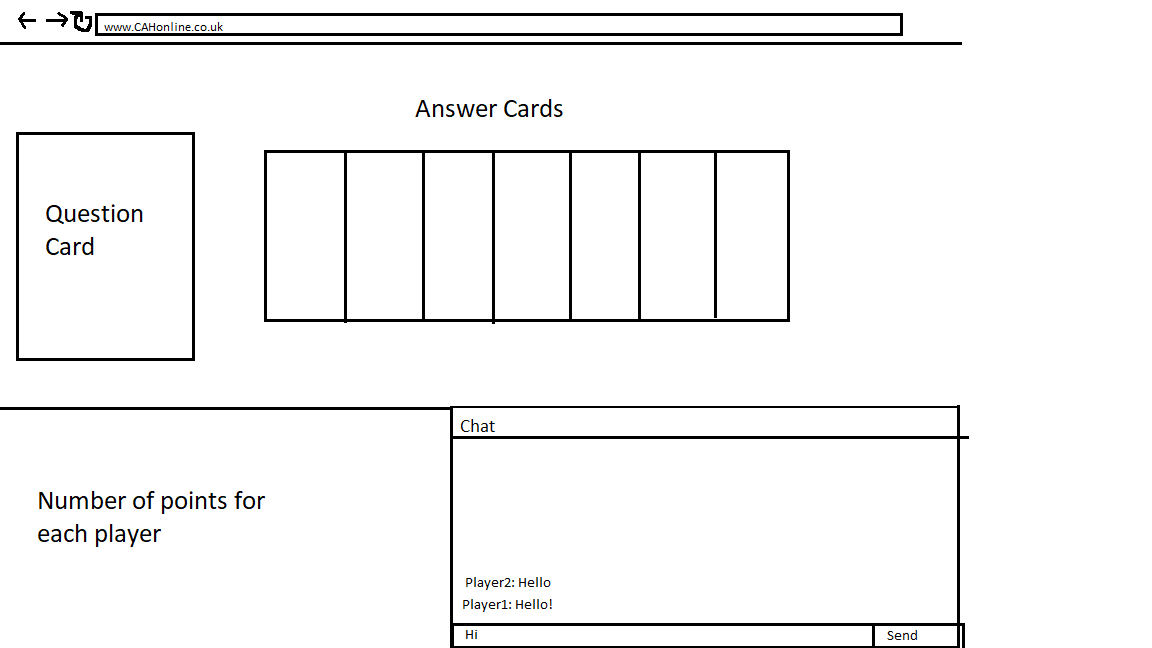
-Button that allows the user to join the specified game

The next interface is will have to be where the user will be allowed to choose a game lobby they would wish to play in or be allowed to create a new game lobby with the properties they want. The implementation of this feature will be hard and I don’t know if I will be able to make it happen as, to allow the user to do this, the program must be able to support more than one game being carried out at the same time

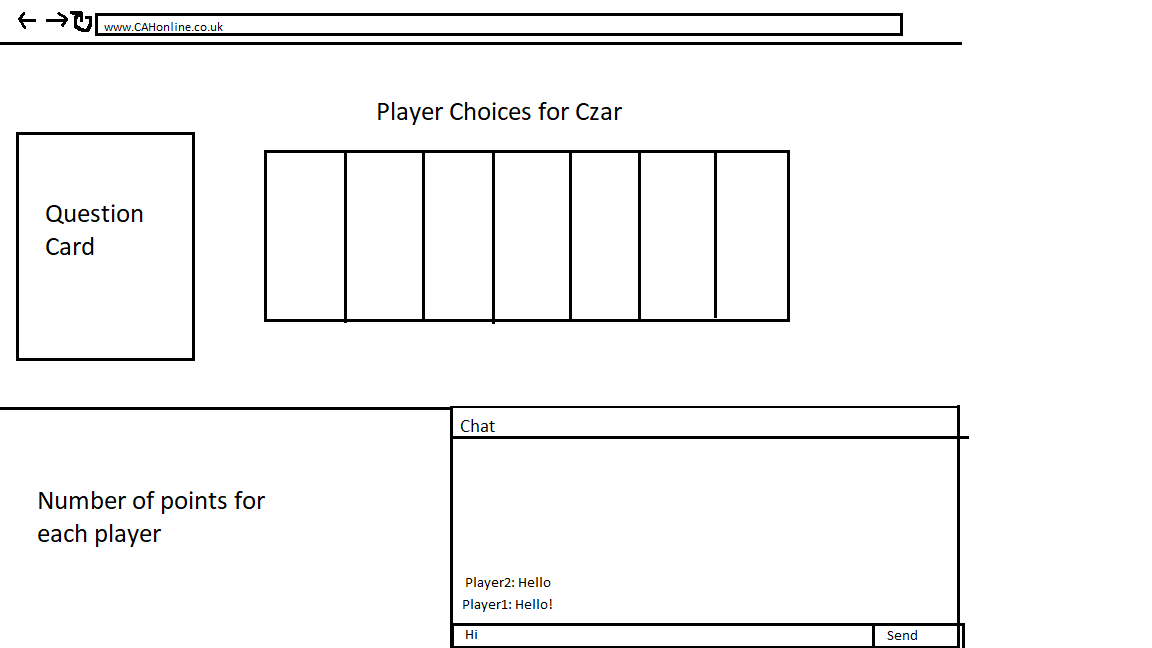
**Navigation of Screens when joining a new game:**

The user will be allowed to start the game, as well as, be given information on how to play the game and a list of the players that are participating in the game





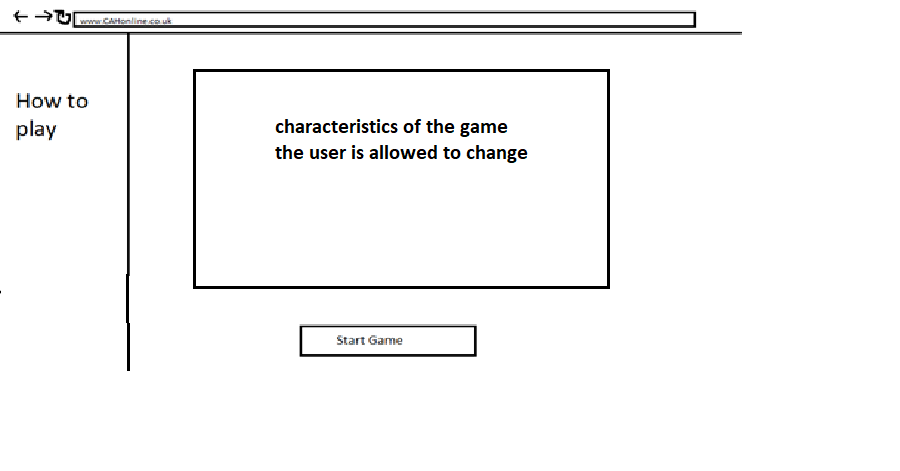
When the game starts the player will then be given the role of either a normal player or the role of the Czar. This Graph shows how it will look like if the user is given the role of a normal player. They will have a list of their answer cards in the center, as well as, the current question card on the left of that. At the bottom left of the screen the user will be able to see how many points each player has



If the player is given the role of the Czar they are shown a similar interface as the normal players, the only difference will be that instead of showing them their answer cards, they will be displayed the answer cards chosen by the rest of the players

When the user Joins a game, each diagram shows the feature of allowing the user to communicate with all the other players in the current game via the chat feature, however, this feature will be very hard to implement and I am unsure if it will even be implemented into the final code. The reason as to why it will be hard to implement is due to the fact that each frontend cannot communicate with each other directly, so to be able to have this feature the frontend must first tell the backend what they want to send to every player and the backend must then send the message to the rest of the frontends playing the game.

**Screen of when the user decides to create a game:**



When the user presses the button to create a game they will be shown this interface. Here the user will be prompted to customize the rules of the game, such as allowing the user to choose which version to play and how many points are needed to win the game

**Current Interface:**

The first screen the user encounters upon opening the web page is this one. The user is prompted to enter a Name into the textbox and then must press the “Create Player” button to continue and save the name chosen. I have to make sure the user is not able to create a player without inserting a name

Graphical user interface, application, Word

Description automatically generated

After creating a player, the user is then displayed this interface. The user is met with a piece of text that welcomes the user. Currently there is only one game, therefore, the user is only able to press the “Join Game” button to allow them to join the current game

Graphical user interface, application, Word

Description automatically generated

If the user is not the Czar they will be displayed this set of interfaces

After joining the game, the user is then displayed this interface. Here the player is shown the question card as text and shown their answer cards in a dropdown box. The user must then choose a card and press the “submit answer” button

Graphical user interface, application, Word

Description automatically generated

With this approach I must make sure the user is not able to press the button without choosing a card beforehand. As well as this, if there is more than one gap in the question card I must make sure that the user is able to choose the correct amount of cards

Graphical user interface, application

Description automatically generated

Displays a text message informing the user that their choice has been submitted

If the user is the Czar they will be displayed this set of interfaces

Graphical user interface, application, Word

Description automatically generated

After joining the game, the user is then displayed this interface. If they are the Czar they are displayed the question card as text and the answer cards submitted by the all the players as text in a dropdown box. The user must then choose an answer and press the “submit choice” button

With this approach I must make sure the user is not able to press the button without choosing an answer beforehand

Graphical user interface, application

Description automatically generated

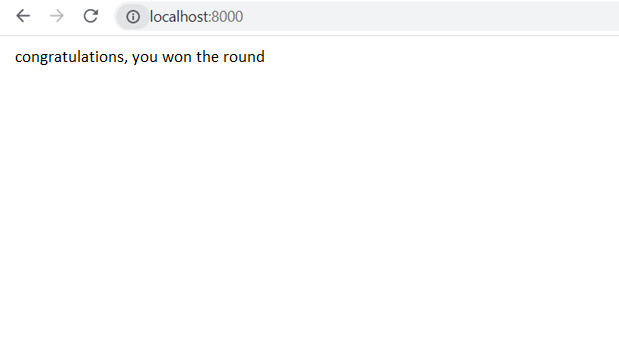
After submitting their answer, they are then displayed this message to inform the user they have successfully chosen an answer

If the user did not win the round they wont be awarded a point and they will be displayed the message bellow:

A picture containing graphical user interface

Description automatically generated

Otherwise, they would be displayed the message below:



In the frontend to change the interface I decided to have all the html text messages, dropdown boxes and buttons hidden from the user at the beginning and using JavaScript, at appropriate moments of the code, displaying each component. I could have done it in a way where the components are created when they are needed in JavaScript and then deleted after using them, however, this way was much easier and so I opted for it.

Write pseudocode of how the program will look like

# Technical Solution

## Most important snippets of code explained

Here I will take snippets of the most important parts of the code and explain each in detail

### Backend

One of the main and most important functions in the backend is the distribution of answer cards to each player. Below are some of the most important parts of the code:

for index in cur.execute("SELECT id FROM Players"):

        num\_players.append(index[0])

Here what the code is doing is its appending to a list the ids of all the players participating in the game. However, currently, as there is only one game, all the players in the database will be playing in the same game and that is why I am taking the ids from the table Players instead of the column ListPlayerID from the table Games.

After extracting all the ids from the players participating in the game the computer must then extract the cards that all the players currently have in their hand and remove it from the possible answer cards the user can obtain randomly. This can be seen in the code below:

for ids in num\_players:

        list\_cards = cur.execute(f"SELECT ListCards FROM Players WHERE id = {ids}").fetchone()[0]

        list\_cards = json.loads(list\_cards)

        for cards in list\_cards:

            unused\_cards.remove(cards)

Here you can see that the code loops through all of the players and takes the answer cards in their hand from the database and removes it from the unused\_cards list which contains all the answer card ids that have not yet been used.

After this the code must realize that if the user already has some answer cards it mut only give the needed cards to fill it up to 7 answer cards and no more. This can be seen in the code below:

            if (ids) == player\_id:

                number\_cards += 1

this if statement is found inside the for loop that loops through the ids of the players in the card and checks if the user has any cards and if they already have some cards it adds to a variable that counts how many cards the player currently has.

list\_cards = cur.execute(f"SELECT ListCards FROM Players WHERE id = {player\_id}").fetchone()[0]

    list\_cards = json.loads(list\_cards)

    for index in range(max\_cards - number\_cards):

        random\_num = random.randint(1, len(unused\_cards))

        list\_cards.append(unused\_cards[random\_num])

        unused\_cards.pop(random\_num)

    cur.execute(f"UPDATE Players SET ListCards = '{list\_cards}' WHERE id = {player\_id}")

Here the program takes the answer card ids the player currently has from the database and gets a random answer card from the unused answer cards appending it to their hand and removing it from the list of unused answer cards. It repeats this until the user has exactly 7 answer cards in their hand.

After this the code must then take the answer card ids the player has in their hand and then find the text of the answer card in the database and return it to the frontend. This can be seen in the code below

    for card\_id in list\_cards:

        acard = await answer\_card(card\_id)

        acard = acard["text"]

        temp\_list.append(acard)

    list\_cards = temp\_list

    con.commit()

    return list\_cards

Another main function is the returning of the choices to the frontend. The main parts of the code can be seen below:

list\_players = cur.execute(f"SELECT ListPlayerID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players = json.loads(list\_players)

    czar\_id = cur.execute(f"SELECT CzarID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players.remove(czar\_id)

Here the code takes all the ids of the players from the column ListPlayerID from the table Games and removes the Czars id from that list as the Czar will not have chosen any cards yet.

    for player\_id in list\_players:

        choice = cur.execute(f"SELECT choice FROM Players WHERE id = {player\_id}").fetchone()[0]

After that it selects all the choices from the players, this can be seen in the code above.

The code must then append the choice to a list of all the player’s choices, however, due to the fact that it is possible that not all the players have submitted their answer cards, it must then perform a try operation to check if the players have all submitted their answer cards and if they haven’t it will return a false statement to the frontend telling it that not all the players have submitted their answer cards and if they all have it will return a true statement along with the choices. This can be seen in the code below:

        try:

            choice = json.loads(choice)

            list\_choices.append((player\_id, choice))

        except:

            choice\_in = False

            break

Another function that is important for the code is the function that allows the player to join the game. This function saves the player’s id into the list of players the game has and makes the last person to join the game the Czar. This can be seen in the code below:

    try:

        games = cur.execute(f"SELECT \* FROM Games").fetchone()[0]

    except:

        game\_id = await create\_game()

        game\_id = game\_id[0]

It first checks if there are any games in the database and if there isn’t it creates one and takes the game’s id. This can be seen above.

After this it takes the list of player ids and appends the players id into the list. This can be seen in the code below:

list\_players = cur.execute(f"SELECT ListPlayerID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players = json.loads(list\_players)

    list\_players.append(player\_id)

After this it takes the updated list of players and saves it to the database. This can be seen in the code below:

    cur.execute(f"UPDATE Games SET ListPlayerID = '{list\_players}' WHERE id = {game\_id}")

    cur.execute(f"UPDATE Games SET CzarID = {player\_id} WHERE id = {game\_id}")

    con.commit()

The last piece of code that is important is the root function, this function is used at the beginning of the code when the website is opened, and it just returns the frontend code to the website so that it can be run. If this was not there the code would not even run and nothing would happen, the user would just see a blank screen. The code is very simple and can be seen below:

@app.get("/")

async def root():

    return HTMLResponse(open("CAH.html", "rt").read())

### Frontend

One of the most important functions in the frontend is when the button to join the game is pressed. Here the code must distinguish whether the user is a normal player or a Czar and depending on that it will either display their answer cards or wait for all the other players to submit their answer cards. The code for this can be seen below:

            await fetch("/join\_game/" + game\_id + "/" + player\_id, { method: "POST" });

            $("#join-game").css("display", "none");

            $("#question").css("display", "inline");

            response = await fetch("/return\_qcard/" + gameId);

            json = await response.json();

            qcard = json;

            numGaps = qcard[1];

            gapCountdown = numGaps;

            $("#question").html(qcard[0]);

First it calls the function join\_game from the backend and hides the button and the displays the question card by calling the backend function to find the current question card. This can be seen in the code above.

response = await fetch("/czar/" + gameId + "/" + player\_id);

            json = await response.json();

            czar = JSON.parse(json);

            if (czar == false) {

.....

            } else {

.....

After this the code calls the backend function czar to check if the player is the current czar. If they are the current Czar the code will wait for the rest of the players to submit their answer cards and if they are not the Czar their answer cards will be displayed allowing them to choose an answer card. This can be seen in the code above

Another key functions is the timeout function. This function is used to periodically check a condition without taking the control from the user. An in depth explanation can be found above in the key algorithms. The code can be found below:

        function timeout(ms) {

            return new Promise(resolve => setTimeout(resolve, ms));

        }

Another key function is the creation of the dropdown box for the czar and the normal players used to represent the answer cards submitted by the players and the answer cards in the players hand respectively. The creation of the dropdown box displayed to the Czar differs slightly to the creation of the dropdown box displayed to the normal players as there is a fixed number of choices for the one displayed to the normal player whereas the other dropdown box does not a have restriction on how many choices there can be. The code for the creation of the dropdown box displayed to the Czar is shown below:

$("#czar-choice").find("option").remove();

It first deletes the dropdown box “czar-choice” to make sure there is no already existing dropdown box with the same name. This can be seen in the code above.

for (var choice of choices) {

                for (var card of choice[1]) {

                    response = await fetch("/answer\_card/" + card);

                    json = await response.json();

                    ans = json;

                    playerChoice = playerChoice + ans["text"] + " + "

                }

After this it goes through the choices submitted by the players and loops through each answer card in the choices adding it to a string to put it in the same line in the dropdown box. This can be seen in the code above.

                var option = new Option(playerChoice, choice[0]);

                $("#czar-choice").append(option);

After this it adds each set of answer cards chosen by the players into an option in the dropdown box. This can be seen in the code above.

The creation of the dropdown box for the normal players is slightly different as it is hardcoded in HTML and is hidden and displayed when it is necessary. The adding of the values is done after the user presses the join game button. The code is shown below:

    <select id="answer" style="display: none;">

        <option id="ans1" value=1></option>

        <option id="ans2" value=2></option>

        <option id="ans3" value=3></option>

        <option id="ans4" value=4></option>

        <option id="ans5" value=5></option>

        <option id="ans6" value=6></option>

        <option id="ans7" value=7></option>

    </select>

The last key function is the submission of the answer cards. The code is described in the code below:

if (gapCountdown != 1) {

.....

            } else {

.....

It first checks if the number of gaps in the question card is greater than one. This can be seen in the code above.

If the remaining question card gaps is more than one then the code creates a string in the style of an unfinished list where all the following answer cards will be added to, as well as, reducing the number of gaps needed to go through by one. This can be seen in the code below:

                choice = $("#answer").val();

                listAns = listAns + choice + ",";

                gapCountdown -= 1;

Otherwise, if the number of gaps remaining is equal to one it saves the last choice in the string and makes it in a format that looks like a list and saves that sting in the backend. This can be seen in the code below:

                choice = $("#answer").val();

                listAns = listAns + choice + "]";

                await fetch("/save\_choice/" + player\_id + "/" + listAns, { method: "POST" });

After that the code resets the variables and hides the dropdown box, question card and the button, and displays a piece of text informing the user that the cards have been submitted successfully. This can be seen in the code below:

                listAns = "[";

                $("#question").css("display", "none");

                $("#answer").css("display", "none");

                $("#submit-answer").css("display", "none");

                $("#answer-submited").css("display", "inline");

After this the code then, using the timeout function, periodically checks if the czar has submitted their answer card and, if they have, continues with the rest of the code. This can be seen in the code below:

while (czarChoice[0] == false) {

                    await timeout(1000);

                    czarChoice = await CheckCzarChoice();

                }

After this the code either displays a text message informing them they either won or lost the round by checking whether the Czar’s choice received from the function CheckCzarChoice is the cards chosen by the users id. This can be seen in the code below:

if (czarChoice[1] == player\_id) {

                    $("#answer-submited").html("Congratulations, you won the round!!!");

                } else {

                    $("#answer-submited").html("Unfortunately, you didnt win the round");

## Backend

# run this with uvicorn main:app and go to http://localhost:8000

# for testing go to http://localhost:8000/docs

from ctypes import pointer

import string

from typing import Awaitable

from fastapi import FastAPI

from fastapi.responses import HTMLResponse

import sqlite3

import json

import random

con = sqlite3.connect('CAH.db') # connects to the database

cur = con.cursor() # creates a cursor

app = FastAPI()

@app.get("/answer\_card/{card\_id}")

async def answer\_card(card\_id: int):

    # returns the answer card with id card\_id

    card = cur.execute(f"SELECT \* FROM AnswerCards WHERE id = {card\_id}").fetchone()

    # returns the card in the form of a dictionary

    return {

        "id": card[0],

        "text": card[1],

    }

@app.get("/question\_card/{card\_id}")

async def question\_card(card\_id: int):

    # returns the question card with id card\_id

    card = cur.execute(f"SELECT \* FROM QuestionCards WHERE id = {card\_id}").fetchone()

    # returns the card in the form of a dictionary

    return {

        "id": card[0],

        "text": card[1],

        "num\_gaps": card[2]

    }

@app.post("/reset\_choices/{game\_id}")

async def reset\_choices(game\_id: int):

    # resets the choices of the players in the db by giving them a (NULL) value

    list\_players = cur.execute(f"SELECT ListPlayerID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players = json.loads(list\_players)

    # loops through all the players and sets their choices to a null value

    for players in list\_players:

        cur.execute(f"UPDATE Players SET choice = '(NULL)' WHERE id = {players}")

    con.commit()

@app.post("/change\_czar/{game\_id}")

async def change\_czar(game\_id: int):

    # changes the czar

    count = 0

    # gets previous czar

    czar\_id = cur.execute(f"SELECT CzarID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players = cur.execute(f"SELECT ListPlayerID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players = json.loads(list\_players)

    # finds the location of the czar's id in the list ListPlayers from the db

    while czar\_id != list\_players[count]:

        count += 1

    # checks if the last czar was at the end of the list and if they were it sets the next czar to be the first one in the list

    if count == len(list\_players) - 1:

        cur.execute(f"UPDATE Games SET CzarID = {list\_players[0]} WHERE id = {game\_id}")

    else:

        cur.execute(f"UPDATE Games SET CzarID = {list\_players[count + 1]} WHERE id = {game\_id}")

    con.commit()

@app.post("/remove\_chosen\_cards/{player\_id}/{cards\_chosen}")

async def remove\_chosen\_cards(player\_id: int, cards\_chosen: str):

    # removes the chosen answer cards of the players

    list\_cards = cur.execute(f"SELECT ListCards FROM Players WHERE id = {player\_id}").fetchone()[0]

    list\_cards = json.loads(list\_cards)

    cards\_chosen = json.loads(cards\_chosen)

    choice\_ids = []

    # due to the fact that it only returns the location of the cards chosen this loops through them and appends the card ids to a list

    for index in cards\_chosen:

        choice\_ids.append(list\_cards[index - 1])

    # loops through the answer card ids and removes them from the list of cards the player has from the db

    for card\_id in choice\_ids:

        list\_cards.remove(card\_id)

    cur.execute(f"UPDATE Players SET ListCards = '{list\_cards}' WHERE id = {player\_id}")

    con.commit()

@app.post("/deal\_qcard/{game\_id}")

async def deal\_qcard(game\_id: int):

    # changes the question card to a new one

    keep\_going = True

    # gets number of question cards in the db

    max\_qcard = await num\_question\_cards()

    last\_qcard = cur.execute(f"SELECT QCard FROM Games WHERE id = {game\_id}").fetchone()[0]

    # loops indefently until the new question card is not the same as the old question card

    while keep\_going:

        # gets random number

        rand\_int = random.randint(1, max\_qcard)

        # checks if the random number is the same number as the old question card id

        if rand\_int != last\_qcard:

            keep\_going = False

    # gets the question card from the db with the id gotten from the random number

    cur.execute(f"UPDATE Games SET QCard = '{rand\_int}' WHERE id = {game\_id}")

    q\_card = await question\_card(rand\_int)

    return q\_card["text"], q\_card["num\_gaps"]

@app.get("/check\_game\_finish/{game\_id}/{points\_to\_win}")

async def check\_game\_finish(game\_id: int, points\_to\_win: int):

    # checks if the game has finished

    # returns a true or false statement depending on whether the game has ended or not, as well as, the id of the winner if there was a winner

    winner\_id = 0

    game\_ended = False

    list\_players = cur.execute(f"SELECT ListPlayerID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players = json.loads(list\_players)

    #loops through all the players and checks if they have enoough points to win

    for player in list\_players:

        points = cur.execute(f"SELECT points FROM Players WHERE id = {player}").fetchone()[0]

        #checks to see whether the player has won the game

        if points == points\_to\_win:

            game\_ended = True

            winner\_id = player

    #if any player has enough points to win it returns a true statament and if no one does it returns a false statement

    return game\_ended, winner\_id

@app.get("/num\_points/{player\_id}")

async def num\_points(player\_id: int):

    # returns the number of points the user with id player\_id has

    # this function is going to be used in the frontend to check if the player has won the game

    points = cur.execute(f"SELECT points FROM Players WHERE id = {player\_id}").fetchone()[0]

    # returns the points the user has

    return points

@app.get("/num\_answer\_cards")

async def num\_answer\_cards():

    # returns the maximum number of answer cards in the db

    # gets the max num in the form of a list

    count = cur.execute("SELECT COUNT() FROM AnswerCards").fetchone()

    # returns the value of the list

    return count[0]

@app.get("/num\_question\_cards")

async def num\_question\_cards():

    # returns the maximum number of question cards in the db

    # gets the max num in the form of a list

    count = cur.execute("SELECT COUNT() FROM QuestionCards").fetchone()

    # returns the value of the list

    return count[0]

@app.get("/czar/{game\_id}/{player\_id}")

async def czar(game\_id: int, player\_id: int):

    # checks if the player is the current czar

    czar\_id = cur.execute(f"SELECT CzarID FROM Games WHERE id = {game\_id}").fetchone()[0]

    if czar\_id == player\_id:

        Czar = True

    else:

        Czar = False

    return Czar

@app.post("/create\_player")

async def create\_player():

    # creates a new player in the db

    cur.execute("INSERT INTO Players (ListCards, points) VALUES ('[]', 0)")

    con.commit()

    return cur.lastrowid

@app.post("/return\_cards/{player\_id}")

async def return\_cards(player\_id: int):

    # returns the answer cards in the players hand from the db

    PlayersCards = cur.execute(f"SELECT ListCards FROM Players WHERE id = {player\_id}").fetchone()

    return PlayersCards[0]

@app.post("/deal\_cards/{player\_id}")

async def deal\_cards(player\_id: int):

    # deals the needed number of cards to get the players hand to 7 cards

    max\_cards = 7

    num\_players = []

    number\_cards = 0

    temp\_list = []

    player\_id = int(player\_id)

    # gets the maximum number of answer cards

    max\_a\_cards = await num\_answer\_cards()

    # loops through all the players in the db and appends their ids to a list

    for index in cur.execute("SELECT id FROM Players"):

        num\_players.append(index[0])

    unused\_cards = []

    # loops through all the answer cards and makes a list of all the possible answer cards that have not been used

    for index in range(max\_a\_cards):

        unused\_cards.append(index + 1)

    # loops through the player ids and removes the answer cards taht are in the player's hand

    for ids in num\_players:

        list\_cards = cur.execute(f"SELECT ListCards FROM Players WHERE id = {ids}").fetchone()[0]

        list\_cards = json.loads(list\_cards)

        # loops through all the cards in the player with player id ids and removes the answer cards in their hand from the list

        for cards in list\_cards:

            unused\_cards.remove(cards)

            # checks to see if the current id is the players id and counts how many answer cards they currently have in their hand

            if (ids) == player\_id:

                number\_cards += 1

    list\_cards = cur.execute(f"SELECT ListCards FROM Players WHERE id = {player\_id}").fetchone()[0]

    list\_cards = json.loads(list\_cards)

    # loops until the player has the 7 cards in their hand and adds a random card into their hand

    for index in range(max\_cards - number\_cards):

        random\_num = random.randint(1, len(unused\_cards))

        # adds the random answer card id to the players hand

        list\_cards.append(unused\_cards[random\_num])

        # removes the chosen answer card from the list of unused answer cards

        unused\_cards.pop(random\_num)

    cur.execute(f"UPDATE Players SET ListCards = '{list\_cards}' WHERE id = {player\_id}")

    # loops through all the answer card ids and turns them into the text equivilant to return it to the frontend

    for card\_id in list\_cards:

        acard = await answer\_card(card\_id)

        acard = acard["text"]

        temp\_list.append(acard)

    list\_cards = temp\_list

    con.commit()

    # returns the list of answer card's text in the players hand

    return list\_cards

@app.post("/create\_game")

async def create\_game():

    # creates a game in the db

    cur.execute(f"INSERT INTO Games (ListPlayerID, EveryoneIn) VALUES ('[]', False)")

    game\_id = cur.lastrowid

    # gets a new question card and saves in the db

    q\_card = await deal\_qcard(game\_id)

    con.commit()

    return game\_id, q\_card

@app.post("/everyone\_in/{game\_id}")

async def everyone\_in(game\_id: int):

    # sets the property EveryoneIn in the db to true

    cur.execute(f"UPDATE Games SET EveryoneIn = True WHERE id = {game\_id}")

    con.commit()

@app.get("/check\_EveryoneIn/{game\_id}")

async def check\_EveryoneIn(game\_id: int):

    # checks if the property EveryoneIn in the db is true or false

    everyone\_in = cur.execute(f"SELECT EveryoneIn FROM Games WHERE id = {game\_id}").fetchone()[0]

    # booleans in a db are stored as 1 (true) or a 0 (false)

    if everyone\_in == 0:

        everyone\_in = False

    else:

        everyone\_in = True

    return everyone\_in

@app.post("/join\_game/{game\_id}/{player\_id}")

async def join\_game(player\_id: int, game\_id: int):

    # saves the players id into the list of players in the db

    # checks if there is no game in the db and if there isnt it creates a new one

    try:

        games = cur.execute(f"SELECT \* FROM Games").fetchone()[0]

    except:

        game\_id = await create\_game()

        game\_id = game\_id[0]

    # saves the player's id into the list of players in the db

    list\_players = cur.execute(f"SELECT ListPlayerID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players = json.loads(list\_players)

    list\_players.append(player\_id)

    cur.execute(f"UPDATE Games SET ListPlayerID = '{list\_players}' WHERE id = {game\_id}")

    cur.execute(f"UPDATE Games SET CzarID = {player\_id} WHERE id = {game\_id}")

    con.commit()

@app.post("/save\_choice/{player\_id}/{choice}")

async def save\_choice(player\_id: int, choice: str):

    # saves the list of answer cards chosen by the player in the db

    choice = json.loads(choice)

    player\_cards = cur.execute(f"SELECT ListCards FROM Players WHERE id = {player\_id}").fetchone()[0]

    player\_cards = json.loads(player\_cards)

    choice\_ids = []

    # loops thorugh the choices and gets their respective ids and saves it to a list

    for index in choice:

        choice\_ids.append(player\_cards[index - 1])

    # saves the list in the db

    cur.execute(f"UPDATE Players SET choice = '{choice\_ids}' WHERE id = {player\_id}")

    con.commit()

@app.post("/delete\_game/{game\_id}")

async def delete\_game(game\_id: int):

    # deletes the game with id game\_id from the db

    cur.execute(f"DELETE FROM Games WHERE id = {game\_id}")

    con.commit()

@app.post("/delete\_player/{player\_id}")

async def delete\_player(player\_id: int):

    # deletes the player with id player\_id from the db

    cur.execute(f"DELETE FROM Players WHERE id = {player\_id}")

    con.commit()

@app.get("/return\_num\_players/{game\_id}")

async def return\_num\_players(game\_id: int):

    # returns the number of players in the game

    list\_player\_id = cur.execute(f"SELECT ListPlayerID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_player\_id = json.loads(list\_player\_id)

    return len(list\_player\_id)

@app.get("/return\_qcard/{game\_id}")

async def return\_qcard(game\_id: int):

    # returns the question card from the db

    q\_card\_id = cur.execute(f"SELECT QCard FROM Games WHERE id = {game\_id}").fetchone()[0]

    q\_card = await question\_card(q\_card\_id)

    num\_gaps = q\_card["num\_gaps"]

    q\_card = q\_card["text"]

    return q\_card, num\_gaps

@app.get("/return\_choices/{game\_id}")

async def return\_choices(game\_id: int):

    # checks if all the players have submitted their answer cards and if they have it returns those choices

    choice\_in = True

    list\_choices = []

    list\_players = cur.execute(f"SELECT ListPlayerID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players = json.loads(list\_players)

    czar\_id = cur.execute(f"SELECT CzarID FROM Games WHERE id = {game\_id}").fetchone()[0]

    list\_players.remove(czar\_id)

    # loops through all the players and gets their answer cards

    for player\_id in list\_players:

        choice = cur.execute(f"SELECT choice FROM Players WHERE id = {player\_id}").fetchone()[0]

        # tries to save the answer cards into a list and if it doesnt work it means that the player has not submitted their answer card

        try:

            choice = json.loads(choice)

            list\_choices.append((player\_id, choice))

        except:

            choice\_in = False

            break

    # returns a false statement if the player has not submitted their answer card yet else it returns a true statement and the cards

    return choice\_in, list\_choices

@app.post("/give\_points/{czar\_id}/{point\_winner}")

async def give\_points(czar\_id: int, point\_winner: int):

    # gives a point to the player who was chosen by the czar

    cur.execute(f"UPDATE Players SET choice = '{point\_winner}' WHERE id = {czar\_id}")

    points = cur.execute(f"SELECT points FROM Players WHERE id = {point\_winner}").fetchone()[0]

    points += 1

    cur.execute(f"UPDATE Players SET points = '{points}' WHERE id = {point\_winner}")

    con.commit()

    return points

@app.get("/return\_czar\_choice/{game\_id}")

async def return\_czar\_choice(game\_id: int):

    # checks if the player has submitted their choice and if they have it returns it

    choice\_in = True

    czar\_id = cur.execute(f"SELECT CzarID FROM Games WHERE id = {game\_id}").fetchone()[0]

    czar\_choice = cur.execute(f"SELECT choice FROM Players WHERE id = {czar\_id}").fetchone()[0]

    # checks if the czar has chosen a set of cards

    # the empty choices can either be (NULL) or None

    if czar\_choice == None or czar\_choice == "(NULL)":

        choice\_in = False

    return choice\_in, czar\_choice

@app.get("/")

async def root():

    #returns the front end code so that it can be run

    return HTMLResponse(open("CAH.html", "rt").read())

## Frontend

<!DOCTYPE html>

<html>

<head>

    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

    <title>Cards Against Humanity</title>

</head>

<body>

    <div id="create-player">

        <input type="text" placeholder="Name" id="name" />

        <button onClick="CreatePlayer();">Create player</button>

    </div>

    <div id="join-game" style="display: none;">

        <span id="welcome"></span>

        <button onClick="JoinGame(gameId, playerId);">Join game</button>

    </div>

    <div id="everyone-in" style="display: none;">

        <button onClick="EveryoneIn();">everyone in</button>

    </div>

    <div id="question" style="display: none;">

        <span id="question-card"></span>

    </div>

    <div id="cannot-join" style="display: none;">

        <span>The game has already started so you cannot enter right now (maybe try later)</span>

    </div>

    <div id="game-finished" style="display: none;">

        <span>The game has ended</span>

    </div>

    <div id="winner" style="display: none;">

        <span>Congratulations, you are the winner</span>

    </div>

    <div id="loser" style="display: none;">

        <span>Unfortunately, you are not the winner (Better luck next time -/(\*-\*)\- )</span>

    </div>

    <select id="answer" style="display: none;">

        <option id="ans1" value=1></option>

        <option id="ans2" value=2></option>

        <option id="ans3" value=3></option>

        <option id="ans4" value=4></option>

        <option id="ans5" value=5></option>

        <option id="ans6" value=6></option>

        <option id="ans7" value=7></option>

    </select>

    <select id="czar-choice" style="display: none;">

    </select>

    <div id="submit-answer" style="display: none;">

        <button onClick="SubmitAnswer(playerId);">Submit answer</button>

    </div>

    <div id="submit-choice" style="display: none;">

        <button onClick="SubmitChoice(playerId);">Submit choice</button>

    </div>

    <div id="answer-submited" style="display: none;">

        <span>Answer submitted</span>

    </div>

    <div id="choice-submited" style="display: none;">

        <span>Choice submitted</span>

    </div>

    <script>

        // decleares global varibales

        var name = "";

        var playerId = null;

        var gameId = 1;

        var numGaps = 0;

        var gapCountdown = 0;

        var listAns = "[";

        var numPointsToWin = 2;

        async function CreatePlayer() {

            // creates a new player in the databse and saves the player's id

            response = await fetch("/create\_player", { method: "POST" });

            json = await response.json();

            playerId = JSON.parse(json);

            name = $("#name").val();

            // displayes a welcome message and the button needed to continue with the program, and hides the button needed to create a player

            $("#welcome").html("Welcome " + name);

            $("#create-player").css("display", "none");

            $("#join-game").css("display", "inline");

        }

        async function JoinGame(game\_id, player\_id) {

            // saves the player's id in the list of players in the database

            // checks if anyone has already started the game before the player was able to join the game and if they have it stops them from joining

            everyoneInPressed = await CheckEveryoneIn();

            if (everyoneInPressed == false) {

                await fetch("/join\_game/" + game\_id + "/" + player\_id, { method: "POST" });

                $("#join-game").css("display", "none");

                // displayes the everyone in button

                $("#everyone-in").css("display", "inline");

                // checks if any player has pressed the everyone in button

                while (everyoneInPressed == false) {

                    // checks every 1 second to see if any player have pressed everyone in button

                    // loops indefently until a player has pressed the button

                    await timeout(1000);

                    playerChoices = await CheckEveryoneIn();

                }

                $("#everyone-in").css("display", "none");

                $("#question").css("display", "inline");

                // gets the current question card

                response = await fetch("/return\_qcard/" + gameId);

                json = await response.json();

                qcard = json;

                numGaps = qcard[1];

                gapCountdown = numGaps;

                // displays the question card

                $("#question").html(qcard[0]);

                response = await fetch("/czar/" + gameId + "/" + player\_id);

                json = await response.json();

                czar = JSON.parse(json);

                // checks if the player is the czar or not

                if (czar == false) {

                    // if the player is not the czar the player is dealt some cards

                    $("#answer").css("display", "inline");

                    $("#submit-answer").css("display", "inline");

                    response = await fetch("/deal\_cards/" + player\_id, { method: "POST" });

                    json = await response.json();

                    cards = json;

                    for (let i = 0; i < 7; i++) {

                        // loops through each answer card and populates the dropdown box with the answer cards

                        $("#ans" + (i + 1)).html(cards[i]);

                    }

                } else {

                    // sets the player choices to be false so that the code goes into the first while loop

                    playerChoices = [false]

                    while (playerChoices[0] == false) {

                        // checks every 1 second to see if all the player have submitted their answer cards

                        // loops indefently until all the players have submitted their answer card

                        await timeout(1000);

                        playerChoices = await CheckPlayerChoices();

                    }

                    // calls the function CreateDropdownCzar to create the dropdown box

                    CreateDropdownCzar(playerChoices[1])

                    $("#submit-choice").css("display", "inline");

                    $("#czar-choice").css("display", "inline");

                }

            } else {

                $("#join-game").css("display", "none");

                // displays a text message infroming the user that they can not join the game

                $("#cannot-join").css("display", "inline");

            }

        }

        async function NextRoundStart(game\_id, player\_id) {

            // after the first round is ran the rest of the rounds will be carried out here

            gapCountdown = numGaps;

            response = await fetch("/czar/" + gameId + "/" + player\_id);

            json = await response.json();

            czar = JSON.parse(json);

            // checks if the player is the czar or not

            if (czar == false) {

                // if the player is not the czar the player is dealt some cards

                $("#answer").css("display", "inline");

                $("#submit-answer").css("display", "inline");

                response = await fetch("/deal\_cards/" + player\_id, { method: "POST" });

                json = await response.json();

                cards = json;

                for (let i = 0; i < 7; i++) {

                    // loops through each answer card and populates the dropdown box with the answer cards

                    $("#ans" + (i + 1)).html(cards[i]);

                }

            } else {

                // sets the player choices to be false so that the code goes into the first while loop

                playerChoices = [false]

                while (playerChoices[0] == false) {

                    // checks every 1 second to see if all the player have submitted their answer cards

                    // loops indefently until all the players have submitted their answer card

                    await timeout(1000);

                    playerChoices = await CheckPlayerChoices();

                }

                // calls the function CreateDropdownCzar to create the dropdown box

                CreateDropdownCzar(playerChoices[1])

                $("#submit-choice").css("display", "inline");

                $("#czar-choice").css("display", "inline");

            }

        }

        async function CreateDropdownCzar(choices) {

            // creates the dropdown box for the czar

            playerChoice = ""

            $("#czar-choice").find("option").remove(); // removes previous dropdwon box

            for (var choice of choices) { // loops through all the choices

                for (var card of choice[1]) { // loops through all the cards in thee choice

                    // gets the answer card with the id card from the backend

                    response = await fetch("/answer\_card/" + card);

                    json = await response.json();

                    ans = json;

                    playerChoice = playerChoice + ans["text"] + " + "

                }

                // creates an option in the dropdwon box with the answer cards submitted from one player, each card separated by a +

                var option = new Option(playerChoice, choice[0]);

                $("#czar-choice").append(option);

                playerChoice = ""

            }

        }

        async function SubmitAnswer(player\_id) {

            // submits the answer card chosen by the player

            // checks if the remaining number of cards needed to submit is more than one

            if (gapCountdown != 1) {

                choice = $("#answer").val();

                // takes the answer card submitted and creates a string that will end up in the format of a list

                listAns = listAns + choice + ",";

                // reduces the number of cards needed to submit by one

                gapCountdown -= 1;

                $("#answer-submited").css("display", "inline");

                // informs the user they have submitted their first answer card using a text message

                await timeout(1000);

                // hides the text message

                $("#answer-submited").css("display", "none");

            } else {

                // sets the czar's choice to be false so that the code goes into the first while loop further on in the code

                czarChoice = [false]

                // takes the answer card submitted and saves it into a string in the format of a list

                choice = $("#answer").val();

                listAns = listAns + choice + "]";

                // saves the answer cards chosen in the backend

                await fetch("/save\_choice/" + player\_id + "/" + listAns, { method: "POST" });

                // removes the answer cards chosen from the list of answer cards the player has in the backend

                await fetch("/remove\_chosen\_cards/" + player\_id + "/" + listAns, { method: "POST" });

                // resets the listAns for later use

                listAns = "[";

                $("#question").css("display", "none");

                $("#answer").css("display", "none");

                $("#submit-answer").css("display", "none");

                $("#answer-submited").css("display", "inline");

                while (czarChoice[0] == false) {

                    // checks every 1 second to see if the czar has submitted their choice

                    // loops indefently until all the czar has submitted their choice

                    await timeout(1000);

                    czarChoice = await CheckCzarChoice();

                }

                // TODO: display the points of each player

                // checks if the player was the one who won the point or not

                if (czarChoice[1] == player\_id) {

                    $("#answer-submited").html("Congratulations, you won the round!!!");

                } else {

                    $("#answer-submited").html("Unfortunately, you didnt win the round");

                }

                await timeout(5000);

                $("#answer-submited").css("display", "none");

                $("#answer-submited").html("Answer Submitted");

                // after 5 seconds checks if any player has gotten enough points to win the game

                response = await fetch("/check\_game\_finish/" + gameId + "/" + numPointsToWin);

                json = await response.json();

                game\_finished = json;

                // chekcs if the game finished

                if (game\_finished[0] == true) {

                    winnerID = game\_finished[1]

                    // checks if the player is the winner

                    if (winnerID == player\_id) {

                        // displays a text message telling the player they won the game

                        $("#game-finished").css("display", "inline");

                        $("#winner").css("display", "inline");

                    } else {

                        // displays a text message telling the player they did not win the game

                        $("#game-finished").css("display", "inline");

                        $("#loser").css("display", "inline");

                    }

                } else {

                    // starts the next round

                    $("#answer-submited").css("display", "none");

                    $("#answer-submited").html("Answer submitted");

                    $("#question").css("display", "inline");

                    // finds the new question card

                    response = await fetch("/return\_qcard/" + gameId);

                    json = await response.json();

                    qcard = json;

                    numGaps = qcard[1];

                    $("#question").html(qcard[0]);

                    // calls the function NextRoundStart to star the next round

                    NextRoundStart(gameId, player\_id)

                }

            }

        }

        // https://stackoverflow.com/questions/33289726/combination-of-async-function-await-settimeout

        function timeout(ms) {

            return new Promise(resolve => setTimeout(resolve, ms));

        }

        async function CheckEveryoneIn() {

            // checks if any player has pressed the everyone is in button

            // due to the fact that if a game is not already created this would crash, it does a try catch function to strop this from crashing

            try {

                response = await fetch("/check\_EveryoneIn/" + gameId);

                json = await response.json();

                everyoneInPressed = json;

            } catch {

                everyoneInPressed = false

            }

            return everyoneInPressed;

        }

        async function CheckCzarChoice() {

            // checks if the Czar has submitted thei choice

            response = await fetch("/return\_czar\_choice/" + gameId);

            json = await response.json();

            choice = json;

            return choice;

        }

        async function CheckPlayerChoices() {

            // checks if all the players have submitted their answer cards

            response = await fetch("/return\_choices/" + gameId);

            json = await response.json();

            choice = json;

            return choice;

        }

        async function SubmitChoice(player\_id) {

            // gets the player's id of the player that submitted the set of answer cards chosen by the czar

            pointWinner = $("#czar-choice").val();

            // gives a point to the winner of the round

            await fetch("/give\_points/" + player\_id + "/" + pointWinner, { method: "POST" });

            // displays a text message informing the player they submitted their choice successfully

            $("#choice-submited").css("display", "inline");

            $("#submit-choice").css("display", "none");

            $("#czar-choice").css("display", "none");

            $("#question").css("display", "none");

            await timeout(5000);

            // after 5 seconds gets the number of points the player who won the round has

            $("#choice-submited").css("display", "none");

            response = await fetch("/num\_points/" + pointWinner);

            json = await response.json();

            points = JSON.parse(json);

            // checks if the winner of the round has enough points to win the game

            if (points == numPointsToWin) {

                // displays a text message informing the player they did not win the game

                $("#game-finished").css("display", "inline");

                $("#loser").css("display", "inline");

            } else {

                // deals a new question card

                response = await fetch("/deal\_qcard/" + gameId, { method: "POST" });

                json = await response.json();

                qcard = json;

                numGaps = qcard[1]

                $("#question").css("display", "inline");

                $("#question").html(qcard[0]);

                // changes the current czar to a different one and resets all the choices submitted by the players

                await fetch("/reset\_choices/" + gameId, { method: "POST" });

                await fetch("/change\_czar/" + gameId, { method: "POST" });

                // calls the function NextRoundStart to start the next round

                NextRoundStart(gameId, player\_id)

            }

        }

        async function EveryoneIn() {

            // changes the propery EveryoneIn in the Game in the db so that the game can start

            await fetch("/everyone\_in/" + gameId, { method: "POST" });

        }

    </script>

</body>

</html>

## Database Creation

Graphical user interface, application, timeline

Description automatically generated

Graphical user interface, text, application

Description automatically generated Text

Description automatically generated Graphical user interface, text, application

Description automatically generated Text

Description automatically generated Graphical user interface, text, application, email

Description automatically generated Graphical user interface, text, application, email

Description automatically generated Text

Description automatically generated Text

Description automatically generated

# Testing

Since the program is a website some of the testing of the code will be done on a simple web user interface provided by API that mocks the get and post requests to allow the programmer to check specific functions

|  |  |  |
| --- | --- | --- |
| Test | Expectation | Result |
| 1) Creating a player with a string of characters for the name | The player will be welcomed by the program with their name and a Player will be created in the backend | Test was successful |
| 2) Creating a player with a string of special characters as a name | The player will be welcomed by the program with the special characters as their name and a Player will be created in the backend | Test was successful |
| 3) Creating a player with no name | The user will be given an error message and prompted to input a valid name and no player will be created in the backend | Test was unsuccessful  There is no error message displayed and the program just carries on with the rest of the program with an empty string for a name |
| 4) Join a Game | The user’s id will be appended to the list of players in the database and no error message will be outputted | Test was successful |
| 5) Submit an answer card | The user’s choice will be saved in the backend and there will beno error message outputted | Test was successful |
| 6) try to submit more than one answer card for a question card that only has one gap | The user will not be allowed to do this as when the first answer card is submitted the dropdown box and the button will both disappear | Test was successful |
| 7) Try to submit only one answer card for a question card that has more than one gap | The user will not be allowed to continue with the rest of the game and will just be stuck with the interface for submitting an answer card | Test was successful |
| 8) choose a set of answer cards if the user is a Czar | The user’s choice is saved in the database and the player that submitted the answer card chosen by the user is given a point | Test was successful |
| 9) try to choose more than one answer card as the Czar | The user will not be allowed to do this as the dropdown box and the button for choosing the answer cards will disappear after choosing one set of answer cards. The first set of answer cards chosen will be stored in the backend | Test was successful |
| 10) The user is not able to choose any set of answer cards if they are the Czar until all the players have submitted their answer cards | The dropdown box displaying the cards submitted by all the players and the button allowing the player to choose the set of answer cards will not appear until all the players have submitted their answer cards | Test was successful |
| 11) Ensure the user is only given 7 answer cards | The user will only be allocated and displayed 7 answer cards | Test was successful |
| 12) Ensure that all the data is stored in the database | All the data will be stored in the database | Test was successful |
| 13) Ensure the player is told if they won the round or not | The user will be shown a text message telling them whether they won the round or not | Test was successful |
| 14) Ensure the chosen answer cards are removed from the | The database will not have the answer card ids that have been chosen by the user | Test was successful |
| 15) Ensure that the player is given a point when they win a game | The player who wins the point will be allocated a point in the backend | Test was successful |
| 16) Ensure that if the player has the needed number of points to win the round the game ends and a message is displayed to all the players | A message is displayed to all the players and the game is stopped | Test was successful |
|  |  |  |
|  |  |  |

Test 1:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

Success

Test 2:

Graphical user interface, text, application, chat or text message

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

Success

Test 3:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

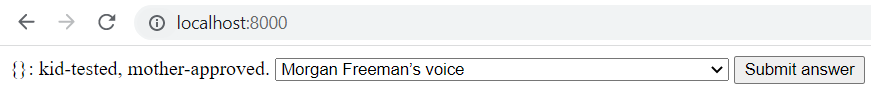
unsuccessful

This was not I intended for the program to do, what I wanted it to do was to display an error message and prevent the user from continuing until they imputed a suitable name

Test 4:

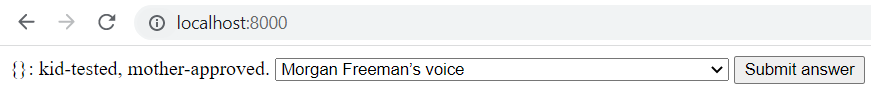
Graphical user interface, text, application, chat or text message

Description automatically generated



success

Test 5:



Graphical user interface, text, application, chat or text message

Description automatically generatedsuccess

Test 6:

It is not possible to show this via pictures, but the user is taken to another interface that does not allow the user to choose a new answer card

Test 7:

Graphical user interface, application, Word

Description automatically generated

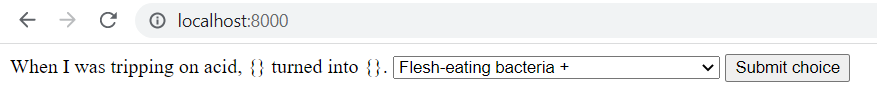
Graphical user interface, text, application

Description automatically generated

Success

The user is not allowed continue with the rest of the game until the second card has been submitted

Test 8:



Graphical user interface, text, application

Description automatically generated

Success

Test 9:

It is not possible to show this via pictures, but the user is taken to another interface that does not allow the Czar to choose a new set of answer cards

Test 10:

Graphical user interface, text, application, chat or text message

Description automatically generatedsuccess

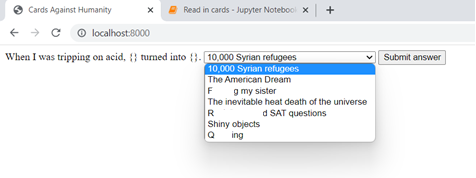
The user is not displayed the button and dropdown box until all the players have submitted their answer cards

Test 11:

Text

Description automatically generated

As shown only seven answer cards were allocated to the player



Here you can see that there are only 7 answer cards displayed to the user

Test 12:

Text

Description automatically generated

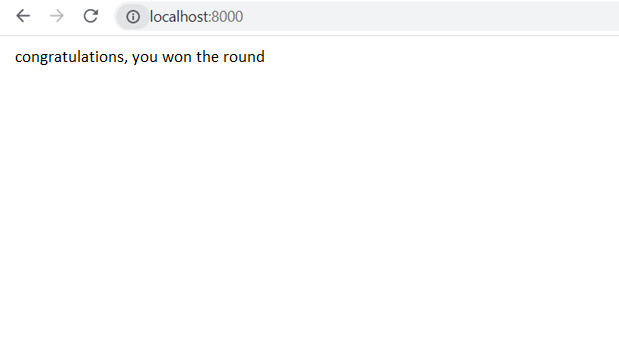
Here you can see the players answer cards, choice and number of points are stored in the database

Graphical user interface, text, application

Description automatically generated

Here you can see that the list of players, the current question card and the current Czar’s id are stored in the database

Test 13:



If the Czar chooses the answer card submitted by the user the screen is displayed above is displayed and if the user is not the one who submitted the answer card chosen by the Czar the screen below is displayed

A picture containing graphical user interface

Description automatically generated

Test 14:

Must show this

Test 15:

Text

Description automatically generated

Text

Description automatically generated

This happened only when the player’s submitted answer card was chosen by the Czar and it shows what happens in the backend

Test 16:

Must show this

# Evaluation

**My opinion:**

I feel like my program could have been greatly improved to make it more user friendly and appealing to the customer, however, it carries out the task at hand with a certain degree of speed and accuracy. I wanted a program that allowed the user to play cards against humanity via the internet with various players at the same time, and I managed to achieve this, however, the cards are not shown in a very eye-catching manner and the user cannot communicate with other players via the website. Overall, I feel like if I had more time I would have been able to greatly improve my program and make it more user friendly.

**Success in meeting my objectives:**

**PHASE 1**

|  |
| --- |
| 1. Download cards for game from official website MET |
| * 1. Extract question and answer cards MET |
| * 1. Process answer and question cards MET |
| * + 1. Count number of blanks for the question cards and replace them with an identifiable string (e.g., “{}”) MET |
| * + 1. Strip unnecessary whitespace and unwanted characters MET |
| * 1. Store cards in database MET |
| * + 1. Create two different tables: one for question cards and the other for answer cards MET |
| * + 1. Ensure each card has a unique identifier MET |
| 1. Allow user to play CAH (cards against humanities) |
| * 1. Allow user to create a player MET |
| * 1. Create a game MET |
| * 1. Allow user to join a game MET |
| * 1. Allocate random answer cards to user MET |
| * + 1. Ensure only a maximum of 7 cards are allocated to user MET |
| * + 1. Ensure each answer card is different MET |
| * + 1. Ensure no more than one player has the same answer card at one time MET |
| * 1. Display player’s answer cards MET |
| * 1. Allocate one question card per round MET |
| * + 1. Ensure every player can see the question card MET |
| * + 1. Ensure that every player sees the same question card MET |
| * 1. Display question card MET |
| * 1. Allocate a player as the Czar MET |
| * + 1. Ensure that every player has been the Czar once before the user is allocated the Czar a second time MEET |
| * 1. Allow every player to choose an answer card MET |
| * + 1. Ensure that the Czar is not able to choose an answer card MET |
| * + 1. Ensure that the user must submit the same number of answer cards as the number of gaps in the question card MET |
| * + 1. Ensure that users cannot submit the same answer card more than once NOT YET |
| * 1. Remove chosen answer cards from player’s hand MET |
| * 1. Show every player’s submitted answer cards MET |
| * + 1. Ensure every player’s answer cards are displayed once MET |
| * 1. Allow Czar to choose a player’s answer card MET |
| * + 1. Ensure the Czar can only choose an answer card when all the players have submitted their answer cards MET |
| * + 1. Ensure the Czar can only choose one player’s answer cards MET |
| * + 1. Ensure the Czar does not know from which player each answer card was submitted MET |
| * 1. Display winner of round NOT YET |
| * 1. Allocate a point to the player, whose answer card was chosen by the Czar MET |
| * + 1. Ensure only one point is allocated to the player MET |
| * + 1. Ensure that the answer card chosen by the Czar corresponds to the player being awarded a point MET |
| * 1. If the number of points the player who won the point has is the same as the number of points needed to win the game, end the game NOT YET |
| * 1. Start a new round NOT YET |
| * 1. Display the winner of the game NOT YET |
| 1. Allow users from different computers to access and play the same game together MET |
| * 1. Ensure games can be resumed and that no data is lost if the computer crashes. Store game state in a database MET |
| * + 1. Create table for players MET |
| * + - 1. Store which answer cards they have in their hand MET |
| * + - 1. Store their current choice MET |
| * + - 1. Store the number of points each player has MET |
| * + 1. Create table for the games currently in progress MET |
| * + - 1. Store the players in the game MET |
| * + - 1. Store the question card from the current round MET |
| * + - 1. Store the player that is the Czar of the current round MET |
| * 1. Ensure that the user interface does not freeze when waiting for an event (for example another player to choose a card) MET |
| * 1. Coordinate between all the users centrally as users cannot communicate directly with each other as they are on different computers MET |

**PHASE 2**

|  |
| --- |
| 1. Create and implement a more attractive graphical design NOT MET |
| 1. Allow for more than one game to be played at once NOT MET |
| * 1. Allow users to create or join an existing game NOT MET |
| * 1. Allow users to invite other users NOT MET |
| * 1. Allow users to make the game private or public NOT MET |
| * 1. Allow users to configure the game (for example allow them to choose which version to play or let users choose how many points are need to win the game) NOT MET |
| * 1. Allow users to delete a game NOT MET |
| * 1. Display number of players and player ids in the game NOT MET |
| 1. Allow users to log in/sign up NOT MET |
| * 1. Allow user to delete account NOT MET |
| 1. Allow users to communicate with each other via a chat NOT MET |
| 1. Be able to cope with a larger amount of players using the website at the same time NOT MET |

**Quote user feedback**

**My analysis:**

The comments received from my program were not all positive but, overall, I am proud with what I created and have learnt a lot on how to create websites and on how to improve on future projects similar to this one. I will now give a more detailed overview of my opinion, stating the things I am most proud of and the section on what I would improve

**What I like most**

* **Ensuring that the user interface does not freeze when waiting for an event:** all programs I have coded previously have been written in a synchronous fashion, however, for the user interface to not freeze while waiting for an event this approach was not suitable. Due to this fact I had to research other ways of programming code in a way that would not freeze the UI when waiting for an event and came across asynchronous programming. I spent various hours of my time trying to decipher and get used to this way of programming, in the end, I managed to produce a working piece of code that would not freeze while waiting for an event. This is one of the things I am most proud of and was a great learning experience for me.
* **Allowing users from different devices to access and play together:** to do this I had to use a client server model, where the client was the frontend and the server was the backend. As I was not familiar with this style of programming I am proud of how it turned out and it resulted in a great learning experience as I also needed to research and spend various hours trying to familiarizes myself with this style.
* **Program being stateless:** to do this I had to store all the data that varied in the database so if the program were to crash there would be no data loss on either the backend or frontend. This wasn’t one of the hardest tasks I needed to complete as I had already had previous encounters with writing SQL commands and saving and extracting data from a database. However, this does not mean I am not proud of what I did as I had to overcome some problems concerning this, like how I would access the data from the frontend (I needed to access the data from the backend and then send it to the frontend).
* **Coding the frontend in JavaScript:** Even though the rest of the program is written in python I am not that proud of coding it in that as I already had prior knowledge and experience in coding in python, however, I have never coded in JavaScript before. To complete this piece of code it took me a while to get used to programming in JavaScript and searching on the internet on how to code some of the basics of JavaScript. Overall, I think was a great learning experience as I was able to learn and branch out into sections of computer science that I had never experienced before.

**What I’d improve:**

There were many things I could have improved but here are some of the main improvements I would have done if I would have had more time for my project

* **User interface:** I think the main improvement I could have done would have been making the user interface more user friendly as right now the cards are written in dropdown boxes and the background is just a plain white background.
* **Improving the speed of the program:** some of the code is very inefficient, for example, when each event is completed the player must be sent to another interface to continue with the program, this can slow down the program as the user must wait until the website has finished loading until they can continue playing the game.
* **Telling the user who won the point:** due to the fact that the database does not store the names of each player the game cannot tell the other users who the round winner is as each player is differentiated using their ids and not their names, which would make it very hard for the players to distinguish who is who

# TODO List

TODO:

* Check dr smiths comments
* Maybe make a diagram 4 observations
* Describe asynchronous programming
* Quote user feedback
* Add comments
* Finish testing
* Paste the creation of database code into the technical solution
* If there are more than one gap in the question card show a text message for a couple of seconds telling the user they have submitted their first answer card
* Reset game at the end
* Deploy to Heroku
* Include to reference to Heroku in writeup
* Include reference to github in writeup
* Update writeup to include latest changes (flowchart, database scheme, evaluation objectives, take more screenshots of gameplay, more testing)
* Tidying up format
* Record and include video
* Say for improving is that if the users can have accounts then if the player leaves they could resume the game