|  |
| --- |
|  |
| Poker Web App |
|  |

|  |
| --- |
| Tom Pollak |

Contents

[Analysis 5](#_Toc36640969)

[Overview 5](#_Toc36640970)

[The Rules of Texas hold ’em 5](#_Toc36640971)

[Pre-Game 5](#_Toc36640972)

[Play 5](#_Toc36640973)

[Showdown 6](#_Toc36640974)

[Algorithms 6](#_Toc36640975)

[Supervisor 7](#_Toc36640976)

[Objectives 7](#_Toc36640977)

[Extension Objectives 8](#_Toc36640978)

[Design 9](#_Toc36640979)

[Overview 9](#_Toc36640980)

[The Django Framework 9](#_Toc36640981)

[Fundamentals of Django 9](#_Toc36640982)

[Django Channels 10](#_Toc36640983)

[User Interface Design 11](#_Toc36640984)

[Website wireframe 11](#_Toc36640985)

[Key 12](#_Toc36640986)

[The Poker Algorithm 12](#_Toc36640987)

[Joining the game 12](#_Toc36640988)

[Finding hand strengths 13](#_Toc36640989)

[Creating winner order queue 14](#_Toc36640990)

[Dividing the pot 15](#_Toc36640991)

[Making the winner message 15](#_Toc36640992)

[Database Design 16](#_Toc36640993)

[Table Design 17](#_Toc36640994)

[User Design 17](#_Toc36640995)

[OOP Design 17](#_Toc36640996)

[Implementation 18](#_Toc36640997)

[2D Array 18](#_Toc36640998)

[Anonymous function 18](#_Toc36640999)

[Strength of hand 18](#_Toc36641000)

[Finding cards of the same rank 18](#_Toc36641001)

[Finding flushes 19](#_Toc36641002)

[Finding straights 20](#_Toc36641003)

[Binary Sort 21](#_Toc36641004)

[List operations 22](#_Toc36641005)

[Queue 22](#_Toc36641006)

[Creating 22](#_Toc36641007)

[Using 23](#_Toc36641008)

[Recursive Function 23](#_Toc36641009)

[Create showdown winner message 24](#_Toc36641010)

[DB Objects 25](#_Toc36641011)

[Creating DB Objects 25](#_Toc36641012)

[Instantiating DB objects and records, getting DB objects 26](#_Toc36641013)

[Threading 26](#_Toc36641014)

[Creating thread 26](#_Toc36641015)

[Thread function 27](#_Toc36641016)

[Starting the game 27](#_Toc36641017)

[Adding and newly joined players and removing players that have left 28](#_Toc36641018)

[Synchronous web sockets 28](#_Toc36641019)

[Server-side 28](#_Toc36641020)

[Client-side JavaScript 31](#_Toc36641021)

[Regular Expressions and Inheritance, Overriding 33](#_Toc36641022)

[Active nav bar tag – regular expressions 33](#_Toc36641023)

[Exception handling and getting and setting private variables 33](#_Toc36641024)

[Private variables 33](#_Toc36641025)

[Exception handling 34](#_Toc36641026)

[Web requests 34](#_Toc36641027)

[Server-side scripting 34](#_Toc36641028)

[URL paths 35](#_Toc36641029)

[Handling form POST request and GET request 35](#_Toc36641030)

[Serializing DB object into JSON using RESTful API 36](#_Toc36641031)

[Serializing data 36](#_Toc36641032)

[Creating and using serialized data 37](#_Toc36641033)

[Reading from file 37](#_Toc36641034)

[Scheduling and custom commands 37](#_Toc36641035)

[Complex SQL – leaderboard view 38](#_Toc36641036)

[Testing 39](#_Toc36641037)

[Test videos 39](#_Toc36641038)

[Test video 39](#_Toc36641039)

[Normal use 39](#_Toc36641040)

[Test table 39](#_Toc36641041)

[Possible tests 41](#_Toc36641042)

[Poker algorithm 41](#_Toc36641043)

[Poker log 42](#_Toc36641044)

[Poker chat 42](#_Toc36641045)

[Website interface 43](#_Toc36641046)

[Evaluation 44](#_Toc36641047)

[Analysis of objectives 44](#_Toc36641048)

[Feedback 47](#_Toc36641049)

[Supervisor comments 47](#_Toc36641050)

[End user feedback 47](#_Toc36641051)

[Improvements 47](#_Toc36641052)

[Improvements based on feedback 47](#_Toc36641053)

[Other improvements 48](#_Toc36641054)

[Code 50](#_Toc36641055)

[Dictionary tree 50](#_Toc36641056)

[censored-word.txt 51](#_Toc36641057)

[db.sqlite3 51](#_Toc36641058)

[manage.py 51](#_Toc36641059)

[accounts 52](#_Toc36641060)

[forms.py 52](#_Toc36641061)

[models.py 52](#_Toc36641062)

[serializers.py 52](#_Toc36641063)

[urls.py 52](#_Toc36641064)

[views.py 53](#_Toc36641065)

[leaderboard 53](#_Toc36641066)

[urls.py 53](#_Toc36641067)

[views.py 53](#_Toc36641068)

[poker 54](#_Toc36641069)

[consumers.py 54](#_Toc36641070)

[models.py 57](#_Toc36641071)

[poker.py 57](#_Toc36641072)

[urls.py 74](#_Toc36641073)

[views.py 74](#_Toc36641074)

[management/ commands 75](#_Toc36641075)

[project 75](#_Toc36641076)

[routing.py 75](#_Toc36641077)

[settings.py 75](#_Toc36641078)

[urls.py 79](#_Toc36641079)

[rules 79](#_Toc36641080)

[urls.py 79](#_Toc36641081)

[views.py 79](#_Toc36641082)

[tables 79](#_Toc36641083)

[consumers.py 79](#_Toc36641084)

[forms.py 80](#_Toc36641085)

[models.py 81](#_Toc36641086)

[serializers.py 81](#_Toc36641087)

[urls.py 82](#_Toc36641088)

[views.py 82](#_Toc36641089)

[management/ commands 83](#_Toc36641090)

[templatetags 84](#_Toc36641091)

[templates 84](#_Toc36641092)

[base.html 84](#_Toc36641093)

[game.html 86](#_Toc36641094)

[how-to-play.html 89](#_Toc36641095)

[index.html 91](#_Toc36641096)

[leaderboard.html 92](#_Toc36641097)

[profile.html 93](#_Toc36641098)

[signup.html 93](#_Toc36641099)

[table-form.html 94](#_Toc36641100)

[registration 94](#_Toc36641101)

# Analysis

## Overview

I am creating a free to play Texas hold ’em poker web application that can be played from any web browser with JavaScript. Existing products I have seen are Appeak Poker, an online poker app for IOS and Android. In this app a user can join different tables to play poker, add friends and keep track of their stats as they play, along with a total amount of money they have earned throughout their game time.

## The Rules of Texas hold ’em

### https://upload.wikimedia.org/wikipedia/commons/thumb/c/c0/Texas_Hold%27em_Poker_Table_with_Blinds.svg/1920px-Texas_Hold%27em_Poker_Table_with_Blinds.svg.pngPre-Game

One player acts as dealer. The dealer is represented by the player with the dealer button, and moves clockwise each round.

The game starts with two forced bets by the two players to the dealers left. These are called the small blind (SB) and the big blind (BB).

The SB is equal to the minimum bet. The BB is twice that of the small blind. After the blinds, play moves into the betting phase.

https://en.wikipedia.org/wiki/Texas\_hold\_%27em

### Play

Following the blinds, each player receives two cards face down. These are your ‘hole’ cards. The play continues clockwise, where each player following the blinds can either:

* **Call** – Calling matches the highest bet, in this cast the BB. If the highest current bet is 0 and the players calls, it is called a ‘check’.
* **Raise** – A player can raise the bet up to the amount they have left on the table.
* **Fold** – By folding, a player is choosing not to continue betting and withdraws from the round.

The betting round continues until every player has put an equal amount of money in the pot or there is only one player left in the round. If a player calls or raises up to all of their money, they are said to have gone ‘all-in’, and a side-pot is created, which contains the pot which equates to the amount of chips the player has put in.

This means that the other players may continue to bet between themselves with the all-in player only eligible to win chips up to the amount he has put in.

Once the betting round has completed, three community cards are shown, known as ‘the flop’. Community cards can be used by any player to make the best 5 card hand possible.

Another betting round then commences where the players have an option to call (or ‘check’ if no bet has been made), raise (or bet if a bet has been made) or fold. When the second betting round is finished, a fourth card is shown, known as ‘the turn’.

Another betting round takes place, after which the final card is shown, known as ‘the river’ and after the final betting round if there are still two or more players left, the game goes to a showdown.

### Image result for texas holdem hand rankShowdown

https://snapguide.com/guides/play-poker/

If a player bets and all other players fold, the player is awarded the pot and is not required to show his cards. However, if there are still two or more players left after the final round of betting, a ‘showdown’ occurs, where the last player to bet is the first to show his hand, unless everyone checks in which case the player to the left of the dealer button is the first to show.

After the first player has shown his cards, the players must show their cards in a clockwise rotation if they can beat the best players hand to show. Otherwise they can choose not to show, and ‘muck’ their losing hand without showing them.

If two players have the same hand strength, ‘kickers’ can be used to break ties. These are extra cards that do not add to the rank of the hand.

For example, **K-K-10-3-2** would beat **K-K-9-8-7** even though they are both pairs of kings, as the **10** would beat the **9**. If the best hand is shared by two or more players, the pot is shared between them. If the pot cannot be equally distributed among all the players in the split pot, the odd chips are given precedence to the winners that are first to play.

## Algorithms

I am using the Django framework for my web application because it is one of the most documented and versatile web app frameworks for Python. It is an extremely stable and scalable framework used by many large companies such as Instagram and Pinterest. I can also develop new features to add to the web app quickly, and add custom programs such as a python poker algorithm relatively easily as it is extremely customisable and written in the same language. I am using sqlite3 to manage the database for its ease of use and that it is the default database that ships with Django. The website will be hosted on my computer on port 8000 and can be accessed by any computer on the same network.

## Supervisor

My supervisor is Archie McMullan who is a regular user of Appeak Poker, and so has some experience with online poker and products like this. Archie also has a good knowledge of Python so can understand the use cases and complexity of different objectives to give more informed feedback, for instance a poker ai algorithm would need to make decisions with only incomplete information, so to create an optimal poker ai is much harder than it sounds and is an area in current cutting edge research.

Archie suggested that the users could be shown around a table with their money, with the pot and cards in the middle of the table to make the game look more akin to real life poker. We also discussed getting card images of the internet, and show them pop up with animations instead of a text-based version of the game. Archie also suggested a friend feature on the app, in which you could see whether they are online and join their game. We thought that users could send friend requests to other users they are in a game with or by searching for their username on the friends page. He also suggested creating a leaderboard page so you could compete with other players with your total money.

My end users are people aged 16 and over who enjoy playing poker without having to commit any real money to the game.

## Objectives

1. Players will be able to register with a username and password, and log in/out. The users account information is then stored in the database
2. A player’s total money is stored in a database
   1. On entering a table, the buy in of the table is deducted from the user’s total money and is stored in a temporary table, in which the user can bet with
   2. The money in the table is updated during play
   3. Once a player leaves a table their money from the table is added to the player’s total money and the database is updated
3. User passwords are stored with the SHA-256 hash
4. Players will be able to create table, and the and join them through the home page where all current tables are displayed
5. If a table has not been used in 15 minutes, it is deleted
6. The pack in each table is shuffled between rounds
7. The dealer button moves to the dealers left each round
8. Tables have a maximum number of players (between 2 and 8) which can be specified by the table’s creator
9. Tables have a buy-in to enter, specified by the table’s creator which is used to determine how much a player must put in the table
   1. Used to calculate the size of the blinds, the small blind is 1/100th the size of the buy in.
10. Once in a table, players can play live poker according to the rules in the Texas hold ’em section in analysis
11. Players will only be able to see their pocket cards and the table’s community cards
12. If a player leaves a table, they will automatically fold
13. A leader board shows players ranked by their total money
14. Players can interact with each other using an in-table chat
    1. The chat can be filtered so that swear words would be censored
15. A how to play page is provided for new players
16. Multiple games on the website can be played at the same time

## Extension Objectives

1. Players can create private tables, which require a password to enter
2. Players can add friends and join the table they are in if possible
3. Players can view active friends on the friend’s page and join the table through the page
4. Players can add friends by visiting their profile through leaderboard or game
5. Temporary storage of data in poker games could be stored in a NO SQL database as it offers quicker read and write speeds
6. Players can keep track of other stats such as number of hands played, % hands won etc.
7. Players can join multiple tables at once, and switch between them
8. The table could be shown with players in a circle and a pot and community cards in the middle, rather than being text based
9. Cards shown with images instead of text
10. Card and chip animations
11. A re buy in button could be shown once a player runs out of money in a table
12. A poker AI could optionally be added to tables to add players to the game
13. Site uses HTTPS to prevent username and passwords being intercepted, along with poker hands

# Design

## Overview

I am designing a web-application that allows users to create accounts and play live poker against other players. In each table, there is also a chat allowing players to interact with each other. Users are able to view their ranking on the leaderboard.

I will create a CRUD like interface to begin with (to create and delete tables) and user accounts, so that the web app functions in the way it should before adding the poker aspect to the website. To make the poker game interactive, I will use sockets to achieve two-way communication, however logic must be run on the backend server and send values to the frontend service as any script or logic running on the client device could be easily manipulated by malicious users.

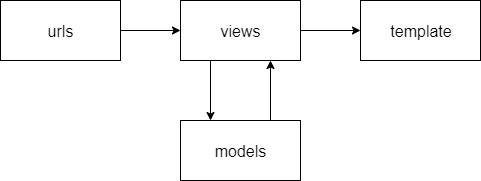
## The Django Framework

### Fundamentals of Django

The Django Framework is designed to encapsulate each aspect of the project in its own ‘app’.

Django is fundamentally made up of 4 different types of files:

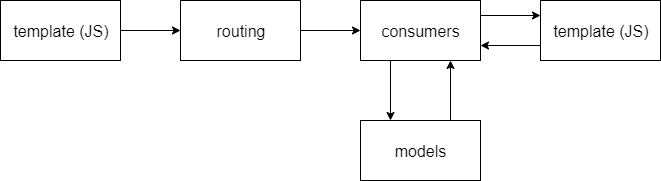
* models.py – Defines your data model and contains the fields and behaviours of the data you are storing. Each model maps to a database table and each attribute to a field. A model is defined in a class and as such this allows Django to interact with DB records like objects. This is extremely useful in creating queries and editing records as it does not require a manual SQL query. The model can include methods to manipulate the DB data.
* forms.py – Django can create forms that you can interact with like objects in a very similar form to the way it handles models. You can use a model to map out and save a form. For example, to create a poker table, the form can use fields from the table model instead of writing out the fields again. As the form is modelled directly off the table model, it can be validated and saved directly into the database.
* urls.py – Uses regex to capture URL patterns to retrieve a view. It can parse arguments in the URL to pass to the view e.g. a URL regex ‘tables/<int:pk>’ can be mapped to a view that takes pk as a parameter to fetch data from a specific table.
* views.py – Called by urls.py. Takes the web request and returns a web response. When rendering a web page, it can pass context (a dictionary of variables) to the template. The view function can do anything a standard python function can do, as long as it returns a web response.
* template – an HTML file with special syntax describing how dynamic content can be inserted into the HTML.



### Django Channels

A synchronous socket library for Django. Its base features work very similarly to Django:

* The templates’ JavaScript creates a web socket.
* routing.py – uses regex to capture the web socket retrieves a consumer (just like urls.py).
* consumer.py – A class similar to views. It can interact with the models and create web socket groups to send data to multiple users. The consumer has specific methods to process users connecting and disconnecting from the sockets, as well as sending and receiving data to and from JavaScript.



## User Interface Design

### Website wireframe

### Key

* Light blue box – object
* Dark blue box – button
* White box – input
* () – variables
* The leader board table are CSS buttons that link to each user’s profile

This is a wireframe of the web-app. The table page acts as a home page where users can view, create and join tables. A public leaderboard displays every user by their money, and users can view other players profiles by clicking on them through the leaderboard.

Users can also view their profile by clicking on their username in the nav bar. The fold, call and raise buttons can be hidden if it is not the user’s turn or if they cannot perform the action at that stage.

Users must be logged in and have sufficient money to join the table.

## The Poker Algorithm

### Joining the game

#### Game view

When the user sits down at a table, the program verifies whether they have sufficient money to play at the table, and that the table is not full. If verified, the user joins the game otherwise they are redirected back to the index page.

login required to access

Function game

table <- get Table object

if users money >= tables buy in and players in table < max players in table

start daemon thread on poker main function

return render of game.html

return a redirect to the index view

#### Adding player to table

When the poker algorithm runs, it determines whether they are the only player at the table, in which case a new poker instance is instantiated. Otherwise it adds the player to the table and returns.

Function main

get Room object for Table

add player to Room

if Room does not exist

create Room object

add player to Room

startGame()

### Finding hand strengths

#### Finding cards of the same rank

hand <- players cards sorted in descending order with Aces counting high and low

finalHand <- []

Function checkRank(hand)

sameRank <- []

i <- 0

while i < 6

temp <- hand[i]

while hand[i] has same rank as hand[i+1]

temp += hand[i+1]

i += 1

if length of temp > 1

sameRank += temp

sort sameRank by length of arrays

sameRank <- first two arrays of sameRank

if sameRank[0] has length of 4

sameRank <- sameRank[0]

hand is 4 of a kind

elif length of sameRank = 2

if length of sameRank[0] = 3

hand full house

else

hand is two pair

else

if length of sameRank[0] = 3

hand is 3 of a kind

else

hand is pair

put all cards in sameRank in 1D array

add all other loose cards to sameRank

finalHand <- first 5 cards in sameRank

#### Finding flushes

Function flush(hand)

for each suit

flush <- []

for each card in hand

if card is same suit as the suit that’s being compared

flush += card

if length of flush is 5 and flush is the highest strength found

hand is flush

finalHand <- flush

#### Finding straights

If cards are the same in a hand, the function is recursively called to check for opportunities for straight flushes.

Function straight(hand)

straightHand <- []

for each card in hand

if card is one more than next card

add the cards to straightHand

else if the card is the same as the next card

straight(hand without the card)

else

straightHand <- []

if there are 5 cards in straightHand

if cards are of same suit

finalHand <- straightHand

if straightHand begins with high Ace

hand is royal flush

else

hand is straight flush

else

if straight is the highest strength found

hand is straight

finalHand <- straightHand

### Creating winner order queue

The algorithm finds which players win over others by creating a queue of players in winning order. If two (or more) players have the exact same strength, they share the pot, and are added to playerWin in a single array.

#### Sorting the players

Sort the players by hand strength and add players of same strength to repeated array for further processing

win <- [players sorted by hand strength]

Function clash

repeated <- []

binary sort win

if win items are equal

repeated += both items

#### Grouping players of same strength in arrays

Many players may have the same strength hand and the players are added in pairs, so if a player is the same as a player in the previous iteration then all 3 players in the current and previous iteration have the same strength hand. So, the other player in the current iteration is appended to the previous iterations array.

split <- []

Function splitWork(repeated)

for player in repeated, step of 2

if player in previous split array

split[last item in array] += next player in repeated

else:

split += [player, next player]

#### Adding players to playerWin queue

Players are added to playerWin array in strength order, and if they appear in a split array, they share hand strength with other players so the split array is added instead. Single players are still added in their own arrays to keep the array depth consistent. As two or more players are in each split array the array will be added for each player. Therefore, any duplicate arrays after the first one in the queue are removed.

playerWin <- []

Function WinQueue

for player in win

if player in a split array

playerWin += split array the player is in

else

playerWin += [player]

remove duplicate arrays in playerWin

### Dividing the pot

#### Determining the winners

For some all-in scenarios some players in the game have not put the same amount of money as other players, so the algorithm iterates through the playerWin queue until the entire pot is given out.

Function winner

a <- 0

playerWin <- list of players sorted by hand strength

while pot != 0

winners <- []

for player in playerWin[a]

if player has not folded

winners += player

pot = distributeMoney(players, winners, pot)

a+=1

#### Distributing money

A recursive function evenly distributes to the winners the minimum amount of money a player in the list has put in multiplied by the number of players to put that amount of money in.

players <- players in table

Function distributeMoney(players, winners, pot)

if there are still winners left

money <- least amount of money a player in players has put in \* no players

moneyWon <- money / no of winners

if money cannot be evenly distributed between winners

newWinners <- winners without the winner who bet last

pot += distributeMoney(players, newWinners, odd money in pot)

increment each winner by moneyWon

pot <- pot – money

players <- players without the player who put the least money in

winners <- removed corresponding winner if needed

distributeMoney(players, winners, pot)

return pot

### Making the winner message

The hands of the players are shown in a clockwise manner from the dealer. If the player’s hand strength is the same as or beats every other hand currently shown, their hand is shown to all players.

Function makeWinnerMessage

message += ‘’

showHands <- []

winningIndex <- 999

player <- player to dealers left

while not iterated through every player or in first loop

get index of player in playerWin

moneyWon <- money player won in round

if index <= winningIndex and player hasn’t folded

showHands += [player, player’s cards, hand strength, moneyWon]

for player in showHands

message += data from player

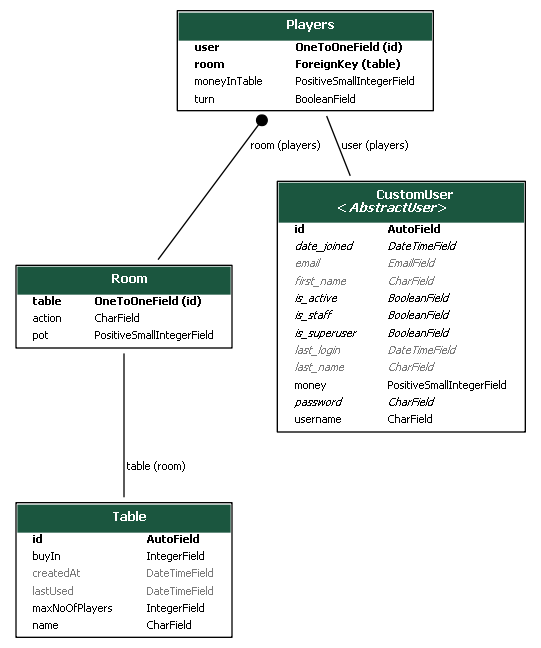
## Database Design

The database stores each user’s information and their encrypted password in the table **CustomUser**. It also stores the table information in **Table**. Once the poker algorithm is run, it creates a **Room** object that carries information about the current game. This has a one to one relationship with the **Table**.

**Room** persists as long as there are players in the table. Every time a player joins the game, they make a **Players** object that persists until they leave the table. This has a one to one relationship with **CustomUser** and a many to one relationship with the **Room**.

The **Players** object stores the money in play on the table and whether it is the players turn. This is so the poker algorithm can communicate with the consumer program, that only accepts player actions if it is the players turn.

The **Players** and **Room** tables could be replaced with a temporary storage NoSQL database using MongoDB for better read and write performance as the data changes frequently.



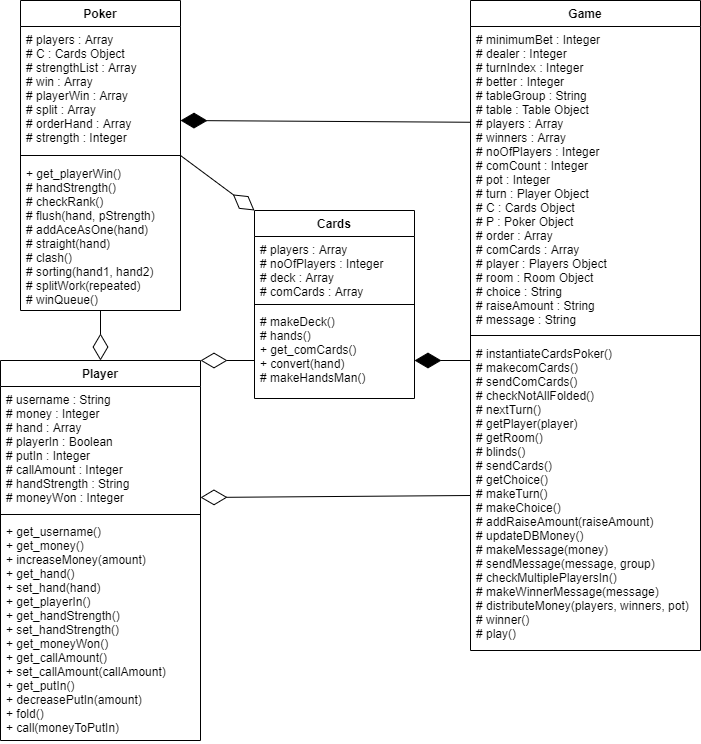
## Table Design

The online poker tables on the web app will be designed so that any user can create a poker table, specifying the buy in and the maximum number of players. The minimum bet is 1/100th of the buy in. If a table is empty for over 15 minutes it will be deleted. The buy in must be from 100 – 100000000 credits and the maximum number of players must be from 2-8.

## User Design

Each user will be able to create and join any table if they have enough money to meet the buy in. The starting amount of money for a user will be 1000 credits, which can be reset if they go below that. Users will be able to gain credits by winning them off other people in poker tables, and the credits will persist even after they log out. The users can view their rank against others on the leaderboard.

## OOP Design



# Implementation

## 2D Array

self.win is a 2D array that holds the players cards and hand strength in order to find the winning hands.

self.win.append([self.strength, player, self.orderHand[:]])

## Anonymous function

sortedPlayers = sorted(players, key = lambda x: x.putIn)

## Strength of hand

### Finding cards of the same rank

The algorithm iterates through every card in the hand, adding arrays of cards of the same rank to the sameRank array. It then sorts the array by the length of the arrays inside of it as 3 cards of the same rank will always be better than two and so on. The array is then truncated to only include the first two items in the array or less as a hand can only be made up of 5 cards so 3 pairs for example are never necessary. The hand strength is then calculated by checking the length of the sameRank array and the length of the first item in the array. The sameRank array is then converted into a 1D array and any cards not in sameRank appended in order onto the end. The sameRank array is then truncated to only include 5 cards and written to orderHand.

def checkRank(self, hand):

    #determines whether cards are a pair or 3 of a kind,

    #alternatively a two pair or fullhouse

    def twoThree(pStrength2, pStrength3):

        if len(sameRank[0]) == 3:

            return pStrength3

        else:

            return pStrength2

    i = 0

    sameRank = []

    while i < 6:

        temp = [hand[i]]

        try:

            #adds cards of same rank to temp

            while hand[i][0] == hand[i+1][0]:

                temp.append(hand[i+1])

                i+=1

            else:

                i+=1

        except IndexError:

            pass

        #if more than one card of same rank add to sameRank

        if len(temp) > 1:

            sameRank.append(temp[:])

    #sort by length of same ranked cards,

    #e.g. 4 of a kind > 3 of a kind > pair

    sameRank.sort(key = lambda x: len(x), reverse=True)

    sameRank = sameRank[:2]

    if len(sameRank) != 0:

        #four of a kind

        if len(sameRank[0]) == 4:

            sameRank = sameRank[0]

            self.strength = 7

        #two pair or full house

        elif len(sameRank) == 2:

            self.strength = twoThree(2, 6)

        #pair or three of a kind

        else:

            self.strength = twoThree(1, 3)

    #put all cards from sameRank in 1D array

    temp = []

    for cards in sameRank:

        for card in cards:

            temp.append(card)

    #add cards not included in sameRank

    for card in hand:

        if card not in temp:

            temp.append(card)

    #make orderHand

    self.orderHand = temp[:5]

### Finding flushes

def flush(self, hand, pStrength):

    #iterates over all 4 suits

    for i in range(4):

        flush = []

        for card in hand:

            #appends card to flush array if same suit

            if card[1] == i:

                flush.append(card)

        if len(flush) == 5 and self.strength < pStrength:

            self.strength = pStrength

            self.orderHand = flush[:5]

### Finding straights

The algorithm first temporarily adds an ace, usually represented as 14, as a 1 to the hand as aces can be used as low or high in straights. The algorithm then iterates through this new hand to check if any 5 cards are in sequential order. As the cards are sorted from high to low the algorithm will find and use the highest straights first. If two cards are of the same rank, the method recursively calls itself excluding one of the same ranked cards each time. This is because one of the cards may be part of a straight flush while the other may only lead to a straight. Once the algorithm has found a straight hand, it then calls the flush method to check whether the hand is a flush or not.

def addAceAsOne(self, hand):

    #temporarily adds ace as 1

    for card in hand:

        if 14 in card:

            if [1, card[1]] not in hand:

                hand.append([1, card[1]])

    return hand

def straight(self, hand):

    straightHand = []

    for j in range(len(hand)):

        if len(hand) > j+1:

            #as hand sorted reversed compare less 1 to the card

            #rank below it

            if hand[j][0]-1 == hand[j+1][0]:

                if len(straightHand) == 0:

                    straightHand.append(hand[j])

                straightHand.append(hand[j+1])

            elif hand[j][0] == hand[j+1][0]:

                #for straight flushes make a new straight check without

                #duplicate card evey time same number is found

                self.straight(hand[:j+1][:] + hand[j+2:][:])

                self.straight(hand[:j][:] + hand[j+1:][:])

            else:

                straightHand = []

            if len(straightHand) == 5:

                #checks if straight is straight flush

                self.flush(straightHand, 8)

                if self.strength < 4:

                    self.strength = 4

                    self.orderHand = straightHand

    #if the straight flush is Ace to 10 then it is a royal flush

    if self.strength == 8 and self.orderHand[0][0] == 14:

        self.strength = 9

## Binary Sort

If two players have the same hand strength, the program needs to sort the cards by the ‘strength’ of the card e.g. Ace > King. The cards are already ordered so the cards that make hand strength are at the front of the hand, e.g. the 10 pair comes before the Ace.

A binary sort is used to sort the hands as there will be at most 8 players in the table so the overhead necessary for a merge sort would not be efficient. If two players have exactly the same strength then the two players are appended to the split array in an array.

#binary sort but adds the players to repeated array if values are the same

def clash(self):

    repeated = []

    flip = True

    while flip:

        flip = False

        for a in range(len(self.win)):

            if len(self.win) > a+1:

                if self.win[a][0] == self.win[a+1][0]:

                    flip = self.sorting(self.win[a][2], self.win[a+1][2])

                    if flip == 'split':

                        flip = False

                        repeated.append(self.win[a][1])

                        repeated.append(self.win[a+1][1])

                    elif flip:

                        temp = self.win[a]

                        self.win[a] = self.win[a+1][:]

                        self.win[a+1] = temp[:]

    self.splitWork(repeated)

def sorting(self, hand1, hand2):

    a = 0

    #finds the first card where the values differ

    while hand1[a][0] == hand2[a][0] and a < 4:

        a+=1

    if hand1[a][0] > hand2[a][0]:

        return False

    elif hand1[a][0] < hand2[a][0]:

        return True

    else:

        return 'split'

#adds players with the the same hand to split in an array

def splitWork(self, repeated):

    for a in range(0, len(repeated), 2):

        if a - 1 >= 0:

            #the players are added in pairs, so if a player is the same as

            #a player in the previous iteration then all 3 players in the

            #current and previous iteration have the same strength hand.

            #So the other player in the current iteration is appended to the

            #previous iteration array

            if repeated[a] == repeated[a-1]:

                #-1 is the index of the last item in the array

                self.split[-1].append(repeated[a+1])

            else:

                self.split.append([repeated[a], repeated[a+1]])

        else:

            self.split.append([repeated[a], repeated[a+1]])

## List operations

def makeDeck(self):

    self.\_\_deck = [[k, j] for j in range(4) for k in range(2,15)]

## Queue

### Creating

From the sorted array created by the binary sort, the player object from each hand is appended to the playerWin queue in an array. If the player object is in the split array, i.e. the player has the same strength as another player and so would share the pot if won, the split array is appended to the win queue.

This is why the player object is appended in an array in first part so all the player objects are 2 layers deep. Also includes list operations append and remove.

As there is more than one player in the split array, the array will be appended more than once to playerWin. The bottom 3 lines remove any duplicate arrays in playerWin by converting the player arrays to tuples and creating a dictionary using the newly made tuples as keys. As dictionary keys must be unique, any tuples that are already a key are ignored. The dictionary and tuples are then converted back into arrays.

#adds each player to playerWin in an array

def winQueue(self):

    for strength, player, hand in self.win:

        added = False

        for players in self.split:

            if player in players:

                #if players in split it adds the split array instead

                self.playerWin.append(players)

                added = True

        if not added:

            self.playerWin.append([player])

    #remove duplicate split arrays

    self.playerWin = [tuple(x) for x in self.playerWin]

    self.playerWin = list(dict.fromkeys(self.playerWin))

    self.playerWin = [list(x) for x in self.playerWin]

### Using

To distribute the money according to how much money each player has put in the pot, the program de-queues the first array in the playerWin queue and passes all the players in the array that haven’t folded to the distributeMoney method.

def winner(self):

    a = 0

    while self.pot != 0:

        for player in self.P.playerWin[a]:

            if player.playerIn:

                self.winners.append(player)

        print('winners', self.winners)

        self.pot = self.distributeMoney(self.players[:], self.winners[:], self.pot)

        print(self.pot)

        self.updateDBMoney()

        a+=1

    self.makeWinnerMessage()

    self.sendMessage(self.message, self.tableGroup)

## Recursive Function

The recursive function sorts the players by the amount of money each player has put in. It calculates the amount of money each winner is due based on the number of winners and players in the array and decreases the amount each player has put in.

With each iteration it calls the function with one less player and deletes winners that have been de-queued from the players array. In certain situations, the total pot cannot be divided equally by the number of players.

The extra chips must be shared out as equally as possible, so the program divides the money by all the winners except the one furthest to the dealers right.

def distributeMoney(self, players, winners, pot):

    sortedPlayers = sorted(players, key = lambda x: x.putIn)

    winners.sort(key = lambda x: x.putIn)

    if len(winners) != 0:

        money = sortedPlayers[0].putIn

        #money given out equal to the minimum players putIn

        #or pot if in the oddMoney recursion

        moneyMade = money \* len(sortedPlayers)

        if moneyMade > pot:

            moneyMade = pot

            money = pot // len(sortedPlayers)

        #if the money cannot be shared equally

        oddMoney = moneyMade % len(winners)

        if oddMoney != 0:

            print('odd money in pot:', str(oddMoney))

            #share the money between the all the winners except the last

            a = -1

            while players[a] not in winners:

                a-=1

            print('removing winner:', players[a].username)

            tempWin = winners[:]

            tempWin.remove(players[a])

            pot += self.distributeMoney(players[:], tempWin, oddMoney)

        #decrease each players putIn by the min players putIn

        #increase each winners by the (min players putIn \* players)// no winners

        moneyWon = moneyMade // len(winners)

        for player in sortedPlayers:

            player.decreasePutIn(money)

            if player in winners:

                player.increaseMoney(moneyWon)

        #decrease pot by money given out

        pot -= moneyMade

        #delete minimum putIn player

        players.remove(sortedPlayers[0])

        if winners[0] == sortedPlayers[0]:

            del winners[0]

        pot = self.distributeMoney(players, winners, pot)

    return pot

## Create showdown winner message

Iterates through players from dealers left, and if their hand can beat or match any hand shown, they show their cards.

def makeWinnerMessage(self):

    self.message = '\n------------------------------------------'

    showHands = []

    startIndex = currentIndex = (self.dealer+1)%self.noOfPlayers

    winningIndex = 999

    firstRun = True

    #iterates through each player from dealers left as they show first

    while currentIndex != startIndex or firstRun:

        firstRun = False

        for a in range(len(self.P.playerWin)):

            if self.players[currentIndex] in self.P.playerWin[a]:

                currentWin = a

        if self.players[currentIndex].playerIn and currentWin <= winningIndex:

            winningIndex = currentWin

            playerStats = {

                'username': self.players[currentIndex].username,

                'moneyWon': self.players[currentIndex].moneyWon

            }

            if self.checkNotAllFolded():

                playerStats['hand'] = Cards.convert(self.players[currentIndex].hand)

                playerStats ['strength'] = ': ' + self.players[currentIndex].handStrength + ' '

            else:

                playerStats['hand'] = ''

                playerStats ['strength'] =  ''

            showHands.append(playerStats)

        currentIndex = (currentIndex+1)%self.noOfPlayers

    for player in showHands:

        winnings = ''

        if player['moneyWon'] != 0:

            winnings = ' won ' + str(player['moneyWon'])

        self.message += '\n' + player['username'] + winnings + player['strength'] + player['hand']

    self.message += '\n------------------------------------------\n'

## DB Objects

In Django, DB records can be modelled as Python objects. Each DB table is represented as its own class.

### Creating DB Objects

Each variable represents a row of the table. The **Table** object can also reference the **Room** object because there’s a one-to-one relationship between the **Table** and the **Room**, as instantiated in the **Room** object.

DB classes can also contain methods that can be run on the data in the table. These can be run by calling the method from a **Table** object as with normal OOP. For example, the **Room** table contains the players in a table. However, the **Room** object only exists while players are in the table. So, if the table has no players in it, trying to get the number of players in the table will return an error.

class Table(models.Model):

    def getNoOfPlayers(self):

        try:

            players = Players.objects.filter(room=self.room)

            for player in players:

                if player.moneyInTable == 0:

                    players.remove(player)

            noOfPlayers = len(players)

        except:

            noOfPlayers = 0

        return noOfPlayers

    name = models.CharField(max\_length=24, unique=True)

    buyIn = models.IntegerField(

validators=[MinValueValidator(100), MaxValueValidator(100000000)]

)

    maxNoOfPlayers = models.IntegerField(

validators=[MinValueValidator(2), MaxValueValidator(8)]

)

    createdAt = models.DateTimeField(auto\_now\_add=True)

    lastUsed = models.DateTimeField(auto\_now\_add=True)

### Instantiating DB objects and records, getting DB objects

The function gets the player’s user data from the database using the player’s username. Getting an object is similar to WHERE in SQL syntax, however must only return one DB object.

It then creates a new DB record using the **CustomUser** object and the **Room** object. This is stored as a one-to-one relationship with **CustomUser** and a foreign key with **Room**.

def addPlayer(room, table, username):

    player = CustomUser.objects.get(username=username)

    playerInstance = Players.objects.create(

user=player, room=room, moneyInTable=table.buyIn

)

    player.money -= table.buyIn

    player.save()

## Threading

### Creating thread

The view function has an @login\_required decorator which means that only logged in users can access the function. If the user has the minimum required money to join the table and the table is not full, it starts a daemon poker thread.

A daemon thread runs in the background so the view can return and the program can end to accept more web requests without the daemon thread ending.

@login\_required

def game(request, pk):

    table = get\_object\_or\_404(Table, pk=pk)

    if request.user.money >= table.buyIn and table.getNoOfPlayers() < table.maxNoOfPlayers:

        pokerThread = threading.Thread(

target=main, args=(pk, request.user.username), daemon=True

)

        pokerThread.start()

        context = {

            'table': table,

        }

        return render(request, 'game.html', context)

    return redirect('index')

### Thread function

The function checks for an active **Room**. If a room is found, it adds the player and exits. If the **Room** doesn’t exist, it creates a **Room** record and adds a player to the room. It then starts the game.

def main(pk, username):

    table = Table.objects.get(pk=pk)

    #check to see if table exists

    try:

        room = Room.objects.get(table=table)

        addPlayer(room, table, username)

    #if room dosen't exist create one

    except Room.DoesNotExist:

        room = Room.objects.create(table=table)

        addPlayer(room, table, username)

        startGame(table)

## Starting the game

This function runs in an infinite loop starting the next poker round when the previous round finishes.

def startGame(table):

    TESTING = False

    playersInGame = []

    dealer = 0

    tableGroup = 'table\_' + str(table.pk)

    while True:

        #waits until their is more than one player in the table to start

        table.refresh\_from\_db()

        while table.getNoOfPlayers() == 1:

            table.refresh\_from\_db()

            time.sleep(0.2)

        #if single player leaves table before anyone joins

        if table.getNoOfPlayers() == 0:

            print('player left, not in game')

            table.lastUsed = datetime.now(timezone.utc)

            table.save()

            sys.exit()

        #gets players in table

        players = Players.objects.filter(room\_id=table.id) #table group is the primary key of Room

        makePlayerOrder(playersInGame, players)

        if not TESTING:

            dealer = (dealer+1)%len(playersInGame)

        #starts game

        Game((table.buyIn)//100, dealer, tableGroup, table, playersInGame)

        time.sleep(0.4)

### Adding and newly joined players and removing players that have left

def makePlayerOrder(playersInGame, players):

    for player in playersInGame:

        #sets all the Player objects back to their base values

        player.newRound()

        #check whether player in playersInGame is in players

        #if not, the player has left

        if not any(x for x in players if x.user.username == player.username):

            playersInGame.remove(player)

    for newPlayer in players:

        #check whether Player object is not already in playersInGame

        #if so, new player has joined the table

        if newPlayer.moneyInTable > 0 and not any(x for x in playersInGame if x.username == newPlayer.user.username):

            P = Player(newPlayer.user.username, newPlayer.moneyInTable)

            print(P.username, 'has joined')

            playersInGame.append(P)

    print('playersInGame:', playersInGame)

## Synchronous web sockets

### Server-side

#### Connecting

This method adds a player to the table group and creates a unique group to receive their cards.

class PokerConsumer(WebsocketConsumer):

    #adds the player to the poker group to recieve the community cards and bets

    #adds the player to a unique group to recieve his cards

    def connect(self):

        self.pk = self.scope['url\_route']['kwargs']['pk']

        self.player = self.scope['user']

        self.username = self.player.username

        print('player:', self.username)

        self.tableGroup = 'table\_' + self.pk

        self.room = Room.objects.get(table\_id=self.pk)

        self.censoredList = getCensoredWords()

        #group socket

        async\_to\_sync(self.channel\_layer.group\_add)(

            self.tableGroup,

            self.channel\_name

        )

        #unique socket

        async\_to\_sync(self.channel\_layer.group\_add)(

            str(self.username),

            self.channel\_name

        )

        #accepts all communication with web socket

        self.accept()

#### Receiving

If the message is a chat message, sent to group. Else the message must be a poker action, so only accepted if it is the players turn.

def receive(self, text\_data):

    player = Players.objects.get(user=self.player)

    textDataJson = json.loads(text\_data)

    action = textDataJson['action']

    if action == 'message':

        message = textDataJson['message']

        if message != '':

            message = self.username + ': ' + message

            message = censor(message, self.censoredList)

            async\_to\_sync(self.channel\_layer.group\_send)(

            self.tableGroup,

            {

                'type': 'chatMessage',

                'text': message

            })

    elif player.turn:

        player.turn = False

        textDataJson = json.loads(text\_data)

        message = textDataJson['action']

        if message == 'fold':

            action = 'f'

        elif message == 'raise':

            raiseAmount = textDataJson['raiseAmount']

            action = 'r' + raiseAmount

        elif message == 'call':

            action = 'c'

        self.room.action = action

        self.room.save()

        player.save()

#### Sending

##### Poker method

This method gathers the data for each player to send to the consumer.

def sendCards(self):

    for player in self.players:

        if player.playerIn:

            hand = Cards.convert(player.hand)

            async\_to\_sync(get\_channel\_layer().group\_send)(

                player.username,

                {

                    'type': 'cards',

                    'hand': hand,

                    'comCards': self.comCards,

                    'dealer': self.players[self.dealer].username,

                    'moneyInTable': str(player.money)

                })

##### Consumer method

Method sends the message to the client.

def cards(self, event):

    message = 'cards'

    hand = event['hand']

    comCards = event['comCards']

    dealer = event['dealer']

    moneyInTable = event['moneyInTable']

    self.send(text\_data=json.dumps({

        'message': message,

        'hand': hand,

        'comCards': comCards,

        'dealer': dealer,

        'moneyInTable': moneyInTable

    }))

#### Disconnecting

Leave socket groups and add money from table to the players total money

def disconnect(self, closeCode):

    #disconnects from group sockets

    async\_to\_sync(self.channel\_layer.group\_discard)(

        self.tableGroup,

        self.channel\_name

    )

    async\_to\_sync(self.channel\_layer.group\_discard)(

        str(self.username),

        self.channel\_name

    )

    #update player money

    playerInstance = Players.objects.get(user=self.player)

    self.player.money += playerInstance.moneyInTable

    self.player.save()

    playerInstance.delete()

    #if noone left in table delete table

    self.room.refresh\_from\_db()

    players = Players.objects.filter(room=self.room)

    if len(players) == 0:

        self.room.delete()

### Client-side JavaScript

#### Connecting

Connects to socket using a URL, which is captured by routing.py

var pokerSocket = new WebSocket(

  'ws://' + window.location.host +

  '/ws/tables/' + pk + '/');

#### Receiving

Part of the receiving function. Mainly receiving data and writing it to specific locations on the page.

  pokerSocket.onmessage = function(e) {

    var data = JSON.parse(e.data);

    var message = data['message'];

    if (message === 'It\'s your turn') {

      var putIn = data['putIn'];

      showButton('buttons')

      if (putIn > 0) {

        document.getElementById('call').value = ('Call ' + putIn.toString(10));

        showButton('fold')

      } else {

        document.getElementById('call').value = ('Check')

        hideButton('fold')

      }

      document.querySelector('#poker-log').value = (message + '\n') +

      document.querySelector('#poker-log').value;

    } else if (message !== 'message') {

      hideButton('buttons')

    }

    if (message == 'cards') {

      var hand = data['hand'];

      var comCards = data['comCards'];

      var dealer = data['dealer'];

      var moneyInTable = data['moneyInTable'];

      document.getElementById('hand').innerHTML = ('Hand: ' + hand);

      document.getElementById('com-cards').innerHTML = ('Community cards: ' + comCards);

      document.getElementById('money-in-table').innerHTML = ('Money: ' + moneyInTable);

      document.getElementById('dealer').innerHTML = ('Dealer: ' + dealer)

    } else if (message == 'winner') {

      var showdown = data['showdown'];

      var log = data['log'];

      var pokerLog = document.querySelector('#poker-log');

      pokerLog.value = (log + '\n' + showdown + '\n') +

      pokerLog.value;

      document.getElementById('pot').innerHTML = ('Pot: 0');

#### Disconnecting

pokerSocket.onclose = function(e) {

  pokerSocket.send(JSON.stringify({

    'action': 'fold',

  }));

## Regular Expressions and Inheritance, Overriding

The user model inherits from the default **AbstractUser** class that comes by default with Django. The **AbstractUser** class contains user characteristics such as username, password.

The **CustomUser** class uses polymorphism to overwrite the default username field and regex expressions to check that the username is alphanumeric. The username is used for the players unique socket name and must be alphanumeric.

A \_\_str\_\_ method is called if the object is interpreted as a string e.g. a print statement of **CustomUser** would print the user’s username instead of an object instance.

class CustomUser(AbstractUser):

    alphanumeric = RegexValidator(

        regex=r'^[0-9a-zA-Z\_]\*$',

        message='Username must consist only of alphanumeric characters and underscores.'

    )

    username = models.CharField(max\_length=25, validators=[alphanumeric], unique=True)

    money = models.PositiveSmallIntegerField(default=1000)

    def \_\_str\_\_(self):

        return self.username

## Active nav bar tag – regular expressions

To determine whether the user is on a page referenced by the nav bar, the templates calls the tag that searches whether the URL referenced by the button is in the request path (URL). If so, it returns that the button should be active.

register = template.Library()

@register.simple\_tag

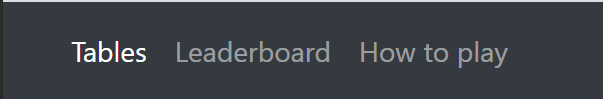
def active(request, pattern):

    stringPattern = '^' + pattern + '$'

    if re.search(stringPattern, request.path):

        return 'nav-item active'

    return 'nav-item'



## Exception handling and getting and setting private variables

### Private variables

class Player:

    def \_\_init\_\_(self, username, money):

        self.\_\_username = username

        self.\_\_money = money

        self.\_\_hand = []

        self.\_\_playerIn = True

        self.\_\_callAmount = self.\_\_putIn = 0

        self.\_\_handStrength = ''

        self.\_\_moneyWon = 0

### Exception handling

A property decorator works so that you can use the property function to get the value like a variable and must return a value. It is equivalent to a getter.

A setter function is called when you attempt to set the property function. It verifies that the input is valid and sets the value or raises an exception if not.

@property

def callAmount(self):

    return self.\_\_callAmount

@callAmount.setter

def callAmount(self, callAmount):

    if callAmount >= 0:

        self.\_\_callAmount = callAmount

    else:

        raise Exception('call amount less than 0')

## Web requests

### Server-side scripting

#### Class Based views

class SignUp(generic.CreateView):

    form\_class = CustomUserCreationForm #variables must be underscores

    success\_url = reverse\_lazy('login')

    template\_name = 'signup.html'

#### Function based view

@login\_required

def game(request, pk):

    table = get\_object\_or\_404(Table, pk=pk)

    if request.user.money >= table.buyIn and table.getNoOfPlayers() < table.maxNoOfPlayers:

        pokerThread = threading.Thread(target=main, args=(pk, request.user.username), daemon=True)

        pokerThread.start()

        context = {

            'table': table,

        }

        return render(request, 'game.html', context)

    return redirect('index')

### URL paths

#### HTTP

urlpatterns = [

    path('signup/', views.SignUp.as\_view(), name='signup'),

    path('p/<str:username>/', views.profile, name='profile'),

]

#### Consumer routing for sockets

application = ProtocolTypeRouter({

    #http to views is default

    'websocket': AuthMiddlewareStack(URLRouter([

        path('ws/user/<str:username>/', MoneyConsumer),

        path('ws/tables/<str:pk>/', PokerConsumer),

        ])),

})

### Handling form POST request and GET request

#### Form Model

The create table function uses a form created using the **Table** model. The form can then be saved directly as a **Table** record in the DB

class TableForm(forms.ModelForm):

    class Meta:

        #specifys what model to use

        model = Table

        #fields from Table model included in form

        fields = ('name', 'buyIn', 'maxNoOfPlayers')

        #read friendly names

        labels = {

            'name': 'Name',

            'buyIn': 'Buy in',

            'maxNoOfPlayers': 'Maximum number of players'

        }

#### Form View

If the form is invalid, the POST request will contain any errors that can be displayed to the user, e.g. password must be more than 8 characters.

@login\_required

def createTable(request):

    #user submitting the form

    if request.method == 'POST':

        #gets form submission based on the POST request

        form = TableForm(request.POST)

        if form.is\_valid():

            #saves form as Table model

            table = form.save()

            return redirect('game', pk=table.pk)

    #user GETting the form

    elif request.method == 'GET':

        form = TableForm()

    context = {'form': form}

    return render(request, 'table-form.html', context)

## Serializing DB object into JSON using RESTful API

### Serializing data

To monitor the number of players in each table and the tables available in real time, I used the Django REST framework to serialize the **Table** model into JSON format. This data is passed to the JavaScript using Django channels as JavaScript cannot interpret Python objects.

The name and maxNoOfPlayers come from the **Table** model and the noOfPlayers field comes from the noOfPlayers MethodField. This runs a method that returns a value. The default method it runs is get\_<field name>. In get\_noOfPlayers it runs the **Table** object method getNoOfPlayers.

class TableSerializer(serializers.ModelSerializer):

    noOfPlayers = serializers.SerializerMethodField()

    class Meta:

        model = Table

        #the fields serailized

        fields = ['name', 'maxNoOfPlayers', 'noOfPlayers']

    def get\_noOfPlayers(self, obj):

        #uses the Table method to get the no of players

        return obj.getNoOfPlayers()

### Creating and using serialized data

This combines the player’s money with the money the player has in the table in order to get the total money. The method then serializes the table and sends the serialized data and money to the client using Django channels. This process is repeated until the threading event; stopEvent is set. This is set on the disconnection of the socket, otherwise the thread would stay alive until the termination of the server.

def checkMoney(self, stopEvent):

    while not stopEvent.is\_set():

        self.player = CustomUser.objects.get(username=self.username)

        self.totalMoney = self.player.money

        self.moneyInTable = 0

        try:

            self.playerGame = Players.objects.get(pk=self.player)

            self.moneyInTable = self.playerGame.moneyInTable

            self.totalMoney += self.moneyInTable

        except Players.DoesNotExist:

            pass

        self.tables = Table.objects.all()

        self.serializedTables = TableSerializer(self.tables, many=True)

        self.tableJSON = JSONRenderer().render(self.serializedTables.data)

        self.send(text\_data=json.dumps({

            'money': self.totalMoney,

            'moneyInTable': self.moneyInTable,

            'tables': json.loads(self.tableJSON),

        }))

        time.sleep(1)

## Reading from file

def getCensoredWords():

    censoredList = []

    path = 'C:\\Users\\Tom\\Desktop\\projects\\web\_poker\\censored-words.txt'

    with open(path, 'r') as censoredWords:

        for word in censoredWords:

            w = word.replace('\n', '')

            censoredList.append(w)

    return censoredList

## Scheduling and custom commands

I created a custom command called startserver that clears any players currently in tables on start-up and runs a thread to delete tables that have not been used for 15 minutes. It then runs the server.

class Command(BaseCommand):

    help = 'Calls runserver and cleartables, creates and a daemon thread that removes \

tables that have been inactive for more than 15 minutes'

    def add\_arguments(self, parser):

        parser.add\_argument(

'addrport', nargs='?', type=str, default='127.0.0.1:8000', help='ipaddr:port'

)

    def handle(self, \*args, \*\*kwargs):

        #os.system('docker run --name channels\_app -p 6379:6379 -d redis:2.8')

        call\_command('cleartables')

        addrport = kwargs['addrport']

        thread = threading.Thread(target=self.removeTables, daemon=True)

        thread.start()

        call\_command('runserver', '--noreload', addrport)

    def removeTables(self):

        while True:

            tables = Table.objects.all()

            for table in tables:

                timeDiff = datetime.now(timezone.utc) - table.lastUsed

                timeDiff = timeDiff.total\_seconds()/60

                if timeDiff > 15 and table.getNoOfPlayers() == 0:

                    print('deleting %s, not used for: %d minutes' % (table.name, timeDiff))

                    table.delete()

            time.sleep(10)

## Complex SQL – leaderboard view

def leaderboard(request):

    users = CustomUser.objects.filter().values('username', 'money').order\_by('-money')

    context = {

        'users': users

    }

    return render(request, 'leaderboard.html', context)

# Testing

## Test videos

### Test video

[bit.ly/project-full-tests](http://bit.ly/project-full-tests)

### Normal use

[bit.ly/project-normal-use](http://bit.ly/project-normal-use)

## Test table

Hands are ordered by card rank

Player 1: dealer

Player 2: SB

Player 3: BB

| Test no | Test description | Expected outcome | Actual outcome | Video timestamp |
| --- | --- | --- | --- | --- |
| 1 | User signs up and logs in | If the data given by the user complies with profile restraints, the users profile is stored in the database.  The user can then log in with username and password. | As expected | 00:14 |
| 2 | Passwords stored with the strong hash SHA-256 | The password ‘testing11’ will be stored in the database as: | As expected | 01:14 |
| 3 | Logged in user clicks their username in the top right of the dashboard | The website recognises the user is accessing their own profile and shows your profile on the profile page | As expected | 01:20 |
| 4 | A logged in user creates a table called table1 with buy-in 100 and max no of players 3 | A table is created and stored in the database. The user will be redirected to the table | As expected | 01:55 |
| 5 | A player joins a table while another user is viewing the home page | The number of players in the table is incremented by 1 on the user’s home page without refreshing the page | As expected | 02:28 |
| 6 | Player 1 sends message ‘hello’ in chat | Every player in the table receive a message in the chat log: ‘Player 1: hello’ | As expected | 02:31 |
| 7 | Player has cards:  K♣ K♦ 10♦ 8♦ 4♥ 4♦ 4♠ | Detects a full house hand, uses cards:  4♥ 4♦ 4♠ K♣ K♦ | As expected | 03:21 |
| 8 | Player has cards:  8♥ 7♥ 6♥ 5♥4♥ 3♥ 2♥ | Detects a straight flush hand, uses cards:  8♥ 7♥ 6♥ 5♥4♥ | As expected | 04:15 |
| 9 | Player 1 hand:  2♥ 2♦ 2♠ A♦ K♦  Player 2 hand:  A♥ A♦ K♠ K♦ Q♠ | Player 1 beats player 2 as three of a kind is of greater strength than two pairs | As expected | 05:03 |
| 10 | Player 1 hand:  7♥ 7♦ K♠ J♠ 6♠  Player 2 hand:  7♥ 7♦ K♠ 10♦ 6♠ | Player 1 beats player 2, as the Jack kicker is higher rank than the 10 | As expected | 05:59 |
| 11 | Player 1 hand:  7♥ 7♦ K♠ J♠ 6♠  Player 2:  K♠ J♠ 6♠ 5♠ 2♠  Player 3:  7♥ 7♦ K♠ 10♦ 6♠ | Player 1’s cards are shown as he is first to the left of the dealer, player 2 cards are hidden as he has folded, even though he has the strongest hand. Player 3 cards are shown even though he cannot beat player 1’s hand because the hands are shown from the dealers left and as player 1 is the dealer his hand is shown last. | As expected | 06:30 |
| 12 | Player 1 and 2 calls BB, player 3 raises the bet by 10, player 1 goes all-in, player 2 folds, player 3 calls the all-in | The betting round continues back round to the player 3 as he was the last to bet in the round. The game is then skipped to showdown as only player 2 is capable of betting. | As expected | 06:30 |
| 13 | Player 1 hand:  7♥ 7♦ K♠ J♠ 6♠  Player 2 hand:  7♠ 7♦ K♠ J♠ 6♠  Player 3 hand:  K♠ J♠ 7♦ 6♠ 4♣  Player 1 raises 1 and player 2 and 3 call the bet. All players check till the round is finished. | Player 1 and player 2 have the same hand, so the pot is split between them, showing them both as winners in the poker log. However, the pot cannot be fairly split between the two players so the odd money is allocated to player 1 as the dealer is first in the betting order. | As expected | 07:23 |
| 14 | Player 1 hand:  7♥ 7♦ 7♣ K♠ J♠  Player 2 hand:  K♠ K♣ J♠ 7♦ 6♠  Player 3 hand:  K♠ J♠ 7♦ 6♠ 5♣  Player 1 goes all-in with 50, player 2 re-raises 50 and 3 call the bet. | A side pot is created as player 1 cannot stand to win all the money in the pot. Player 1 has the strongest hand so wins the side pot of 150, the remaining 100 goes to player 2 with the second strongest hand. | As expected | 07:50 |
| 15 | Player leaves the table | Hand is automatically folded | As expected | 08:21 |
| 16 | User tries to enter a full poker table then tries to enter a table with not enough money to meet the buy-in and finally a not logged in users try to join a table | Users are redirected to the home page | As expected | 09:08 |
| 17 | User accesses leaderboard page, clicks on themself and other users from the leaderboard. | A table of all players on the website shown, ordered by their total money. The rows of the table are buttons that redirect the user to the requested user’s profile. The website recognises a user accessing their own profile and displays the message your profile when clicked on. | As expected | 10:20 |

## Possible tests

### Poker algorithm

* The deck is randomly shuffled each game
* 2 random unique cards are dealt to each player and 5 more dealt as community cards
* Hands consist of the strongest set of 5 of the above 7 cards
* The algorithm detects hands:
  + High card
  + Pair
  + Two Pairs
  + Three of a kind
  + Straight
  + Flush
  + Full house
  + Straight flush
  + Royal flush
* The card rank is used to judge cards of same strength e.g. Ace beats a King
* Flush and straight use highest ranking cards first
* The full house 3 of a kind takes precedence in comparing hand rank
* Kickers (cards not used to contribute to the hand strength) used to break ties in strength
* Stronger hands beat weaker hands e.g. three of a kind beats a pair
* The dealer button moves clockwise each round
* Betting turns move clockwise around the table from the dealer
* If a player bets betting continues to that player, i.e. the round continues until every player has either matched the largest bet, gone all-in or folded
* If a player goes all-in, they will not be asked to bet again
* If players are all-in and can no longer bet the community cards are shown and the game goes to showdown
* The algorithm detects if people have same ranking hand and shares pot between them
* Shared pots are calculated based on how much money each player put in
* A player’s hand not shown if folded
* Player hands are shown from dealer left
* A hand is only shown if it can beat the current best shown hand
* In situations where the pot cannot be shared fairly, the odd chips go to the players that is first in the betting order
* Every round the SB and BB are compulsory posted by the two players to the dealers left
* The SB and BB rotate with dealer each game
* If a player leaves the table, they automatically fold their current hand
* When a player leaves their money in the table is added to their total money
* The amount of money in a game stays consistent i.e. no lost chips
* Any players still in a game can either raise call or fold on their turn
* If the current bet is 0 the call button replaced with check and the fold button is removed
* Players cannot make an action (call, raise fold) if it is not their turn
* Player action buttons disappear if not their turn
* Invalid raise values, i.e. not a positive integer, are caught and the user is prompted to enter the value again
* Pot, players hand, current money in table, dealer, community cards shown to player at the top left of the page
* Total player money (money in table + money outside table) updated live on top right

### Poker log

* The round winners, necessary players cards and hand strength along with money won by each player is shown at the end of the round
* Once a player chooses an action it is printed in the chat
* Community cards printed once turned
* SB BB players shown
* 'It's your turn’ message to indicate to the user it is their turn

### Poker chat

* Players can send and receive messages in chat
* When a player sends a message, it has <username>: prepended to message
* Words in censor list file censored out of the message with an appropriate amount of \*
* Players cannot enter messages greater than 100 characters

### Website interface

* Players on leaderboard page shown in a descending order by their total money
* Tables that have not been used for more than 15 minutes are removed
* All records in the temporary storage DB tables Players and Room are removed when the server is first started
* The current page in the navigation bar is highlighted
* If the user scrolls over navigation bar button the text is highlighted

#### Any user

* Can view every player on leaderboard page
* Users can click on player on the leaderboard to view their profile
* Can see the current tables
* Can view the number of players in a table be updated live
* Can read the how to play page
* Can sign up and log in
* Sign up users must comply with user restraints:
  + Username:
    - Can have a maximum length of 25
    - Can only use alphanumeric characters and underscores
  + Password:
    - Can't be too similar to other personal information.
    - Must contain at least 8 characters.
    - Can't be a commonly used password.
    - Can't be entirely numeric.
* CSRF tokens used in sign up and log in process to prevent CSRF attacks
* All new users start with 1000 money
* Users’ passwords stored using strong hashing algorithm with salt

#### Logged in users

* Users can join tables in which they have enough money to meet the initial buy-in and are not full, if not they are redirected to the home page
* Can create tables
* Cannot create tables with buy in under 100, max players 2-8. If the form data does not meet the requirements then an error is shown on how to fix it and the user can submit the form again
* Users can click their username on the top right to reach their profile
* Website recognises a user accessing their own profile
* Users can reset their money to 1000 if less than that amount from the home page
* Users over 1000 cannot reset their money
* Users can log out
* Users cookies saved in browser so can be automatically logged in after the tab is closed

# Evaluation

## Analysis of objectives

1. **Players will be able to register with a username and password, and log in/out. The users account information is then stored in the database**

The home page of the website has log in and sign up buttons for users not logged in. The buttons lead to web forms in which the users can enter their account information. If the data is not valid (e.g. not valid email format) or incorrect (wrong username password combination) the forms provide feedback and the user can enter it again. Once a user is logged in a log out button is given at the bottom of every page. The users account information is stored in the **CustomUser** table in the SQLite3 database. This objective has been fully met.

1. **A player’s total money is stored in a database.**

A player’s money is stored in their **CustomUser** record in the database.

* 1. **On entering a table, the buy in of the table is deducted from the user’s total money and is stored in a temporary table, in which the user can bet with**

On joining the table, the addPlayer function reduces the players money by the table buy in. The players current money in table is stored in the python Player object in the poker algorithm and in the temporary **Players** table.

* 1. **The money in the table is updated during play**

The money in table is stored in the poker algorithm Player object and is synced with **Players** every turn.

* 1. **Once a player leaves a table their money from the table is added to the player’s total money and the database is updated**

On disconnection from the poker socket in consumers the players total money is incremented by the players money in table.

* 1. **Players can reset their money if it reduces below 1000**

If a player’s money is below 1000, a button appears on the home page ‘Reset money’. Which redirects to reset-money/. This calls the resetMoney view, which checks the user’s money is below 1000 and if so sets the users money to 1000. It then redirects back to the home page.

1. **User passwords are stored with the SHA-256 hash**

The user’s passwords are stored with the SHA-256 hash in the database, using the PBKDF hashing algorithm

1. **Players will be able to create table, and the and join them through the home page where all current tables are displayed**

On the home page there is a button for logged in users to create a table which redirects to a web form to create a table. When the user submits a form which is valid, they are redirected to the table. From the home page users can join tables they have enough money to enter. If a user tries to join a table they have insufficient money to enter they are redirected back to the home page.

1. **If a table has not been used in 15 minutes, it is deleted**

On starting the server, a daemon thread is created that infinitely loops, checking every 10 seconds how long since a table has been last used. If the table hasn’t been used for over 15 minutes then it is deleted. The program updates a tables last use on when the last player left, but doesn’t delete tables with players still in.

1. **The pack in each table is shuffled between rounds**

The poker algorithm initializes a new deck array every round, and uses the random library to shuffle the array.

1. **The dealer button moves to the dealers left each round**

The dealer button is represented by an integer that dictates the position of the dealer in the playersInGame array and players array in Game. It is incremented by 1 each round and taken mod of the number of players.

1. **Tables have a maximum number of players (between 2 and 8) which can be specified by the table’s creator**

The restrictions are made in the **Table** model which the model form that creates the table inherits. The web form has an input to enter the maximum number of players and invalidates the form, prompting the user to enter it again if it is outside that range.

1. **Tables have a buy-in to enter (between 100 and 100,000,000), specified by the table’s creator which is used to determine how much a player must put in the table**

The table buy in is also specified in the **Table** model through the same procedures as above. The table buy is the amount of money that the player plays with in the table.

* 1. **Used to calculate the size of the blinds, the small blind 1/100th the size of the buy in.**

The buy in is used to calculate the size of the small blind, 1/100th of the buy in in the poker algorithm. The big blind is conversely twice the small blind so 2/100th of the buy in.

1. **Once in a table, players can play live poker according to the rules in the Texas hold ’em section in analysis**

Currently, neither I or any end user can find any bugs in the poker algorithm or game. The poker game on my website follows the Texas hold ‘em rules specified in the in the analysis exactly, which includes every rule in Texas hold ‘em cash games.

1. **Players will only be able to see their pocket cards and the table’s community cards**

Each player is a member of the tables group socket, which sends pot and player actions and chat messages. Players also have a private socket that sends the players money and cards, and dealer. This way no player can see anyone else’s cards.

1. **If a player leaves a table, they will automatically fold**

When a player leaves the table, they disconnect from the web socket in which their **Players** record is deleted. On their turn, if no record relating to a player is found, it means the player has left and the player is folded.

1. **A leader board shows players ranked by their total money**

A SQL query is used to get all users usernames and money ordered by their money in descending order. This is then passed to the dynamic template which iterates through each of them adding them to a table to create a static web page. Each record of the table is a CSS button that redirects to the user’s profile.

1. **Players can interact with each other using an in-table chat**

The message is sent over poker web socket in each table. Once the user presses enter, the JavaScript takes the input from the chat box and sends it over the web socket. The algorithm appends <username>: to the front of the message and sends it to the players in the table, who receive it and pass it to the JavaScript which writes it to the top of the chat text box.

* 1. **The chat can be filtered so that swear words would be censored**

Once a player joins a table, the web socket reads a censored word file and adds each word to an array. Once a message is received from the user, it checks whether any words in the message are in the censored words array. If so it replaces the word with an appropriate number of asterisks.

1. **A how to play page is provided for new players**

A static web page is served at the URL how-to-play/ which contains the poker rules from the analysis. This is available to all users and is a static HTML resource.

1. **Multiple games on the website can be played at the same time**

Each game is running on its own poker thread and has its own **Room** and **Table** record, and each player in the table has a **Players** object, which has a foreign key to the corresponding **Room** they are in. This system can sustain as many simultaneous poker games as needed up to the servers computing power.

## Feedback

Supervisor and end user comments – ***Bold and italic***

### Supervisor comments

When I performed my whole system test the overall feedback was positive and I feel like I’ve made a good base product that meet all the objectives I had for the project. My supervisor commented that it was always clear in a game what was currently happening, and the general UI of the website was clean and intuitive. However, Archie suggested that ***I could use images of cards in the game and animations rather than the cards being text based and that the users could be shown to be sat round the table to easily show the betting order.*** I totally agree with this and these were a few of my extension objectives that I decided not to implement as it would not add much to the complexity of the program.

Archie also suggested that ***the censored words in the text chat could be improved to match all cases and 1337 speak*** (e.g. replacing e with 3)***, and also match even with spacing and punctuation between letters.*** This could also apply to table names. We also found a minor bug where if players go all-in, and after which only 1 player has money left in the table, the river is shown in the poker log but not in the current community cards. This was easily fixed with a moving of a function and was a very minor bug and so after much testing I believe the underlying algorithms and programs are solid.

### End user feedback

My end users were 6 other members of my class, aged 17-18, playing on their own computers connected to the web server over local Wi-Fi. I supervised them but let them use the website themselves. They created many different tables and accounts and users played in different tables simultaneously.

Users suggested that there could be a way of ***adding friends on the app, which you could join and play with if they were online.*** They also suggested ***more stats to be shown on the player profile such as games played and best hand as it looks quite bare***.

In the poker game, my end users thought that adding a way to ***see all other players in game and their money*** rather than just the poker log showing what each players action was. This could again be also used to represent the player betting order so players have a better idea on what position they are in the table – which can be extremely important in poker. Also suggested was a ***re buy button so when players run out of money, they are prompted to re buy in the table instead of spectating and having to leave and re-join the table.*** There could also be a spectator feature in each game where you optionally watch the game without buying in.

## Improvements

### Improvements based on feedback

As stated in my feedback, I could improve the GUI in the poker game by using card and poker chip images and animations, and the players could be shown in a circle around the table to easily show betting order and remove a layer of abstraction to the game. A rebuy button could be shown for when players run out of money so that they can easily keep playing instead of leaving the table and re-joining.

Users could be given the ability to add friends, in which they can view who is active and join their table if possible, from the ‘friends’ sidebar. Players could add friends through poker games, visiting their profile, or searching their username in the sidebar. Conversely players can invite their friends to tables. Another feature could be that players could create private tables that are only available to their friends or require a password to enter.

User stats could be added, which would be linked with each user to the stored on the database. For example: hands played, hands won, best hand. If a user visits a profile, they can view the players stats. I could also add a settings page so when a user visits their own profile, they can edit settings on their account, such as changing email and password or turning censored chat on or off.

The chat could also be filtered as mentioned in my feedback.

### Other improvements

Use NO-SQL database for Players and Room tables, as they are temporary tables that are written to often, so for scalability if the website had more visitors, a NO-SQL database such as MongoDB would be ideal for these tables to reduce the number of read and writes in the database. The web application could also use HTTPS to secure the communication, which would be essential if it were available on the open internet. While my sign up and log in pages already have CSRF tokens, if I was to make it publicly available, I would need a re-captcha authentication to prevent bots, and using email confirmation to prove accounts are real.

The website could be hosted on the internet through a service such as AWS and purchasing a domain name. This would require some maintenance and the database would have to be migrated from SQLlite3 to a more stable system such as PostgreSQL.

Currently, the site is manually tested, which means I manually test that every part of the site works the way it should myself. As the application becomes larger, it becomes increasingly harder to check new updates don’t affect other existing parts of the site. Django by default includes the means to write automated tests, these are simply requests or inputs on parts of the website where the correct value is already known. (e.g. giving a hand with a pair to poker algorithm and testing it recognises the pair). This is known as ‘unit testing’.

A bet slider could be used to raise instead of a text box to better the user experience. Players could also decide on joining a table whether they want to sit down or spectate the game. A feature could be added that allows players to join multiple tables at once and switch between them.

I could have used a JavaScript framework on the frontend such as Vue.js to make the web app more reactive such as newly created tables showing on the home screen without refreshing. It could also reduce the amount of custom JS I have written.

The backend of the web application already uses Django Rest Framework to serialize table and player data for the frontend. I could use this to create an API that could power other services such as a mobile app.

A poker AI could be developed that users could optionally add to tables to fill player spots. When playing with an AI bot, the user would use temporary money that would not affect their total money they have gained so users cannot exploit the bots.

Instead of using fake money, the website could allow users to bet with real money against other players. However, this would require the website to be regularly and extensively security tested as the website would be storing bank information, which is a much higher security risk and also makes the website a target to hackers. It would also have to comply with government laws and have a system of verifying ID documents to prevent under 18s from using this feature. So realistically this would be extremely hard or impossible to maintain without a larger team and beyond the scope of this project.

# Code

## Dictionary tree

web\_poker

│ censored-words.txt

│ db.sqlite3

│ manage.py

│

├───accounts

│ forms.py

│ models.py

│ serializers.py

│ urls.py

│ views.py

│

├───leaderboard

│ urls.py

│ views.py

│

├───poker

│ │ consumers.py

│ │ models.py

│ │ poker.py

│ │ urls.py

│ │ views.py

│ │

│ └───management

│ └───commands

│ cleartables.py

│

├───project

│ routing.py

│ settings.py

│ urls.py

│

├───rules

│ urls.py

│ views.py

│

├───tables

│ │ consumers.py

│ │ forms.py

│ │ models.py

│ │ serializers.py

│ │ urls.py

│ │ views.py

│ │

│ ├───management

│ │ └───commands

│ │ startserver.py

│ │

│ └───templatetags

│ tags.py

│

└───templates

│ base.html

│ game.html

│ how-to-play.html

│ index.html

│ leaderboard.html

│ profile.html

│ signup.html

│ table-form.html

│

└───registration

login.html

### censored-word.txt

dick

smelly

heck

### db.sqlite3

Sqlite3 database

### manage.py

Premade program that comes bundled with Django to start the web server

#!/usr/bin/env python

"""Django's command-line utility for administrative tasks."""

import os

import sys

def main():

    os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'project.settings')

    try:

        from django.core.management import execute\_from\_command\_line

    except ImportError as exc:

        raise ImportError(

            "Couldn't import Django. Are you sure it's installed and "

            "available on your PYTHONPATH environment variable? Did you "

            "forget to activate a virtual environment?"

        ) from exc

    execute\_from\_command\_line(sys.argv)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

## accounts

### forms.py

from django import forms

from django.contrib.auth.forms import UserCreationForm, UserChangeForm

from .models import CustomUser

class CustomUserCreationForm(UserCreationForm):

    class Meta(UserCreationForm):

        model = CustomUser

        fields = ('username', 'email')

class CustomUserChangeForm(UserChangeForm):

    class Meta:

        model = CustomUser

        fields = ('username', 'email')

### models.py

from django.db import models

from django.contrib.auth.models import AbstractUser

from django.core.validators import RegexValidator

class CustomUser(AbstractUser):

    alphanumeric = RegexValidator(

        regex=r'^[0-9a-zA-Z\_]\*$',

        message='Username must consist only of alphanumeric characters and underscores.'

    )

    username = models.CharField(max\_length=25, validators=[alphanumeric], unique=True)

    money = models.PositiveSmallIntegerField(default=1000)

    def \_\_str\_\_(self):

        return self.username

### serializers.py

from rest\_framework import serializers

from .models import CustomUser

class UserSerailizer(serializers.ModelSerializer):

    class Meta:

        model = CustomUser

        fields = '\_\_all\_\_'

### urls.py

from django.urls import path

from . import views

urlpatterns = [

    path('signup/', views.SignUp.as\_view(), name='signup'),

    path('p/<str:username>/', views.profile, name='profile'),

]

### views.py

from django.contrib.auth.forms import UserCreationForm

from django.urls import reverse\_lazy

from django.views import generic

from django.shortcuts import render

from .models import CustomUser

from .forms import CustomUserCreationForm

class SignUp(generic.CreateView):

    form\_class = CustomUserCreationForm #variables must be underscores

    success\_url = reverse\_lazy('login')

    template\_name = 'signup.html'

def profile(request, username):

    player = CustomUser.objects.get(username=username)

    context = {

        'player': player

    }

    return render(request, 'profile.html', context)

## leaderboard

### urls.py

from django.urls import path

from . import views

urlpatterns = [

     path('', views.leaderboard, name='leaderboard'),

]

### views.py

from django.shortcuts import render

from accounts.models import CustomUser

def leaderboard(request):

    users = CustomUser.objects.filter().values('username', 'money').order\_by('-money')

    context = {

        'users': users

    }

    return render(request, 'leaderboard.html', context)

## poker

### consumers.py

from asgiref.sync import async\_to\_sync

from channels.generic.websocket import WebsocketConsumer

from .models import Players, Room

from accounts.models import CustomUser

from tables.models import Table

import json

class PokerConsumer(WebsocketConsumer):

    #adds the player to the poker group to recieve the community cards and bets

    #adds the player to a unique group to recieve his cards

    def connect(self):

        self.pk = self.scope['url\_route']['kwargs']['pk']

        self.player = self.scope['user']

        self.username = self.player.username

        print('player:', self.username)

        self.tableGroup = 'table\_' + self.pk

        self.room = Room.objects.get(table\_id=self.pk)

        self.censoredList = getCensoredWords()

        #group socket

        async\_to\_sync(self.channel\_layer.group\_add)(

            self.tableGroup,

            self.channel\_name

        )

        #unique socket

        async\_to\_sync(self.channel\_layer.group\_add)(

            str(self.username),

            self.channel\_name

        )

        #accepts all communication with web socket

        self.accept()

    def disconnect(self, closeCode):

        #disconnects from group sockets

        async\_to\_sync(self.channel\_layer.group\_discard)(

            self.tableGroup,

            self.channel\_name

        )

        async\_to\_sync(self.channel\_layer.group\_discard)(

            str(self.username),

            self.channel\_name

        )

        #update player money

        playerInstance = Players.objects.get(user=self.player)

        self.player.money += playerInstance.moneyInTable

        self.player.save()

        playerInstance.delete()

        #if noone left in table delete table

        self.room.refresh\_from\_db()

        players = Players.objects.filter(room=self.room)

        if len(players) == 0:

            self.room.delete()

    def receive(self, text\_data):

        player = Players.objects.get(user=self.player)

        textDataJson = json.loads(text\_data)

        action = textDataJson['action']

        if action == 'message':

            message = textDataJson['message']

            if message != '':

                message = self.username + ': ' + message

                message = censor(message, self.censoredList)

                async\_to\_sync(self.channel\_layer.group\_send)(

                self.tableGroup,

                {

                    'type': 'chatMessage',

                    'text': message

                }

                )

        elif player.turn:

            player.turn = False

            textDataJson = json.loads(text\_data)

            message = textDataJson['action']

            if message == 'fold':

                action = 'f'

            elif message == 'raise':

                raiseAmount = textDataJson['raiseAmount']

                action = 'r' + raiseAmount

            elif message == 'call':

                action = 'c'

            self.room.action = action

            self.room.save()

            player.save()

    def pokerMessage(self, event):

        message = event['message']

        pot = event['pot']

        self.send(text\_data=json.dumps({

            'message': message,

            'pot': pot,

        }))

    def playerTurn(self, event):

        message = 'It\'s your turn'

        putIn = event['putIn']

        self.send(text\_data=json.dumps({

            'message': message,

            'putIn': putIn

        }))

    def cards(self, event):

        message = 'cards'

        hand = event['hand']

        comCards = event['comCards']

        dealer = event['dealer']

        moneyInTable = event['moneyInTable']

        self.send(text\_data=json.dumps({

            'message': message,

            'hand': hand,

            'comCards': comCards,

            'dealer': dealer,

            'moneyInTable': moneyInTable

        }))

    def showWinner(self, event):

        message = 'winner'

        winner = event['winner']

        showdown = event['showdown']

        log = winner + ' wins'

        self.send(text\_data=json.dumps({

            'message': message,

            'showdown': showdown,

            'log': log

        }))

    def chatMessage(self, event):

        text = event['text']

        self.send(text\_data=json.dumps({

            'message': 'message',

            'text': text

        }))

def getCensoredWords():

    censoredList = []

    path = 'C:\\Users\\Tom\\Desktop\\projects\\web\_poker\\censored-words.txt'

    with open(path, 'r') as censoredWords:

        for word in censoredWords:

            w = word.replace('\n', '')

            censoredList.append(w)

    return censoredList

def censor(message, censoredList):

    words = message.split(' ')

    for word in words:

        if word in censoredList:

            message = message.replace(word, '\*' \* len(word))

    return message

### models.py

from django.db import models

class Players(models.Model):

    user = models.OneToOneField(

'accounts.CustomUser', max\_length=24, primary\_key=True, on\_delete=models.CASCADE

)

    room = models.ForeignKey('Room', on\_delete=models.CASCADE, null=True)

    moneyInTable = models.PositiveSmallIntegerField()

    turn = models.BooleanField(default=False)

class Room(models.Model):

    table = models.OneToOneField(

'tables.Table', primary\_key=True, on\_delete=models.CASCADE, related\_name='room'

)

    action = models.CharField(max\_length=15, default=None, null=True)

    pot = models.PositiveSmallIntegerField(default=0)

### poker.py

import random

from .models import Players, Room

from channels.layers import get\_channel\_layer

from asgiref.sync import async\_to\_sync

from tables.models import Table

import time

from accounts.models import CustomUser

import sys

from datetime import datetime, timezone

class Player:

    def \_\_init\_\_(self, username, money):

        self.\_\_username = username

        self.\_\_money = money

        self.\_\_hand = []

        self.\_\_playerIn = True

        self.\_\_callAmount = self.\_\_putIn = 0

        self.\_\_handStrength = ''

        self.\_\_moneyWon = 0

    @property

    def username(self):

        return self.\_\_username

    @property

    def money(self):

        return self.\_\_money

    def increaseMoney(self, amount):

        self.\_\_money += amount

        self.\_\_moneyWon += amount

    @property

    def hand(self):

        return self.\_\_hand

    @hand.setter

    def hand(self, hand):

        self.\_\_hand = hand

    @property

    def playerIn(self):

        return self.\_\_playerIn

    @property

    def handStrength(self):

        return self.\_\_handStrength

    @handStrength.setter

    def handStrength(self, strength):

        self.\_\_handStrength = strength

    @property

    def moneyWon(self):

        return self.\_\_moneyWon

    @property

    def callAmount(self):

        return self.\_\_callAmount

    @callAmount.setter

    def callAmount(self, callAmount):

        if callAmount >= 0:

            self.\_\_callAmount = callAmount

        else:

            raise Exception('call amount less than 0')

    @property

    def putIn(self):

        return self.\_\_putIn

    def decreasePutIn(self, amount):

        self.\_\_putIn -= amount

    def fold(self):

        self.\_\_playerIn = False

    def newRound(self):

        self.\_\_playerIn = True

        self.\_\_callAmount = self.\_\_putIn = 0

        self.\_\_moneyWon = 0

        self.\_\_hand = []

        self.\_\_handStrength = ''

    def call(self, moneyToPutIn):

        if self.\_\_money > moneyToPutIn:

            self.\_\_money -= moneyToPutIn

        else: #all-in situation

            moneyToPutIn = self.\_\_money

            self.\_\_money = 0

        self.\_\_putIn += moneyToPutIn

        self.\_\_callAmount = 0

        return moneyToPutIn

class Cards:

    def \_\_init\_\_(self, players):

        TESTING = False #change for testing purposes

        self.\_\_players = players

        self.\_\_deck = []

        self.\_\_comCards = []

        self.makeDeck()

        if not TESTING:

            self.hands()

        else:

            self.makeHandsMan()

    def makeDeck(self):

        self.\_\_deck = [[k, j] for j in range(4) for k in range(2,15)]

    def hands(self):

        random.shuffle(self.\_\_deck)

        self.\_\_comCards = self.\_\_deck[:5][:]

        del self.\_\_deck[:5]

        for player in self.\_\_players:

            playerHand = self.\_\_deck[:2][:]

            del self.\_\_deck[:2]

            player.hand = playerHand

    @property

    def comCards(self):

        return self.\_\_comCards

    #converts cards into human readable form

    def convert(hand):

        numbers = [[11, 'J'], [12, 'Q'], [13, 'K'], [14, 'A'], [1, 'A']]

        suits = ['♥', '♦', '♠', '♣']

        convertHand = ''

        for a in range(len(hand)):

            add = False

            for item in numbers:

                if hand[a][0] == item[0]:

                    convertHand += item[1]

                    add = True

            if not add:

                convertHand += str(hand[a][0])

            for b in range(4):

                if hand[a][1] == b:

                    convertHand += (suits[b] + ' ')

        return convertHand

    def makeHandsMan(self):

        self.\_\_comCards = [[5, 3], [7, 1], [13, 2], [6, 2], [11, 2]]

        hands = [[

                [7, 3], [7, 0] #first player hand

            ], [

                [13, 3], [4, 2] #second player hand etc

            ], [

                [2, 1], [4, 3]

        ]]

        for player, hand in zip(self.\_\_players, hands):

            player.hand = hand

class Poker:

    def \_\_init\_\_(self, players, C):

        self.players = players

        self.C = C

        self.strengthList = ['High Card', 'Pair', 'Two Pair',\

        'Three of a kind', 'Straight', 'Flush', 'Full House', 'Four of a kind',\

        'Straight Flush', 'Royal Flush']

        self.win = []

        self.\_\_playerWin = []

        self.split = []

        self.handStrength()

        self.winQueue()

    @property

    def playerWin(self):

        return self.\_\_playerWin

    @playerWin.setter

    def playerWin(self, playerWin):

        self.\_\_playerWin = playerWin

    def handStrength(self):

        for player in self.players:

            self.orderHand = []

            self.strength = 0

            hand = player.hand + self.C.comCards

            hand.sort(reverse=True)

            self.checkRank(hand)

            self.flush(hand, 5)

            tempHand = self.addAceAsOne(hand)

            self.straight(tempHand)

            player.handStrength = self.strengthList[self.strength]

            self.win.append([self.strength, player, self.orderHand[:]])

        self.win.sort(key = lambda x: x[0], reverse=True)

        self.clash()

    def checkRank(self, hand):

        #determines whether cards are a pair or 3 of a kind,

        #alternatively a two pair or fullhouse

        def twoThree(pStrength2, pStrength3):

            if len(sameRank[0]) == 3:

                return pStrength3

            else:

                return pStrength2

        i = 0

        sameRank = []

        while i < 6:

            temp = [hand[i]]

            try:

                #adds cards of same rank to temp

                while hand[i][0] == hand[i+1][0]:

                    temp.append(hand[i+1])

                    i+=1

                else:

                    i+=1

            except IndexError:

                pass

            #if more than one card of same rank add to sameRank

            if len(temp) > 1:

                sameRank.append(temp[:])

        #sort by length of same ranked cards,

        #e.g. 4 of a kind > 3 of a kind > pair

        sameRank.sort(key = lambda x: len(x), reverse=True)

        sameRank = sameRank[:2]

        if len(sameRank) != 0:

            #four of a kind

            if len(sameRank[0]) == 4:

                sameRank = sameRank[0]

                self.strength = 7

            #two pair or full house

            elif len(sameRank) == 2:

                self.strength = twoThree(2, 6)

            #pair or three of a kind

            else:

                self.strength = twoThree(1, 3)

        #put all cards from sameRank in 1D array

        temp = []

        for cards in sameRank:

            for card in cards:

                temp.append(card)

        #add cards not included in sameRank

        for card in hand:

            if card not in temp:

                temp.append(card)

        #make orderHand

        self.orderHand = temp[:5]

    def flush(self, hand, pStrength):

        #iterates over all 4 suits

        for i in range(4):

            flush = []

            for card in hand:

                #appends card to flush array if same suit

                if card[1] == i:

                    flush.append(card)

            if len(flush) == 5 and self.strength < pStrength:

                self.strength = pStrength

                self.orderHand = flush[:5]

    def addAceAsOne(self, hand):

        #temporarily adds ace as 1

        for card in hand:

            if 14 in card:

                if [1, card[1]] not in hand:

                    hand.append([1, card[1]])

        return hand

    def straight(self, hand):

        straightHand = []

        for j in range(len(hand)):

            if len(hand) > j+1:

                #as hand sorted reversed compare less 1 to the card

                #rank below it

                if hand[j][0]-1 == hand[j+1][0]:

                    if len(straightHand) == 0:

                        straightHand.append(hand[j])

                    straightHand.append(hand[j+1])

                elif hand[j][0] == hand[j+1][0]:

                    #for straight flushes make a new straight check without

                    #duplicate card evey time same number is found

                    self.straight(hand[:j+1][:] + hand[j+2:][:])

                    self.straight(hand[:j][:] + hand[j+1:][:])

                else:

                    straightHand = []

                if len(straightHand) == 5:

                    #checks if straight is straight flush

                    self.flush(straightHand, 8)

                    if self.strength < 4:

                        self.strength = 4

                        self.orderHand = straightHand

        #if the straight flush is Ace to 10 then it is a royal flush

        if self.strength == 8 and self.orderHand[0][0] == 14:

            self.strength = 9

    #binary sort but adds the players to repeated array if values are the same

    def clash(self):

        repeated = []

        flip = True

        while flip:

            flip = False

            for a in range(len(self.win)):

                if len(self.win) > a+1:

                    if self.win[a][0] == self.win[a+1][0]:

                        flip = self.sorting(self.win[a][2], self.win[a+1][2])

                        if flip == 'split':

                            flip = False

                            repeated.append(self.win[a][1])

                            repeated.append(self.win[a+1][1])

                        elif flip:

                            temp = self.win[a]

                            self.win[a] = self.win[a+1][:]

                            self.win[a+1] = temp[:]

        self.splitWork(repeated)

    def sorting(self, hand1, hand2):

        a = 0

        #finds the first card where the values differ

        while hand1[a][0] == hand2[a][0] and a < 4:

            a+=1

        if hand1[a][0] > hand2[a][0]:

            return False

        elif hand1[a][0] < hand2[a][0]:

            return True

        else:

            return 'split'

    #adds players with the the same hand to split in an array

    def splitWork(self, repeated):

        for a in range(0, len(repeated), 2):

            if a - 1 >= 0:

                #the players are added in pairs, so if a player is the same as

                #a player in the previous iteration then all 3 players in the

                #current and previous iteration have the same strength hand.

                #So the other player in the current iteration is appended to the

                #previous iteration array

                if repeated[a] == repeated[a-1]:

                    #-1 is the index of the last item in the array

                    self.split[-1].append(repeated[a+1])

                else:

                    self.split.append([repeated[a], repeated[a+1]])

            else:

                self.split.append([repeated[a], repeated[a+1]])

    #adds each player to playerWin in an array

    def winQueue(self):

        for strength, player, hand in self.win:

            added = False

            for players in self.split:

                if player in players:

                    #if players in split it adds the split array instead

                    self.playerWin.append(players)

                    added = True

            if not added:

                self.playerWin.append([player])

        #remove duplicate split arrays

        self.playerWin = [tuple(x) for x in self.playerWin]

        self.playerWin = list(dict.fromkeys(self.playerWin))

        self.playerWin = [list(x) for x in self.playerWin]

        print('playerWin', self.playerWin)

class Game:

    def \_\_init\_\_(self, minimumBet, dealer, tableGroup, table, playersInGame):

        self.minimumBet = minimumBet

        self.dealer = self.turnIndex = self.better = dealer

        self.tableGroup = tableGroup

        self.table = table

        self.players = playersInGame

        self.winners = []

        self.noOfPlayers = len(self.players)

        self.comCount = 0

        self.pot = 0

        self.instantiateCardsPoker()

        self.play()

    def instantiateCardsPoker(self):

        self.C = Cards(self.players)

        self.P = Poker(self.players, self.C)

    def makeComCards(self):

        if self.comCount == 0:

            self.comCards = ''

            message = ''

        if self.comCount == 1:

            self.comCards = Cards.convert(self.C.comCards[:3])

            message = 'Flop: '

        elif self.comCount == 2:

            self.comCards = Cards.convert(self.C.comCards[:4])

            message = 'Turn: '

        elif self.comCount == 3:

            self.comCards = Cards.convert(self.C.comCards[:])

            message = 'River: '

        self.comCount+=1

        return message

    def sendComCards(self, message):

        message += self.comCards

        if message != '':

            self.sendMessage(message, self.tableGroup)

    def checkNotAllFolded(self):

        count = 0

        for player in self.players:

            if player.playerIn:

                count += 1

        if count > 1:

            return True

        else:

            return False

    def nextTurn(self):

        self.turnIndex = (self.turnIndex+1)%self.noOfPlayers

        self.turn = self.players[self.turnIndex]

    def getPlayer(self, player):

        try:

            userInstance = CustomUser.objects.get(username=player.username)

            player = Players.objects.get(user\_id=userInstance.id)

        except Players.DoesNotExist:

            self.getRoom()

            return (False, '')

        return (True, player)

    def getRoom(self):

        try:

            self.room = Room.objects.get(table=self.table)

        except Room.DoesNotExist:

            print('everyone left')

            self.table.lastUsed = datetime.now(timezone.utc)

            self.table.save()

            sys.exit()

    def blinds(self):

        sb = self.addRaiseAmount(self.minimumBet)

        self.nextTurn()

        bb = self.addRaiseAmount(self.minimumBet)

        self.nextTurn()

        message = self.turn.username + ' posted BB (' + str(bb + sb) + ')\n'

        message += self.turn.username + ' posted SB (' + str(sb) + ')'

        self.sendMessage(message, self.tableGroup)

    def sendCards(self):

        for player in self.players:

            if player.playerIn:

                hand = Cards.convert(player.hand)

                async\_to\_sync(get\_channel\_layer().group\_send)(

                    player.username,

                    {

                        'type': 'cards',

                        'hand': hand,

                        'comCards': self.comCards,

                        'dealer': self.players[self.dealer].username,

                        'moneyInTable': str(player.money)

                    }

                )

    def getChoice(self):

        putIn = str(self.turn.callAmount)

        async\_to\_sync(get\_channel\_layer().group\_send)(

            self.turn.username,

            {

                'type': 'playerTurn',

                'putIn': putIn,

            }

        )

        playerLeft = False

        self.getRoom()

        while self.room.action is None and not playerLeft:

            self.getRoom()

            #everyone leaves while its your turn

            if self.table.getNoOfPlayers() == 1:

                self.room.action = 'c'

                self.room.save()

                self.choice = 'c'

            elif self.room.action is not None:

                #the first character is the action the user wants to take

                #after that it is the optional raiseAmount

                self.choice = self.room.action[0]

                if self.choice == 'r':

                    try:

                        self.raiseAmount = self.room.action[1:]

                        if not int(self.raiseAmount) > 0:

                            raise ValueError()

                    except ValueError:

                        self.sendMessage('Raise amount must be a positive integer', self.turn.username)

                        self.makeTurn()

            if not self.getPlayer(self.turn)[0]:

                self.choice = 'f'

                playerLeft = True

                print(self.turn.username, 'left')

        self.room.action = None

        self.room.save()

    def makeTurn(self):

        playerExists, player = self.getPlayer(self.turn)

        if playerExists:

            player.turn = True

            player.save()

            self.getChoice()

        else:

            self.choice = 'f'

    def makeChoice(self):

        money = 0

        if self.choice == 'c':

            money = self.turn.call(self.turn.callAmount)

            self.pot += money

        elif self.choice == 'r':

            money = self.addRaiseAmount(int(self.raiseAmount))

        elif self.choice == 'f':

            self.turn.fold()

        self.makeMessage(money)

    def addRaiseAmount(self, raiseAmount):

        self.better = self.turnIndex

        callAmount = self.turn.call(self.turn.callAmount)

        raiseAmount = self.turn.call(raiseAmount)

        self.pot += (raiseAmount + callAmount)

        for player in self.players:

            if self.turn != player:

                player.callAmount += raiseAmount

        return raiseAmount

    def updateDBMoney(self):

        for user in self.players:

            playerExists, player = self.getPlayer(user)

            if playerExists:

                player.moneyInTable = user.money

                player.save()

    def makeMessage(self, money):

        if self.choice == 'f':

            message = self.turn.username + ' folded'

        elif self.choice == 'r':

            if self.turn.money == 0:

                message = self.turn.username + ' went all-in'

            else:

                message = self.turn.username + ' raised ' +  str(money)

        if self.choice == 'c':

            if money == 0:

                message = self.turn.username + ' checked'

            else:

                message = self.turn.username + ' called ' + str(money)

        self.sendMessage(message, self.tableGroup)

    def sendMessage(self, message, group):

        async\_to\_sync(get\_channel\_layer().group\_send)(

            group,

            {

                'type': 'pokerMessage',

                'message': message,

                'pot': str(self.pot),

            }

        )

    def checkMultiplePlayersIn(self):

        count = 0

        for player in self.players:

            if player.money > 0:

                count+=1

        if count > 1:

            if self.table.getNoOfPlayers() > 1:

                return True

        return False

    def makeWinnerMessage(self):

        self.message = '\n------------------------------------------'

        showHands = []

        startIndex = currentIndex = (self.dealer+1)%self.noOfPlayers

        winningIndex = 999

        firstRun = True

        #iterates through each player from dealers left as they show first

        while currentIndex != startIndex or firstRun:

            firstRun = False

            for a in range(len(self.P.playerWin)):

                if self.players[currentIndex] in self.P.playerWin[a]:

                    currentWin = a

            if self.players[currentIndex].playerIn and currentWin <= winningIndex:

                winningIndex = currentWin

                playerStats = {

                    'username': self.players[currentIndex].username,

                    'moneyWon': self.players[currentIndex].moneyWon

                }

                if self.checkNotAllFolded():

                    playerStats['hand'] = Cards.convert(self.players[currentIndex].hand)

                    playerStats ['strength'] = ': ' + self.players[currentIndex].handStrength + ' '

                else:

                    playerStats['hand'] = ''

                    playerStats ['strength'] =  ''

                showHands.append(playerStats)

            currentIndex = (currentIndex+1)%self.noOfPlayers

        for player in showHands:

            winnings = ''

            if player['moneyWon'] != 0:

                winnings = ' won ' + str(player['moneyWon'])

            self.message += '\n' + player['username'] + winnings + player['strength'] + player['hand']

        self.message += '\n------------------------------------------\n'

    def distributeMoney(self, players, winners, pot):

        sortedPlayers = sorted(players, key = lambda x: x.putIn)

        winners.sort(key = lambda x: x.putIn)

        if len(winners) != 0:

            money = sortedPlayers[0].putIn

            #money given out equal to the minimum players putIn

            #or pot if in the oddMoney recursion

            moneyMade = money \* len(sortedPlayers)

            if moneyMade > pot:

                moneyMade = pot

                money = pot // len(sortedPlayers)

            #if the money cannot be shared equally

            oddMoney = moneyMade % len(winners)

            if oddMoney != 0:

                print('odd money in pot:', str(oddMoney))

                #share the money between the all the winners except the last

                a = -1

                while players[a] not in winners:

                    a-=1

                print('removing winner:', players[a].username)

                tempWin = winners[:]

                tempWin.remove(players[a])

                pot += self.distributeMoney(players[:], tempWin, oddMoney)

            #decrease each players putIn by the min players putIn

            #increase each winners by the (min players putIn \* players)// no winners

            moneyWon = moneyMade // len(winners)

            for player in sortedPlayers:

                player.decreasePutIn(money)

                if player in winners:

                    player.increaseMoney(moneyWon)

            #decrease pot by money given out

            pot -= moneyMade

            #delete minimum putIn player

            players.remove(sortedPlayers[0])

            if winners[0] == sortedPlayers[0]:

                del winners[0]

            pot = self.distributeMoney(players, winners, pot)

        return pot

    def winner(self):

        a = 0

        while self.pot != 0:

            for player in self.P.playerWin[a]:

                if player.playerIn:

                    self.winners.append(player)

            print('winners', self.winners)

            self.pot = self.distributeMoney(self.players[:], self.winners[:], self.pot)

            print(self.pot)

            self.updateDBMoney()

            a+=1

        self.makeWinnerMessage()

        self.sendMessage(self.message, self.tableGroup)

    def play(self):

        print('in game')

        self.nextTurn()

        for a in range(4):

            #one to the dealers left

            self.better = (self.dealer+1)%self.noOfPlayers

            firstRun = True

            if a == 0:

                self.blinds()

            message = self.makeComCards()

            self.sendCards()

            if self.checkNotAllFolded():

                if self.checkMultiplePlayersIn():

                    self.sendComCards(message)

                    while (self.turnIndex != self.better or firstRun):

                        self.updateDBMoney()

                        self.sendCards()

                        firstRun = False

                        if self.turn.money != 0 and self.turn.playerIn:

                            self.makeTurn()

                            self.makeChoice()

                        self.nextTurn()

                elif a == 3:

                    self.sendComCards(message)

                    self.sendCards()

        self.winner()

def addPlayer(room, table, username):

    player = CustomUser.objects.get(username=username)

    playerInstance = Players.objects.create(user=player, room=room, moneyInTable=table.buyIn)

    player.money -= table.buyIn

    player.save()

def makePlayerOrder(playersInGame, players):

    for player in playersInGame:

        #sets all the Player objects back to their base values

        player.newRound()

        #check whether player in playersInGame is in players

        #if not, the player has left

        if not any(x for x in players if x.user.username == player.username):

            playersInGame.remove(player)

    for newPlayer in players:

        #check whether Player object is not already in playersInGame

        #if so, new player has joined the table

        if newPlayer.moneyInTable > 0 and not any(x for x in playersInGame if x.username == newPlayer.user.username):

            P = Player(newPlayer.user.username, newPlayer.moneyInTable)

            print(P.username, 'has joined')

            playersInGame.append(P)

    print('playersInGame:', playersInGame)

def startGame(table):

    TESTING = False

    playersInGame = []

    dealer = 0

    tableGroup = 'table\_' + str(table.pk)

    while True:

        #waits until their is more than one player in the table to start

        table.refresh\_from\_db()

        while table.getNoOfPlayers() == 1:

            table.refresh\_from\_db()

            time.sleep(0.2)

        #if single player leaves table before anyone joins

        if table.getNoOfPlayers() == 0:

            print('player left, not in game')

            table.lastUsed = datetime.now(timezone.utc)

            table.save()

            sys.exit()

        #gets players in table

#table group is the primary key of Room

        players = Players.objects.filter(room\_id=table.id)

        makePlayerOrder(playersInGame, players)

        if not TESTING:

            dealer = (dealer+1)%len(playersInGame)

        #starts game

        Game((table.buyIn)//100, dealer, tableGroup, table, playersInGame)

        time.sleep(0.4)

def main(pk, username):

    table = Table.objects.get(pk=pk)

    #check to see if table exists

    try:

        room = Room.objects.get(table=table)

        addPlayer(room, table, username)

    #if room dosen't exist create one

    except Room.DoesNotExist:

        room = Room.objects.create(table=table)

        addPlayer(room, table, username)

        startGame(table)

### urls.py

from django.urls import path

from . import views

urlpatterns = [

    path('<int:pk>/', views.game, name='game'),

]

### views.py

from django.shortcuts import render, redirect, get\_object\_or\_404

from tables.models import Table

import threading

from .poker import main

from django.contrib.auth.decorators import login\_required

@login\_required

def game(request, pk):

    table = get\_object\_or\_404(Table, pk=pk)

    if request.user.money >= table.buyIn and table.getNoOfPlayers() < table.maxNoOfPlayers:

        pokerThread = threading.Thread(

target=main, args=(pk, request.user.username), daemon=True

)

        pokerThread.start()

        context = {

            'table': table,

        }

        return render(request, 'game.html', context)

    return redirect('index')

### management/ commands

#### cleartables.py

from django.core.management.base import BaseCommand, CommandError

from poker.models import Players, Room

class Command(BaseCommand):

    help = 'Clears Room and Players from DB'

    def handle(self, \*args, \*\*kwargs):

        Players.objects.all().delete()

        Room.objects.all().delete()

        print('Tables cleared')

## project

### routing.py

from channels.routing import ProtocolTypeRouter, URLRouter

from channels.auth import AuthMiddlewareStack

from django.urls import path

from poker.consumers import PokerConsumer

from tables.consumers import MoneyConsumer

application = ProtocolTypeRouter({

    #http to views is default

    'websocket': AuthMiddlewareStack(URLRouter([

        path('ws/user/<str:username>/', MoneyConsumer),

        path('ws/tables/<str:pk>/', PokerConsumer),

        ])),

})

### settings.py

Preconfigured file with some altered settings

"""

Django settings for tablemanager project.

Generated by 'django-admin startproject' using Django 2.2.1.

For more information on this file, see

https://docs.djangoproject.com/en/2.2/topics/settings/

For the full list of settings and their values, see

https://docs.djangoproject.com/en/2.2/ref/settings/

"""

import os

# Build paths inside the project like this: os.path.join(BASE\_DIR, ...)

BASE\_DIR = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))

# Quick-start development settings - unsuitable for production

# See https://docs.djangoproject.com/en/2.2/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!

SECRET\_KEY = 'd938i2060c)\_!q16$fg6@\*+@4tbuzd&cc5cqg-4hi^%p@q$feh'

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = True

ALLOWED\_HOSTS = [

    '\*'

]

# Application definition

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'tables',

    'django\_extensions',

    'channels',

    'poker',

    'leaderboard',

    'accounts',

    'crispy\_forms',

    'rest\_framework'

]

MIDDLEWARE = [

    'django.middleware.security.SecurityMiddleware',

    'django.contrib.sessions.middleware.SessionMiddleware',

    'django.middleware.common.CommonMiddleware',

    'django.middleware.csrf.CsrfViewMiddleware',

    'django.contrib.auth.middleware.AuthenticationMiddleware',

    'django.contrib.messages.middleware.MessageMiddleware',

    'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

ROOT\_URLCONF = 'project.urls'

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [os.path.join(BASE\_DIR, 'templates')],

        'APP\_DIRS': True,

        'OPTIONS': {

            'context\_processors': [

                'django.template.context\_processors.debug',

                'django.template.context\_processors.request',

                'django.contrib.auth.context\_processors.auth',

                'django.contrib.messages.context\_processors.messages',

            ],

        },

    },

]

WSGI\_APPLICATION = 'project.wsgi.application'

# Database

# https://docs.djangoproject.com/en/2.2/ref/settings/#databases

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),

    }

}

# Password validation

# https://docs.djangoproject.com/en/2.2/ref/settings/#auth-password-validators

AUTH\_PASSWORD\_VALIDATORS = [

    {

        'NAME': 'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator',

    },

    {

        'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator',

    },

    {

        'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator',

    },

    {

        'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator',

    },

]

# Internationalization

# https://docs.djangoproject.com/en/2.2/topics/i18n/

LANGUAGE\_CODE = 'en-gb'

TIME\_ZONE = 'GMT'

USE\_I18N = True

USE\_L10N = True

USE\_TZ = True

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/2.2/howto/static-files/

STATIC\_URL = '/static/'

STATICFILES\_DIRS = (

    os.path.join(BASE\_DIR, 'static'),

)

STATIC\_ROOT = os.path.join(BASE\_DIR, 'staticfiles')

LOGIN\_REDIRECT\_URL = '/'

LOGOUT\_REDIRECT\_URL = '/'

AUTH\_USER\_MODEL = 'accounts.CustomUser'

ASGI\_APPLICATION = 'project.routing.application'

CHANNEL\_LAYERS = {

    'default': {

        'BACKEND': 'channels\_redis.core.RedisChannelLayer',

        'CONFIG': {

            "hosts": [('127.0.0.1', 6379)],

        },

    },

}

CRISPY\_TEMPLATE\_PACK = 'bootstrap4'

REST\_FRAMEWORK = {

    'DEFAULT\_PERMISSION\_CLASSES': ('rest\_framework.permissions.IsAdminUser',),

    'PAGINATE\_BY': 10

}

DOCKER\_HOST = '192.168.99.100:2376'

### urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('', include('tables.urls')),

    path('poker/', include('poker.urls')),

    path('leaderboard/', include('leaderboard.urls')),

    path('admin/', admin.site.urls),

    path('accounts/', include('accounts.urls')),

    path('accounts/', include('django.contrib.auth.urls')),

    path('how-to-play/', include('rules.urls')),

]

## rules

### urls.py

from django.urls import path

from . import views

urlpatterns = [

     path('', views.pokerRules, name='pokerRules'),

]

### views.py

from django.shortcuts import render

def pokerRules(request):

    return render(request, 'how-to-play.html')

## tables

### consumers.py

from django.core import serializers

from channels.generic.websocket import WebsocketConsumer

from accounts.models import CustomUser

from .models import Table

from .serializers import TableSerializer

from rest\_framework.renderers import JSONRenderer

from rest\_framework.parsers import JSONParser

import time

from poker.consumers import Players

import json

import threading

class MoneyConsumer(WebsocketConsumer):

    def connect(self):

        self.accept()

        self.username = self.scope['url\_route']['kwargs']['username']

        self.player = CustomUser.objects.get(username=self.username)

        self.stopEvent = threading.Event()

        self.thread = threading.Thread(

target=self.checkMoney, args=(self.stopEvent,), daemon=True

)

        self.thread.start()

    def disconnect(self, closeCode):

        self.stopEvent.set()

    def checkMoney(self, stopEvent):

        while not stopEvent.is\_set():

            self.player = CustomUser.objects.get(username=self.username)

            self.totalMoney = self.player.money

            self.moneyInTable = 0

            try:

                self.playerGame = Players.objects.get(pk=self.player)

                self.moneyInTable = self.playerGame.moneyInTable

                self.totalMoney += self.moneyInTable

            except Players.DoesNotExist:

                pass

            self.tables = Table.objects.all()

            self.serializedTables = TableSerializer(self.tables, many=True)

            self.tableJSON = JSONRenderer().render(self.serializedTables.data)

            self.send(text\_data=json.dumps({

                'money': self.totalMoney,

                'moneyInTable': self.moneyInTable,

                'tables': json.loads(self.tableJSON),

            }))

            time.sleep(1)

### forms.py

from django import forms

from .models import Table

class TableForm(forms.ModelForm):

    class Meta:

        #specifys what model to use

        model = Table

        #fields from Table model included in form

        fields = ('name', 'buyIn', 'maxNoOfPlayers')

        #read friendly names

        labels = {

            'name': 'Name',

            'buyIn': 'Buy in',

            'maxNoOfPlayers': 'Maximum number of players'

        }

### models.py

from django.db import models

from django.core.validators import MaxValueValidator, MinValueValidator

from poker.models import Players

class Table(models.Model):

    def getNoOfPlayers(self):

        try:

            players = Players.objects.filter(room=self.room)

            for player in players:

                if player.moneyInTable == 0:

                    players.remove(player)

            noOfPlayers = len(players)

        except:

            noOfPlayers = 0

        return noOfPlayers

    name = models.CharField(max\_length=24, unique=True)

    buyIn = models.IntegerField(

validators=[MinValueValidator(100), MaxValueValidator(100000000)]

)

    maxNoOfPlayers = models.IntegerField(

validators=[MinValueValidator(2), MaxValueValidator(8)]

)

    createdAt = models.DateTimeField(auto\_now\_add=True)

    lastUsed = models.DateTimeField(auto\_now\_add=True)

### serializers.py

from .models import Table

from poker.models import Room

from rest\_framework import serializers

class TableSerializer(serializers.ModelSerializer):

    noOfPlayers = serializers.SerializerMethodField()

    class Meta:

        model = Table

        #the fields serailized

        fields = ['name', 'maxNoOfPlayers', 'noOfPlayers']

    def get\_noOfPlayers(self, obj):

        #uses the Table method to get the no of players

        return obj.getNoOfPlayers()

### urls.py

from django.urls import path

from . import views

urlpatterns = [

     path('', views.index, name='index'),

     path('reset-money/', views.resetMoney, name='resetMoney'),

     path('create-table/', views.createTable, name='tableCreateView')

]

### views.py

from django.shortcuts import render, redirect

from tables.models import Table

from poker.models import Room

from django.contrib.auth.decorators import login\_required

from .forms import TableForm

def index(request):

    tables = Table.objects.all()

    context = {

        'tables': tables

    }

    return render(request, 'index.html', context)

@login\_required

def resetMoney(request):

    if request.user.money < 1000:

        print('reset')

        request.user.money = 1000

        request.user.save()

        return redirect('index')

    else:

        print('ERROR attempted reset')

        return redirect('index')

@login\_required

def createTable(request):

    #user submitting the form

    if request.method == 'POST':

        #gets form submission based on the POST request

        form = TableForm(request.POST)

        if form.is\_valid():

            #saves form as Table model

            table = form.save()

            return redirect('game', pk=table.pk)

    #user GETting the form

    elif request.method == 'GET':

        form = TableForm()

    context = {'form': form}

    return render(request, 'table-form.html', context)

### management/ commands

#### startserver.py

from django.core.management.base import BaseCommand, CommandError

import threading

from django.core.management import call\_command

from tables.models import Table

from datetime import datetime, timezone

import time

import os

class Command(BaseCommand):

    help = 'Calls runserver and cleartables, creates and a daemon thread that removes \

tables that have been inactive for more than 15 minutes'

    def add\_arguments(self, parser):

        parser.add\_argument(

'addrport', nargs='?', type=str, default='127.0.0.1:8000', help='ipaddr:port'

)

    def handle(self, \*args, \*\*kwargs):

        #os.system('docker run --name channels\_app -p 6379:6379 -d redis:2.8')

        call\_command('cleartables')

        addrport = kwargs['addrport']

        thread = threading.Thread(target=self.removeTables, daemon=True)

        thread.start()

        call\_command('runserver', '--noreload', addrport)

    def removeTables(self):

        while True:

            tables = Table.objects.all()

            for table in tables:

                timeDiff = datetime.now(timezone.utc) - table.lastUsed

                timeDiff = timeDiff.total\_seconds()/60

                if timeDiff > 15 and table.getNoOfPlayers() == 0:

                    print('deleting %s, not used for: %d minutes' % (table.name, timeDiff))

                    table.delete()

            time.sleep(10)

### templatetags

#### tags.py

from django import template

import re

register = template.Library()

@register.simple\_tag

def active(request, pattern):

    stringPattern = '^' + pattern + '$'

    if re.search(stringPattern, request.path):

        return 'nav-item active'

    return 'nav-item'

## templates

### base.html

<head>

  <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

</head>

{% load tags %}

{% url 'index' as index %}

{% url 'leaderboard' as leaderboard %}

{% url 'profile' username=user.username as profile %}

{% url 'pokerRules' as pokerRules %}

<nav class="navbar navbar-expand-sm bg-dark navbar-dark">

  <div class="container-fluid">

    <ul class="navbar-nav">

        <li class="{% active request index %}">

            <a class="nav-link" href="{{ index }}">Tables</a>

        </li>

        <li class="{% active request leaderboard %}">

            <a class="nav-link" href="{{ leaderboard }}">Leaderboard</a>

        </li>

        <li class="{% active request pokerRules %}">

            <a class="nav-link" href="{{ pokerRules }}">How to play</a>

        </li>

      </ul>

    <ul class="nav navbar-nav navbar-right">

      <li class="{% active request profile %}">

          <a class="nav-link" href="{{ profile }}">{{ user.get\_username }}</a>

      </li>

      <span id="money" class="navbar-text"></span>

    </ul>

  </div>

</nav>

<br />

<script>

  {% if user.is\_authenticated %}

    var webSocket = new WebSocket(

      'ws://' + window.location.host + '/ws/user/' + '{{ user.get\_username }}' + '/');

    webSocket.binaryType = 'arraybuffer';

    webSocket.onmessage = function(e) {

      var data = JSON.parse(e.data);

      var totalMoney = data['money'];

      console.log(totalMoney);

      document.getElementById('money').innerHTML = totalMoney;

    }

    webSocket.onclose = function(e) {

      console.error('Chat socket closed unexpectedly');

    };

  {% endif %}

</script>

{% block pageContent %}{% endblock %}

{% if not user.is\_authenticated %}

  <p class="ml-4">You are not logged in</p>

  <a class="ml-4" href="{% url 'login' %}">login</a>

  <a href="{% url 'signup' %}">signup</a>

{% else %}

  <p><a class="ml-4" href="{% url 'logout' %}">logout</a></p>

{% endif %}

<!--javascript to help it function-->

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js" integrity="sha384-ZMP7rVo3mIykV+2+9J3UJ46jBk0WLaUAdn689aCwoqbBJiSnjAK/l8WvCWPIPm49" crossorigin="anonymous"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js" integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy" crossorigin="anonymous"></script>

### game.html

{% extends "base.html" %}

{% block pageContent %}

<body>

  {% if table.buyIn <= user.money %}

    <div class="container-fluid">

      <div class="row">

        <div class="col-sm-6">

          <p id="dealer">Dealer:</p><br/>

          <p id="pot">Pot: 0</p><br/>

          <p id="money-in-table">Money:</p><br/>

          <p id="hand">Hand: </p><br/>

          <p id="com-cards">Community cards:</p>

          <br/>

          <div id='buttons'>

            <div class="input row">

              <div class="col-md-2">

                <input id="fold" class="btn btn-danger" type="button" value="Fold"/>

              </div>

              <div class="col-md-2">

                <input id="call" class="btn btn-info" type="button" value="Call"/>

              </div>

              <div class="col-md-2">

                <input id="raise-submit" class="btn btn-success" type="button" value="Raise"/>

              </div>

            </div>

            <input id="raise-amount-input" class="form-control" type="text"/>

          </div>

          <br/>

          <textarea readonly class="form-control" id="poker-log" rows="15"></textarea>

        </div>

        <div class="col-sm-6">

          <h3>Chat</h3>

          <div class="input-group">

            <input id="chat-message-input" type="text" maxlength="100" class="form-control"/>

          </div>

          <textarea readonly class="form-control" id="chat-log" rows="15"></textarea><br/>

        </div>

      </div>

    </div>

  {% endif %}

</body>

<script>

  function showButton(button) {

    var buttonToggle = document.getElementById(button);

    buttonToggle.style.display = "block";

  }

  function hideButton(button) {

    var buttonToggle = document.getElementById(button);

    buttonToggle.style.display = "none";

  }

  function submit(input, button) {

    document.querySelector(input).focus();

    document.querySelector(input).onkeyup = function(e) {

      if (e.keyCode === 13) {  // enter

        document.querySelector(button).click();

      }

    };

  }

  console.log('js working');

  var pk = {{ table.pk }};

  hideButton('buttons')

  var pokerSocket = new WebSocket(

    'ws://' + window.location.host +

    '/ws/tables/' + pk + '/');

  pokerSocket.onmessage = function(e) {

    var data = JSON.parse(e.data);

    var message = data['message'];

    if (message === 'It\'s your turn') {

      var putIn = data['putIn'];

      showButton('buttons')

      if (putIn > 0) {

        document.getElementById('call').value = ('Call ' + putIn.toString(10));

        showButton('fold')

      } else {

        document.getElementById('call').value = ('Check')

        hideButton('fold')

      }

      document.querySelector('#poker-log').value = (message + '\n') +

      document.querySelector('#poker-log').value;

    } else if (message !== 'message') {

      hideButton('buttons')

    }

    if (message == 'cards') {

      var hand = data['hand'];

      var comCards = data['comCards'];

      var dealer = data['dealer'];

      var moneyInTable = data['moneyInTable'];

      document.getElementById('hand').innerHTML = ('Hand: ' + hand);

      document.getElementById('com-cards').innerHTML = ('Community cards: ' + comCards);

      document.getElementById('money-in-table').innerHTML = ('Money: ' + moneyInTable);

      document.getElementById('dealer').innerHTML = ('Dealer: ' + dealer)

    } else if (message == 'winner') {

      var showdown = data['showdown'];

      var log = data['log'];

      var pokerLog = document.querySelector('#poker-log');

      pokerLog.value = (log + '\n' + showdown + '\n') +

      pokerLog.value;

      document.getElementById('pot').innerHTML = ('Pot: 0');

    } else if (message == 'message') {

      var text = data['text'];

      var chatLog = document.querySelector('#chat-log');

      chatLog.value += (text + '\n');

      chatLog.scrollTop = chatLog.scrollHeight;

    } else if (message == 'It\'s your turn') {

    } else {

      var pot = data['pot'];

      var pokerLog = document.querySelector('#poker-log');

      document.getElementById('pot').innerHTML = ('Pot: ' + pot.toString(10)); //not sure need toString

      pokerLog.value = (message + '\n') + pokerLog.value;

    }

  };

  pokerSocket.onclose = function(e) {

    pokerSocket.send(JSON.stringify({

      'action': 'fold',

    }));

  };

  submit('#raise-amount-input', '#raise-submit');

  submit('#chat-message-input', '#chat-message-input');

  document.querySelector('#raise-submit').onclick = function(e) {

    var raiseAmountInput = document.querySelector('#raise-amount-input');

    var raiseAmount = raiseAmountInput.value;

    pokerSocket.send(JSON.stringify({

      'action': 'raise',

      'raiseAmount': raiseAmount,

    }));

    raiseAmount.value = '';

  };

  document.querySelector('#chat-message-input').onclick = function(e) {

    var messageInputDom = document.querySelector('#chat-message-input');

    var message = messageInputDom.value;

    pokerSocket.send(JSON.stringify({

      'action': 'message',

      'message': message

    }));

    messageInputDom.value = '';

  };

  document.querySelector('#fold').onclick = function(e) {

    pokerSocket.send(JSON.stringify({

      'action': 'fold',

    }));

  };

  document.querySelector('#call').onclick = function(e) {

    var raiseAmount = document.querySelector('#call');

    pokerSocket.send(JSON.stringify({

      'action': 'call',

    }));

  };

</script>

{% endblock %}

### how-to-play.html

{% extends "base.html" %}

{% block pageContent %}

  <body>

    <div class="container-fluid">

      <div class="col-sm-10 pt-2">

        <h2>How to play</h2><br>

        <h4>Pre-Game</h4>

        <p>

          One player acts as dealer. The dealer is represented by the player with the dealer button, and moves clockwise each round.

          The game starts with two forced bets by the two players to the dealers left. These are called the small blind (SB) and the big blind (BB).

          The SB is equal to the minimum bet. The BB is twice that of the small blind. After the blinds, play moves into the betting phase.

        </p>

        <br>

        <h4>Play</h4>

        <p>

          Following the blinds, each player receives two cards face down. These are your ‘hole’ cards. The play continues clockwise, where each player following the blinds can either:

        </p>

        <p>

          • Call – Calling matches the highest bet, in this cast the BB. If the highest current bet is 0 and the players calls, it is called a ‘check’.

        </p>

        <p>

          • Raise – A player can raise the bet up to the amount they have left on the table.

        </p>

        <p>

          • Fold – By folding, a player is choosing not to continue betting and withdraws from the round.

        </p>

        <p>

          The betting round continues until every player has put an equal amount of money in the pot or there is only one player left in the round. If a player calls or raises up to all of their money, they are said to have gone ‘all-in’, and a side-pot is created, which contains the pot which equates to the amount of chips the player has put in.

          This means that the other players may continue to bet between themselves with the all-in player only eligible to win chips up to the amount he has put in.

          Once the betting round has completed, three community cards are shown, known as ‘the flop’. Community cards can be used by any player to make the best 5 card hand possible.

          Another betting round then commences where the players have an option to call (or ‘check’ if no bet has been made), raise (or bet if a bet has been made) or fold. When the second betting round is finished, a fourth card is shown, known as ‘the turn’.

          Another betting round takes place, after which the final card is shown, known as ‘the river’ and after the final betting round if there are still two or more players left, the game goes to a showdown.

        </p>

        <br>

        <h4>Showdown</h4>

        <p>

          If a player bets and all other players fold, the player is awarded the pot and is not required to show his cards. However, if there are still two or more players left after the final round of betting, a ‘showdown’ occurs, where the last player to bet is the first to show his hand, unless everyone checks in which case the player to the left of the dealer button is the first to show.

          After the first player has shown his cards, the players must show their cards in a clockwise rotation if they can beat the best players hand to show. Otherwise they can choose not to show, and ‘muck’ their losing hand without showing them.

          If two players have the same hand strength, ‘kickers’ can be used to break ties. These are extra cards that do not add to the rank of the hand.

          For example K-K-10-3-2 would beat K-K-9-8-7 even though they are both pairs of kings, as the 10 would beat the 9. If the best hand is shared by two or more players, the pot is shared between them. If the pot cannot be equally distributed among all the players in the split pot, the odd chips are given precedence to the winners that are first to play.

        </p>

      </div>

    </div>

  </body>

{% endblock %}

### index.html

{% extends "base.html" %}

{% block pageContent %}

<a id="reset-money" href="{% url 'resetMoney' %}" class="btn btn-info ml-4">Reset money</a>

{% if user.is\_authenticated %}

  <a class="btn btn-success ml-4" href="{% url 'tableCreateView' %}">Create table</a><br /><br />

{% endif %}

<div class="row">

  {% for table in tables %}

    <div class="col-md-4">

      <div class="card mb-2 ml-4" id="{{ table.name }}">

        <div class="card-body">

          <h5 class="card-title">{{ table.name }}</h5>

          <p class="card-text">Buyin: {{ table.buyIn }}</p>

          <p class="card-text" id="{{ table.name }}-noOfPlayers">Players: {{ table.getNoOfPlayers }}/{{ table.maxNoOfPlayers }}</p>

          {% if table.buyIn <= user.money %}

            <a href="{% url 'game' pk=table.pk %}"

            class="btn btn-primary">

              Sit Down

            </a>

          {% endif %}

        </div>

      </div>

    </div>

  {% endfor %}

</div>

<script>

  document.getElementById('reset-money').style.display = "none";

  {% if user.is\_authenticated %}

    webSocket.onmessage = function(e) {

      var data = JSON.parse(e.data);

      var totalMoney = data['money'];

      var tables = data['tables'];

      document.getElementById('money').innerHTML = totalMoney;

      showReset(totalMoney);

      for (i=0; i < tables.length; i++) {

        console.log(tables)

        document.getElementById(tables[i].name + '-noOfPlayers').innerHTML =

        'Players: ' + tables[i].noOfPlayers + '/' + tables[i].maxNoOfPlayers;

      }

    }

  {% endif %}

  function showReset(money) {

    var resetButton = document.getElementById('reset-money');

    if (money < 1000) {

      resetButton.style.display = "block";

    } else {

      resetButton.style.display = "none";

    }

  }

</script>

{% endblock %}

### leaderboard.html

{% extends "base.html" %}

{% block pageContent %}

  {% url 'profile' as profile %}

  <table class="table">

    <thead>

      <tr class='clickable-row' data-href='url://'>

        <th scope="col">#</th>

        <th scope="col">Username</th>

        <th scope="col">Money</th>

      </tr>

    </thead>

    <tbody>

      {% for player in users %}

        <tr onclick="window.location.href = '{% url 'profile' username=player.username %}';">

          <th scope="row">{{ forloop.counter }}</th>

          <td>{{ player.username }}</td>

          <td>{{ player.money }}</td>

        </tr>

      {% endfor %}

    </tbody>

  </table>

{% endblock %}

### profile.html

{% extends "base.html" %}

{% block pageContent %}

  {% if user.username == player.username %}

    <h3>Your profile</h3><br />

  {% endif %}

  <p>user: {{ player.username }}</p>

  <p>money: {{ player.money }}</p>

{% endblock %}

### signup.html

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

{% block title %}Sign Up{% endblock %}

{% block pageContent %}

  <h2>Sign up</h2>

  <form method="post">

    {% csrf\_token %}

    {{ form.as\_p }}

    <button type="submit">Sign up</button>

  </form>

{% endblock %}

### table-form.html

{% extends 'base.html' %}

{% load crispy\_forms\_tags %}

{% block pageContent %}

  <form method="post" novalidate>

    {% csrf\_token %}

    {{ form|crispy }}

    <button type="submit" class="btn btn-success">Save table</button>

  </form>

{% endblock %}

### registration

#### login.html

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

{% block pageContent %}

<h2>Login</h2>

<br />

<form method="post">

  {% csrf\_token %}

  {{ form.as\_p }}

  <button type="submit">Login</button>

</form>

{% endblock %}