# 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 have to 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

### 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 %}