# 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.

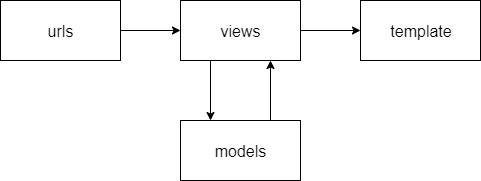
## The Django Framework

### 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:

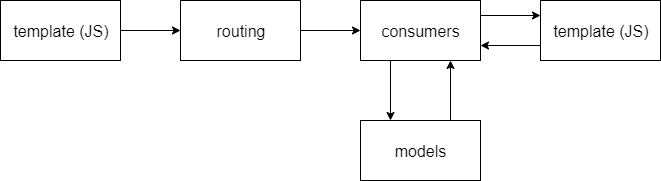
* models.py – Defines your data model and contains the fields and behaviours of the data you are storing. Each model maps to a database table and each attribute to a field. A model is defined in a class and as such this allows Django to interact with DB records like objects. This is extremely useful 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

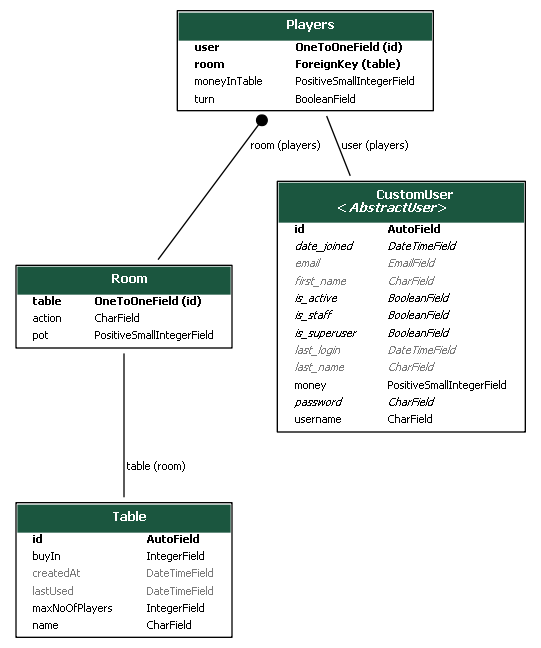
## Database design

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.



## OOP Design

