# Design

TODO:

* OOP diagram showing methods and var
* OOP class hierarchy
* Redo pseudocode
* Redo DB tables
* Add dealer to wireframe

## Overview

I am designing a web-application that allows users to 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.

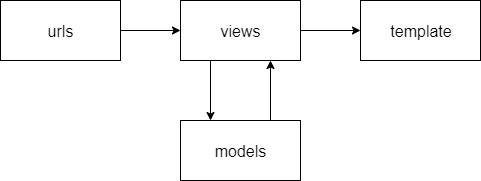
## The Django Framework

### 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:

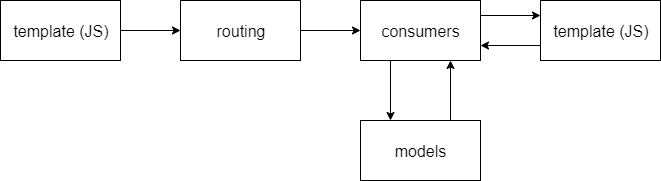
* models.py – Defines your data model and contains the fields and behaviours of the data you are storing. Each model maps to a database table and each attribute to a field. A model is defined in a class and as such this allows Django to interact with DB records like objects. This is extremely useful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

### 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

Function called before the game starts to determine the strength of each players’ hand.

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

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

hand is straight flush with straightHand

if straightHand begins with high Ace

hand is royal flush

else

hand has a straight with straightHand

### Dealing with split pots

If two (or more) players have the exact same strength, they share the pot, and are added to playerWin in a single list.

playerWin <- []

repeated <- []

win <- [players sorted by hand strength]

Function clash, splitWork, winStack

binary sort win

if win items are equal

repeated += both items

for player in repeated, step of 2

if player

### Determining the winner

After the game has finished, the winner function runs to determine how much each player has won.

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 the money

players <- players in table

winners <- players who have the highest hand in the table who have not folded

Function distributeMoney(players, winners, pot)

if there are still winners left

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

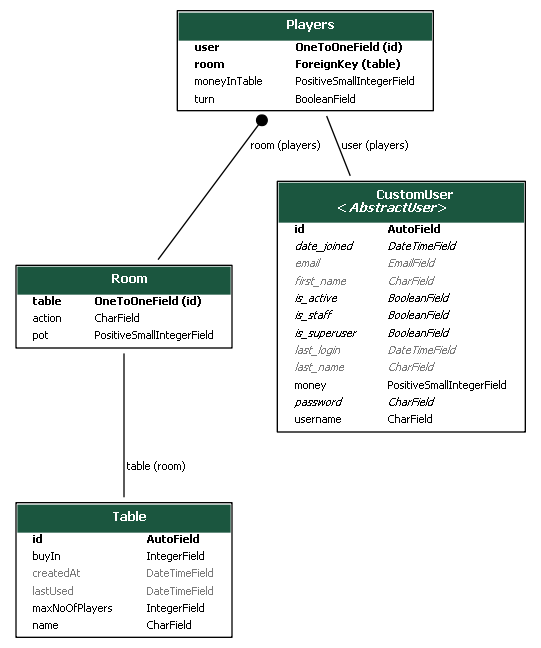
## Database design

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

