Proyecto de fin de curso: Desarrollo web usando Python/Django

Diego Camargo Calvo
2º DAW

Tabla de contenido

1. Introducción:

El proyecto se centrará en la creación de una aplicación web que desarrollará la implementación de una variación del juego de cartas "Cards against humanity", como un juego por turnos en tiempo real multijugador .El desarrollo principal será sobre el juego en sí y su implementación usando python/django para la parte servidor, html/css(bootstrap) para la parte visual de cliente y javascript para el funcionamiento del juego en la parte cliente.

2. Justificación y objetivos del proyecto:

El proyecto surge a raíz de la exploración del las posibilidades que ofrecía la tecnología base escogida para su realización (Django). Como la realización de un juego en tiempo real puede ser algo compleja, el objetivo principal será el desarrollo del juego en sí, mientras que el resto de la página será un desarrollo simple para crear un ejemplo algo más funcional y cercano a una aplicación web real.

3. Definición del proyecto y tecnologías a usar:

3.1. El juego:

El juego es una variación del conocido juego "Cards against humanity" en español. El juego consta de una serie de cartas:
Blancas: Frases,lugares,personas,acciones...
Negras: Afirmaciones o preguntas, en general.
Se reparten 10 cartas blancas a cada jugador, se elige un "Zar" y se saca una carta negra para empezar. El resto de jugadores tienen que elegir una carta que crean que se "relacione" bien con la carta negra que ha salido (una combinación graciosa, con sentido o no) entre las que poseen sin que el resto de jugadores la vean. Cuando todos han elegido una carta,se mezclan y se levantan a la vez, y el zar elige la combinación de cartas (la negra y alguna de las blancas) que más le guste.

A) Ejemplo de carta negra y cartas de un jugador en el juego original.

El jugador ganador gana un punto y pasa a ser el nuevo "Zar". Todos los jugadores que usaron carta roban una carta blanca del montón hasta tener de nuevo 10, se saca otra carta negra y continúa el juego. Éste termina cuando algún jugador llegue a 10 puntos.

3.2. El lado servidor:

Para el desarrollo en el lado servidor se utiliza "Django", un framework de código abierto escrito en Python para desarrollo web. Dentro de Django, se utilizarán para el juego los "canales", una tecnología que permite la comunicación utilizando websockets y funciones de comunicación broadcast y que se encargará de hacer funcionar la parte del juego en el servidor y distribuir el estado del juego al resto de jugadores (se explicará a fondo más adelante).
Para persistencia de datos se usa sqlite al ser una base de datos relativamente ligera, y además se usa Redis para ejecutar algunas de las funciones en tiempo real del juego.

3.3. El lado cliente:

Debido a la naturaleza del proyecto y el tiempo dedicado, el lado cliente se encuentra desarrollado de tal manera que sea meramente funcional en el aspecto visual. La mayor carga de trabajo viene de la ejecución del juego en el lado cliente, para el que utilizamos javascript. El desarrollo del juego se hace comunicándose con el lado servidor mediante sockets, y procesando con javascript la respuesta en el cliente.

4. Tareas a desarrollar:

- Creación del esquema web: páginas que contendrá la aplicación web.
- Desarrollo del lado servidor:
    * Creación del proyecto django.
    * Creación de los templates que servirán de esqueleto a la web.
    * Asociación de urls y views que los manejan. Renderización de templates web.
    * Creación de la estructura de datos de la web (salas de juego y cartas)
    * Creación de un "canal" manejador de los websockets para atender al cliente (desarrollo del juego).

- Desarrollo del lado cliente:
    * Verificación de urls y contenido de la web
    * Creación de un manejador del juego en cliente mediante javascript. Creación del websocket de comunicación del juego con el servidor.

- Valoración del proyecto y posibles mejoras.

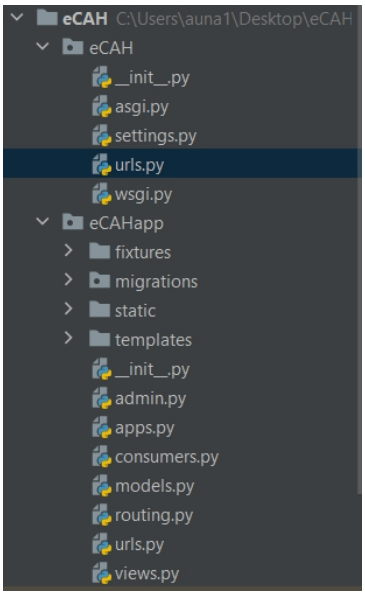
5. Desarrollo del proyecto:

5.1. Desarrollo del lado servidor:

5.1.1. Creación del proyecto Django:

Para el desarrollo de la página web vamos a utilizar Python, en este caso se ha elegido la versión 3.9 por ser de las más nuevas pero siendo estable, además del framework Django para simplificar el desarrollo. Para trabajar con ello, además se ha utilizado el IDE PyCharm Community.
Lo primero es instalar todos los módulos necesarios. A medida que se avance el proyecto se irán necesitando más módulos, de momento necesitamos instalar Django, por lo que usaremos la herramienta pip para hacer un "*pip install django"*.

Una vez instalado, podemos empezar a crear nuestro proyecto, para lo que usaremos el comando "*django-admin startproject eCAH*" , siendo eCAH el nombre del proyecto a crear, y posteriormente crearemos una app para nuestro proyecto django llamada eCAHapp mediante el comando "*python manage.py startapp eCAHapp*". Esto resultará en una estructura de carpetas similar a la siguiente:



Estructura de ficheros del proyecto

A continuación detallaré brevemente las carpetas/ficheros:

* __init__.py: fichero de configuración de arranque del proyecto/aplicación. En nuestro caso no tocamos este fichero ya que no es necesario.
* asgi.py: un fichero de configuración, en nuestro caso es donde indicaremos el fichero que se encargará de procesar las llamadas a url a través del websocket.
* settings.py: fichero de configuración general del proyecto. Lleva todo tipo de configuración: apps instaladas, configuración de bases de datos, localización de ficheros estáticos,etc.
* urls.py: fichero de procesamiento de urls. En este caso hay dos, pero como el proyecto solo tiene una app el urls.py general solo tiene una referencia al urls.py de la app, además de la referencia a la página de administración del proyecto.
  Asocia las urls con las funciones que procesarán la petición, que se encuentran en el fichero views.py.
* wsgi.py: fichero de configuración similar a asgi.py. En este caso no lo utilizaremos, por lo que se deja tal cual.
* fixtures: Carpeta donde se almacenarán ficheros con el que precargaremos las base de datos. En este caso se almacenan las cartas del juego.
* migrations: Carpeta que almacena información sobre los cambios a la base de datos.
* static: Carpeta que almacena los ficheros estáticos (css,javascript,imágenes,etc).
* templates: Carpeta donde se guardan los html que darán forma a la página.
*admin.py: Fichero de administración de la aplicación. Permite modificar la base de datos y la información visible de ella.
* apps.py: Fichero que guarda el nombre de la app. No se modificará en este proyecto.
*consumers.py: Fichero que contiene las funciones que se encargar de tratar con las peticiones a través de websockets.
*models.py: Fichero que contiene información del modelado de los objetos de la base de datos.

* routing.py: Es el equivalente a urls.py pero para llamadas a websockets.
* views.py: Fichero que se encarga del procesado de las llamadas a url en el servidor, así como de devolver la respuesta y el renderizado de la página a mostrar.

5.1.2. Creación de las templates de la web:

En este punto no se ahondará mucho, ya que se hablará más en profundidad en el desarrollo de la parte cliente. Destacar aquí que Django permite la reutilización de código en las templates, por lo que se ha creado un fichero llamadado "*squeleton.html*", en el que se ha creado una base con el contenido html que se repite en todas las pantallas de la web: la cabecera con la barra de navegación y el footer. Para ello utilizamos el siguiente código:

{% block content %}{% endblock %}

Éste código permite poder reutilizar el fichero en otros, haciendo que solo sea necesario añadir el cuerpo de la página mediante el uso del comando extends acompañado de lo anterior:

{% extends "squeleton.html" %}
*{% block content %}*Tu código nuevo aquí*{% endblock %}*

```
<!DOCTYPE html>
{% load static %}
{% load bootstrap4 %}
{% bootstrap_css %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <link rel="stylesheet" type="text/css" href="{% static 'css/squeleton.css' %}"/>
    <link rel="stylesheet" type="text/css" href="{% static 'css/index.css' %}"/>
        <script src="{% static 'js/jquery-3.5.1.min.js' %}"></script>
    <script src="{% static 'js/nick_check.js' %}"></script>
    <script src="{% static 'js/websocket.js' %}"></script>
</head>
<body>
<nav class="bg-dark sticky-top py-1">
  <div class="container d-flex flex-column flex-md-row justify-content-between">
     <a class="py-2 text-white text-decoration-none" href="#"><img src="{% static 'img/logo.svg' %} " width="30" height="30" class="bg-light">
         CARTAS CONTRA LA HISPANIDAD</a>
    <a class="py-2 d-none d-md-inline-block text-white" href="/acerca_del_juego">Sobre el juego</a>
    <a class="py-2 d-none d-md-inline-block text-white" href="/salas">Juega ahora</a>
    <a class="py-2 d-none d-md-inline-block text-white" href="/contacto">Contacto</a>
  </div>
</nav>

{% block content %}{% endblock %}

<footer class="myfooter bg-dark"><div class="container text-center">Cartas contra la humanidad - 2021</div></footer>

</body>
</html>
```

Código del esqueleto

5.1.3.Relación Urls - views:

Para el realizar la navegación web, django crea una relación entre las urls por las que se puede navegar y unas funciones que se encargan de su procesamiento. Para ello se crea un fichero "urls.py". En este proyecto tenemos los siguientes enlaces:

```
urlpatterns = [
    path('', views.index, name='index'),
    path('salas/', views.lobby, name='lobby'),
    path('salas/nuevasala/', views.new_room, name='newroom'),
    path('salas/verificanick/', views.nickcheck, name='nickCheck'),
    path('salas/<str:room_name>/', views.created_room, name='room'),
    path('acerca_del_juego', views.about, name='about'),
    path('contacto', views.contact, name='contact'),
]
```

Urls del proyecto

Como se puede apreciar, cada uno de las url tiene asociado una función nominada por views.nombre_función. Estas funciones se encuentran en el fichero views.py y van desde simplemente renderizar el template correspondiente a una serie de procesados antes de mostrar la página.

```
def index(request):
    return render(request, 'index.html')


def about(request):
    return render(request, 'about.html')


def lobby(request, fullroom='false'):
    Player.objects.all().delete()
    Room.objects.all().delete()
    return render(request, 'lobby.html', {
        'fullroom': fullroom
    })


def contact(request):
    return render(request, 'contact.html')


def nickcheck(request):
    username = request.GET.get('username', None)
    data = {
        'is_taken': Player.objects.filter(nickname__iexact=username).exists()
    }
    return JsonResponse(data)
```

Parte del fichero views.py

Nótese que algunos solo devuelven el render (es decir, solo muestra el fichero html), mientras que otros como el del lobby tiene cierto procesamiento, en este caso modifica la base de datos para borrar a los jugadores de una sala (hablaré de la estructura de datos en el punto siguiente).
Por último la función nickcheck no muestra una página nueva, si no que verifica en base de datos si el nick existe y devuelve a la misma página la respuesta en formato JSON.

5.1.4. Estructuras de datos

En la persistencia de datos solo tenemos 3 elementos: salas, cartas y jugadores. Las salas tienen un nombre y un estado (esperando inicio,jugando,iniciado pero faltan jugadores y juego terminado), las cartas tienen nombre y si son o no negras y los jugadores tiene un nick y una sala asociada. Esto queda definido a través del fichero models.py, en el que se describe el modelo de datos con el que se creará la base de datos posteriormente:

```
class Room(models.Model):
    WAITING_START = 'WAITING_START'
    PLAYING = 'PLAYING'
    STARTED_BUT_NEED_PLAYERS = 'STARTED_BUT_NEED_PLAYERS'
    GAME_FINISH = 'GAME_FINISH'

    GAME_STATUS = [
        (WAITING_START, 'WAITING_START'),
        (PLAYING, 'PLAYING'),
        (STARTED_BUT_NEED_PLAYERS, 'STARTED_BUT_NEED_PLAYERS'),
        (GAME_FINISH, 'GAME_FINISH')
    ]

    name = models.TextField()
    label = models.SlugField(unique=True)
    game_status = models.CharField(choices=GAME_STATUS, default=WAITING_START, max_length=50)

    def __unicode__(self):
        return self.label


class Card(models.Model):
    text = models.TextField()
    is_black = models.BooleanField()


class Player(models.Model):
    room = models.ForeignKey(Room, related_name='players', on_delete=models.CASCADE)
    nickname = models.TextField()
```

Definición del modelo de datos en models.py

Tanto los jugadores como las salas se irán creando a medida que usuarios vayan usando la aplicación, mientras que las cartas son precargadas de un fichero json al arrancar.

5.1.5. Manejo del juego.Canales y websockets (lado servidor):

Para poder trabajar con canales y conexión a través de websockets, primero tenemos que configurar dos cosas: las url que se encargará de manejar la conexión del websocket al iniciar un juego desde el lado cliente, para lo que configuraremos el fichero asgi.py para que indique en qué fichero se encuentra el matcheo de urls:

```
application = ProtocolTypeRouter({
  "http": get_asgi_application(),
  "websocket": AuthMiddlewareStack(
        URLRouter(
            eCAHapp.routing.websocket_urlpatterns
        )
    ),
})
```

asgy.py refenciando al fichero de enrutamiento del websocket

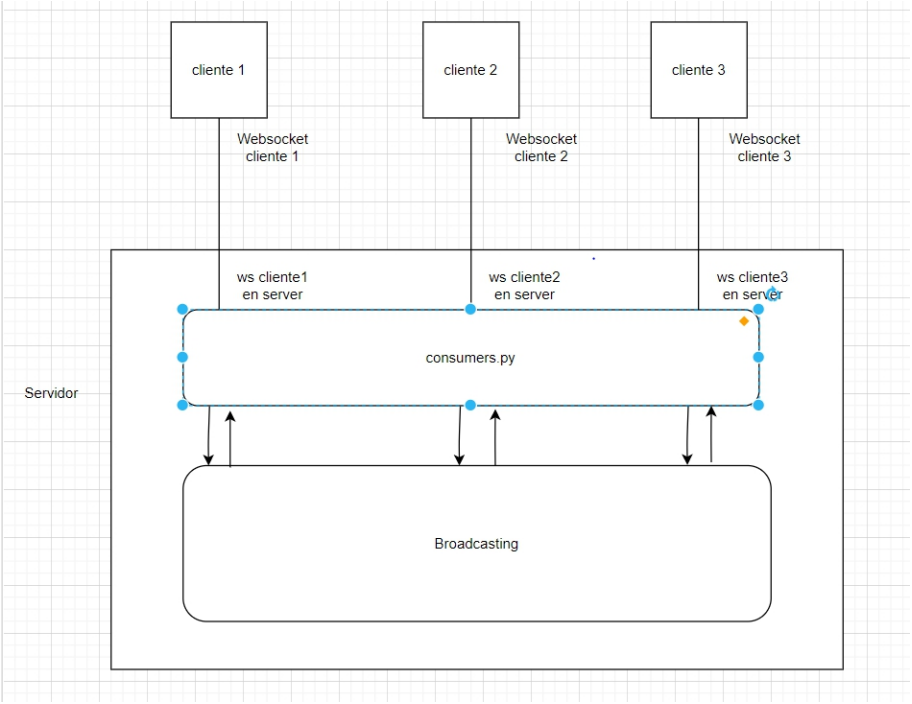
Una vez indicado de donde hara la comprobación de urls, se crea un fichero routing.py, que contendrá las urls que atenderá el websocket y que funciones procesarán estas peticiones, de manera equivalente a la relación entre urls.py/views.py mencionada anteriormente:

```
websocket_urlpatterns = [
    re_path(r'ws/salas/(?P<room_name>\w+)/$', consumers.ChatConsumer.as_asgi()),
]
```

Indicamos el consumer, que se encargará de procesar lo que venga a la url de websocket indicada

Una vez configurado todo, podemos proceder a crear el fichero consumer.py, que se encargará de procesar las peticiones a través del websocket. En este punto conviene explicar los "canales" que serán los que se encargan del procesamiento en conjunto con los websockets.

Cuando un jugador entra en una sala, abre un websocket en el cliente y lanza una petición al servidor, que a su vez recibe esta petición y crea un canal, que será único para los miembros de la sala. Este canal funciona como un mailbox para todos los usuarios de la sala, y se encargará de informar de los cambios efectuados por uno de sus miembros a los demás cuando sea necesario. Cuando uno de los miembros sale de la sala se elimina su conexión con el canal.



Funcionamiento simplificado de un canal.

Para el procesamiento del juego tenemos una serie de funciones, que se encargarán de atender el flujo del mismo a lo largo del tiempo.

Tenemos 3 funciones básicas, que son las de conexión, desconexión y recepción del mensaje. Las primeras se encargan de agregar/eliminar a los usuarios del grupo del canal, y la tercera es un manejador que llama a otras funciones dependiendo del tipo de mensaje recibido.

```python
def connect(self):

    self.room_name = self.scope['url_route']['kwargs']['room_name']
    self.room_group_name = 'chat_%s' % self.room_name

    # Join room group
    async_to_sync(self.channel_layer.group_add)(
        self.room_group_name,
        self.channel_name
    )
    self.accept()

def disconnect(self, close_code):
    self.player_disconnect()
    # Leave room group
    async_to_sync(self.channel_layer.group_discard)(
        self.room_group_name,
        self.channel_name
    )
```

Creación/Eliminación de usuarios de un canal

El manejador por lo general tiene dos funciones, primero atender la petición del cliente y después informar al resto de usuarios del canal de lo ocurrido (aunque esto no es siempre necesario). Tenemos eventos al conectarse un nuevo usuario a nuestro grupo, al desconectarse, al empezar el juego, al jugar una carta, al final de cada ronda y al finalizar el juego. Cada uno de estos eventos se notifican a todos los usuarios para que el cliente actualice el juego.

```
def receive(self, text_data):
    text_data_json = json.loads(text_data)
    event_type = self.event_switch(text_data_json['event_type'])

    getattr(self, event_type)(text_data_json)

    # Send message to room group
    async_to_sync(self.channel_layer.group_send)(
        self.room_group_name,
        {
            'type': event_type + '_notify',
            'info': text_data_json
        }
    )

def player_connect(self, info):
    self.nickname = info['player_name']
    this_room = Room.objects.get(label=self.room_name)
    this_room.players.create(room=this_room, nickname=self.nickname)
def player_connect_notify(self, event):
    this_room = Room.objects.get(label=self.room_name)
    players_list = []
    for player in Player.objects.filter(room=this_room):
        players_list.append(player.nickname)

    self.send(text_data=json.dumps({
        'event_type': 'player_connect',
        'player_name': event['info']['player_name'],
        'nplayers': Player.objects.filter(room=this_room).count(),
        'players_list': players_list
    }))
```

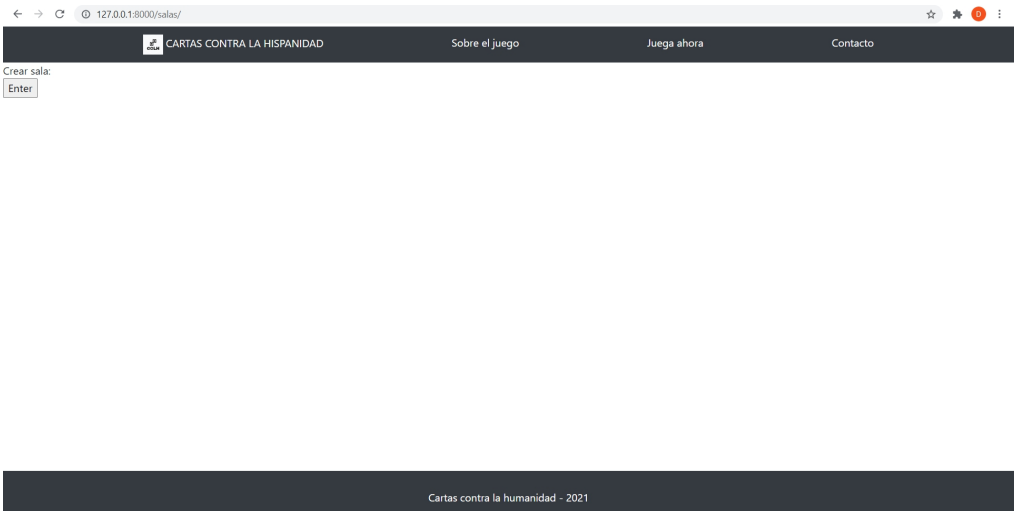
El dispatcher y el evento "nuevo jugador", con su respectiva notificación al resto

5.2. Desarrollo del lado cliente:

5.2.1. Verificación de urls y contenido de la web:

Después del desarrollo web del lado servidor, toca comprobar que se ha realizado correctamente el enrutamiento. En nuestra web contamos con 4 páginas: inicio, sobre el juego, jugar y contacto. Se puede acceder a cualquiera de las páginas a través de la barra de navegación.

Pantalla principal de la web



Salas

Acerca del juego

Para el desarrollo de la web del lado cliente se ha simplificado a lo mínimo para que el resultado sea funcional, ya que la carga del juego es bastante alta y en un proyecto corto acabaría alargándose demasiado con una carga alta de css y funcionalidad. En este caso se ha utilizado bootstrap de manera básica, tanto para la creación de la barra de navegación y el footer, el formateo del contenido y la creación de las cartas (de manera visual) del juego.

El nombre de la web es una variación del nombre del juego original “Cards against humanity (CAH)” derivandolo a cartas contra la hispanidad (CCLH) por tener un nombre algo más llamativo en español. El icono y los colores de la web van de acuerdo a la temática del juego, siendo sus colores principales el blanco (cartas de juego) y el negro (cartas de pregunta o afirmación).

La navegación de la web queda de la siguiente manera:

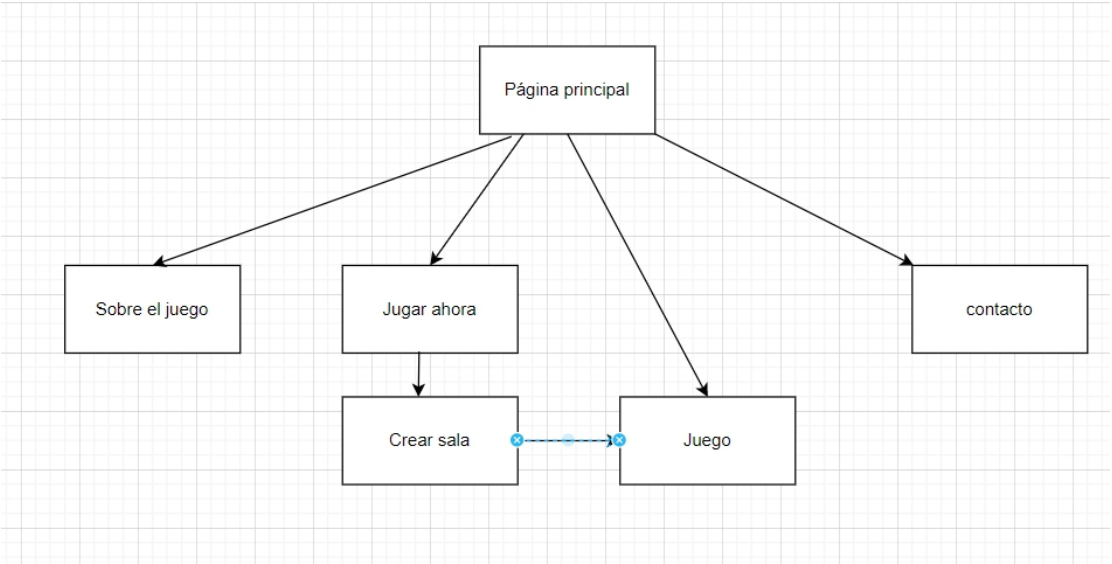


Diagrama de navegación de la web

Para jugar, el usuario puede tanto crear una sala como unirse a una sala ya creada. Esto último se puede hacer tanto a través del lobby (seleccionando la sala) o a través del enlace directo a dicha sala.

La unión de un nuevo usuario se puede ver en el siguiente caso de uso:

Funcionalidad	Crear sala nueva
Papel en el trabajo	Permite a un usuario crea una nueva sala de juegos a la espera de más participantes
Actores	Usuario
Precondiciones	Que no se haya alcanzado el máximo de salas
Postcondición	El usuario crea una nueva sala de juegos
Proceso	El usuario accede al lobby mediante el botón “jugar ahora”. Una vez dentro hace click en “enter” a crear sala. Se le pide al usuario que introduzca un nick y se crea la sala en espera de más usuarios.
Alternativas	El usuario intenta crear una sala cuando se ha llegado al máximo. Se devuelve un mensaje de error y se vuelve al lobby.

Si lo que se desea es unirse a una sala nueva tenemos el siguiente caso:

Funcionalidad	Unirse a sala
Papel en el trabajo	Permite a un usuario unirse a una sala ya existente.
Actores	Usuario
Precondiciones	Que exista una sala. Si se accede por enlace esta sala debe existir.
Postcondición	El usuario se une a sala de juegos
Proceso	El usuario accede al lobby mediante el botón “jugar ahora”. Una vez dentro elige una sala a la que unirse. Se le pide al usuario que introduzca un nick y se crea la sala en espera de más usuarios.
Alternativas	El usuario accede a una sala a través de un enlace dado a una sala. Esta sala debe existir, si no se devuelve un error y se redirecciona al lobby.

5.2.2. Desarrollo del juego en el lado cliente:

Una vez creado una sala, se verifica que el nick sea válido y se procede a crear un websocket que conecte con el servidor (se unirá a la sala del lado servidor ,como se ha explicado anteriormente). Una vez establecida la conexión, el cliente tiene un switch similar al desarrollado en el consumer.py del lado servidor. Este se encarga de enviar mediante el websocket el evento que se ha producido (si ha clicado en comenzar juego, si ha elegido una carta para jugar, si se ha desconectado,etc.). Posteriormente el websocket se encarga de recibir la respuesta del servidor, ya sea único para nosotros (por ejemplo, darnos una carta nueva si hemos jugado una) o un broadcast, ya sea por la acción de otro jugador o porque se ha producido un evento que afecta a todos los jugadores (otro jugador se ha conectado/desconectado, ha jugado una carta, se ha elegido al ganador de la ronda o el final del juego con su correspondiente ganador.

Todo lo enunciado anteriormente se hace mediante javascript, y en la mayoría de casos se hace un send a través del websocket al servidor con una serie de información compactada en JSON, dependiendo de la acción a realizar.

```
document.querySelector('#submit').onclick = function(e) {
    nickname = document.querySelector('#new_nick').value;

    if(nick_is_taken(nickname)){
        alert("nick is taken");
    }else{
        document.querySelector('#nick-input').hidden=true;
        document.querySelector('#game').hidden=false;

        mywebsocket.send(JSON.stringify({
            'event_type': 'player_connect',
            'player_name': nickname,
            'room_name': roomName,
        }));
    }
}
```

Ejemplo de comprobación del nick, si es correcto se notifica al resto de jugadores del nuevo jugador.

6. Conclusiones y posibles mejoras:

Como conclusión podría decir que he aprendido las grandes posibilidades que ofrecen las aplicaciones Django y en general cualquier desarrollo en Python, tanto por su simpleza a la hora de desarrollar como por su potencia y versatilidad. El proyecto ha sido más complejo de desarrollar de lo que esperaba, aunque cabría esperarlo ya que el funcionamiento en tiempo real requiere un checkeo constante en el cliente y su acompasamiento con las respuestas del servidor, de tal manera que la respuesta siempre sea uniforme a todos los clientes.

Como posibles mejoras podría empezar con aumentar la complejidad de la web en sí para hacerla más realista, ya que más allá del juego no deja de ser una web bastante simple.También se podría añadir a usuarios registrados para no tener que introducir el nick cada vez que se quiera jugar, y poder registrar cosas asociadas a la cuenta, como listas de amigos, set de cartas personalizadas, etc. También es muy mejorable el desarrollo en la parte cliente, ya que es bastante limitada y la codificación no es todo lo ordenada que me gustaría (se podría dividir el código en distintos ficheros javascript separando la funcionalidad, en lugar de poner todo junto).

Para finalizar, ha sido un proyecto muy interesante de desarrollar, y he aprendido mucho de cosas que apenas había tocado durante el curso escolar (Django, manejo de juegos en tiempo real, manejo de websockets, etc).

7. Bibliografía:

https://channels.readthedocs.io/en/stable/tutorial/index.html

https://simpleisbetterthancomplex.com/tutorial/2016/08/29/how-to-work-with-ajax-request-with-django.html

https://blog.heroku.com/in_deep_with_django_channels_the_future_of_real_time_apps_in_django

https://developer.mozilla.org/es/docs/Web/JavaScript

https://getbootstrap.com/docs/4.0/getting-started/introduction/

https://websockets.readthedocs.io/en/stable/intro.html