

Proyecto de multimedios Dudo!

Juan Mucarquer
2830027-1

`<juan.mucarquer@alumnos.usm.cl>`

Sebastian Cáceres
2830010-7

`<sebastian.caceresb@alumnos.usm.cl>`

Victor Fernandez
2604041-8

`<victor.fernandez@alumnos.usm.cl>`

26 de noviembre de 2012



Índice

1. Introducción	3
1.1. Trasfondo	3
1.2. Resumen	3
2. Descripción General	4
2.1. Objetivos	4
2.1.1. General	4
2.1.2. Especificos	4
2.2. Problemática que enfrenta	4
2.3. Descripción de la solución	4
2.4. Tecnologías utilizadas	4
3. Especificación de los requerimientos	6
3.1. Reglas del juego	6
3.2. Usuarios del sistema	6
3.3. Descripción de los requerimientos	7
3.3.1. Funcionales	7
3.3.2. No funcionales	7
3.4. Tareas de usuario	9
3.5. Funciones del sistema	10
4. Diseño de la interfaz de usuario	11
4.1. Esquema navegacional	11
4.2. Vistas	12
5. Diseño del sistema	16
5.1. Diseño de la arquitectura	16
5.2. Diseño lógico	17
5.2.1. Diagrama de casos de uso	17
5.2.2. Modelo de datos	17
5.2.3. Diagrama de clases	17
5.2.4. Diagrama de secuencia	18
5.2.5. Diagrama de flujo de la lógica del juego	18
6. Implementación	19
6.1. Descripción de componentes	19
6.2. Aplicación del juego	20
6.2.1. NoSQL Redis	23
6.2.2. Modelo de datos	24
6.3. Cliente Javascript	25
6.4. Notas	26
7. Conclusiones	26
A. Figuras	27
B. Referencias	29

1. Introducción

1.1. Trasfondo

En la actualidad, el desarrollo de aplicaciones web es muy popular. La mayoría de la gente tiene acceso a internet, y el mercado es muy prometedor. Existen cientos (o miles) de aplicaciones con distintos fines, muchas muy buenas. Otras, quizás no tanto.

Un subconjunto de estas aplicaciones son los juegos online. Estos juegos se han vuelto muy populares últimamente, probablemente debido a la capacidad que poseen para manejar partidas multijugadores.

Esto es muy atractivo, y existe una gran variedad de juegos que disponen de esta característica. Sin embargo, por lo general son solo típicos juegos de cartas. El enfoque que tomaremos será hacer algo distinto, implementar un juego de mesa que no sea de cartas, pero que tenga características similares a las descritas anteriormente.

El juego que implementaremos será el dudo.

1.2. Resumen

El proyecto descrito en este informe será la implementación de un juego basado en la arquitectura cliente-servidor. Este juego será el típico juego latinoamericano de especulación, el dudo.

Éste juego será multijugador, de manera de que puedan jugar múltiples personas en red, ubicadas en distintos lugares físicos, pero compartiendo el mismo contexto de la partida. Éstas partidas de desarrollarán en salas de juego independientes unas de otras, y cada una tendrá un ganador.

Dentro de cada sala habrá un chat para que los usuarios puedan comunicarse entre ellos, libremente. La implementación también ofrecerá un sistema de usuarios. Para jugar será necesario registrarse. Por último, la dinámica del juego está regida por reglas detalladas en otra sección dentro de este mismo informe

2. Descripción General

2.1. Objetivos

2.1.1. General

El objetivo principal de la aplicación es ofrecer una plataforma para un juego popular chileno, en línea, activo y rápido. La lógica del juego es de relativa sencillez y las reglas son ampliamente conocidas, por lo que pretende que el juego sea fácil e intuitivo para el usuario.

2.1.2. Específicos

- Promover el juego del dudo.
- Disponer de un juego rápido e instantáneo
- Utilizar tecnologías que permitan su jugabilidad de forma nativa, no es necesario descargar nada para empezar a jugar.
- Contar con un chat y salas de juego.
- Establecer un protocolo de comunicación entre el cliente y el servidor (definición de mensajes).

2.2. Problemática que enfrenta

La problemática que enfrenta el juego es, por un lado, el aburrimiento de las pobres personas. La gente necesita un juego bien codificado y funcional que les brinde un alto nivel de entretenimiento.

Por otro lado, si bien existen bastantes juegos online en la actualidad, creemos que no existe ninguno con el cual el usuario se pueda identificar, debido a que ninguno de los que existe ahora es propiamente latinoamericano.

2.3. Descripción de la solución

La solución para ambas problemáticas es implementar el dudo multijugador online. Éste está dividido en dos partes: el cliente y el servidor.

En el lado del cliente, existirá un sistema de login, seguido de un menú de selección de salas, el cual mostrará las salas disponibles y los usuarios que están en cada sala. El cliente entonces podrá elegir una sala para jugar su partida, la cual también contará con un chat en el cual participarán los jugadores que comparten una partida.

El servidor se encargará del paso de mensajes asíncrono, y de las interacciones con la base de datos. Con esto, el servidor definirá la estructura del juego, y su secuencia. Esto se refiere a manejo de turnos, de interacciones entre usuarios, de mensajes de chat, de usuarios dentro de la sala, de datos disponibles por usuario y su aleatorización, etc.

2.4. Tecnologías utilizadas

Para la parte del servidor, constará de una aplicación MVC, utilizando el framework Django (Python). Con el fin de tener una comunicación asíncrona y activa entre el cliente y el servidor, se utilizará un binding de la biblioteca Socket.IO (Javascript), construido sobre event, una biblioteca basada en co-rutinas para aplicaciones de red, llamado event-socketio. Como

Key-value store se utilizó la base de datos NoSQL Redis

En el cliente se utilizará HTML5 y CSS para el diseño, y para la comunicación, Javascript con la biblioteca Socket.IO, basada a su vez en node.js. En específico, para la interfaz gráfica, se utilizará Twitter Bootstrap, la cual es una colección de herramientas para la creación de diseños web. Contiene plantillas HTML y CSS, y interacciones con javascript para un diseño eficiente.

Se posee un repositorio git para el control de versiones, hospedado en github.com. Este informe también se encuentra disponible en el repositorio. Se puede acceder a él mediante el código QR disponible en la portada del informe, o accediendo a <https://github.com/jamonardo/cacho>.

3. Especificación de los requerimientos

Para especificar los requerimientos, primero es necesario establecer las reglas del juego.

3.1. Reglas del juego

El juego parte cuando hay más de un jugador listo para comenzar. Cada jugador parte con un total de 5 dados dentro de su *cacho*. Las rondas del juego se realizan en sentido antihorario.

Al comienzo de cada *ronda*, los jugadores agitan los dados del *cacho*, es decir, reciben aleatoriamente los valores contenidos en sus dados. Sólo ellos son capaces de ver el contenido de su *cacho*, no así los demás.

En base a el contenido de su *cacho*, cada jugador especula el resultado de la suma de todos los dados cara arriba - con un determinado número - obtenidos en total por todos los jugadores. Los ases, aparte de ser números concretos, con comodines, y pueden obtener el valor del número que se esta contando.

El jugador que parte *canta* un número estimado de dados. (ej: "Hay tres quinas").

El jugador siguiente tiene dos opciones: elevar el número especulado, o dudar.

- Para elevar el número, existen tres maneras posibles. Una es aumentar la "*pinta*" del dado (ej: De tres quinas a tres sextas). Otra es aumentar el número de dados totales en la especulación (ej: De tres quinas a cuatro quinas). La última manera es una combinación de las dos maneras anteriores.
- Si el jugador duda, todos los jugadores muestran los dados contenidos en sus cachos, y estos se cuentan. Si la especulación era correcta, es decir, hay sobre la mesa el número total de dados cantados por el primer jugador, el jugador que dudó pierde un dado. En el caso contrario, el jugador que cantó, pierde un dado.

El jugador que comienza cantando la siguiente ronda es el que pierde.

Cuando a un jugador le queda un sólo dado, el jugador **obliga**. En este caso, y sólo para esta ronda, sólo él puede ver el contenido de su *cacho*, nadie mas. Una vez que el jugador cante, nadie puede cambiar la *pinta* cantada, y los ases dejan de ser comodines.

El juego se termina cuando sólo queda un jugador con dados en su *cacho*, y es él quien gana.

3.2. Usuarios del sistema

Administrador

El administrador es el encargado de crear y regular los espacios y las características de la sala. Dentro de estos se incluyen la creación de salas, usuarios, etc.

Clientes

Serán los principales usuarios del sistema. Los clientes acceden al sistema, seleccionan salas, y juegan partidas.

3.3. Descripción de los requerimientos

3.3.1. Funcionales

Requisito 1

Implementar login y registro de usuarios.

Requisito 2

Implementar login y panel de control de administrador.

Requisito 3

Lista de salas de juego disponibles.

Requisito 4

Lista de usuarios dentro de salas de juego.

Requisito 5

Chat dentro de la sala de juego.

Requisito 6

Botón confirmar inicio del juego, el cual es necesario que pulsen todos los jugadores dentro de una sala para comenzar la partida.

Requisito 7

Implementación de lógica de juego del lado del servidor.

Requisito 8

Display con dados disponibles por usuario, y dados obtenidos en cada turno.

Requisito 9

Determinar el ganador de la partida, y boton de salida de la sala dinámico, el cual aparece una vez que la partida termine.

Requisito 10

Display que diga quien está jugando actualmente.

3.3.2. No funcionales

Sencillez de uso

El uso del sistema y de sus componentes debe ser intuitivo, lo mas sencillo posible. Las opciones para el usuario deben estar acordes a estos principios. Para esto, se diseñará la interfaz gráfica de la manera mas sencilla que se pueda, no utilizando interacciones complejas, y haciendo que las opciones disponibles lo más evidentes y sencillas posibles.

Fluidez

Por otra parte, el juego debe funcionar lo más fluidamente posible. Esto significa que toda la interacción del usuario, desde su login, mientras dure la partida y su finalización, deben transcurrir dentro de lo posible sin retardos ni cortes. Para esto, se utilizará gevent-socketio, enfatizando que las interacciones con dependencia de otros componentes sean lo más breves posibles, de manera que los tiempos de respuesta sean mínimos.

Escalabilidad

El sistema debe ser modular en su construcción, permitiendo así añadir futuras funcionalidades al juego que puedan mejorar su calidad. Es importante que se pueda hacer esto sin afectar la codificación hecha anteriormente. Para esto, se diseñará modularmente, haciendo que cada parte del código sea un módulo, y permitiendo la posibilidad de integrar varios más en el caso de que se desee agregar mas componentes.

3.4. Tareas de usuario

Administrador

La tarea del administrador es velar por el correcto orden esquemático del sitio. El estará encargado de crear salas, administrarlas o borrarlas, dependiendo del caso. También tiene las mismas responsabilidades con respecto a los usuarios. Al final de cuentas, es un usuario que interactúa directamente con la base de datos.

Clientes

La tarea principal (y única) de los clientes será jugar partidas. Ellos ingresarán a el juego, logeandose. Luego buscarán una sala disponible. Al ingresar a una sala disponible, podrán salirse de ella, o confirmar su participación para el inicio del juego. Una vez terminada la partida, el usuario se sale de la sala, y vuelve a escoger otra sala disponible para jugar, o derechamente se sale del sistema.

3.5. Funciones del sistema

Las funciones del sistema serán:

Servidor:

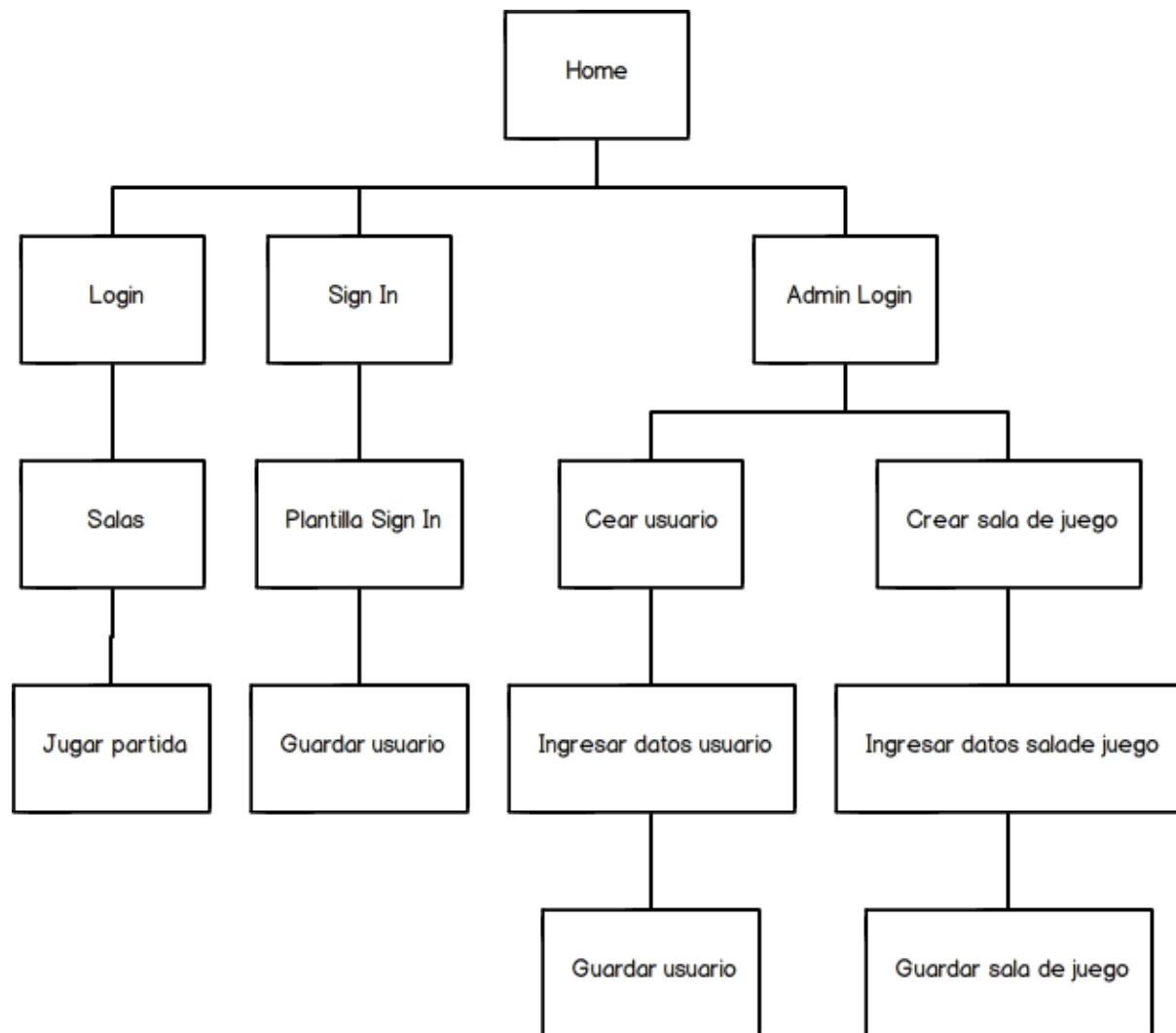
- Manejar las sesiones de los usuarios, entrando mediante un formulario de login y permitir el registro de usuarios, usando un formulario de registro.
- Derivar las peticiones hechas por HTTP a la vista correspondiente (urls.py)
- Formatear la informacion para cada vista.
- Mostrar una lista de salas de juego con los usuarios dentro de cada sala y el estado de la sala (jugando o esperando).
- Derivar las conexiones y mensajes hechas mediante el protocolo de Socket.IO hacia un modulo que se encargará de responder a estos mensajes (cacho_socketio.py)
- Inicializar y responder a los mensajes enviados por los clientes, según un protocolo de juego, descrito posteriormente en el diagrama de secuencia.
- Mantener e interactuar con el modelo de datos para las operaciones que lo requieran.

Cliente:

- Registrarse con un nombre de usuario y contraseña
- Unirse a una sala de juego que actualmente esté esperando jugadores.
- Enviar un mensaje a una sala de juego
- Confirmar el inicio del juego por parte de un usuario.
- Recibir y mostrar los dados para el jugador
- Enviar una jugada, un dudo o calzo.
- Salir de la sala una vez terminada la partida.

4. Diseño de la interfaz de usuario

4.1. Esquema navegacional



4.2. Vistas

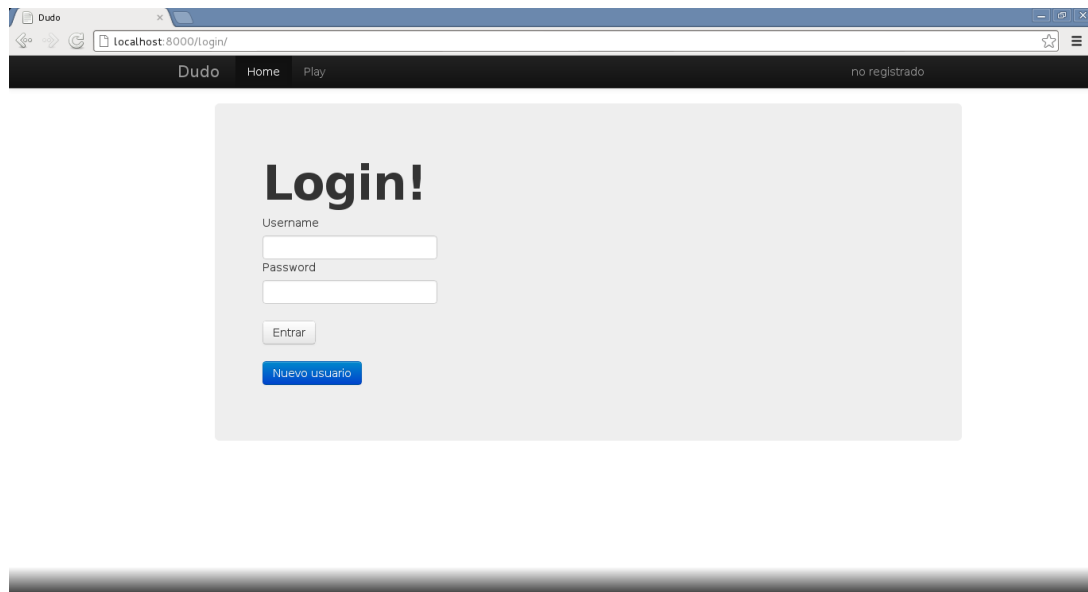


Figura 1: Login

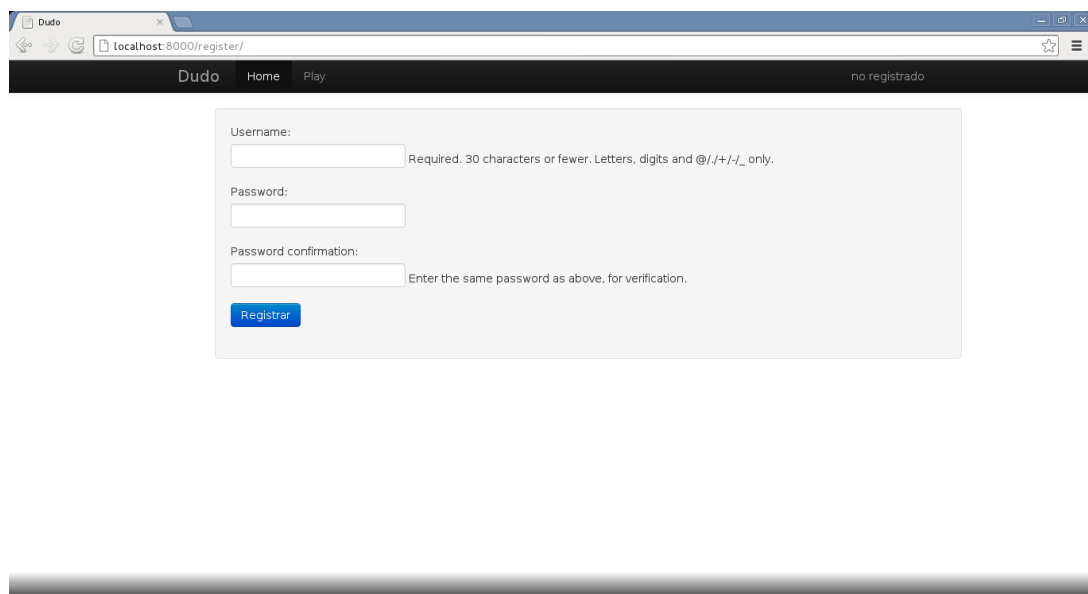


Figura 2: Sign in

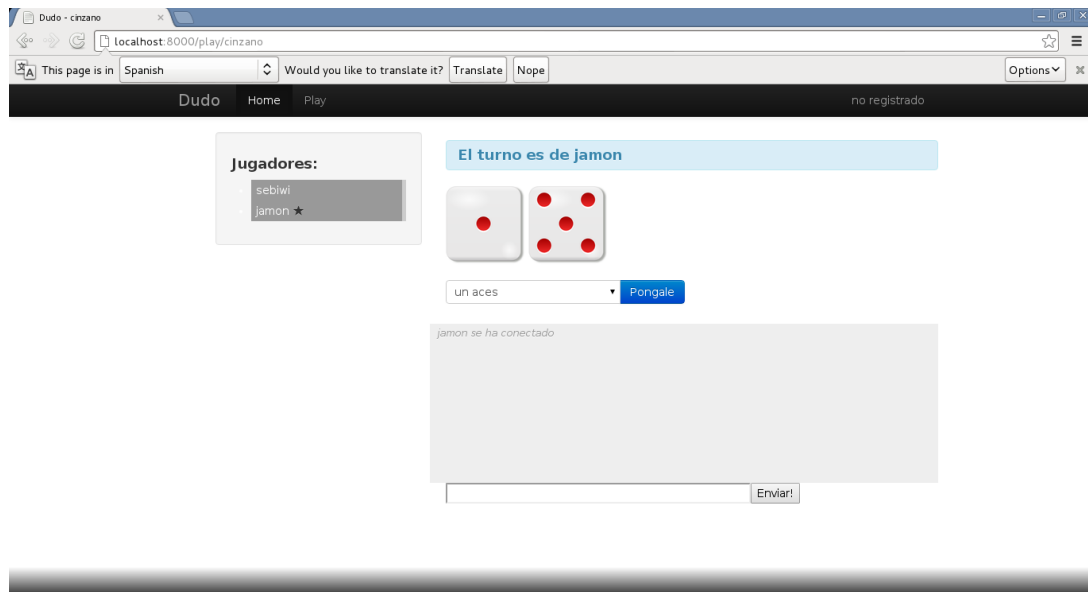


Figura 3: Sala de juego

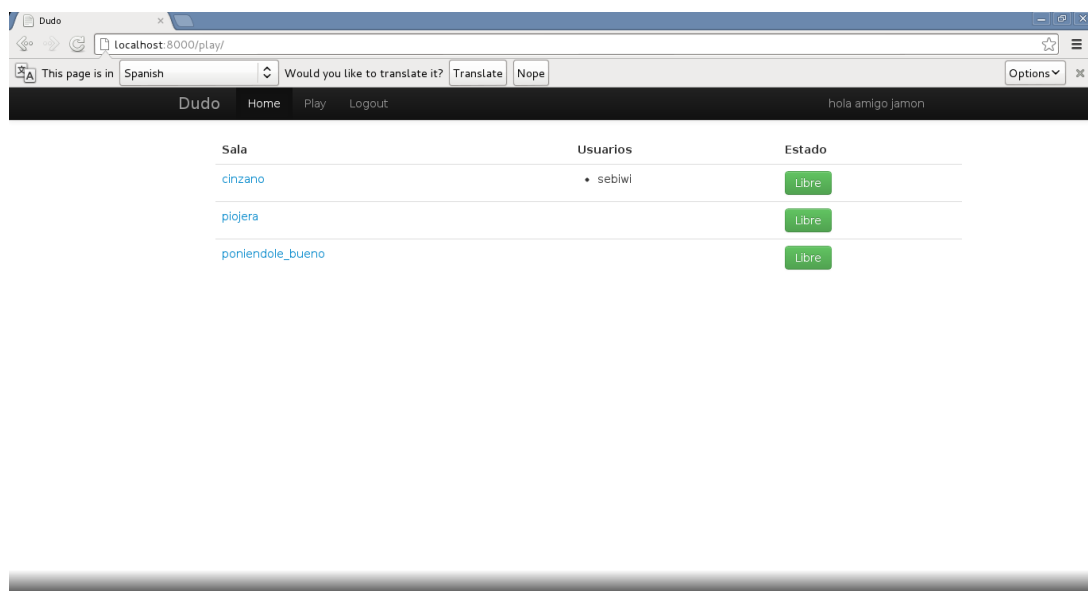


Figura 4: Lista de salas de juego

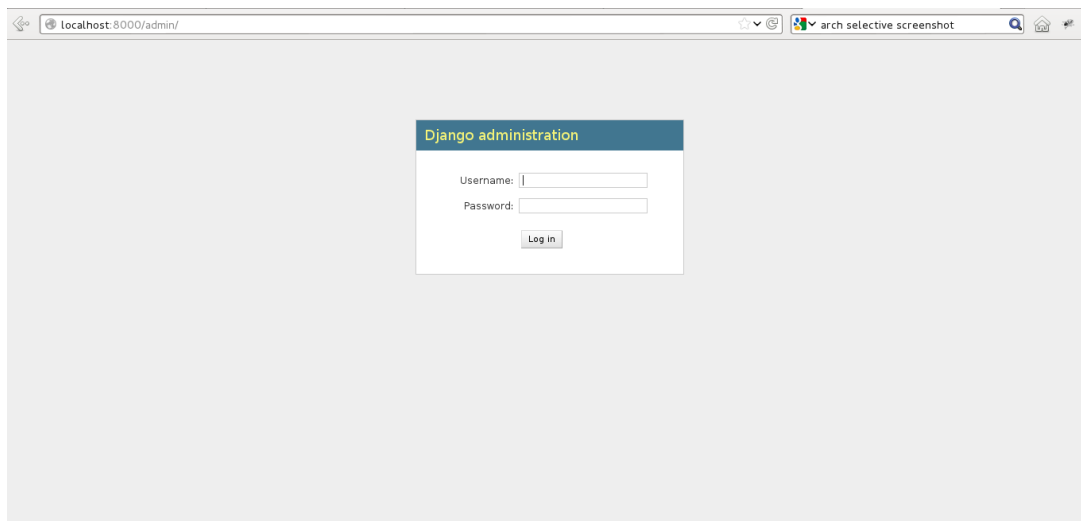


Figura 5: Log in del panel de administración

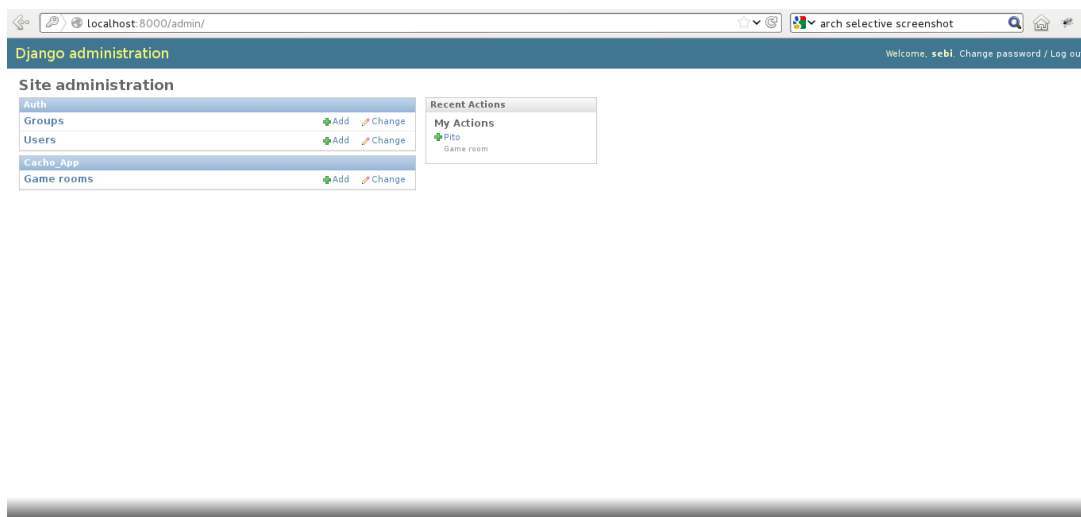
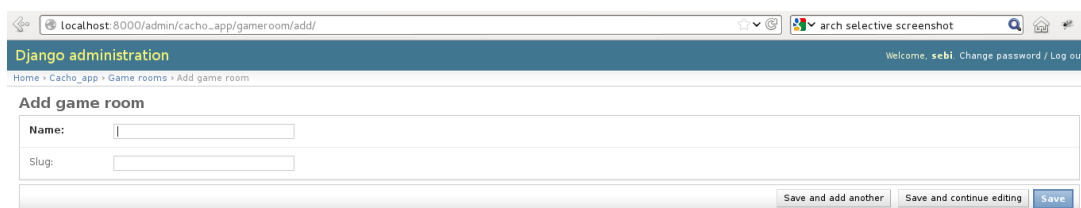
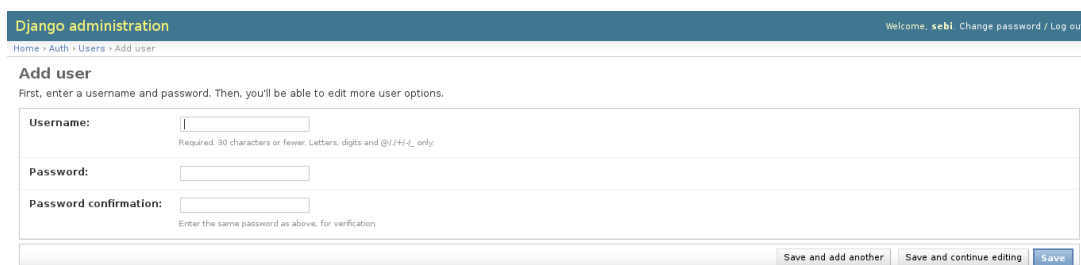


Figura 6: Vista principal del panel de administración



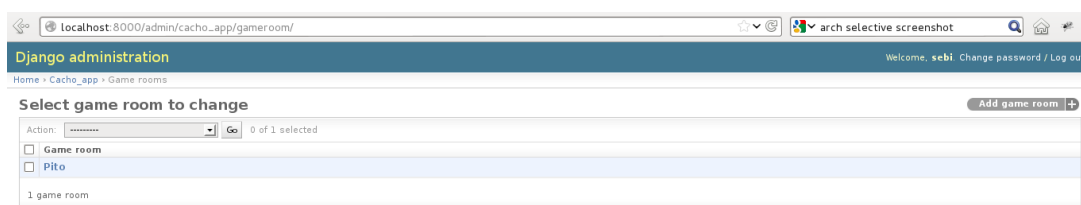
The screenshot shows the Django administration interface for adding a new game room. The browser address bar displays 'localhost:8000/admin/cacho_app/gamerroom/add/'. The page title is 'Django administration' and the user is logged in as 'sebi'. The breadcrumb trail is 'Home > Cacho_app > Game rooms > Add game room'. The form has two input fields: 'Name:' and 'Slug:'. At the bottom, there are three buttons: 'Save and add another', 'Save and continue editing', and 'Save'.

Figura 7: Agregar nueva sala



The screenshot shows the Django administration interface for adding a new user. The browser address bar displays 'localhost:8000/admin/cacho_app/gamerroom/'. The page title is 'Django administration' and the user is logged in as 'sebi'. The breadcrumb trail is 'Home > Auth > Users > Add user'. The form has three input fields: 'Username:', 'Password:', and 'Password confirmation:'. Below the 'Username:' field, there is a note: 'Required: 30 characters or fewer: Letters, digits and @/./+/-/_ only'. Below the 'Password confirmation:' field, there is a note: 'Enter the same password as above, for verification'. At the bottom, there are three buttons: 'Save and add another', 'Save and continue editing', and 'Save'.

Figura 8: Agregar nuevo usuario



The screenshot shows the Django administration interface for selecting a game room to change. The browser address bar displays 'localhost:8000/admin/cacho_app/gamerroom/'. The page title is 'Django administration' and the user is logged in as 'sebi'. The breadcrumb trail is 'Home > Cacho_app > Game rooms'. The form has a table with two columns: 'Action:' and 'Go'. The table contains one row with the text '1 game room'. At the bottom, there are three buttons: 'Save and add another', 'Save and continue editing', and 'Save'.

Figura 9: Editar sala

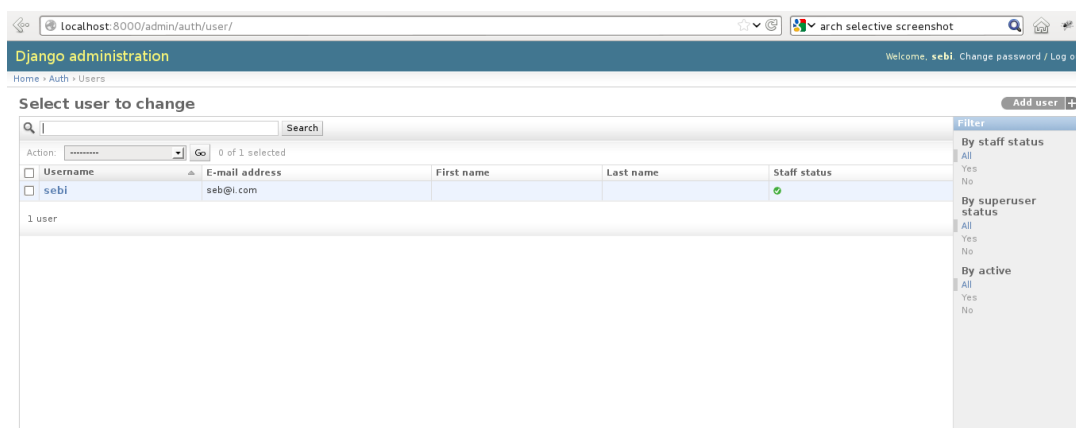


Figura 10: Editar usuario

5. Diseño del sistema

5.1. Diseño de la arquitectura

El diseño de arquitecturas que utilizaremos será el modelo MVC. Este se divide en 3 partes:

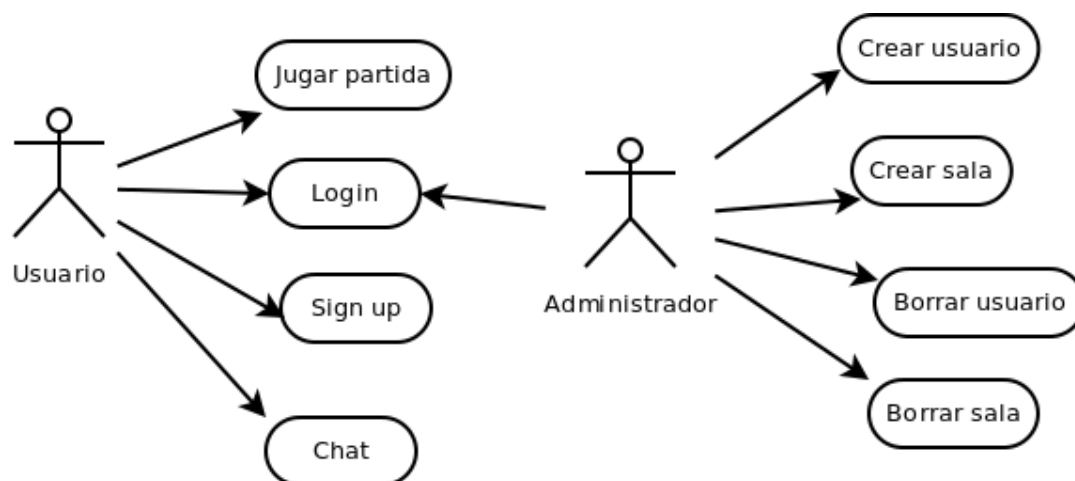
- **Modelo:** El modelo es una entidad que representa la información específica con la cual el sistema opera (La base de datos). Interactúa con la **Vista** y el **Controlador** cuando hay un cambio de su estado. Esta interacción permite a la vista entregar contenido actualizado, y a el controlador a cambiar el set de comandos disponibles.
- **Controlador:** El controlador maneja el modelo, interactúa con él y lo modifica, en respuesta a eventos generados por el usuario. El controlador también puede interactuar con las vistas, actualizandolas.
- **Vista:** La vista interactúa con el modelo, pidiendole la información que necesita para representar los datos para el usuario.

Elegimos esta arquitectura porque resulta natural trabajar con ella para el tipo de proyecto que estamos desarrollando. La mayoría del funcionamiento del sistema esta basado en eventos que van a gatillar interacciones con el modelo, y estas modificaciones del mismo producirán cambios en la vista, ocurriendo todo esto asincrónicamente.

5.2. Diseño lógico

5.2.1. Diagrama de casos de uso

El diagrama de casos de uso presenta a los dos actores presentes en el juego: el cliente y el administrador. En el se especifican las funciones e interacciones que tienen cada uno de ellos con el sistema.



5.2.2. Modelo de datos

El modelo de datos se adjunta en el apéndice. Solo fue necesaria la creación de `GameRoom`

5.2.3. Diagrama de clases

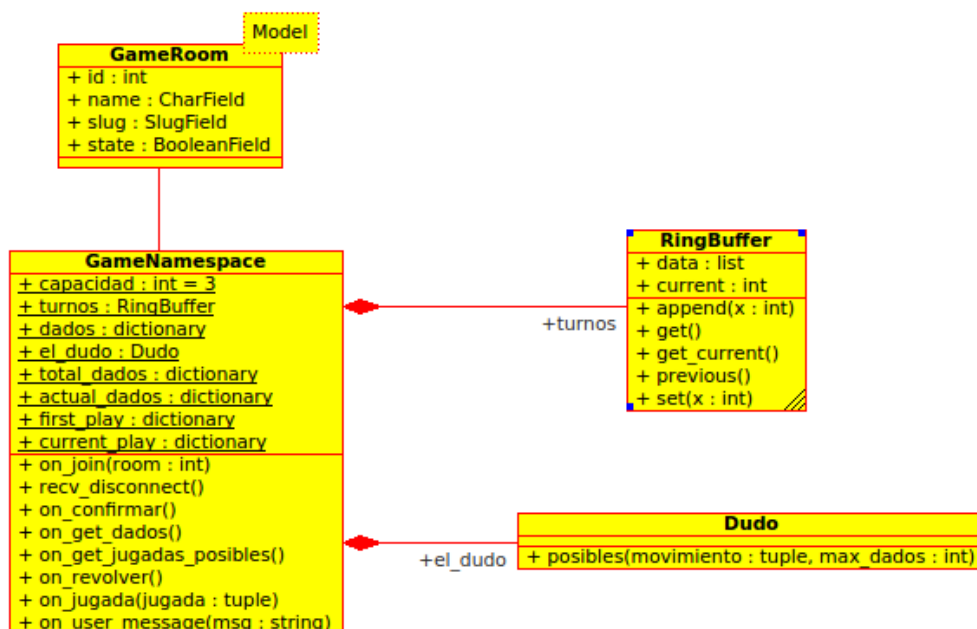
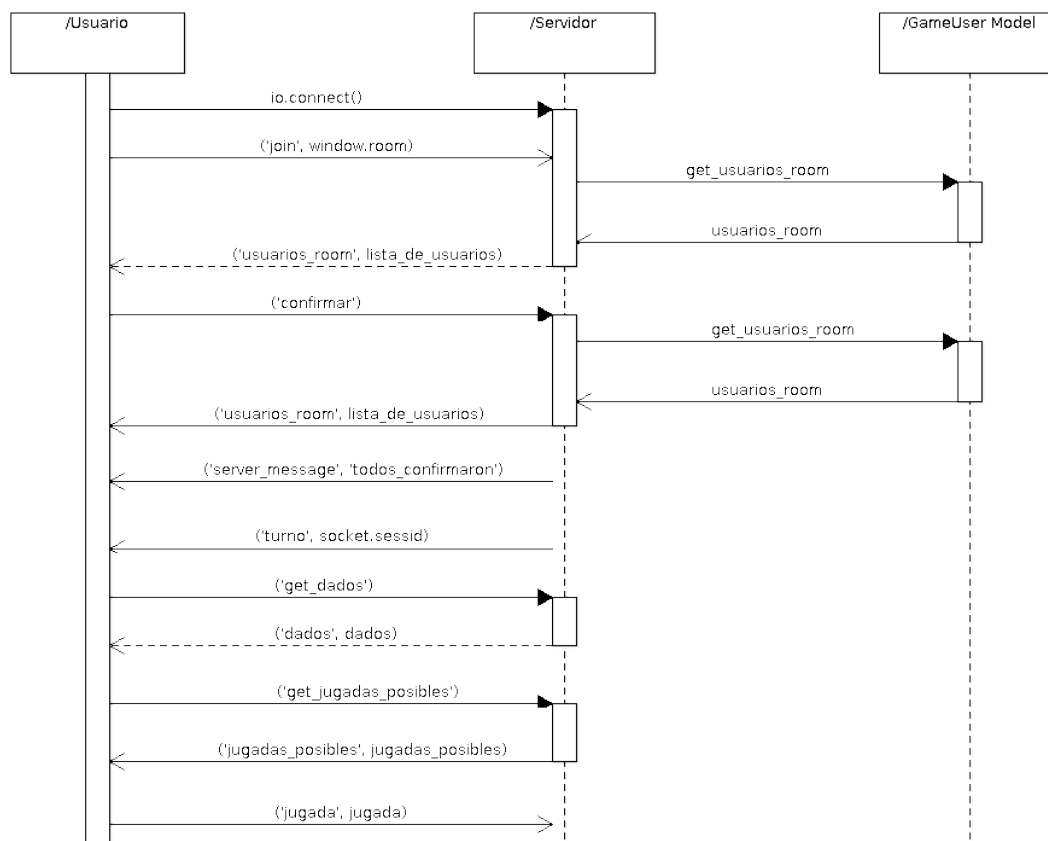


Figura 11: Diagrama de clases

5.2.4. Diagrama de secuencia

El siguiente diagrama de secuencia muestra el protocolo de intercambio de mensajes entre el cliente y el servidor para un juego. Mas adelante se explicará el contexto y el significado de cada mensaje.



5.2.5. Diagrama de flujo de la lógica del juego

El diagrama de flujo adjunto en el apéndice muestra cómo se desarrolla el juego una vez que a un jugador le llega un turno. Esto se ejecuta desde el inicio por cada turno jugado.

6. Implementación

6.1. Descripción de componentes

Actualmente, el servidor consta de un proyecto de Django, el cual tiene dos aplicaciones, `cache_site` y `cache_app`.

`cache_site` contiene objetos y sistemas relacionados con el sitio entero, se compone de los siguientes módulos:

urls.py: Derivará una URL hacia una vista o módulo. En el caso de `/socket.io`, lo derivará al Namespace específico al cual se quiere conectar. Al ingresar a `/play/` se cargarán las URL de la aplicación `cache_app`.

```
urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^login/$', 'django.contrib.auth.views.login'),
    url(r'^play/', include("cache_app.urls")),

    # socket.io
    url("^socket\.io", include(socketio.sdjango.urls)),
)

urlpatterns += patterns('cache_site.views',
    # login logout
    url(r'^logout/', 'logout_view'),
```

Listing 1: `cache_site/urls.py`

views.py: Renderizará un template para el index y hará el logout. El objeto `request.user` mantiene las variables de sesión del usuario logueado. `@login_required` es un decorador de la aplicación de autenticación que provee Django (`django.contrib.auth`), un filtro para usar el método que le sigue.

```
@login_required
def index(request):
    context = {"user": request.user}
    return render(request, 'index.html', context)
```

Listing 2: `cache_site/views.py`

6.2. Aplicación del juego

cacho_app pretende mantener objetos y módulos relacionados solo con la aplicación de juego, usando las sesiones y modelos de **cacho.site**.

Se compone de los siguientes módulos:

urls.py: Cuando se ingresa a / (/play/) se mostrará la lista de las salas, mientras que por un nombre, se ingresará a la sala con el slug de la URL.

```
urlpatterns = patterns("cacho_app.views",
    url("^$", "rooms", name="rooms"),
    url("(?P<slug>.*)$", "room", name="room"),
)
```

Listing 3: cacho_app/urls.py

views.py: Dos vistas, una para listar todos los rooms, haciendo referencia al modelo (GameRoom) y luego renderizando la vista mediante el template **rooms.html**. **room** buscará el slug en el modelo y lo enviará al template para renderizarlo. Si no encuentra el slug arrojará un error 404.

```
@login_required
def rooms(request, template="rooms.html"):
    """
    Listar todos los rooms y sus participantes
    """
    context = {"rooms": GameRoom.objects.all()}
    return render(request, template, context)

@login_required
def room(request, slug, template="room.html"):
    """
    Entrar a una sala de juego
    """
    logged_user = request.user.get_full_name()
    context = {"room": get_object_or_404(GameRoom, slug=slug), "user":
        logged_user}
    return render(request, template, context)
```

Listing 4: cacho_app/views.py

Dudo.py: Contiene funciones utilitarias para la logica del juego. RingBuffer es usado para el manejo de turnos, es una lista circular, la cual a traves del metodo `get()` se obtiene el proximo elemento de la lista.

Dudo es una clase que implementa un solo metodo hasta el momento, `posibles(mov, max_dados)`, donde `mov` es una 2-tupla que indicará un movimiento. Este metodo devolverá los movimientos (2-tuplas) que son posibles de realizar (y que van con las reglas del dudo), segun una cantidad de dados. Así, cuando al jugador le llegue el turno, se le enviará una lista de movimientos posibles en base al movimiento efectuado por el jugador anterior. Con esta lista se pretende poblar un formulario en el cliente que tiene el turno, limitando sus jugadas a jugadas que sean posibles y tengan sentido.

```
class RingBuffer:
    def __init__(self):
        pass

    def remove(self, key):
        pass

    def length(self):
        pass

    def append(self,x):
        pass

    def get(self):
        pass

    def get_current(self):
        pass

    def previous(self):
        pass

    def set(self, x):
        pass

class Dudo:
    def posibles(self, movimiento, maximo_dados):
        pass
```

Listing 5: Dudo.py

cacho_socketio.py: A este módulo llegarán las conexiones y mensajes que provienen del cliente Javascript. Según el protocolo de Socket.IO, el cliente se debe conectar a un Namespace, un espacio de nombres específico que mantiene un conjunto de funciones (event handlers). En este caso, se definió el namespace `/game`, el cual agrupa eventos referentes al juego.

Cada petición HTTP GET será una instancia de esta clase (GameNamespace), por lo tanto manejará la interacción con un solo cliente logeado. Cada una de estas funciones se ejecutará dependiendo del mensaje que envíe el cliente. El objeto `request` al cual se tenía acceso en `views.py` es duplicado, y accesible desde aquí como `self.request`, pudiendo manejar variables de sesión del sitio en el ambiente que interactúa con el cliente Javascript.

Un mensaje/evento se compone de un nombre y sus datos, los cuales serán serializados a JSON para su posterior entendimiento en el cliente Javascript. Un método del Namespace es el nombre de un evento, precedido por `on_`. e.g: el mensaje enviado desde el cliente (`'join', room`), gatillará el evento `on_join(self, room_in)`, donde el argumento será su dato. Este dato como se dijo, puede ser cualquier objeto notado en JSON.

Este Namespace hereda métodos y variables que son útiles para la comunicación: `emit()`, `emit_to_room()`, `socket.sessid`, por ejemplo. Para una completa lista revisar la API de referencia ¹

```
from socketio.namespace import BaseNamespace
from socketio.mixins import RoomsMixin, BroadcastMixin
from socketio.sdjango import namespace

@namespace('/game')
class GameNamespace(BaseNamespace, RoomsMixin, BroadcastMixin):
    def initialize(self):
        pass

    def on_join(self, room_in):
        pass

    def on_confirmar(self, action):
        pass

    def on_get_datos(self):
        pass

    def on_get_jugadas_posibles(self):
        pass

    def on_jugada(self, jugada):
        pass

    ...
```

Listing 6: `cacho_app/cacho_socketio.py`

¹<https://event-socketio.readthedocs.org/en/latest/namespace.html>

6.2.1. NoSQL Redis

Para el almacenamiento de datos de una partida (datos y usuarios actuales en una partida) se usó una base de datos Redis² (NoSQL), el cual almacena en la memoria ram un par de datos (Key-value store).

Redis provee comandos para setear y obtener una campo segun su key, los campos utilizados acá fueron strings serializados en JSON, con el fin de guardar información formateada. Mas información acerca del formato se puede encontrar en los comentarios de `cacho_socketio.py`, método `on_join`.

`redis-py`³ es un cliente para Python para los comandos de Redis. Se implementó el modulo `redisutils.py`, el cual inicia la conexion al servidor de Redis y contiene funciones útiles para varios metodos/modulos.

```
redisdb = redis.StrictRedis(host='localhost', port=6379, db=0)

def get_members_info(room):
    members = []
    room_members = list(redisdb.smembers('room_' + room))

    for sessid in room_members:
        members.append(json.loads(redisdb.get('user_' + sessid)))

    return members

def get_members(room):
    return len(redisdb.smembers('room_' + str(room)))

def get_members_name(room):
    members = []
    if get_members(room) == 0:
        return members

    room_members = list(redisdb.smembers('room_' + str(room)))

    for sessid in room_members:
        members.append(json.loads(redisdb.get('user_' + sessid))['user_name'])

    return members
```

Listing 7: `cacho_app/redisutils.py`

²<http://redis.io/>

³<https://github.com/andymccurdy/redis-py>

6.2.2. Modelo de datos

El modelo de datos se definió en `models.py` como sigue. La función `slugify` convierte un nombre que puede contener espacios y otros caracteres en un string que puede ser concatenado a una URL, ese será el método de acceso a una sala.

```
class GameRoom(models.Model):

    name = models.CharField(max_length=20)
    slug = models.SlugField(blank=True)
    state = models.BooleanField(default=False)

    class Meta:
        ordering = ("name",)

    def __unicode__(self):
        return self.name

    @models.permalink
    def get_absolute_url(self):
        return ("room", (self.slug,))

    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.name)
        super(GameRoom, self).save(*args, **kwargs)
```

Listing 8: `cacho_app/models.py`

6.3. Cliente Javascript

La parte del cliente se compone de los templates de cada pagina HTML, junto a las hojas de estilo (CSS) y scripts JS. Estos archivos se ubican en las carpetas `templates` y `static` respectivamente.

En una sala de juego, se carga la biblioteca Socket.IO (`socket.io.js`), sobre la cual se construirán manipuladores de eventos asincronicos de emisión y recepción de mensajes con el servidor.

Al gatillarse estos eventos, irán a producir cambios en el HTML (DOM, GUI), ayudandonos con la biblioteca JQuery.

El siguiente codigo muestra la implementacion (reducida) del cliente. Considerar lo siguiente:

- `io.connect('/game')`; conectará el cliente Socket.IO al Namespace `game`, definido anteriormente en `cache_socketio.py`
- `socket.on('connect', function (username) ...` manipulará la llegada del evento `connect`, con argumento `username`, luego enviará la peticion a unirse a la sala mediante `emit`
- `socket.on('usuarios_room', ...` manipulará la llegada del evento `usuarios_room`, la cual contendrá como argumento a una lista de usuarios con informacion acerca de ellos en JSON.
- En sintesis, el metodo `emit(event_name, args)` enviará al servidor el evento `event_name`, conteniendo como información a `args`. El metodo `on(event_name, callback)` ejecutará la funcion `callback` (con argumentos si es que posee mensaje) en la llegada del evento `event_name`.

```
// socket.io specific code
// funciones de socket del cliente.
var socket = io.connect("/game");
var sessid;
var user_list;

var numeros = ['ningun', 'un', 'dos', 'tres', 'cuatro', 'cinco',
               'seis', 'siete', 'ocho', 'nueve', 'diez', 'once',
               'doce', 'trece', 'catorce', 'quince', 'dieciseis',
               'diecisiete', 'dieciocho', 'diecinueve', 'veinte']
var pintas = ['0', 'aces', 'tontos', 'trenes', 'cuartas', 'quintas', 'sextas']

socket.on('connect', function (username) {
    $('#chat').addClass('connected');
    socket.emit('join', window.room);
});

socket.on('user_sessid', function(id) { sessid = id; });
socket.on('turno', function(turno) { ... });
socket.on('usuarios_room', function (usernames) { ... });
...
```

Listing 9: `cache_app/static/js/cache.js`

6.4. Notas

- Se usó como base el framework Bootstrap⁴
- Más información técnica acerca de la implementación puede ser encontrada en los comentarios del código.

7. Conclusiones

Al finalizar el desarrollo de la aplicación, se ha concluido que:

- Las funcionalidades del servidor están completas a cabalidad. El cliente es completamente funcional al momento de la entrega. Se puede seguir escalando su funcionamiento, para aplicar distintas opciones, pero para efectos del trabajo asignado, esta terminado.
- Diseñar un juego es un proceso muy complicado, y muy extenso. La cantidad de componentes que se le pueden agregar son innumerables, por lo que es vital programar dejando espacio para futuras modificaciones, mejoras, y elementos nuevos. La escalabilidad es necesaria completamente.
- Es muy importante para este tipo de desarrollo el proceso de testing. Dado que es un juego, existen muchísimas interacciones distintas entre los jugadores, dando lugar a muchas instancias que pueden no haber sido previstas por los programadores. En éstas, pueden ocurrir errores, o "bugs", que alteren el curso normal del juego. El proceso debe realizarse con el fin de encontrar la mayor cantidad de errores posibles antes de sacar al producto de la fase de desarrollo.
- El modelo MVC es muy útil y eficiente a la hora de realizar aplicaciones con paso de mensajes asíncronos, ya que para efectos de visualización por parte del cliente, las vistas se actualizan al haber un cambio en el modelo. Resulta muy natural.

⁴<http://twitter.github.com/bootstrap/>

A. Figuras

Figura 12: Diagrama de flujo del juego

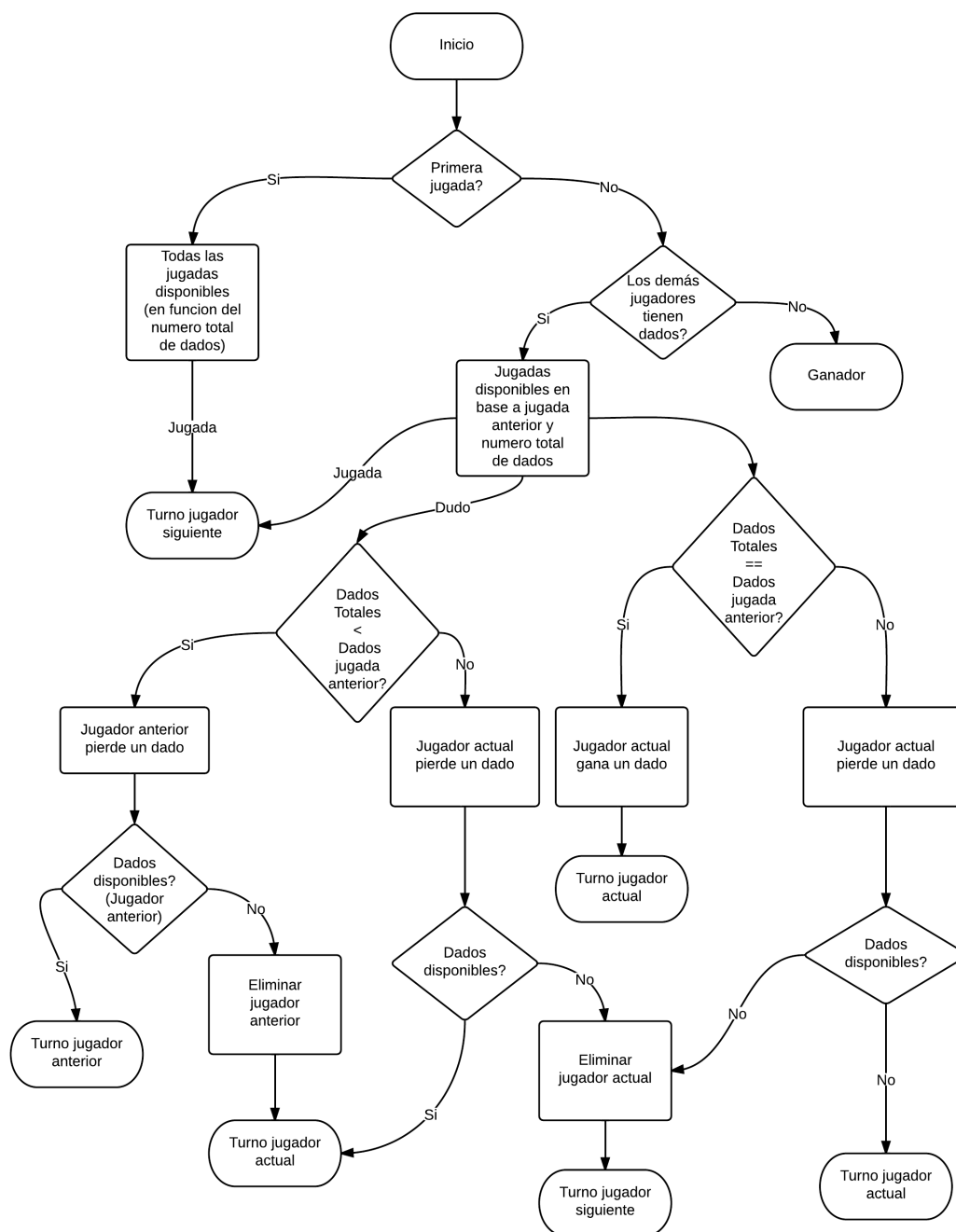
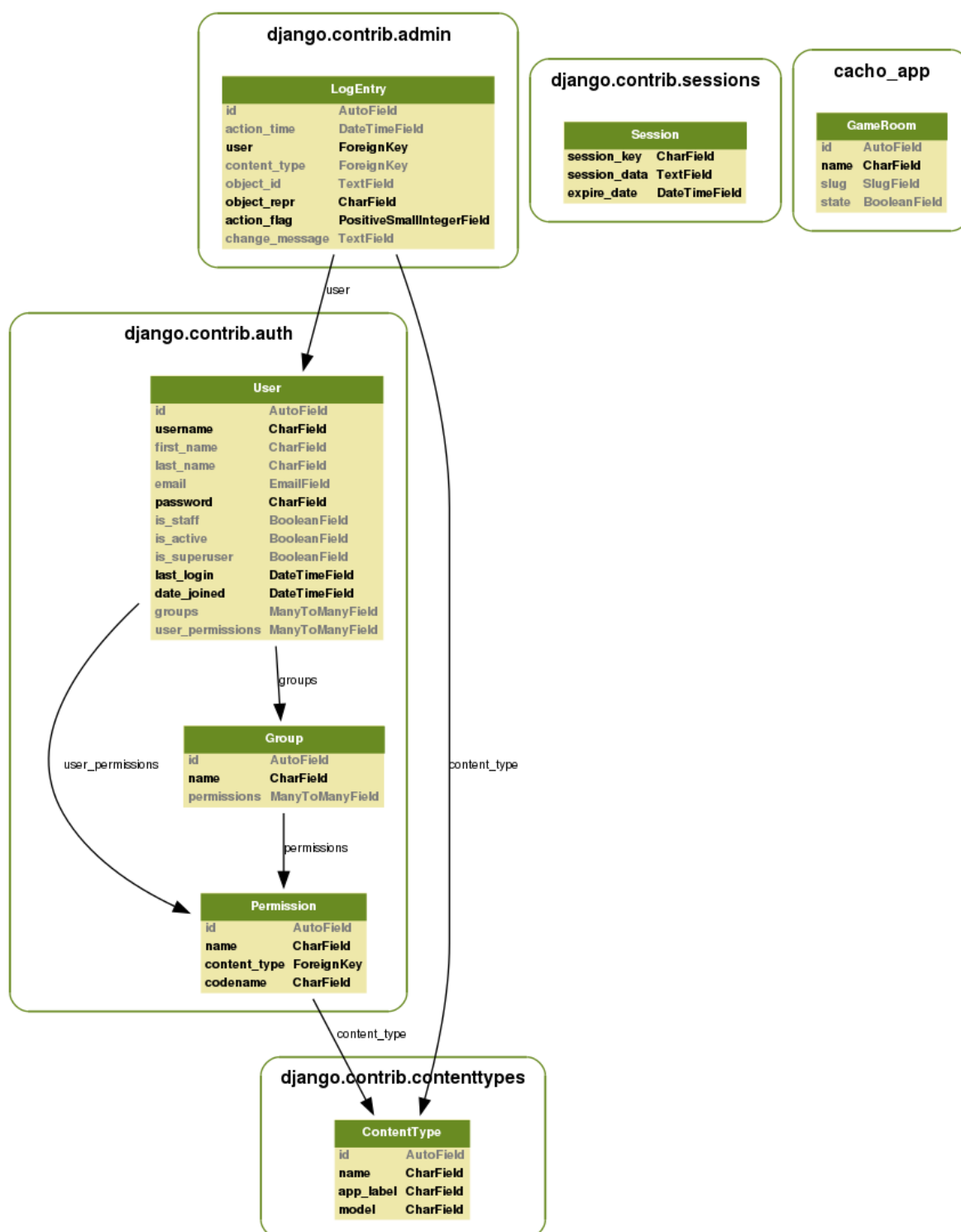


Figura 13: Modelo de datos



B. Referencias

1. Documentación oficial de Django – <https://docs.djangoproject.com/en/1.4/>
2. Documentacion de Python – <http://docs.python.org>
3. gevent-socketio API docs. – <https://gevent-socketio.readthedocs.org/>
4. Socket.IO recipes – <https://github.com/LearnBoost/socket.io>
5. Developing Django apps with zc.buildout – <http://jacobian.org/writing/django-apps-with-buildout/>
6. Evented Django – <http://codysoyland.com/2011/feb/6/evented-django-part-one-socketio-and-gevent/>
7. IRC, #django y #python en irc.freenode.net