Universidad Autónoma de Madrid

Departamento de Informática

Proyecto de Sistemas Informáticos
Práctica - 3

Roberto Latorre

Índice

1. Objetivos

En esta práctica se enuncia el proyecto a implementar en la asignatura, se crea el modelo de datos y se realiza una implementación básica del mismo.

2. Juego del Ratón y el Gato

El juego se desarrolla sobre un tablero de tamaño 8×8 como el de la figura.

1	(1,1)	1	(1,3)	3	(1,5)	5	(1,7)	7
2	8	(2,2)	10	(2,4)	12	(2,6)	14	(2,8)
3	(3,1)	17	(3,3)	19	(3,5)	21	(3,7)	23
4	24	(4,2)	26	(4,4)	28	(4,6)	30	(4,8)
5	(5,1)	33	(5,3)	35	(5,5)	37	(5,7)	39
6	40	(6,2)	42	(6,4)	44	(6,6)	46	(6,8)
7	(7,1)	49	(7,3)	51	(7,5)	53	(7,7)	55
8	56	(8,2)	58	(8,4)	60	(8,6)	62	(8,8)
	1	2	3	4	5	6	7	8

Cada casilla del tablero se identifica por la fila (coordenada vertical) y la columna (coordenada horizontal) que ocupa, teniendo además un identificador numérico entre 0 y 63 en función de sus coordenadas ($(1,1) \rightarrow 0$, $(1,2) \rightarrow 1$, y así sucesivamente hasta $(8,8) \rightarrow 63$).

Las reglas básicas del juego son las siguientes:

- Es un juego para dos jugadores. El jugador que inicia el juego siempre será el *Jugador 1*, quedando la partida a la espera que se una el *Jugador 2* para poder comenzar a jugar.

- Al comienzo de la partida, se colocan cuatro fichas de tipo "gato" para el *Jugador 1*. Inicialmente, los gatos ocuparán las cuatro casillas blancas de la fila 1 (0, 2, 4 y 6).
- En la cuarta casilla de la fila 8 contando desde la izquierda, se coloca una ficha de tipo "ratón" para el *Jugador 2* (casilla $(8, 4)$).
- Una vez se inicia la partida, cada jugador mueve uno de los gatos o el ratón por turnos según corresponda.
- Los gatos solo pueden moverse un paso en diagonal hacia abajo, pasando siempre a ocupar una casilla blanca.
- El ratón también se mueve un paso en diagonal por las casillas blancas, pero puede hacerlo tanto hacia arriba como hacia abajo.
- Ganan los gatos si consiguen atrapar al ratón de modo que no puede moverse.
- Gana el ratón si consigue escapar más allá de la línea del último gato.
- El *Jugador 1* será el gato y mueve siempre primero.
- Puedes ver un ejemplo de partida en `https://www.youtube.com/watch?v=895kc9NFCi4` (a diferencia del enunciado, en este ejemplo las fichas se colocan en las casillas negras).

Requerimientos

Los principales requisitos funcionales de la aplicación son:

- Los usuarios se identifican usando un nombre de usuario y una clave.
- Se puede crear nuevos usuarios.
- Los dos jugadores de una partida pueden jugar simultáneamente conectándose desde navegadores diferentes.
- Se puede jugar más de una partida simultáneamente.

- No se permite realizar movimientos ilegales.
- Cuando uno de los jugadores ha ganado la partida, el juego se da por finalizado.
- Se guarda en base de datos registro de cada partida, incluyendo cada uno de los movimientos y quién lo realiza.
- Se puede reproducir cualquier partida almacenada en la base de datos.

Recuerda que tu código debe satisfacer los criterios de estilo marcados por *Flake8*

3. Modelo de datos

El modelo de datos que dará soporte a la aplicación seguirá el esquema ORM (Object Relational Mapping) de *Django*, permitiendo persistir toda la información relacionada con las partidas jugadas. A continuación se muestra el diseño mínimo que debes usar en tu solución. A partir de este diseño, puedes añadir cualquier entidad o atributo que consideres necesario, pero **no eliminar** ninguno de los propuestos:

```
User(username, password)
  username => String
  password => String

Game(cat_user, mouse_user, cat1, cat2, cat3, cat4, mouse, cat_turn,
     status)
  cat_user   => Foreign Key (User)
  mouse_user => Foreign Key (User)
  cat1       => Int, not null -- posición del gato 1
  cat2       => Int, not null -- posición del gato 2
  cat3       => Int, not null -- posición del gato 3
  cat4       => Int, not null -- posición del gato 4
  mouse      => Int, not null -- posición del ratón
  cat_turn   => Boolean, not null
                -- true si mueve el gato, false si mueve el ratón
```

```
  status     => GameStatus, not null
                -- CREATED -- esperando al jugador 2
                -- ACTIVE
                -- FINISHED

Move(origin, target, game, player, date)
  origin => Int, not null
  target => Int, not null
  game   => Foreign Key (Game)
  player => Foreign Key (User)
  date   => Date, not null
```

3.1. Requerimientos a satisfacer por el Modelo de Datos

- Usar la clase `User` definida en *Django* (`django.contrib.auth.models`) de forma que se tenga acceso al sistema de verificación y validación de claves proporcionado por el propio framework.

- Usar las variables `cat1`, `cat2`, `cat3`, `cat4` y `mouse` para almacenar, respectivamente, la casilla que ocupa cada gato y el ratón en el tablero (0 a 63 ((1,1) $\rightarrow$ 0, (1,2) $\rightarrow$ 1 y así sucesivamente). Estas variables deben admitir únicamente valores en el rango $[0, 63]$ que se correspondan con casillas blancas ($\{0, 2, 4, 6, 9, 11, 13, 15, \ldots\}$). Implementar esta restricción sobrescribiendo la función `save` del modelo `Game`.

- La variable `status` debe admitir únicamente un subconjunto predefinido de valores. Para definir este subconjunto, definir una clase auxiliar `GameStatus`.

- Sobrescribir la función `__str__` para las clases definidas en `models.py`. Puedes ver ejemplos de cómo debe ser la salida en el fichero de test llamado `tests_models`, función `test11`.

- En el modelo de datos tenemos dos claves extranjeras desde `Game` a `User`. *Django* crea automáticamente la relación inversa de `User` a `Game`. Por defecto,

esta relación inversa se llama `game_user` pero al existir dos relaciones inversas entre los mismos modelos se produce un conflicto.

- Debe ser posible crear un juego en el que se introduce sólo el jugador "gato" pero **no** crear un juego en el que no se ha introducido el jugador "gato".
- Al realizar un movimiento, actualizar automáticamente los valores correspondientes del juego (utilizar la función `save` de `Move`).

3.2. Tests

Para verificar la correcta definición del modelo, se proporciona una batería de tests unitarios (no necesariamente completa) que debe satisfacer tu código. En concreto, en el fichero `tests_models.py` se proporcionan dos clases, `GameModelTests` y `MoveModelTests`, con tests para las clases `Game` y `Move`, respectivamente.

NOTA IMPORTANTE: A la hora de realizar la implementación, se recomienda seguir una estrategia TDD (test-driven development) tratando de satisfacer uno a uno y siguiendo el orden establecido cada uno de los tests incluidos en las clases `GameModelTests` y `MoveModelTests`. Esto aplica para toda la implementación del proyecto con lostests funcionales proporcionados.

4. Trabajo a realizar durante la primera y segunda semana

4.1. Modelo de datos

Crear un proyecto *Django* llamado *ratonGato* que incluya la aplicación *datamodel* con el modelo ORM que dará soporte a la aplicación satisfaciendo los siguientes requerimientos:

- Contener una página de administración (interfaz *Django* en la dirección `http://hostname:8000/admin/`) que permita introducir y borrar datos de forma

coherente en la base datos. Tanto el nombre de usuario como la password del usuario de adminitración deben ser *alumnodb*.

NOTA IMPORTANTE: un error típico a la hora de crear la página de administración es no registrar los modelos en `admin.py`.

- Los datos deben persistirse en una base de datos *PostgreSQL* llamada *ratongato*.
- Si se añade cualquier elemento al esquema de datos descrito en el enunciado, se deberá presentar un diagrama entidad-relación describiendo la base de datos que será usada por la aplicación (formato pdf). Añade este fichero al repositorio para incluirlo en la entrega.
- El código creado debe satisfacer los tests definidos en las clases `GameModelTests` y `MoveModelTests` pero no en `CounterModelTests`.

4.2. Acceso y manipulación de datos

Una vez creado y validado el modelo de datos, crear en la raíz del proyecto un script *Python* llamado `test_query.py` que realice las siguientes tareas:

- Comprobar si existe un usuario con id=10 y si no existe crearlo. Recordar que *Django* añade automáticamente a todas las tablas del modelo un atributo `id` que actúa como clave primaria.
- Comprobar si existe un usuario con id=11 y si no existe crearlo.
- Crear un juego y asignárselo al usuario con id=10. Si os hiciera falta en el futuro, tras persistir el objeto de tipo *Game*, su id se puede obtener como `nombre_objeto_game.id`.
- Buscar todos los juegos con un solo usuario asignado. Imprimir el resultado de la búsqueda por pantalla.
- Unir al usuario con id=11 al juego con menor id encontrado en el paso anterior y comenzar la partida. Imprimir el objeto de tipo `Game` encontrado por pantalla.

- En la partida seleccionada, mover el segundo gato pasándolo de la posición 2 a la 11. Imprimir el objeto de tipo `Game` modificado por pantalla.
- En la partida seleccionada, mover el ratón de la posición 59 a la 52. Imprimir el objeto de tipo `Game` modificado por pantalla.

5. Trabajo a presentar al final de la segunda semana

- Subir a *Moodle* el fichero obtenido al ejecutar el comando `git archive --format zip --output ../assign3_first_delivery.zip master`.
- **Desplegar la aplicación en *Heroku* como una aplicación nueva (no uséis la creada en la práctica anterior)**. Recordar que para que el despliegue funcione correctamente es necesario:
  - Añadir la dirección de *Heroku* a la variable `ALLOWED_HOSTS` definida el fichero `settings.py`.
  - Migrar el modelo de datos.
  - Crear el usuario de administración con nombre de usuario y password *alumnodb*.
  - El script `test_query.py` se puede ejecutar con el comando `heroku run python test_query.py`.

6. Trabajo a desarrollar durante la tercera y cuarta semana

6.1. Expansión del modelo de datos

Implementar una nueva clase llamada *Counter* que mantendrá un registro de las peticiones recibidas con independencia del usuario y la sesión. Esta clase debe almacenar un contador (que será persistido en la base de datos) y debe ser capaz de incrementar, leer e inicializar el valor de dicho contador. Para conseguir este fin seguir el patrón de diseño "singleton" que produce un, y sólo un, objeto de una clase dada. Vuestra implementación debe satisfacer los tests definidos en la clase `CounterModelTests` del fichero `tests_models.py`.

6.2. Lógica de la Aplicación

Vamos a crear diferentes funciones/controladores en el fichero `views.py` de una nueva aplicación llamada *logic*. A la hora de publicar un servicio `xxxx_service`, sugerimos crear una entrada en `urls.py` con url=`xxxx/`, funcion_a_ejecutar=`views.xxxx` y alias=`xxxx`. Adicionalmente, sugerimos mapear la url `index/` y la url vacía a la función `views.index` usando como alias `index` y `landing`, respectivamente. Alguno de los controladores proporcionados va acompañado de un "template" que **NO SE DEBE MODIFICAR** y que obliga a nombrar las variables de la sesión y/o la "request" de cierta forma. En la página inicial (`index.html`) hay un enlace que permite invocar a cada uno de los servicios publicados. Esta página **TAMPOCO DEBE MODIFICARSE**. Como referencia del comportamiento esperado de cada función podéis conectaros a `https://psi1920-p3.herokuapp.com/mouse_cat/`, donde se ha publicado una implementación de los servicios solicitados.

El catálogo de servicios a desarrollar es:

- **login_service:**
  Comprueba en el sistema de autenticación de *Django* que un par usuario-clave recibido por POST es válido y, de serlo, crea una nueva sesión para él.

Entrada: dos cadenas de caracteres con el nombre del usuario y su clave. Utilizar el template `login.html` para mostrar el formulario de login.
Resultado: usuario almacenado en la sesión o, en su caso, mensaje de error. Si el login es incorrecto repintar el formulario de login junto al mensaje de error.
Accesibilidad: solo puede ser invocado por usuarios anónimos. Para desarrollar esta funcionalidad se proporciona el decorador `anonymous_required` que implementa un filtro que redirige a una página de error en caso que la función que "decora" sea invocada por un usuario autenticado:

```
def anonymous_required(f):
    def wrapped(request):
        if request.user.is_authenticated:
            return HttpResponseForbidden(
                errorHTTP(request, exception="..."))
        else:
            return f(request)
    return wrapped


def errorHTTP(request, exception=None):
    context_dict = {}
    context_dict[constants.ERROR_MESSAGE_ID] = exception
    return render(request, "mouse_cat/error.html", context_dict)
```

NOTA: Como el sistema de autentificación que utilizará la aplicación es el mismo que el de la aplicación de administración de base de datos, en las pruebas iniciales puedes probar a autenticarte con el superusuario de esta aplicación.

- **logout_service:**
  Cierra la sesión de un usuario previamente autenticado. Para ello, borra todas las variables de sesión.
  Entrada: ninguna.
  Resultado: el usuario y todos sus datos asociados dejan de estar almacenados

en la sesión. Utilizar el template `logout.html` para reportar la acción.
Accesibilidad: solo puede ser invocado por usuarios previamente autenticados. Para implementar este requisito se recomienda revisar la documentación referente al decorador `login_required`.

- **signup_service:**
  Registra un nuevo usuario en la aplicación.
  Entrada: dos cadenas de caracteres con el nombre del nuevo usuario y su clave.
  Resultado: se creará un nuevo usuario quedando además autenticado en la aplicación de forma automática.
  Accesibilidad: solo puede ser invocado por usuarios anónimos.

- **counter_service:**
  Actualiza y muestra en pantalla una serie de contadores de peticiones recibidas.
  Entrada: ninguna.
  Resultado: actualización de dos contadores de peticiones:

  - `counter_session` indica el número de peticiones realizadas desde la sesión actual. Como ayuda para gestionar el valor del contador de sesión, el siguiente fragmento de código muestra cómo fijar y recuperar en *Django* el valor de una variable de sesión. Para más detalles, se recomienda revisar la documentación del framework:

    ```
    # set session variable
    request.session["idempresa"] = profile.idempresa


    # get session variable
    if "idempresa" in request.session:
        idempresa = request.session["idempresa"]
    else:
        idempresa = ...
    ```

  - `counter_global` mantendrá un registro del número total de peticiones recibidas independientemente del usuario y la sesión. Implementarla usando la clase `Counter` implementada en `models.py`.

- **create_game_service:**
  Crea y persiste en base de datos un nuevo juego.
  Entrada: usuario que realiza la llamada según la información almacenada en sesión.
  Resultado: siguiendo las reglas establecidas (ver Sección 2), el usuario que invoca al servicio y crea el juego es el *Jugador 1*, quedando la partida a la espera de que el *Jugador 2* se una al juego.
  Accesibilidad: solo es ejecutable por usuarios autenticados. De no conocerse la identidad del usuario, se redirigirá al usuario a la pantalla de login. Implementar este requisito usando el sistema de verificación de *Django*.

- **join_game_service:**
  Incorpora al usuario que realiza la llamada al juego con mayor `id` pendiente de comenzar.
  Entrada: usuario que realiza la llamada según la información almacenada en sesión.
  Resultado: el juego seleccionado se activa (cambia su estado) y el *Jugador 1* puede realizar su primer movimiento.
  Accesibilidad: solo es ejecutable por usuarios autenticados. De no conocerse la identidad del usuario, se redirigirá al usuario a la pantalla de login. Implementar este requisito usando el sistema de verificación de *Django*.

- **select_game_service:**
  Muestra los juegos activos en los que participa el usuario que invoca el servicio (GET) y permite seleccionar uno de ellos para jugar (POST).
  Entrada: GET $\rightarrow$ usuario que realiza la llamada según la información almacenada en sesión. POST $\rightarrow$ id del juego al que se quiere jugar.
  Resultado: una vez seleccionado uno de los juegos disponibles, su `id` se guarda en una variable de sesión llamada `game_selected`.
  Accesibilidad: solo es ejecutable por usuarios autenticados.

- **show_game_service:**
  Muestra los datos del juego seleccionado incluyendo el "tablero de juego".

Entrada: jugador y juego que se desea mostrar (ambos valores se toman de las variables de sesión).
Salida: el tablero se representará como un array $[0, 63]$ de enteros según el identificador de cada una de sus celdas. A los gatos se les asignará el valor 1, a los ratones -1 y el resto de casillas valdrá 0.
Accesibilidad: solo es ejecutable por usuarios autenticados. De no conocerse la identidad del usuario, se redirigirá al usuario a la pantalla de login. Implementar este requisito usando el sistema de verificación de *Django*.

- **move_service:**
  Realiza un movimiento sobre el juego seleccionado.
  Entrada: jugador, juego, posición inicial y final del movimiento. Tanto el jugador como el juego se obtienen de la sesión. El resto de parámetros se recibirán por POST.
  Resultado: siempre que el movimiento sea posible, se actualizan las posiciones del tablero de juego y cambia el turno.
  Accesibilidad: solo es ejecutable por usuarios autenticados que previamente hayan seleccionado un juego. De no conocerse la identidad del usuario, se redirigirá al usuario a la pantalla de login. Implementar este requisito usando el sistema de verificación de *Django*.

Para probar la funcionalidad solicitada, el fichero `tests_services.py` proporciona una batería de test (no necesariamente completa) de todos los servicios descritos. Además, la clase `GameMoveTests` del fichero `tests_function.py` incluye una serie de tests para probar el correcto funcionamiento del servicio *move_service*. Al igual que en el caso de los modelos, se recomienda tratar de satisfacer uno a uno y por orden cada uno de los tests incluidos en las clases definidas en estos ficheros.

NOTA: Desarrollar el proyecto localmente y cuando funcione subirlo a *Heroku* **sin pisar** la aplicación subida en la práctica 2.

7. Trabajo a presentar al finalizar la práctica

- Aseguraros que vuestro código satisface todos los tests proporcionados. No es admisible que se modifique el código de los tests.
- Implementar todos los tests que consideres necesarios para cubrir la funcionalidad desarrollada. Estos tests se deben incluir en clases dentro de un fichero llamado `tests_additional.py`.
- Incluir en la raíz del proyecto un fichero llamado coverage.txt que contenga el resultado de ejecutar el comando `coverage` para cada uno de los tests.
- Desplegar y probar la aplicación en *Heroku*.
- Subir a *Moodle*, en un único fichero zip, el proyecto de *Django* conteniendo la aplicación desarrollada. En concreto se debe subir el fichero obtenido ejecutando el comando `git archive --format zip --output ../assign3_final.zip master` desde el directorio del proyecto. En este punto es importante asegurarse que la variable `ALLOWED_HOSTS` del fichero `settings.py` incluido en la entrega contiene tu dirección de despliegue en *Heroku* (si no aparece no podremos corregir tu aplicación). Asimismo, comprobar que tanto el nombre de usuario como la password del usuario de administración son *alumnodb*.

8. Criterios de evaluación

Para aprobar con 5 puntos es necesario satisfacer en su totalidad los siguientes criterios:

- Todos los ficheros necesarios para ejecutar la aplicación se han entregado a tiempo.

- El script `test_query.py` hace lo que se solicita.
- La aplicación se puede ejecutar localmente.
- Al ejecutar en local los tests no marcados como optativos, el número de fallos no es superior a cinco y el código que los satisface es funcional.
- No se han modificado los templates/tests propuestos.

Cumpliendo los siguientes criterios se puede optar a una nota máxima de 6.9:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- La aplicación está desplegada en *Heroku*. En el fichero `settings.py` se encuentra añadida la dirección de *Heroku* a la variable `ALLOWED_HOSTS`. Además de estar desplegada, la aplicación funciona correctamente en *Heroku*.
- Al ejecutarse los tests no marcados como optativos en *Heroku*, el número de fallos no es superior a cuatro y el código que los satisface es funcional.
- Se utiliza correctamente el sistema de autentificación de usuarios de *Django*.

Cumpliendo los siguientes criterios se puede optar a una nota máxima de 7.9:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- El sistema soporta que varias partidas se jueguen simultáneamente entre diferentes jugadores. Se asume que cada jugador se conecta usando un navegador con una sesión diferente.
- La aplicación de administración de base de datos se ha desplegado y está accesible en *Heroku* usando el username/password *alumnodb*.
- La gestión de datos en la aplicación de administración es correcta y coherente.
- Al ejecutarse los tests no marcados como optativos en *Heroku*, el número de fallos no es superior a dos y el código que los satisface es funcional.

Cumpliendo los siguientes criterios se puede optar a una nota máxima de 8.9:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- El código es legible, eficiente, está bien estructurado y comentado.
- Se utilizan las herramientas que proporciona el framework.
- Sirva como ejemplo de los puntos anteriores:
  - Las búsquedas las realiza la base de datos no se cargan todos los elementos de una tabla y se busca en las funciones definidas en `views.py`.
  - Los errores se procesan adecuadamente y se devuelven mensajes de error comprensibles.
  - El código presenta un estilo consistente y las funciones están comentadas incluyendo su autor. Nota: el autor de una función debe ser único.
  - La comprobación de que el movimiento del ratón/gato es correcto es exhaustiva y no se basa el elementos "hardcodeados".
  - Se indenta consistentemente sin mezclar espacios y tabuladores.
  - Se es coherente con los criterios de estilos marcados por *Flake8*.

Para optar a la nota máxima se debe cumplir lo siguientes criterios:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- Todos los tests (incluidos los opcionales) y todas las pruebas ejecutadas dan resultados satisfactorios.
- Se han añadido nuevos tests no triviales (por ejemplo, se comprueba que es posible jugar varias partidas simultáneamente)
- Se reporta la cobertura (programa coverage) obtenida por los tests antes y después de añadir tus nuevos tests. La cobertura debe incrementarse.

Nota: Entrega parcial retrasada → substraer un punto por cada entrega.

Nota: Entrega final fuera de plazo → substraer un punto por cada día (o fracción) de retraso en la entrega.

Nota: El código usado en la corrección de la práctica será el entregado en *Moodle*. Bajo ningún concepto se usará el código existente en *Heroku*, *Github* o cualquier otro repositorio.