

DISEÑO Y MANTENIMIENTO DEL SOFTWARE

Proyecto 2019-2020 – Memoria entrega 2

Fecha: 28/12/2019

Aleix Rodríguez Silanes

José Antonio Cerrado Casillas

María Ojeda Ruiz

Miguel Martín Valdivielso

Pablo Seoane Fuente

Contenido

[MANUAL DE INSTALACIÓN 3](#_Toc28550089)

[MANUAL DE USO 3](#_Toc28550090)

[ARQUITECTURA Y DISEÑO 4](#_Toc28550091)

[Documentación del protocolo de comunicaciones cliente-servidor 4](#_Toc28550092)

[Arquitectura y diseño del juego 5](#_Toc28550093)

[Arquitectura y diseño del servidor 6](#_Toc28550094)

[Arquitectura y diseño de cliente 7](#_Toc28550095)

# MANUAL DE INSTALACIÓN

Para poder llevar a cabo el uso de nuestro juego necesitaremos disponer de dos elementos que nos serán totalmente imprescindibles, y que serán fácilmente adquiribles en el sistema operativo Linux. Por un lado contamos con la herramienta **Docker** que nos permitirá crear contenedores ligeros y portables para las aplicaciones software que puedan ejecutarse en cualquier máquina con Docker instalado, independientemente del sistema operativo que la máquina tenga por debajo, facilitando así también los despliegues. Por otra parte necesitaremos llevar a cabo la la instalación de **Python 3,** puesto que es la plataforma sobre la que hemos construido el código de nuestro juego y la necesitaremos para poder jugarlo.

Para llevar a cabo la instalación de Docker recurrimos a un tutorial que nos indicó de forma guiada, fácil y sencilla los pasos a seguir en nuestra consola:

* Requisitos previos:
  + Servidor Ubuntu 18.04
  + Cuenta en Docker Hub si se desea crear imágenes propias o pushearlas.
* Actualizar los paquetes existentes:
  + sudo apt update
* Instalar paquetes para usar APT con paquetes HTTPS:
  + sudo apt install apt-transport-https ca-certificates curl software-properties-common
* Agregar la clave GPG para el repositorio oficial de Docker en el equipo:
  + curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
* Agregar el repositorio Docker a las fuentes APT:
  + sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"
* Actualizar la base de datos de paquetes con los nuevos paquetes Docker del repositorio:
  + sudo apt update
* Asegurar la instalación desde el repositorio Docker en vez de Ubuntu:
  + apt-cache policy docker-ce
* Instalar Docker:
  + sudo apt install docker-ce

Respecto a la instalación de Python 3, es mucho más sencilla puesto que sólo necesitaremos instalar la aplicación de Visual Studio Code mediante un proceso convencional de descarga y ejecución de un instalador. Realizando, una vez acabada la primera parte, la elección de la extensión Python 3 y comenzando a usarla.

# MANUAL DE USO

Una vez disponemos de todo lo necesario expresado en el apartado de MANUAL DE INSTALACIÓN, podremos comenzar con los pasos a seguir para la utilización de nuestro juego.

Es imprescindible para el correcto funcionamiento de la aplicación desarrollada, que tengamos la capacidad de levantar las imágenes del servidor y los servicios del mismo, para ello utilizaremos Docker y nuestra línea de comandos consiguiendo ambos objetivos con las siguientes instrucciones:

* Construir imágenes: *docker-compose -f docker/config/base.yml build*
* Levantar servicios: *docker-compose -f docker/config/base.yml up -d*

Podremos comprobar el correcto funcionamiento de estas órdenes usando el comando *curl* con la dirección que mantiene por un lado el servidor de autentificación y por otro el servidor de hub, viendo si están en estado de “*running*” o no:

* Auth server: *curl 172.10.1.10:1234*
* Hub server:  *curl 172.10.1.20:4567*

No obstante, este proceso está automatizado mediante los scripts lanzar\_server.sh y relanzar.sh, siendo el segundo un reseteo del primero que borra las imágenes establecidas con anterioridad para poder iniciar un nuevo servidor. Ambos se encuentran en nuestro repositorio de GitHub.

Como último paso recurriremos a un script desarrollado por uno de los miembros del grupo, que deberá ser ejecutado una vez es introducido en la carpeta en la que se encuentre el código de la aplicación y que lanzará el servidor de juego mediante la clase test\_connection.py y llamará al *Main.py* de la aplicación iniciando así un menú escrito de fácil entendimiento que nos permitirá la interacción total con la aplicación desarrollada.

# ARQUITECTURA Y DISEÑO

Introducción hablando de Python.

## Documentación del protocolo de comunicaciones cliente-servidor

**El protocolo usado para la comunicación entre los clientes y el servidor es API.**

**Una API separa clientes de servidores y proporciona una forma de comunicación entre ellos.**

**En este caso los clientes pueden utilizar la API del servidor a través de varios endpoints.**

**Un endpoint es la ruta del servidor a la que se conecta el cliente para realizar una operación en el servidor. Por ejemplo /unirse es el endpoint del servidor que permite a un usuario unirse a la partida.**

**Hemos implementado 8 endpoints para que el usuario pueda comunicarse con el servidor:**

* ***/***: Endpoint básico para comprobar si el servidor se está ejecutando
* ***/unirse***: Endpoint para que un cliente se una a una partida
* ***/register***: Endpoint para dar de alta al servidor en el hub
* ***/unregister***: Endpoint para dar de baja al servidor en el hub
* ***/obtener/estado***: Endpoint para obtener el estado del juego
* ***/obtener/juegos***: Endpoint para obtener una lista con los juegos
* ***/seleccionar/juego***: Endpoint para seleccionar un juego
* ***/mover***: Endpoint para realizar un movimiento

Los detalles de cada endpoint están en el README.md de github (<https://github.com/ars1004/practica-dms-2019-2020>)

## Arquitectura y diseño del juego

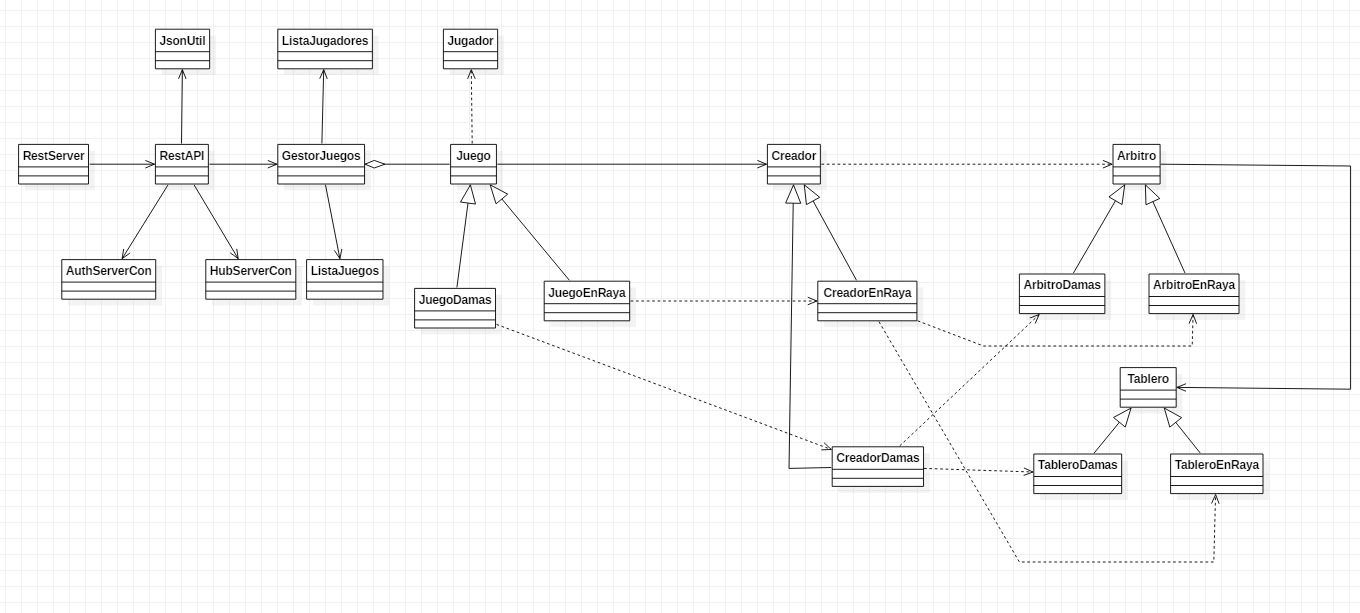
El juego se queda dividido en dos capas principalmente:

* **Datos**
  + Es la parte del juego en que se van a reunir todas las capas cuya función principal es la creación de los datos y no manipulan los objetos. Aquí se encuentran Jugador.py, Tablero.py y ListaJugadores.py.
    - Jugador.py: se encarga de realizar la creación de los jugadores asignando a cada uno su id.
    - Tablero.py: se encuentra dividido en Interfaz y clases concretas lo que podría ayudar a la incorporación de nuevos juegos de tablero.
      * En TableroInterfaz.py se crea el modelo que se sigue en el resto de los tableros.
      * TableroEnRaya.py está diseñado heredando del anterior e implementando los métodos necesarios.
      * TableroDamas.py hereda del anterior e implementa los métodos necesarios, además de realizar alguno de ellos de una forma algo especial para las necesidades del juego.
* **Lógica**
  + En esta capa del juego, se lleva a cabo la creación y mantenimiento de los objetos, así como la supervisión de las normas de cada uno de los juegos. Se encuentran las capas de GestorJuegos.py, Juego.py, Creador.py, arbitro.py.
    - En el Gestor de Juegos, se crea un diccionario a partir del cual se puede seleccionar uno de los juegos implementados y guardados en este.
    - A través de la clase Juego.py y sus hijos, JuegoEnRaya.py y JuegoDamas.py, se realiza la creación del Creador pasando a este la lista de jugadores y el tamaño del tablero. En las implementaciones concretas de los hijos se implementan las características concretas de cada uno de ellos (el tamaño) y se heredan el resto.
    - Usando la clase Creador.py y sus implementaciones concretas, se realiza la creación del arbitro del juego y el tablero. En esta clase, se llevará el control del tablero. Cada uno de los juegos tendrá su clase concreta, CreadorEnRaya.py y CreadorDamas.py, en las que se realizarán llamadas a las clases de la misma familia.
    - Por último, tenemos el árbitro, que es la clase que se encarga de la lógica del juego, movimientos legales, turno, mover piezas… Como en los casos anteriores, se encuentra una clase concreta de cada uno de los juegos ArbitroEnRaya.py y ArbitroDamas.py, cada una de ellas trabaja con las clases de su misma familia (juego).
* Como se puede observar, se ha realizado algo similar a una fábrica abstracta, por el hecho de haber creado dos familias (de juegos), donde cada una de las clases trabaja de forma concreta únicamente con las de su misma familia. Por lo que la creación y unión de nuevos elementos a la arquitectura, simplemente supondría la creación de una clase por cada elemento necesario de la familia, heredando de las interfaces que se encuentran ya creadas y añadiendo al gestor de juegos el nuevo creado.

## Arquitectura y diseño del servidor

El servidor utiliza una arquitectura de 3 capas:

* **Capa de presentación:**
  + RestServer: Es el servidor que proporciona la API REST. Contiene una fachada de la API y los endpoints de la API que van a llamar los clientes y por esto es parte de la capa de presentación (define la forma en que los clientes interactúan con el servidor mediante los diferentes endpoints). Los endpoints redirigen la petición al método correspondiente de la fachada de la API. De esta manera el servidor solo tiene una responsabilidad, recibir peticiones de los clientes y enviarlas la fachada.
  + RestAPI: Es la fachada de la API. Permite separar los juegos (subsistema) de las peticiones a la API. La responsabilidad de esta clase es realizar las acciones correspondientes a cada endpoint. Se encuentra en la capa de presentación porque está manejando la interacción entre el cliente y el servidor al gestionar las llamadas a los juegos.
  + JsonUtil: Es la clase que se encarga de “traducir” los mensajes complejos (solo se utiliza en objetos complejos como listas ya que una cadena de texto va a ser igual en formato JSON) que se transmiten entre cliente y servidor (por esta razón se encuentra en la capa de presentación). Aquí se codifica el estado del juego para enviarlo al cliente (se transforma una lista a una cadena de texto en formato JSON) y se obtiene el movimiento. De esta forma, si se cambia la forma de enviar datos solo se tiene que modificar esta clase. Además, si se quiere añadir un juego que requiera un tipo de movimiento diferente (más posiciones, letras…) solo habría que modificar esta clase ya que el movimiento se pasa al juego como una tupla que se puede desempaquetar para soportar métodos con un número diferente de parámetros.
* **Capa de datos:**
  + ListaJugadores: Es una clase que contiene una relación entre jugadores y sus tokens. De esta forma, se puede pasar fácilmente de la representación de un cliente en el servidor (token) a su representación en el juego (jugador). Está en la capa de datos puesto que solo guarda y devuelve tokens y jugadores. La abstracción de esta clase (y la proporcionada por Python) permite modificar la forma de representar a un usuario (en vez de token se usa un nombre) y un jugador (se cambia la clase Jugador) sin tener que modificar esta clase.
  + ListaJuegos: Esta clase simplemente guarda los juegos disponibles y permite obtener los nombres de los juegos o uno en concreto. De esta manera se puede modificar la forma en la que se guardan los juegos sin afectar al resto de clases. Por ejemplo, se pueden guardar los juegos en un fichero de texto (un JSON, por ejemplo) y cargarlos dinámicamente en esta clase. Al separar esta clase de GestorJuegos se cumple con la responsabilidad única porque ya no tiene que preocuparse de como se guardan o se obtienen los juegos. Esta clase está en la capa de datos porque se encarga de guardar y devolver los diferentes juegos.
  + AuthServerCon: Es la conexión con el servidor de autenticación. Su responsabilidad es conectarse e intercambiar mensajes con el servidor de autenticación. Se encuentra en la capa de datos porque solamente envía y recibe mensajes con un servidor. Al separar la conexión con el servidor de auntenticación se puede, por ejemplo, modificar la forma de autenticar un usuario (puede que el servidor cambie de endpoint para la verificación) y solo se tendría que modificar esta clase.
  + HubServerCon: Es lo mismo que AuthServerCon, pero con el servidor hub.
* **Capa de lógica:**
  + GestorJuegos: Esta clase permite puede tener un juego y permite cambiarlo entre los juegos disponibles. Es por esto que se corresponde con el contexto del patrón estrategia. La estrategia sería Juego y las estrategias concretas son JuegoDamas y JuegoEnRaya. Esto permite añadir juegos nuevos a ListaJuegos y poder jugarlos sin problemas. Además se cumple con el principio de responsabilidad única porque solo tiene que gestionar el juego al que se va a jugar.

****

## Arquitectura y diseño de cliente

El cliente queda dividido en dos capas principalmente:

* **Datos:** 
  + en esta parte se toman los datos del cliente, que la capa de lógica utilizara para interactuar con el hub, con el game-server y con el authserver. La clase que contiene esta capa se denomina en el código como “datos.py” y en ella, existen las funciones de mostrar interfaz para logearse o registrarse, para mostrar los distintos servidores a los que unirse, también muestra la lista de juegos a los que se pueden jugar, y para finalizar muestra las opciones que se pueden realizar en el juego
* **Lógica:**
  + en esta parte se interactúa con el hub, el authserver y el game-server a partir de los datos que se toman en la capa de datos. La clase que contiene esta capa se denomina en el código como “lógica.py”, y en ella, existen las funciones de registrarse o logearse, de conectar al servidor, seleccionar el juego y mostrar el estado del juego y realizar un movimiento en el juego
* **Main:**
  + en esta clase, que es la que el cliente utiliza, se juntan las dos capas, de manera que la capa de datos va pidiendo datos al cliente y con estos datos la capa de lógica va realizando las acciones necesarias para que el usuario pueda jugar al juego que desee
* Como se puede observar, se ha utilizado la arquitectura de fachada, ya que el cliente nunca interacciona directamente con la capa de lógica, si no que la capa de datos es la que actúa de intermediaria entre el cliente y la capa de lógica.