

DISEÑO Y MANTENIMIENTO DEL SOFTWARE

Proyecto 2019-2020 – Memoria entrega 1

Fecha: 24/11/2019

Aleix Rodríguez Silanes

José Antonio Cerrado Casillas

María Ojeda Ruiz

Miguel Martín Valdivielso

Pablo Seoane Fuente

# 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 servidor

El servidor está dividido en dos partes principales:

* Conexión:
  + La parte de conexión contiene la API del servidor y las conexiones con el juego y los servidores de autenticación y hub.
  + La implementación se divide en 6 partes para intentar cumplir los principios SOLID:
    - El servidor con la API rest (src/components/game-server/bin/rest-server.py) contiene los métodos necesarios para lanzar un servidor Flask y recibir peticiones de los clientes. Es donde se lanza el servidor y por ello contiene una instancia de la fachada de la API, que se encargara de la lógica de las conexiones.
      * Al separar la conexión del cliente de la lógica se consigue respetar el principio de responsabilidad única. El servidor simplemente se encarga de recibir mensajes y enviarlos y delega la lógica a la fachada. Esta parte se corresponde con la capa de presentacion ya que es la que interacciona con el usuario.
    - La fachada de la API (src/components/game-server/lib/presentation/restapi.py) contiene la lógica de la API, es decir, es la clase que se encarga de que lleguen los mensajes adecuados a cada endpoint y los envía a las clases correspondientes, por ejemplo, al juego.
      * La fachada permite separar el subsistema del juego del servidor implementando métodos para ello. Además, utiliza las clases de utilidad para realizar comunicaciones con otros servidores, entre otras cosas.
    - Cuatro clases de utilidad cuyas funciones son:
      * Separar subsistemas: las clases de utilidad del servidor de autenticación y hub que implementan métodos para comprobar si un usuario ha iniciado sesión, dar de alta y de baja el servidor, separando las conexiónes con otros subsistemas de la lógica de la API. Estas dos clases se corresponden con la parte de origen de datos ya que aunque no accedan directamente a la base de datos, utilizan llamadas a APIs que si que lo hacen .
      * Asegurar el principio de responsabilidad unica: las clases json\_util y player\_util implementan métodos que se tienen que modificar si se cambia el juego o el envio de datos. De esta manera se asegura que la lógica de la API sea independiente de estos cambios.
* Juego:
  + Board.py:
    - Creamos la clase board, con su método init que inicializa el tablero con su tamaño correspondiente, los métodos colocar y dibujarTablero no los definimos por la herencia.
    - Creamos la clase BoardEnRaya, con su método init y su método colocar para poner las fichas correspondientes a cada jugador
  + Player.py:
    - Creamos la clase Player donde definimos los métodos init, returnName que devuelve el nombre y returnColor que devuelve el color.
  + Fade.py:
    - Creamos la clase Fade donde mostramos el turno del jugador correspondiente en el método mostrarEstado y el método enRaya va pidiendo dónde quieren colocar las fichas los jugadores comprueba que el movimiento es legal y termina el juego cuando los uno de los dos jugadores gana
  + menu.py:
    - Creamos la clase menu donde definimos los siguientes métodos print\_menu que imprime el menú con las para acceder a los diferentes juegos o salir del programa, display\_menu que recoge la opción escogida del menú y la ejecuta, y el método clear que limpia la terminal.
  + arbitroEnRaya.py:
    - Llamamos a la creación de tablero y de jugadores en el init.
    - Creamos jugadores con CrearJugador.
    - Movemos piezas con movePiece.
    - Comprobamos la legalidad de un movimiento con legalMove.
    - Comprobamos si el juego ha llegado a su fin con isFinished.
    - Comprobamos si hay 3 fichas seguidas de un mismo jugador en una fila con enFila.
    - Comprobamos si hay 3 fichas seguidas de un mismo jugador en una columna con enColumna.

## Arquitectura y diseño del cliente

Dividremos la arquitectura y diseño del cliente en 3 funciones que serán *conexionauthserver, conexionservidor y conexionhub:*

* Conexionauthserver:
  + \_\_init\_\_: inicializa la clase con una IP y un puerto.
  + register: permite el registro de usuarios.
  + login: permite el login de usuarios.
* Conexionservidor:
  + \_\_init\_\_: inicializa la clase con una IP y un puerto.
  + register: Da de alta el servidor indicado.
  + unirse: El jugador indicado se une a la partida.
  + obtenerEstado: Obtiene el estado del juego.
  + obtenerJuegos: muestra la lista de juegos a los que se puede jugar (a implementar en la siguiente practica).
  + seleccionarJuego: seleccionas en juego al que quieres jugar (a implementar en la siguiente practica).
  + mover: realiza el movimiento indicado, es decir, coloca la ficha en la casilla indicada.
* Conexionhub:
  + \_\_init\_\_: inicializa la clase con una IP y un puerto.
  + obtenerLista: Devuelve la lista de los servidores a los que se puede acceder.