

HACER PROYECTOS CON GRADLE UDP

Gradle es una herramienta para compilar proyectos y subproyectos y cargas dependencia en diferentes proyectos y crear archivos ejecutables, que en el caso de java es .jar

1. primero debes de hacer las carpetas de java

- proyecto
 - bin
 - libs
 - librerias
 - src
 - main
 - java

comando (mkdir -p src/main/java)

2. instalas Gradle (la version 9 necesita una version mayor a java 11), necesitas una version de java mayor o igual a la 17

- instalas la version completa de GRADLE y extraes los archivos
- debes de copiar el path de donde esten los archivos de GRADLEW

Alternativa :

- comando para ver versiones de java
 - update-java-alternatives -l
- copias la ruta de la version que quieras
- entras en el vim
 - vim . bashrc
- colocas el comando :
 - export JAVA_HOME=/usr/lib/jvm/java-1.17.0-openjdk-amd64
- estableces como fuente el bashrc
 - source bashrc
- apuntar a la variable de entorno
 - \$GRADLE_HOME/bin:\$ICE_HOME/bin:\$JAVA_HOME/bin:\$PATH
- guardas

3. luego lo debes de agregar a las variables de entorno (**lo de arriba es una alternativa de cambiar las variables de entorno**) **Aunque es mucho mas facil utilizar las variables de entorno**

- debes de estar en la carpeta donde extrajiste los archivos de GRADLE de la descarga y tener su path para abrir las variables de entorno
- la creas, le colocas el path y le colocas como nombre GRADLE_HOME
- la agregas en path

4. con eso ya puedes entrar en la carpeta del proyecto y ejecutar

- **gradle init**

en el gradle init le undes a las siguientes opciones :

- yes
- 4
- 2
- no

5. luego haces el gradle build para crear las carpetas de gradle

- ./gradlew clean build

6. luego debes de crear las carpetas Client y Server en el proyecto con la estructura de java

- Client
 - src
 - main
 - java
- Receiver
 - src
 - main
 - java

7. luego creamos las clases en el java de cada carpeta

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;

public class Client {
    public static void main(String [] args){
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket();
        } catch (SocketException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        String message = "hola soy carol andrea, que tal estas camilo?, A00403934, すばらしい一日を!";

        byte[] data = message.getBytes();

        DatagramPacket packet = null;
        try {
            packet = new
DatagramPacket(data,data.length,InetAddress.getByName("192.168.13
1.160"), 5000 );
        } catch (UnknownHostException e) {
```

```

        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    try {
        socket.send(packet);
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    socket.close();
}
}

```

```

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;

public class Server {
    public static void main(String [] args){

        DatagramSocket socket = null;

        try {
            socket = new DatagramSocket(5000);
        } catch (SocketException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        byte [] data = new byte[1024];

        DatagramPacket packet = new DatagramPacket(data,
data.length);

        try {
            socket.receive(packet);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        String message = new String(data);
        System.out.println("receive: "+message);

        socket.close();
    }
}

```



8. te vas al archivo settings.gradle y agregas lo siguiente, asegurate de revisar el nombre del proyecto de java que estas utilizando

```
rootProject.name = 'chat_udp'

include('Client','Server')
```

9. luego debemos de obtener las dependencias de gson, para eso nos vamos a gson maven y buscamos la opcion de gradlew y le colocamos la opcion de Groovy short

The screenshot shows the Maven repository page for the Gson library. The page includes a sidebar with popular categories, a main content area with details about the library (version 2.13.2), and a right sidebar with a Starlink advertisement. The 'Gradle' tab is selected, showing the dependency in Groovy Short format: 'com.google.code.gson:gson:2.13.2'. The 'Maven' tab is also visible, showing the dependency in Maven format: 'com.google.code.gson:gson:2.13.2'.

10. copias el link y te vas a la carpeta de build.gradle y construyes un subprojects, dentro de subjects creas repositories y dependencies, en dependencies pegas el link de la pagina de maven

```
subprojects{
    apply plugin: 'java'

    repositories{
        mavenCentral()
    }

    dependencies{
        //
        https://mvnrepository.com/artifact/com.google.code.gson/gson
        implementation 'com.google.code.gson:gson:2.13.2'
    }
}
```

```
}  
  
}
```

11. luego vuelves hacer un gradle clean

- **./gradlew clean build**

12. con eso ya puedes ejecutar el servidor y recibir mensajes

- **java -jar Server/build/libs/Server.jar**

13. o puedes ejecutar cliente y mandar mensajes

- **java -cp Client/build/libs/Client.jar Client**

14. por cada cambio que realices siempre debes de volver a ejecutar el **./gradlew clean build**

HACER PROYECTOS CON GRADLE TCP

- ese comando para mostrar todos los procesos ue esten activos en ese momento en java
 - **jps -l**
- para ver los procesos de los hilos :
 - jconsole
 - cada vez que haces un new thread estas haciendo un proceso nuevo y separe memoria ram
 - patron de diseño para hilos es thread pool ya que define una cantidad fija de hilos de la aplicacion
- clase de solo un mensaje :

```
import java.io.OutputStreamWriter;  
import java.net.Socket;  
import java.util.Scanner;  
  
public class Client {  
  
    public static void main (String[] args) throws Exception  
    {  
  
        Socket sc = new Socket("localhost", 9090);  
  
        BufferedReader reader =  
            new BufferedReader(new  
InputStreamReader(sc.getInputStream()));  
  
        BufferedWriter writer =  
            new BufferedWriter(new  
OutputStreamWriter(sc.getOutputStream()));
```

```

        writer.write("THIS IS STRAVOSKA, THE GREATEST
COUNTRY!!!");
        writer.newLine();
        writer.flush();

        String response = reader.readLine();

        System.out.println("Response from server : "+
response);

        Scanner lector = new Scanner(System.in);

        String line = lector.nextLine();

        if(line.equals("exit")){
            reader.close();
            writer.close();
            sc.close();
        }

        lector.close();

    }
}

```

- clase de servidor un mensaje :

```

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;

public class Server {

    public static void main (String[] args) throws Exception{

        ServerSocket socket = new ServerSocket(9090);

        Socket sc = socket.accept();

        BufferedReader reader =
new BufferedReader(new InputStreamReader(sc.getInputStream()));

        BufferedWriter writer =
new BufferedWriter(new
OutputStreamWriter(sc.getOutputStream()));

```

```

String msg= reader.readLine();
System.out.println("Mensaje del cliente: "+msg);

writer.write("Recibido desde el server");
writer.newLine();
writer.flush();

Scanner lector = new Scanner(System.in);

String line = lector.nextLine();

if(line.equals("exit")){
    reader.close();
    writer.close();
    socket.close();
}

lector.close();
}

```

jdvc con postgres

¿Qué es JDBC?

para utilizar PostgreSQL con java necesitas del driver JDBC el cual toma el language que habla el jvm y lo traduce a un lenguaje que entiende PostgreSQL.

JDBC (Java Database Connectivity) es una **API estándar de Java** que permite a las aplicaciones Java interactuar con bases de datos relacionales. Es el equivalente de Java a ODBC.

JDBC con PostgreSQL

Cuando usas **JDBC con PostgreSQL**, estás utilizando:

- **El driver JDBC de PostgreSQL:** Un conector específico que permite la comunicación entre tu aplicación Java y la base de datos PostgreSQL
- **La API JDBC estándar:** Las interfaces y clases que Java proporciona para operaciones con bases de datos

¿Para qué sirve?

1. Ejecutar consultas SQL

java

```
Statement stmt = conn.createStatement();
```

```
ResultSet rs = stmt.executeQuery("SELECT * FROM usuarios");
```

2. Operaciones CRUD

- **Crear, Leer, Actualizar, Eliminar** datos
- Ejecutar stored procedures
- Manejar transacciones

Ejemplo práctico

java

// Insertar datos

```
String sql = "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";
```

```
PreparedStatement pstmt = conn.prepareStatement(sql);
```

```
pstmt.setString(1, "Juan");
```

```
pstmt.setString(2, "juan@email.com");
```

```
pstmt.executeUpdate();
```

PostgreSQL JDBC Driver » 42.2.2

// <https://mvnrepository.com/artifact/org.postgresql/postgresql>

implementation 'org.postgresql:postgresql:42.2.2'

lo debes de pegar en el pom.xml del proyecto maven

HACER PROYECTO DE CHAT

para realizar una aplicacion de chat que te permita enviar y recibir audios necesitaras dos clases principales (cliente) que es el encargado de realizar la accion de enviar un mensaje y (servidor) que es el encargado de recibir el mensaje.

hay que aclarar que hay dos formas de comunicacion, dos protocolos, esta el protocolo UDP que es para comunicaciones mas inmediatas pero que no se mantiene un orden o se guarda por mucho tiempo (videollamadas o llamadas en vivo).

el protocolo STP realiza el handshake de 3 vias, por lo que aunque es mas lento ya que necesita de las 3 confirmaciones, es mas ordenado y guarda por mas tiempo la informacion (chats escritos o grabaciones de audio).

HACER CHAT UDP :

para realizar un chat con protocolo UDP necesitas de cierto elementos :

1. **DatagramSocket:**

- **Qué es:** DatagramSocket es una clase en Java que representa un socket para enviar y recibir paquetes de datagramas (paquetes UDP). sirve para manejar conexión sin un flujo determinado, basada en datagramas individuales.
- **Para qué sirve:** Se utiliza para la comunicación basada en UDP (User Datagram Protocol). En el lado del cliente, se usa para enviar paquetes a un servidor. En el lado del servidor, se usa para escuchar en un puerto específico y recibir paquetes.

- **Implementación:**
 - En el cliente: `socket = new DatagramSocket();` crea un socket vinculado a cualquier puerto disponible en la máquina local.
 - En el servidor: `socket = new DatagramSocket(5000);` crea un socket vinculado al puerto 5000.
- 2. **socket (la variable no el objeto):**
 - **Qué es:** Es una instancia de la clase `DatagramSocket`.
 - **Para qué sirve:** Esta variable se utiliza para realizar operaciones de envío y recepción de paquetes UDP.
 - **Implementación:** Se crea en el cliente y en el servidor, y se cierra al final con `socket.close()`.
- 3. **byte:**
 - **Qué es:** byte es un tipo de dato primitivo en Java que representa un valor de 8 bits (un octeto).
 - **Para qué sirve:** En el contexto de red, los datos se envían y reciben como arrays de bytes. Por ejemplo, el mensaje de texto se convierte a un array de bytes para ser enviado, y el array de bytes recibido se convierte de vuelta a texto.
 - **Implementación:**
 - Cliente: `byte[] data = message.getBytes();` convierte el String en un array de bytes.
 - Servidor: `byte [] data = new byte[1024];` crea un buffer de bytes para recibir los datos.
- 4. **DatagramPacket packet (la variable packet):**
 - **Qué es:** `DatagramPacket` es una clase que representa un paquete de datagrama (un paquete UDP).
 - **Para qué sirve:** Contiene los datos a enviar o recibir, junto con la dirección y el puerto del destinatario (para enviar) o del remitente (para recibir).
 - **Implementación:**
 - En el cliente: `packet = new DatagramPacket(data, data.length, InetAddress.getByName("192.168.131.160"), 5000);` crea un paquete con los datos, la longitud, la dirección IP del servidor y el puerto.
 - En el servidor: `DatagramPacket packet = new DatagramPacket(data, data.length);` crea un paquete para recibir datos, especificando el buffer y su longitud.
- 5. **packet (la variable):**
 - **Qué es:** Es una instancia de `DatagramPacket`.
 - **Para qué sirve:** En el cliente, contiene el mensaje a enviar y la información de destino. En el servidor, se utiliza para recibir el mensaje del cliente.
 - **Implementación:** Se utiliza en `socket.send(packet)` en el cliente y en `socket.receive(packet)` en el servidor.
- 6. **socket.receive(packet):**
 - **Qué es:** Es un método de la clase `DatagramSocket` que recibe un paquete de datagrama.
 - **Para qué sirve:** Este método bloquea el programa hasta que recibe un paquete. Cuando se recibe, los datos se copian en el buffer del paquete y se actualiza la información del remitente (dirección y puerto) en el paquete.
 - **Implementación:** En el servidor, `socket.receive(packet);` espera a que llegue un paquete y luego lo procesa.
- 7. **socket.close():**
 - **Qué es:** Es un método de la clase `DatagramSocket` que cierra el socket.
 - **Para qué sirve:** Libera el puerto asociado al socket y los recursos del sistema. Es importante cerrar el socket cuando ya no se necesita.

- **Implementación:** Se llama al final del programa, tanto en el cliente como en el servidor.
8. **Buffer de recepción:**
- **Qué es:** Es un array de bytes que se utiliza para almacenar los datos recibidos en un paquete.
 - **Para qué sirve:** En UDP, cuando se recibe un paquete, los datos se colocan en este buffer. La longitud del buffer debe ser suficiente para contener los datos recibidos.
 - **Implementación:** En el servidor, `byte [] data = new byte[1024];` crea un buffer de 1024 bytes. Luego, este buffer se pasa al `DatagramPacket` para que lo llene cuando llegue un paquete.

Implementación en el código:

Cliente:

- Se crea un `DatagramSocket` sin puerto específico (el sistema operativo asigna uno).
- Se define un mensaje y se convierte a bytes.
- Se crea un `DatagramPacket` con los bytes, la dirección IP del servidor y el puerto.
- Se envía el paquete con `socket.send(packet)`.
- Se cierra el socket.

Servidor:

- Se crea un `DatagramSocket` en el puerto 5000.
- Se crea un buffer de bytes y un `DatagramPacket` asociado a ese buffer.
- Se espera a recibir un paquete con `socket.receive(packet)`.
- Se convierte el buffer de bytes a `String` y se imprime.
- Se cierra el socket.

Lógica de Comunicación

1. **Flujo del Cliente:**
 - Crea socket → Convierte mensaje a bytes → Empaqueta datos + dirección destino → Envía paquete → Cierra conexión.
2. **Flujo del Servidor:**
 - Crea socket en puerto específico → Prepara buffer de recepción → Espera bloqueando por datos → Procesa datos recibidos → Cierra conexión.

IMPLEMENTACION DE CLIENT :

```
import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.SocketException;

import java.net.UnknownHostException;
```

```

public class Client {

    public static void main(String [] args){

        //se crea un objeto de tipo DatagramSocket para poder enviar un
mensaje

        //el socket sirve para enviar o recibir un mensaje

        DatagramSocket socket = null;

        try {

            //el socket que se crea no se le asigna puerto ya que va
ser el encargado de

            //enviar no recibir

            socket = new DatagramSocket();

        } catch (SocketException e) {

            e.printStackTrace();

        }

        //mensaje que se va a enviar

        String message = "hola soy carol andrea, que tal estas camilo?,
A00403934, すばらしい一日を!";

        //se crea un array de bytes los cuales van a contener la
informacion del mensaje

        byte[] data = message.getBytes();

        //se crea un objeto de tipo DatagramPacket para crear un
paquete y enviar los bytes

        //al servidor

        DatagramPacket packet = null;

        try {

```

```

        //al paquete creado le asignas unos bytes (data), una
longitud de esos bites

        //, un host (direeccion ip) y un puerto

        //el host y puerto tienen que ser segun el servidor

        packet = new
DatagramPacket(data,data.length,InetAddress.getByName("192.168.131.160"
), 5000 );

        } catch (UnknownHostException e) {

            e.printStackTrace();

        }

        try {

            //envias el socket

            socket.send(packet);

        } catch (IOException e) {

            e.printStackTrace();

        }

        //cierras el socket

        socket.close();

    }

}

```

IMPLEMENTACION DE SERVIDOR :

```

import java.io.IOException;

import java.net.DatagramPacket;

```

```
import java.net.DatagramSocket;

import java.net.SocketException;

public class Server {

    public static void main(String [] args){

        //creas un objeto de tipo DatagramSocket para poder recibir un
mensaje

        DatagramSocket socket = null;

        try {

            //defines el puerto de escucha en un puerto

            socket = new DatagramSocket(5000);

        } catch (SocketException e) {

            e.printStackTrace();

        }

        //defines cuantos bytes vas a utilizar para almacenar los bytes
que lleguen

        //se le conoce como buffer de bytes

        byte [] data = new byte[1024];

        // se crea un paquete con la estructura de los bytes definidos

        DatagramPacket packet = new DatagramPacket(data, data.length);

        try {

            //segun el socket que recibiste, se espera de un puerto
recibir unos bytes

            //y lo guardas en un paquete

            socket.receive(packet);

        } catch (IOException e) {

            e.printStackTrace();

        }

        //descomprime los bits del data y lo muestra en un mensaje
    }
}
```

```
String message = new String(data);

System.out.println("receive: "+message);

socket.close();

}

}
```

HACER CHAT TCP :

cuando estas trabajando con TCP en vez de utilizar los objetos tipo DatagramSocket (UDP) utilizas Socket (TCP)

Socket (TCP)

java

```
Socket sc = new Socket("192.168.1.100", 9090);
```

Qué es: Conexión orientada a flujo, confiable y con establecimiento de conexión.

DatagramSocket (UDP)

java

```
DatagramSocket socket = new DatagramSocket();
```

Qué es: Conexión sin conexión, basada en datagramas individuales.

2. Tabla Comparativa Completa

Característica	Socket (TCP)	DatagramSocket (UDP)
Tipo de conexión	Orientado a conexión	Sin conexión
Fiabilidad	Garantizada (ACK)	No garantizada
Orden de paquetes	Preservado	No preservado

Control de flujo	Sí	No
Control de congestión	Sí	No
Overhead	Alto (headers + ACK)	Bajo
Velocidad	Más lento	Más rápido
Uso de recursos	Mayor	Menor

3. Diferencias en la Implementación

Establecimiento de Conexión

TCP (Socket):

```
java
```

```
// Cliente - Conexión explícita
```

```
Socket socket = new Socket("server.com", 8080); // 3-way handshake
```

```
// Servidor - Aceptación de conexión
```

```
ServerSocket serverSocket = new ServerSocket(8080);
```

```
Socket clientSocket = serverSocket.accept(); // Espera conexión
```

UDP (DatagramSocket):

```
java
```

```
// No hay establecimiento de conexión
```

```
DatagramSocket socket = new DatagramSocket(); // Listo para enviar/recibir
```

```
// Envío directo sin handshake
```

IMPLEMENTACION DE CLIENT :

```
//BufferedReader: Lee texto de un flujo de entrada (input stream)
buffereado, lo que mejora la eficiencia de la lectura.

import java.io.BufferedReader;

//BufferedWriter: Escribe texto en un flujo de salida (output stream)
buffereado, mejorando la eficiencia de la escritura.

import java.io.BufferedWriter;

//InputStreamReader: Es un puente entre bytes y caracteres, convierte
un flujo de bytes en un flujo de caracteres.

import java.io.InputStreamReader;

//OutputStreamWriter: Convierte un flujo de caracteres en un flujo de
bytes.

import java.io.OutputStreamWriter;

//Socket: Clase para crear un socket TCP que se conecta a un servidor.

import java.net.Socket;


import java.util.Scanner;


public class Client {

    public static void main (String[] args)throws Exception {

        //Simular 10 clientes conectándose simultáneamente al servidor

        for(int i=0; i<10; i++){

            /*

            * Crea un nuevo hilo usando expresión lambda
```



```

        new Thread(()->{ ... }): Crea hilo con tarea definida
inline

        sendMessage(): Método que contiene la lógica de
comunicación

        */

        new Thread(()->{

            try {

                sendMessage();

            } catch (Exception e) {

                e.printStackTrace();

            }

            //inicializa hilo

        }).start();

    }

}

//metodo que establece conexion

static void sendMessage()throws Exception{

    //como el servidor funciona en la ip especifica, el cliente
debe de tener esa misma ip para funcionar

    //se crea un socket para poder realizar conexion y enviar
informacion

    Socket sc = new Socket("192.168.131.42",9090);

```

```
//creamos un BufferedReader para poder recibir un mensaje del
servidor

/*
    * sc.getInputStream(): Obtiene stream de bytes de entrada del
socket

    InputStreamReader: Convierte bytes a caracteres
(decodificación)

    BufferedReader: Aplica buffering para lectura eficiente por
líneas

    Propósito: Leer respuestas del servidor

*/

BufferedReader reader =new BufferedReader(new
InputStreamReader(sc.getInputStream()));

//creamos un BufferedWriter para poder enviar un mensaje al
servidor

BufferedWriter writer =new BufferedWriter(new
OutputStreamWriter(sc.getOutputStream()));

//utilizamos el BufferedWriter para enviar un String

writer.write("carol stravoskiana : THIS IS STRAVOSKA, THE
GREATEST COUNTRY!!!");

//esto se coloca para que los BufferedReader reader lean hasta
que se encuentran un salto de linea (/n)
```

```

        //por lo que se debe de colocar uno para que reader sepa cuando
        parar

        writer.newLine();

        //flush(): Vacía el buffer y envía todos los datos al servidor

        writer.flush();

        //recibe respuesta del servidor

        String response = reader.readLine();

        System.out.println("Response from server : " + response);

        //cierra todo

        reader.close();

        writer.close();

        sc.close();

    }
}

```

IMPLEMENTACION DE SERVIDOR :

```

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.net.InetAddress;

```

```
import java.net.InetAddress;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.Scanner;

import java.util.concurrent.Executor;

import java.util.concurrent.Executors;

import java.util.concurrent.Semaphore;


public class Server {

    private static int acc=0;


    //crea un semaforo que es una funcion de sincronizacion que
    gestiona el flujo de los hilos

    //new Semaphore(1); --> hace que solo pueda pasar un hilo a la vez

    private static Semaphore semaphore = new Semaphore(1);


    public static void main (String[] args) throws Exception{

        /*

        *esto es para aplicar el patron de diseño de thread pool que
        basicamente define una cantidad de hilos a utilizar

        genera una cola de hilos determinados que se envia a la cpu,

        los hilos que se genera en el thread pool no se destrullen al
        terminar proceso si no que quedan en espera
```

```
la fabrica executor es quien me crea los hilos

*/

Executor ex = Executors.newFixedThreadPool(15);

//ip del servidor (el cliente debe de estar en el mismo
segmento de red)

/*
    * el archivo host contiene los nombres vinculados a las
direcciones ip

    * lo que permite utilizar ip y nombres

*/

InetAddress address = InetAddress.getByName("localhost");

/*
    * problema condicion de carrera : cuando ejecutas hilos, un
proceso puede ser

    * mas rapido que el otro, por lo que se comienzan a
sobreescribir informacion ya

    * que un proceso subio informacion antes que otro y eso afecta
a la memoria.

    *

    * problema dead lock : es cuando varios hilos comparten
recursos y un se detienen entre si

    * porque ocupan el mismo recursos que el otro necesita. se
forma un bucle (ABRAZO MORTAL)

    * un hilo esta bloqueado porque necesita el recurso de otro
hilo pero ese hilo esta bloqueado
```

```

        * porque necesita al otro hilo

        *

        * para evitar problemas de concurrencia se utiliza el
synchronized

        */

        //establece un puerto

        //( PUERTO : 9090, CANTIDAD DE CONEXIONES : 50, DIRECCION IP :
addres);

        //el Serversocket te crea una instancia de conexion de 3 vias
(coloca en escucha el puerto)

        final ServerSocket socket = new ServerSocket(9090,50,addres);

        //despues de la linea anterior el servidor que en estado de
escucha y se llama al metodo de recibir

        //crea un primer hilo en donde va a cerra el socket en un nuevo
hilo

        //getRuntime().addShutdownHook --> lo que indica es que cuando
se cierre la maquina virtual se realiza la operacion

        //en este caso que cierre el socket

        Runtime.getRuntime().addShutdownHook(new Thread()->{

            try {

                socket.close() ;

            } catch (Exception e) {

                e.printStackTrace();

            }

```

```
    ));  
  
    //ejecuta los hilos de la pool de hilos para en un bucle ir  
    //reciviendo mensajes de un clienete  
  
    do{  
  
        //establece la coneccion de las 3 vias  
  
        //se utiliza un nuevo socket ya que el socket del puerto  
        //9090 esta ocupado asi que se utiliza un nuevo socket para recibir  
  
        Socket sc = socket.accept();  
  
        //defines la funcion que quieres ejecutar dentro del hilo  
  
        ex.execute()->{  
  
            try {  
  
                System.out.println("new client");  
  
                //llama a la funcion para recibir del cliente  
                //mediante el socket que esta abierto  
  
                responder(sc);  
  
            } catch (Exception e) {  
  
                e.printStackTrace();  
  
            }  
  
        });  
  
    }while(true);  
  
}
```

```
//metodo que se utiliza para recibir un mensaje del cliente

public static void responder(Socket sc) throws Exception{

    //pausa el hilo 5 segundos

    Thread.sleep(5000);

    //Crea un lector de texto (BufferedReader) que lee datos del
stream de entrada del socket.

    //Es decir, se prepara para leer lo que el cliente envíe al
servidor a través del socket.

    BufferedReader reader = new BufferedReader(new
InputStreamReader(sc.getInputStream()));

    //Crea un escritor de texto (BufferedWriter) que escribe datos
en el stream de salida del socket.

    //Esto permite enviar respuestas del servidor al cliente.

    BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(sc.getOutputStream()));

    //Lee una línea de texto enviada por el cliente.

    String msg= reader.readLine();

    System.out.println("Mensaje del cliente: "+msg);

    //quieres sincronizar el acumulador para que sea segun cada
hilo

    //por cada acquire se necesita un release o la aplicacion
falla!!!

    //cuando utilizas el acquires estas pausando el hilo para
realizar el proceso

    semaphore.acquire();
```



```
        acc++;

        //Prepara una línea de texto para enviar al cliente, indicando
que el servidor recibió el mensaje y mostrando el número de conexión
(valor de acc).

        writer.write("Recibido desde el server #: "+acc);

        //lo que hace realease es que reinicia el semaforo para que se
vuelva a ejecutar con los siguientes hilos

        semaphore.release();

        writer.newLine();

        writer.flush();

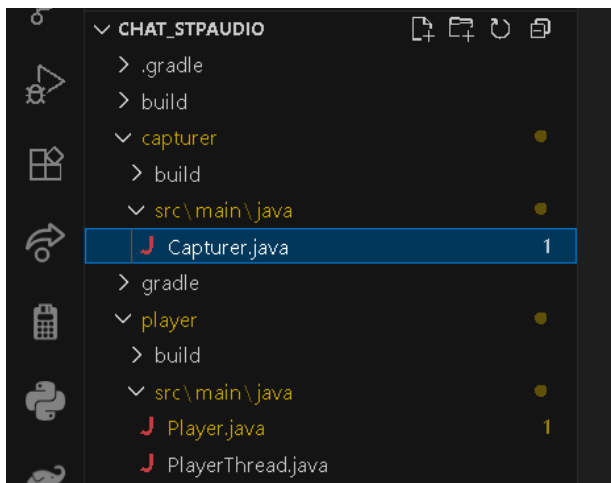
        reader.close();

        writer.close();

        sc.close();

    }
}
```

HACER CHAT TCP AUDIO :



- **build**

```
subprojects{

    apply plugin : 'java'

    repositories{

        mavenCentral()

    }

    dependencies{

    }

    jar{

        manifest{

            attributes(

                'Main-Class': project.name.capitalize()

            )

        }

    }

}
```

```
}  
  
}
```

- settings

```
rootProject.name = 'chat_stpAudio'  
  
include('player','capturer')
```

- capeta caputere

```
import java.net.DatagramPacket;  
  
import java.net.DatagramSocket;  
  
import java.net.Inet4Address;  
  
import java.net.InetAddress;  
  
import java.net.Socket;  
  
  
import javax.sound.sampled.AudioFormat;  
  
import javax.sound.sampled.DataLine;  
  
import javax.sound.sampled.SourceDataLine;  
  
import javax.sound.sampled.TargetDataLine;  
  
import javax.sound.sampled.AudioSystem;  
  
  
//Esta clase captura audio del micrófono y lo envía por UDP a un  
destinatario.  
  
//mientras que la clase PlayerThread es la que se encarga de  
reproducirlo  
  
public class Capturer {
```

```
public static void main (String[] args) throws Exception{

    //configuraciones de audio y sonido, los true es para que esten
    en bigIndian

    // Define el formato de audio (debe coincidir con el del
    reproductor)

    AudioFormat format = new AudioFormat(44100,16,1,true,true);

    //se utiliza el formato creado (format) para preparar la linea
    del microfono

    DataLine.Info infoMic = new DataLine.Info(TargetDataLine.class,
    format);

    //se conecta con el microfono mediante el infoSpeaker

    TargetDataLine mic = (TargetDataLine)
    AudioSystem.getLine(infoMic);

    // // Crea un socket UDP ya que es informacion mas instantanea
    y no se necesita TCP

    DatagramSocket sc = new DatagramSocket();

    //se define una direccion ip (depende del servidor)

    InetAddress address = InetAddress.getByName("127.0.0.1");

    //abre el microfono y el speaker y los inicializa

    mic.open();

    mic.start();
```

```
//crea una array de bytes para capturar audio

byte[] buffer = new byte[1024];


// Bucle infinito de captura y envío

while (true) {

    // Lee datos del micrófono (bloqueante)

    int resp = mic.read(buffer,0,buffer.length); //utiliza el
array de bytes para leer el microfono

    //dependiendo si hay captura de datos o no en el microfono

    if (resp>0){

        System.out.println("CAPTURANDO Y ENVIANDO: " + resp +
" bytes.");

        //se crea un paquete con los bytes, el largo de los
bytes (resp), la direccin ip y el puerto

        DatagramPacket packet = new DatagramPacket(buffer,
resp, address,9090);

        //se envia

        sc.send(packet);

    }

}
```

```
}  
  
}
```

- carpeta player (player)

```
import java.net.DatagramPacket;  
  
import java.net.DatagramSocket;  
  
import javax.sound.sampled.AudioFormat;  
  
public class Player {  
  
    public static void main(String[] args) throws Exception {  
  
        AudioFormat format = new AudioFormat(44100, 16, 1, true, true);  
  
        PlayerThread thread = new PlayerThread(format);  
  
        thread.start();  
  
        thread.setPlay(true); // ✓ CORREGIDO: pasa true  
  
        DatagramSocket socket = new DatagramSocket(9090);  
  
        while(true) {  
  
            byte[] data = new byte[1024]; // ✓ Nuevo buffer cada  
iteración  
  
            DatagramPacket packet = new DatagramPacket(data,  
data.length);  
  
            socket.receive(packet);  
  
        }  
    }  
}
```

```

        System.out.println("RECIBIDO: " + packet.getLength() + "
bytes en puerto 9090.");

        // ✓ CREAR COPIA SEGURA de los datos

        byte[] audioCopy = new byte[packet.getLength()];

        System.arraycopy(data, 0, audioCopy, 0,
packet.getLength());

        thread.play(audioCopy);
    }
}
}

```

- carpeta player playerThread

```

import java.util.LinkedList;

import java.util.Queue;

import javax.sound.sampled.AudioFormat;

import javax.sound.sampled.AudioSystem;

import javax.sound.sampled.DataLine;

import javax.sound.sampled.LineUnavailableException;

import javax.sound.sampled.SourceDataLine;

//esta clase es la encargada de manejar reproducir audio.

//Recibe los datos de audio a través de una cola y los envía a la línea
de audio (altavoz).

//va a ser llamada cada vez que se quiera reproducir audio

public class PlayerThread extends Thread {

```

```

        //se crea una cola de bytes y de esa forma almacena buffers de
        audio pendientes por reproducir

        private Queue<byte[]> audioByte = new LinkedList<>();

        //objeto para recibir informacion que recibe el altavoz

        private DataLine.Info infoSpeaker;

        //Flag para controlar reproducción

        private boolean isPlay;

        //objeto que se encarga de controlar el altavoz

        private SourceDataLine speaker;

        // Constructor: Recibe el formato de audio y prepara la línea de
        altavoz (SourceDataLine) (controla altavoz)

        public PlayerThread (AudioFormat format) throws
        LineUnavailableException{

            // Crea un DataLine.Info para la línea de altavoz con el
            formato dado (el formato depende del formato de audio)

            //esto es lo que vamos a utilizar para configurar el altavoz

            infoSpeaker = new DataLine.Info(SourceDataLine.class, format);

            // Obtiene la línea de altavoz del sistema (se conecta con
            altavoz)

            speaker = (SourceDataLine) AudioSystem.getLine(infoSpeaker);

            // Abre la línea (adquiere los recursos del sistema)

            speaker.open();

            // Inicia la línea (permite que comience a recibir datos)

            speaker.start();

        }

        //metodo set para atributo booleano

```



```

public void setPlay(boolean isPlay){

    this.isPlay = isPlay;

}

// Método play: Añade un chunk de audio (array de bytes) a la cola,
añade bytes

public void play(byte[] batch){

    audioByte.add(batch) ;

}

// Método run: Bucle principal del hilo

@Override

public void run(){

    while (true) {

        try{

            if(isPlay){ // Si está activada la reproducción

                if(!audioByte.isEmpty()){ // y si ya hay datos en
la cola

                    byte[] current = audioByte.poll(); //toma el
primer grupo de bytes de la cola

                    // Escribe los datos en la línea de altavoz
(reproducción)

                    speaker.write(current, 0, current.length);

```

```
        }

        }else{

            // Si no está reproduciendo, espera 5 segundos y
vuelve a comprobar

            Thread.sleep(5000);

        }

    }catch(Exception e){

        e.printStackTrace();

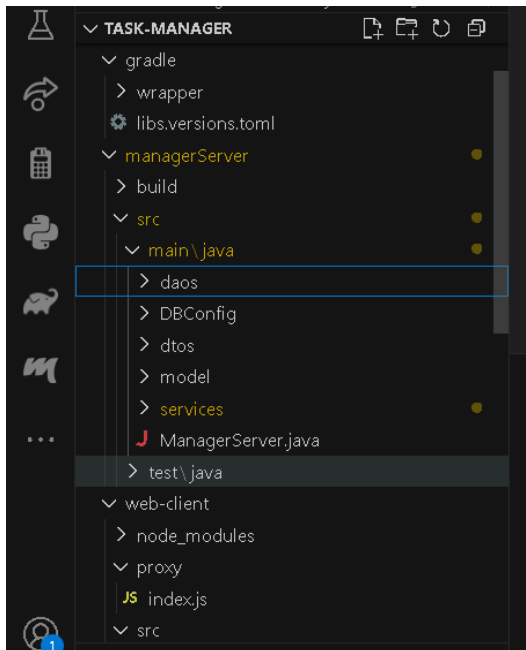
    }

}

}

}
```

TASK MANAGER :



- **build gradle:**

```
/*
 * This file was generated by the Gradle 'init' task.
 *
 * This is a general purpose Gradle build.
 *
 * To learn more about Gradle by exploring our Samples at
https://docs.gradle.org/8.6/samples
 */

subprojects{

    apply plugin : 'java'

    repositories{

        mavenCentral()

    }

    dependencies{

        // https://mvnrepository.com/artifact/org.postgresql/postgresql
        implementation 'org.postgresql:postgresql:42.7.8'
    }
}
```

```

        implementation 'com.google.code.gson:gson:2.10.1'

        testImplementation 'junit:junit:4.13.2'

    }

    jar{

        manifest{

            attributes(

                'Main-Class': project.name.capitalize()

            )

        }

        from {

            configurations.runtimeClasspath.collect { it.isDirectory()
? it : zipTree(it) }

        }

        duplicatesStrategy = DuplicatesStrategy.EXCLUDE

    }

}

```

- **settings**

```

rootProject.name = 'task-manager'

include('managerServer')

```

- **paquete DAO (DAO):**

```

package daos;

//1. Paquete daos (Data Access Objects)

//Este paquete contiene las clases e interfaces que se encargan de la
interacción con los datos

```

```

// (ya sea en memoria o en base de datos).

/*
Interfaz Dao<T, P>: Define las operaciones básicas de CRUD (Create,
Read, Update, Delete) para cualquier entidad.

findAll(): Devuelve una lista de todas las entidades.

findById(P id): Busca una entidad por su id.

update(T newEntity): Actualiza una entidad.

delete(T entity): Elimina una entidad.

save(T entity): Guarda una nueva entidad.
*/

import java.util.List;

public interface Dao<T, P> {

    public List<T> findAll();

    public T findById(P id);

    public T update(T newEntity);

    public void delete(T entity);

    public void save(T entity);
}

```

- **paquete DAO (StageDao):**

```
package daos;

/*
Clase StageDao: Implementa Dao<TaskStage, Integer>. Gestiona las etapas
(TaskStage) en memoria.

En el constructor, crea tres etapas predeterminadas: TO DO, IN
PROGRESS, DONE.

Utiliza una lista en memoria (stages) y un contador para asignar ids.

SE UTILIZA PARA ACTUALIZAR Y MANEJAR TaskStage (maneja estados de las
task y tasks como tal)
*/

import java.util.ArrayList;
import java.util.List;

import model.TaskStage;

//esta clase se encarga de crear los TaskDtage que vamos a utilizar
//ESTA CLASE ES EL CONTROLADOR DE TASKSTAGE
public class StageDao implements Dao<TaskStage, Integer>{

    private List<TaskStage> stages = new ArrayList<>();

    private int count = 0;

    //como por default la aplicacion utiliza tres estados, los crea
    desde el inicio
```

```
public StageDao() {

    //crea el TaskStage para el estado TO DO

    TaskStage todo = new TaskStage();

    todo.setName("TO DO");

    todo.setDescription("Tasks to be done");

    save(todo);

    //crea el TaskStage para el estado IN PROGRESS

    TaskStage inProgress = new TaskStage();

    inProgress.setName("IN PROGRESS");

    inProgress.setDescription("Tasks in progress");

    save(inProgress);

    //crea el TaskStage para el estado DONE

    TaskStage done = new TaskStage();

    done.setName("DONE");

    done.setDescription("Completed tasks");

    save(done);

}

//tiene una lista de TaskStages

@Override

public List<TaskStage> findAll() {

    return stages;

}

//te encuentra tu STAGE mediante su id
```

```
@Override

public TaskStage findById(Integer id) {

    //metodo para filtrar

    return stages.stream()

        .filter(s -> s.getId() == id)

        .findFirst().orElse(null);

}


@Override

public TaskStage update(TaskStage newEntity) {

    TaskStage exist = findById(newEntity.getId());

    if(exist != null){

        exist.setDescription(newEntity.getDescription());

        exist.setName(newEntity.getName());

        exist.setTasks(newEntity.getTasks());

    }

    return exist;

}


@Override

public void delete(TaskStage entity) {

    TaskStage ent = findById(entity.getId());

    if(ent != null){

        stages.remove(ent);

    }

}
```



```

        //esta guardando

        @Override

        public void save(TaskStage entity) {

            entity.setId(++count);

            //guarda en su lista de estados

            stages.add(entity);

        }

    }
}

```

- **paquete DAO (TaskDaoDB):**

```

package daos;

/*

Clase TaskDaoDB: Implementa Dao<Task, Integer>. Está diseñada para
interactuar con una base de datos PostgreSQL.

Por ahora, solo está implementado findAll(), que obtiene todas las
tareas de la base de datos y las mapea a objetos Task.

Los demás métodos lanzan UnsupportedOperationException.

*/

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

```

```
import DBConfig.ConnectionManager;

import model.Task;

import model.TaskStage;

//guarda las task en la base de base de datos

public class TaskDaoDB implements Dao<Task, Integer>{

    //

    @Override

    public List<Task> findAll() {

        try {

            List<Task> tasks = new ArrayList<>();

            Connection conn = ConnectionManager

                .getInstance(

) .getConnection();

            String query = "Select * from task";

            Statement statement = conn.createStatement();

            ResultSet result = statement.executeQuery(query);

            while (result.next()) {

                Task t = new Task();

                //TODO completar mapeo de las entidades

                t.setId(result.getInt("id"));

                t.setTitle(result.getString("title"));

                t.setDescription(result.getString("description"));

            }

        } catch (SQLException e) {

            e.printStackTrace();

        }

        return tasks;

    }

}
```

```

        int stageID = result.getInt("stage_id");

        TaskStage stage = new TaskStage();

        stage.setId(stageID);

        tasks.add(t);
    }

    conn.close();

    return tasks;

} catch (Exception e) {

    throw new RuntimeException(e);

}

}

@Override

public Task findById(Integer id) {

    // TODO Auto-generated method stub

    throw new UnsupportedOperationException("Unimplemented method 'findById'");

}

@Override

public Task update(Task oldEntity) {

    // TODO Auto-generated method stub

    throw new UnsupportedOperationException("Unimplemented method 'update'");

}

```

```

    }

    @Override

    public void delete(Task entity) {

        // TODO Auto-generated method stub

        throw new UnsupportedOperationException("Unimplemented method 'delete'");

    }

    @Override

    public void save(Task entity) {

        // TODO Auto-generated method stub

        throw new UnsupportedOperationException("Unimplemented method 'save'");

    }

}

```

- paquete DAO (TaskMemDao) IMPLEMENTACION NICO:

```

package daos;

import model.TaskStage;

import java.util.List;

import org.w3c.dom.xpath.XPathResult;

```

```
import DBConfig.ConnectionManager;

import java.util.ArrayList;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.PreparedStatement;

public class StageDaoDB implements Dao<TaskStage, Integer> {

    public StageDaoDB() {

    }

    //MÉTODO findAll() - Obtener todas las etapas

    @Override

    public List<TaskStage> findAll(){

        // Lista para almacenar resultados

        List<TaskStage> stages = new ArrayList<>();

        String query = "SELECT * FROM task_stage";

        //intenta realizar la conexion
```

```
try(Connection conn =
ConnectionManager.getInstance().getConnection()){

    // Crear statement

    //Un Statement es un objeto que representa una sentencia SQL.

    //e crea a partir de una conexión a la base de datos

    // se utiliza para ejecutar consultas SQL (como SELECT, INSERT,
UPDATE, DELETE) y obtener los resultados.

    Statement statement = conn.createStatement();

    // Ejecutar consulta

    ResultSet result = statement.executeQuery(query);

    System.out.println(result);

    // Iterar sobre cada fila del resultado

    while (result.next()) {

        // Crear nueva instancia

        TaskStage stage = new TaskStage();

        // Obtener columna "id" como int

        stage.setId(result.getInt("id"));

        // Obtener columna "id" como int

        stage.setName(result.getString("name"));

        stage.setDescription(result.getString("description"));

    }

}
```

```
        stages.add(stage);

    }

}

catch(Exception e) {

}

return stages;

}

@Override

public TaskStage findById(Integer id){

    TaskStage stage = null;

    String query = "SELECT * FROM task_stage WHERE id == ?";

    try(Connection conn = ConnectionManager.

        getInstance().getConnection()){

        PreparedStatement statement = conn.prepareStatement(query);

        statement.setInt(1, id);

        try(ResultSet result = statement.executeQuery()){

            if (result.next()) {
```

```
        stage = new TaskStage();

        stage.setId(result.getInt("id"));

        stage.setName(result.getString("name"));

        stage.setDescription(result.getString("description"));

    }

}

}

catch (Exception e) {

}

return stage;

}

@Override

public TaskStage update(TaskStage newEntity) {

    throw new UnsupportedOperationException("La actualización de  
stages no está permitida.");

}

@Override

public void delete(TaskStage entity) {

    throw new UnsupportedOperationException("La eliminación de  
stages no está permitida.");

}
```



```

    }

    @Override

    public void save(TaskStage entity) {

        throw new UnsupportedOperationException("La creación de nuevos
stages no está permitida.");

    }

}

```

- paquete DAO (TaskMemDao):

```

package daos;

/*
Clase TaskMemDao: Implementa Dao<Task, Integer>. Gestiona las tareas
(Task) en memoria.

*/

import java.util.ArrayList;
import java.util.List;

import model.Task;

//ES EL CONTROLADOR DE TASK
public class TaskMemDao implements Dao<Task, Integer>{

```

```
//maneja una lista de Task

private List<Task> tasks = new ArrayList<>();

private int count = 0;

//devuelve la lista de task

@Override

public List<Task> findAll() {

    return tasks;

}

//encuentra una task por su id

@Override

public Task findById(Integer id) {

    //metodo de busqueda de una linea

    return tasks.stream().filter(t -> t.getId() ==
id).findFirst().orElse(null);

}

//metodo para actualizar

@Override

public Task update(Task newEntity) {

    Task exist = findById(newEntity.getId());

    if(exist != null){

        exist.setDescription(newEntity.getDescription());

        exist.setTitle(newEntity.getTitle());

        exist.setStage(newEntity.getStage());

        exist.setDueDate(newEntity.getDueDate());

    }

}
```

```
        exist.setPriority(newEntity.getPriority());

    }

    return exist;

}

//metodo para eliminar de lista

@Override

public void delete(Task entity) {

    Task ent = findById(entity.getId());

    if(ent != null){

        tasks.remove(ent);

    }

}

//metodo para guardar en lista

public void save(Task entity){

    //utilizamos el contador global para manejar para manejar el id
de la task nueva que se guarda en el computador

    entity.setId(++count);

    //lo guarda en la lista de task

    tasks.add(entity);

}

}
```

- **paquete DBConfig (ConnectionManager):**

```
package DBConfig;

/*
2. Paquete DBConfig
Contiene la configuración para la conexión a la base de datos.
*/

import java.sql.Connection;
import java.sql.DriverManager;

/*
Clase ConnectionManager: Gestiona la conexión a la base de datos usando
el patrón Singleton.

Proporciona métodos estáticos para obtener una instancia, con o sin
parámetros (url, usuario, contraseña).

El método getConnection() devuelve una conexión JDBC a PostgreSQL.
*/

//METODO PARA BASE DE DATOS
public class ConnectionManager {

    private String url;

    private String user;

    private String password;
```

```
private static ConnectionManager manager;

public static ConnectionManager getInstance(){

    if(manager == null){

        manager = new ConnectionManager();

    }

    return manager;

}

public static ConnectionManager getInstance(String url, String
user, String password){

    if(manager == null){

        manager = new ConnectionManager(url, user, password);

    }

    return manager;

}

private ConnectionManager(String url, String user, String
password){

    this.url = url;

    this.user = user;

    this.password = password;

}

private ConnectionManager(){

    this.url = System.getenv("url");

    this.user = System.getenv("user");

}
```

```

        this.password = System.getenv("password");
    }

    public Connection getConnection(){

        try {

            Class.forName("org.postgresql.Driver");

            Connection con = DriverManager.getConnection(url, user,
password);

            return con;

        } catch (Exception e) {

            e.printStackTrace();

        }

        return null;
    }
}

```

- paquete de dtos (Request):

```

package dtos;

/*

3. Paquete dtos (Data Transfer Objects)

Contiene clases que se utilizan para transferir datos entre el cliente
y el servidor.

*/

import com.google.gson.JsonObject;

import com.google.gson.annotations.Expose;

```

```
public class Request {

    /*
    Clase Request: Representa una solicitud del cliente.

    Tiene un campo command (String) que indica la operación a realizar.

    Un campo data (JsonObject) que contiene los datos necesarios para
    la operación.
    */

    //ESTA CLASE SOLO EXISTE PARA MANEJAR LOS COMANDOS

    //recibe un comando y un objeto json
    @Expose private String command;
    @Expose private JsonObject data;

    //constrcutor
    public Request(String command, JsonObject data) {
        this.command = command;
        this.data = data;
    }

    public String getCommand() {
        return command;
    }

    public void setCommand(String command) {
        this.command = command;
    }
}
```

```

    public JsonObject getData() {

        return data;

    }

    public void setData(JsonObject data) {

        this.data = data;

    }

}

```

- **paquete model (task):**

```

package model;

/*
4. Paquete model

Contiene las clases que representan las entidades del dominio.

*/

import com.google.gson.annotations.Expose;

/*
Clase Task: Representa una tarea.

Campos: id, título, descripción, fecha de vencimiento, prioridad y una
etapa (TaskStage).

Anotaciones @Expose de Gson para controlar la
serialización/deserialización.

*/

```



```
public class Task {

    @Expose private int id;

    @Expose private String title;

    @Expose private String description;

    @Expose private String dueDate;

    @Expose private String priority;

    //

    @Expose(serialize = false, deserialize = true)
    private TaskStage stage;

    public Task() {

    }

    public String getDueDate() {

        return dueDate;

    }

    public String getPriority() {

        return priority;

    }

    public void setDueDate(String dueDate) {

        this.dueDate = dueDate;

    }

}
```

```
public void setPriority(String priority) {  
    this.priority = priority;  
}  
  
public void setId(int id) {  
    this.id = id;  
}  
  
public int getId() {  
    return id;  
}  
  
public void setTitle(String name) {  
    this.title = name;  
}  
  
public String getDescription() {  
    return description;  
}  
  
public void setDescription(String description) {  
    this.description = description;  
}  
  
public TaskStage getStage() {  
    return stage;  
}
```

```

    public void setStage(TaskStage stage) {

        this.stage = stage;

    }

    public String getTitle() {

        return title;

    }

    @Override

    public String toString() {

        return id + " - " + title;

    }

}

```

- **paquete model (TaskStage):**

```

package model;

import java.util.ArrayList;
import java.util.List;

import com.google.gson.annotations.Expose;

/*
Clase TaskStage: Representa una etapa de tarea (por ejemplo, TO DO, IN
PROGRESS, DONE).

```

Campos: id, nombre, descripción y una lista de tareas (Task) en esa etapa.

CONTIENE UNA LISTA DE Tasks, contiene metodos para afectar a task individuales

```
*/

public class TaskStage {

    //el expose es para evitar el overflow de json

    @Expose private int id;

    @Expose private String name;

    @Expose private String description;

    @Expose private List<Task> tasks;

    //constructor de taskStage

    public TaskStage(int id, String name, String description) {

        this.id = id;

        this.name = name;

        this.description = description;

    }

    //segundo constructor vacio

    public TaskStage() {

    }

    public List<Task> getTasks() {

        if (tasks == null) {
```

```
        tasks = new ArrayList<>();

    }

    return tasks;

}

public void setTasks(List<Task> tasks) {

    this.tasks = tasks;

}

public void setId(int id) {

    this.id = id;

}

public int getId() {

    return id;

}

public String getName() {

    return name;

}

public String getDescription() {

    return description;

}

public void setName(String name) {

    this.name = name;

}

public void setDescription(String description) {

    this.description = description;

}
```

```
}  
  
}
```

- **paquete services (TaskServices):**

```
package services;  
  
/*  
5. Paquete services  
Contiene la lógica de negocio.  
*/  
  
import java.util.List;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Semaphore;  
  
import daos.Dao;  
import daos.StageDao;  
import daos.TaskMemDao;  
import model.Task;  
import model.TaskStage;  
  
/*  
Clase TaskServices: Orquesta las operaciones entre las tareas y las  
etapas.  
  
Utiliza dos DAOs: uno para tareas (en memoria) y otro para etapas.
```

Métodos:

saveTask(Task t): Guarda una tarea y la asocia a una etapa.

changeStage(int taskId, int stageId): Cambia la etapa de una tarea.

getTask(): Devuelve todas las etapas con sus tareas.

*/

```
public class TaskServices {

    //es una interfaz creada para objetos tipo task
    private Dao<Task,Integer> repository;

    //es una interfaz para objetos tipo TaskStage
    private Dao<TaskStage,Integer> repositoryStage;

    public TaskServices(){

        //se crea el controlador de task
        repository = new TaskMemDao();

        //se crea el controlador de taskStage
        repositoryStage = new StageDao();

    }

    //guarda una task mediante el controlador de task
    public Task saveTask(Task t){

        //una task se realcion con un taskStage, entonces se trae al
        taskStage que esta relacionado con esa task

        TaskStage s = repositoryStage.findById(t.getStage().getId());
```

```

        //guarda la nueva task en su taskStage

        s.getTasks().add(t);

        //setea el stage de esa task segun su taskStage

        t.setStage(s);

        //lo guarda en el controlador

        repository.save(t);

        return t;
    }

    //esta actualia la task

    //recibe un int de taskId para encontrar la task especifica

    //recibe un int de stageId para encontrar el stageTask especifico
    public Task changeStage(int taskId, int stageId){

        //se encuentra una task mediante de task

        Task t = repository.findById(taskId);

        //se utiliza esa task para encontrar su stage actual

        TaskStage oldStage = t.getStage();

        //se remueve su stage actual

        oldStage.getTasks().remove(t);

        //se encuentra el nuevo stageTask en su controlados mediante el
int

        TaskStage s = repositoryStage.findById(stageId);

        //se le adiciona la task al stageTask

        s.getTasks().add(t);

        //se setea

        t.setStage(s);

        //se actualiza el controlador de task

        repository.update(t);
    }

```



```

        return t;
    }

    //regresa todas las task del controlador de task
    public List<TaskStage> getTask() {
        return repositoryStage.findAll();
    }
}

```

- **clase main (implementando lambda thread pool):**

```

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

import com.google.gson.Gson;

```

```
import com.google.gson.GsonBuilder;

import com.google.gson.JsonObject;

import dtos.Request;

import model.Task;

import model.TaskStage;

import services.TaskServices;

public class ManagerServer {

    private Gson gson;

    private TaskServices services;

    private boolean running;

    // ✓ NUEVO: Thread Pool para manejar múltiples clientes
    private ExecutorService threadPool;

    // ✓ NUEVO: Semáforo para controlar acceso a recursos compartidos
    private Semaphore connectionSemaphore;

    // ✓ NUEVO: Configuración
    private static final int MAX_CONCURRENT_CLIENTS = 10;

    private static final int THREAD_POOL_SIZE = 20;

    private static final int SERVER_PORT = 5000;

    public static void main(String[] args) throws Exception {

        new ManagerServer();
    }
}
```

```

    }

    public ManagerServer() throws Exception {

        // Se inicializan el Gson (para serializar/deserializar JSON)

        gson = new
GsonBuilder().excludeFieldsWithoutExposeAnnotation().create();

        // se inicializa el TaskService

        services = new TaskServices();

        // ✅ NUEVO: Inicializar Thread Pool y Semáforo

        threadPool = Executors.newFixedThreadPool(THREAD_POOL_SIZE);

        connectionSemaphore = new Semaphore(MAX_CONCURRENT_CLIENTS);

        System.out.println("🚀 Servidor iniciado con:");

        System.out.println("    - Thread Pool size: " +
THREAD_POOL_SIZE);

        System.out.println("    - Máximo clientes concurrentes: " +
MAX_CONCURRENT_CLIENTS);

        System.out.println("    - Puerto: " + SERVER_PORT);

        ServerSocket socket = new ServerSocket(SERVER_PORT);

        running = true;

        while (running) {

            // ✅ MODIFICADO: Ahora manejamos cada cliente en un hilo
separado

            Socket clientSocket = socket.accept();

            handleNewClient(clientSocket);

        }
    }

```

```

        // ✅ NUEVO: Cierre ordenado

        gracefulShutdown(socket);

    }

    // ✅ NUEVO: Método para manejar nuevos clientes con lambda
    private void handleNewClient(Socket clientSocket) {

        String clientInfo = clientSocket.getInetAddress() + ":" +
clientSocket.getPort();

        try {

            // Intentar adquirir permiso del semáforo (máximo 10
clientes)

            if (connectionSemaphore.tryAcquire(5, TimeUnit.SECONDS)) {

                // ✅ LAMBDA: Enviar cliente al Thread Pool

                threadPool.execute(() -> {

                    try {

                        System.out.println("🔗 Cliente conectado: " +
clientInfo +

                                                " [Hilo: " +
Thread.currentThread().getName() + "]" );

                        System.out.println("📊 Semáforo - Permisos
disponibles: " +
connectionSemaphore.availablePermits());

                        // Procesar el cliente

                        resolveClient(clientSocket);

```

```

        } catch (Exception e) {

            System.err.println("❌ Error con cliente " +
clientInfo + ": " + e.getMessage());

        } finally {

            // ✅ IMPORTANTE: Liberar permiso del semáforo
            connectionSemaphore.release();

            System.out.println("🔓 Cliente desconectado: "
+ clientInfo +

                                " [Permisos: " +
connectionSemaphore.availablePermits() + "]");

        }

    });

    } else {

        // ✅ NUEVO: Rechazar conexión si hay demasiados
clientes

        System.out.println("❌ Cliente rechazado - Límite
alcanzado: " + clientInfo);

        rejectClient(clientSocket, "Servidor ocupado. Intente
más tarde.");

    }

    } catch (InterruptedException e) {

        System.err.println("🛑 Interrupción durante adquisición de
semáforo");

        Thread.currentThread().interrupt();

        rejectClient(clientSocket, "Servidor interrumpido");

    }

}

```

```
// Método para rechazar clientes cuando hay sobrecarga

private void rejectClient(Socket clientSocket, String message) {

    try {

        BufferedWriter writer = new BufferedWriter(

            new

OutputStreamWriter(clientSocket.getOutputStream()));

        Map<String, String> error = Map.of("error", message);

        String response = gson.toJson(error);

        writer.write(response + "\n");

        writer.flush();

        clientSocket.close();

    } catch (IOException e) {

        System.err.println("Error rechazando cliente: " +

e.getMessage());

    }

}

//Cierre ordenado del servidor

private void gracefulShutdown(ServerSocket socket) {

    System.out.println("🛑 Apagando servidor...");

    running = false;

    try {

        // Cerrar Thread Pool de forma ordenada

        threadPool.shutdown();

    }

}
```

```

        if (!threadPool.awaitTermination(30, TimeUnit.SECONDS)) {

            System.out.println("⚠ Forzando cierre de hilos
pendientes...");

            threadPool.shutdownNow();

        }

        // Cerrar socket

        if (socket != null && !socket.isClosed()) {

            socket.close();

        }

        System.out.println("✅ Servidor apagado correctamente");

    } catch (Exception e) {

        System.err.println("Error durante apagado: " +
e.getMessage());

    }

}

// Ahora se ejecuta en un hilo del Thread Pool

public void resolveClient(Socket sc) throws IOException {

    try {

        //creamos un BufferedReader para poder recibir un mensaje
del servidor

        /*

        * sc.getInputStream(): Obtiene stream de bytes de entrada
del socket

        InputStreamReader: Convierte bytes a caracteres
(decodificación)

```

BufferedReader: Aplica buffering para lectura eficiente por líneas

Propósito: Leer respuestas del servidor

*/

//el br optiene lo que se recibe del socket

```
BufferedReader br = new BufferedReader(new  
InputStreamReader(sc.getInputStream()));
```

//creamos un BufferedWriter para poder enviar un mensaje al servidor

```
BufferedWriter writer = new BufferedWriter(new  
OutputStreamWriter(sc.getOutputStream()));
```

```
while (true) {
```

//guardamos lo que recibimos del BufferedReader en un String

```
String line = br.readLine();
```

//Validación más robusta

```
if (line == null) {
```

```
    System.out.println("🚩 Cliente cerró conexión");
```

```
    break;
```

```
}
```

```
if (line.trim().isEmpty()) {
```

```
    System.out.println("🚩 Línea vacía recibida");
```

```
    continue;
```

```
}
```



```

        try {

            //El servidor lee una línea (JSON) del cliente y la
            convierte en un objeto Request.

            //comvierte a (line) en un objeto tipo request

            //AQUI SE CREA EL REQUEST que se utiliza para
            recibir y manejar los COMANDOS

            Request rq = gson.fromJson(line, Request.class);

            // Validación del request

            if (rq.getCommand() == null) {

                Map<String, String> error = Map.of("error",
"Command is required");

                writer.write(gson.toJson(error) + "\n");

                writer.flush();

                continue;

            }

            //System.out.println("🎯 Comando recibido: " +
            rq.getCommand() +

                // " [Hilo: " +
                Thread.currentThread().getName() + "]" );

            //es objeto request lo convierte en un objeto json
            (JsonObject)

            //se utiliza para tener el data de json

            JsonObject obj = rq.getData();

            //crea un string nulo para poder utilizarlo para
            guardar infromacion

            String resp = null;

```

```

        //Sincronización para operaciones thread-safe

        switch (rq.getCommand()) {

            //se realiza un switch segun el tipo de comando

            case "CREATE_TASK":

                //Convierte el data a un Task

                //se recibe una task desde json (obj)

                Task t = gson.fromJson(obj, Task.class);

                //se llama a services = (TaskServices) para
guardar la nueva task en la base de datos (controlador)

                synchronized (services) {

                    //conviertes la task a gson y la
guardas en resp

                    t = services.saveTask(t);

                }

                resp = gson.toJson(t);

                break;

            case "UPDATE_TASK":

                //mediante el data que trae obj se obtiene
el id de task

                String taskId =
obj.get("taskId").getAsString();

                //tambien obtiene el numero del stage

                int stage = obj.get("stage").getAsInt();

                synchronized (services) {

                    //utiliza el controller de la task para
actualizar su stage

                    t =
services.changeStage(Integer.parseInt(taskId), stage);

                }

```

```

        //regresa la task actualizada y se guarda
en resp

        resp = gson.toJson(t);

        break;

    case "GET_TASKS":

        List<TaskStage> stages;

        synchronized (services) {

            //este metodo me regresa todas las task
guardadas en service

            stages = services.getTask();

        }

        //las convierte en gson y las guarda en
gson

        resp = gson.toJson(stages);

        break;

    case "HELLO":

        Map<String,String> hello =
Map.of("message", "Hello from Java server");

        resp = gson.toJson(hello);

        break;

    case "SHUTDOWN":

        //  NUEVO: Comando para apagar servidor
remotamente

        running = false;

        Map<String,String> shutdown =
Map.of("message", "Server shutting down");

        resp = gson.toJson(shutdown);

        break;

    case "STATUS":

```

```

//  NUEVO: Comando para ver estado del
servidor

        Map<String, Object> status = new
HashMap<>();

        status.put("activeThreads",
Thread.activeCount());

        status.put("availablePermits",
connectionSemaphore.availablePermits());

        status.put("queueSize",
((java.util.concurrent.ThreadPoolExecutor)
threadPool).getQueue().size());

        resp = gson.toJson(status);

        break;

    default:

        Map<String, String> map = new HashMap<>();

        map.put("error", "Operation not supported:
" + rq.getCommand());

        resp = gson.toJson(map);

        break;

    }

    writer.write(resp + "\n");

    writer.flush();

} catch (Exception e) {

    System.err.println("✗ Error procesando request: "
+ e.getMessage());

    Map<String, String> error = Map.of("error",
"Processing error: " + e.getMessage());

    writer.write(gson.toJson(error) + "\n");

    writer.flush();

```

```

        }

    }

    } catch (Exception e) {

        System.err.println("💥 Error en conexión con cliente: " +
e.getMessage());

    } finally {

        try {

            sc.close();

        } catch (IOException e) {

            System.err.println("Error cerrando socket: " +
e.getMessage());

        }

    }

}

}
}

```

- clase main (sin thread pool):

```

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

```

```
import java.util.concurrent.Semaphore;
```

```
import java.util.concurrent.TimeUnit;
```

```
import com.google.gson.Gson;
```

```
import com.google.gson.GsonBuilder;
```

```
import com.google.gson.JsonObject;
```

```
import dtos.Request;
```

```
import model.Task;
```

```
import model.TaskStage;
```

```
import services.TaskServices;
```

```
/*
```

6. Clase ManagerServer

Es el punto de entrada del servidor.

Crea un ServerSocket en el puerto 5000.

Espera conexiones de clientes y para cada una, lanza el método resolveClient.

resolveClient(Socket sc): Lee las solicitudes del cliente, las procesa y envía respuestas.

Las solicitudes son objetos JSON que se convierten a Request.

Según el comando, realiza una operación (crear tarea, cambiar etapa, obtener tareas, etc.) y devuelve una respuesta JSON.

```
*/
```

```
public class ManagerServerSinThread {

    private Gson gson;

    private TaskServices services;

    private boolean running;

    //NUEVO: Thread Pool para manejar múltiples clientes
    private ExecutorService threadPool;

    //NUEVO: Semáforo para controlar acceso a recursos compartidos
    private Semaphore connectionSemaphore;

    // Configuración de puertos y tamaños
    private static final int MAX_CONCURRENT_CLIENTS = 10;
    private static final int THREAD_POOL_SIZE = 20;
    private static final int SERVER_PORT = 5000;

    public static void main(String[] args) throws Exception {

        //Se inicia el ManagerServer que escucha en el puerto 5000.
        new ManagerServerSinThread();
    }

    public ManagerServerSinThread() throws Exception {

        //Se inicializan el Gson (para serializar/deserializar JSON)

        gson = new
GsonBuilder().excludeFieldsWithoutExposeAnnotation().create();

        //se inicializa el TaskService
```

```

services = new TaskServices();

// Inicializar Thread Pool y Semáforo
threadPool = Executors.newFixedThreadPool(THREAD_POOL_SIZE);
connectionSemaphore = new Semaphore(MAX_CONCURRENT_CLIENTS);

//crea el socket que se maneja en el puerto 5000
//escuchas en el puerto 5000
ServerSocket socket = new ServerSocket(SERVER_PORT);

running = true;
while (running) {

    //aquí se espera a que se acepte un socket
    Socket clientSocket = socket.accept();

    handleNewClient(clientSocket);
}

}

public void resolveClient(Socket sc) throws IOException {
    try {
        //creamos un BufferedReader para poder recibir un mensaje
del servidor

        /*
        * sc.getInputStream(): Obtiene stream de bytes de entrada
del socket

```


InputStreamReader: Convierte bytes a caracteres
(decodificación)

BufferedReader: Aplica buffering para lectura eficiente por
líneas

Propósito: Leer respuestas del servidor

*/

//el br optiene lo que se recibe del socket

```
BufferedReader br = new BufferedReader(new  
InputStreamReader(sc.getInputStream()));
```

//creamos un BufferedWriter para poder enviar un mensaje al
servidor

```
BufferedWriter writer = new BufferedWriter(new  
OutputStreamWriter(sc.getOutputStream()));
```

```
while (true) {
```

//guardamos lo que recibimos del BufferedReader en un
String

```
String line = br.readLine();
```

//El servidor lee una línea (JSON) del cliente y la
convierte en un objeto Request.

//convierte a (line) en un objeto tipo request

//AQUI SE CREA EL REQUEST que se utiliza para recibir y
manejar los COMANDOS

```
Request rq = gson.fromJson(line, Request.class);
```

```
System.out.println(rq.getCommand());
```

```
System.out.println(rq.getData());
```

```

        //es objeto request lo convierte en un objeto json
(JsonObject)

        //se utiliza para tener el data de json

        JsonObject obj = rq.getData();

        //crea un string nulo para poder utilizarlo para
guardar informacion

        String resp = null;

        try {

            //se realiza un switch segun el tipo de comando

            switch (rq.getCommand()) {

                case "CREATE_TASK":

                    //Convierte el data a un Task

                    //se recibe una task desde json (obj)

                    Task t = gson.fromJson(obj, Task.class);

                    synchronized (services) {

                        //se llama a services = (TaskServices)
para guardar la nueva task en la base de datos (controlador)

                        t = services.saveTask(t);

                        //conviertes la task a gson y la
guardas en resp

                    }

                    resp = gson.toJson(t);

                    break;

                case "UPDATE_TASK":

                    //mediante el data que trae obj se obtiene
el id de task

                    String taskId =
obj.get("taskId").getAsString();

                    //tambien obtiene el numero del stage

```

```

        int stage = obj.get("stage").getAsInt();

        synchronized (services) {

            //utiliza el controller de la task para
actualizar su stage

            t =
services.changeStage(Integer.parseInt(taskId), stage);

        }

        //regresa la task actualizada y se guarda
en resp

        resp = gson.toJson(t);

        break;

    case "GET_TASKS":

        List<TaskStage> stages;

        synchronized (services) {

            //este metodo me regresa todas las task
guardadas en service

            stages = services.getTask();

        }

        //las convierte en gson y las guarda en
gson

        resp = gson.toJson(stages);

        break;

    case "HELLO":

        //hace un mapeo de strings para enviar un
mensaje de hola

```

```

        Map<String,String> hello =
Map.of("message", "Hello from Java server");

        resp = gson.toJson(hello);

        break;

        default:

            //hace un mapeo de strings para enviar un
mensaje de error

            Map<String, String> map = new HashMap<>();

            map.put("msg", "Operation not supported");

            resp = gson.toJson(map);

            break;

        }

    } catch (Exception e) {

        Map<String,String> error = Map.of("error",
e.getMessage());

        resp = gson.toJson(error);

        e.printStackTrace();

    }

    //envias el resp

    writer.write(resp+"\n");

    writer.flush();

}

} catch (Exception e) {

    e.printStackTrace();

    sc.close();

}

}

// Método para manejar nuevos clientes con lambda

```

```
private void handleNewClient(Socket clientSocket) {

    //informacion del cliente

    String clientInfo = clientSocket.getInetAddress() + ":" +
clientSocket.getPort();

    try {

        // Intentar adquirir permiso del semáforo (máximo 10
clientes)

        if (connectionSemaphore.tryAcquire(5, TimeUnit.SECONDS)) {

            // LAMBDA: Enviar cliente al Thread Pool

            threadPool.execute(() -> {

                try {

                    // Procesar el cliente

                    resolveClient(clientSocket);

                } catch (Exception e) {

                    System.err.println(" Error con cliente " +
clientInfo + ": " + e.getMessage());

                } finally {

                    // Liberar permiso del semáforo

                    connectionSemaphore.release();

                }

            });

        } else {

            // Rechazar conexión si hay demasiados clientes
```

```

        rejectClient(clientSocket, "Servidor ocupado. Intente
más tarde.");

    }

    } catch (InterruptedException e) {

        System.err.println("■ Interrupción durante adquisición de
semáforo");

        Thread.currentThread().interrupt();

        rejectClient(clientSocket, "Servidor interrumpido");

    }

}

//NUEVO: Método para rechazar clientes cuando hay sobrecarga
private void rejectClient(Socket clientSocket, String message) {

    try {

        BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(clientSocket.getOutputStream()));

        Map<String, String> error = Map.of("error", message);

        String response = gson.toJson(error);

        writer.write(response + "\n");

        writer.flush();

        clientSocket.close();

    } catch (IOException e) {

        System.err.println("Error rechazando cliente: " +
e.getMessage());

    }
}

```

```
}  
  
}
```

- **clase main (implementando clase cliente externa):**

```
// ManagerServer.java  
  
import java.io.BufferedWriter;  
import java.io.IOException;  
import java.io.OutputStreamWriter;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.util.Map;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.Semaphore;  
import java.util.concurrent.TimeUnit;  
  
import com.google.gson.Gson;  
import com.google.gson.GsonBuilder;  
  
import services.ClientHandler;  
import services.TaskServices;  
  
/**  
 * Servidor principal que maneja conexiones entrantes con Thread Pool y  
 * Semáforos  
 */  
  
public class ManagerServer {
```

```
// ✓ COMPONENTES EXISTENTES

private Gson gson;

private TaskServices services;

private boolean running;


// ✓ NUEVOS: Componentes para concurrencia

private ExecutorService threadPool;

private Semaphore connectionSemaphore;


// ✓ CONFIGURACIÓN

private static final int MAX_CONCURRENT_CLIENTS = 10;

private static final int THREAD_POOL_SIZE = 20;

private static final int SERVER_PORT = 5000;


public static void main(String[] args) throws Exception {

    new ManagerServer();

}


public ManagerServer() throws Exception {

    initializeComponents();

    startServer();

}


/**
 * ✓ NUEVO: Inicializa todos los componentes del servidor
 */
```



```

private void initializeComponents() {

    // Componentes existentes

    gson = new
GsonBuilder().excludeFieldsWithoutExposeAnnotation().create();

    services = new TaskServices();

    // ✅ NUEVOS: Componentes de concurrencia

    threadPool = Executors.newFixedThreadPool(THREAD_POOL_SIZE);

    connectionSemaphore = new Semaphore(MAX_CONCURRENT_CLIENTS);

    System.out.println("🚀 Servidor iniciado con configuración:");

    System.out.println("    - Thread Pool size: " +
THREAD_POOL_SIZE);

    System.out.println("    - Máximo clientes concurrentes: " +
MAX_CONCURRENT_CLIENTS);

    System.out.println("    - Puerto: " + SERVER_PORT);

}

/**
 * ✅ NUEVO: Inicia el servidor y maneja conexiones entrantes
 */

private void startServer() throws IOException {

    ServerSocket serverSocket = new ServerSocket(SERVER_PORT);

    running = true;

    System.out.println("📡 Servidor escuchando en puerto " +
SERVER_PORT);

    try {

```

```

        while (running) {

            // Esperar nueva conexión

            Socket clientSocket = serverSocket.accept();

            // Manejar la nueva conexión

            handleNewConnection(clientSocket);

        }

    } finally {

        gracefulShutdown(serverSocket);

    }

}

/**
 *  NUEVO: Maneja cada nueva conexión de cliente
 */

private void handleNewConnection(Socket clientSocket) {

    try {

        // Intentar adquirir permiso del semáforo (con timeout)

        if (connectionSemaphore.tryAcquire(5, TimeUnit.SECONDS)) {

            //  CREAR ClientHandler externo con dependencias
            inyectadas

            ClientHandler clientHandler = new ClientHandler(

                clientSocket, gson, services, connectionSemaphore

            );

            // Ejecutar en el Thread Pool

            threadPool.execute(clientHandler);

```

```

        } else {

            // Rechazar conexión si no hay permisos disponibles

            rejectClient(clientSocket, "Servidor ocupado. Intente
más tarde.");

        }

    } catch (InterruptedException e) {

        System.err.println("❌ Interrupción durante adquisición de
semáforo: " + e.getMessage());

        rejectClient(clientSocket, "Error interno del servidor");

    }

}

/**
 * ✅ NUEVO: Rechaza clientes cuando el servidor está sobrecargado
 */

private void rejectClient(Socket clientSocket, String message) {

    try {

        System.out.println("🔒 Cliente rechazado: " +

            clientSocket.getInetAddress() + " - " +

message);

        // Enviar mensaje de error al cliente

        Map<String, String> error = Map.of("error", message);

        String errorResponse = gson.toJson(error);

        BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(clientSocket.getOutputStream()));

        writer.write(errorResponse + "\n");

```

```

        writer.flush();

        // Cerrar conexión

        clientSocket.close();

    } catch (IOException e) {

        System.err.println("❌ Error rechazando cliente: " +
e.getMessage());

    }

}

/**
 * ✅ NUEVO: Cierre ordenado del servidor
 */

private void gracefulShutdown(ServerSocket serverSocket) {

    System.out.println("🔴 Iniciando apagado ordenado del
servidor...");

    running = false;

    try {

        // 1. Cerrar el ServerSocket para no aceptar más conexiones

        if (serverSocket != null && !serverSocket.isClosed()) {

            serverSocket.close();

        }

        // 2. Apagar el Thread Pool de forma ordenada

        if (threadPool != null) {

            threadPool.shutdown();

```

```

        // Esperar a que los hilos terminen (máximo 30
segundos)

        if (!threadPool.awaitTermination(30, TimeUnit.SECONDS))
        {

            System.out.println("⚠ Forzando cierre de hilos
pendientes...");

            threadPool.shutdownNow();

        }

    }

    System.out.println("✅ Servidor apagado correctamente");

    } catch (Exception e) {

        System.err.println("❌ Error durante apagado: " +
e.getMessage());

    }

}
}

```

- paquete service :clase externa cliente (ClientHandler)

```

package services;

// ClientHandler.java

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

```

```
import java.net.Socket;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.concurrent.Semaphore;


import com.google.gson.Gson;

import com.google.gson.JsonObject;



import dtos.Request;

import model.Task;

import model.TaskStage;

import services.TaskServices;


/**
 * Maneja cada cliente en un hilo separado
 * Implementa Runnable para ejecutarse en el Thread Pool
 */
public class ClientHandler implements Runnable {

    //  DEPENDENCIAS INYECTADAS desde ManagerServer

    private final Socket clientSocket;

    private final Gson gson;

    private final TaskServices services;

    private final Semaphore connectionSemaphore;

    private final String clientInfo;
```

```
/**
 * Constructor - Recibe todas las dependencias necesarias
 */

public ClientHandler(Socket socket, Gson gson, TaskServices
services, Semaphore semaphore) {

    this.clientSocket = socket;

    this.gson = gson;

    this.services = services;

    this.connectionSemaphore = semaphore;

    this.clientInfo = socket.getInetAddress() + ":" +
socket.getPort();

}

/**
 * Método principal que se ejecuta cuando el Thread Pool inicia
este handler
 */

@Override
public void run() {

    try {

        logConnection();

        processClient();

    } catch (Exception e) {

        logError(e);

    } finally {

        cleanup();

    }

}
```

```

/**
 * Registra la conexión del cliente con información del hilo
 */

private void logConnection() {

    System.out.println("🔗 Cliente conectado: " + clientInfo +
        " [Hilo: " + Thread.currentThread().getName()
+ "]" );

    System.out.println("🚦 Semáforo - Permisos disponibles: " +
        connectionSemaphore.availablePermits());

}

/**
 * Procesa la comunicación con el cliente
 * Este método reemplaza el resolveClient original
 */

private void processClient() throws IOException {

    try (

        BufferedReader br = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

        BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(clientSocket.getOutputStream()))

    ) {

        while (true) {

            String line = br.readLine();

            // ✅ VALIDACIÓN: Si el cliente cierra conexión

            if (line == null) {

                System.out.println("🚪 Cliente cerró conexión: " +
clientInfo);

```



```

        break;

    }

    // ✅ VALIDACIÓN: Línea vacía
    if (line.trim().isEmpty()) {

        System.out.println("🚫 Línea vacía recibida de: " +
clientInfo);

        continue;

    }

    // Procesar el comando

    processCommand(line, writer);

}

} catch (Exception e) {

    System.err.println("💥 Error en conexión con cliente " +
clientInfo + ": " + e.getMessage());

    } finally {

        clientSocket.close();

    }

}

/**
 * Procesa un comando específico recibido del cliente
 */


private void processCommand(String line, BufferedWriter writer)
throws IOException {

    try {

        Request rq = gson.fromJson(line, Request.class);

```

```

        //  VALIDACIÓN: Comando nulo

        if (rq.getCommand() == null) {

            sendError(writer, "Command is required");

            return;

        }

        System.out.println("🎯 Comando recibido de " + clientInfo +
": " + rq.getCommand());

        JsonObject obj = rq.getData();

        String response = processRequest(rq.getCommand(), obj);

        // Enviar respuesta al cliente

        writer.write(response + "\n");

        writer.flush();

    } catch (Exception e) {

        System.err.println("❌ Error procesando request de " +
clientInfo + ": " + e.getMessage());

        sendError(writer, "Processing error: " + e.getMessage());

    }

}

/**
 * Procesa el request según el comando y retorna la respuesta JSON
 */

private String processRequest(String command, JsonObject data) {

    try {

```

```

switch (command) {

    case "CREATE_TASK":

        Task task = gson.fromJson(data, Task.class);

        // ✓ SINCRONIZACIÓN: Para acceso thread-safe al
servicio

        synchronized (services) {

            task = services.saveTask(task);

        }

        return gson.toJson(task);

    case "UPDATE_TASK":

        String taskId = data.get("taskId").getAsString();

        int stage = data.get("stage").getAsInt();

        Task updatedTask;

        // ✓ SINCRONIZACIÓN: Para acceso thread-safe al
servicio

        synchronized (services) {

            updatedTask =
servicios.changeStage(Integer.parseInt(taskId), stage);

        }

        return gson.toJson(updatedTask);

    case "GET_TASKS":

        List<TaskStage> stages;

        // ✓ SINCRONIZACIÓN: Para acceso thread-safe al
servicio

        synchronized (services) {

            stages = services.getTask();

        }

```

```

        return gson.toJson(stages);

        case "HELLO":

            Map<String, String> hello = Map.of("message",
"Hello from Java server");

            return gson.toJson(hello);

        case "STATUS":

            // ✅ NUEVO: Comando para ver estado del servidor

            return getServerStatus();

        default:

            Map<String, String> error = Map.of("error",
"Operation not supported: " + command);

            return gson.toJson(error);

    }

    } catch (Exception e) {

        System.err.println("❌ Error en processRequest: " +
e.getMessage());

        Map<String, String> error = Map.of("error", "Server error:
" + e.getMessage());

        return gson.toJson(error);

    }

}

/**
 * ✅ NUEVO: Retorna el estado actual del servidor
 */

private String getServerStatus() {

```

```

        Map<String, Object> status = new HashMap<>();

        status.put("activeThreads", Thread.activeCount());

        status.put("availablePermits",
connectionSemaphore.availablePermits());

        status.put("clientInfo", clientInfo);

        status.put("threadName", Thread.currentThread().getName());

        return gson.toJson(status);

    }

    /**
     * Envía un mensaje de error al cliente
     */

    private void sendError(BufferedWriter writer, String message)
throws IOException {

        Map<String, String> error = Map.of("error", message);

        String errorResponse = gson.toJson(error);

        writer.write(errorResponse + "\n");

        writer.flush();

    }

    /**
     * Registra errores durante el procesamiento
     */

    private void logError(Exception e) {

        System.err.println("✗ Error con cliente " + clientInfo + ": "
+ e.getMessage());

    }

    /**

```

```
    * Limpieza de recursos y liberación del semáforo
    */

    private void cleanup() {

        connectionSemaphore.release();

        System.out.println("🔒 Cliente desconectado: " + clientInfo +
            " [Permisos disponibles: " +
connectionSemaphore.availablePermits() + "]);

    }
}
```

