

# 4 in line

3/05/2022 - 07/06/2022

Sergi García Tocados

Javier Marín Guimerà

Enlace: [GitHub](#)

## Índice

<b>Índice</b>	<b>1</b>
Presentación del proyecto.	<b>2</b>
Descripción del proyecto.	2
Valoración global del proyecto.	2
Fase de análisis.	<b>3</b>
Análisis de requerimientos.	3
Estudio de viabilidad.	3
Análisis del target del proyecto.	4
Fase de diseño.	<b>4</b>
Diseño de la solución propuesta.	4
Cliente Android.	4
GameServer.	4
REST Server.	6
Fase de implementación.	<b>8</b>
Implementación del proyecto propuesto.	8
Cliente Android	8
MainActivity	9
UserLoginRegister Activity	10
Board Selection	12
Game	13
GameServer.	15
Funcionamiento hilo principal:	15
Funcionamiento hilo consola:	18
Funcionamiento hilo Game Match (lo más complejo del proyecto):	19
REST Server.	30
Fase de explotación de pruebas.	<b>32</b>
Pruebas unitarias.	32
Pruebas funcionales.	32
Pruebas con Usuarios:	32
Pruebas con Scores:	34
Pruebas de seguridad.	36
Contraseña encriptada	36

Token	36
EncryptedSharedPreferences	36
Configuración del entorno de ejecución de la solución.	37
Fase de mantenimiento y documentación del proyecto.	<b>37</b>
Manual de usuario de la aplicación.	37
Pruebas finales.	43

## 1. Presentación del proyecto.

La idea de este proyecto surgió de manera muy simple.

Queríamos hacer un proyecto basado en un juego y en Android y, de todos los que pensamos, llegamos a la conclusión que el más consistente y más lógico por el tiempo que teníamos era un 4 en raya.

### 1. Descripción del proyecto.

Es un 4 en raya de toda la vida con interfaz Android que se conecta a un servidor de juego que es el encargado de emparejar a los jugadores y calcular todo lo referente al juego.

Aparte de esto, tenemos un guardado en base de datos utilizando un Servicio REST creado con Spring Boot.

### 1. Valoración global del proyecto.

Consideramos que el proyecto es muy sólido para el poco tiempo que hemos tenido. Nos hubiese gustado añadir muchísimas cosas más, pero no nos ha dado tiempo.

En general y, de nuevo, con el tiempo que hemos tenido, estamos muy satisfechos con el resultado final.

## 2. Fase de análisis.

Nuestra fase de análisis la hemos tenido en varios momentos de la creación del proyecto.

### 1. Análisis de requerimientos.

Antes de empezar el proyecto, tuvimos que pensar cómo íbamos a hacer la conexión Cliente - GameServer y la conexión Cliente - REST Server.

Conforme hemos ido avanzando el proyecto, llegamos a la conclusión junto con Toni Moreno que nos hacía falta otra conexión GameServer - REST Server para guardar la información en base de datos y que fuese el GameServer el encargado de esto.

Aparte de lo anterior, también tuvimos que pensar cómo poder realizar desde el cliente Android las peticiones HTTP con las que crear su usuario, iniciar sesión, actualizarlo, etc.

Hablando del REST Server, el análisis de requerimientos fue sencillo. Las cosas estaban muy claras. Necesitábamos un apartado de usuarios encargado de todo lo necesario. También necesitábamos un apartado de tokens para poder validar la identificación de los usuarios a la hora de poder realizar cualquier tipo de operación. Por último, si o si necesitábamos algo con lo que poder guardar la información de las partidas, ahí se nos ocurrieron los scores.

## 2. Estudio de viabilidad.

Al ser un juego, la viabilidad es muy susceptible. Basta con que el juego se haga famoso, pille una época buena para su salida, etc.

En nuestro caso, creemos que la viabilidad es buena si contamos a pequeña escala. El juego de 4 en raya es famoso en todo el mundo, por lo que tendríamos público de todas las edades y de todos los lugares del mundo.

### 3. Análisis del target del proyecto.

Por suerte nuestro proyecto no tiene un target específico puesto que es un juego para todos los públicos con dispositivo Android.

Evidentemente la app no funciona en dispositivos Apple ni en ordenadores por falta de interfaz.

## 3. Fase de diseño.

Nuestra fase de diseño ha sido un tanto peculiar puesto que el tiempo no nos sobraba y decidimos ponernos manos a la obra directamente.

### 1. Diseño de la solución propuesta.

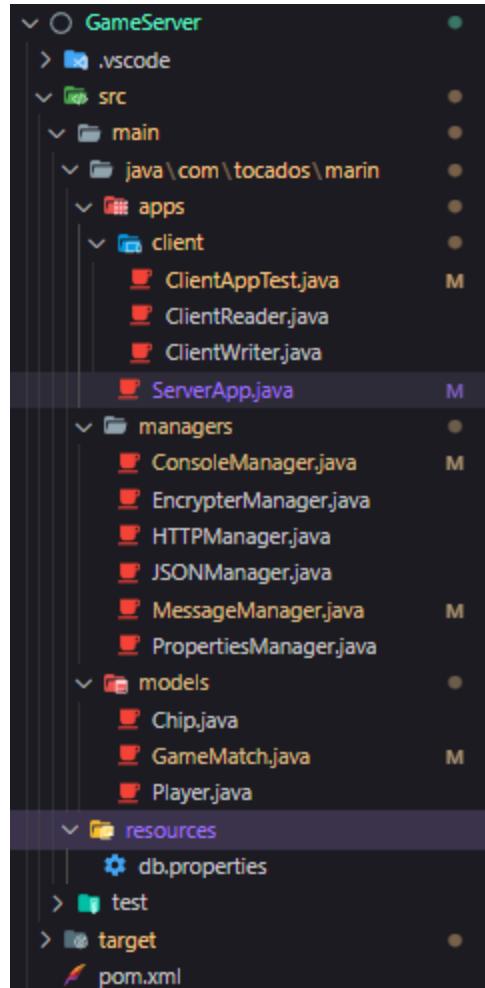
Hemos dividido el diseño en 3 partes:

- Cliente Android.
- GameServer.
- REST Server.

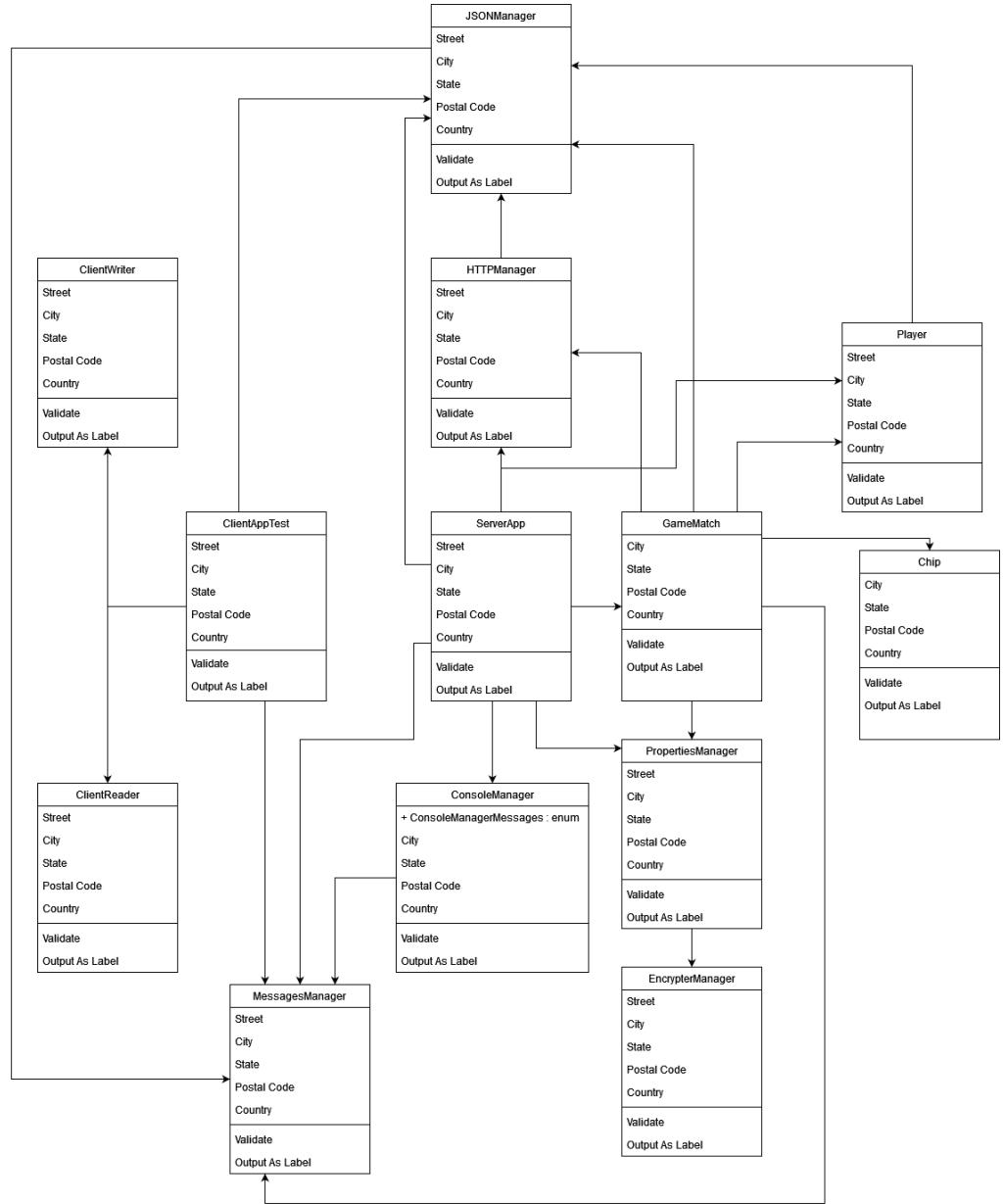
1. Cliente Android.
2. GameServer.

El diseño del GameServer ha sido creado sobre la marcha por la escasez de tiempo. No obstante, la estructura es bastante compleja pero tiene su orden.

Esta es la estructura de carpetas:



Este es su UML:



(Por escasez de tiempo, no hemos podido poner todas los atributos y métodos de cada clase)

### 3. REST Server.

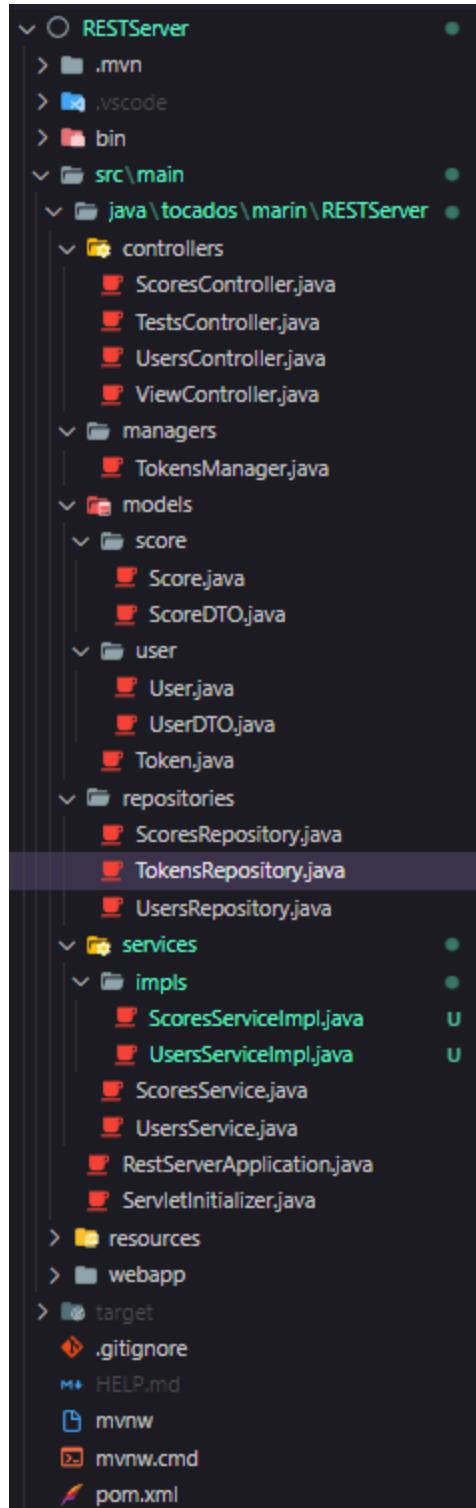
En cuanto al diseño del REST Server, hemos seguido los patrones típicos de los MVC.

Hemos utilizado la siguiente estructura de carpetas:

- Controladores.
- Servicios.
  - Implementaciones.
- Repositorios.
- Modelos.

- Managers

Esta es la estructura de carpetas:



Este es su UML:

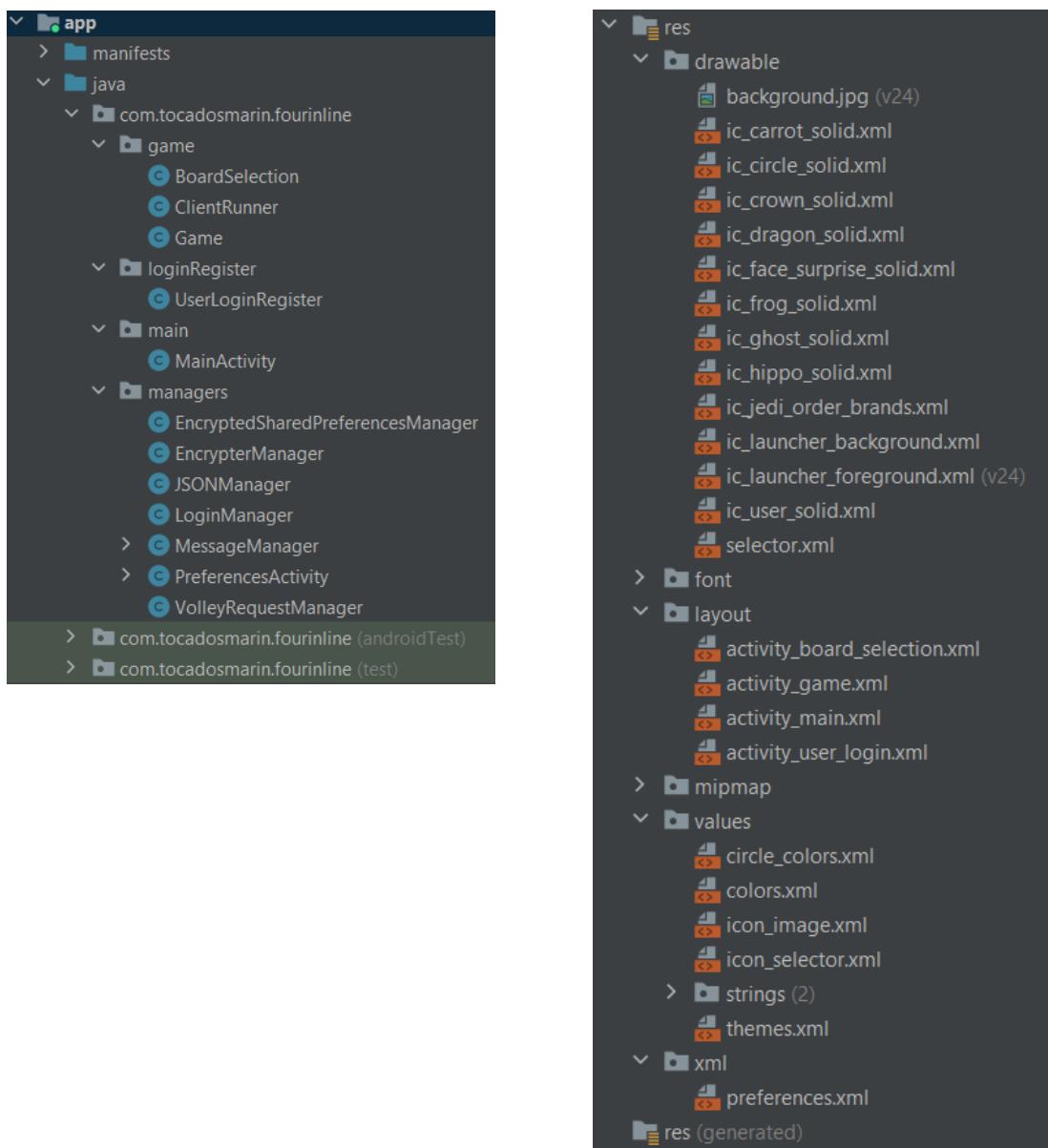
## 4. Fase de implementación.

### 1. Implementación del proyecto propuesto.

#### 1. Cliente Android

Tenemos la siguiente estructura en nuestra aplicación.

Disponemos de una actividad principal desde la cual accedemos al resto de actividades.



- **MainActivity**

En la actividad principal disponemos de cuatro botones, algunos de ellos realizan una función distinta dependiendo de si hemos iniciado sesión o no.

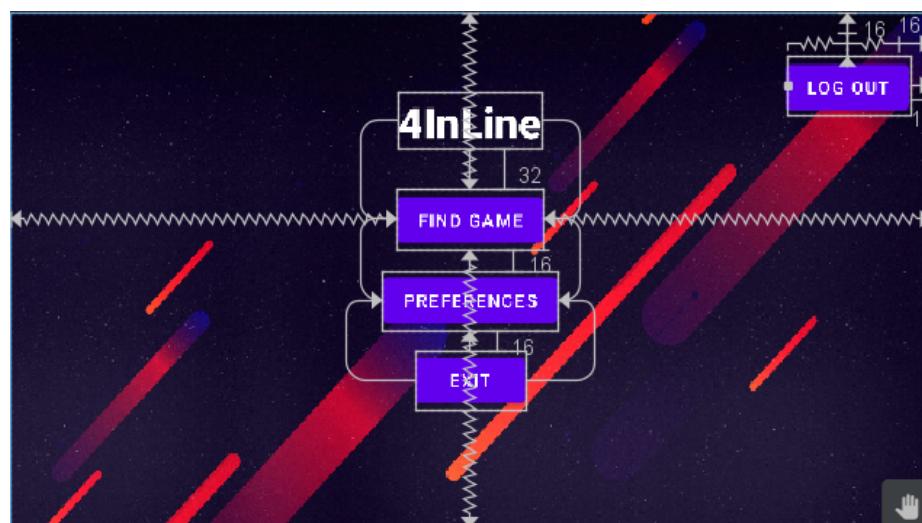
Al iniciar esta actividad, leerá las EncryptedSharedPreferences, en el caso de que encuentre datos de usuario comprobará si el token es válido, si no lo es pero el usuario marcó la opción de recordar contraseña, hará un inicio de sesión automático, en el caso de que no la marcase se eliminarán los datos del usuario.

**Find Game** → Si no tenemos la sesión iniciada, al pulsar este botón nos llevará a la actividad en la cual podremos iniciar sesión o registrarnos.

**Preferences** → Nos llevará a una nueva actividad en la que podremos seleccionar el ícono y color del jugador 1 y 2, también podremos activar o desactivar la música de fondo.

**Exit** → Cierra la aplicación.

**Log Out** → Solo si se detectan datos de usuario y el token no está caducado, se mostrará este botón. Al hacer click eliminará los datos de usuario y desaparecerá hasta que vuelva a encontrar datos de usuario.



- **UserLoginRegister Activity**

Si no tenemos sesión iniciada y queremos jugar una partida, se nos mostrará esta actividad. Podremos iniciar sesión si ya tenemos una cuenta o registrarnos en el caso de que no la tengamos.

#### **Login:**

El login, al igual que el register, realiza una petición HTTP a través de la librería Volley.

Hemos decidido utilizar esta librería ya que nos la recomendó un profesor y facilita el trabajo.

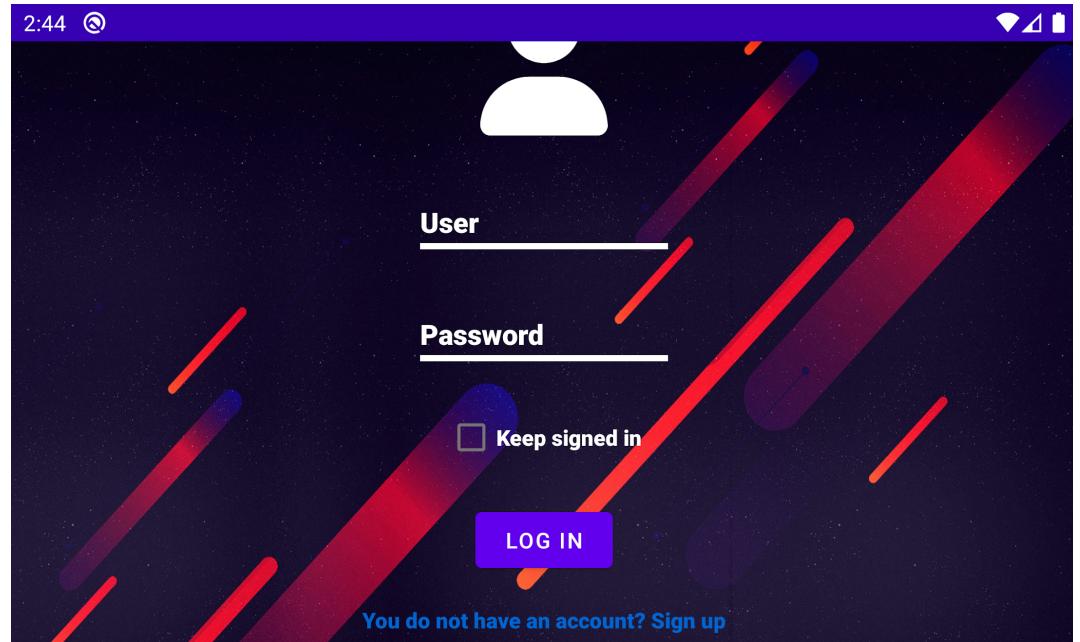
```
public static void makeRequest(String URL, Activity loginActivity, Context context, String user, String encryptedPwd, b  
String loginData = JSONManager.mountUsernameAndPasswordJson(user, encryptedPwd);  
JsonObjectRequest jsonObjectRequest = new JsonObjectRequest  
    (Request.Method.POST, URL, JSONManager.getJSONFromString(loginData), new Response.Listener<JSONObject>() {  
        @Override  
        public void onResponse(JSONObject response) {
```

Utilizamos un método en el que le pasamos los parámetros necesarios para hacer la petición.

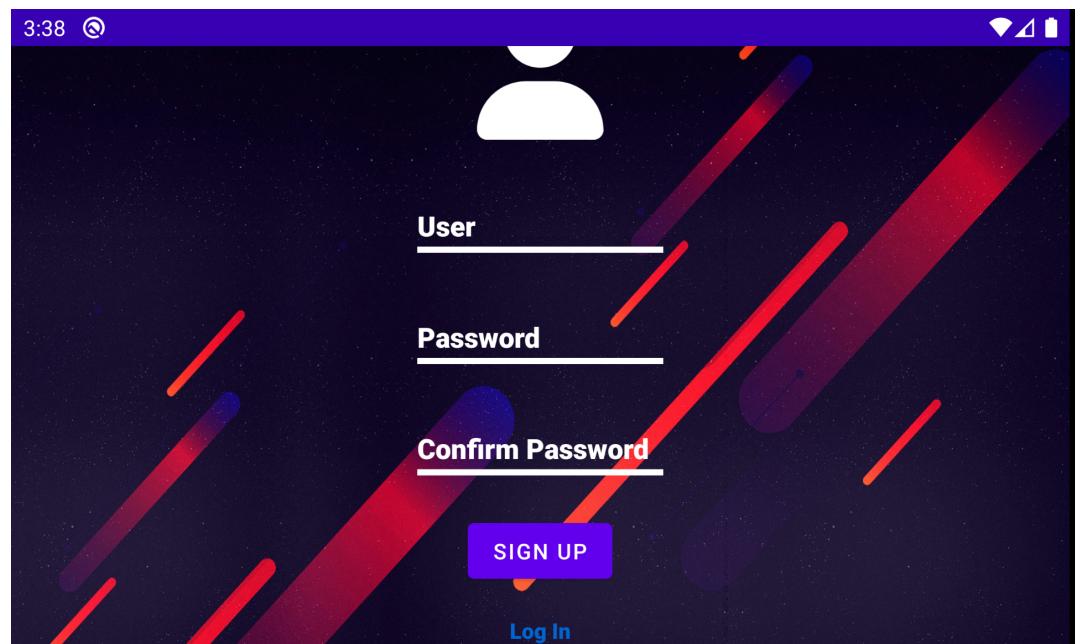
En el caso de hacer login, pasaremos usuario y contraseña, la cual se encriptará. Después se creará un JSON con estos dos datos y se mandarán en la petición.

La petición se generará en un nuevo hilo que estará esperando la respuesta, esta respuesta se procesará con los métodos sobreescritos **onResponse** y **onErrorResponse**.

Al entrar en esta actividad si es el caso de que nos queremos registrar, en la parte inferior encontraremos un texto clicable que nos llevará al registro.



### Register:



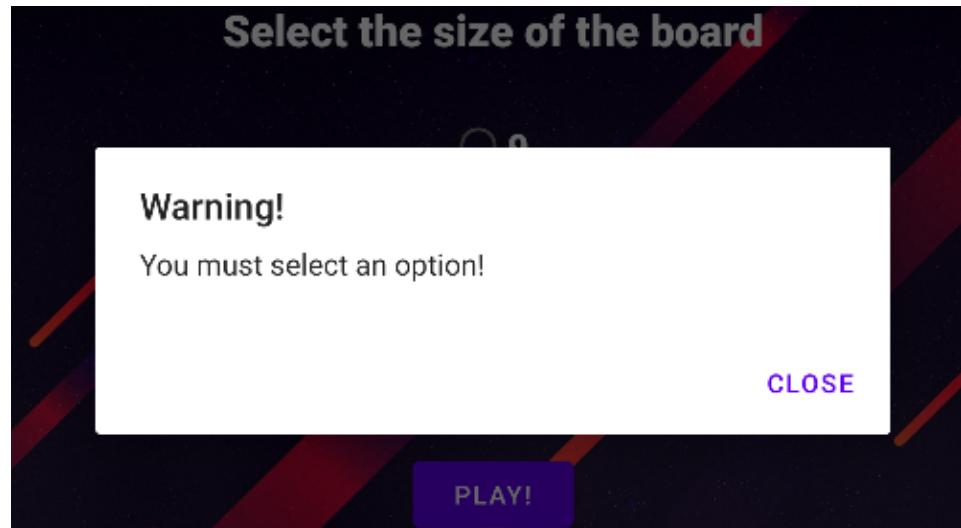
En este apartado introduciremos un nombre de usuario y contraseña, la aplicación comprobará que las dos contraseñas coincidan, si no se da el caso se mostrará un error informativo al usuario.

Si los datos son correctos se realizará una petición HTTP a través de la librería Volley.

- **Board Selection**

En el caso de que queramos jugar una partida y tengamos la sesión iniciada, se nos llevará a una pantalla en la que seleccionaremos el tamaño del tablero en el cual jugar, el tamaño variará en columnas, no en filas.

Si intentamos entrar en partida sin haber seleccionado una de las opciones se mostrará un mensaje de error indicativo.



Una vez seleccionemos una opción nos conectaremos al servidor de juego y entraremos en una lista de espera hasta que se encuentre otro jugador que haya seleccionado el mismo tamaño de tablero.

Una vez se encuentre un contrincante que haya seleccionado el mismo se lanzará la actividad Game.

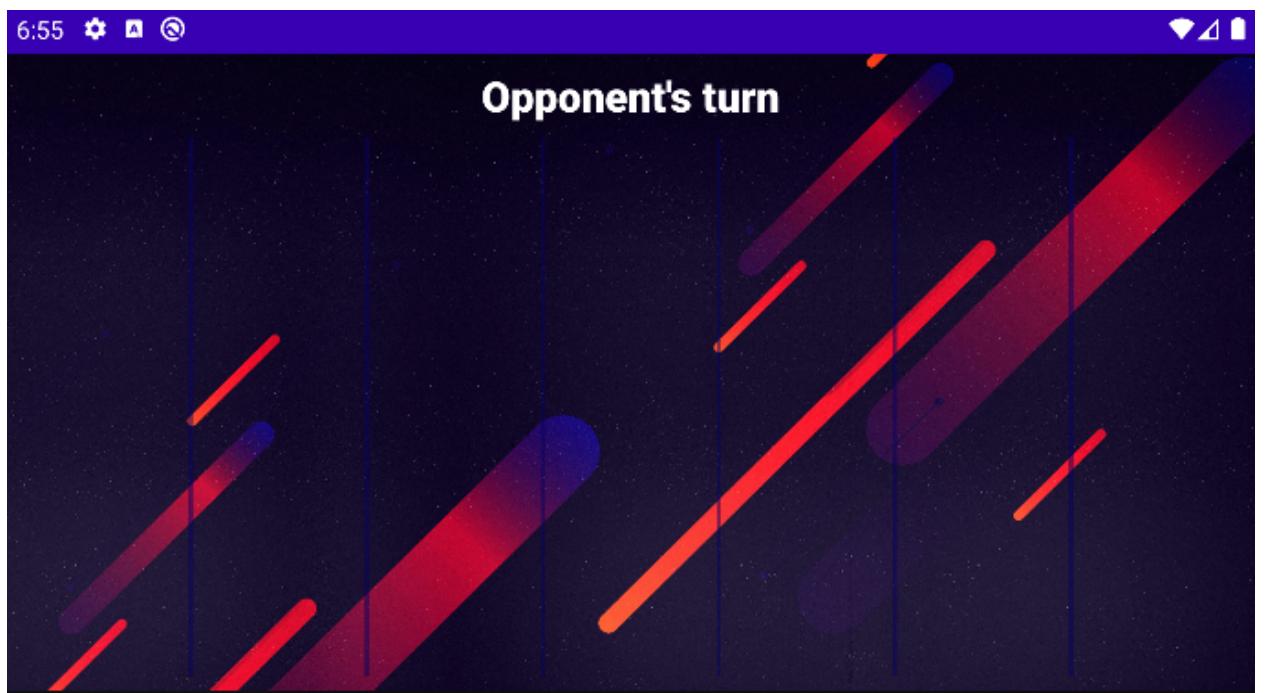
El servidor le asignará a cada uno una posición, el jugador que haya entrado primero a la cola de espera será el jugador 1 y el que haya entrado después será el jugador 2.

Esta información se enviará a través de un JSON a los clientes.

- **Game**

Una vez emparejados se lanzará esta actividad, que a la vez lanzará un hilo que será el encargado de enviar y recibir datos.

Al iniciar la actividad se crearán layouts automáticamente según el número de columnas seleccionado.



Por cada layout se creará también un listener para poder gestionar los eventos onClick.

También se recibirá un JSON en el cual se indica la posición de cada jugador.

Si el cliente recibe que es el jugador 1 podrá escribir en el tablero.

Una vez haga click en alguno de los layouts se enviará el número de

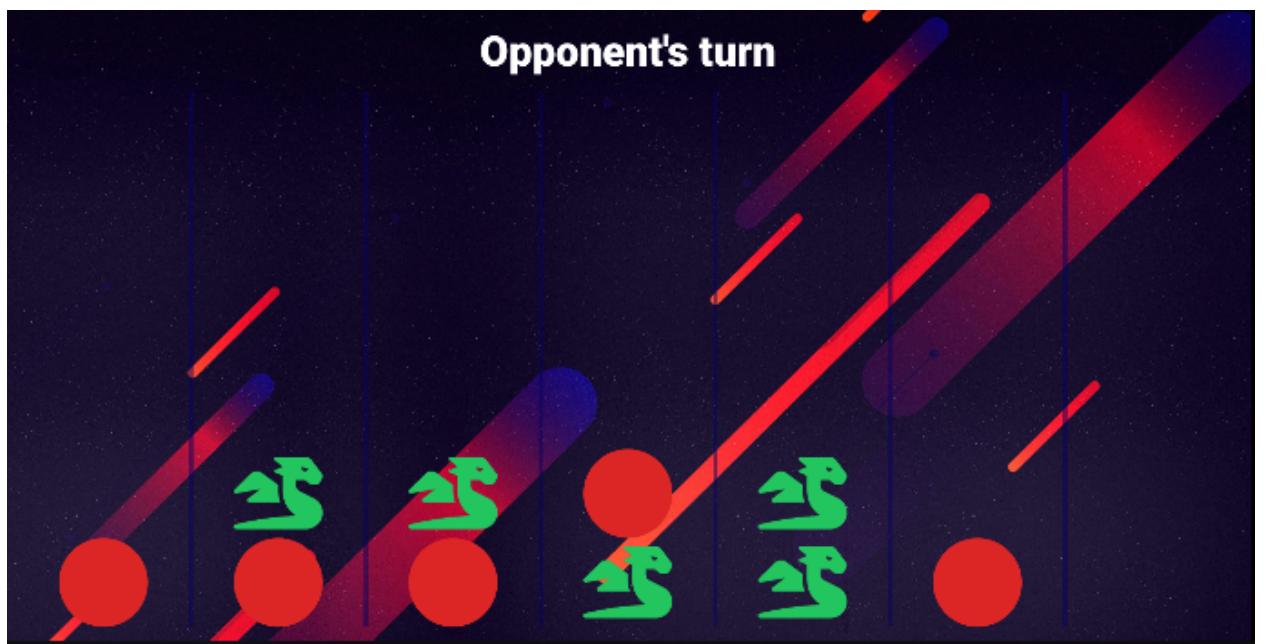
columnas a través de un JSON al servidor y el cliente pasará a estar en estado de espera hasta que reciba una respuesta.

En el caso de ser el jugador 2, iniciaremos la actividad igual pero entraremos en un estado de espera hasta que recibamos un dato del servidor.

El servidor mandará al jugador 2 los datos enviados por el jugador 1, cuando los reciba mostrará en pantalla una imagen en la columna que ha seleccionado y saldrá del modo espera.

En este momento el jugador 2 podrá hacer click en el tablero e introducir una ficha en la posición que quiera.

El funcionamiento del juego se basa en la sincronización de los Threads, cuando uno escribe el otro lee, hasta que alguno de los dos jugadores haga un 4 en raya y finalice la partida.



## 2. GameServer.

El GameServer es un servidor Java creado con una estructura Maven para hacer mucho más sencillo la inyección de dependencias.

**El funcionamiento del servidor consta de 3 hilos.**

**Funcionamiento hilo principal:**

1. Cuando el servidor se inicia, lo primero que se intenta hacer es un login en el servidor para saber si seguimos teniendo un token válido. Si el login falla, el servidor no llega a iniciarse y muestra un mensaje de error descriptivo.

```
public static void main(String[] args) throws Exception {
    if (!loginServer()) {
        MessageManager.showXMessage(Messages.LOGIN_FAILED);
        return;
    }
    LOGIN_FAILED(msg: "Login failed! Check server credentials on db.properties.");
}
```

2. Si el login ha sido correcto, el servidor inicia el hilo de la consola encargado de mostrar la información actual del servidor en el caso de que la queramos ver. También inicia el método por el cuál esperamos conexiones de clientes e inicia partidas.

```

public static void main(String[] args) throws Exception {
    if (!loginServer()) {
        MessageManager.showXMessage(Messages.LOGIN_FAILED);
        return;
    }

    /**
     * Here starts the game server:
     */

    startConsoleThread();
    startGameServerListener();
}

private static void startConsoleThread() {
    ConsoleManager consoleManager = new ConsoleManager();
    consoleManager.start();
}

private static void startGameServerListener() throws IOException, SocketException {
    ServerSocket serverSocket = new ServerSocket(PORT);
    serverSocket.setSoTimeout(TIMEOUT);

    MessageManager.showXMessage(Messages.WAITING_FOR_USERS);
    while (run) {

        // Wait the user connection with the server.
        try {
            Player player = new Player(serverSocket.accept());
            if (player.getCreated()) {
                checkListForMatches(player);
                MessageManager.showXMessage(Messages.USER_FOUND);
            }
        } catch (SocketTimeoutException e) {
            continue;
        }
    }

    if (serverSocket != null) {
        serverSocket.close();
    }
}

```

3. A la hora de esperar jugadores, tiene abierto un ServerSocket que está siempre escuchando. Cuando recibe una conexión, crea un nuevo jugador y con este comprueba si existe otro jugador con su mismo tablero esperando.

En el caso de que no exista, este último jugador entra a la lista de espera de su tablero en un mapa de Java.

```

private static void startGameServerListener() throws IOException, SocketException {
    ServerSocket serverSocket = new ServerSocket(PORT);
    serverSocket.setSoTimeout(TIMEOUT);

    MessageManager.showXMessage(Messages.WAITING_FOR_USERS);
    while (run) {

        // Wait the user connection with the server.
        try {
            Player player = new Player(serverSocket.accept());

            if (player.getCreated()) {
                checkListForMatches(player);
                MessageManager.showXMessage(Messages.USER_FOUND);
            }
        } catch (SocketTimeoutException e) {
            continue;
        }
    }

    if (serverSocket != null) {
        serverSocket.close();
    }
}

```

```

public static void checkListForMatches(Player newPlayer) {
    for (Map.Entry<Integer, Player> playerWaiting : playersWaiting.entrySet()) {
        if (playerWaiting != null) {
            checkCurrentPlayers(newPlayer, playerWaiting.getKey());
        }
    }
}

private static void checkCurrentPlayers(Player player, Integer columns) {
    if (player.getColumns() == columns) {
        if (playersWaiting.get(columns) != null) {
            startMatch(playersWaiting.get(columns), player, columns);

            // Remove the player waiting on
            playersWaiting.put(columns, value: null);
        } else {
            playersWaiting.put(columns, player);
        }
    }
}

```

4. Cuando 2 jugadores coinciden en tableros, este hilo construye una partida que a su vez es otro hilo para poder tener varias partidas simultáneas. Ambos jugadores son eliminados de la lista de espera.

```

private static void startMatch(Player player1, Player player2, Integer columns) {
    MessageManager.showXMessage(Messages.MATCH_FOUND);
    GameMatch gameMatch = new GameMatch(player1, player2, columns);
    gameMatch.start();
    currentGameMatches.add(gameMatch);
}

```

## Funcionamiento hilo consola:

1. Cuando se inicia el hilo de la consola, muestra un pequeño menú descriptivo por el que se pueden lanzar operaciones dentro del servidor.

Las operaciones disponibles actualmente las tenemos en un enum:

```
public enum ConsoleManagerMessages {
    END_SERVER(msg: "0: End server and matches."),
    SHOW_MATCHES(msg: "1: Show current matches."),
    SHOW_WAITING_PLAYERS(msg: "2: Show players count.");

    private String msg;

    ConsoleManagerMessages(String msg) {
        this.msg = msg;
    }

    /**
     * @return the msg
     */
    public String getMsg() {
        return msg;
    }

    /**
     * @param msg the msg to set
     */
    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

2. Todo esto funciona desde el método “run” de la clase ConsoleManager.

```

@Override
public void run() {
    String option;
    MessageManager.showXMessage(Messages.WELCOME_MESSAGE);

    while (ServerApp.run) {
        option = showMenuAndScan();

        makeTaskWithOption(option);
    }

    MessageManager.showXMessage(Messages.FAREWELL_MESSAGE);

    sc.close();
}

private String showMenuAndScan() {
    for (ConsoleManagerMessages consoleMessage : ConsoleManagerMessages.values()) {
        System.out.println(consoleMessage.getMsg());
    }
    return sc.nextLine();
}

private void makeTaskWithOption(String option) {
    switch (option) {
        case "0":
            ServerApp.run = false;
            break;

        case "1":
            ServerApp.showMatches();
            break;

        case "2":
            ServerApp.showPlayersCount();
            break;

        default:
            System.out.println("Wrong option");
            break;
    }

    System.out.println();
}

```

### **F**uncionamiento hilo GameMatch (lo más complejo del proyecto):

**E**l funcionamiento de este hilo es probablemente lo más complejo de todo el proyecto.

1. Para comenzar con esta parte, empezaremos por lo más importante, la creación de la partida.  
Básicamente tenemos los siguientes atributos y el siguiente constructor:

```

public class GameMatch extends Thread {
    private static int TIMEOUT = 10000;
    private static int MAX_ROWS = 6;

    private Player player1;
    private Player player2;
    private Stack<Stack<Integer>> board;
    private Integer rounds = 0;
    private Boolean matchEnded = false;

    /**
     * Constructor of the class.
     *
     * @param player1 Player 1 playing.
     * @param player2 Player 2 playing.
     * @param columns Columns of the board.
     */
    public GameMatch(Player player1, Player player2, Integer columns) {
        this.player1 = player1;
        this.player2 = player2;
        this.board = new Stack<>();

        for (int i = 0; i < columns; i++) {
            this.board.add(new Stack<Integer>());
        }
    }
}

```

Esto nos genera la partida con los 2 jugadores, genera un tablero que hemos hecho en formato Stack de Stacks que contienen Integer que serán las fichas de los jugadores (1 para el jugador 1 y 2 para el jugador 2).

2. Cuando la partida está creada, el servidor se encarga de iniciarla.
3. Nada más se inicia la partida, se preparan los sockets de los usuarios, los BufferedReader y los PrintStream para el recibo y envío de datos al jugador correspondiente y primeramente se les envía un mensaje conforme ya tienen

ponente y pueden pasar al juego.

```
@Override
public void run() {
    try {
        this.player1.getPlayerSocket().setSoTimeout(TIMEOUT);
        this.player2.getPlayerSocket().setSoTimeout(TIMEOUT);
    } catch (SocketException e) {
        e.printStackTrace();
    }

    // Player 1 streams;
    BufferedReader player1Reader = this.player1.getReader();
    PrintStream player1Writer = this.player1.getWriter();

    // Player 2 streams;
    BufferedReader player2Reader = this.player2.getReader();
    PrintStream player2Writer = this.player2.getWriter();

    // Send to the users that they have a match and what is their position.
    player1.setPosition(position: 1);
    player2.setPosition(position: 2);
    player1Writer.println(JSONManager.mountHasOpponentJson(position: 1));
    player2Writer.println(JSONManager.mountHasOpponentJson(position: 2));
}
```

4. Toda la partida sucede en estas líneas de código:

```
Integer column = null;
while (!this.matchEnded) {
    this.rounds++;

    /**
     * Part 1: Player 1 sends, player 2 gets.
     */

    /**
     * This method checks everything on the current loop and returns something different than null if the server is closed unexpectedly.
     */
    if ((column = doALoopAndCheckEverything(column, this.player1, player1Reader, player1Writer, this.player2,
        player2Writer)) != null) {
        break;
    }

    if (this.matchEnded) {
        break;
    }

    /**
     * Part 2: Player 2 sends, player 1 gets.
     */

    /**
     * This method checks everything on the current loop and returns something different than null if the server is closed unexpectedly.
     */
    if ((column = doALoopAndCheckEverything(column, this.player2, player2Reader, player2Writer, this.player1,
        player1Writer)) != null) {
        break;
    }
}
```

5. Este método es el que se encarga en general de mirar toda la semi ronda actual, es decir, desde que el jugador x envía,

el servidor recibe, calcula y envía al jugador y.

```
/*
 * This method checks everything from the player that is sending until the
 * player that is getting the info.
 *
 * @param column          Column in null because we need to check if we
 *                        receive anything from the current player playing.
 * @param playerPlaying    Player playing.
 * @param playerPlayingReader Player playing reader.
 * @param playerPlayingWriter Player playing writer for the opponent response.
 * @param playerWaiting     Player waiting.
 * @param playerWaitingWriter Player waiting writer to send the response
 *                            whatever it is.
 *
 * @return -1 if the server is closed.
 */
private Integer doALoopAndCheckEverything(Integer column, Player playerPlaying, BufferedReader playerPlayingReader,
                                         PrintStream playerPlayingWriter, Player playerWaiting, PrintStream playerWaitingWriter) {
    if ((column = getUserColumn(playerPlayingReader, column)) == null) {
        return -1;
    }

    if (checkMatchStatus(column, this.board.get(column).size(), playerPlaying, playerWaiting)) {
        playerPlayingWriter.println(JSONManager.mountColumnAndResultAndScoreJson(column,
            Messages.WINNER, playerPlaying.getScore()));
        playerWaitingWriter.println(JSONManager.mountColumnAndResultAndScoreJson(column,
            Messages.LOSER, playerWaiting.getScore()));
    } else {
        playerWaitingWriter.println(JSONManager.mountColumnJson(column));
        column = null;
    }

    return column;
}
```

6. Dentro del anterior método leemos del usuario x y comprobamos si ha hecho 4 en raya. En el caso de haberlo hecho, enviamos la respuesta a ambos clientes.

7. El cálculo del 4 en raya se hace en el siguiente método:

```
/*
 * Check the match status with the new chip.
 *
 * @param column      The new chip column.
 * @param playerPlaying The player that has entered the chip.
 * @param playerToNotify The player that is waiting the response.
 * @return true if the current player won the match; false if not won.
 */
private Boolean checkMatchStatus(Integer column, Integer row, Player playerPlaying, Player playerToNotify) {
    if (this.board.get(column).size() < MAX_ROWS) {
        // Here we add the new chip to the GameMatch board
        this.board.get(column).add(playerPlaying.getPosition());
        this.currentChipsCount++;
        playerPlaying.setScore(playerPlaying.getScore() + 5);

        Chip newestChip = new Chip(column, row, this.board.get(column).get(row));
        List<Chip> arroundChips = getArroundChipsList(column, this.board.get(column).size() - 1);
        Integer wins = 0;

        if (arroundChips != null) {
            for (Chip arroundChip : arroundChips) {
                if (checkFor4InLine(arroundChips, arroundChip, newestChip)) {
                    wins++;
                }
            }
        }

        if (wins != 0) {
            if (wins == 1) {
                // Single victory adds 20 points to the player.
                playerPlaying.setScore(playerPlaying.getScore() + 20);
            } else if (wins > 1) {
                /**
                 * Multiple victories add 20 points per win + other 10 points per win because is
                 * more difficult to hit more than 1 win at the same time.
                 */
                playerPlaying.setScore(playerPlaying.getScore() + 20 * wins + (10 * wins));
            }
        }

        playerPlaying.setIsWinner(isWinner: true);
        playerToNotify.setIsWinner(isWinner: false);
        this.matchEnded = true;
    }

    return true;
}

if (isBoardFull()) {
    this.matchEnded = true;
    return true;
}

return false;
}
```

8. Este anterior método primero comprueba si la columna en la que se supone que ha introducido el usuario tiene hueco. No debe darse el caso de esto porque está controlado también en front, pero no está de más.

```
if (this.board.get(column).size() < MAX_ROWS) {
```

9. Cuando el anterior condicional da true, pasa a la comprobación del 4 en raya comenzando por coger todas las fichas del mismo jugador que tiene alrededor de la última que ha introducido. Esto se hace en el siguiente método:

```
List<Chip> arroundChips = getArroundChipsList(column, this.board.get(column).size() - 1);
```

```

/**
 * Mounts a map with the chips around the last one entered by the user
 */
private List<Chip> getAroundChipsList(Integer column, Integer row) {
    List<Chip> aroundChips = new ArrayList<>();

    /**
     * If the column is the first one, the start value will be 0 and we will need to
     * check ONLY the next column if exist.
     */
    Integer x = Math.max(column - 1, 0), xMax = Math.min(x + (x == 0 ? 2 : 3), this.board.size());
    // If the row is the first one, the start value will be 0.
    Integer y = Math.max(this.board.get(column).size() - 2, 0), yMax;

    for (int i = x; i < xMax; i++) {
        /**
         * For each column, we need to check if has at least one value to check.
         * Example (the X is the newest user chip):
         *
         * 1 2 3
         * - X -
         * - o x
         * x o o x
         *
         * On the example, the first column will not be read, the second will only read
         * the bottom "o" and on the third will read only the "x".
         */
        if (this.board.get(i).size() - 1 >= y) {
            yMax = Math.min(y + (y == 0 ? 2 : 3), this.board.get(i).size());
            for (int j = y; j < yMax; j++) {
                /**
                 * If we are looking on the newest user's chip, we will be avoid it.
                 *
                 * Else if: Will check if the newest user's chip is the same chip that we are
                 * checking right now.
                 */
                if ((i == column && j == this.board.get(column).size() - 1)) {
                    continue;
                } else if (this.board.get(column).get(row) == this.board.get(i).get(j)) {
                    aroundChips.add(new Chip(i, j, this.board.get(i).get(j)));
                }
            }
        }
    }
    return aroundChips;
}

```

Resumiendo este método, busca en un 3x3 alrededor de la última ficha del usuario y guarda las fichas en una lista de Chip y por cada uno guarda su posición de x, posición de y y su valor actual en el tablero (un 1 o un 2).

```

public class Chip {
    private Integer x;
    private Integer y;
    private Integer value;

    public Chip(Integer x, Integer y, Integer value) {
        this.x = x;
        this.y = y;
        this.value = value;
    }
}

```

10. Si ha encontrado alguna ficha, por cada una que haya encontrado buscará si esa hace 4 en raya.

```

Integer wins = 0;

if (arroundChips != null) {
    for (Chip arroundChip : arroundChips) {
        if (checkFor4InLine(arroundChips, arroundChip, newestChip)) {
            wins++;
        }
    }
}

```

11. Dentro de este método comprueba de la siguiente manera. Primero comienza buscando el posible límite del

4 en raya en la dirección de donde se encuentra la siguiente ficha. Esto lo hace el método **hasFound4inLineEdge()** que tiene esta estructura:

```
/*
 * This will find the edge of the 4 in line.
 *
 * @param x           The next coord x.
 * @param y           The next coord y.
 * @param arroundChip The chip that we started from.
 * @return True if we found board border, null object or a 2 in case we are the
 *         player 1.
 */
private boolean hasFound4inLineEdge(Integer x, Integer y, Chip arroundChip) {
    // This will check if we are out of the board.
    if (x < 0 || x == this.board.size() || y < 0 || y == this.board.get(x).size()) {
        return true;
    }

    // This will check if we are on a null object.
    if (this.board.get(x).size() <= y) {
        return true;
    }

    // This will check if we are on a different chip than the newest chip.
    if (this.board.get(x).get(y) != arroundChip.getValue()) {
        return true;
    }

    /**
     * If none of the conditionales got true, that means that we have reached
     * another equal chip.
     */
    return false;
}
```

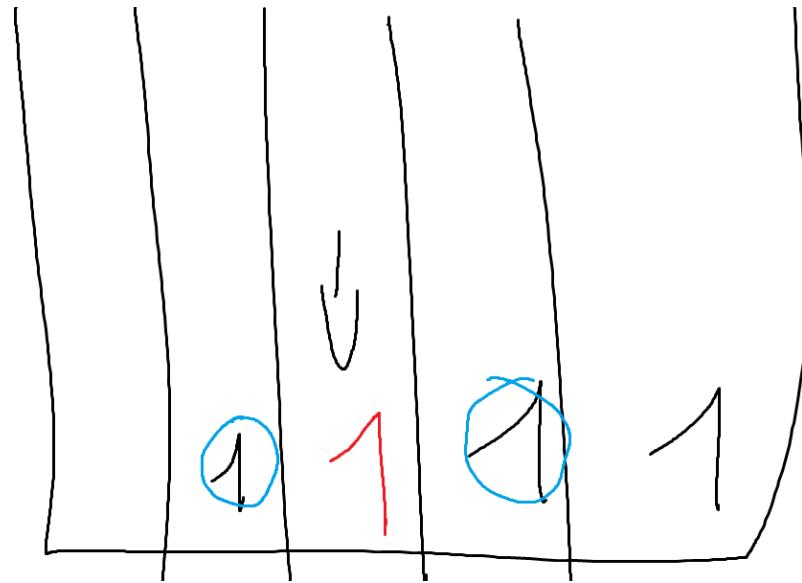
Cuando encuentra una ficha distinta, borde de tablero o una posición donde no hay ficha, invierte la dirección y empieza a contar 4 posiciones.

Por cada posición mira que la ficha que estamos viendo es del mismo jugador, que la ficha no es nula y que no hemos salido del tablero.

Una de las cosas que tenemos que tener en cuenta es que, si encontramos una de las fichas que el método **getArroundChips()** había encontrado y que no es la ficha por la que hemos empezado en este método, tenemos que eliminarla de la lista para que no vuelva a comprobar el 4 en raya por esa ficha. Es decir y resumiendo esto último, poniendo el siguiente caso y si no contáramos esto, podría darse un doble 4 en raya con solo 1 en pantalla:

**(Círculos azules indican las fichas que se han encontrado alrededor de la roja que es la última que ha introducido el usuario)**

Espero que ninguno sea daltónico jajaja :)



Si el bucle “for” consigue llegar al final y no ha saltado ningún “return false”, entonces habremos encontrado un 4 en raya y devolveremos un true.

Foto del código del método:

```

/**
 * Check if we have a 4 in line.
 *
 * @param arroundChips
 *
 * @param map           Integer array with 2 values of one of the closest
 *                      chips.
 *                      First value = x, second value = y.
 * @param chipColumn   The last entered chip column to calculate the
 *                      direction of
 *                      the 4 in line.
 * @param chipRow      The last entered chip row to calculate the direction
 *                      of the
 *                      4 in line.
 * @return True if 4 in line is encountered, false if not.
 */
private Boolean checkFor4InLine(List<Chip> arroundChips, Chip arroundChip,
                                Chip newestChip) {

    Chip currentChip;
    // Get the coords of the arrounded chip.
    Integer x = arroundChip.getX(), y = arroundChip.getY();

    // Get the direction of the possible 4 in line.
    Integer xDirection = x - newestChip.getX(),
            yDirection = y - newestChip.getY();

    /**
     * This checks if the next chip is equals the newest chip.
     * If we find a different chip, the while will end because we have reached the
     * edge of the possible 4 in line.
     */
    while (!hasFound4inLineEdge(x + xDirection, y + yDirection, arroundChip)) {
        x += xDirection;
        y += yDirection;
    }

    /**
     * Because we have reached the edge of the possible 4 in line, we need to read
     * on the opposite way.
     */
    xDirection = -xDirection;
    yDirection = -yDirection;

    for (int i = 0; i < 4; i++) {
        // Check if we are out of the board or looking on empty tile.
        if (x < 0 || x == this.board.size() || y < 0 || y == this.board.get(x).size()
            || y > this.board.get(x).size()) {
            return false;
        }
    }
}

```

```

        currentChip = new Chip(x, y, this.board.get(x).get(y));

        /**
         * If we are playing as the player 1, if we find a "2" on the board means that
         * we didnt make a 4 in line on this direction.
         */
        if (currentChip.getValue() != arroundChip.getValue()) {
            return false;
        }

        /**
         * Remove the current chip of the arroundChips if is not the first that we
         * started from.
         */
        if (!arroundChip.equals(currentChip) && arroundChips.contains(currentChip)) {
            arroundChips.remove(currentChip);
        }

        x += xDirection;
        y += yDirection;
    }

    return true;
}

```

12. Cuando el **checkFor4InLine()** nos devuelve un true, le añadimos una victoria al jugador que está jugando actualmente.
13. Cuando comprobamos todas las fichas que tenía alrededor la última ficha que ha introducido el usuario, salimos del bucle. Al salir miramos si el jugador ha ganado alguna vez. En el caso de haberlo hecho, miramos si ha sido más de una vez con la misma ficha. De ser así, la puntuación que se le sumará un total de, en lugar de 20 puntos por victoria, 30 puntos por vistoria.  
Aparte de eso, se le pondrá a cada jugador su correspondiente Boolean indicando si ha ganado o ha perdido, a la partida se le pondrá que ha finalizado y devolveremos un true conforme se ha encontrado un 4 en raya para poder notificar a los jugadores.

```

if (wins != 0) {
    if (wins == 1) {
        // Single victory adds 20 points to the player.
        playerPlaying.setScore(playerPlaying.getScore() + 20);
    } else if (wins > 1) {
        /**
         * Multiple victories add 20 points per win + other 10 points per win because is
         * more difficult to hit more than 1 win at the same time.
         */
        playerPlaying.setScore(playerPlaying.getScore() + 20 * wins + (10 * wins));
    }
}

playerPlaying.setIsWinner(isWinner: true);
playerToNotify.setIsWinner(isWinner: false);
this.matchEnded = true;

return true;
}

```

14. En el caso de no haber ganado, hay que comprobar si el tablero está lleno. Si el tablero está lleno, hay que finalizar la partida y se les notificará a los jugadores del empate.

```
if (isBoardFull()) {
    this.matchEnded = true;
    return true;
}
```

15. Poniendo el caso de que se ha encontrado un 4 en raya o el tablero está lleno, la partida entra en este condicional y envía los resultados a cada jugador:

```
if (checkMatchStatus(column, this.board.get(column).size(), playerPlaying, playerWaiting)) {
    if (this.player1.getIsWinner() != null &amp; this.player2.getIsWinner() != null) {
        playerPlayingWriter.println(JSONManager.mountColumnAndResultAndScoreJson(column,
            Messages.WIN, playerPlaying.getScore()));
        playerWaitingWriter.println(JSONManager.mountColumnAndResultAndScoreJson(column,
            Messages.LOSE, playerWaiting.getScore()));
    } else {
        playerPlayingWriter.println(JSONManager.mountColumnAndResultAndScoreJson(column,
            Messages.DRAW, playerPlaying.getScore()));
        playerWaitingWriter.println(JSONManager.mountColumnAndResultAndScoreJson(column,
            Messages.DRAW, playerWaiting.getScore()));
    }
} else {
    playerWaitingWriter.println(JSONManager.mountColumnJson(column));
    column = null;
}
```

16. Después no se cumplirá el siguiente condicional puesto que el server no se ha cerrado de manera inesperada y entrará en el condicional que comprueba si la partida ha finalizado cosa que hará que se rompa el bucle y la partida habrá terminado definitivamente.

```
/** 
 * Part 1: Player 1 sends, player 2 gets.
 */
/** 
 * This method checks everything on the current loop and returns something
 * different than null if the server is closed unexpectedly.
 */
if ((column = doALoopAndCheckEverything(column, this.player1, player1Reader, player1Writer, this.player2,
    player2Writer)) != null) {
    break;
}

if (this.matchEnded) {
    break;
}
```

17. Al salir del while, todavía tenemos 2 pequeñas tareas pendientes. La primera de todas y más importante es guardar los datos de los usuarios en la base de datos. En el condicional tenemos que comprobar si el servidor se ha cerrado. En el caso de haberse cerrado, hay que notificar a los jugadores para que salgan de la partida. Si el servidor no ha finalizado, significa que solo lo ha hecho la partida. En este caso guardamos sus datos en la base de

datos.

```
if (!ServerApp.run) {
    closePlayers(player1Writer, player2Writer);
} else {
    savePlayersResults(player1, player2);
}
```

18. Estos son los 2 métodos encargados de esto:

```
private void savePlayersResults(Player player1, Player player2) {
    savePlayerInfo(player1);
    savePlayerInfo(player2);
}

private void savePlayerInfo(Player player) {
    Map<String, Object> serverInfo = new HashMap<>();
    serverInfo.put(key: "username", PropertiesManager.getPropertyByName(PropertiesStrings.SERVER_NAME));
    serverInfo.put(key: "token", PropertiesManager.getPropertyByName(PropertiesStrings.SERVER_TOKEN));

    Map<String, Object> playerInfo = new HashMap<>();
    playerInfo.put(key: "username", player.getUsername());
    playerInfo.put(key: "score", player.getScore());
    playerInfo.put(key: "isWinner", player.getIsWinner());

    Map<String, Object> resultsMap = new HashMap<>();
    resultsMap.put(key: "server", serverInfo);
    resultsMap.put(key: "player", playerInfo);

    MessageManager.showXMessage(Messages.REST_SERVER_RESPONSE);
    System.out.println(HTTPManager.makeRequest(Paths.SCORES_INSERT_ONE, resultsMap));
}

private void closePlayers(PrintStream player1Writer, PrintStream player2Writer) {
    player1Writer.println(JSONManager.mountServerCloseJson());
    player2Writer.println(JSONManager.mountServerCloseJson());
}
```

### 3. REST Server.

El REST Server es un web service que se dedica a escuchar peticiones HTTP que le lleguen y responderlas con los datos solicitados o una respuesta a una operación realizada.

El funcionamiento principal es el siguiente:

1. El servidor REST se inicia.

```
@SpringBootApplication
public class RestServerApplication {

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(primarySource: RestServerApplication.class, args);
    }
}
```

- Desde el momento en que se inicia, se queda listo para recibir peticiones HTTP y responderlas.

```

2022-06-06 20:41:49.794 [main] i.m.RESTServer.RestServerApplication : Starting RestServerApplication using Java 11.0.13 on DESKTOP-MASTER with PID 11436 (C:\Users\Javier\OneDrive\Proyectos\Final\43InLineRESTServer\target\classes)
2022-06-06 20:41:49.796 [main] i.m.RESTServer.RestServerApplication : No active profile set, falling back to 1 default profile: 'default'
2022-06-06 20:41:49.539 [INFO ] i.m.R.S.RestServerApplication : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2022-06-06 20:41:49.540 [INFO ] i.m.R.S.RestServerApplication : Finished Spring Data JPA repository scanning in 35 ms. Found 3 JPA repository interfaces.
2022-06-06 20:41:49.582 [INFO ] i.m.R.S.RestServerApplication : Tomcat initialized with port(s): 8081 (http)
2022-06-06 20:41:50.715 [INFO ] i.m.R.S.RestServerApplication : Starting service [Tomcat]
2022-06-06 20:41:50.716 [INFO ] i.m.R.S.RestServerApplication : Tomcat started on port(s): [http://0.0.0.0:8081/] in [Apache Tomcat/9.0.42]
2022-06-06 20:41:50.966 [INFO ] o.a.apache.jasper.servlet.TldScanner : At least one TLD was scanned for TLDs. Enable debug logging for this logger for a complete list of TLDs that were scanned but no TLDs were found in them. Skipping unneeded TLDs during scanning can improve startup time and JSP compilation time.
2022-06-06 20:41:50.967 [INFO ] o.a.apache.jasper.servlet.TldScanner : No TLDs were found in them. Skipping unneeded TLDs during scanning can improve startup time and JSP compilation time.
2022-06-06 20:41:50.971 [INFO ] i.m.R.S.RestServerApplication : Root WebApplicationContext: initialization completed in 1624 ms
2022-06-06 20:41:51.140 [INFO ] o.a.n.i.InternalWebServlet:logOpen : [Root] Processing PersistenceContextOpenEvent [name: default]
2022-06-06 20:41:51.141 [INFO ] o.a.n.i.InternalWebServlet:logClose : [Root] Processing PersistenceContextCloseEvent [name: default]
2022-06-06 20:41:51.331 [INFO ] o.a.n.hibernate.annotations.common.Version : [ICARO00001]: Hibernate Commons Annotations (5.1.3.Final)
2022-06-06 20:41:51.424 [INFO ] o.a.n.i.InternalWebServlet:logOpen : [Root] Processing PersistenceContextOpenEvent [name: default]
2022-06-06 20:41:51.425 [INFO ] o.a.n.i.InternalWebServlet:logClose : [Root] Processing PersistenceContextCloseEvent [name: default]
2022-06-06 20:41:51.435 [INFO ] i.m.R.S.RestServerApplication : [ICARO00001]: Hibernate Commons Annotations (5.1.3.Final)
2022-06-06 20:41:51.436 [INFO ] o.a.n.i.InternalWebServlet:logOpen : [Root] Processing PersistenceContextOpenEvent [name: default]
2022-06-06 20:41:51.437 [INFO ] o.a.n.i.InternalWebServlet:logClose : [Root] Processing PersistenceContextCloseEvent [name: default]
2022-06-06 20:41:51.477 [INFO ] i.m.R.S.RestServerApplication : [ICARO00001]: Using dialect: org.hibernate.dialect.MySQL50Dialect
2022-06-06 20:41:51.478 [INFO ] i.m.R.S.RestServerApplication : [ICARO00001]: Initializing PMW EntityManagerFactory for persistence unit 'default'
2022-06-06 20:41:51.479 [INFO ] i.m.R.S.RestServerApplication : [ICARO00001]: Initializing PMW EntityManagerFactory for persistence unit 'default'
spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.
2022-06-06 20:41:52.989 [INFO ] o.a.b.w.s.WelcomePageHandlerMapping : Adding welcome page: class path resource [public/index.html]
2022-06-06 20:41:53.055 [INFO ] i.m.R.S.RestServerApplication : Tomcat started on port(s): 8081 (http) with context path ''
2022-06-06 20:41:53.056 [INFO ] i.m.R.S.RestServerApplication : If you are seeing this message, your application is not yet mapped
2022-06-06 20:42:03.501 [INFO ] i.m.R.S.RestServerApplication : [nio-8081-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]
2022-06-06 20:42:03.501 [INFO ] i.m.R.S.RestServerApplication : [nio-8081-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2022-06-06 20:42:03.501 [INFO ] i.m.R.S.RestServerApplication : [nio-8081-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : DispatcherServlet 'dispatcherServlet' initialized in 0 ms

```

- Cuando le llega una petición HTTP, mira que controlador es el que tiene que recibir esa petición dependiendo de qué URL ha utilizado el usuario, servidor de juego, etc.

```

@RestController
@RequestMapping("/users")
public class UsersController {

    @RestController
    @RequestMapping("/scores")
    public class ScoresController {

```

- Una vez llega la petición al controlador, este se encarga de llamar al servicio correspondiente para resolver la petición.
- Este servicio a su vez llama al repositorio pidiendo lo que necesita para resolver la petición.
- El repositorio coge el modelo de objeto como referencia para utilizar la base de datos vía JPA.
- El servicio trama todos los datos, hace las operaciones necesarias declaradas en el método que se haya tenido que llamar.
- Cuando este ya ha terminado todo el proceso, le devuelve al controlador la información de respuesta en formato JSON y es el controlador el que devuelve la información en la response del paquete HTTP.

## 5. Fase de explotación de pruebas.

### 1. Pruebas unitarias.

Al no tener demasiado tiempo de sobra, hemos tenido que saltarnos las pruebas unitarias e ir directamente a las pruebas funcionales.

En cuanto a ambos servidores, el GameServer y el REST Server, tuvimos que realizar mucho código deduciendo que iba a funcionar y luego perfeccionarlo y corregir los errores sobre la marcha.

### 2. Pruebas funcionales.

Hablando del REST Server, todas las pruebas funcionales las realizamos con PostMan puesto que todo el servidor funciona con peticiones HTTP con distintos métodos.

Todas las pruebas que hemos realizado son las siguientes y con sus resultados:

#### Pruebas con Usuarios:

- Get all users:

The screenshot shows the Postman application interface. At the top, there are several tabs: 'GET Get all users' (selected), 'POST Create one user', 'POST Login', 'PUT Update', and 'DELETE Delete user'. Below these tabs, the URL 'http://localhost:8080/users' is displayed. The main area shows a 'GET' request with the body content '1'. The 'Body' tab is selected, showing the raw JSON response:

```
1 [{"id": 1, "username": "BxJpE7TsQFGauhdVtY2JvmZTzdIv", "scores": []}, {"id": 3, "username": "Javier", "scores": [25]}, {"id": 4, "username": "Sergi", "scores": [0]}]
```

The status bar at the bottom right indicates: Status: 200 OK Time: 422 ms Size: 326 B Save Response.

- Create one user:

The screenshot shows the Postman application interface. At the top, there are several tabs: **GET Get all users**, **POST Create one user**, **POST Login**, **PUT Update**, and **DELETE Delete user**. To the right of these tabs, there are buttons for **Save**, **...**, and **Send**. The status bar indicates "No Environment".

The main workspace shows a **POST** request to **http://localhost:8080/users/register**. The **Body** tab is selected, showing the following JSON payload:

```
1  {  
2      "username": "Prueba",  
3      "password": "1234"  
4 }
```

Below the body, the **Params**, **Authorization**, **Headers**, **Tests**, and **Settings** tabs are visible. The **Body** dropdown shows options: **none**, **form-data**, **x-www-form-urlencoded**, **raw**, **binary**, **GraphQL**, and **JSON**.

The response section at the bottom shows the following JSON data:

```
1  {  
2      "created": true  
3 }
```

The status bar at the bottom right shows: Status: 200 OK, Time: 621 ms, Size: 180 B, and Save Response.

- Login:

The screenshot shows the Postman interface with a successful POST request to `http://localhost:8080/users/login`. The request body contains:

```
1 ...
2   ...
3     "username": "Javier",
4     "password": "5994471abb01112afcc18159f6cc74b4f51b998086da59b3caf5a9c173cacfc5"
5 ...
```

The response status is 200 OK, time 77 ms, size 296 B. The response body is:

```
1 ...
2   ...
3     "expiration_time": 1654587476241,
4     "token": "myAGc1EowXEe3ubEZHJmGHJY-11SEXnJAx-d9_bca0G82KMw6wPaayjdDL-VOYHtpm1fuRvkLuv0F6P-2Ew2EQ=="
5 ...
```

- Update user:

The screenshot shows the Postman interface with a successful PUT request to `http://localhost:8080/users/update`. The request body contains:

```
1 ...
2   ...
3     "user": {
4       ...
5         "username": "Javier",
6         "password": "83AC674216F3E15C761EE1A5E255F867953623C88388B4459E13F978D7C846F4"
7       ...
8     },
9     ...
10    "userUpdated": {
11      ...
12        "username": "Javier",
13        "password": "5994471abb01112afcc18159f6cc74b4f51b998086da59b3caf5a9c173cacfc5"
14      ...
15    },
16    ...
17    "token": "myAGc1EowXEe3ubEZHJmGHJY-11SEXnJAx-d9_bca0G82KMw6wPaayjdDL-VOYHtpm1fuRvkLuv0F6P-2Ew2EQ=="
18 ...
```

The response status is 200 OK, time 1 m 1.08 s, size 180 B. The response body is:

```
1 ...
2   ...
3     "updated": true
4 ...
```

- Delete user:

The screenshot shows the Postman interface with a successful DELETE request to `http://localhost:8080/users/delete`. The request body contains:

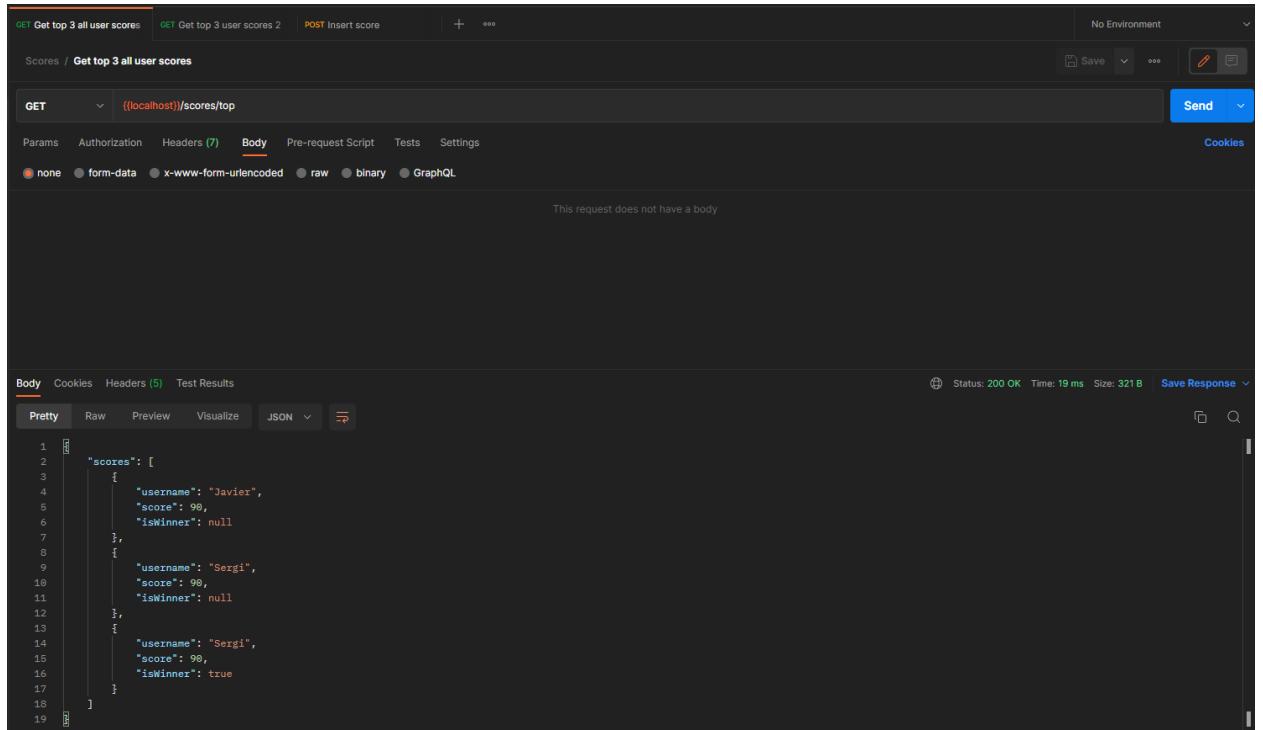
```
1 ...
2   ...
3     "username": "Prueba",
4     "password": "1234",
5     ...
5     "token": "zkR4IUaQoSuy5FS14da0lxRiPCkyFEvLknBY1JD0xoEQ2fHYGrME3UkIx0OKKn4x-F_7fIyA-mju5QebX_C74g=="
6 ...
```

The response status is 200 OK, time 403 ms, size 180 B. The response body is:

```
1 ...
2   ...
3     "deleted": true
4 ...
```

Pruebas con Scores:

- Get top 3 all user scores:

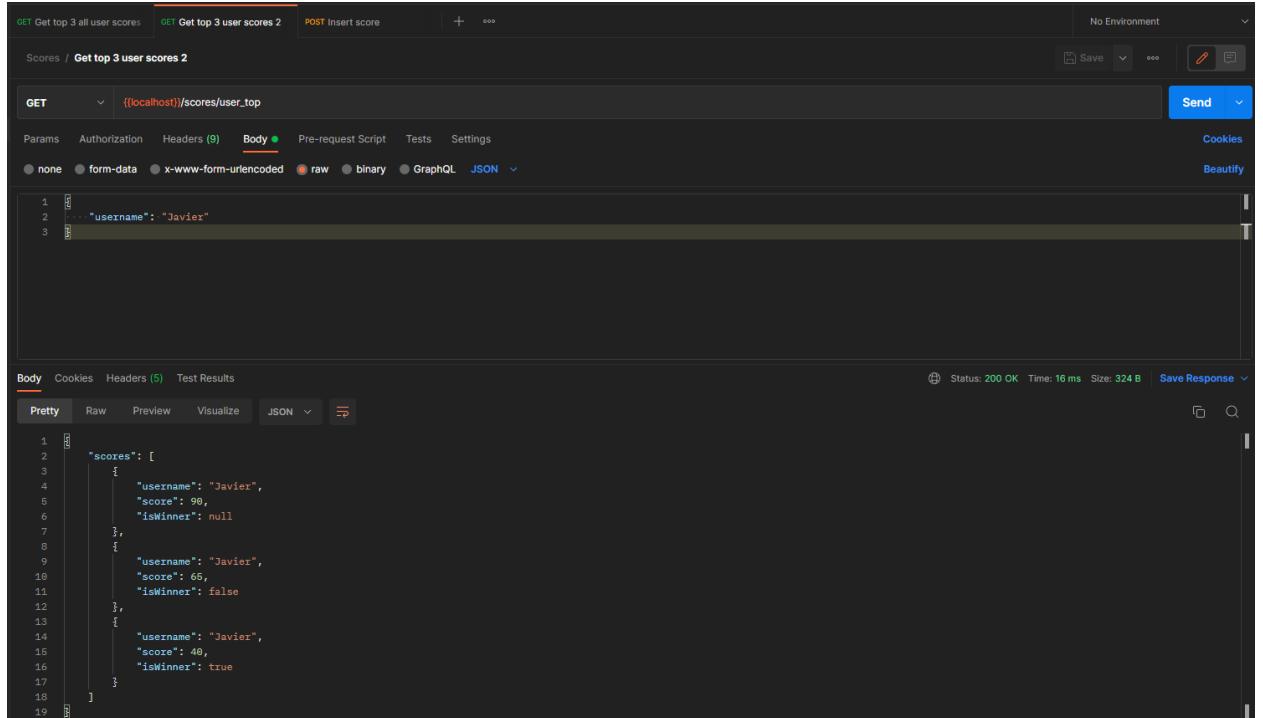


```

1
2   "scores": [
3     {
4       "username": "Javier",
5       "score": 98,
6       "isWinner": null
7     },
8     {
9       "username": "Sergi",
10      "score": 98,
11      "isWinner": null
12    },
13    {
14      "username": "Sergi",
15      "score": 98,
16      "isWinner": true
17    }
18  ]
19

```

- Get top 3 user scores:



```

1
2   ...
3     "username": "Javier"
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

```

```

1
2   "scores": [
3     {
4       "username": "Javier",
5       "score": 98,
6       "isWinner": null
7     },
8     {
9       "username": "Javier",
10      "score": 66,
11      "isWinner": false
12    },
13    {
14      "username": "Javier",
15      "score": 40,
16      "isWinner": true
17    }
18  ]
19

```

- Insert score:

The screenshot shows the Postman interface with a successful API call. The URL is `POST {{localhost}}/scores/insert_score`. The request body is a JSON object:

```

1 {
2   "server": {
3     "username": "BxJpE7Ts2QFGAuhdVtY2jvmZTzdIu",
4     "token": "b599cd5a7183db5c8388cc0iba20b783f9ad6c490b14d701217b685c754da4"
5   },
6   "player": [
7     {
8       "username": "Sergi",
9       "score": 90,
10      "isWinner": true
11    }
12  ]
13}

```

The response status is 200 OK, time 174 ms, size 180 B, and the message is "created": true.

## Pruebas en tiempo real dentro el GameServer:

Para poder realizar pruebas de posibles casos que den error, decidimos crear unos métodos que nos añadían los casos de manera estática. Estos métodos básicamente añadían unos o díoses al tablero justo antes de empezar la partida:

```

@Override
public void run() {
    try {
        this.player1.getPlayerSocket().setSoTimeout(TIMEOUT);
        this.player2.getPlayerSocket().setSoTimeout(TIMEOUT);
    } catch (SocketException e) {
        e.printStackTrace();
    }

    // Player 1 streams;
    BufferedReader player1Reader = this.player1.getReader();
    PrintStream player1Writer = this.player1.getWriter();

    // Player 2 streams;
    BufferedReader player2Reader = this.player2.getReader();
    PrintStream player2Writer = this.player2.getWriter();

    // Send to the users that they have a match and what is their position.
    player1.setPosition(position: 1);
    player2.setPosition(position: 2);
    player1Writer.println(JSONManager.mountHasOpponentJson(position: 1));
    player2Writer.println(JSONManager.mountHasOpponentJson(position: 2));

    // TODO - COMENTAR ESTO PARA QUE EL JUEGO FUNCIONE CORRECTAMENTE:
    // mountCustomBoardTest();
    // mountCustomBoard1();
    // mountCustomBoard2();
    // mountCustomBoard3();
}

```

De estos métodos, nos hemos quedado 3 para poder mostrarlos en la presentación:

```

/**
 * Caso momentáneos:
 */
private void mountCustomBoardTest() {
    this.board.get(index: 0).add(e: 1);
    this.board.get(index: 1).add(e: 1);
    this.board.get(index: 2).add(e: 1);
}

```

```
/**  
 * Caso complejo: quintuple 4 en raya.  
 */  
private void mountCustomBoard1() {  
    // Columna 0:  
    this.board.get(index: 0).add(e: 1);  
    this.board.get(index: 0).add(e: 2);  
    this.board.get(index: 0).add(e: 1);  
    this.board.get(index: 0).add(e: 1);  
    this.board.get(index: 0).add(e: 2);  
  
    // Columna 1:  
    this.board.get(index: 1).add(e: 2);  
    this.board.get(index: 1).add(e: 1);  
    this.board.get(index: 1).add(e: 2);  
    this.board.get(index: 1).add(e: 1);  
  
    // Columna 2:  
    this.board.get(index: 2).add(e: 1);  
    this.board.get(index: 2).add(e: 2);  
    this.board.get(index: 2).add(e: 1);  
    this.board.get(index: 2).add(e: 1);  
  
    // Columna 3:  
    this.board.get(index: 3).add(e: 1);  
    this.board.get(index: 3).add(e: 1);  
    this.board.get(index: 3).add(e: 1);  
    // El 4 en raya quintuple entra aqui  
  
    // Columna 4:  
    this.board.get(index: 4).add(e: 1);  
    this.board.get(index: 4).add(e: 2);  
    this.board.get(index: 4).add(e: 1);  
    this.board.get(index: 4).add(e: 1);  
  
    // Columna 5:  
    this.board.get(index: 5).add(e: 2);  
    this.board.get(index: 5).add(e: 1);  
    this.board.get(index: 5).add(e: 2);  
    this.board.get(index: 5).add(e: 1);  
  
    // Columna 6:  
    this.board.get(index: 6).add(e: 1);  
    this.board.get(index: 6).add(e: 2);  
    this.board.get(index: 6).add(e: 1);  
    this.board.get(index: 6).add(e: 1);  
    this.board.get(index: 6).add(e: 2);  
}
```

```

    /**
     * Caso especial: 4 en raya, pero última ficha en medio de 3.
     */
    private void mountCustomBoard2() {
        // Columna 0:
        this.board.get(index: 0).add(e: 1);
        // Columna 1:

        // Columna 2:
        this.board.get(index: 2).add(e: 1);
        // Columna 3:
        this.board.get(index: 3).add(e: 1);
    }

    /**
     * Caso especial: Tablero lleno y 4 en raya última ficha.
     */
    private void mountCustomBoard3() {
        // Columna 0:
        this.board.get(index: 0).add(e: 2);
        this.board.get(index: 0).add(e: 1);
        // Columna 1:
        this.board.get(index: 1).add(e: 2);
        this.board.get(index: 1).add(e: 1);
        // Columna 2:
        this.board.get(index: 2).add(e: 2);
        this.board.get(index: 2).add(e: 1);
        // Columna 3:
        this.board.get(index: 3).add(e: 2);
        this.board.get(index: 3).add(e: 1);
        this.board.get(index: 3).add(e: 1);
        this.board.get(index: 3).add(e: 1);
        this.board.get(index: 3).add(e: 2);
        this.board.get(index: 3).add(e: 1);
        // Columna 4:
        this.board.get(index: 4).add(e: 1);
        this.board.get(index: 4).add(e: 1);
        this.board.get(index: 4).add(e: 1);
        this.board.get(index: 4).add(e: 1);
        this.board.get(index: 4).add(e: 2);
    }
}

```

### 3. Pruebas de seguridad.

En cuanto a pruebas de seguridad se refiere, tenemos las siguientes

- **Contraseña encriptada**

A través de nuestra clase **EncrypterManager** encriptamos todas las contraseñas de los usuarios, y se guardan en la base de datos encriptadas.

Esta clase contiene un método para encriptar, pero no para desencriptar, por lo que nos da un plus en seguridad.

#### **password**

b599cd5a7103db5c8388cc81bafa20b783f9ad6c490b14d701...

5994471abb01112afcc18159f6cc74b4f511b99806da59b3ca...

5994471abb01112afcc18159f6cc74b4f511b99806da59b3ca...

- **Token**

Hemos decidido utilizar un token para cada usuario, este token es una clave que tiene un tiempo de validez, cuando caduca, en la aplicación android se le solicita al usuario que vuelva a iniciar sesión, en caso contrario se eliminan sus datos de la aplicación.

Cada vez que se inicia sesión se genera un nuevo token.

<b>current_token</b>	<b>creation_date</b>
I6obbyGPJSDXOJcEgSqxjRNj3tytPG4AMRMdbRdanzPk8V0Hhw...	2022-06-05 19:14:50
47tnkJGnSUjp7jEyOYJotzyjWMNrE6dlPTRfRMXfGIS7YLmyBY...	2022-06-05 21:15:01
9jdoZhcalePeRaayBS8OzXbhTQyRAmUgy2KALalivf2LBqCatU...	2022-06-05 19:55:13

- **EncryptedSharedPreferences**

Una vez iniciemos sesión, guardaremos el token proporcionado por el REST server junto con los datos del usuario en las EncryptedSharedPreferences, de esta manera cuando necesitemos los datos del usuario no tendremos que volver a hacer una petición al servidor.

Estas SharedPreferences encriptan los datos introducidos y muestran un texto no legible para el usuario.

En el caso de la contraseña, tendremos la contraseña encriptada por parte del servidor y nuevamente encriptada por las EncryptedSharedPreferences

```
<map>
    <string name="__androidx_security_crypto_encrypted_prefs_key_keyset__">12a901d5f8f63b9b08836eb1df141c3519a8ddc3088de832ea40a460ea46be
    <string name="AV7+ArQgS5HqhpDd6jPI0a5ETL0ic0gCcSX0USc">ARCQBtg7A9S1/p3p0UeaV6Z5qZkst8gFe2k8CjlGPMwnNDalWvuUAs9+ef2K7Dp56IN+yV4Hd49fW
    <string name="AV7+ArSXVNGCUmgQqXD0yBQV4BYZOCJpgF4quZ8=">ARCQBtj5vtK+cS0sv5ia0GZJ0/n6bFaFMrxJN36NQw0/OclfSz/37UVnqvSw==</string>
    <string name="AV7+ArRBjTUP6R9XMcIqXSpAWebUVC41spzxuF4f8TU578p">ARCQBthoeA6g35+4L1gWLXRuMzdTbdSfz7AANBT3QihYafCSFR57zlFzhpJk</string>
    <string name="__androidx_security_crypto_encrypted_prefs_value_keyset__">1288019a07edc74a427416192f98e4124fcecf439ff2989f2dc779c4b6f1
    <string name="AV7+ArQeHjoQ0eebvJI2mU9XSju51YTAo=o">ARCQBth+rVEj4T04x7Ja48XJFJD3ur34ZBGDQWxULIOqBlaOLZ+Lc0gLxvgXEJA5dk1q1yPRw519SIpv
</map>
```

## 4. Configuración del entorno de ejecución de la solución.

## 6. Fase de mantenimiento y documentación del proyecto.

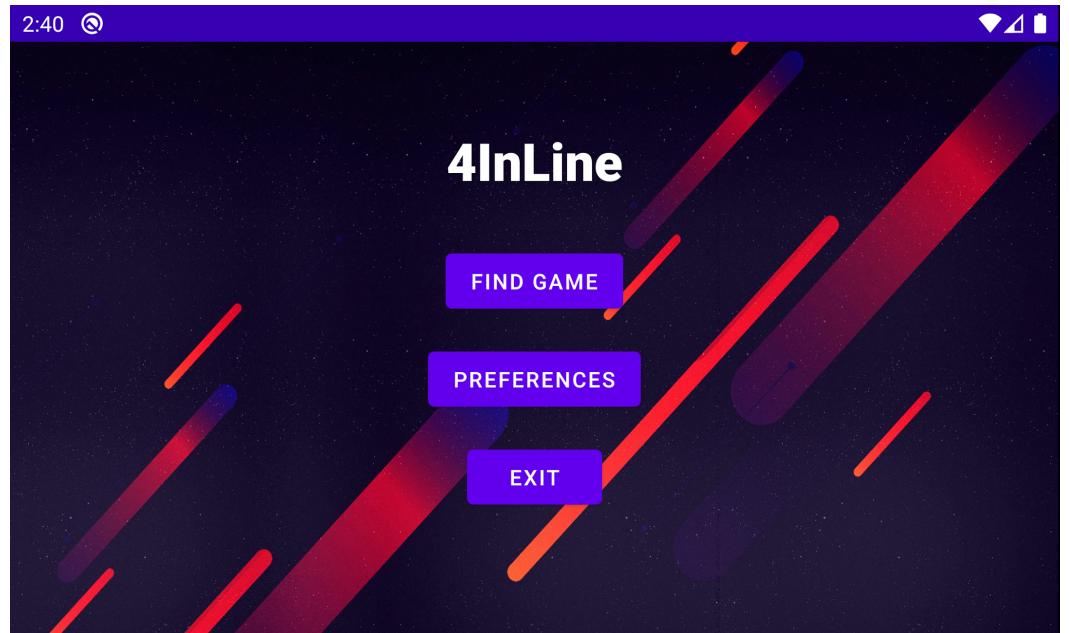
El mantenimiento de esta aplicación es relativamente simple puesto que, una vez funciona todo, solo se le pueden añadir mejoras.

### 1. Manual de usuario de la aplicación.

El uso de la aplicación es de lo más simple.

Al entrar podemos ver 3 botones.

- Buscar partida.
- Preferencias de la aplicación.
- Salir de la aplicación.



Empezando desde el más sencillo, el último botón simplemente nos cierra la aplicación.

El botón de preferencias nos abre una pantalla donde podemos personalizar opciones de la aplicación. Dentro de estas opciones, encontramos las siguientes:

- Activar o desactivar la música.
- Personalización de nuestro jugador.
- Personalización del segundo jugador.

Estas personalizaciones incluyen tipo de ícono y color del mismo.

**Music**

Enables background music



---

Player1

---

Player2

---

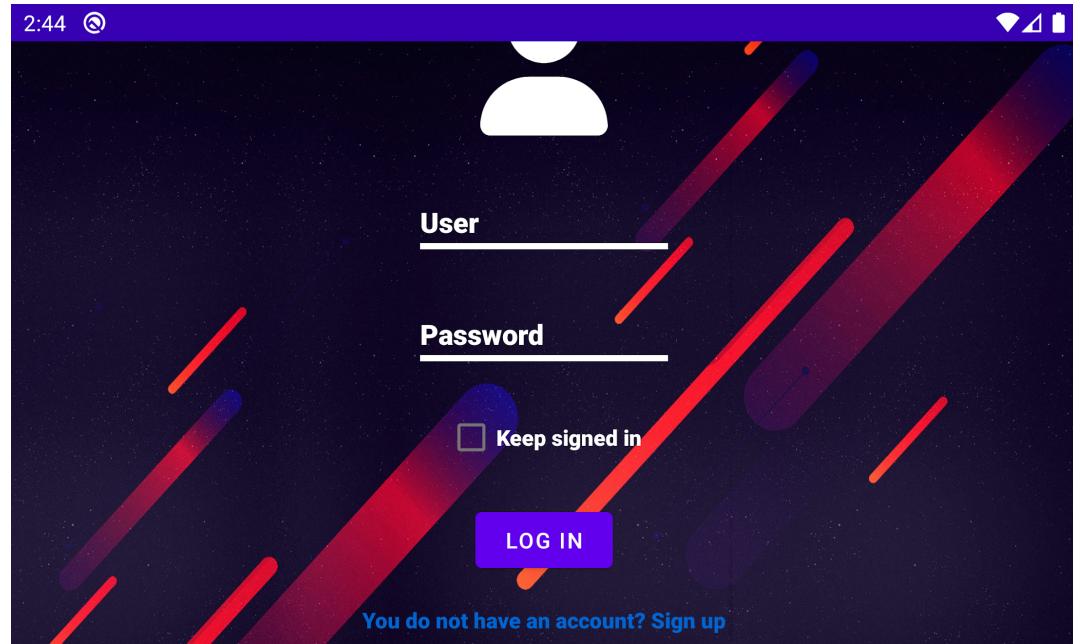
**Icon**

Changes the icon

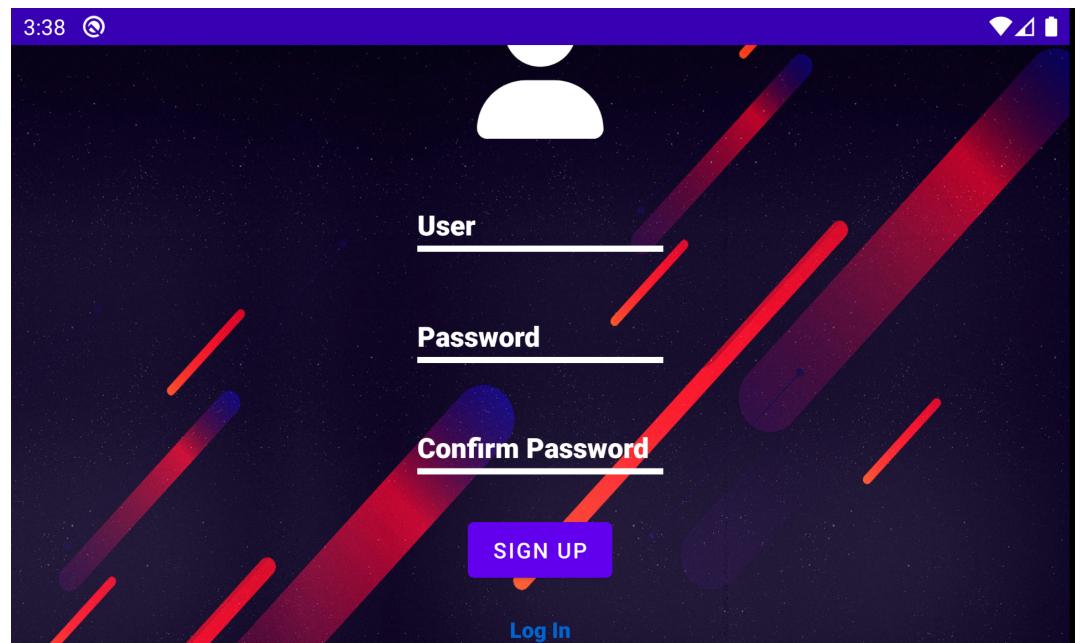
**Color**

Color selector

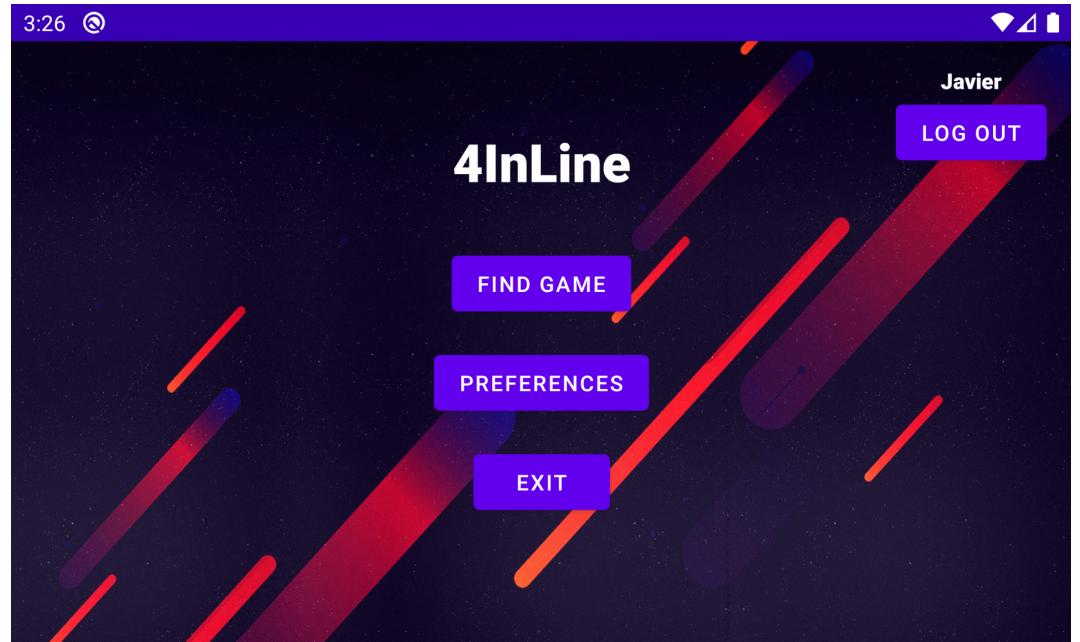
Yendo con el botón más complejo, al darle a “Buscar partida”, si no tenemos usuario iniciado, nos dirigirá a la pantalla de login.



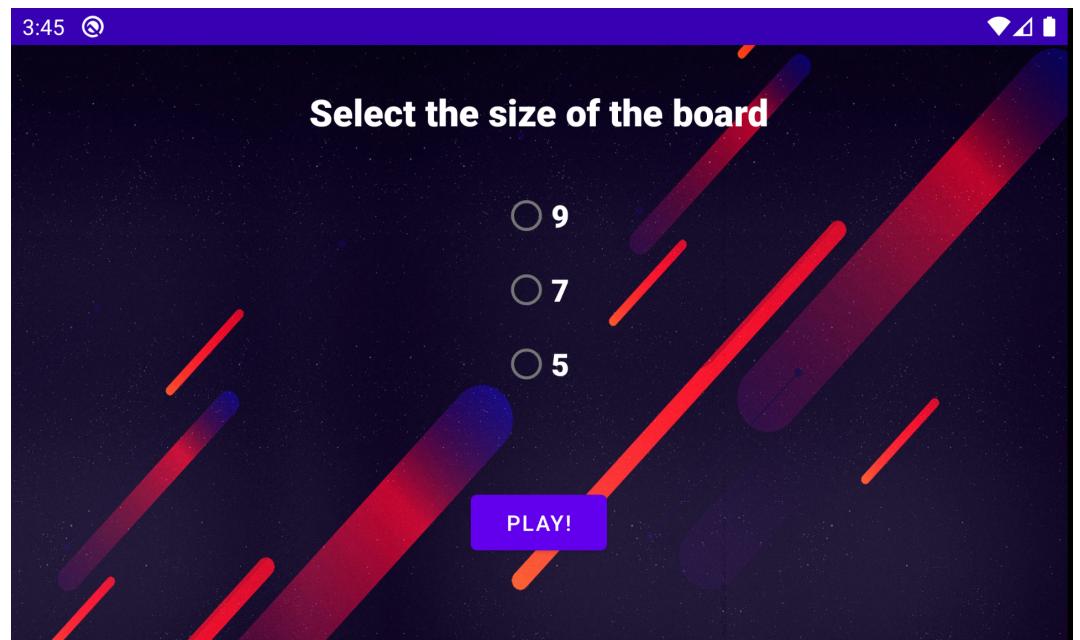
En el caso de no tener siquiera un usuario, podemos crearlo en el formulario de registro.



Una vez iniciamos sesión, podremos ver en la pantalla principal la opción de "Logout".

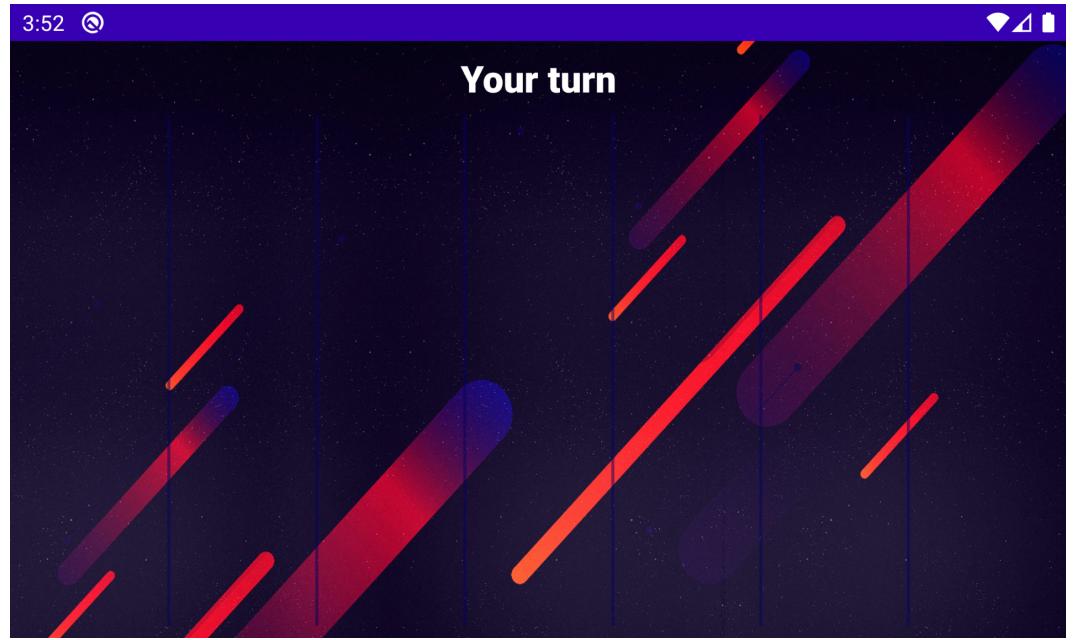


Con el usuario iniciado, al darle a buscar partida, accederemos a la pantalla de elección de tablero.



Al seleccionar un tablero y darle a jugar, nos quedaremos esperando hasta que el servidor nos empareje con otro jugador.

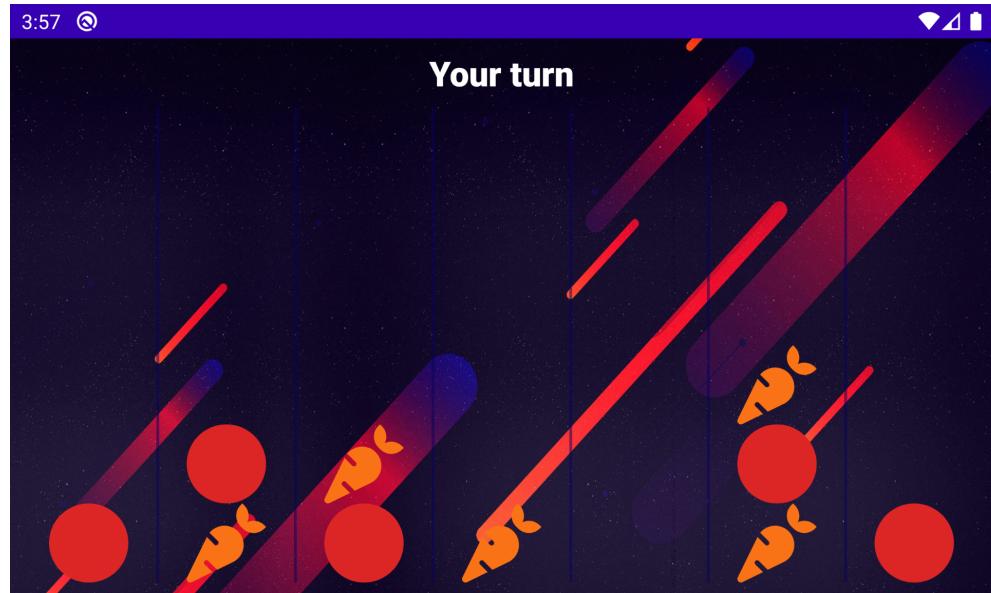
En el instante en que el servidor nos empareja, entraremos a la partida.



Todo la partida consiste en el siguiente bucle hasta que el servidor encuentre un 4 en raya, el tablero se llene al completo o el servidor se cierre:

- Jugador 1 introduce ficha.
- Servidor recibe ficha, hace cálculo interno de 4 en raya. Envía respuesta al jugador 2.
- Jugador 2 lee la ficha.
- Jugador 2 introduce ficha.
- Servidor recibe ficha, hace cálculo interno de 4 en raya. Envía respuesta al jugador 1.
- Jugador 1 lee la ficha.

Ejemplo posible tablero:



Cuando se finaliza la partida, se le muestra un mensaje al usuario en forma de Alert que contiene un mensaje informativo diciéndole si ha ganado o ha perdido.

## 2. Pruebas finales.

Como prueba final, hemos probado a instalar la aplicación Android en un dispositivo real y conectarnos al GameServer y REST Server por dirección IP local a nuestro ordenador.

La prueba ha funcionado con éxito.

