UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Aplicación web de ajedrez

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

AUTOR/A: Rangel García, Cristian

Tutor/a: Martínez Zaldívar, Francisco José

CURSO ACADÉMICO: 2023/2024

DEDICATORIA

A mi madre por haberme apoyado
durante estos últimos años de carrera.

Resumen

El presente TFG consiste en una aplicación web del juego de mesa y también considerado deporte, conocido como ajedrez, a través de una API externa la cual devolverá los movimientos de las piezas de ajedrez utilizando una base de datos MongoDB. Se implementará una serie de funciones extras, como por ejemplo dos modos de juego (para jugar con personas reales o con un *bot*), temporizadores de 5 minutos para modo normal y de 1 minuto para modo rápido. Además, se creará otra API REST para gestionar una lista de partidas y jugar varias sin necesidad de esperar a terminar una partida para empezar otra (principalmente para el modo de 5 minutos), gestionar los usuarios, las victorias…

Resum

Aquest TFG consisteix en una aplicació web del joc de taula i també considerat esport, conegut com a escacs, a través d'una API externa la qual retornarà els moviments de les peces d'escacs utilitzant una base de dades MongoDB. S'implementaran una sèrie de funcions extres, com ara dos modes de joc (per jugar amb persones reals o amb un *bot*), temporitzadors de 5 minuts per a mode normal i d'1 minut per a mode ràpid. A més, es crearà una altra API REST per gestionar una llista de partides i jugar-ne diverses sense necessitat d'esperar a acabar una partida per començar-ne una altra (principalment per a la manera de 5 minuts), gestionar els usuaris, les victòries…

Abstract

The present TFG consists of a web application of the board game and also considered sport, known as chess, through an external API which will return the moves of the chess pieces using a MongoDB database. A series of extra functions will be implemented, such as two game modes (to play with real people or with a *bot*), timers of 5 minutes for normal mode and 1 minute for fast mode. In addition, another REST API will be created to manage a list of games and play several games without having to wait to finish a game to start another (mainly for the 5 minutes mode), manage users, victories...

RESUMEN EJECUTIVO

La memoria del TFG del Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación debe desarrollar en el texto los siguientes conceptos, debidamente justificados y discutidos, centrados en el ámbito de la ingeniería de telecomunicación

CONCEPT (ABET)	**CONCEPTO (traducción)**	**¿Cumple? (S/N)**	**¿Dónde? (páginas)**
1. IDENTIFY:	1. IDENTIFICAR:		
1.1. Problem statement and opportunity	1.1. Planteamiento del problema y oportunidad	S	1
1.2. Constraints (standards, codes, needs, requirements & specifications)	1.2. Toma en consideración de los condicionantes (normas técnicas y regulación, necesidades, requisitos y especificaciones)	S	2-3
1.3. Setting of goals	1.3. Establecimiento de objetivos	S	2
2. FORMULATE:	2. FORMULAR:		
2.1. Creative solution generation (analysis)	2.1. Generación de soluciones creativas (análisis)	S	21-24, 40-44
2.2. Evaluation of multiple solutions and decision-making (synthesis)	2.2. Evaluación de múltiples soluciones y toma de decisiones (síntesis)	S	40-44
3. SOLVE:	3. RESOLVER:		
3.1. Fulfilment of goals	3.1. Evaluación del cumplimiento de objetivos	S	45
3.2. Overall impact and significance (contributions and practical recommendations)	3.2. Evaluación del impacto global y alcance (contribuciones y recomendaciones prácticas)	S	46

Índice

Capítulo 1. Introducción

El ajedrez, un juego de mesa y también considerado un deporte que ha perdurado durante años por todo el mundo, desafiando y creando grandes mentes como el actual campeón Magnus Carlsen basado en estrategia e inteligencia, ahora se fusiona con la tecnología para ofrecer una experiencia aún más dinámica y accesible. En este proyecto se presenta una aplicación web que aprovecha una API externa (referenciada en el capítulo 5) para brindar una plataforma interactiva y envolvente para los amantes del ajedrez.

Una API es un mecanismo que permite a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos. Permite que sus servicios se comuniquen con otros sin necesidad de saber cómo están implementados simplificando el desarrollo de aplicaciones permitiendo ahorrar tiempo y dinero.

Esta aplicación no solo ofrece la opción de disfrutar del juego del ajedrez *online*, sino que también integra características adicionales que aumentan la experiencia de juego. Con la utilización de una base de datos MongoDB y una API externa, la aplicación proporciona los movimientos de las piezas de ajedrez de manera precisa y eficiente.

Uno de los aspectos destacados de esta aplicación es la posibilidad de incluir dos modos de juego distintos. Por un lado, se ofrece la oportunidad de enfrentarse a la máquina en el que se podrá hacer donde y cuando se desee. Es muy útil para aprender a jugar por tu propia cuenta e incluso para mejorar tu rendimiento. El otro modo de juego que se implementará será el modo *multiplayer*, un modo *multiplayer* en el que el jugador podrá disfrutar de la experiencia y la oportunidad de desafiar a otros oponentes en cualquier lado del mundo.

Además, la aplicación incorpora otros dos modos de juego para el modo *multiplayer*, un modo de juego con temporizadores de 5 minutos en el que poder jugar de manera más relajada y poder pensar y analizar más estrategias de juego. Por otro lado, se dispondrá de un modo rápido de juego con temporizadores de 1 minuto, en el que ambos usuarios disfrutarán de una partida más intensa y para jugadores que dispongan de poco tiempo.

Para mejorar aún más la experiencia de juego, se desarrollará una API REST que facilitará la gestión de usuarios, estadísticas de victorias y otras características que enriquecerán la experiencia global de la aplicación.

La memoria se desarrollará haciendo un seguimiento de los ficheros y rutas correspondientes del proyecto referenciado al final de esta memoria y subido en un repositorio alojado en la plataforma GitHub.

1.1 Objetivos

El objetivo del proyecto es desarrollar una aplicación web que facilite la interacción inicial y el progreso continuo de partidas de ajedrez, ya sea enfrentando a un jugador contra una máquina o permitiendo partidas entre personas proporcionando una plataforma intuitiva y funcional. Estas características permiten a los usuarios disfrutar del ajedrez en línea de manera flexible y accesible. Además, se fomenta tanto la práctica individual contra un oponente virtual como la competencia amistosa entre jugadores reales.

1.2 Metodología y herramientas a utilizar

Se adoptaron varias tecnologías en la metodología y herramientas utilizadas para el desarrollo de la aplicación, lo que garantizó un proceso eficiente. Durante el desarrollo, la API del juego de ajedrez se autohospeda localmente utilizando Node.js como entorno de ejecución. Como base de datos se emplea Mongodb para respaldar la API y, además, se implementó un contenedor Docker para facilitar la gestión y portabilidad del entorno de desarrollo.

En cuanto al desarrollo de la aplicación en sí, se emplearon las siguientes herramientas y tecnologías:

- **Editor de código:** se utiliza principalmente Visual Studio Code para escribir y depurar el código de la aplicación. El desarrollo eficiente fue facilitado por su integración con extensiones y su sólido soporte para JavaScript y TypeScript.

- **Frontend**: se emplea Angular como marco de trabajo para desarrollar la interfaz de usuario. Angular es comúnmente utilizado para el uso de aplicaciones web interactivas y escalables.

- **Backend**: se emplea Node.js por su compatibilidad con JavaScript, lo que facilitó mantener una coherencia en el desarrollo tanto del *frontend* como del *backend* siguiendo una arquitectura cliente-servidor.

- **Base de Datos**: se usa MongoDB como base de datos para la API y la aplicación, lo que garantiza la coherencia en el manejo de datos y simplifica la integración entre la capa de datos y la lógica comercial.

- **PM2** (Process-Manager 2): se ha empleado para levantar tanto el *backend* como la API en producción. Con PM2 la aplicación está en funcionamiento sin interrupciones, reiniciándose de forma automática en caso de fallo, de esta forma se mantiene el *backend* ejecutándose.

- **Docker**: se ha usado para facilitar la gestión de la base de datos, con Docker se facilita instalar todas las dependencias de MongoDB.

- **OVHcloud:** como proveedor del servicio VPS (máquina virtual).

- **GitHub:** para alojar el proyecto.

El desarrollo exitoso de la aplicación web del juego de ajedrez se debió en gran parte al uso fundamental de estas herramientas y tecnologías, que permitieron una implementación coherente, eficiente y escalable del proyecto.

Capítulo 2. Desarrollo y resultados del trabajo

2.1 API

Para el desarrollo de esta aplicación web, se ha empleado una API externa para la lógica del juego, es decir, para el movimiento en el tablero de ajedrez de todas y cada una de las piezas que componen cada partida de ambos jugadores.

Además, permite la opción poder utilizar 2 principales modos de juego para el diseño de la aplicación:

- Uno de ellos es la opción de poder jugar contra un *bot*, de esta forma no se requiere la conectividad contra otro jugador y el tiempo de espera empleado hasta que este se conecte en la partida. También, cuenta con la posibilidad de poder entrenar para mejorar en cualquier momento y de esta forma agudizar y hacer sólido el nivel de planificación y estrategia que este juego requiere.

- El otro modo principal con el que la API cuenta, además de la ya mencionada, es la opción de poder jugar de manera multijugador, bien con desconocidos o mejor aún con amigos. Con la importante cualidad de demostrar quien de todos es el mejor, creando torneos entre grupos o incluso simplemente probando aperturas y modos creativos de hacer jaque mate desarrollando más el nivel de estrategia mezclado con diversión.

Esta API referenciada al final de la memoria, utiliza la base de datos MongoDB cuyos modelos se encuentran en la carpeta "`api/models`".

Para poder emplearla, la API se ha hecho operativa con el comando "`node index.js`" en el desarrollo y en producción se utilizó PM2 (Process-Manager 2) para mantener el *backend* ejecutándose y de esta forma puede ser utilizada en el proyecto a través de la URL de la API.

2.1.1 Login

Para que la base de datos almacene las credenciales de cada jugador, así como sus victorias y derrotas correspondientes, es necesario pasar por un proceso de registro y seguidamente cada usuario pueda identificarse con los datos introducidos previamente.

Para ello, se ha creado un formulario con los campos de nombre de usuario (username), email y contraseña, además de dos botones para cancelar la operación en cualquier momento o directamente para confirmar una vez se hayan introducido los datos correspondientes (register).

Una vez que el usuario haya rellenado dicho formulario y confirme sus credenciales en el botón correspondiente, automáticamente se le redirigirá a la pantalla de identificación de inicio y aparecerá un mensaje confirmando que su cuenta ha sido creada correctamente como bien se puede ver en la figura 1:

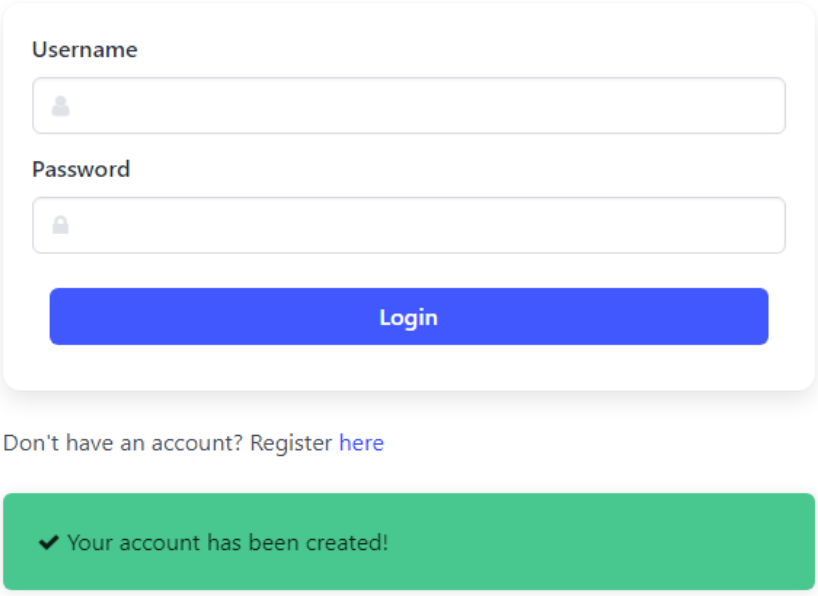


Figura 1. Registro con éxito.

Dicha implementación se realiza en los ficheros de la ruta "`frontend/src/app/service/user.service.ts`" en el cual se crea el método "`register`" ("`frontend/src/app/register`") para establecer la lógica del formulario mencionado, utilizando TypeScript y mostrándolo en la página web con HTML, además, "`frontend/src/app/notification`" para crear y revelar la notificación mostrada en la figura 1 incorporando además un fichero CSS para diseñar el estilo mostrado produciendo una animación de aparición temporal gracias a instrucciones de código como `@keyframes, visibility` y `fade-in` principalmente.

Una vez que el usuario ha sido registrado, se comprueba si dispone del token (JSON Web Token), es decir, un fragmento de información que se utiliza para verificar la identidad de un usuario. Si el usuario dispone de dicho JWT será redirigido automáticamente al menú de la app, pero sino será enviado al "`login`".

Cuando un usuario se registra y no existe en la base de datos, a su contraseña se le aplicará una función hash con un "*salt*", esto es un valor aleatorio de ruido esencialmente útil para evitar que hayan más de un hash idéntico en el caso de que existan contraseñas iguales y de esta forma impedirlo. Posteriormente, tanto el hash como el *salt* se almacenan en la base de datos y cuando el usuario se identifique, la contraseña se le aplicará una función hash con el *salt* del mismo usuario. Si los *hashes* coinciden, se genera un JWT que contiene tanto el ID como el nombre del usuario utilizando la clave secreta para firmar dicho token, luego se envía al cliente para que este lo guarde en su almacenamiento local. A partir de ahí, todas las peticiones del cliente llevarán el JWT en la cabecera de autorización y el *backend* verificará este JWT en cada petición, de esta forma, cada vez que el usuario actualice la página evitará tener que volver a identificarse.

Llevando esta explicación teórica a nivel de programación, los ficheros creados y el orden de su uso por pasos son:

1. "`App.component.ts`" ("`frontend/src/app`"): donde se crea un método con condiciones para redirigir al usuario al sitio correspondiente acuerdo a si dispone de un token o no una vez registrado. Además, se emplea el fichero "`user.service.ts`" ("`frontend/src/app/services`") para crear todos los métodos de identificación y registro para ser declarados fácilmente cuando sea necesario, como el método "`doVerification`" en este caso.

2. "`Login.component.ts`" ("`frontend/src/app/components/login`"): en caso de que el usuario no disponga del token, será redirigido aquí para introducir sus credenciales.

3. "`UserController.js`" ("`backend/controller`"): aquí si el usuario existe se imprimirá un mensaje para hacérselo saber y si no existiese, se llamará al método definido en la línea 152 para aplicar la función hash a la contraseña del usuario con el *salt* una vez el jugador se haya registrado y posteriormente se le notificará como registro exitoso como se puede apreciar en la línea 43. Si los datos son correctos, se genera el JWT conteniendo el ID y nombre de usuario firmado por la clave secreta (línea 28). Dicha clave secreta se encuentra en el fichero "`.env`" del *backend*.

4. "`Login.component.ts`": una vez obtenido el JWT se almacenará junto al nombre de usuario en el almacenamiento local y finalmente el jugador será redirigido al menú.

Adicionalmente, se hace uso de un interceptor que intercepta todas las solicitudes del *frontend* al *backend* para añadir la cabecera de autorización. Este aparece en todas las solicitudes una vez que el usuario se ha identificado, entonces dichas solicitudes se interceptan para añadirle el token (menos las solicitudes de la API) con el objetivo de no tener que hacer en cada solicitud "`setHeader`" evitando repetir ese código miles de veces. Dicho interceptor se crea en el *frontend*, pero adicionalmente en el *backend* hay otro interceptor, específicamente en el fichero "`index.js`" para ser utilizado en el fichero "`app.config.ts`" en la carpeta "`frontend/src/app`".

En el "`index.js`", el interceptor creado permite que rutas como "`register`" o "`login`" pasen sin autenticación para que posteriormente, con el uso de condicionales, se obtenga el token del encabezado autorización y se verifique y codifique el token utilizando la clave secreta. Si la verificación es errónea, se marcará el token como inválido, pero si es exitosa se extraerá el nombre del usuario decodificado.

En el fichero "`app.config.ts`", se crea la función del interceptor para clonar la solicitud con la nueva cabecera, es decir, la solicitud original ("`req`") se clona y se le añade una nueva cabecera de autorización con el valor del token.

A continuación, se expondrán una serie de imágenes para mostrar el resultado de lo mencionado con anterioridad:

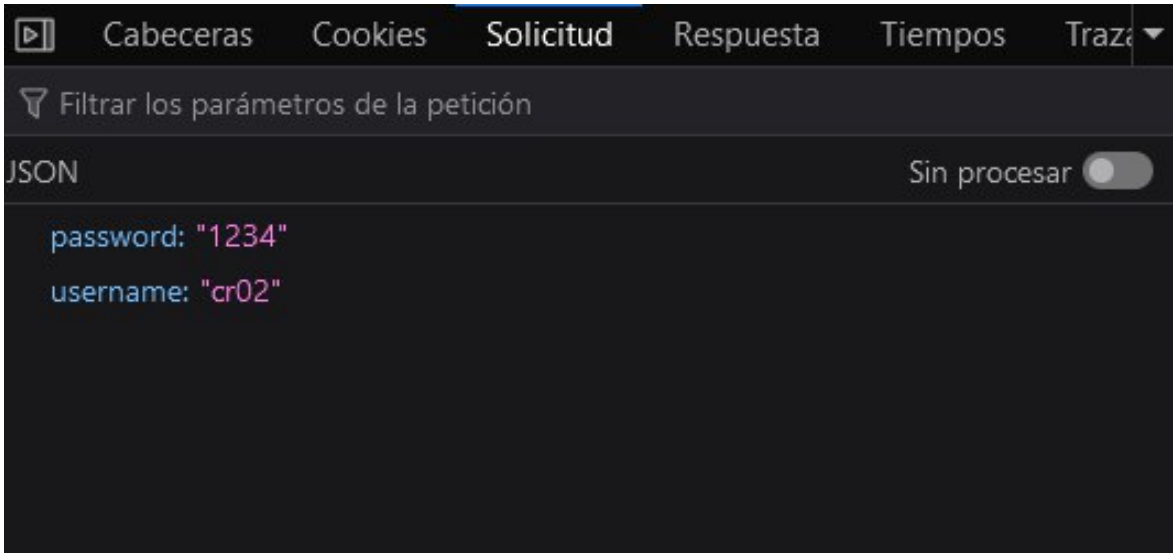


Figura 2. Captura de solicitud cuando un usuario es identificado.

La figura 2 es el resultado de la solicitud cuando un usuario es identificado en la página inicial de la aplicación, en este caso con las credenciales “cr02” como nombre de usuario y “1234” como contraseña.

Figura 3. Captura de la respuesta cuando un usuario es identificado.

La figura 3 representa la respuesta de la solicitud mostrada en la figura 2, que contiene tanto el ID como el nombre del usuario firmado con la clave secreta. Dicho token y nombre de usuario son guardados en el almacenamiento local.

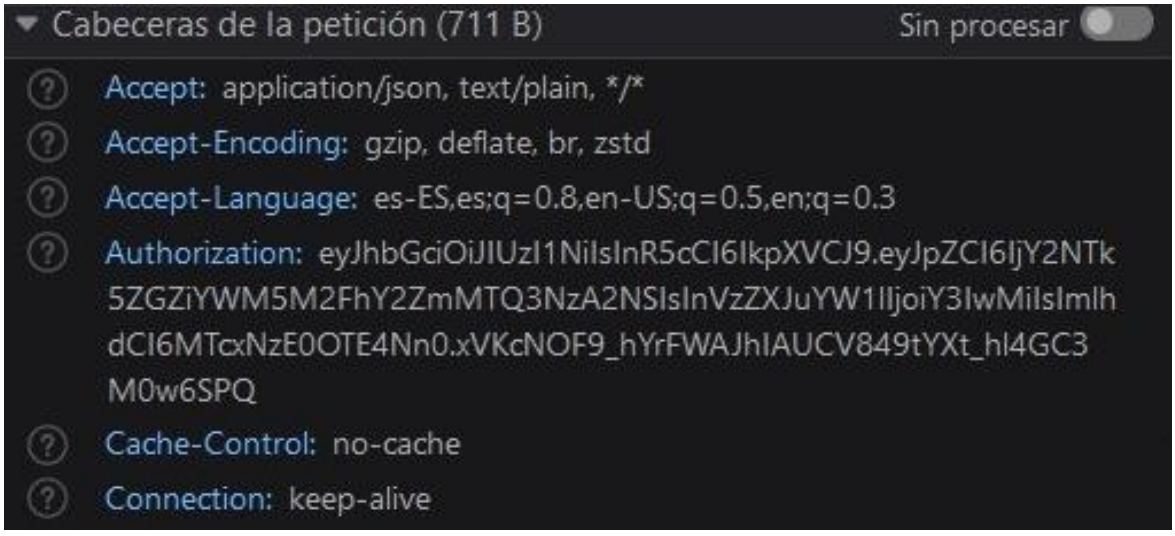


Figura 4. Captura de la cabecera de petición de autenticación.

Todas las solicitudes que se realicen llevarán el JWT en la cabecera de autorización que se muestra. La figura 4 es la captura resultante de la cabecera de petición como ejemplo cuando el jugador accede a su perfil en el menú, el cual se desarrollará en el siguiente punto.

2.1.2 Menú del juego

Una vez que el usuario ha completado exitosamente el proceso de autenticación será redirigido automáticamente a la página del menú. El menú está compuesto por un mensaje de bienvenida junto al nombre del usuario identificado que se extrae de la base de datos local y 4 botones en los que el jugador podrá escoger entre un modo de juego contra un *bot* (`"Singleplayer"`), en el que al clicar en dicho botón se le redirigirá directamente a la partida contra el *bot* facilitado por la API. Otro modo sería una partida normal multijugador, en el que el usuario podrá jugar contra oponentes reales desde cualquier lugar del país una partida con temporizadores de 5 minutos para cada jugador, de esta forma tendrán tiempo de pensar bien sus estrategias de juego y dar lugar a partidas épicas enriqueciendo ese nivel estratégico que cada persona posee. Además, cuenta con otro modo rápido *multiplayer* con la diferencia en que los temporizadores serán de 1 minuto para cada jugador, de esta forma ambos usuarios sentirán la emoción de usar su estrategia antes de que se les venga el tiempo encima.

Por otro lado, tanto en el modo multijugador normal como en el rápido, mientras que un jugador crea una partida (uniéndose primero a esta) y espera al segundo usuario, se le redirigirá a una pantalla de carga donde se le mostrará una serie de consejos, los cuales dispondrán de una duración de 7 segundos para que el usuario pueda leerlo hasta que aparezca el siguiente y así servirle de utilidad y se le sea ameno su tiempo de espera.

La creación de estos consejos en el proyecto se lleva a cabo a través de los ficheros `"tips.ts"` (`"frontend/src/assets"`), donde se crean los tips tanto título como descripción; `"tips.component.ts"` donde se establece un intervalo de 7 segundos en el método `"ngOnInit"`. Cada 7 segundos, `"showTip"` se establece en false para ocultar el consejo mostrado por pantalla y después de 1 segundo, se actualiza `"currentTip"` al siguiente índice y `"showTip"` se establece en true para mostrar el siguiente consejo y finalmente con el método `"ngOnDestroy"` se limpia el intervalo para impedir que se ejecute en segundo plano; y tips.component.html para imprimir los consejos por pantalla.

En la figura 5, se muestra un ejemplo del resultado final de un consejo cuando un jugador espera a que otro se conecte a esa partida:

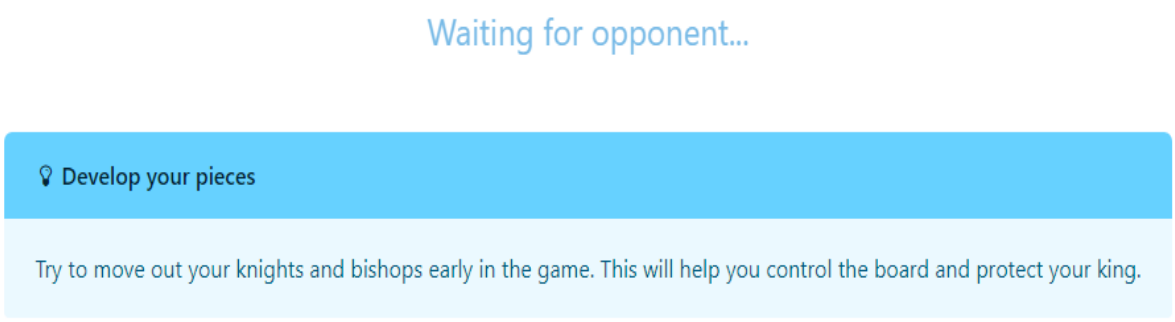


Figura 5. Consejo en tiempo de espera de la conexión del rival.

Por último, el usuario dispone del cuarto y último botón para ver las estadísticas de su perfil, donde se le mostrará su nombre de usuario, email, la fecha de creación de su cuenta, el número total de victorias, empates y derrotas y un botón de "`logout`" que le redirigirá al inicio de la aplicación para identificarse por si el jugador desea cerrar sesión y crearse o usar otra cuenta.

Este proceso lo llevan a cabo ficheros como el "`user.service.ts`" ("`frontend/src/app/services`"), donde se declara el método "`getProfileInfo`" que obtiene la información del perfil del usuario desde el *backend* utilizando una solicitud HTTP GET y el método "`logout`" que elimina el token y el nombre de usuario del almacenamiento local redirigiendo al usuario a la página de inicio de sesión, asegurando que el usuario no esté autenticado. Otro fichero que interviene es el "`user.ts`" ("`frontend/src/app`") para inicializar mediante un constructor las propiedades de cada usuario.

Estos métodos mencionados con anterioridad van a ser empleados en el fichero "`profile.components.ts`" ("`frontend/src/app/components/profile`") para obtener la información del perfil del jugador e imprimirse por consola el número de victorias y derrotas del usuario y además, se empleará el método instanciado "`logout`" para el cierre de sesión ya mencionado. Seguidamente, se realizará un formulario en HTML con las estadísticas del usuario y el botón de "`logout`" como se observa en la figura 6:

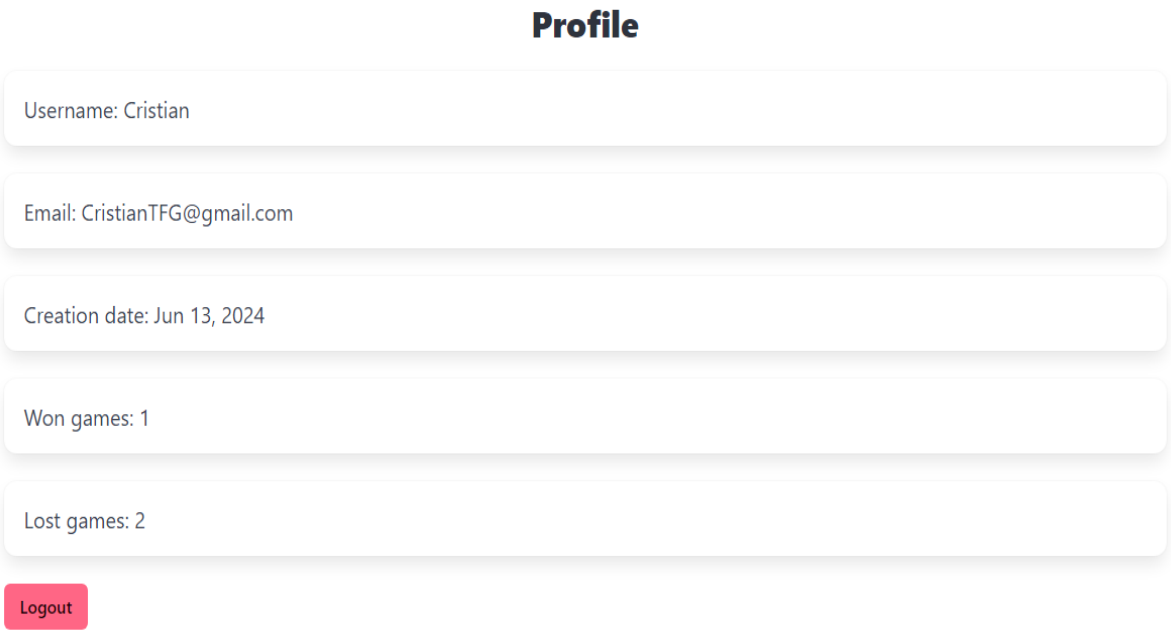


Figura 6. Perfil del usuario.

Finalmente, para los 3 modos de juego y el perfil del jugador que forman el menú de la aplicación se empleará el fichero "`main.menu.component.ts`" ("`frontend/src/app/components/main.menu`") para redirigir al correspondiente modo que el usuario seleccione y extraer del almacenamiento local el nombre del usuario. Dichos botones son diseñados en la misma carpeta con HTML donde además se mostrará el mensaje de bienvenida del jugador identificado dando como resultado la figura 7 de a continuación:

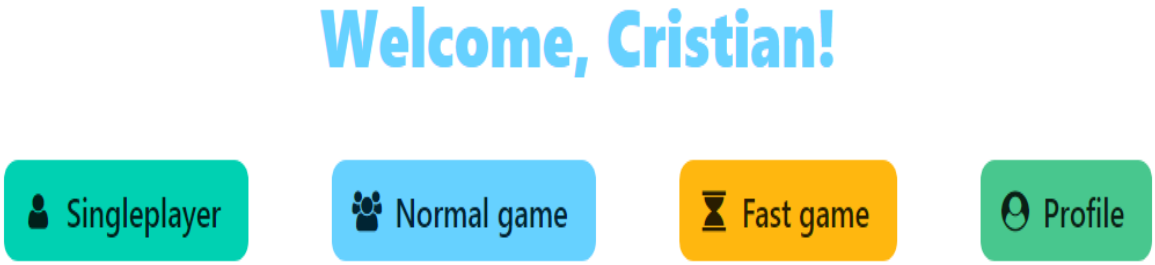


Figura 7. Menú de la aplicación web.

2.1.3 Tablero y piezas del ajedrez

Como muchos juegos de mesa, el ajedrez requiere un tablero para poder jugar. Este está formado por 64 casillas, 8x8, donde el eje horizontal se numera con letras desde la A hasta la H mientras que el eje vertical se enumera con números del 1 al 8 en orden ascendente desde la perspectiva del jugador con piezas blancas o del 8 al 1 en orden descendente para el jugador de piezas negras.

Para dibujar el tablero siguiendo el reglamento correspondiente del juego y teniendo en cuenta que la posición inicial de donde se encuentre la dama debe de coincidir con su mismo color de pieza, se lleva a cabo en los ficheros de la ruta "`frontend/src/app/components/board`". En el fichero TypeScript, se asigna en dos variables las coordenadas de ambos ejes mencionados anteriormente para ser utilizados en el fichero HTML y de esta forma diseñar el tablero de forma correcta. Una vez establecido el diseño 8x8, se alterna cada casilla de un color distinto. Si se observa el fichero de diseño, se establece una condición en el que, si la suma de los índices de letra y número es impar, la casilla será dibujada de color marrón oscuro simulando el negro y si no cumple esa condición se dibujará de color marrón claro simulando el blanco, de esta forma siendo realizado desde la perspectiva de las piezas blancas, se cumple una regla principal de la posición de las damas con su mismo color correspondiente de casilla.

Para la perspectiva del jugador con las piezas negras, se realiza un giro de tablero de 180 grados en HTML y se establece su correcta lógica para dicho jugador en TypeScript, concretamente entre la línea 49 y 64, de esta forma su eje vertical irá desde la posición 8 al 1 en orden descendente.

Una vez establecido el tablero, queda el diseño de las piezas que lo componen: 2 torres, 2 caballos, 2 alfiles, un rey, una dama y 8 peones para cada jugador. Para ello, se emplean imágenes PNG (referenciadas) colocadas en la ruta "`frontend/src/assets`" para ser usadas en el método "`getImageForPiece`" del fichero "`piece.ts`" ("`frontend/src/app/model`") y de esta forma retornar la imagen como una cadena de texto con su color y tipo de pieza correspondiente.

Finalmente, solo queda colocar las piezas en su posición inicial, que según el reglamento del ajedrez el orden correcto para ambos jugadores sería:

- En la primera fila de izquierda a derecha: torre, caballo, alfil, dama, rey, alfil, caballo y torre.
- En la segunda fila: un peón por cada casilla hasta terminar dicha fila.

En el código "`board.component.ts`" ("`frontend/src/app/components/board`"), de forma sencilla se implementó dicha lógica (específicamente en el método de la línea 143) y una vez diseñado todo al completo en el fichero HTML de la misma carpeta, daría como resultado la figura 8:

Figura 8. Tablero y piezas del ajedrez.

2.2 Modos de juego

A continuación, se va a llevar a cabo el desarrollo de los modos de juego Single Player y los dos modos multijugador, tanto el modo normal compuesto por temporizadores de 5 minutos y el modo rápido compuesto por temporizadores de 1 minuto.

2.2.1 Modo Single Player

Para explicar el desarrollo del modo de juego contra la máquina, se va a llevar a cabo una serie de esquemas que faciliten su compresión de manera más conceptiva y visual junto a su correspondiente explicación teórica para que posteriormente sea más sencillo su entendimiento a nivel de programación.

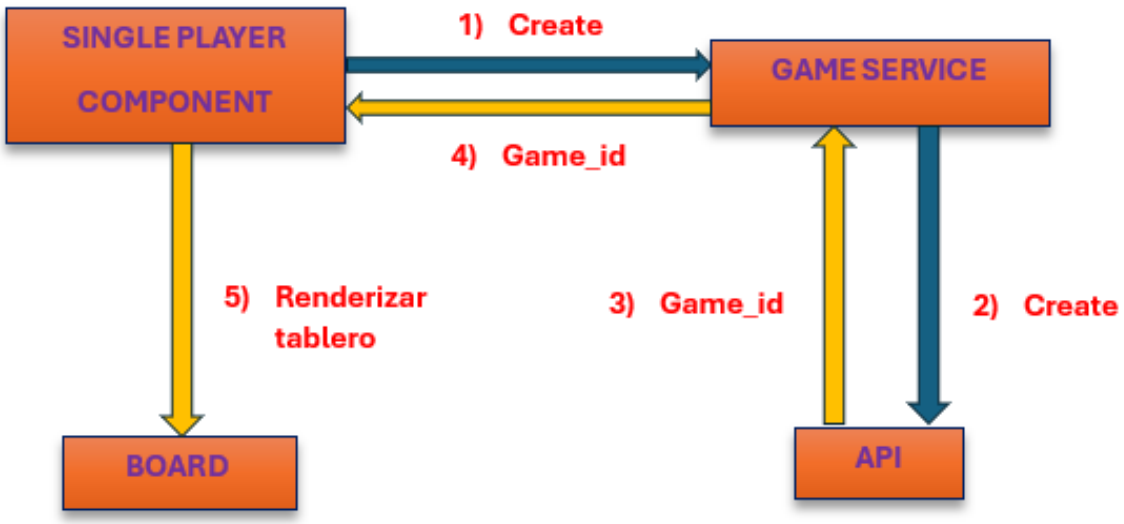


Figura 9. Esquema del Single Player para crear partida.

La figura 9 representa un esquema de la creación de una partida en el que los 4 primeros pasos realizan dicha creación y finalmente se redirige a una partida con el tablero y piezas. Se emplearán 3 ficheros para poder llevarlo a cabo: el fichero "`singleplayer.component.ts`" ("`frontend/src/app/components/singleplayer`"), "`game.service.ts`" ("`frontend/src/app/services`") que será el encargado en comunicarse con la API realizando peticiones HTTP a través de la URL de esta y "`board.component.ts`" ("`frontend/src/app/components/board`").

Cuando "`SingleplayerComponent`" se inicializa, llama a "`newSinglePlayerGame`" del "`Game Service`" (con el método "`ngOnInit`") y este realiza una petición HTTP a la API con el tipo "`one`" del modo individual. Cuando la solicitud se completa, el "`SinglePlayerComponent`" recoge la respuesta/Observable obtenida a través del método "`suscribe`", este es un método de los Observables que se encargará de notificar una vez que se tenga un valor, en este caso el de respuesta de la API. Una vez que se obtiene la respuesta, se actualiza la propiedad "`game_id`" con el valor recibido de la respuesta de la API almacenado en la variable "data" y se creará la partida contra la máquina.

Para la explicación de seleccionar una pieza y que esta devuelva los posibles movimientos a realizar, se lleva a cabo el proceso del siguiente esquema en el que se ha tomado como ejemplo el peón situado en la casilla de inicio B2:

Figura 10. Esquema del Single Player para posibles movimientos por pieza

Como cabe observar en la figura 10, el jugador realiza la selección de la pieza que desea mover y su casilla se marcará en azul, para ello, le comunica dicha selección al "`Single Player`" y este al "`Game Service`" para que realice la petición a la API con los posibles movimientos que puede realizar la pieza seleccionada. En este ejemplo, el jugador ha seleccionado el peón de B2, como está en la posición inicial y siguiendo el reglamento del ajedrez, el usuario tiene 2 opciones: desplazar el peón una casilla (B3) o dos (B4). Esas son las dos posibilidades que la API va a devolver realizando el proceso inverso de petición para comunicárselo al jugador (situado en el "`Board`") y automáticamente esas casillas se marcarán en verde para que el usuario sepa dónde puede mover la pieza.

Para ello, se han desarrollado una serie de métodos en el fichero "`board.component.ts`" como por ejemplo el método "`getPieceAtPosition`" con el objetivo de obtener la pieza que se encuentra en una posición específica del tablero convirtiendo la letra de la fila en un índice numérico y devolviendo la pieza en la posición indicada y el método "`canSelectSquare`" que utiliza el método anterior para verificar si hay una pieza en la casilla indicada en cuyo caso devolverá "true" y de lo contrario "false"; el método "`isPossibleMove`" para comprobar si el movimiento a una casilla específica es posible comparando las coordenadas dadas con una lista de movimientos posibles; y el método "`onClickSquare`" utilizado para manejar el evento de clic en una casilla. Este método gestiona tanto la selección de la pieza como el movimiento de esta, pero esta última parte se comentará en el siguiente esquema. Para ello, primero se verifica si hay una pieza seleccionada (`"this.selectedPiece"`) y si el movimiento a la casilla clicada es posible (`"isPossibleMove"`). A continuación, utiliza el método

"getPieceAtPosition" para obtener la pieza en la casilla clicada y se realiza la condición en la que, si la casilla no está vacía, se selecciona la pieza y se emite un evento "pieceClicked" con la posición de la pieza seleccionada. Con este evento el fichero "board.component.ts" se comunica con el componente padre ("singleplayer.component.html") para realizar la solicitud correspondiente a través del decorador "@Output() pieceClicked".

Seguidamente, en el fichero HTML del "Board", se utiliza la directiva "ngClass" de Angular permitiendo agregar clases CSS de forma dinámica en un elemento HTML. En este caso se utiliza para marcar en azul la casilla en la que, si sitúe la pieza que el jugador seleccione a partir de la condición que se establece utilizando el método "canSelectSquare" definido anteriormente, en el cual se comprueba que una casilla pueda ser seleccionada, lo que implica que en esa casilla haya una pieza devolviendo el valor "true" y con "selectedPiece" se verifica que la casilla actual sea la misma casilla que la pieza seleccionada.

En el fichero del "singleplayer.component.ts", se define el método "pieceClicked" para manejar el evento de una pieza clicada, llamando al método "checkPossibleMoves", que se empleará en su correspondiente fichero HTML. Además, se solicitarán los posibles movimientos de la pieza que se ha seleccionado al "game.service.ts" a través del método "checkPossibleMoves" el cual se encargará de enviar una solicitud HTTP POST a la API solicitando dichos movimientos de la pieza seleccionada a través del parámetro "position" (con "set" se inicia el "body") y además, el "game_id" para indicar que se corresponde con esa partida a través de su indicador (con "append" se añade al "body"). Finalmente se realiza la solicitud con el endpoint, es decir, con la URL indicando que el tipo es "one" (el modo individual), seguido de "/moves" para que la API reconozca que solicitud se está realizando. La estructura del endpoint se obtiene en la documentación de la propia API.

Después, a partir de ese mismo método utilizado en el fichero "singleplayer.component.ts" para realizar la solicitud, la respuesta obtenida de la API con los posibles movimientos (B3 y B4) se verán reflejados en la variable "data" del método "suscribe" y se actualizará la propiedad "possibleMoves" con los movimientos obtenidos en la respuesta.

El "Board" recibirá la respuesta a través del decorador "@Input() possibleMoves" de la línea 2 donde la respuesta se recibe a través del fichero "singleplayer.component.html" entre corchetes ("[possibleMoves]"). Finalmente, se marcarán las casillas en verde para que el usuario sepa que opciones tiene para realizar el movimiento.

En la figura 11 se mostrará el resultado cuando el usuario selecciona la pieza del ejemplo que se ha desarrollado, implementando en el HTML del componente padre el nombre del usuario identificado y el del *bot* que será fijo dicho nombre, el color del nombre se irán alternando en acorde a quien lleve el turno:

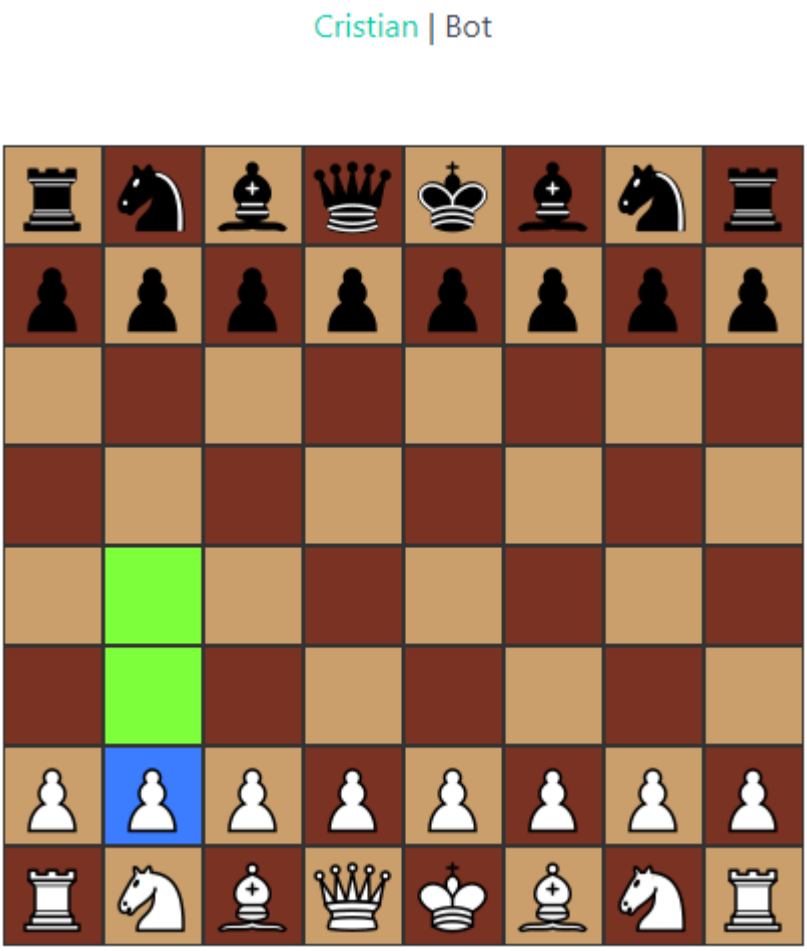


Figura 11. Tablero con la pieza B2 seleccionada.

En el esquema de la figura 12, se definirá la estructura del movimiento de la pieza:

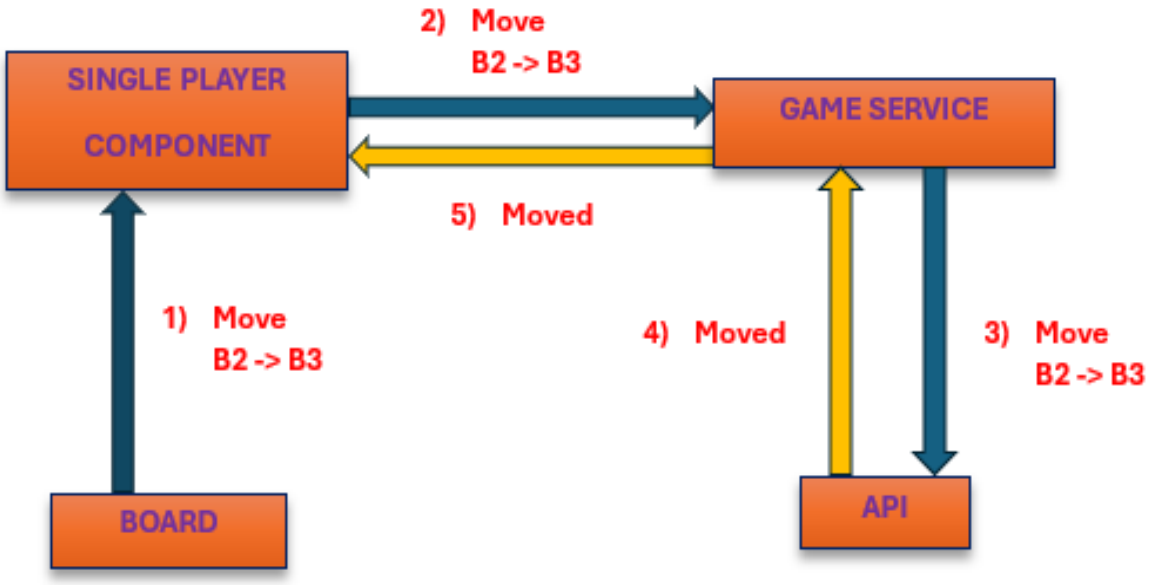


Figura 12. Esquema del Single Player para mover pieza por parte del jugador.

Cuando el jugador finalmente ha decidido a que casilla mover la pieza (en este caso moverá el peón a la casilla B3), el "`Board`" le comunicará la decisión tomada a su padre (`"Single Player"`) y este a su vez al "`Game Service`" con el objetivo de comunicárselo a la API y se obtenga la confirmación de esta.

Para empezar, los métodos a utilizar en este esquema vuelven a ser el método "`isPossibleMove`" utilizado en el método "`onClickSquare`", que además de gestionar la selección de pieza vista en la figura 11, gestiona el movimiento de la pieza. Es decir, se llama a "`onClickSquare`" y se verifica si hay una pieza seleccionada y el movimiento es posible a través del primer método mencionado. Se emite el evento "`pieceMoved`" para que el componente padre maneje la actualización de la API y a continuación se mueva la pieza utilizando el método "`movePieceInBoard`". Este método gestiona que la pieza a su nueva posición en el array que representa el tablero, la casilla original de la pieza se vacíe, se deseleccione la pieza y se limpien los movimientos posibles.

Al igual que se ha comentado en la figura 11, el "`Board`" le comunicará el movimiento realizado al componente padre a través de otro decorador de salida (`"pieceMoved"`) emitiendo así dicho evento.

En el fichero "`singleplayer.component.ts`", se desarrolla el método "`movePiece`". Este método no actualiza el tablero en el *frontend* sino que realiza una solicitud a la API a través

del "`Game Service`" donde llamará al método "`movePieceByPlayer`" pasándole el ID de la partida y las coordenadas origen y destino de la pieza. En esta solicitud se agregan parámetros en el "`body`" como el "`from`" que representa la coordenada inicial de la pieza, el parámetro "`to`" que representa la coordenada final de la pieza y el parámetro "`game_id`" para identificar esa misma partida. Finalmente, se envía una solicitud POST al `endpoint` "`/one/move/player/`" y se obtiene el Observable de respuesta que se almacenará en la variable "data" de la solicitud del componente padre. Cuando se recibe la respuesta, se cambia el estado del turno a "`false`" indicando que el turno lo tiene ahora la máquina, se deselecciona la pieza y se verifica si el movimiento realizado es o no jaque mate, de lo contrario es el turno del *bot* cuya estructura esquemática está representada en la figura 13 siguiente:

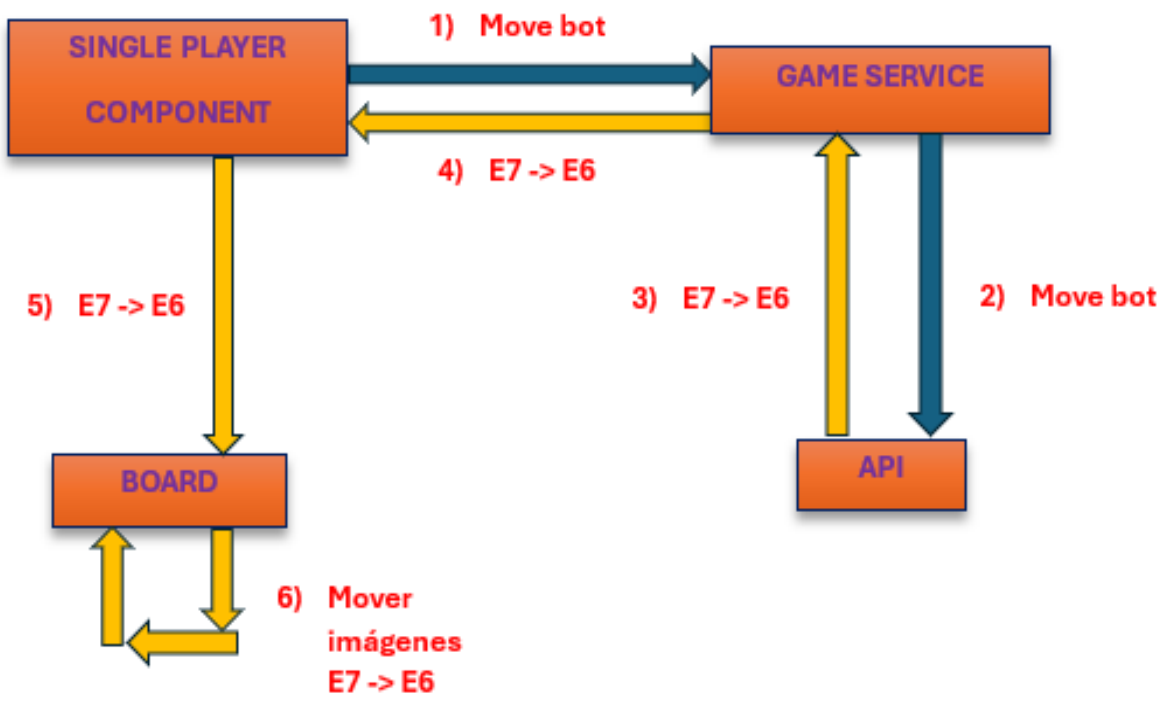


Figura 13. Esquema del Single Player para mover pieza por parte de la máquina.

Este proceso es muy similar al de la figura 12, pero con la diferencia de que antes, cuando el jugador realizaba el movimiento, se veía reflejado al momento en el tablero y después se realizaba el procedimiento de informar a la API. Ahora el proceso es inverso, es decir, cuando la máquina realiza el movimiento, primero se informará a la API y luego se verá reflejado en el tablero. En este ejemplo, la API ha realizado el movimiento de peón de E7 a E6.

Para ello, se define el método "`movePieceByBot`" en el componente padre. Este método lo primero que hace es verificar si el jugador tiene el turno, de lo contrario se introducirá un retardo de un segundo que será el tiempo en que la máquina tarde en realizar su movimiento de pieza. A continuación, se realiza una petición a la API a través del "`Game Service`" utilizando el ID de la partida como parámetro. La solicitud a la API, la cual decide el movimiento de la máquina y devuelve la información del movimiento realizado que será recogido por el método "suscribe" del componente padre. Cuando la respuesta se recibe, se restablece el turno del jugador (`"true"`) y se asignan las coordenadas al atributo "`pieceMovedExternally`" para ser utilizado en el fichero HTML del mismo componente actualizando así la parte visual del tablero en el "`Board`" a través de su correspondiente decorador de entrada.

El resultado obtenido se verá reflejado en la siguiente figura:

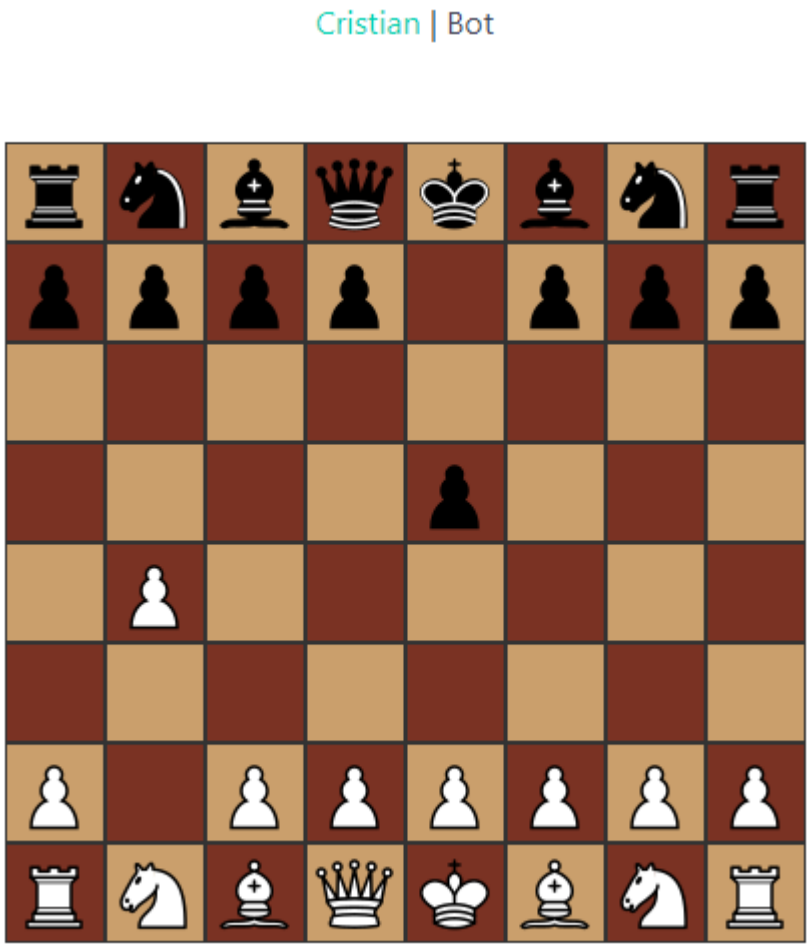


Figura 14. Tablero con el movimiento de la IA.

Adicionalmente, se implementó el movimiento de enroque por parte de la IA. En la siguiente figura se aprecia la arquitectura escogida para su realización:

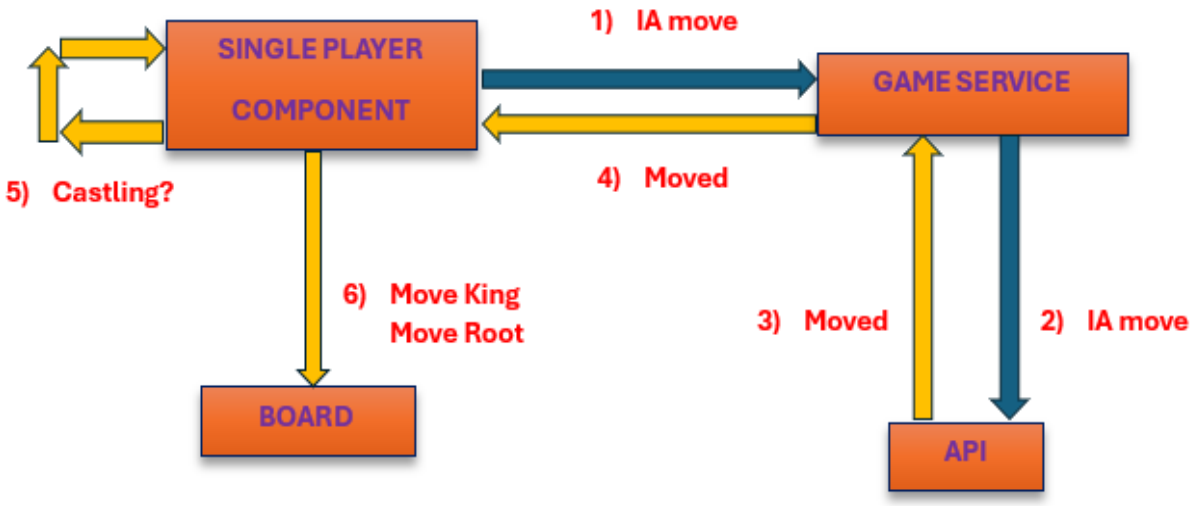


Figura 15. Esquema del Single Player para el enroque por parte de la IA.

Este movimiento consiste en dejar el rey y una de las dos torres de los extremos (donde se quiera enrocar) sin mover, de esta forma cuando no haya obstáculos de por medio (caballo y alfil) el jugador pueda realizar un enroque corto (con nomenclatura SAN O-O) donde el rey se desplaza directamente 2 casillas en dirección a la torre más cercana y la torre se desplaza a la casilla situada al lado del rey, pero en dirección opuesta. O bien un enroque largo (con nomenclatura SAN O-O-O) en el que el rey se mueve dos casillas hacia la dirección de la dama y dicha torre se sitúa nuevamente al lado del rey en dirección opuesta. Según la normativa del ajedrez, para poder llevarse a cabo este movimiento se deben en tener en cuenta 3 condiciones: que el rey no se halla movido de su posición inicial, la torre hacia donde se quiere realizar el enroque tampoco y que no haya ninguna pieza como obstáculo entre el rey y la torre.

La implementación se realiza en el método "`movePieceByBot`" en el componente padre, que tras la respuesta de la petición del movimiento de la IA por parte de la API añade el SAN y llamará al método "`aiDoneCastling`" (definido al final del fichero) para verificar si el movimiento realizado se corresponde a un enroque corto o largo. Tras la respuesta obtenida, se obtiene la posición inicial del rey y se establecen dos condiciones. Una para el enroque corto (O-O) donde se calcula la posición inicial de la torre (en este caso la H) para ser movida a la casilla de la columna F y la del rey a la G. Para la otra condición de enroque largo (O-O-O) es similar, pero con la coordenada inicial de la torre en A para ser movido a la columna D y la del rey a C.

La siguiente figura representa la máquina realizando el enroque corto:

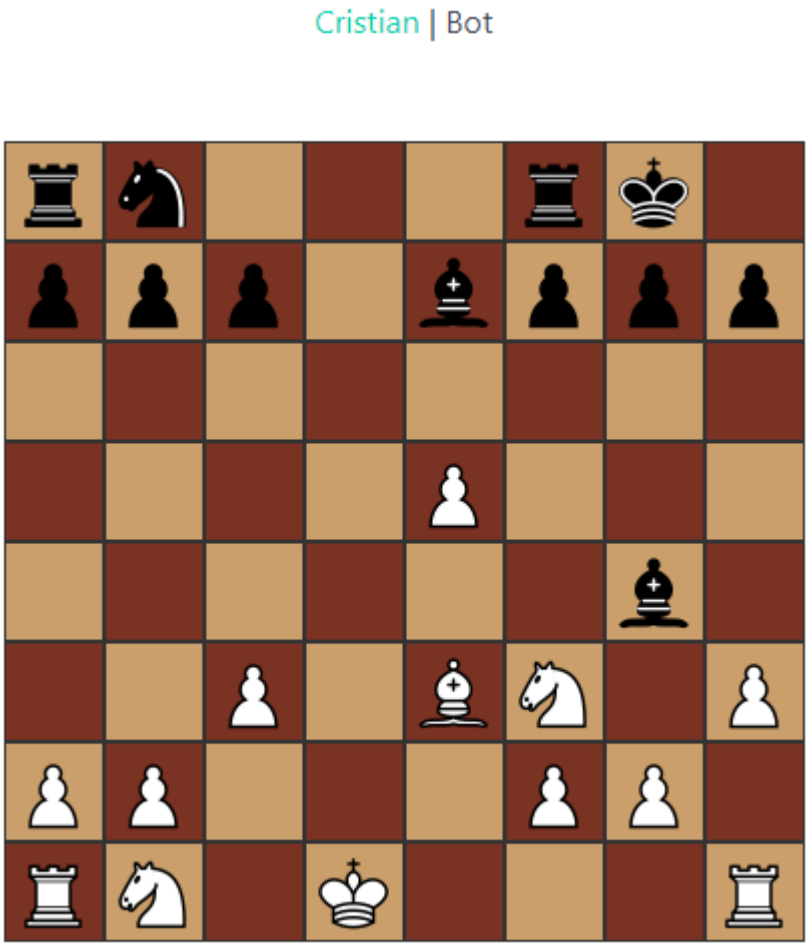


Figura 16. Enroque corto por parte de la IA.

En el esquema de la siguiente figura, se va a estudiar el caso de promoción de piezas por parte de la máquina:

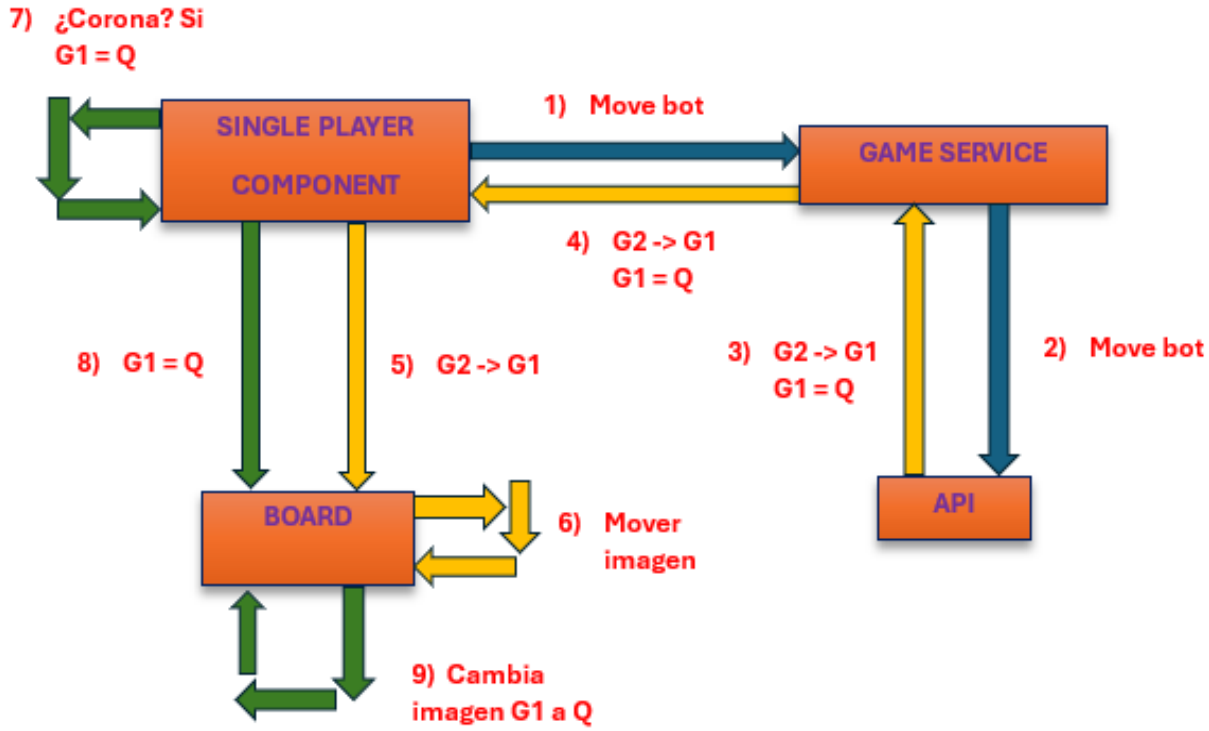


Figura 17. Esquema del Single Player para promocionar una pieza.

El objetivo de promocionar un peón, también conocido por coronar, consiste en llevar el peón hasta la última fila del tablero y una vez que logre llegar a ese objetivo, el jugador (en este caso la máquina) podrá cambiar su peón por cualquier otra pieza del tablero excepto por otro rey. De esta forma se obtendrá más ventaja contra su oponente y poder ganar la partida de forma más sencilla.

En la figura 15, la IA realiza un movimiento de promoción que en este caso es el peón situado en la casilla G2 coronará en la casilla G1 y se verá reflejado en la parte visual (`"Board"`), después de verificar que se ha promocionado a una dama (G1=Q) se volverá a actualizar la parte visual del tablero realizando un cambio de imagen de peón a dama como en este caso.

El procedimiento del movimiento por parte de la máquina es el mismo que el de la figura 14 con la diferencia de la promoción. Cuando se recibe la respuesta por parte de la API, en el método del componente padre `"movePieceByBot"` se verifica si la pieza ha promocionado llamando al método `"canPromote"`. Este método define una expresión que busca patrones de promoción en la cadena SAN, es decir, en SAN se muestra como "=Q", "=R", "=B" y "=N" (en mayúsculas o minúsculas), lo cual indica que el peón a coronado a una dama, torre, alfil o caballo. Si match es `"null"` es que no se ha encontrado ninguna coincidencia pero sino se retomará el primer grupo de captura (`"match[1]"`) que será la pieza promovida. Una vez llegado el mensaje del componente padre al `"Board"` a través de su respectivo `"input"`, se define un método llamado

"PiecePromoted_" el cual se activará en el momento en que se reciba una pieza promocionada por parte del componente padre y tiene como función sacar la pieza de la posición de coronación y reemplazarla por una nueva pieza según el tipo escogido del mismo color. Este método utilizará otros métodos definidos en sus correspondientes ficheros para llevarlo a cabo: en "type.ts" que exporta el tipo de "PieceCoord" para la coordenada de la pieza, "piece.ts" ("frontend/src/app/model") que exporta la clase "Piece" con el tipo de pieza y color y "chessPiece.ts" ("frontend/src/models") que exporta la enumeración "PieceType" donde se asigna cada pieza y color con su inicial correspondiente. En el fichero "piece.ts", se utiliza el método "promote" que verifica el color de la pieza y actualiza la nueva pieza coronada generando la nueva ruta de imagen que se visualizará en el tablero a través del fichero HTML del "Board". A continuación, se mostrará dicha promoción realizada por parte de la máquina:

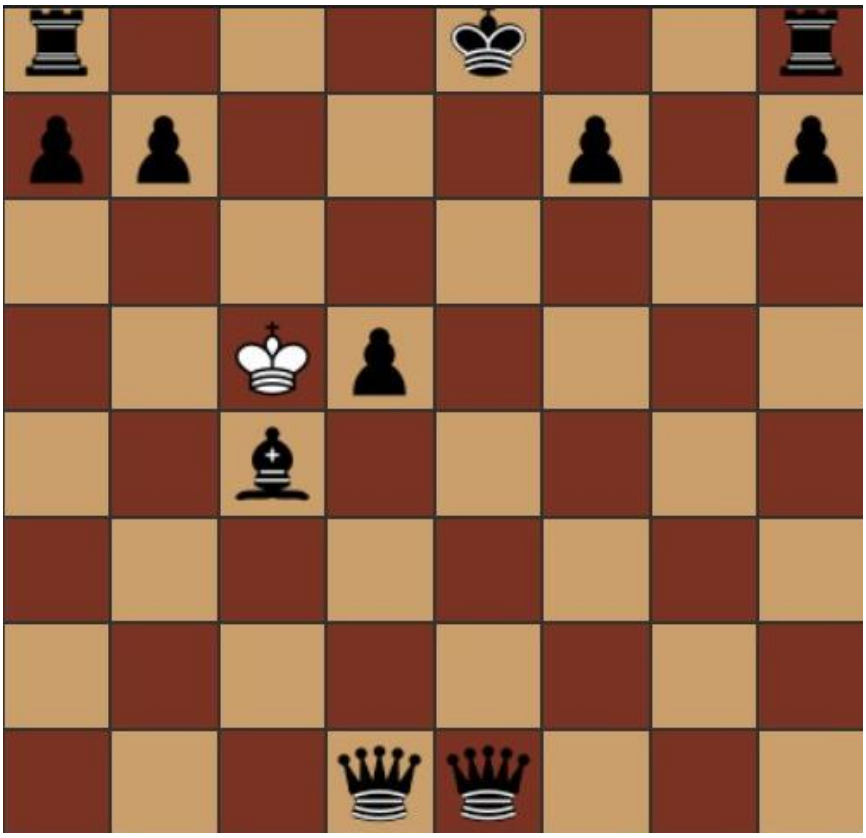


Figura 18. Promoción de dama por parte de la IA.

Finalmente, la partida puede terminar por dos razones: por victoria y derrota en el que se le aplica jaque mate al rey; por empate en caso de que ambos jugadores no dispongan del suficiente material para realizar dicho jaque mate; por rey ahogado, que consiste en dejar al rival sin ninguna opción de movimiento y a la vez sin aplicarle ningún tipo de jaque; y por repetición para que la partida no entre en un bucle infinito.

Esta comprobación la realiza el método "checkCheckMate" en el componente padre, enviando una solicitud POST a la API para verificar el estado actual del juego pasándole como parámetro el ID de la partida. El valor de respuesta de la API se almacenará en la variable data y se asignará a la variable "gameStatus" para así seguidamente evaluar el resultado obtenido con los diferentes casos dentro de un "switch" cubriendo así todas las opciones ya mencionadas de fin de partida en caso de empate y para el caso de victoria o derrota, si es el turno del jugador, el jugador pierde y se incrementará el contador de derrotas realizándose una petición al *backend* para actualizar la base de datos utilizando el fichero routes.js (encargado de especificar todas las rutas y su manejador) y el fichero "use.service.ts" ("frontend/src/app/services"), encargado de realizar dichas peticiones al *backend*. Sino gana se realizará el mismo proceso, pero incrementando el contador de victorias.

Además, se ha implementado el historial de movimientos realizados por ambos jugadores cuando la partida termine. Esto lo lleva a cabo nuevamente el componente padre, que realiza otra solicitud a la API especificando en el endpoint que desea todos los movimientos a través del ID de partida que se pasa como parámetro.

La ejecución del juego se cortará con el método "triggerModal" el cual activará el modal (ventana emergente) de fin de juego lanzando el correspondiente mensaje registrado en el "switch" anterior y devolviendo un resumen del juego con la lista de movimientos obtenidos el cual mostrará cada movimiento con el color de la pieza correspondiente a través del fichero CSS ("frontend/src/app/components/game-over"). Los métodos "isModalActive()" y "closeModal" se encargarán de controlar la visibilidad del modal.

Por último, en esa misma ventana emergente habrá un botón que se encargará de redirigir al usuario al menú utilizando para ello el método "handleReturnToHome" situado en el fichero "game-over.component.ts" y su correspondiente HTML para visualizarlo.

En la siguiente figura se observará el resultado de fin de partida con el historial de movimientos comentado:

Figura 19. Historial de movimientos en fin de partida.

2.2.2 Modo multijugador

En este modo de juego, el jugador podrá fusionar su sabiduría y estrategia con la diversión y entretenimiento jugando contra adversarios desde cualquiera de sus dispositivos a través de una conexión *online* y disfrutar así de los dos modos de juegos en los que se divide el modo multijugador. Para estos dos modos de juego, se emplearán 2 temporizadores diferentes, de 5 minutos para el modo normal y de 1 minuto para el modo rápido.

Al igual que en el punto 2.2.1, se emplearán esquemas para realizar de forma más sencilla la explicación de este modo el cuál se emplearán web sockets para el establecimiento de la conexión *online* en el *backend*.

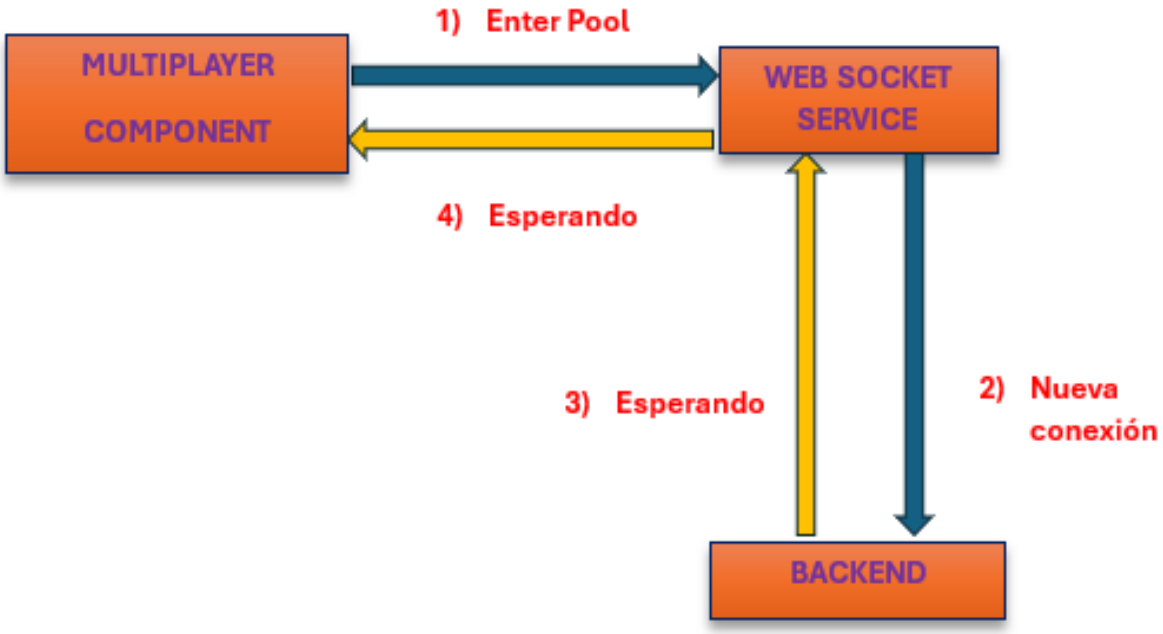


Figura 20. Esquema del multiplayer para establecer una conexión.

Para empezar, el usuario selecciona el modo *multiplayer* que quiere jugar, ya bien sea el modo normal o rápido y automáticamente se ejecutará el método "`ngOnInit`" una sola vez con la finalidad de suscribirse y quedarse escuchando los eventos del WebSocket, empleando para ello el método "`subscribeToMessages()`" ubicado en el fichero "`websocket.service.ts`" ("`frontend/src/app/ services`"). Esto llevará a cabo el envío de un mensaje al servidor a través del WebSocket por parte del cliente e invocará el "`if else`" correspondiente. El método "`ngOnInit`" se ubica en el fichero "`multiplayer.component.ts`"("`frontend/src/app/components/multiplayer`").

A continuación, se realiza el establecimiento de conexión. Desde el componente *multiplayer*, se obtiene en un método constructor el nombre del usuario guardado en el almacenamiento local para a continuación llamar al método "`enterpool`", el cual se encargará de enviar un mensaje al servidor WebSocket informando que ese jugador ha entrado en el pool y se esperará a que el segundo jugador se conecte. Durante ese tiempo de espera, al primer jugador se le mostrará una

serie de consejos para que su espera sea más amena y a la vez le sea útil para elaborar sus estrategias como bien se comentó en el punto 2.1.2 en el menú del juego.

Una vez que haya jugadores en cola, se realiza el proceso de emparejamiento en el *backend* utilizando el método "`matchMaking`" en el fichero "`wsController.js`". Este método verificará si hay un jugador en cola para después buscar en esa misma cola otro jugador que no sea el mismo (línea 165), si encuentra un oponente lo saca de la cola utilizando el método "`splice`" definido en la librería y se le asignará a cada uno un color aleatorio a través del método "`assignColors`" (p1 será el jugador de piezas blancas y p2 el jugador de piezas negras). Posteriormente, el *backend* llama a la API para generar una nueva partida de dos jugadores (método "`createNewGame`"). Este método retorna una promesa que se resuelve con la respuesta del servidor que contiene el ID de la partida ("`game_id`"). Finalmente, ambos jugadores serán notificados indicando que se ha encontrado un oponente y serán redirigidos a la partida donde se iniciará el temporizador para el jugador con las piezas blancas asignadas. Para ello, se pasará por el componente *multiplay*er donde se empleará el método "`handleStartGame`". En este método se asignan las variables con los detalles de la partida: ID de la partida, nombre de ambos jugadores y colores asignados con su turno correspondiente. En "`currentplayer`" se almacena el nombre del usuario que tiene el turno actual y seguidamente se configura un temporizador para disminuir de segundo en segundo aquel jugador que disponga del turno.

La siguiente figura muestra un esquema de la búsqueda y creación exitosa de una partida:

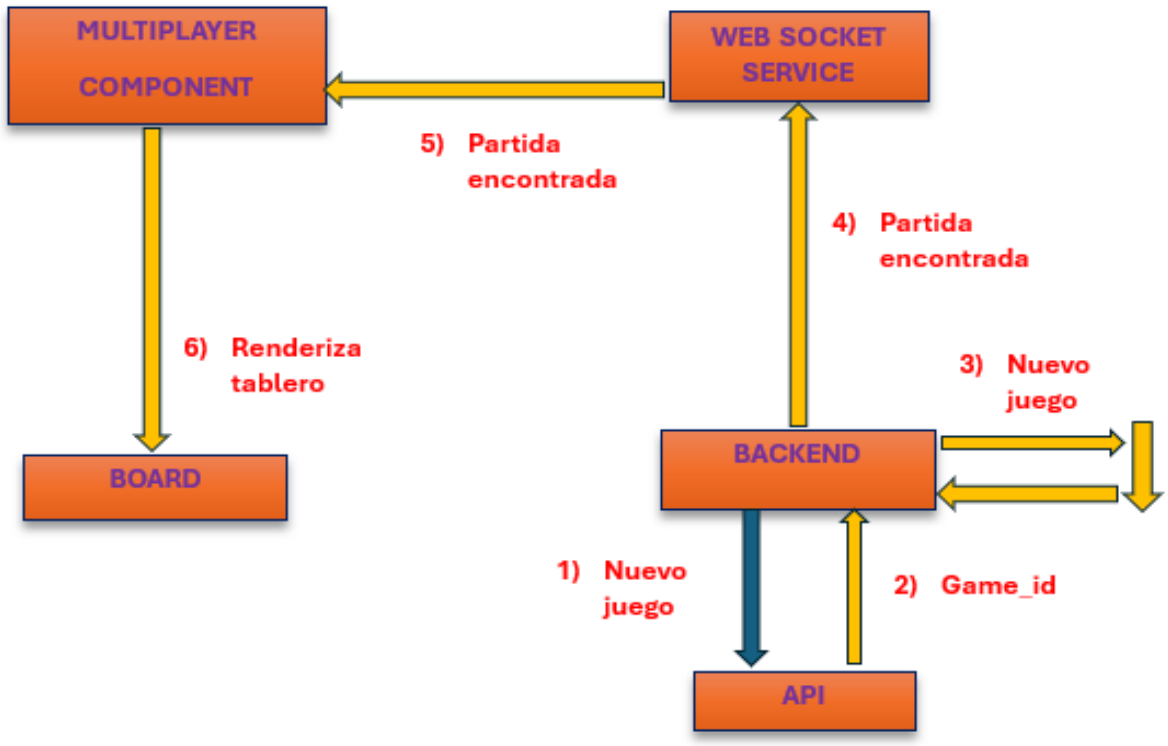


Figura 21. Esquema del multiplayer para crear una partida.

El proceso de clicar una pieza se realiza de forma muy simular al modo Single Player, es decir, el jugador selecciona una pieza desde el "`Board`" utilizando los mismos métodos que se explicaron en el punto 2.2.1. La selección se le comunica al componente padre (*multiplayer*), que tras poner en azul la casilla clicada en el fichero CSS, se vuelve a utilizar un decorador de salida para emitir dicho evento y en ese mismo fichero se desarrollan los métodos "`pieceClicked`" que se activa cuando el jugador clica la pieza de su elección verificando si tiene el turno y el método "`checkPossibleMoves`" que se encargará de solicitar los movimientos posibles a la API a través del "`Game Service`", es decir, el mismo proceso que se realizó en el modo "`Single Player`". Cuando se recibe la respuesta, se actualiza la variable "`possibleMoves`" con el resultado de los movimientos posibles obtenidos, los cuales serán comunicados al "`Board`" a través del decorador de entrada y activándose las casillas de color verde con dicho resultado empleando el fichero CSS del tablero.

Dicha descripción esquemática será representada por la figura 22 de a continuación:

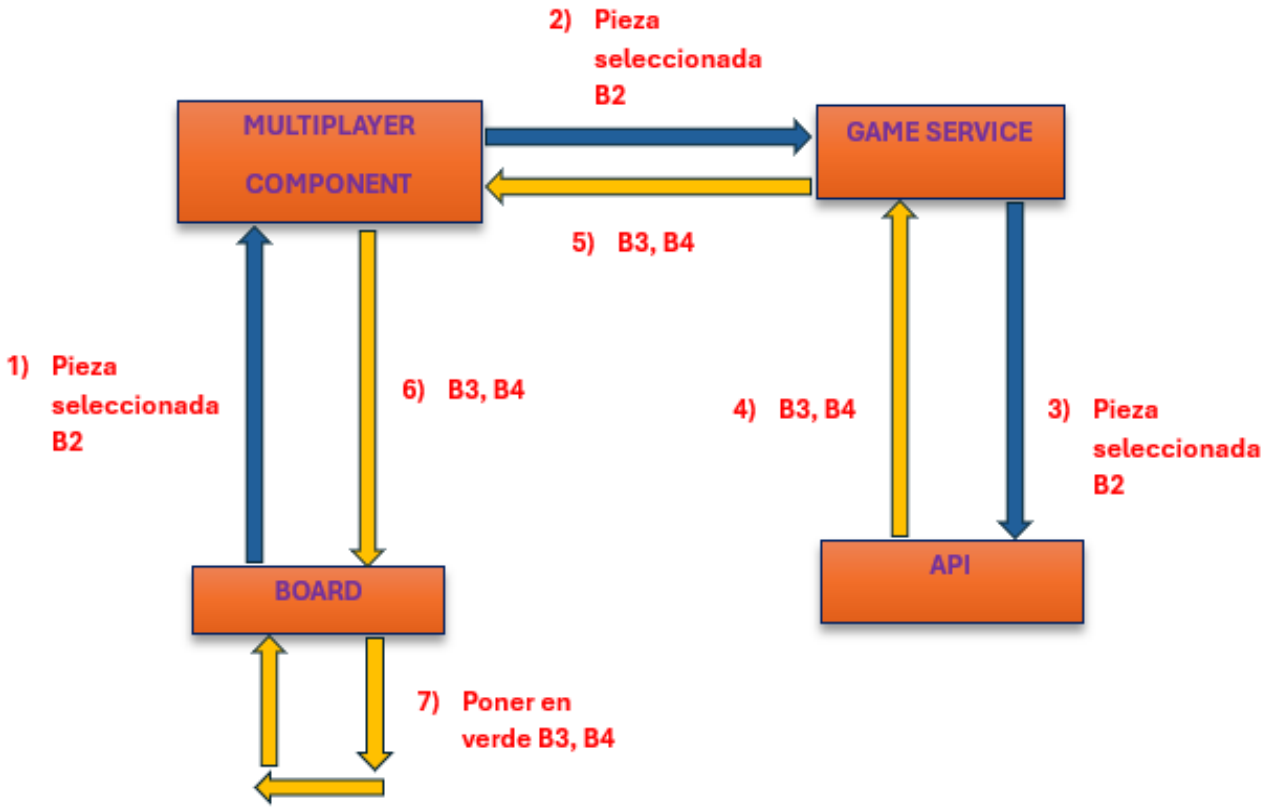


Figura 22. Esquema del multiplayer para seleccionar una pieza.

Después de que el jugador 1 haya obtenido los movimientos posibles, moverá la pieza a la casilla que desee y automáticamente se actualizará la imagen del tablero con ese movimiento. Tras realizar dicho movimiento de la pieza a través del método "`movePieceInBoard`" en el componente "`Board`" el cual, además, especifica las coordenadas de origen y destino de dicha pieza, se emite dicho evento al componente *multiplayer* que a través del método "`movePiece`", verificará el turno del jugador y realizará la petición a la API a través del "`Game Service`" pasándole como parámetros las coordenadas de origen y destino del movimiento realizado y el ID de la partida. Cuando se obtiene la respuesta de la API, se cambiará el turno del jugador al del oponente y se enviará un mensaje a través del WebSocket al servidor para invocar al método "`update`", el cual, actualizará todos los cambios realizados (turno, pieza movida, temporizadores) mandando un mensaje al servidor a través del WebSocket. Además, se empleará el método "`handleMove`" del fichero "`wsController.js`" donde se crean instancias con las coordenadas origen y destino de la pieza movida, el ID de la partida, el jugador 1 y 2 a través de los sockets y seguidamente se llama al método "`switchPlayer`" para cambiar el turnio del jugador. A continuación, se verifica a través de la API, mediante el método "`verifyState`" (definido en la línea 215), el estado de la partida tras realizar cada movimiento y así comprobar si la partida ha terminado o continua. Este método devuelve el estado de la partida haciendo una llamada a la API.

Si el estado de la partida devuelto corresponde al jaque mate, se pasará a la segunda condición donde se comprueba cual es el jugador actual (si el jugador actual es el 1 significa que el jugador 2 hizo el jaque mate debido al "`switchPlayers`" anterior). A continuación, se llaman a los métodos "`incWins`" y "`incLosses`" del fichero "userController.js" los cuales se encargarán de incrementar el número de victorias y derrotas actualizándolo en la base de datos. Finalmente, se manda un mensaje de "`game over`" con el ganador y perdedor y a través del "`if else`" se cierran las conexiones WebSockets y se borra el juego.

En caso de que la partida continúe, se establece el último "`else`", con los temporizadores actualizados.

Por último, el componente padre se encargará de manejar los mensajes enviados desde el "`wsController.js`" y, además, empleará el método "`gameOver`" que se encargará de detener el temporizador, realizar una solicitud a la API para obtener todos los movimientos que se han realizado y abrirá una ventana emergente empleando el "`triggerModal`" que se encarga de activar el modal, el cual imprimirá la lista de movimientos por pantalla en el fichero HTML del componente padre. Para distinguir que movimientos pertenece a cada jugador, se empleará el fichero "`game-over.component.html`" y su correspondiente CSS para intercalar entre color negro y blanco asociado al color de piezas de cada movimiento realizado, es decir, la misma forma explicada en el "`Single Player`".

El resultado de la lista de movimientos en ambos jugadores es el siguiente:

Figura 23. Lista de movimientos multiplayer.

La arquitectura esquemática de este proceso de movimiento de pieza es la siguiente:

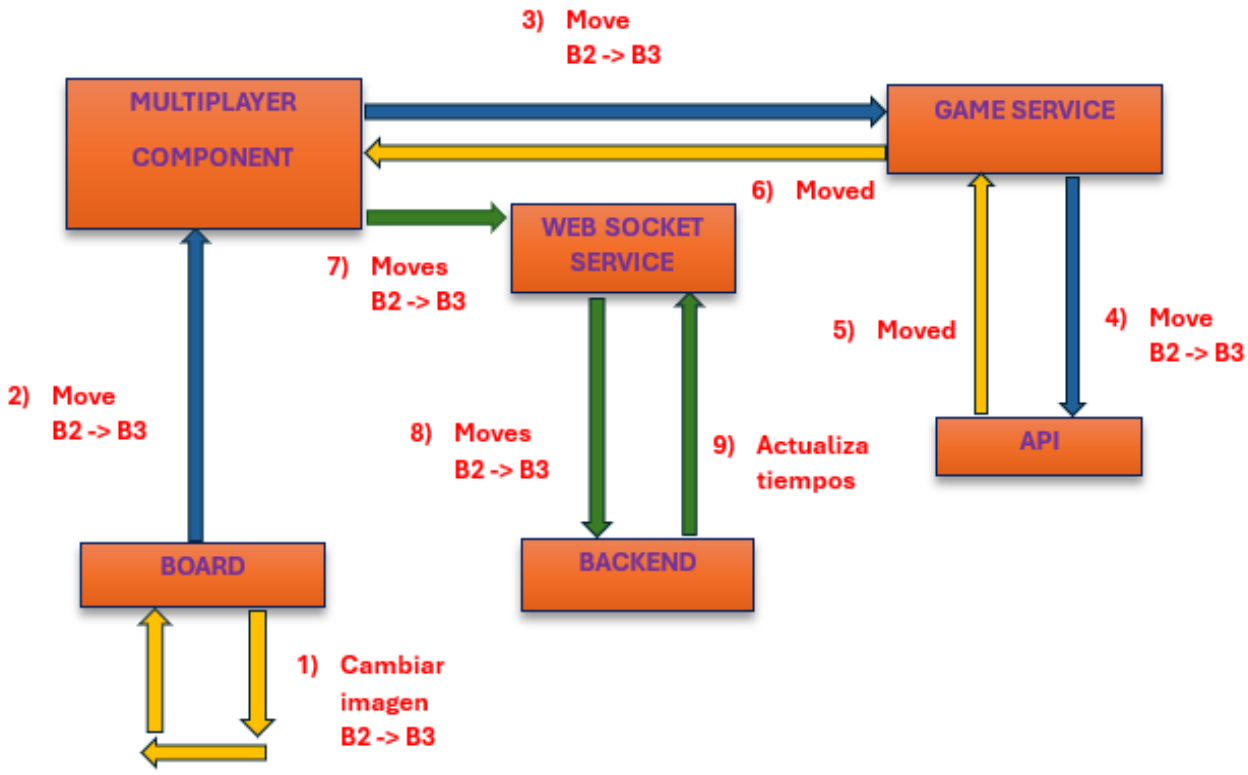


Figura 24. Esquema del multiplayer para mover una pieza.

Desde el punto de vista del oponente, el proceso se realiza de forma inversa, es decir, después de actualizar los datos mencionados en el servidor, se actualizará en el componente padre y se visualizará en el tablero tras el cambio de imagen aplicado:

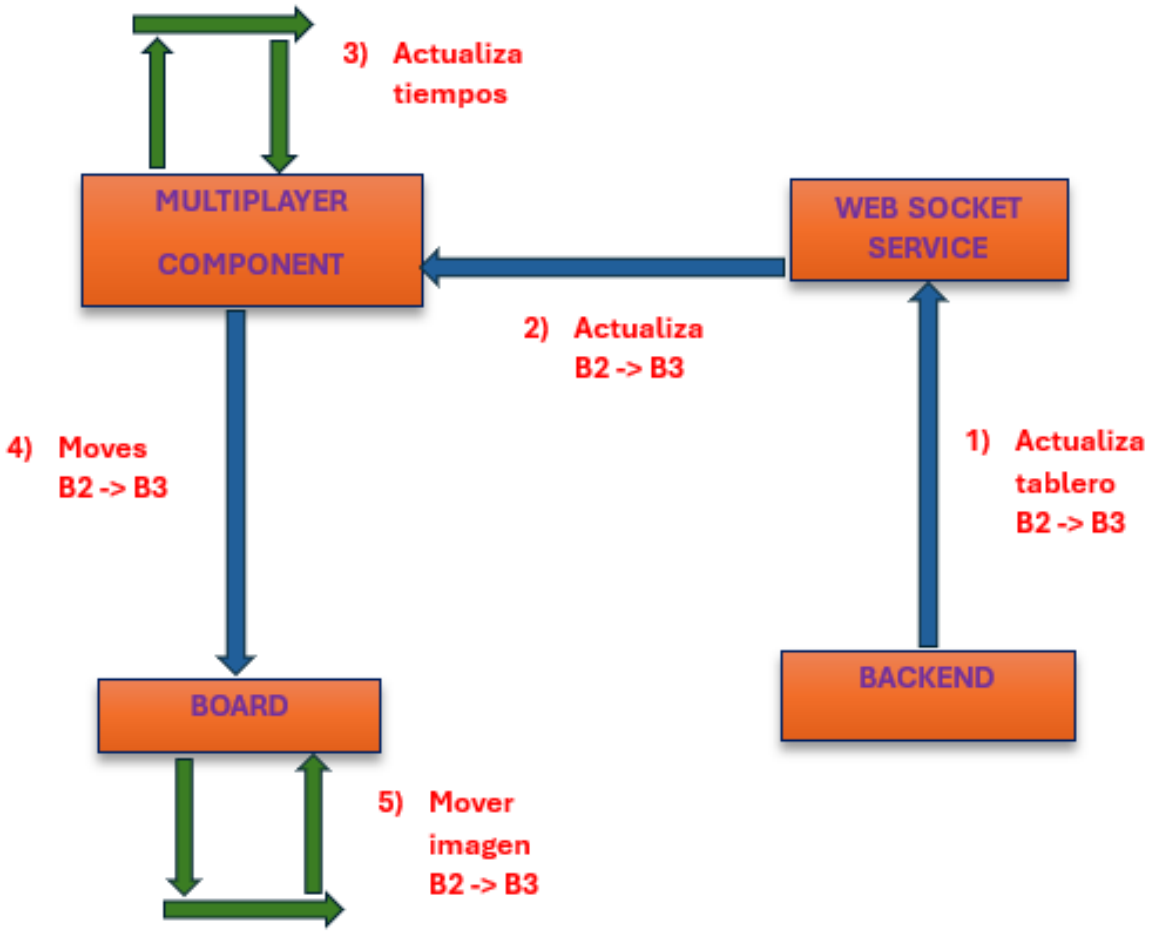


Figura 25. Esquema del multiplayer con el proceso de movimiento desde la perspectiva rival.

Otra de las diferencias e implementaciones que se han realizado en este modo de juego, es el tema de los temporizadores. Se emplean 2 temporizadores en la parte de *frontend* para ser visualizado por ambos jugadores y otro en el servidor, el cuál será el responsable de realizar la cuenta atrás. Cada vez que un jugador realiza un movimiento, recibirá el tiempo actualizado tras la realización del `"update"` en el proceso de movimiento de pieza. Con esta implementación en el servidor se pretende asegurar y garantizar su total funcionamiento evitando problemas de conexiones de red e incluso trampas. Por ejemplo, si el jugador de las piezas blancas (jugador 1) realiza un movimiento, el oponente recibirá el movimiento realizado por el jugador 1 para actualizar el tablero y, además, recibirá los dos temporizadores para que se actualicen en el *frontend*. Por otro

lado, el jugador 1 recibirá como mensaje el temporizador que hay implementado en el servidor y lo actualizará.

El temporizador del servidor se implementa en el fichero "`game.js`", el cual se encargará de llevar la cuenta atrás de los temporizadores y verificará a través de una condición que si el temporizador de un jugador llegase a 0 se emitirá un "`timeout`". Además, debido a que la comunicación es indirecta, una vez alcanzado el "`timeout`" se enviará un mensaje al *frontend* a través del WebSocket y avisará al servidor ("wsController.js"). Cuando alcanza el servidor, ejecutará la función "`handleGameOverDueToTimeout`" que recibirá el juego y sacará el ganador o perdedor en función de que el temporizador esté en 0 para después notificarlo a los jugadores. A continuación, se incrementarán el número de victorias o derrotas del jugador correspondiente, se cerrarán los sockets y se borrará la partida.

Para inicializar los temporizadores, se crea un método constructor en el componente padre, en el que se le asigna a cada ruta de los modos *multiplayer* el tiempo en milisegundos, es decir, 300000 ms (5 minutos) para el modo normal y 60000 ms (1 minuto) para el modo rápido.

La siguiente figura muestra un ejemplo de ambos temporizadores en cuenta atrás después del cambio de turno:

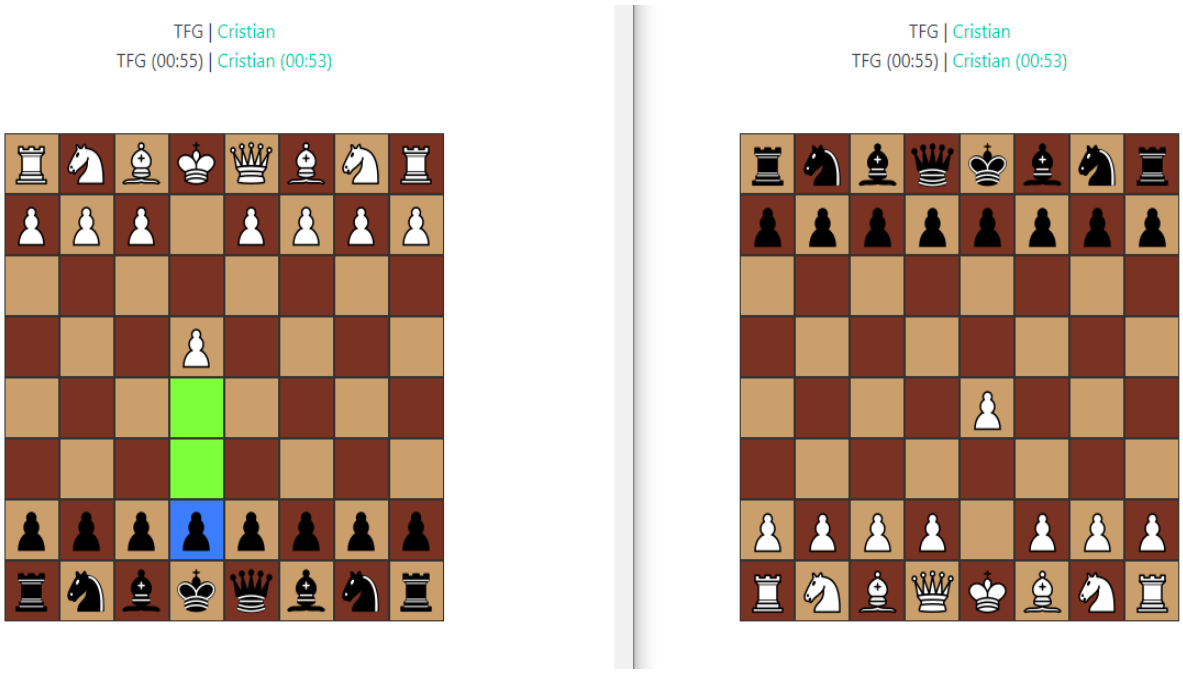


Figura 26. Temporizadores multiplayer en modo Fast Game.

Puede darse la opción, de que un jugador abandone la partida antes de terminarla por cualquier razón. Para cubrir este caso, en el componente padre se realza el método "`ngOnDestroy`" que detectará que jugador ha abandonado la pantalla de la partida cerrando el socket (en el *frontend*) y reseteará los temporizadores.

El método "`closeSocket`" indica que si no existe el ID de partida se permanece en espera ("`exitPool`"), después envía un mensaje indicando el cierre de conexión, elimina el ID de la partida y la resetea. Por último, el usuario es redirigido al menú y se comprueba si hay una suscripción activa para desactivarla, finalmente se recarga la página para garantizar que todo el estado relacionado con el juego se restablezca correctamente.

Por otro lado, en el servidor ("`wsController.js`"), se define el método "`handleClose`" el cual establece como condición de que, si no se encuentra la partida, se eliminará a ese jugador de las dos colas y se cerrará su conexión. Si el jugador 1 se desconecta, se le notificará al jugador 2 y se realizará el incremento de victorias y derrotas al jugador correspondiente. Finalmente, se cerrarán los sockets y se eliminará el juego.

2.3 Manual local

Al principio se seguían estos pasos para el despliegue local durante el desarrollo:

1. Se debe tener un contenedor docker mongoDB corriendo y con el puerto 27107 expuesto.

2. En cada carpeta (*frontend*, *backend*, *chess-api-master*) hacer "`npm install`".

3. Tanto en la carpeta *backend* como chess-api-master hacer "`nodemon index.js`". Nodemon es una utilidad que recarga el "servidor" automáticamente con cada cambio realizado.

4. En la carpeta *frontend* ejecutar "`ng serve –open`".

Actualmente, este enfoque no es válido debido al despliegue del VPS del que se va a hablar en el siguiente punto.

2.4 Servidor privado virtual (VPS)

A continuación, se detallan los pasos seguidos para el correcto despliegue de la aplicación en el servidor privado virtual (VPS). Se han desplegado los siguientes elementos:

- Frontend
- Backend
- API
- Base de datos MongoDB

Cuya estructura esquemática es la siguiente:

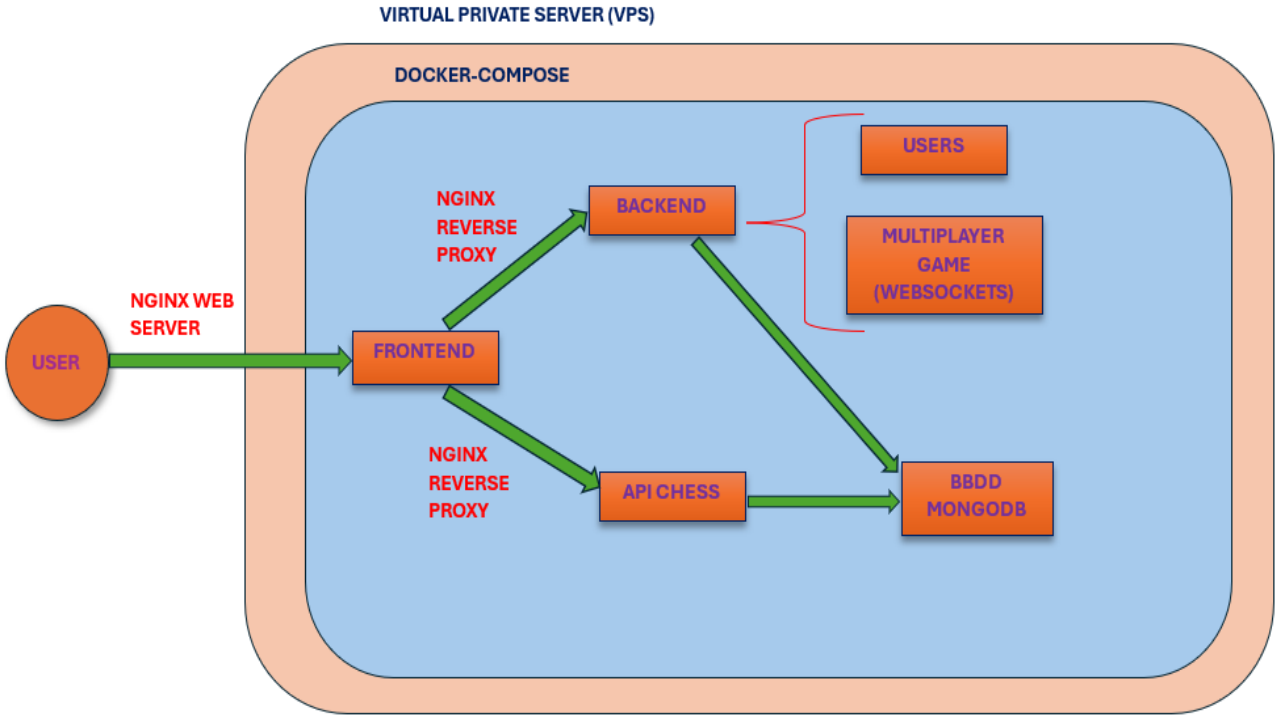


Figura 27. Esquema explicativo VPS.

Se ha utilizado "`docker compose`" para facilitar el despliegue. En el contenedor del *frontend* se ha usado NGINX como proxy inverso para enrutar las solicitudes al *backend* y a la API.

Por razones de seguridad, el único puerto expuesto al host es el 80 (puerto HTTP). Gracias a NGINX, las solicitudes se enrutarán correctamente a los contenedores correspondientes.

A continuación, se muestran los Dockerfiles de cada servicio:

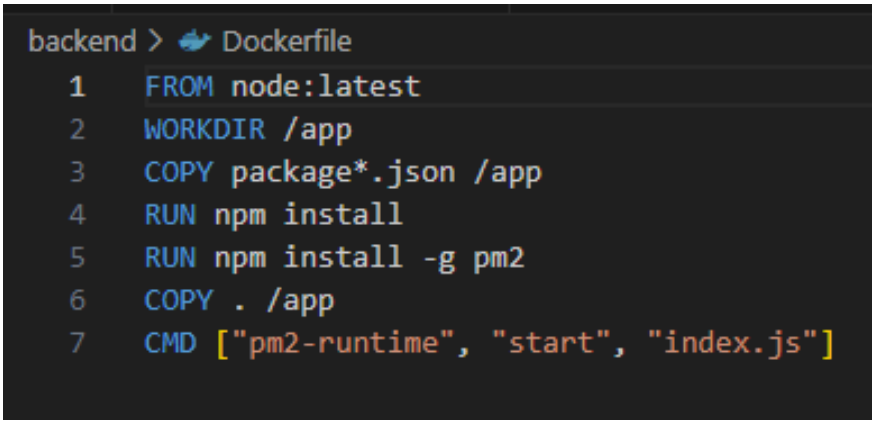


Figura 28. Dockerfile backend.

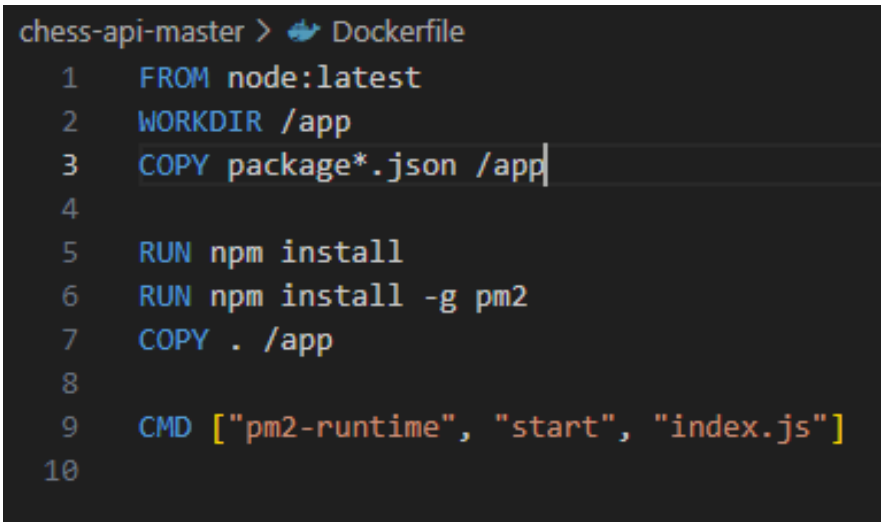


Figura 29. Dockerfile API.

Tanto en el *backend* como en la API se usa pm2 para iniciar el servicio:

frontend > Dockerfile

```
1   FROM node:alpine AS build
2   WORKDIR /dist/src/app
3
4   RUN npm cache clean --force
5
6   COPY . .
7   RUN npm install
8   RUN npm run build --prod
9
10  FROM nginx:alpine AS ngi
11  COPY --from=build /dist/src/app/dist/chess /usr/share/nginx/html/
12  COPY ./nginx.conf /etc/nginx/conf.d/default.conf
13  EXPOSE 80
14
```

Figura 30. Dockerfile frontend.

La construcción de la imagen del *frontend* consta de dos partes:

- 1. Compilación del proyecto usando npm.

- 2. Despliegue del *frontend* usando NGINX.

La configuración de NGINX (`"nginx.conf"`) es la siguiente:

frontend > nginx.conf

```
upstream backend {
    server backend:4000;
}

upstream api {
    server api:3000;
}

server {
    listen 80;
    sendfile on;
    root /usr/share/nginx/html/browser;
    index index.html;

    location / {
        try_files $uri $uri/ /index.html =404;
    }

    location /api/ {
        proxy_pass http://backend;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
    location /api/v1/chess/ {
        proxy_pass http://api;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
    location /ws {
        proxy_pass http://backend/api/ws;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header  Connection "upgrade";
        proxy_set_header Sec-WebSocket-Protocol $http_sec_websocket_protocol;
        proxy_read_timeout 600s;
```

Figura 31. Configuración de NGINX.

A continuación, se explican los detalles de dicho fichero:

- Se establecen dos flujos de salida: *backend* y api.

- Se establecen una serie de reglas en función de la URL, de más específico a más general:

  - Toda dirección acabada en `/api/` será redirigida al *backend*, por ejemplo: "`92.222.177.39/api/login`".

- Toda dirección acabada en `/api/v1/chess/` será redirigida a la API, por ejemplo: "`92.222.177.39/api/v1/chess/one`".

- Toda dirección acabada en `/ws` será redirigda al *backend* como una conexión WebSocket (para el juego multijugador). NGINX cierra automáticamente la conexión a los 60 segundos si no se recibe ninguna respuesta. Con la cláusula "`proxy_read_timeout`" se incrementa este tiempo a 600 segundos. Para el correcto funcionamiento de la conexión WebSocket, se establecen las directrices de actualización del protocolo usado (pues WebSockets son originalmente peticiones HTTP).

- El resto de las direcciones corresponden al *frontend*. Si no existe, resultará en un error 404.

Dichos contenedores se levantan usando "`docker-compose`", para así facilitar el despliegue, reduciéndolo todo a un solo comando (`"docker-compose up"`). A continuación, se detalla el contenido del archivo "`docker-compose.yml`":

Simplemente, se hace mención a los tres servicios principales que hay: *frontend*, *backend* y api. Se crean las imágenes correspondientes a cada servicio, además de descargar y ejecutar la imagen correspondiente a la base de datos. Todos los contenedores se insertan en la misma red virtual de Docker y únicamente el puerto 80 es el expuesto al host. El resto de los servicios permanecen "ocultos", accesibles solo dentro de la red interna.

Adicionalmente, si se desea ejecutar la aplicación a nivel local, habrá que realizar los siguientes pasos:

- Cambiar las IP del fichero "`environment.ts`" (`"frontend/src/environments"`) por "`localhost`".

- Borrar el antiguo contenedor e imágenes del Docker (si las hubiese) y volver a ejecutar el comando "`docker-compose up -d`" en la raíz del proyecto en una terminal.

Una vez realizado estos cambios, se podrá acceder a la aplicación directamente escribiendo "`localhost`" en el navegador.

Capítulo 3. Problemas presentados y soluciones

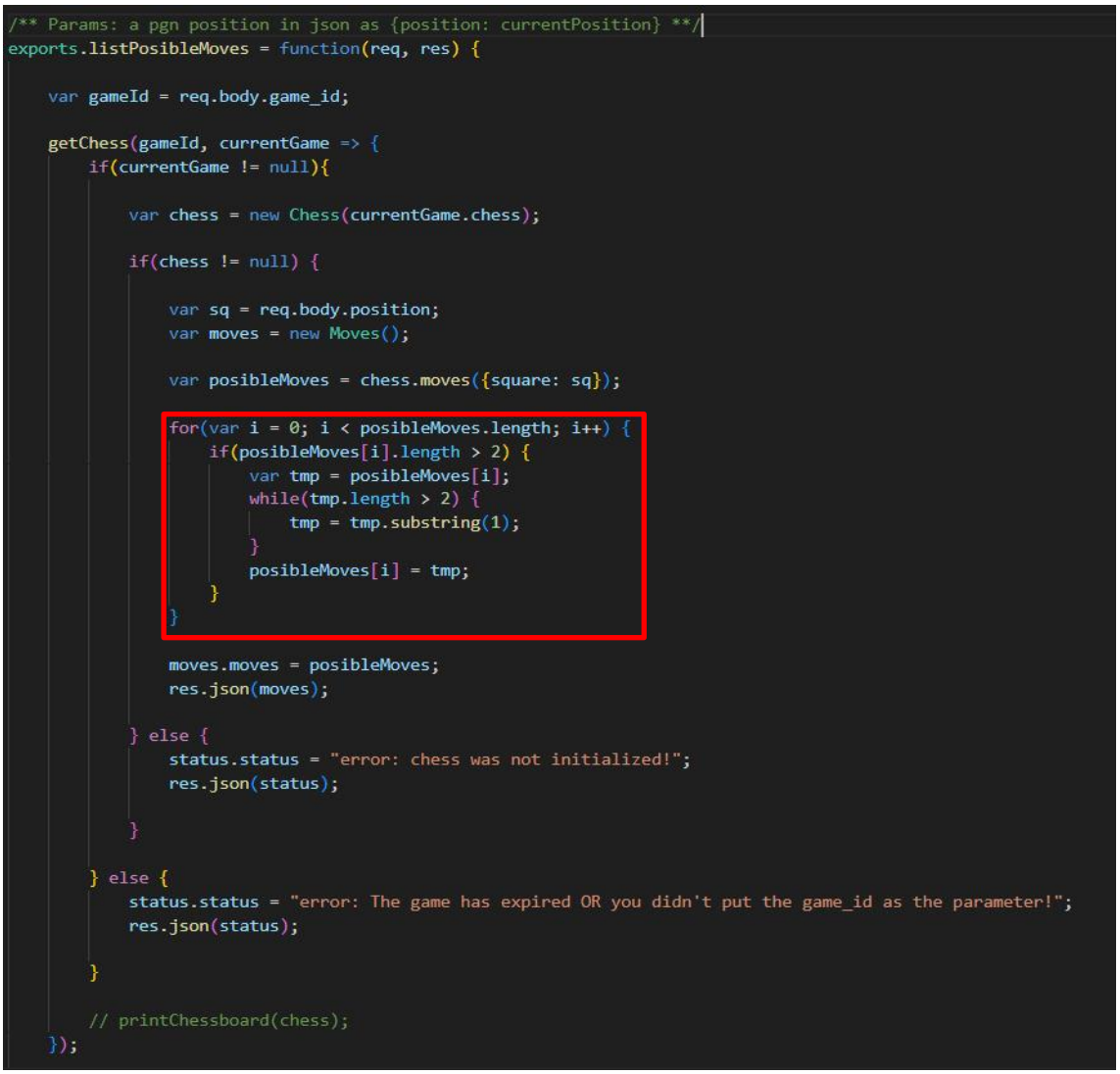
La API utiliza la librería "`ChessBoard.js`", en la parte de movimientos posibles los devuelve en notación algebraica normalizada (SAN). La API hace un intento para procesar esta notación y devolver únicamente la celda, así pues, si por ejemplo un movimiento posible como Qxf2, se debería devolver f2 únicamente.

El procesamiento de la API falla al ser muy primitivo, pues solo va recortando la cadena de texto (extrayendo el prefijo siempre) hasta que tenga una longitud de 2. Así pues, en el caso de Qf7+, que indica que la reina en f7 provocaría un jaque, el procesamiento de la API eliminaría la Q y la f, dejando el 7+ siendo una celda totalmente inválida.

Para solucionar este problema, se implementó un cambio significativo en la lógica de procesamiento de la API. Se sustituyó el método primitivo por una expresión regular (`"[a-h][1-8]"`), que permite extraer correctamente las coordenadas de las celdas según la notación SAN utilizada por "`Chessboard.js`". Esta mejora garantiza que las coordenadas extraídas sean precisas y válidas.

Este cambio ha sido aplicado de manera consistente en todos los controladores de la API relacionados con el manejo de movimientos de ajedrez, asegurando así una funcionalidad coherente y correcta en toda la aplicación.

A continuación, se mostrará como se ha realizado dicho cambio en el código de la función "`listPossibleMoves`" en la figura 32:



```
/** Params: a pgn position in json as {position: currentPosition} **/
exports.listPosibleMoves = function(req, res) {

    var gameId = req.body.game_id;

    getChess(gameId, currentGame => {
        if(currentGame != null){

            var chess = new Chess(currentGame.chess);

            if(chess != null) {

                var sq = req.body.position;
                var moves = new Moves();

                var posibleMoves = chess.moves({square: sq});

                for(var i = 0; i < posibleMoves.length; i++) {
                    if(posibleMoves[i].length > 2) {
                        var tmp = posibleMoves[i];
                        while(tmp.length > 2) {
                            tmp = tmp.substring(1);
                        }
                        posibleMoves[i] = tmp;
                    }
                }

                moves.moves = posibleMoves;
                res.json(moves);

            } else {
                status.status = "error: chess was not initialized!";
                res.json(status);

            }

        } else {
            status.status = "error: The game has expired OR you didn't put the game_id as the parameter!";
            res.json(status);

        }

        // printChessboard(chess);
    });
```

Figura 32. Código que se va a modificar.

En la figura 33, la parte enmarcada en rojo es la parte que va a sufrir dichas modificaciones comentadas con anterioridad siendo la enmarcación de la figura 2 el resultado de dichos cambios:

```
/** Params: a pgn position in json as {position: currentPosition} **/
exports.listPosibleMoves = function(req, res) {

    var gameId = req.body.game_id;

    getChess(gameId, currentGame => {
        if(currentGame != null){

            var chess = new Chess(currentGame.chess);

            if(chess != null) {

                var sq = req.body.position;
                var moves = new Moves();

                var posibleMoves = chess.moves({square: sq, verbose: false});
                let legalMoves = [];
                let reg = new RegExp(/[a-h][1-8]/);
                for(var i = 0; i < posibleMoves.length; i++) {
                    let match = posibleMoves[i].match(reg);
                    if (match != null) {
                       legalMoves.push(match[0]);
                    }
                }

                moves.moves = legalMoves;
                res.json(moves);

            } else {
                status.status = "error: chess was not initialized!";
                res.json(status);

            }

        } else {
            status.status = "error: The game has expired OR you didn't put the game_id as the parameter!";
            res.json(status);

        }

        // printChessboard(chess);
    });
```

Figura 33. Código modificado.

Los ficheros que se han visto afectados son: `"api/controllers/chessOnePlayerController.js"` y el análogo `"chessTwoPlayerController.js"`.

Otro problema presentado en la primera fase del desarrollo era que la API bloqueaba las peticiones del *frontend* debido a que utilizaba la política CORS. Para solucionarlo, se modificó el fichero index.js añadiendo las líneas de código siguientes: `"const cors = require('cors');"` `"app.use(cors());"` permitiendo así que el servidor responda adecuadamente a las peticiones que provienen de otros dominios, autorizando de esta forma el intercambio de recursos entre diferentes orígenes de manera segura.

Adicionalmente, se empleaba "`Date.now()`" que representaba una constante, no una función. Dicha constante representaba el momento en el que se levantaba la API que contaba con un tiempo de expiración establecido de 2 horas. Esto ocasionaba que por ejemplo al entrar al juego, en cualquier momento inferior a esas dos horas establecidas, la sesión de juego caducase por ese tiempo de expiración. Para solucionarlo, en el fichero "`api/models/chessboardModel.js`" línea de código 94, se había establecido unos paréntesis vacíos en una constante, por lo que dichos paréntesis fueron eliminados permitiendo así que el tiempo de expiración se inicie solamente cada vez que se entra en el juego. Quedando la parte de código solucionado de la siguiente manera:

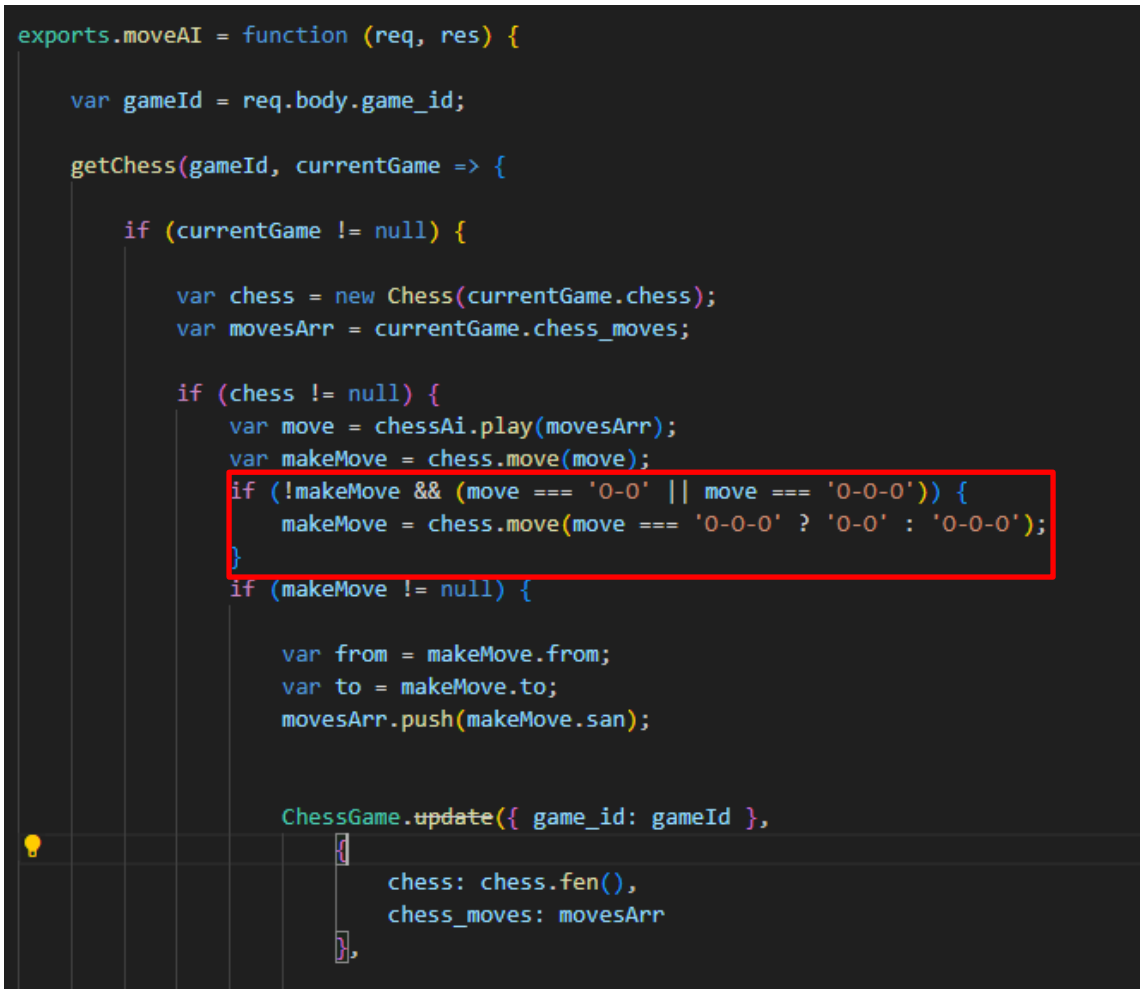
```
createAt: {
    type: Date,
    default: Date.now,
    index: { expires: '2h' },
    expireAfterSeconds: 7200
}
```

Figura 34. Solución al problema de expiración.

El siguiente problema es que la base de datos estaba retrasada 2 horas respecto a la hora española. Para solucionarlo, se incorporó la variable de entorno "`TZ = Europe/Madrid`" en la base de datos de la API para que cuando se inserten los datos, haga correctamente la conversión de la zona horaria española a la UTC (internacional).

El último problema se debe al concepto del enroque mencionado en las líneas futuras, pero por parte de la IA. En la carpeta de la API, hay un método llamado "`moveAI`" situado en el fichero "`chessOnePlayerController.js`"("`chess-api-master/api/controllers`"), el cuál devolvía un valor `null` cada vez que la máquina intentaba enrocarse, por lo que provocaba un manejo erróneo de la partida ya que la librería que usaba *chess* no lo soportaba.

Para solucionarlo, se realizó una búsqueda entre los "`issues`" del GitHub y se utilizó un fork que soportaba el enroque. Además, se modificó el método "`moveAI`" añadiendo la siguiente condición:



```
exports.moveAI = function (req, res) {

    var gameId = req.body.game_id;

    getChess(gameId, currentGame => {

        if (currentGame != null) {

            var chess = new Chess(currentGame.chess);
            var movesArr = currentGame.chess_moves;

            if (chess != null) {
                var move = chessAi.play(movesArr);
                var makeMove = chess.move(move);
                if (!makeMove && (move === 'O-O' || move === 'O-O-O')) {
                    makeMove = chess.move(move === 'O-O-O' ? 'O-O' : 'O-O-O');
                }
                if (makeMove != null) {

                    var from = makeMove.from;
                    var to = makeMove.to;
                    movesArr.push(makeMove.san);


                    ChessGame.update({ game_id: gameId },
                        {
                            chess: chess.fen(),
                            chess_moves: movesArr
                        },
```

Figura 35. Código API de enroque implementado.

Esta condición valida si el movimiento de la IA es enroque, si es así lo realiza y si es `null` hará el otro enroque. Es decir, si el enroque fallido es el corto (O-O) hará el enroque largo (O-O-O) y viceversa.

Capítulo 4. Conclusiones y líneas futuras

Con esta aplicación web, cada usuario podrá disfrutar de una gran experiencia tecnológica sobre este famoso juego de mesa y a la vez deporte. Cualquier jugador que desee practicar o divertirse en cualquier momento podrá hacerlo gracias al modo de juego Single Player, el cual le será esencialmente útil en base a su formación y entretenimiento. Además, podrá gozar de una experiencia contra jugadores reales a través del establecimiento de conexión por medio de WebSockets y permitiendo así jugar desde cualquier punto del país sin necesidad de estar situados en la misma red local. Gracias a los distintos modos de juego *multiplayer*, el usuario podrá combinar su tiempo libre en un modo rápido de juego o tomarse más tiempo que le permita pensar y jugar con mayor claridad por medio de la implantación de temporizadores de 5 minutos o 1 minuto dependiendo del modo seleccionado.

Por otro lado, cada jugador podrá visualizar su número de victorias y derrotas acumuladas e incluso sus datos personales una vez que se identificó en la aplicación web.

La implementación de la API fue fundamental para la realización de este proyecto y poder así desarrollar estos modos de juego e incluso los movimientos de cada pieza del tablero por medio de endpoints, consiguiendo una aplicación mucho más dinámica y eficaz gracias a la utilización de estas herramientas.

Como líneas futuras, se pueden considerar una serie de mejoras e implementaciones que permitan mejorar este proyecto en un futuro y hacer que sea una de las mejores aplicaciones web de ajedrez que se hayan visto.

Una de esta implementación serie la utilización de un chat para que ambos jugadores puedan comentar sus mejores jugadas o hablar sobre cualquier tema, e incluso analizar sus mejores movimientos. Otra implementación sería el movimiento de enroque por parte del usuario. Este movimiento les será muy útil para proteger el rey de forma más rápida, aunque debido a que esta API no lo implementa se podrá realizar de forma más compleja. Además, de la coronación de peones en el modo multijugador y en el modo Single Player para el usuario ya que la API únicamente facilitó estas dos implementaciones para la IA, pero no para el jugador real.

En cuanto a mejoras, se podrá desarrollar una IA con distintas dificultades para aquellas personas que sean novatas y quieran aprender de forma más sencilla hasta dificultades altas para los grandes expertos y maestros del ajedrez. Además, de poder analizar todos los movimientos realizados una vez que se acabe la partida, aconsejando sobre los errores cometidos y posibles soluciones ayudando a mejorar el rendimiento del usuario.

Finalmente, otra implementación futura sería un modo de juego basado en torneos, donde de forma más cómoda un grupo de usuarios podrá competir por el primer puesto y demostrar quién es el mejor, con el fin de, en un futuro, llevarlo a cabo a través de los servidores que la empresa proporcione.

En cuanto al cumplimiento de la ODS se destacan los siguientes aspectos:

1. **ODS 4. Educación de calidad:**

- Desarrollo de habilidades cognitivas como la memoria, concentración y el pensamiento crítico.
- La aplicación puede ser utilizada como acceso a la educación.

2. **ODS 8. Trabajo decente y crecimiento económico:**

- Fomento del emprendimiento y la innovación inspirando a otros a desarrollar soluciones innovadoras.

3. **ODS 9. Industria, innovación e infraestructura:**

- Innovación tecnológica desarrollando una plataforma de ajedrez en línea.
- Acceso a la información y las comunicaciones permitiendo que más personas participen en el mundo digital.

4. **ODS 10. Reducción de las desigualdades:**

- Se proporciona una plataforma gratuita y accesible para jugar al ajedrez.
- Fomenta la inclusión social permitiendo que personas de diversas regiones participen en una actividad común.

5. **ODS 16. Paz, justicia e instituciones sólidas:**

- El ajedrez enseña habilidades como la paciencia, estrategia y resolución de problemas.
- El juego promueve valores como la integridad, la equidad y el respecto por las reglas.

Capítulo 5. Referencias bibliográficas

- API utilizada para el proyecto: https://github.com/anzemur/chess-api
- ¿Qué es una API?: https://aws.amazon.com/es/what-is/api/
- ¿Cómo funciona una API?: https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces#:~:text=Una%20API%20o%20interfaz%20de,el%20software%20de%20las%20aplicaciones
- Documentación de la API utilizada: https://documenter.getpostman.com/view/1741165/chess-api/7Lof2bk#intro
- Diseño de piezas de ajedrez: https://commons.wikimedia.org/wiki/Category:SVG_chess_pieces
- Documentación Angular: https://docs.angular.lat/guide/displaying-data
- Documentación Node.js: https://nodejs.org/docs/latest/api/documentation.html
- Documentación del uso de token (JWT): https://jwt.io/libraries
- Consejos utilizados: https://www.chess.com/article/view/how-to-win-a-chess-game
- Como jugar al ajedrez: https://www.chess.com/es/como-jugar-ajedrez
- Observables en Angular: https://docs.angular.lat/guide/observables-in-angular
- Uso de WebSockets: https://es.javascript.info/websocket
- Fork de enroque para la IA: https://github.com/sesse/chess.js?tab=BSD-2-Clause-1-ov-file

- Librería express-ws: https://github.com/HenningM/express-ws
- Librería nodemon: https://github.com/remy/nodemon
- Manual Docker: https://docs.docker.com/manuals/
- Proyecto de la aplicación web del ajedrez en la máquina virtual: http://92.222.177.39/
- Proyecto final en Github: https://github.com/CristianCr02/Proyecto_TFG_Ajedrez