

# Desarrollo en Entorno Servidor

Segunda Práctica: Maze



**Nombre:** Cristina Picón Serrano  
**Curso:** 2<sup>a</sup> DAW  
**Colegio:** Es Liceu  
**Fecha:** 07/12/24

## **Índice alfabético**

INTRODUCCIÓN.....	3
1.MANUAL DEL USUARIO.....	4
2 ESTRUCTURA DEL PROGRAMA: SERVIDOR.....	11
2.1 Modelos (models).....	11
2.2 DAO (Data Access Objects).....	14
2.3. Excepciones (exceptions).....	16
2.4 Servicios (servers).....	17
2.5 Controladores (controllers).....	21
3. ESTRUCTURA DEL PROGRAMA: FRONT-END.....	33
4. EXPERIENCIA EN LA PRÁCTICA.....	42

## **INTRODUCCIÓN**

Este proyecto consiste en el desarrollo de una aplicación web para implementar un juego de laberinto interactivo. El objetivo principal es que el jugador pueda moverse por las diferentes habitaciones, recoger monedas, y obtener llaves para abrir las puertas necesarias hasta llegar a la última habitación que es la salida en el menor tiempo posible. Para garantizar una experiencia completa, la aplicación incluye funciones como el registro y la autenticación de usuarios, la selección de distintos mapas para jugar, el seguimiento de las partidas de los jugadores y la posibilidad de reiniciar el juego en cualquier momento.

La aplicación implementa una arquitectura modular utilizando el patrón Modelo-Vista-Controlador (MVC). En el backend se utiliza Java con Spring Framework, mientras que el frontend está construido con HTML, CSS y JavaScript. Además se implementa el uso de una base de datos para la persistencia de información como usuarios, mapas, progreso del jugador y resultados. Para optimizar la gestión de dependencias y simplificar el proceso de construcción del proyecto, se ha implementado Maven lo que ha facilitado la organización del código y la integración de bibliotecas externas, como Gson, de una forma más fácil y eficiente

Por último en este trabajo se incluye un manual de uso de la aplicación dirigido a los usuarios y también se explicarán los aspectos técnicos tanto del backend como del frontend.

# **1 1. MANUAL DEL USUARIO: MAZE**

## **Inicio de sesión** ("login")

Selecciona esta opción si ya tienes una cuenta registrada.

The image consists of two side-by-side screenshots of a mobile application's login screen. Both screens have a dark background with white text and light gray input fields. The left screen shows the initial state of the form with empty input fields and standard UI elements. The right screen shows the same form after an attempt, with a pink rectangular box at the bottom containing the text "Usuario i/o contraseña incorrectos" (User or password incorrect) in white. This visual cue indicates that the credentials entered were invalid.

Para acceder a tu cuenta en la aplicación, sigue estos pasos:

### **1. Ingresá tus Credenciales:**

- **Nombre de usuario:** Introduce tu nombre de usuario asociado a tu cuenta.
- **Contraseña:** Escribe tu contraseña en el campo correspondiente.

### **2. Haz clic en "Enviar":** Despues de ingresar tus datos, selecciona el botón "Enviar" para acceder a tu cuenta.

### **3. Si quierés borrar los datos introducidos porque te has equivocado, puedes darle al botón "Borrar".**

### **4. Problemas de Acceso:**

- Asegúrate de que tus credenciales sean correctas. De no ser así te saldrá un mensaje de error como se observa en la imagen.
- Si sigues teniendo problemas, verifica tu conexión a internet o intenta nuevamente más tarde.

## Registro en la aplicación ("register")

Elije esta opción si es tu primera vez en la aplicación y deseas crear una nueva cuenta.

The image contains two side-by-side screenshots of a mobile application's registration screen. Both screens have a dark background with white text and light gray input fields. The top bar on both says "Regístrate". Below it, a instruction "Completa los siguientes campos" is displayed. The first screenshot shows three input fields: "Nombre Completo", "Nombre Usuario", and "Contraseña", each with a placeholder text and a red border indicating they are required. At the bottom are two buttons: "Enviar" and "Borrar". A yellow link "¿Ya tienes cuenta? Login" is located below the first input field. The second screenshot shows the same layout but with a red error box at the bottom containing the text "El nombre es demasiado corto, tiene que tener 6 caracteres." This indicates that the user has attempted to register with a name that is too short.

Para crear una nueva cuenta en la aplicación, sigue estos pasos y asegúrate de cumplir con las condiciones requeridas:

### **1. Completa el Formulario de Registro:**

- **Nombre Completo:**
  - Debe tener un mínimo de 6 caracteres.
- **Username:** Verifica que el nombre de usuario no esté siendo utilizado por otra cuenta. Si el nombre de usuario ya existe, se te pedirá que elijas uno diferente.
- **Contraseña:**
  - Debe tener al menos 5 caracteres.
  - Asegúrate de que sea segura y no fácil de adivinar. Es recomendable usar una combinación de letras, números y símbolos.

### **2. Finaliza el Registro:** Haz clic en el botón "**Registrarse**" para enviar tu información.

En el caso de que no cumplas alguno de los requisitos saldrá un mensaje de error.

Una vez completados estos pasos, podrás iniciar sesión en la aplicación con tu nuevo nombre de usuario y contraseña. Asegúrate de recordar tus credenciales para futuros accesos.

## Empezar a jugar

Para poder acceder al juego debes **Iniciar sesión**.

Una vez que hayas iniciado sesión, serás dirigido a la pantalla principal donde habrá un desplegable para elegir un modo de juego:

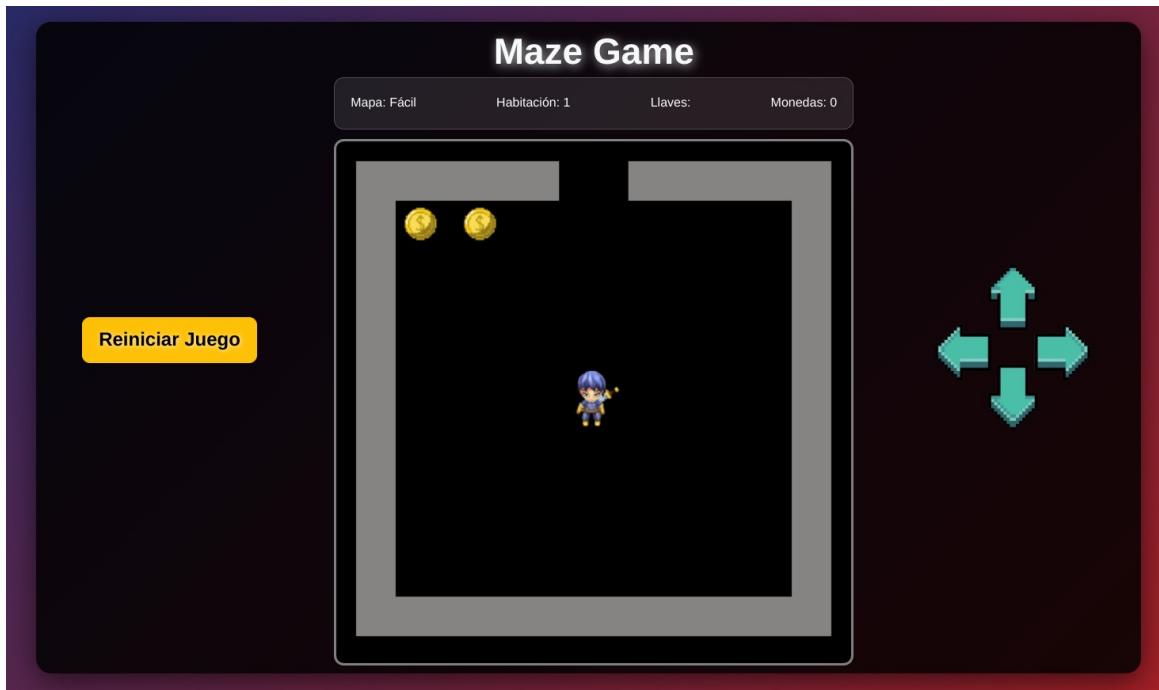


- 1. Fácil:** Es el nivel más sencillo, está compuesto por tres habitaciones y sirve para familiarizarse con el juego, aprender las mecánicas básicas y practicar la navegación a través del laberinto
- 2. Medio:** Un nivel más complicado. Está diseñado para jugadores que ya comprenden las mecánicas básicas y buscan una experiencia más exigente.
- 3. Difícil:** El nivel más avanzado, cuenta con un laberinto extenso, múltiples habitaciones y una mayor cantidad de puertas, monedas y desafíos

Una vez hayas escogido el modo de juego debes dar a **Enviar** para cargar el laberinto.

# Pantalla principal del juego

En la pantalla principal del juego se muestra el laberinto, tus recursos, una opción de controles para moverte y avanzar y una opción para reiniciar el juego."



## 1. Barra de Información Superior

- **Mapa:** Indica el nivel de dificultad seleccionado (por ejemplo, "Fácil").
- **Habitación:** Muestra el número de la habitación en la que te encuentras actualmente.
- **Llaves:** Muestra el nombre de las llaves que has recogido.
- **Monedas:** Indica la cantidad de monedas que tienes disponibles.

## 2. Laberinto

- Representa la habitación en la que estás situado.
- La habitación contiene puertas (espacio entre las paredes grises). En el caso de que estas puertas estén abiertas se verán negras (como en el ejemplo). En el caso de que la puerta esté cerrada se verá de color rojo.
- En el centro de la habitación se encuentra el personaje que podrá moverse en las cuatro direcciones (vease más adelante "controles de navegación").
- Además del personaje, en el interior de la habitación podemos encontrar llaves o monedas. Objetos necesarios para poder llegar al final del juego.

## 3. Controles de Navegación

El jugador tiene la capacidad de moverse en cuatro direcciones: Norte, Sur, Este y Oeste. Para realizar estos movimientos, dispone de dos opciones principales: utilizar las flechas direccionales mostradas en la pantalla o los botones de navegación en el teclado de su dispositivo.

- **Flechas en pantalla:**

En el lado derecho de la interfaz principal, se encuentran flechas que representan los movimientos disponibles:

- **Arriba (↑):** Moverse hacia el norte.
- **Abajo (↓):** Moverse hacia el sur.
- **Izquierda (←):** Moverse hacia el oeste.
- **Derecha (→):** Moverse hacia el este.

Para desplazarte, simplemente haz clic en la flecha correspondiente a la dirección deseada.

- **Teclado:**

Alternativamente, puedes usar las teclas de flechas de tu teclado para desplazarte:

- **Tecla de flecha arriba (↑):** Moverse hacia el norte.
- **Tecla de flecha abajo (↓):** Moverse hacia el sur.
- **Tecla de flecha izquierda (←):** Moverse hacia el oeste.
- **Tecla de flecha derecha (→):** Moverse hacia el este.

## **Reglas de Movimiento**

- Si existe una **puerta abierta** en la dirección seleccionada, el jugador avanzará a la siguiente habitación sin inconvenientes.
- Si la puerta está **cerrada**, el juego mostrará un mensaje de error indicando que no puedes pasar por esa dirección. En este caso, el jugador permanecerá en el centro de la habitación actual.



- Si no hay una puerta en la dirección elegida (es decir, hay una pared sólida), el juego mostrará un mensaje de error similar, y el jugador tampoco podrá avanzar.



#### **4. Botón "Reiniciar Juego"**

- Ubicado a la izquierda, este botón permite reiniciar la partida desde el inicio, eliminando todo tu progreso actual. Además de reiniciar la partida, el botón también cerrará tu sesión, redirigiéndote automáticamente a la pantalla de inicio de sesión.

### **Dinámica del juego**

El objetivo principal del juego es escapar del laberinto moviéndote entre las diferentes habitaciones. Para lograrlo, deberás recolectar las llaves necesarias para abrir las puertas cerradas que bloquean tu camino. Sin embargo, obtener estas llaves no será gratis: necesitarás reunir suficientes monedas para comprarlas. Cada vez que consigas una llave, se descontará automáticamente el costo correspondiente en monedas de tu inventario.

#### **Recoger Monedas**

- Dentro de ciertas habitaciones hay monedas que podrás recolectar haciendo clic sobre ellas.
- Las monedas son necesarias para adquirir las llaves que permiten abrir las puertas cerradas.

*Si no hay monedas en la habitación y el jugador intenta recogerlas, se mostrará un mensaje de error.*

#### **Obtener Llaves**

- Algunas habitaciones contienen llaves que puedes recoger haciendo clic sobre ellas.
- Cada llave tiene un coste en monedas, por lo que debes asegurarte de haber recolectado suficientes monedas antes de intentar recoger una llave.
- Una vez que el jugador recoge una llave, esta se mostrará en el inventario, junto con su nombre.
- Si haces click en una llave y no tienes monedas suficientes se te mostrará un mensaje de error.

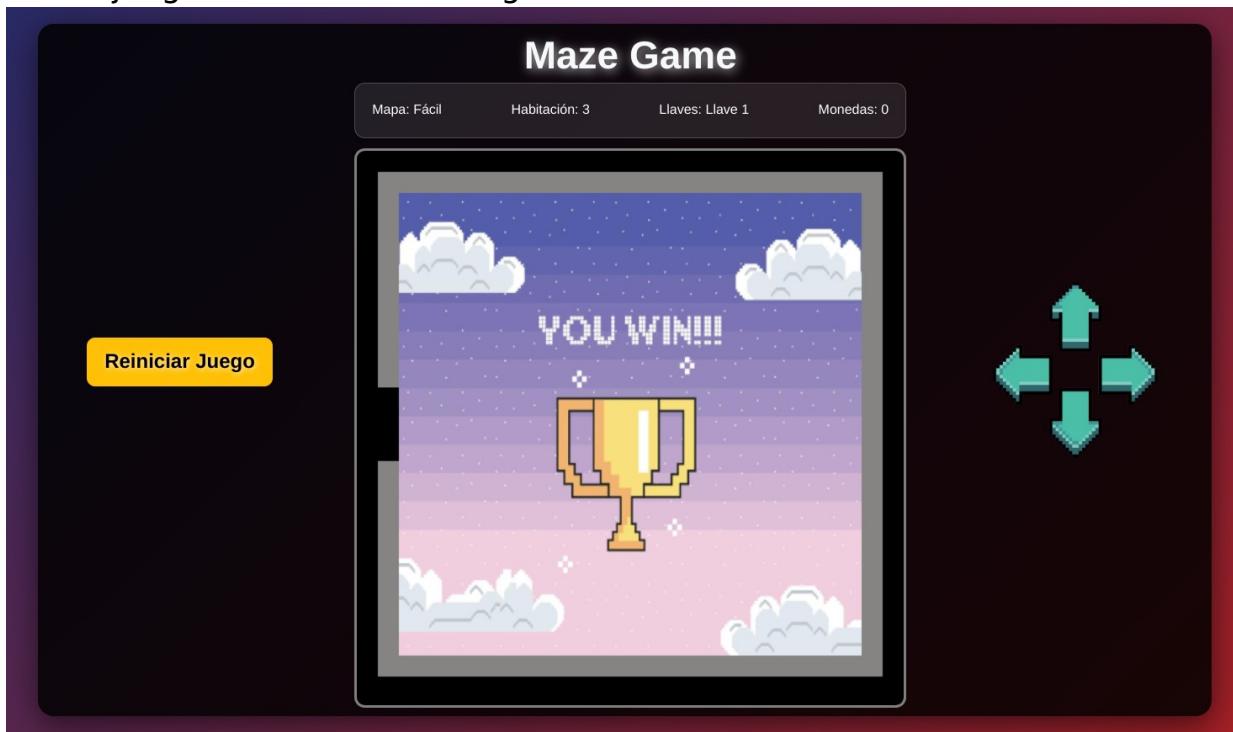
*Si no hay llaves en la habitación y el jugador intenta recogerlas, se mostrará un mensaje de error.*

#### **Abrir Puertas**

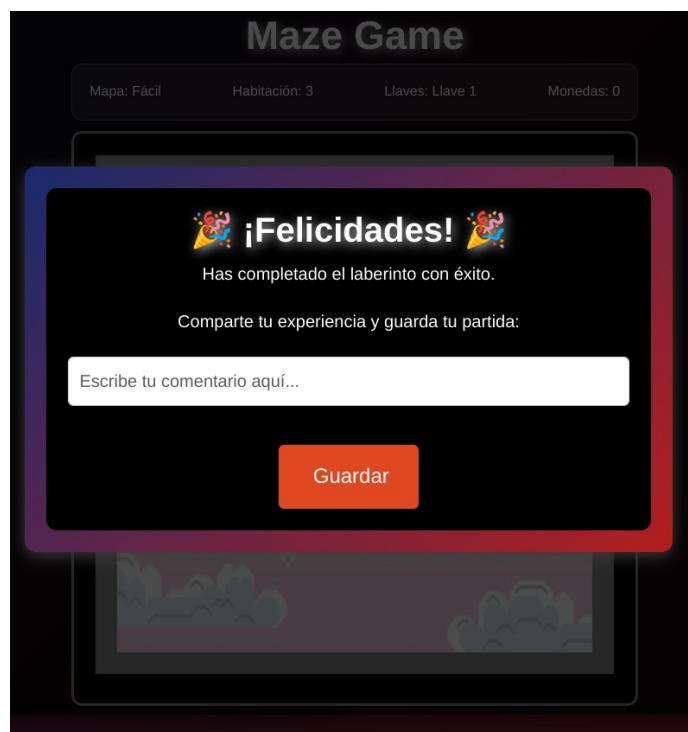
- Para abrir una puerta cerrada, debes poseer la llave correspondiente en tu inventario.
- Al hacer clic sobre una puerta cerrada, si tienes la llave correcta, la puerta se abrirá y podrás avanzar.
- Si no tienes la llave adecuada, el juego mostrará un mensaje de error indicando que no es posible abrir la puerta.

## Final del Juego

- El juego finaliza cuando llegas a la habitación marcada como final.



- Para parar el cronómetro de la partida debes hacer click en la habitación.
- Al hacer click se abrirá una ventana emergente donde tendrás la opción de añadir un comentario sobre tu experiencia en la partida, si así lo deseas.



- Finalmente (al darle a guardar), serás redirigido a una pantalla que muestra los resultados y el ranking de jugadores que han completado el juego.



The screenshot shows a dark-themed user interface for a game session summary. At the top, there are two yellow rectangular buttons: "Volver a Jugar" on the left and "Cerrar Sesión" on the right. Below these buttons is a horizontal bar with a dark blue gradient. The main content area features a table with a dark header row containing the columns: "Nombre de Usuario", "Nombre del Mapa", "Tiempo de Juego", and "Comentarios". The table body contains six rows of data, each representing a player's performance:

Nombre de Usuario	Nombre del Mapa	Tiempo de Juego	Comentarios
bill gates	Mapa 1	00:00:18	ha ido muy bien
Cristina Picon	Mapa 1	00:01:59	partida correcta
bill gates	Mapa 1	00:01:54	super
tom hanks	Difícil	00:02:28	el mapa dificil esta ok
Cristina Picon	Medio	00:01:44	nivel medio ok
Cristina Picon	Fácil	00:00:16	nivel facil y volver a jugar ok

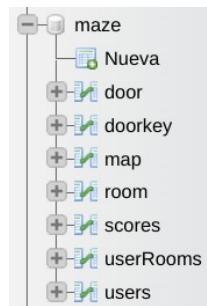
- En esta última pantalla puedes decidir entre volver a jugar (serás redirigido a la pantalla de inicio donde podrás escoger el nivel que quieras) o cerrar sesión.

## **2 ESTRUCTURA DEL PROGRAMA: BASE DE DATOS**

La estructura de la base de datos ha sido diseñada para organizar y almacenar la información necesaria para el correcto funcionamiento del juego.

Se implementa utilizando MySQL a través de phpMyAdmin, lo que facilita la administración y gestión de la base de datos de forma visual y sencilla.

Esta base de datos incluye tablas que almacenan información relacionada con usuarios, mapas, habitaciones, puertas, llaves, puntuaciones y progreso de los jugadores.



### **Users**

	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	id	name	username	password	roomId	coins	gameTime	idKeys	openDoors	mapName
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	1	bill gates	bill		NULL	0	NULL	NULL	NULL	NULL
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	2	tom hanks	tom		1	0	1733584101664	NULL	NULL	Fácil
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	3	Cristina Picon	cris		1	0	1733583897965	NULL	NULL	Fácil
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	4	miquel	miquel		NULL	0	NULL	NULL	NULL	NULL

La tabla "users" almacena información de los jugadores registrados en la app. Incluye campos como el identificador del usuario, su nombre, nombre de usuario, contraseña cifrada, la habitación actual en el juego, cantidad de monedas recogidas, tiempo de juego, llaves recogidas, puertas abiertas y el nombre del mapa en el que se encuentra el jugador.

### **Room**

	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	id	mapId	north	south	east	west	coins	doorKeyId	
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	1	1	1	NULL	NULL	NULL	2	NULL	
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	2	1	NULL	1	4	NULL	NULL	1	
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	3	1	NULL	NULL	NULL	4	NULL	NULL	
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	8	2	5	12	9	15	NULL	NULL	
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	9	2	6	5	NULL	NULL	1	NULL	
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	10	2	NULL	6	8	7	NULL	NULL	
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	11	2	NULL	NULL	7	NULL	1	NULL	
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	12	2	NULL	NULL	NULL	8	1	NULL	
	<input type="checkbox"/>	<input type="checkbox"/>	Editar	<input type="checkbox"/>	Copiar	<input type="checkbox"/>	Borrar	19	2	10	11	NULL	9	NULL	NULL	

La tabla room almacena la información básica de todas las habitaciones de los mapas del juego. Contiene campos como mapId, que relaciona la habitación con su mapa correspondiente, north, south, east, y west, que representan las puertas hacia otras habitaciones, y coins, que indica la cantidad de monedas en la habitación. También incluye el campo doorKeyId, que almacena el identificador de una llave, si existe. Esta tabla es estática y no se modificará durante el progreso del juego, sirviendo como referencia original de las habitaciones en los mapas.

## UserRooms

<input type="checkbox"/>	Perfilando	<a href="#">[ Editar en línea ]</a>	<a href="#">[ Editar ]</a>	<a href="#">[ Explicar SQL ]</a>	<a href="#">[ Crear código PHP ]</a>
<b>id userId roomId mapId north south east west coins doorkeyId</b>					

La tabla userRooms registra el progreso individual de cada usuario dentro del juego, permitiendo que cada usuario interactúe con su propia versión de las habitaciones sin afectar los datos originales de la tabla room. Contiene campos como userId, que identifica al usuario, roomId, que representa la habitación específica a la que el usuario tiene acceso, y mapId, que relaciona la habitación con el mapa correspondiente. También tiene referencias a las puertas en las direcciones north, south, east, y west, así como información sobre las monedas y llaves que el jugador puede recoger en esa habitación (coins, doorKeyId). Esta tabla es dinámica y se actualiza con el progreso de cada jugador en el juego.

## Map

← T →	▼	<b>id</b>	<b>mapName</b>	<b>startRoomId</b>	<b>finishRoomId</b>				
<input type="checkbox"/>		<a href="#">Editar</a>		<a href="#">Copiar</a>		<a href="#">Borrar</a>	1 Fácil	1	3
<input type="checkbox"/>		<a href="#">Editar</a>		<a href="#">Copiar</a>		<a href="#">Borrar</a>	2 Medio	8	26
<input type="checkbox"/>		<a href="#">Editar</a>		<a href="#">Copiar</a>		<a href="#">Borrar</a>	3 Difícil	27	36

La tabla map almacena información sobre los mapas disponibles en el juego. Contiene los campos id, que es el identificador único del mapa; mapName, que almacena el nombre del mapa ("Fácil", "Medio" o "Difícil"); startRoomId, que indica el identificador de la habitación inicial del mapa, y finishRoomId, que representa la habitación donde el jugador debe llegar para completar el mapa.

## Door

			id	room1Id	room2Id	state	doorKeyId				
<input type="checkbox"/>		Editar		Copiar		Borrar	1	1	2	1	NULL
<input type="checkbox"/>		Editar		Copiar		Borrar	4	2	3	0	1
<input type="checkbox"/>		Editar		Copiar		Borrar	5	8	9	0	2
<input type="checkbox"/>		Editar		Copiar		Borrar	6	9	10	1	NULL
<input type="checkbox"/>		Editar		Copiar		Borrar	7	10	11	1	NULL
<input type="checkbox"/>		Editar		Copiar		Borrar	8	10	12	1	NULL
<input type="checkbox"/>		Editar		Copiar		Borrar	9	8	19	1	NULL

La tabla door representa las puertas que conectan diferentes habitaciones dentro del juego. Contiene los campos id, que es el identificador único de la puerta; room1Id y room2Id, que son los identificadores de las dos habitaciones que la puerta conecta; state, que indica si la puerta está abierta (1) o cerrada (0); y doorKeyId, que almacena el identificador de la llave requerida para abrir la puerta si es necesario.

## Doorkey

			id	neededCoins				
<input type="checkbox"/>		Editar		Copiar		Borrar	1	2
<input type="checkbox"/>		Editar		Copiar		Borrar	2	2
<input type="checkbox"/>		Editar		Copiar		Borrar	3	4
<input type="checkbox"/>		Editar		Copiar		Borrar	4	2
<input type="checkbox"/>		Editar		Copiar		Borrar	5	3

La tabla doorkey almacena información sobre las llaves necesarias para desbloquear puertas dentro del juego. Contiene dos campos principales: id, que es el identificador único de cada llave, y neededCoins, que indica la cantidad de monedas que el jugador debe tener para obtener la llave.

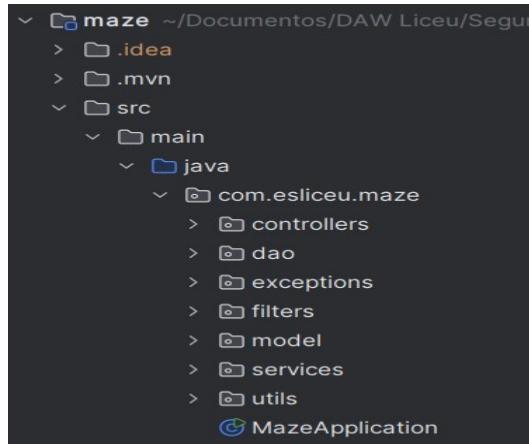
## Scores

			id	userName	mapName	gameTime	comentary		
<input type="checkbox"/>		Editar		Copiar		Borrar	7 bill gates	Mapa 1	18 ha ido muy bien
<input type="checkbox"/>		Editar		Copiar		Borrar	13 Cristina Picon	Mapa 1	119 partida correcta
<input type="checkbox"/>		Editar		Copiar		Borrar	14 bill gates	Mapa 1	114 super
<input type="checkbox"/>		Editar		Copiar		Borrar	22 tom hanks	Difícil	148 el mapa dificil esta ok
<input type="checkbox"/>		Editar		Copiar		Borrar	23 Cristina Picon	Medio	104 nivel medio ok

La tabla scores almacena información relacionada con las partidas de los jugadores en diferentes mapas. Contiene los siguientes campos: id, un identificador único para cada registro; mapName, el nombre del mapa jugado; gameTime, el tiempo que tardó el jugador en completar el mapa; comentary, un comentario personalizado sobre la partida; y userName, el nombre del usuario que jugó la partida.

### **3 . ESTRUCTURA DEL PROGRAMA: SERVIDOR**

El servidor de la aplicación está dividido en varias capas que incluyen controladores, servicios, DAOs (Data Access Objects), modelos, filtros, excepciones y otros útiles, siguiendo un diseño MVC (Modelo-Vista-Controlador).



## 3.1 Modelos (model)

Los modelos son clases que funcionan como representación de las filas de las tablas de la base de datos para que la aplicación pueda manejarlas de una manera más fácil y organizada. Es importante que las clases tengan como atributos los mismos nombres que las columnas de la tabla a la que hacen referencia.

## User

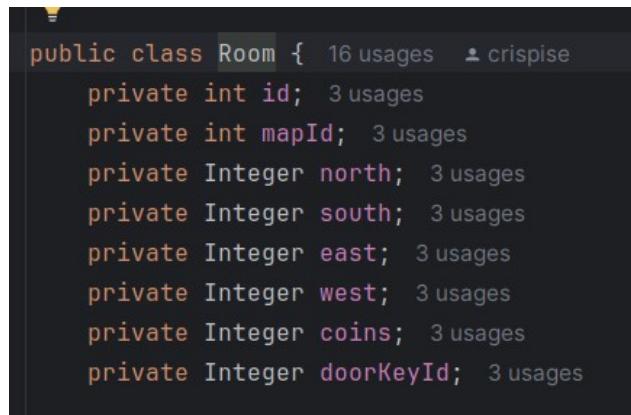
```
public class User {    crispie
    private int id; 3 usages
    private String name; 3 usages
    private String username; 3 usages
    private String password; 3 usages
    private String mapName; 3 usages
    private Integer roomId; 3 usages
    private Integer coins; 3 usages
    private Long gameTime; 3 usages
    private String idKeys; 3 usages
    private String openDoors; 3 usages
```

Esta clase representa una fila de la tabla Users. Almacena datos como el identificador del usuario (id), el nombre, el nombre de usuario, la contraseña, el nombre del mapa actual con el que está jugando (mapName) , la habitación en la que se encuentra (roomId), las monedas recolectadas (coins), las llaves

recogidas (idKeys) y las puertas abiertas (openDoors). También registra el tiempo invertido en el juego a través del atributo gameTime.

Para acceder y modificar los valores de los atributos, la clase proporciona métodos getter y setter para cada campo.

## **Room**



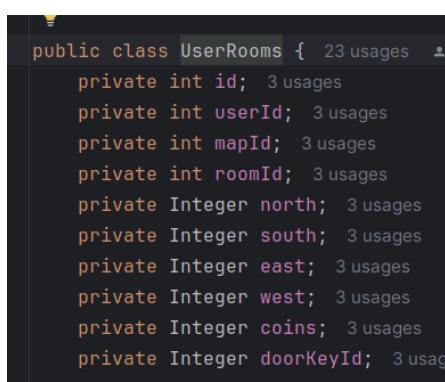
```
public class Room { 16 usages  crispise
    private int id;  3 usages
    private int mapId;  3 usages
    private Integer north;  3 usages
    private Integer south;  3 usages
    private Integer east;  3 usages
    private Integer west;  3 usages
    private Integer coins;  3 usages
    private Integer doorKeyId;  3 usages
```

La clase Room corresponde a una fila de la tabla Room y representa una habitación del laberinto. Cada instancia de Room tiene un identificador único (id) y un identificador de mapa (mapId) que determina a qué mapa pertenece. Además, incluye referencias a las puertas en las direcciones norte, sur, este y oeste (north, south, east, west). Estas referencias representan los id de las puertas que conectan la habitación con las adyacentes, siendo null si no existe una puerta en esa dirección.

El atributo coins indica la cantidad de monedas que el jugador puede recoger en esa habitación, mientras que doorKeyId almacena el identificador de una llave. Si el valor de doorKeyId es null, significa que no hay ninguna llave para recoger en esa habitación. Por el contrario, si contiene un número, ese número representa el identificador de una llave que el jugador puede recoger y utilizar para abrir puertas.

Para acceder y modificar los valores de los atributos, la clase proporciona métodos getter y setter para cada campo.

## **UserRooms**



```
public class UserRooms { 23 usages  crispise
    private int id;  3 usages
    private int userId;  3 usages
    private int mapId;  3 usages
    private int roomId;  3 usages
    private Integer north;  3 usages
    private Integer south;  3 usages
    private Integer east;  3 usages
    private Integer west;  3 usages
    private Integer coins;  3 usages
    private Integer doorKeyId;  3 usag
```

La clase UserRooms representa una fila de la tabla UserRooms de la base de datos, que almacena la información de las habitaciones de un mapa específico para cada usuario que lo selecciona. Esta clase es importante para garantizar que cada jugador pueda jugar su propia partida de forma independiente, con sus propios cambios en las habitaciones, sin afectar las configuraciones originales de los mapas.

Cada atributo en esta clase almacena información clave para el progreso del usuario, como el identificador de usuario (userId), el identificador del mapa (mapId), el identificador de la habitación actual (roomId), los identificadores de las puertas en las direcciones (north, south, east, west), la cantidad de monedas en la habitación (coins) y el identificador de la llave (doorKeyId).

## **Map**

```
public class Map { 18 usages  ± crispise
    private int id;  3 usages
    private String mapName;  3 usages
    private int startRoomId;  3 usages
    private int finishRoomId;  3 usages
```

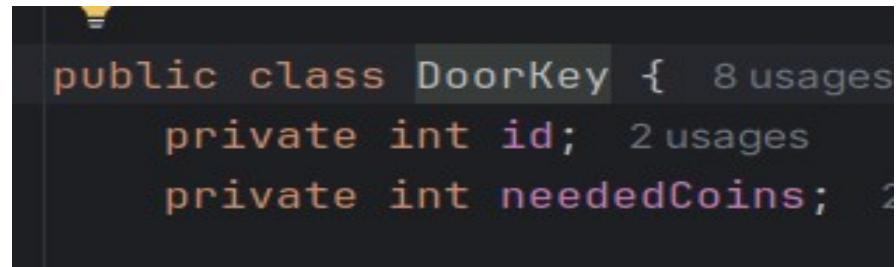
La clase Map representa un mapa dentro del juego. Cada instancia de esta clase contiene información relevante sobre el mapa, como su identificador único (id), el nombre del mapa (mapName), el identificador de la habitación de inicio (startRoomId) y el identificador de la habitación de salida o final (finishRoomId).

## **Door**

```
public class Door { 20 usages  ± crispise
    private int id;  3 usages
    private int room1Id;  3 usages
    private int room2Id;  3 usages
    private int state; //1 = open, 0 = cl
    private Integer doorKeyId;  3 usages
```

La clase Door representa una fila de la tabla Door. Se corresponde con una puerta que conecta dos habitaciones en el laberinto. Contiene atributos como el identificador de la puerta, los id de las habitaciones que conecta (room1Id y room2Id), el estado de la puerta (state, donde 1 es abierta y 0 es cerrada) y el identificador de la llave asociada en el caso de que este cerrada (doorKeyId). Incluye métodos getter y setter para interactuar con estos atributos y un método `toString()` para una mejor visualización de sus datos

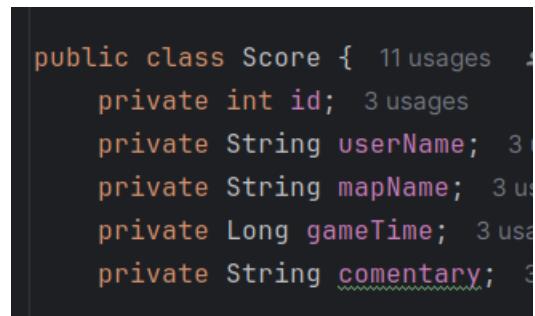
## DoorKey



```
public class DoorKey { 8 usages
    private int id; 2 usages
    private int neededCoins; 2
```

La clase DoorKey representa una fila de la tabla DoorKey. Se corresponde con una llave que el jugador puede recoger en el juego. Contiene un identificador único (id) y la cantidad de monedas necesarias (neededCoins) para obtenerla. Proporciona métodos para acceder y modificar estos atributos.

## Score



```
public class Score { 11 usages
    private int id; 3 usages
    private String userName; 3 usages
    private String mapName; 3 usages
    private Long gameTime; 3 usages
    private String commentary; 3 usages
```

La clase Score almacena la información de una fila de la tabla Score. Da información sobre una partida de un jugador, incluyendo un identificador único, el nombre del jugador, el nombre del mapa, el tiempo empleado para completar el juego y un comentario opcional dejado por el jugador. Proporciona métodos para acceder y modificar estos datos.

## 3.2 DAO (Data Access Objects)

La capa DAO es la encargada de gestionar el acceso a la base de datos, realizando operaciones de lectura, escritura, actualización y eliminación de manera segura y organizada. Cada tabla tendrá su interficie e implementación correspondiente.

Se utilizará **JdbcTemplate**, una herramienta de Spring para ejecutar consultas SQL de manera eficiente.

### UserDAO y UserDAOImpl

```
@Repository no usages ▲ crispise
public class UserDAOImpl implements UserDAO {
    @Autowired 9 usages
    JdbcTemplate jdbcTemplate;

    @Override ▲ crispise
    public User getUserByUsername(String username) {
        String sql = "select * from users where username = ?";
        try {
            return jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<>(User.class), username);
        } catch (EmptyResultDataAccessException e) {
            return null;
        }
    }

    @Override 1 usage ▲ crispise
    public void saveUser(User user) {
        jdbcTemplate.update("insert into users (name, username, password) values (?, ?, ?)",
                           user.getName(), user.getUsername(), user.getPassword());
    }

    @Override 1 usage ▲ crispise
    public void updateUserRoomStatus(String username, int roomId) {
        jdbcTemplate.update("UPDATE users SET roomId = ? WHERE username = ?", roomId, username);
    }

    @Override 2 usages ▲ crispise
    public void updateTotalUserCoins(String username, int totalCoins) {
        jdbcTemplate.update("UPDATE users SET coins = ? WHERE username = ?", totalCoins, username);
    }

    @Override 1 usage ▲ crispise
    public void updateTotalUserKeys(String username, String userKeys) {
        jdbcTemplate.update("UPDATE users SET idKeys = ? WHERE username = ?", userKeys, username);
    }
```

```
@Override 1 usage ▲ crispise
public void updateOpenDoors(String username, String doorKeyId) {
    jdbcTemplate.update("update users set openDoors = ? where username = ?", doorKeyId, username);
}

@Override 1 usage ▲ crispise
public void resetUser(Object roomId, int initialCoins, Object gameTime, Object idKeys, Object openDoors, String username, String mapName) {
    jdbcTemplate.update(
        "UPDATE users SET roomId = ?, coins = ?, gameTime = ?, idKeys = ?, openDoors = ?, mapName = ? WHERE username = ?",
        roomId, initialCoins, gameTime, idKeys, openDoors, mapName, username
    );
}

@Override 1 usage ▲ crispise
public void updateGameTime(String username, long currentTime) {
    jdbcTemplate.update("update users set gameTime = ? where username = ?", currentTime, username);
}

@Override 1 usage ▲ crispise
public void updateMapName(String username, String mapName) {
    jdbcTemplate.update("update users set mapName = ? where username = ?", mapName, username);
}
```

La clase UserDAOImpl implementa la interfaz UserDAO y se encarga de gestionar la información de los usuarios en la base de datos.

El método **getUserByUsername(username)** busca un usuario en la base de datos por su nombre de usuario. Si el usuario existe, se devuelve el objeto correspondiente; de lo contrario, se retorna null. Por otro lado, el método **saveUser(user)** permite guardar un nuevo usuario en la base de datos con su nombre, nombre de usuario y contraseña.

Además, la clase cuenta con otros métodos que actualizan la información del usuario, como **updateUserRoomStatus(username, roomId)**, que actualiza la habitación actual del usuario, y **updateTotalUserCoins(username, totalCoins)**, que permite modificar la cantidad total de monedas del usuario. También están disponibles métodos para actualizar las llaves del usuario (**updateTotalUserKeys**) y las puertas abiertas (**updateOpenDoors**).

Finalmente, el método **resetUser** es utilizado para restablecer el progreso de un usuario con valores predeterminados, actualizando información como monedas, tiempo de juego y el mapa seleccionado. Por último, los métodos **updateGameTime** y **updateMapName** permiten actualizar el tiempo de juego y el nombre del mapa que el usuario está explorando, respectivamente.

## RoomDAO y RoomDAOImpl

```
@Repository no usages  ± crispise
public class RoomDAOImpl implements RoomDAO {
    @Autowired 2 usages
    JdbcTemplate jdbcTemplate;

    @Override 2 usages  ± crispise
    public Room getRoomById(int roomId) {
        return jdbcTemplate.queryForObject(sql: "select * from room where id = ?",
            new BeanPropertyRowMapper<>(Room.class), roomId);
    }

    @Override 1 usage  ± crispise
    public List<Room> getAllRoomsByMapId(int mapId) {
        return jdbcTemplate.query(sql: "select * from room where mapId = ?",
            new BeanPropertyRowMapper<>(Room.class), mapId);
    }
}
```

La clase RoomDAOImpl implementa la interfaz RoomDAO y se encarga de manejar la información relacionada con las habitaciones en la base de datos.

El método **getRoomById(int roomId)** permite buscar una habitación específica en la base de datos utilizando su identificador único (id). Si la habitación existe, se devuelve el objeto correspondiente; de lo contrario, se manejará la excepción automáticamente.

Por otro lado, el método **getAllRoomsByMapId(int mapId)** devuelve una lista de todas las habitaciones asociadas a un mapa específico identificado por el parámetro mapId.

## UserRoomsDAO y UserRoomsDAOImpl

```
@Repository no usages ± crispise
public class UserRoomsDAOImpl implements UserRoomsDAO {
    @Autowired 6 usages
    JdbcTemplate jdbcTemplate;

    public void insertUserRoom(int userId, int roomId, int mapId, Integer north, Integer south, Integer east, Integer west, Integer coins, Integer doorKeyId) {
        jdbcTemplate.update(sql: "insert into userRooms (userId, roomId, mapId, north, south, east, west, coins, doorKeyId) values (?, ?, ?, ?, ?, ?, ?, ?, ?)", userId, roomId, mapId, north, south, east, west, coins, doorKeyId);
    }

    @Override 11 usages ± crispise
    public UserRooms getUserRoomByRoomIdAndUserId(int userId, int roomId) {
        return jdbcTemplate.queryForObject(sql: "select * from userRooms where userId = ? AND roomId = ?", new BeanPropertyRowMapper<>(UserRooms.class), userId, roomId);
    }

    @Override 1 usage ± crispise
    public void updateTotalCoins(int userId, int roomId, int roomTotalCoins) {
        jdbcTemplate.update(sql: "UPDATE userRooms SET coins = ? WHERE roomId = ? AND userId = ?", roomTotalCoins, roomId, userId);
    }

    @Override 1 usage ± crispise
    public void updateTotalKeys(int userId, int roomId, Integer roomTotalKeys) {
        jdbcTemplate.update(sql: "UPDATE userRooms SET doorKeyId = ? WHERE roomId = ? AND userId = ?", ...args: null, roomId, userId);
    }

    @Override 1 usage ± crispise
    public void deleteUserRoomsByUserIdAndMapId(int userId, int mapId) {
        jdbcTemplate.update(sql: "DELETE FROM userRooms WHERE userId = ? AND mapId = ?", userId, mapId);
    }

    @Override 1 usage ± crispise
    public void deleteUserRoomsByUserIdExcludingMapID(int userId, int mapId) {
        jdbcTemplate.update(sql: "DELETE FROM userRooms WHERE userId = ? AND mapId != ?", userId, mapId);
    }
}
```

La clase UserRoomsDAOImpl implementa la interfaz UserRoomsDAO y es responsable de manejar las operaciones relacionadas con la relación entre usuarios y habitaciones en la base de datos.

El método **insertUserRoom(int userId, int roomId, int mapId, Integer north, Integer south, Integer east, Integer west, Integer coins, Integer doorKeyId)** permite insertar un nuevo registro en la tabla userRooms.

El método **getUserRoomByRoomIdAndUserId(int userId, int roomId)** busca un registro en la base de datos que coincida con el identificador de usuario y el identificador de habitación proporcionados. Si se encuentra el registro, devuelve el objeto correspondiente.

Además, el método **updateTotalCoins(int userId, int roomId, int roomTotalCoins)** permite actualizar la cantidad de monedas asociada a una habitación específica para un usuario determinado. También, el método **updateTotalKeys(int userId, int roomId, Integer roomTotalKeys)** actualiza la llave correspondiente para una habitación específica de un usuario.

Por último, los métodos **`deleteUserRoomsByUserIdAndMapId(int userId, int mapId)`** y **`deleteUserRoomsByUserIdExcludingMapID(int userId, int mapId)`** permiten eliminar registros de la tabla userRooms. El primero elimina los registros para un usuario en un mapa específico, mientras que el segundo elimina registros para el usuario en otros mapas que no coincidan con el parámetro especificado.

## MapDAO y MapDAOImpl

```
@Repository no usages ▲ crispise
public class MapDAOImpl implements MapDAO {
    @Autowired 2 usages
    JdbcTemplate jdbcTemplate;

    @Override 1 usage ▲ crispise
    public List<Map> getAllMaps() {
        return jdbcTemplate.query(sql: "select * from map", new BeanPropertyRowMapper<>(Map.class));
    }

    @Override 4 usages ▲ crispise
    public Map getMapById(int id) {
        return jdbcTemplate.queryForObject(sql: "select * from map where id = ?",
            new BeanPropertyRowMapper<>(Map.class), id);
    }
}
```

La clase MapDAOImpl implementa la interfaz MapDAO y es responsable de manejar las operaciones relacionadas con los mapas en la base de datos.

El método **`getAllMaps()`** recupera todos los mapas de la tabla map en la base de datos. Devuelve una lista de objetos Map correspondientes a todas las entradas de la tabla.

Por otro lado, el método **`getMapById(int id)`** busca un mapa específico en la base de datos utilizando su identificador. Si se encuentra un registro con el ID correspondiente, devuelve el objeto Map.

## DoorDAO y DoorDAOImpl

```
@Repository no usages ▲ crispise
public class DoorDAOImpl implements DoorDAO {
    @Autowired 2 usages
    JdbcTemplate jdbcTemplate;

    @Override 1 usage ▲ crispise
    public List<Door> findAllDoorsByRoomId(int roomId) {
        return jdbcTemplate.query(sql: "select * from door where room1Id = ? OR room2Id = ?",
            new BeanPropertyRowMapper<>(Door.class), roomId, roomId);
    }

    @Override 4 usages ▲ crispise
    public Door getDoorById(int doorId) {
        return jdbcTemplate.queryForObject(sql: "select * from door where id = ?",
            new BeanPropertyRowMapper<>(Door.class), doorId);
    }
}
```

La clase DoorDAOImpl implementa la interfaz DoorDAO y es responsable de realizar operaciones relacionadas con las puertas en la base de datos.

El método ***findAllDoorsByRoomId(int roomId)*** busca todas las puertas en la base de datos que estén asociadas a una habitación específica, ya sea como room1Id o room2Id. Devuelve una lista de objetos Door que representan las puertas encontradas.

Por otro lado, el método ***getDoorById(int doorId)*** busca una puerta específica en la base de datos utilizando su identificador. Si se encuentra un registro con el ID proporcionado, devuelve el objeto Door.

## **DoorKeyDAO y DoorKeyDAOImpl**

```
@Repository no usages ▾ crispise
public class DoorKeyDAOImpl implements DoorKeyDAO {
    @Autowired 1 usage
    JdbcTemplate jdbcTemplate;

    @Override 1 usage ▾ crispise
    public DoorKey getKeyById(Integer doorKeyId) {
        return jdbcTemplate.queryForObject(sql: "select * from doorkey where id = ?", new BeanPropertyRowMapper<>(DoorKey.class), doorKeyId);
    }
}
```

La clase DoorKeyDAOImpl implementa la interfaz DoorKeyDAO y es responsable de realizar operaciones relacionadas con las llaves de puertas en la base de datos.

El método ***getKeyById(Integer doorKeyId)*** busca una llave específica en la base de datos utilizando su identificador (doorKeyId). Si se encuentra un registro con el ID proporcionado, devuelve el objeto DoorKey.

## **ScoreDAO y ScoreDAOImpl**

```
@Repository no usages ▾ crispise
public class ScoreDAOImpl implements ScoreDAO {
    @Autowired 2 usages
    JdbcTemplate jdbcTemplate;

    @Override 1 usage ▾ crispise
    public void insertScore(User user, long gameTime, String commentary) {
        jdbcTemplate.update(sql: "insert into scores (userName, mapName, gameTime, commentary) values (?, ?, ?, ?)"
            , user.getName(), user.getMapName(), gameTime, commentary);
    }

    @Override 1 usage ▾ crispise
    public List<Score> getAllScores() {
        return jdbcTemplate.query(sql: "select * from scores", new BeanPropertyRowMapper<>(Score.class));
    }
}
```

La clase ScoreDAOImpl implementa la interfaz ScoreDAO y se encarga de realizar operaciones relacionadas con los puntajes de los usuarios en la base de datos.

El método ***insertScore(User user, long gameTime, String commentary)*** permite insertar una nueva puntuación en la base de datos. El método ***getAllScores()*** recupera todas las puntuaciones almacenadas en la base de datos y las devuelve como una lista de objetos Score.

### 3.3. Excepciones (exceptions)

Las excepciones personalizadas permiten una gestión de errores específica.

**NameTooShortException:** Se lanza cuando el nombre tiene menos de 6 caracteres.

**PasswordTooShortException:** Se lanza cuando la contraseña tiene menos de 5 caracteres.

**UserExistsException:** Se utilizan para manejar errores en el proceso de registro cuando se introduce un username que ya existe.

### 3.4 Servicios (services)

La capa de servicios encapsula la lógica de negocio y es llamada por los controladores para gestionar acciones específicas.

#### RegisterService

La clase RegisterService tiene como función principal gestionar el registro de nuevos usuarios en la aplicación. Para ello, utiliza una instancia de UserDAO y Encryptor. El UserDAO es el encargado de acceder y manipular la base de datos, mientras que Encryptor se usa para cifrar las contraseñas de los usuarios antes de guardarlas de forma segura.

```
@Service 2 usages  ± crispise
public class RegisterService {
    @Autowired 2 usages
    UserDAO userDAO;
    @Autowired 1 usage
    Encryptor encryptor;

    public void registerUser(String name, String username, String password) throws Exception { 1 usage  ± crispise
        boolean correctUsername = checkIfUsernameExists(username);
        boolean correctPassword = checkPasswordLength(password);
        boolean correctName = checkNameLength(name);

        if (!correctUsername) {
            throw new UserExistsException("El username ya existe.");
        } else if (!correctPassword) {
            throw new PasswordTooShortException("La contraseña tiene que tener un mínimo de 5 caracteres.");
        } else if (!correctName) {
            throw new NameTooShortException("El nombre es demasiado corto, tiene que tener 6 caracteres.");
        }
        User user = new User();
        user.setName(name);
        user.setUsername(username);
        String encryptPasw = encryptor.encryptString(password);
        user.setPassword(encryptPasw);
        userDAO.saveUser(user);
    }

    private boolean checkNameLength(String name) { 1 usage  ± crispise
        if (name.length() < 6) {
            return false;
        }
        return true;
    }
}
```

```

private boolean checkPasswordLength(String password) { 1 usa
    if (password.length() < 5) {
        return false;
    }
    return true;
}

private boolean checkIfUsernameExists(String username) { 1
    User user = userDao.getUserByUsername(username);
    if (user != null) {
        return false;
    }
    return true;
}

```

El método más importante es ***registerUser(name, username, password)***, que recibe como parámetros el nombre completo, el nombre de usuario y la contraseña del nuevo usuario. Antes de registrar al usuario, se realizan varias validaciones a través de métodos privados para garantizar que los datos introducidos sean válidos. Por ejemplo, el método ***checkNameLength(name)*** verifica que el nombre tenga al menos 6 caracteres, mientras que ***checkPasswordLength(password)*** comprueba que la contraseña tenga un mínimo de 5 caracteres.

Por otro lado, el método ***checkIfUsernameExists(username)*** verifica si el nombre de usuario ya existe en la base de datos. Si alguna de las validaciones falla, se lanza una excepción específica con un mensaje que indica el problema, ya sea un nombre de usuario duplicado o una contraseña o nombre demasiado cortos. Si las validaciones son correctas, la contraseña es cifrada mediante el método ***encryptString(password)*** de la clase Encryptor, y se procede a crear un nuevo objeto User.

Finalmente, una vez validada la información y cifrada la contraseña, se guarda el usuario en la base de datos mediante el método ***saveUser(user)*** de UserDao.

## LoginService

La clase LoginService tiene como objetivo principal gestionar el proceso de inicio de sesión de los usuarios en la aplicación. Para ello, utiliza una instancia de UserDao para acceder a la información de los usuarios en la base de datos y una instancia de Encryptor para cifrar contraseñas de manera segura.

```

@Service 2 usages  ▲ críspise
public class LoginService {
    @Autowired 1 usage
    UserDAO userDAO;
    @Autowired 1 usage
    Encryptor encryptor;

    public User checkUser(String username, String password) throws NoSuchAlgorithmException {
        User u = userDAO.getUserByUsername(username);
        String encryptPassw = encryptor.encryptString(password);
        if (u != null) {
            if (u.getPassword().equals(encryptPassw)) {
                return u;
            }
        }
        return null;
    }
}

```

Contiene el método ***checkUser(username, password)***, el cual recibe como parámetros el nombre de usuario y la contraseña introducidos por el usuario. Primero, se busca el usuario en la base de datos utilizando el método ***getUserByUsername(username)*** de UserDAO. Luego, se cifra la contraseña proporcionada utilizando el método ***encryptString(password)*** de la clase Encryptor.

Si se encuentra un usuario con el nombre de usuario introducido (*u != null*) y la contraseña cifrada coincide con la guardada en la base de datos (*u.getPassword().equals(encryptPassw)*), se devuelve el objeto User. Si no se cumple esta condición, se retorna null.

## StartService

La clase StartService tiene como principal responsabilidad manejar la lógica relacionada con la inicialización de mapas, la carga de habitaciones, el estado de las puertas y la interacción inicial de los usuarios en la aplicación. Utiliza múltiples servicios DAO como MapDAO, RoomDAO, DoorDAO, UserDAO y UserRoomsDAO para acceder a la base de datos y realizar las operaciones necesarias.

```

public class StartService {
    @Autowired 3 usages
    MapDAO mapDAO;
    @Autowired 2 usages
    RoomDAO roomDAO;
    @Autowired 1 usage
    DoorDAO doorDAO;
    @Autowired 5 usages
    UserDAO userDAO;
    @Autowired 6 usages
    UserRoomsDAO userRoomsDAO;

    public List<Map> getAllMaps() { 1 usage
        return mapDAO.getAllMaps();
    }
}

```

La función **`getAllMaps()`** simplemente ejecuta una consulta a través de `mapDAO.getAllMaps()`, el cual devuelve una lista de todos los mapas almacenados en la base de datos.

```

public String getFirstJson(String mapId, String username) { 1 usage  ± crispise
    Map map = mapDAO.getMapById(Integer.parseInt(mapId));
    List<Room> mapRooms = roomDAO.getAllRoomsByMapId(map.getId());
    User user = userDao.getUserByUsername(username);
    updateGameTime(user);
    updateMapNameInUser(user, map);
    UserRooms actualRoom;
    if (user.getRoomId() == null) {
        updateUserRoomsWithRoomMaps(mapRooms, user);
        actualRoom = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), map.getStartRoomId());
    } else {
        Room room = roomDAO.getRoomById(user.getRoomId());
        if (room.getMapId() == map.getId()){
            actualRoom = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), user.getRoomId());
        }else {////casos en los que al cerrar sesion y volver a jugar en diferentes navegadores escogas otro mapa
            userRoomsDAO.deleteUserRoomsByUserIdExcludingMapID(user.getId(), map.getId());
            updateUserRoomsWithRoomMaps(mapRooms, user);
            actualRoom = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), map.getStartRoomId());
        }
    }
    return createJson(username, actualRoom, errorMessage: "");
}

private void updateGameTime(User user) { 1 usage  ± crispise
    long currentTime = System.currentTimeMillis(); // Tiempo actual en milisegundos
    userDao.updateGameTime(user.getUsername(), currentTime);
}

private void updateMapNameInUser(User user, Map map) { 1 usage  ± crispise
    userDao.updateMapName(user.getUsername(), map.getMapName());
}

private void updateUserRoomsWithRoomMaps(List<Room> mapRooms, User user) { 2 usages  ± crispise
    for (Room room : mapRooms) {
        userRoomsDAO.insertUserRoom(user.getId(), room.getId(), room.getMapId(),
            room.getNorth(), room.getSouth(), room.getEast(), room.getWest(), room.getCoins(), room.getDoorKeyId());
    }
}

```

El método **`getFirstJson(mapId, username)`** es fundamental, ya que se encarga de preparar y devolver la información inicial en formato JSON para el usuario al ingresar en una partida. Este proceso involucra varios pasos clave utilizando los DAOs MapDAO, RoomDAO, UserDao, DoorDAO y UserRoomsDAO. Primero, se obtiene el mapa con `mapDAO.getMapById(mapId)` y las habitaciones de ese mapa con `roomDAO.getAllRoomsByMapId(map.getId())`. Luego, se verifica el estado del usuario en relación con el mapa mediante userDao. También actualiza información del usuario, como la hora de inicio del juego **`updateGameTime(user)`** o el nombre del mapa escogido **`updateMapNameInUser(user, map)`**.

Posteriormente revisa si el usuario tiene una habitación asignada. Si el usuario no tiene una, significa que es la primera vez que empieza la partida. Como acaba de empezar la partida, se tendrán que cargar todas las habitaciones del

mapa seleccionado por el usuario en la tabla UserRooms a través del método ***updateUserRoomsWithRoomMaps(mapRooms, user)***.

En el caso de que el usuario si tuviese una habitación asignada, se verifica que esta pertenezca al último mapa seleccionado. Si no coincide (*casos en los que al cerrar sesión y volver a jugar en diferentes navegadores se escogen mapas diferentes*), se eliminan las asignaciones anteriores usando userRoomsDAO.deleteUserRoomsByUserIdExcludingMapID y se asigna nuevamente el estado de las habitaciones del nuevo mapa.

Finalmente, el método devuelve un JSON construido mediante la función ***createJson(username, actualRoom, "")***.

```
public String createJson(String username, UserRooms userRooms, String errorMessage) { 15 usages  ▲ crispise
    userDAO.updateUserRoomStatus(username, userRooms.getRoomId());
    User user = userDAO.getUserByUsername(username);
    List<Door> doors = doorDAO.findAllDoorsByRoomId(userRooms.getRoomId());
    HashMap<String, Object> mapa = new HashMap<>();
    updateHashMap(userRooms, errorMessage, user, mapa, doors);
    Gson gson = new Gson();
    return gson.toJson(mapa);
}

private void updateHashMap(UserRooms userRooms, String errorMessage, User user, HashMap<String, Object> mapa, List<Door> doors) {
    if (checkIfUserHasKey(user, userRooms)) mapa.put("keys", userRooms.getDoorKeyId());
    if (user.getOpenDoors() != null) updateDoorState(user, doors);
    if (checkFinalRoom(user)) mapa.put("finalRoom", user.getRoomId());
    System.out.println(user.getMapName());
    mapa.put("mapName", user.getMapName());
    mapa.put("north", userRooms.getNorth());
    mapa.put("south", userRooms.getSouth());
    mapa.put("east", userRooms.getEast());
    mapa.put("west", userRooms.getWest());
    mapa.put("coins", userRooms.getCoins());
    mapa.put("userRoom", user.getRoomId());
    mapa.put("userCoins", user.getCoins());
    mapa.put("userKeys", getKeyInfo(user));
    mapa.put("doors", getDoorsInfo(doors));
    mapa.put("errorMessage", errorMessage);
}

private boolean checkIfUserHasKey(User user, UserRooms userRooms) { 1 usage  ▲ crispise
    if (user.getIdKeys() == null || user.getIdKeys().isEmpty()) {
        return true;
    }
    if (userRooms.getDoorKeyId() != null) {
        List<String> userKeys = Arrays.asList(user.getIdKeys().split(regex: "[ ]"));
        String roomKey = String.valueOf(userRooms.getDoorKeyId());
        if (!userKeys.contains(roomKey)) return true;
    }
    return false;
}
```

```

private static void updateDoorState(User user, List<Door> doors) { 1 usage  ± crispie
    Set<String> openDoorIds = new HashSet<>(Arrays.asList(user.getOpenDoors().split( regex: " ")));
    for (Door door : doors) {
        if (openDoorIds.contains(String.valueOf(door.getId()))) {
            door.setState(1);
        }
    }
}

private boolean checkFinalRoom(User user) { 1 usage  ± crispie
    UserRooms userRooms = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), user.getRoomId());
    Map map = mapDAO.getMapById(userRooms.getMapId());
    if (user.getRoomId() == map.getFinishRoomId()) {
        return true;
    }
    return false;
}

private String getKeysInfo(User user) { 1 usage  ± crispie
    if (user.getIdKeys() != null) {
        List<String> idKeys = Arrays.asList(user.getIdKeys().split( regex: " "));
        StringBuilder stringBuilder = new StringBuilder();
        for (int i = 0; i < idKeys.size(); i++) {
            stringBuilder.append("Llave ").append(idKeys.get(i));
            if (i != idKeys.size() - 1) {
                stringBuilder.append(",");
            }
        }
        return stringBuilder.toString();
    }
    return "";
}

```

```

private static List<HashMap<String, Object>> getDoorsInfo(List<Door> doors) {
    List<HashMap<String, Object>> doorsInfo = new ArrayList<>();
    if (doors != null && !doors.isEmpty()) {
        for (Door door : doors) {
            HashMap<String, Object> doorInfo = new HashMap<>();
            doorInfo.put("doorId", door.getId());
            doorInfo.put("state", door.getState());
            doorsInfo.add(doorInfo);
        }
    }
    return doorsInfo;
}

```

El método ***createJson(String username, UserRooms userRooms, String errorMessage)*** es responsable de generar la estructura JSON que contiene toda la información relevante para enviar al cliente. En primer lugar, actualiza en la base de datos el ID de la habitación en la que el usuario se encuentra actualmente mediante la llamada a `userDAO.updateUserRoomStatus(username, userRooms.getRoomId())`. Luego, recupera un objeto User con la información actualizada correspondiente a ese usuario.

A continuación, obtiene información sobre todas las puertas presentes en la habitación actual mediante el método `doorDAO.findAllDoors_ByRoomId(userRooms.getRoomId())`. Con estos datos, se procede a crear un `HashMap` que servirá como la estructura principal para almacenar la información que se enviará en el JSON. Este `HashMap` es modificado posteriormente a través del método ***updateHashMap()***, donde se agregan elementos clave como el

estado de las puertas **`getDoorsInfo(doors)`**, la cantidad de monedas, las llaves disponibles **`getKeysInfo(user)`** y cualquier error que pueda haber ocurrido durante el juego. También se hacen comprobaciones importantes, como verificar si el usuario tiene llaves **`checkIfUserHasKey(user, userRooms)`**, actualizar el estado de las puertas abiertas **`updateDoorState(user, doors)`** y determinar si el jugador ha alcanzado la última habitación del mapa **`checkFinalRoom(User user)`**, lo que indica el final del juego.

El método mencionado anteriormente **`getKeysInfo(User user)`**, convierte las llaves que posee el jugador en un String comprensible para el cliente. Del mismo modo, el método **`getDoorsInfo(List<Door> doors)`** es responsable de recopilar el estado de las puertas abiertas en la habitación actual y convertirlas en una lista JSON para enviarlas al cliente.

Con todos estos métodos, se puede enviar al cliente una representación coherente y actualizada del estado actual del usuario en el juego.

## **NavService**

La clase NavService tiene como objetivo principal gestionar la lógica de navegación del jugador dentro del juego. Para ello, interactúa con varias clases importantes como UserDAO, RoomDAO, DoorDAO, MapDAO, StartService y UserRoomsDAO. Estas, permiten acceder a datos relacionados con usuarios, habitaciones, puertas y mapas, así como enviar información al cliente mediante el servicio StartService.

```
@Service 3 usages ▲ crispise
public class NavService {
    @Autowired 1 usage
    UserDAO userDAO;
    @Autowired no usages
    RoomDAO roomDAO;
    @Autowired| 4 usages
    DoorDAO doorDAO;
    @Autowired 1 usage
    MapDAO mapDAO;
    @Autowired 4 usages
    StartService startService;
    @Autowired 3 usages
    UserRoomsDAO userRoomsDAO;

    public String trySelectedDirection(String direction, String username) { 1 usage ▲ crispise
        User user = userDAO.getUserByUsername(username);
        if (user.getRoomId() == null) return "goToStart";
        UserRooms actualUserRoom = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), user.getRoomId());
        Map map = mapDAO.getMapById(actualUserRoom.getMapId());
        if (user.getRoomId() == map.getFinishRoomId()) {
            return startService.createJson(username, actualUserRoom, errorMessage: "El juego ha acabado");
        }

        Door door = findDoor(direction, actualUserRoom);
        if (door == null) return startService.createJson(username, actualUserRoom, errorMessage: "No hay puertas en esa dirección");

        return treatDoors(username, user, door, actualUserRoom);
    }
}
```

```

private String treatDoors(String username, User user, Door door, UserRooms actualUserRoom) { 1 usage  ± c
    if (user.getOpenDoors() != null) {
        Set<String> openDoorIds = new HashSet<>(Arrays.asList(user.getOpenDoors().split(regex: ",")));
        if (openDoorIds.contains(String.valueOf(door.getId()))) {
            door.setState(1);
        }
    }
    if (door.getState() == 1) {
        UserRooms newUserRooms = null;
        if (door.getRoom1Id() == actualUserRoom.getRoomId()) {
            newUserRooms = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), door.getRoom2Id());
        } else if (door.getRoom2Id() == actualUserRoom.getRoomId()) {
            newUserRooms = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), door.getRoom1Id());
        }
        return startService.createJson(username, newUserRooms, errorMessage: "");
    }
    return startService.createJson(username, actualUserRoom, errorMessage: "La puerta esta cerrada");
}

```

```

public Door findDoor(String direction, UserRooms actualUserRoom) { 2 usages  ± crispise
    switch (direction) {
        case "north":
            if (actualUserRoom.getNorth() != null) return doorDAO.getDoorById(actualUserRoom.getNorth());
            break;
        case "south":
            if (actualUserRoom.getSouth() != null) return doorDAO.getDoorById(actualUserRoom.getSouth());
            break;
        case "east":
            if (actualUserRoom.getEast() != null) return doorDAO.getDoorById(actualUserRoom.getEast());
            break;
        case "west":
            if (actualUserRoom.getWest() != null) return doorDAO.getDoorById(actualUserRoom.getWest());
            break;
    }
    return null;
}

```

El método ***trySelectedDirection(String direction, String username)*** es esencial para procesar el intento de movimiento del jugador en una dirección específica. El método comienza verificando si el usuario tiene una habitación asignada. Si no es así, se devuelve la cadena "goToStart", lo que indica que el jugador debe regresar al inicio del juego.

En caso contrario, se obtiene la habitación actual del usuario utilizando `userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), user.getRoomId())`. Luego, se obtiene el mapa correspondiente con `mapDAO.getMapById(actualUserRoom.getMapId())`.

Si la habitación actual del usuario coincide con la sala final presente en la información del mapa, significa que es la última habitación y que se ha llegado al final del juego. Por tanto, el método devuelve la estructura JSON generada por ***startService.createJson(username, actualUserRoom, "El juego ha acabado")***, lo que indica que el jugador ha completado el juego.

Si no es así, el método busca la puerta en la dirección seleccionada utilizando ***findDoor(direction, actualUserRoom)***. En caso de que no haya una puerta en la dirección indicada, se devuelve un mensaje con el error generado a través de ***startService.createJson(username, actualUserRoom, "No hay puertas en esa dirección")***.

Si se encuentra una puerta, el método ***treatDoors(username, user, door, actualUserRoom)*** se encarga de procesar la lógica de interacción con dicha puerta y ver si el usuario puede atravesarla.

En primer lugar, se verifica si el usuario tiene puertas abiertas registradas (`user.getOpenDoors()`). Si es así, se convierten estas puertas en un set mediante el método `split(",")`. Esto permite una búsqueda rápida para determinar si la puerta actual está abierta para el usuario.

A continuación, si la puerta actual forma parte de la lista de puertas abiertas por el usuario, se marca su estado como abierto con `door.setState(1)`. Esto indica que la puerta está accesible para el usuario y puede cruzarse. Por otro lado, si la puerta no está abierta, el usuario no podrá avanzar hacia otra habitación a través de ella y se devolverá un json con un mensaje indicando que la puerta está cerrada.

Si la puerta se encuentra abierta (`door.getState() == 1`), se determina a qué nueva habitación se moverá el usuario. Esto se maneja mediante una comprobación para identificar en qué dirección se encuentra la puerta desde la habitación actual del usuario. Si la puerta está asociada a la Room1Id desde la habitación actual (`actualUserRoom.getRoomId()`), se accede a la información de la Room2Id. Por el contrario, si la puerta está asociada a la Room2Id, se accede a la información de la Room1Id. Estas habitaciones se buscan utilizando el método `userRoomsDAO.getUserRoomByRoomIdAndUserId`. Una vez identificada la nueva habitación, se envía la información al cliente a través del método ***startService.createJson(username, newUserRooms, "")***.

Esta lógica asegura que el usuario solo pueda cruzar puertas si estas están abiertas.

## GetCoinService

El servicio GetCoinService es una clase encargada de manejar la lógica relacionada con la interacción del usuario y las monedas en el juego. Su objetivo principal es permitir que el usuario pueda recoger monedas de su habitación actual, actualizar su cantidad de monedas y actualizar la cantidad de monedas restantes en la habitación después de realizar la operación.

```
@Service 2 usages  ± crispise
public class GetCoinService {
    @Autowired 2 usages
    UserDAO userDAO;
    @Autowired 2 usages
    StartService startService;
    @Autowired 3 usages
    UserRoomsDAO userRoomsDAO;

    public String addCoinToUser(String username) { 1 usage  ± crispise
        User user = userDAO.getUserByUsername(username);
        if (user.getRoomId() == null) return "goToStart";
        UserRooms actualUserRoom = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), user.getRoomId());
        if (actualUserRoom.getCoins() == null || actualUserRoom.getCoins() == 0) {
            return startService.createJson(username, actualUserRoom, errorMessage: "No hay monedas en esta habitación");
        }
        int userTotalCoins;
        if (user.getCoins() == null) {
            userTotalCoins = 1;
        } else {
            userTotalCoins = user.getCoins() + 1;
        }
        int roomTotalCoins = actualUserRoom.getCoins() - 1;
        userDAO.updateTotalUserCoins(username, userTotalCoins);
        userRoomsDAO.updateTotalCoins(user.getId(), actualUserRoom.getRoomId(), roomTotalCoins);
        UserRooms updateUserRoom = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), actualUserRoom.getRoomId());
        return startService.createJson(username, updateUserRoom, errorMessage: "");
    }
}
```

El método principal es **addCoinToUser**, que gestiona todo el proceso para que el usuario pueda recoger una moneda. En primer lugar, se obtiene el objeto User correspondiente al nombre de usuario proporcionado a través de userDAO.getUserByUsername(username). Si el usuario no tiene una habitación asignada (user.getRoomId() == null), se devuelve un mensaje redirigiendo al usuario a la pantalla de inicio mediante return "goToStart".

Luego, se consulta la habitación actual donde el usuario se encuentra utilizando userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), user.getRoomId()). Si en la habitación actual no hay monedas disponibles (actualUserRoom.getCoins() == null || actualUserRoom.getCoins() == 0), se devuelve un mensaje informando que no hay monedas para recoger en esa habitación mediante **startService.createJson(username, actualUserRoom, "No hay monedas en esta habitación")**.

Si el usuario puede recoger una moneda, se calcula la cantidad total de monedas que tendrá después de la operación. Si el usuario no tiene monedas

previamente (`user.getCoins() == null`), se inicializa su total con 1; de lo contrario, se suma 1 a su cantidad actual.

Por otro lado, la cantidad de monedas en la habitación se actualiza restando 1 a partir de la cantidad actual de monedas en la habitación, utilizando `actualUserRoom.getCoins() - 1`.

Una vez calculados los nuevos totales, se procede a actualizar la información en la base de datos mediante dos llamadas: `userDAO.updateTotalUserCoins(username, userTotalCoins)` para actualizar la cantidad total de monedas del usuario y `userRoomsDAO.updateTotalCoins(actualUserRoom.getId(), roomTotalCoins)` para actualizar la cantidad de monedas restantes en la habitación.

Finalmente, se recupera la información actualizada de la habitación utilizando `userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), actualUserRoom.getRoomId())` para asegurarse de que la información devuelta al usuario refleje el estado más reciente. Se devuelve un String utilizando `startService.createJson(username, updateUserRoom, "")` que indica que la operación se completó con éxito y que se puede continuar con el juego.

## GetKeyService

El servicio GetKeyService es una clase responsable de manejar la lógica relacionada con la interacción del usuario con las llaves en el juego. Se encarga de verificar si el usuario puede recoger una llave, si cuenta con suficientes monedas para hacerlo y actualizar la información del usuario una vez que obtiene una llave.

```
@Service 2 usages  ± crispise
public class GetKeyService {
    @Autowired 3 usages
    UserDAO userDao;
    @Autowired 1 usage
    DoorKeyDAO doorKeyDAO;
    @Autowired 4 usages
    StartService startService;
    @Autowired 2 usages
    UserRoomsDAO userRoomsDAO;

    public String checkClickInKey(String username) { 1 usage  ± crispise
        User user = userDao.getUserByUsername(username);
        if (user.getRoomId() == null) return "goToStart";
        UserRooms actualUserRoom = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), user.getRoomId());
        if (actualUserRoom.getDoorKeyId() == null) {
            return startService.createJson(username, actualUserRoom, errorMessage: "En esta habitación no hay llaves.");
        }
        DoorKey doorKey = doorKeyDAO.getKeyById(actualUserRoom.getDoorKeyId());
        if (user.getCoins() == null || user.getCoins() == 0) {
            return startService.createJson(username, actualUserRoom, errorMessage: "No tienes monedas");
        }
        if (user.getCoins() >= doorKey.getNeededCoins()) {
            userRoomsDAO.updateTotalKeys(user.getId(), actualUserRoom.getRoomId(), roomTotalKeys: null);
            return addKeyToUser(doorKey, user, actualUserRoom);
        } else {
            int coins = doorKey.getNeededCoins() - user.getCoins();
            return startService.createJson(username, actualUserRoom, errorMessage: "Te faltan " + coins + " monedas.");
        }
    }
}
```

```

private String addKeyToUser(DoorKey doorKey, User user, UserRooms actualUserRoom) { 1 usag
    String doorKeyId = String.valueOf(doorKey.getId());
    String userKeys = user.getIdKeys();
    if (userKeys == null || userKeys.isEmpty()) {
        userKeys = doorKeyId;
    } else {
        userKeys += "," + doorKeyId;
    }
    int actualUserCoins = user.getCoins() - doorKey.getNeededCoins();
    userDao.updateTotalUserKeys(user.getUsername(), userKeys);
    userDao.updateTotalUserCoins(user.getUsername(), actualUserCoins);
    return startService.createJson(user.getUsername(), actualUserRoom, errorMessage: "");
}

```

El método principal es **ckeckClickInKey**, el cual se encarga de procesar la lógica para determinar si el usuario puede recoger la llave en su habitación actual. En primer lugar, se obtiene el objeto User correspondiente al nombre de usuario proporcionado a través de userDao.getUserByUsername(username). Si el usuario no tiene una habitación asignada (user.getRoomId() == null), se devuelve un mensaje que redirige al usuario a la pantalla de inicio a través de return "goToStart".

Posteriormente, se consulta la habitación en la que se encuentra el usuario con userDao.getUserRoomByRoomIdAndUserId(user.getId(), user.getRoomId()).

Si la habitación actual no contiene ninguna llave (actualUserRoom.getDoorKeyId() == null), se devuelve un mensaje utilizando **startService.createJson**, informando al usuario que no hay llaves en su habitación. En caso contrario, se intenta obtener la información de la llave con doorKeyDAO.getKeyById(actualUserRoom.getDoorKeyId()).

Si el usuario no tiene monedas (user.getCoins() == null || user.getCoins() == 0), se le devuelve un mensaje que indica que no tiene monedas suficientes para recoger la llave. Si el usuario cuenta con suficientes monedas para adquirir la llave (user.getCoins() >= doorKey.getNeededCoins()), se procede a actualizar la información de la base de datos para que el usuario obtenga la llave, mediante la llamada al método **addKeyToUser**.

El método privado **addKeyToUser** se encarga de procesar la lógica para otorgar la llave al usuario y actualizar su información. Se convierte el identificador de la llave (doorKey.getId()) en un String para agregarla a la lista de llaves que el usuario tiene. Si el usuario no tiene ninguna llave previamente (userKeys == null || userKeys.isEmpty()), se le asigna directamente la nueva llave como su primer elemento. En caso contrario, la llave se agrega a la cadena de llaves separándola con comas. Luego, se calcula el nuevo total de monedas del usuario restando el costo de la llave (doorKey.getNeededCoins()) de sus monedas actuales. Finalmente, se actualizan los datos del usuario con

userDAO.updateTotalUserKeys(user.getUsername(), userKeys) y userDAO.updateTotalUserCoins(user.getUsername(), actualUserCoins). Esto garantiza que la información del usuario en la base de datos quede actualizada con la nueva llave obtenida y la cantidad de monedas restante.

## **OpenService**

El servicio OpenService es una clase encargada de gestionar la lógica relacionada con la interacción del usuario para abrir puertas en el juego. Su función principal es determinar si el usuario puede abrir una puerta en la dirección indicada y verificar si tiene las llaves necesarias para ello. Esto se realiza utilizando servicios y acceso a datos a través de UserDAO, UserRoomsDAO, NavService y StartService.

```
@Service 2 usages  ± crispie
public class OpenService {
    @Autowired 2 usages
    UserDAO userDAO;
    @Autowired 1 usage
    UserRoomsDAO userRoomsDAO;
    @Autowired 1 usage
    NavService navService;
    @Autowired 4 usages
    StartService startService;

    public String tryOpenDoor(String direction, String username) { 1 usage  ± crispie
        User user = userDAO.getUserByUsername(username);
        if (user.getRoomId() == null) return "goToStart";
        UserRooms actualUserRoom = userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(), user.getRoomId());
        Door door = null;
        if (actualUserRoom != null) {
            door = navService.findDoor(direction, actualUserRoom);
        }
        if (door == null) return startService.createJson(username, actualUserRoom, errorMessage: "No hay puerta");

        if (door.getState() == 0) {
            return treatCloseDoor(username, user, actualUserRoom, door);
        }
        return "error";
    }
}
```

```
private String treatCloseDoor(String username, User user, UserRooms actualUserRoom, Door door) { 1 usage  ± crispie
    if (user.getIdKeys() == null || user.getIdKeys().isEmpty())
        return startService.createJson(username, actualUserRoom, errorMessage: "No tienes ninguna llave");

    String doorKeyId = String.valueOf(door.getDoorKeyId());
    List<String> userIdKeys = Arrays.asList(user.getIdKeys().split( regex: "\\" ));
    boolean hasKey = checkUserKeys(userIdKeys, doorKeyId);
    if (hasKey) {
        userDAO.updateOpenDoors(username, String.valueOf(door.getId()));
        return startService.createJson(username, actualUserRoom, errorMessage: "");
    } else {
        return startService.createJson(username, actualUserRoom, errorMessage: "La llave correcta es la Llave " + doorKeyId);
    }
}

private static boolean checkUserKeys(List<String> userIdKeys, String doorKeyId) { 1 usage  ± crispie
    boolean hasKey = false;
    for (String idKey : userIdKeys) {
        if (idKey.equals(doorKeyId)) hasKey = true;
    }
    return hasKey;
}
```

El método principal es ***tryOpenDoor***, el cual se ejecuta cuando el usuario intenta abrir una puerta en una dirección específica. En primer lugar, se obtiene el objeto User correspondiente al nombre de usuario proporcionado utilizando userDAO.getUserByUsername(username). Si el usuario no tiene una habitación asignada (user.getRoomId() == null), la función devuelve un mensaje redirigiendo al usuario al inicio del juego mediante return "goToStart".

Luego, se recupera la información de la habitación actual utilizando userRoomsDAO.getUserRoomByRoomIdAndUserId(user.getId(),user.getRoomId()). Si la información de la habitación es válida, se intenta localizar la puerta en la dirección especificada utilizando el método ***navService.findDoor(direction, actualUserRoom)***. Si no se encuentra ninguna puerta en la dirección indicada (door == null), se devuelve un mensaje informando que no existe ninguna puerta en esa dirección con ***startService.createJson***.

En caso de encontrar una puerta, el método verifica su estado. Si la puerta está cerrada (door.getState() == 0), se procede a llamar al método ***treatCloseDoor***, que se encarga de manejar la lógica para tratar la situación en la que el usuario intenta abrir una puerta cerrada. Si la puerta ya está abierta, se devuelve un mensaje de error mediante return "error".

El método ***treatCloseDoor*** se encarga de verificar si el usuario tiene las llaves necesarias para abrir la puerta. Primero, comprueba si el usuario tiene algún conjunto de llaves (user.getIdKeys()). Si no tiene ninguna llave, se devuelve un mensaje indicando que el usuario no tiene ninguna llave mediante ***startService.createJson***. Si el usuario tiene llaves, se extrae la identificación de la llave correspondiente para la puerta a través de door.getDoorKeyId(). A continuación, se verifica si el usuario posee la llave correcta mediante el método ***checkUserKeys***.

El método ***checkUserKeys*** es un simple bucle que recorre las llaves del usuario para determinar si contiene la llave correspondiente a la puerta. Si el usuario tiene la llave correcta, se actualiza el estado de las puertas abiertas para ese usuario mediante la llamada a userDAO.updateOpenDoors(username, String.valueOf(door.getId())) y se devuelve una respuesta de éxito mediante ***startService.createJson***. Si el usuario no tiene la llave correcta, se devuelve un mensaje informando que el jugador necesita la llave específica para abrir la puerta.

## ResetService

El servicio ResetService es una clase encargada de borrar el progreso de un usuario en el juego, reiniciando su estado para que pueda comenzar de nuevo si así lo desea.

```
@Service 3 usages ▲ crispise
public class ResetService {
    @Autowired 1 usage
    MapDAO mapDAO;
    @Autowired 2 usages
    UserDAO userDAO;
    @Autowired 1 usage
    RoomDAO roomDAO;
    @Autowired no usages
    StartService startService;
    @Autowired 1 usage
    UserRoomsDAO userRoomsDAO;

    public void resetGame(String username) { 2 usages ▲ crispise
        User user = userDAO.getUserByUsername(username);
        if (user.getRoomId() == null) return;
        Room actualRoom = roomDAO.getRoomById(user.getRoomId());
        Map map = mapDAO.getMapById(actualRoom.getMapId());
        userDAO.resetUser(roomId: null, initialCoins: 0, gameTime: null, idKeys: null, openDoors: null, username, mapName: null);
        userRoomsDAO.deleteUserRoomsByUserIdAndMapId(user.getId(), map.getId());
    }
}
```

El método principal es **resetGame**, que recibe como parámetro el nombre de usuario (username) que desea reiniciar su progreso. En primer lugar, se recupera el objeto User correspondiente al usuario mediante userDAO.getUserByUsername(username). Si el usuario no tiene una habitación asignada (user.getRoomId() == null), el método simplemente termina su ejecución, ya que no es necesario realizar el proceso de reinicio.

Luego, se obtiene la información de la habitación actual del usuario utilizando roomDAO.getRoomById(user.getRoomId()). Con la información de la habitación, se accede al objeto Map correspondiente a través de mapDAO.getMapById(actualRoom.getMapId()).

A continuación, se restablece el estado del usuario mediante la llamada a userDAO.resetUser. Este método se encarga de borrar la información de progreso del usuario, estableciendo valores nulos o por defecto para las propiedades relevantes, como la ubicación, el estado de progreso, las llaves y otros datos.

Por último, se eliminan todas las relaciones de las habitaciones asignadas al usuario en el mapa utilizando userRoomsDAO.deleteUserRoomsByUserIdAndMapId (user.getId(), map.getId()). Esto borra la información de progreso específica del usuario en ese mapa, garantizando que la información de las habitaciones se restablezca por completo y que el usuario pueda empezar de nuevo.

## **ScoresService**

El servicio ScoresService es una clase encargada de manejar la lógica relacionada con la puntuación de los usuarios, actualizandola al finalizar una partida, calculando el tiempo de juego y permitiendo obtener el historial de puntuaciones de los otros jugadores.

```
@Service 2 usages  ↳ crispise
public class ScoresService {
    @Autowired 1 usage
    UserDAO userDAO;
    @Autowired 2 usages
    ScoreDAO scoreDAO;
    @Autowired 1 usage
    ResetService resetService;

    public void updateScores(String username, String gameComent) { 1 usage  ↳ crispise
        User user = userDAO.getUserByUsername(username);
        long gameTime = calculateFinalTime(user);
        scoreDAO.insertScore(user, gameTime, gameComent);
        resetService.resetGame(username);
    }

    private long calculateFinalTime(User user) { 1 usage  ↳ crispise
        long currentTime = System.currentTimeMillis(); // Tiempo actual en milisegundos
        long elapsedTime = currentTime - user.getGameTime();
        return elapsedTime / 1000;
    }

    public List<Score> obtainScores(String username) { 2 usages  ↳ crispise
        List<Score> usersScore = scoreDAO.getAllScores();
        return usersScore;
    }
}
```

El método principal es **updateScores**, que recibe como parámetros el nombre de usuario (username) y un comentario de la partida (gameComent). En primer lugar, se recupera el objeto User correspondiente al nombre de usuario utilizando el método userDAO.getUserByUsername(username). Luego, se calcula el tiempo total que el usuario dedicó a la partida mediante la llamada al método **calculateFinalTime(user)**. Este cálculo determina la diferencia entre el tiempo actual en milisegundos (System.currentTimeMillis()) y el tiempo almacenado previamente en la propiedad user.getGameTime(). El tiempo resultante se convierte de milisegundos a segundos dividiendo por 1000.

Con el tiempo de juego calculado, el método registra la puntuación utilizando el método scoreDAO.insertScore. Posteriormente, el método llama a

resetService.resetGame(username) para reiniciar el progreso del usuario. Esto garantiza que después de completar una partida, el usuario vuelve a un estado inicial limpio, listo para jugar de nuevo.

Finalmente, el método obtainScores permite recuperar todas las puntuaciones almacenadas en el sistema. Este método simplemente llama al método scoreDAO.getAllScores(), el cual devuelve una lista de objetos Score con la información de las puntuaciones registradas en la base de datos.

## 3.5 Controladores (controllers)

Los controladores actúan como intermediarios entre el usuario y la lógica de negocio, gestionando las solicitudes del cliente y devolviendo respuestas adecuadas a través de vistas o datos procesados.

### RegisterController

La clase RegisterController es responsable de manejar las solicitudes relacionadas con el registro de nuevos usuarios en la aplicación. Su función principal es gestionar tanto la visualización del formulario de registro como el procesamiento de los datos enviados para crear una nueva cuenta. Para llevar a cabo estas tareas, este controlador colabora con el servicio RegisterService, encargado de implementar la lógica de negocio asociada al registro de usuarios.

```
@Controller no usages ▲ crispise*
public class RegisterController {
    @Autowired 1 usage
    RegisterService registerService;

    ▶ Rename usages
    @GetMapping("/register") ▲ crispise *
    public String getRegister() { return "register"; }

    @PostMapping("/register") no usages ▲ crispise
    public String register(Model model, @RequestParam String name, @RequestParam String username, @RequestParam String password) {
        try {
            registerService.registerUser(name, username, password);
            model.addAttribute( attributeName: "messageType", attributeValue: "success");
            model.addAttribute( attributeName: "message", attributeValue: "El registro se ha realizado correctamente");
        } catch (UserExistsException userException) {
            model.addAttribute( attributeName: "messageType", attributeValue: "errorUserExists");
            model.addAttribute( attributeName: "message", attributeValue: "El username ya existe.");
        } catch (PasswordTooShortException passwordTooShortException) {
            model.addAttribute( attributeName: "messageType", attributeValue: "errorPassword");
            model.addAttribute( attributeName: "message", attributeValue: "La contraseña tiene que tener un mínimo de 5 caracteres.");
        } catch (NameTooShortException nameTooShortException) {
            model.addAttribute( attributeName: "messageType", attributeValue: "errorName");
            model.addAttribute( attributeName: "message", attributeValue: "El nombre es demasiado corto, tiene que tener 6 caracteres.");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return "register";
    }
}
```

- **Método getRegister**

El método getRegister está vinculado al endpoint /register mediante la anotación @GetMapping. Este método se ejecuta cuando un usuario accede a la página de registro. Su única responsabilidad es devolver la vista register, que corresponde al formulario HTML donde los usuarios pueden introducir sus datos para registrarse.

- **Método register**

El método register está vinculado al mismo endpoint /register, pero utiliza la anotación @PostMapping, lo que significa que procesa solicitudes HTTP POST. Este método recibe tres parámetros del formulario de registro enviado por el usuario: el nombre (name), el nombre de usuario (username) y la contraseña (password).

Primero, el método intenta registrar al usuario utilizando el servicio RegisterService, llamando al método **registerUser(name, username, password)**. Si el registro se realiza correctamente, se añade un mensaje de éxito al modelo para mostrar al usuario una notificación de que el proceso se completó sin problemas. Finalmente, el método siempre devuelve la vista register, lo que garantiza que el usuario permanezca en la página de registro para ver los mensajes correspondientes, ya sea de éxito o error.

## **LoginController**

La clase LoginController gestiona las solicitudes relacionadas con el inicio de sesión de los usuarios en la aplicación.

```
@Controller no usages ▲ crispile
public class LoginController {

    @Autowired 1 usage
    LoginService loginService;

    @GetMapping("/login") no usages ▲ crispile
    public String getLogin() { return "login"; }

    @PostMapping("/login") no usages ▲ crispile
    public String login(Model m, HttpSession session, @RequestParam String username, @RequestParam String password) throws NoSuchAlgorithmException {
        User user = loginService.checkUser(username, password);
        if (user != null) {
            session.setAttribute("user", username);
            return "redirect:/start";
        } else {
            m.addAttribute(attributeName: "errorMessage", attributeValue: "Usuario i/o contraseña incorrectos");
            return "login";
        }
    }
}
```

- **Método getLogin**

El método getLogin está vinculado al endpoint /login mediante la anotación @GetMapping. Este método es responsable de responder a las

solicitudes HTTP GET dirigidas al formulario de inicio de sesión. Su única función es devolver la vista login, que corresponde a la página HTML donde los usuarios pueden introducir sus credenciales (nombre de usuario y contraseña) para iniciar sesión.

- **Método login**

El método login está asociado al mismo endpoint /login, pero utiliza la anotación @PostMapping, lo que indica que procesa solicitudes HTTP POST.

Su principal tarea es verificar las credenciales proporcionadas llamando al método **checkUser(username, password)** del servicio LoginService. Si las credenciales son válidas, se establece el nombre de usuario en la sesión (session.setAttribute("user", username)) y se redirige al endpoint /start, que representa el inicio del juego.

Si las credenciales no son válidas, el método añade un mensaje de error al modelo (m.addAttribute("errorMessage", "Usuario i/o contraseña incorrectos")) para notificar al usuario sobre el problema y devuelve la vista login, permitiendo que el usuario intente ingresar nuevamente.

## **StartController**

La clase StartController es un controlador diseñado para gestionar las solicitudes relacionadas con la selección e inicio de mapas dentro de la aplicación. Este controlador interactúa con el servicio StartService, que maneja la lógica de negocio asociada a los mapas y la configuración inicial del juego.

```
@Controller no usages ▲ crispise
public class StartController {
    @Autowired 2 usages
    StartService startService;

    @GetMapping("/start") no usages ▲ crispise
    public String startPage(Model m) {
        List<Map> maps = startService.getAllMaps();
        m.addAttribute(attributeName: "maps", maps);
        return "start";
    }

    @PostMapping("/start") no usages ▲ crispise
    public String selectMap(Model m, @RequestParam String mapId, HttpSession session) {
        String username = (String) session.getAttribute(s: "user");
        String jsonToSend = startService.getFirstJson(mapId, username);
        m.addAttribute(attributeName: "jsonInfo", jsonToSend);
        return "game";
    }
}
```

- **Método startPage**

El método startPage está vinculado al endpoint /start mediante la anotación @GetMapping. Cuando se accede a esta ruta, el método recopila todos los mapas disponibles en la base de datos mediante el método `getAllMaps()` del servicio **StartService**. Estos mapas se agregan al modelo para que puedan ser renderizados en la vista start, permitiendo al jugador explorar las opciones disponibles antes de seleccionar un mapa.

- **Método selectMap**

El método selectMap está vinculado al mismo endpoint pero para solicitudes POST. Procesa la selección de un mapa realizada por el usuario. Recupera el nombre de usuario de la sesión activa y utiliza el servicio **StartService** para generar un JSON que contiene el estado inicial del juego en función del mapa elegido. Este JSON, que incluye información clave como el progreso del jugador y la configuración del mapa, se agrega al modelo y se pasa a la vista game.

## **NavController**

La clase NavController es un controlador encargado de gestionar la navegación dentro del juego. Su propósito principal es procesar las solicitudes relacionadas con los movimientos del jugador entre habitaciones, verificando la dirección seleccionada y devolviendo el estado actualizado del juego.

```
@Controller no usages ▾ crispise
public class NavController {
    @Autowired 1 usage
    NavService navService;

    @GetMapping("/nav") no usages ▾ crispise
    public String move(Model m, @RequestParam("dir") String direction, HttpSession session) {
        String username = (String) session.getAttribute(s: "user");
        String jsonToSend = navService.trySelectedDirection(direction, username);
        if (jsonToSend.equals("goToStart")){
            return "redirect:/start";
        }else {
            m.addAttribute(attributeName: "jsonInfo", jsonToSend);
            return "game";
        }
    }
}
```

El método move esta asociado al endpoint /nav mediante la anotación @GetMapping. Este método recibe dos parámetros principales: la dirección en

la que el jugador desea moverse (dir) y la sesión actual del usuario, de donde se extrae el nombre de usuario activo.

Al invocar el método **trySelectedDirection** del servicio NavService, el controlador evalúa si el movimiento seleccionado es válido y si el jugador puede proceder en la dirección indicada. Dependiendo de la respuesta generada por el servicio:

1. Si el resultado es "goToStart", significa que el usuario no puede continuar porque no tiene una habitación inicial asignada. En este caso, el método redirige al usuario al endpoint /start para reiniciar su progreso.
2. Si la respuesta incluye un JSON con información sobre el estado del juego tras el movimiento, este se agrega al modelo como atributo (jsonInfo), permitiendo a la vista game renderizar el estado actualizado del jugador, incluyendo su nueva posición y cualquier cambio relevante en el entorno.

## GetCoinController

La clase GetCoinController es un controlador diseñado para gestionar las solicitudes relacionadas con la recolección de monedas por parte del jugador dentro del juego. Este controlador se encarga de procesar la acción de recoger una moneda en la habitación actual del usuario y devolver la información actualizada del estado del juego a la vista.

```
@Controller no usages  ± crispie
public class GetCoinController {
    @Autowired 1 usage
    GetCoinService getCoinService;

    @GetMapping("/getcoin") no usages  ± crispie
    public String getCoin(Model m, HttpSession session) {
        String username = (String) session.getAttribute("user");
        String jsonToSend = getCoinService.addCoinToUser(username);
        if (jsonToSend.equals("goToStart")){
            return "redirect:/start";
        }else {
            m.addAttribute(attributeName: "jsonInfo", jsonToSend);
            return "game";
        }
    }
}
```

El método getCoin está asociado al endpoint /getcoin mediante la anotación @GetMapping. Este método se encarga de gestionar la solicitud de recolección de monedas por parte del jugador. Primero, recupera el nombre de usuario activo de la sesión para asociar la acción al jugador correspondiente.

Luego, invoca el método **addCoinToUser** del servicio GetCoinService, que contiene la lógica para determinar si el jugador puede recoger monedas en la

habitación actual y devuelve un JSON con el resultado de la operación. Según la respuesta del servicio:

1. Si la respuesta es "goToStart", indica que el jugador no tiene una habitación asignada o que el progreso debe reiniciarse. En este caso, el método redirige al jugador al endpoint /start.
2. Si la respuesta contiene un JSON válido, este se añade al modelo como atributo (jsonInfo) para que la vista game pueda renderizar la información actualizada, reflejando los cambios en el número de monedas del jugador y el estado de la habitación.

## **GetKeyController**

El controlador GetKeyController es el encargado de gestionar las solicitudes relacionadas con la acción de recoger llaves en el juego. Se encarga de llamar al servicio GetKeyService para verificar si el jugador puede obtener una llave y procesa la respuesta para determinar el siguiente paso:

```
@Controller no usages -> crispise
public class GetKeyController {
    @Autowired 1 usage
    GetKeyService getKeyService;

    @GetMapping("/getkey") no usages -> crispise
    public String getKey(Model m, HttpSession session) {
        String username = (String) session.getAttribute("s: user");
        String jsonToSend = getKeyService.checkClickInKey(username);
        if (jsonToSend.equals("goToStart")){
            return "redirect:/start";
        }else {
            m.addAttribute(attributeName: "jsonInfo", jsonToSend);
            return "game";
        }
    }
}
```

El método getKey está asociado al endpoint /getkey mediante la anotación @GetMapping. Este método se encarga de gestionar la solicitud para que el jugador intente recoger una llave en la habitación actual. Primero, recupera el nombre de usuario activo de la sesión para asegurarse de que la acción esté asociada al jugador correcto.

A continuación, invoca el método **ckeckClickInKey** del servicio GetKeyService, que contiene la lógica para verificar si el jugador puede recoger una llave. Dependiendo de la respuesta del servicio:

1. Si la respuesta es "goToStart", indica que el jugador no tiene una habitación asignada o que el progreso debe reiniciarse. En este caso, el método redirige al jugador al endpoint /start.
2. Si la respuesta incluye un JSON válido, este se añade al modelo como atributo (jsonInfo) para que la vista game pueda actualizar la interfaz con la información relevante sobre el intento de recoger la llave y su resultado.

## **OpenController**

El controlador OpenController es responsable de manejar las solicitudes relacionadas con la acción de intentar abrir puertas en el juego.

```
@Controller no usages ▲ crispise
public class OpenController {
    @Autowired 1usage
    OpenService openService;
    @GetMapping("/open") no usages ▲ crispise
    public String openDoor(Model m, HttpSession session, @RequestParam("dir") String direction) {
        String username = (String) session.getAttribute(s: "user");
        String jsonToSend = openService.tryOpenDoor(direction, username);
        if (jsonToSend.equals("goToStart")){
            return "redirect:/start";
        }else {
            m.addAttribute(attributeName: "jsonInfo", jsonToSend);
            return "game";
        }
    }
}
```

El método openDoor está asociado al endpoint /open mediante la anotación @GetMapping. Este método se encarga de gestionar la solicitud de apertura de puertas por parte del jugador. Primero, recupera el nombre de usuario activo de la sesión para identificar al jugador correspondiente. Luego, invoca el método **tryOpenDoor** del servicio OpenService, el cual contiene la lógica para evaluar si el jugador puede abrir la puerta en la dirección seleccionada y devuelve un JSON con el resultado de la operación. Según la respuesta del servicio:

1. Si la respuesta es "goToStart", indica que el jugador no puede avanzar y debe regresar al inicio. En este caso, el método redirige al jugador al endpoint /start.
2. Si la respuesta contiene un JSON válido, este se añade al modelo como atributo (jsonInfo) para que la vista game pueda actualizar la información del estado del juego, reflejando los cambios en el progreso del jugador y el entorno actual.

## ResetController

El ResetController es un controlador encargado de gestionar las acciones relacionadas con el reinicio del progreso del jugador y la sesión de usuario.

```
@Controller no usages ± crispie
public class ResetController {
    @Autowired 1 usage
    ResetService resetService;

    @GetMapping("/reset") no usages ± crispie
    public String reset(Model m, HttpSession session) {
        String username = (String) session.getAttribute("user");
        resetService.resetGame(username);
        session.invalidate();
        return "redirect:/login";
    }

    @PostMapping("/resetGame") no usages ± crispie
    public String restartGame(Model m, HttpSession session) {
        return "redirect:/start";
    }

    @PostMapping("/closeSession") no usages ± crispie
    public String closeSession(Model m, HttpSession session) {
        session.invalidate();
        return "redirect:/login";
    }
}
```

- **Método reset**

Está asociado al endpoint /reset mediante la anotación @GetMapping. Este método se encarga de gestionar la solicitud para reiniciar el progreso del jugador y cerrar la sesión. Primero, recupera el nombre de usuario activo de la sesión para identificar al jugador correspondiente. Luego, invoca el método **resetGame** del servicio ResetService, el cual reinicia el progreso del jugador. Después de realizar el reinicio, invalida la sesión y redirige al jugador al endpoint /login.

- **Método restartGame**

Está asociado al endpoint /resetGame mediante la anotación @PostMapping. Su propósito es redirigir al jugador al inicio del juego, es decir, al endpoint /start. No realiza ninguna lógica adicional más allá de la redirección.

- **Método closeSession**

Está asociado al endpoint /closeSession mediante la anotación @PostMapping. Su funcionalidad es cerrar la sesión del jugador al invalidar los datos de la sesión activa y redirigir al jugador al endpoint /login. Esto permite al jugador salir de forma segura y regresar a la página de inicio de sesión.

## ScoresController

El ScoresController es un controlador responsable de gestionar las solicitudes relacionadas con la visualización y actualización de las puntuaciones de los jugadores.

```
@Controller no usages ▾ crispise
public class ScoresController {
    @Autowired 3 usages
    ScoresService scoresService;

    @GetMapping("/scores") no usages ▾ crispise
    public String getScores(Model m, HttpSession session) {
        String username = (String) session.getAttribute("user");
        List<Score> usersScores = scoresService.obtainScores(username);
        m.addAttribute(attributeName: "usersScore", usersScores);
        return "scores";
    }

    @PostMapping("/scores") no usages ▾ crispise
    public String getFormScores(Model m, HttpSession session, @RequestParam String gameComent) {
        String username = (String) session.getAttribute("user");
        scoresService.updateScores(username, gameComent);
        List<Score> usersScores = scoresService.obtainScores(username);
        m.addAttribute(attributeName: "usersScore", usersScores);
        return "scores";
    }
}
```

- **Método getScores**

Está asociado al endpoint /scores mediante la anotación @GetMapping. Este método se ejecuta cuando el jugador accede a la página de puntuaciones. Recupera el nombre de usuario activo de la sesión y utiliza el servicio ScoresService para obtener la lista de puntuaciones del usuario mediante el método **obtainScores**. Luego, agrega esta información al modelo como atributo (usersScore) para que la vista scores pueda mostrar las puntuaciones del jugador.

- **Método getFormScores**

También está asociado al endpoint /scores, pero utiliza la anotación @PostMapping. Este método procesa las solicitudes enviadas por el formulario de puntuaciones. Recupera el nombre de usuario activo de la sesión y el comentario del juego enviado por el jugador como parámetro (gameComent). Luego, invoca el método **updateScores** del servicio ScoresService, que actualiza la puntuación del jugador y reinicia el progreso según la lógica implementada. Después de actualizar la puntuación, vuelve a obtener la lista de puntuaciones actualizadas mediante **obtainScores** y la agrega al modelo como atributo (usersScore). Finalmente, redirige la vista scores con la información actualizada para que el jugador pueda visualizar sus puntuaciones.

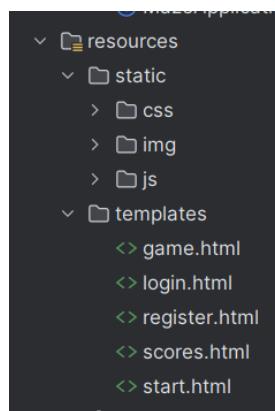
## 3.6 Filters

### LoginInterceptor

```
@Component 2 usages  ↳ crispise
public class LoginInterceptor implements HandlerInterceptor {
    @Override no usages ↳ crispise
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        HttpSession session = request.getSession();
        String user = (String) session.getAttribute("user");
        if (user == null) {
            response.sendRedirect("/login");
            return false;
        }
        return true;
    }
}
```

El LoginInterceptor es un interceptor que implementa la interfaz HandlerInterceptor. Su función principal es verificar si el usuario tiene una sesión activa antes de permitir el acceso a ciertos endpoints. En el método preHandle, se comprueba si la sesión contiene un atributo user. Si el usuario no está autenticado (es decir, no existe el atributo user en la sesión), el interceptor redirige la solicitud al endpoint /login y detiene el procesamiento de la solicitud. Si el usuario está autenticado, la solicitud continúa normalmente. Esto garantiza que solo los usuarios autenticados puedan acceder a las partes protegidas de la aplicación.

## 4. ESTRUCTURA DEL PROGRAMA: FRONT-END



En la estructura del programa front-end, que corresponde a la parte más visual, encontramos diferentes tipos de archivos clasificados en carpetas según su tipo. Esto permite que a la hora de modificar alguno se más sencillo localizarlo.

**Archivos CSS (css):** Incluyen el diseño web de cada sección de la aplicación (gallery.css, login&register.css, etc.).

**Imágenes (img):** Contienen elementos gráficos como las monedas, llaves, etc.

**JavaScript (js):** Permiten la manipulación de la interfaz del juego.

**Templates:** Contiene las vistas (archivos HTML) usadas para renderizar la información en el lado del cliente.

### **4.1 Templates**

La carpeta templates es la ubicación central de todas las vistas que se renderizan dinámicamente para mostrar la interfaz de usuario de la aplicación. Estas plantillas se utilizan para proporcionar una experiencia interactiva, integrando lógica y datos enviados por los controladores.

#### **Start.html**

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="UTF-8">
    <title>Start-Maze</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-E7yJ1WZU1dCtPzXq7gjgHnGxWZ51ZD0l1wqEgR33K1QWQcDfRrOvqYQ==" crossorigin="anonymous" referrerpolicy="no-referrer">
    <link rel="stylesheet" href="/css/login&register.css">
</head>
<body>
    <div class="container w-75">
        <form method="post" action="/start" class="border rounded p-4 shadow">
            <h1>Bienvenido</h1>
            <div class="form-group">
                <label for="mapOption">Selecciona el nivel</label>
                <select id="mapOption" name="mapId" class="form-select my-1 text-center">
                    <option th:each="map : ${maps}" th:value="${map.id}" th:text="${map.mapName}"></option>
                </select>
            <div class="d-flex justify-content-between mt-4">
                <input type="submit" value="Enviar" class="btn">
            </div>
        </div>
    </form>
</div>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-YvpcrYf0tY3l" crossorigin="anonymous" referrerpolicy="no-referrer">
</body>
</html>
```

El archivo login.html es una plantilla Thymeleaf que proporciona la interfaz para el inicio de sesión del usuario en la aplicación. Contiene un formulario que permite al usuario ingresar su nombre de usuario y contraseña. Este formulario se envía mediante el método POST al servidor en el endpoint /login. La interfaz incluye campos de texto para el nombre de usuario y la contraseña, además de un botón de inicio de sesión.

Si el inicio de sesión falla, se utiliza un contenedor para mostrar mensajes de error dinámicos con la información devuelta desde el servidor (errorMessage) gracias a Thymeleaf.

## Game.html

```
<body class="d-flex justify-content-center align-items-center bg-dark p-4">
<script th:inline="javascript">
    var jsonInfo = /*[$jsonInfo]*/ {};
</script>
<div class="game-container d-flex justify-content-around align-items-center w-75 p-2">
    <a href="/reset" class="btn btn-warning btn-lg mb-3" id="resetButton">Reiniciar Juego</a>
    <div class="d-flex flex-column justify-content-center align-items-center">
        <h1 class="game-title text-center text-light mb-1">Maze Game</h1>
        <div id="infoGame" class="info-bar w-100 d-flex justify-content-between p-3"></div>
        <div class="canvas-wrapper">
            <canvas id="myCanvas" width="520" height="520"></canvas>
        </div>
        <div id="errorContainer" class="error-message text-danger"></div></div>
        <div id="crossContainer" class="d-flex justify-content-center">
            
        </div>
    </div>
    <div class="overlay" id="overlay"></div>
    <div class="windowFinalScore">
        <div class="windowFinalScore">
            <div class="containerFormScore">
                <h2 class="text-center">¡Felicitaciones!</h2>
                <p class="text-center">Has completado el laberinto con éxito.</p>
                <p class="text-center">Comparte tu experiencia y guarda tu partida:</p>
                <form method="post" action="/scores" class="d-flex flex-column align-items-center">
                    <div class="mb-3 w-100">
                        <input type="text" class="form-control" name="gameComment" placeholder="Escribe tu comentario aquí...">
                    </div>
                    <button type="submit" class="btn btn-primary btn-lg w-25">Guardar</button>
                </form>
            </div>
        </div>
    </div>
</div>
```

El archivo game.html es una plantilla Thymeleaf para la interfaz principal del juego. Incluye un área central donde se renderiza el juego a través de un elemento <canvas> con dimensiones de 520x520 píxeles. También contiene un contenedor con información del juego, como la barra de estado (infoGame) y un contenedor de error para mostrar mensajes relevantes al jugador.

El código tiene un formulario modal que se despliega al completar el laberinto con éxito, permitiendo al usuario enviar un comentario mediante un formulario con el método POST al servidor en el endpoint /scores. Thymeleaf se usa para pasar datos dinámicos entre el servidor y el cliente con la línea de script th:inline="javascript". Aquí se inicializa una variable jsonInfo que contiene

datos específicos generados por el servidor, lo que permite actualizar dinámicamente la información en el lado del cliente.

También contiene un botón "Reiniciar Juego" (/reset) para reiniciar el progreso del jugador. Todo el comportamiento dinámico depende de JavaScript externo (maze.js) y la lógica generada por Thymeleaf a través de las respuestas del servidor.

## Scores.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="UTF-8">
    <title>Start-Maze</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
    <link rel="stylesheet" href="/css/scores.css">
</head>
<body>
    <div class="m-4 w-100 d-flex justify-content-between">
        <div class="m-4">
            <form method="post" action="/resetGame" class="mt-4">
                <button type="submit" class="btn btn-warning">Volver a Jugar</button>
            </form>
        </div>
        <div class="m-4">
            <form method="post" action="/closeSession" class="mt-4">
                <button type="submit" class="btn btn-warning">Cerrar Sesión</button>
            </form>
        </div>
    </div>
    <div class="w-100 d-flex justify-content-center align-items-center">
        <table class="table table-bordered score-table">
            <thead class="table-dark">
                <tr>
                    <th>Nombre de Usuario</th>
                    <th>Nombre del Mapa</th>
                    <th>Tiempo de Juego</th>
                    <th>Comentarios</th>
                </tr>
            </thead>
            <tbody class="scrollable-table">
                <tr th:each="score : ${usersScore}">
                    <td th:text="${score.userName}"></td>
                    <td th:text="${score.mapName}"></td>
                    <td class="game-time" th:text="${score.gameTime}"></td>
                    <td th:text="${score.comentary}"></td>
                </tr>
            </tbody>
        </table>
    </div>
    <script src="/js/scores.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-KHS�yfDQ3JM4GQkRdZU26nGPFtH1o4mR8ftJHhGcPv33NzO2g5" crossorigin="anonymous"></script>
</body>
</html>
```

El archivo scores.html es una plantilla Thymeleaf que presenta la interfaz para mostrar la tabla de puntuaciones del juego. La página contiene dos formularios con botones: uno para volver a jugar, enviando una solicitud al endpoint /resetGame, y otro para cerrar sesión, enviando una solicitud al endpoint /closeSession.

El núcleo de la página es una tabla dinámica que muestra la información de puntuaciones utilizando Thymeleaf. A través de la directiva th:each, se recorren los elementos de la lista usersScore, la cual se pasa desde el servidor. Cada fila de la tabla muestra el nombre de usuario, el nombre del mapa, el tiempo de juego y cualquier comentario que el usuario haya dejado, todo extraído de la lista generada por el servidor.

Además, se incluye un script scores.js, que se ejecuta para formatear el tiempo obtenido en segundos a horas, minutos y segundos.

## 4.2 JavaScript

### Maze.js

Este archivo gestiona la lógica principal para renderizar y controlar la experiencia de juego dentro del laberinto utilizando un lienzo (canvas). Además el archivo se encarga de cargar recursos gráficos, procesar las interacciones del jugador con elementos interactivos (monedas, llaves, puertas) y realizar animaciones como el movimiento del personaje.

### Funciones y Componentes Claves

#### 1. Cargar Elementos Gráficos y Recursos:

- **loadAllAssets(callback):** Se asegura de que las imágenes necesarias para el juego estén cargadas antes de comenzar.
- **loadedCharacterImage, loadedKeyImage, loadedCoinImage:** Cargan imágenes para el personaje, las llaves y monedas en el juego.

#### 2. Configuración del Lienzo y Canvas:

- canvas y ctx configuran el lienzo principal donde se renderizarán los elementos del juego.
- bufferCanvas y su contexto (bufferCtx) permiten hacer un renderizado en segundo plano antes de copiar la imagen al lienzo principal.

#### 3. Obtención de Información:

- obtainInfo(): Comprueba si la variable global jsonInfo tiene datos disponibles. Si es así, convierte esos datos en un objeto JSON con JSON.parse(jsonInfo) y llama a la función loadRoom(roomInfo) para procesar la información obtenida.

#### 4. Dibujo de Elementos en el Canvas:

- **drawWall():** Dibuja las paredes del laberinto.
- **drawUserInfo(roomInfo):** Muestra información del usuario, como nombre del mapa, llaves y monedas.
- **drawDoor(state, direction):** Dibuja puertas dependiendo de su estado (cerrado/abierto).
- **drawCoins(roomInfo) y drawKey(roomInfo):** Dibuja las monedas y la llave en posiciones específicas.

#### 5. Manejo de Interacciones del Usuario:

- Detección de clics en monedas y llaves:
  - **detectCoinClick(event)** y **detectKeyClick(event):** Detectan clics en la posición de las monedas o la llave para ejecutar acciones.
- Detección de clics en puertas:
  - **detectCloseDoorClick(event):** Permite al jugador interactuar con puertas.

#### 6. Animaciones y Movimiento del Jugador:

- **animateMovement(direction, callback):** Realiza animaciones de movimiento utilizando imágenes de dirección (loadedImages). Se ejecuta con requestAnimationFrame para una animación fluida.

#### 7. Lógica de Navegación:

- Se detectan las flechas y clics en áreas específicas para redirigir al jugador a otra ubicación (window.location.href = "/nav?dir=\${direction}").

#### 8. Configuración de Coordenadas y Direcciones:

- **getMovementCoordinates(direction):** Devuelve las coordenadas de inicio y destino para la animación según la dirección elegida.

#### 9. Ventana de Información, modales y final:

- Se utiliza la ventana modal para mostrar información de puntuación después de ganar la partida.
- **drawFinal():** Renderiza la imagen de victoria cuando el jugador alcanza el objetivo final.