
Proyecto de desarrollo de aplicaciones multiplataforma Juego Bullet Hell Arcade

CICLO FORMATIVO DE GRADO SUPERIOR
Desarrollo de Aplicaciones Multiplataforma (IFCS02)

Curso 2022-23

Autor/a/es:

Mario González Resa

Tutor/a:

José Luis González Sánchez

Departamento de Informática y Comunicaciones
I.E.S. Luis Vives

ÍNDICE

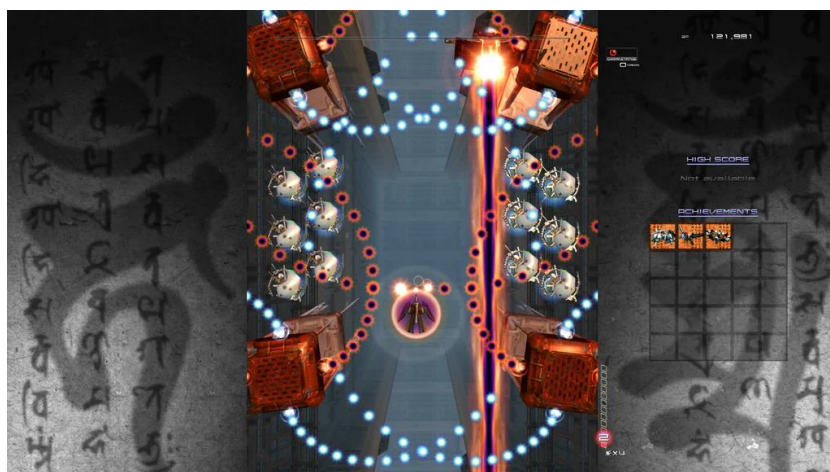
1	Introducción.....	4
1.1	Descripción.....	4
2	Requisitos.....	5
2.1	Requisitos Funcionales.....	5
2.2	Requisitos No Funcionales	5
2.3	Requisitos de Información	5
3	Análisis de mercado.....	7
4	Análisis tecnológico	9
4.1	Backend.....	9
4.1.1	Gestión de la información	9
4.1.2	Servicio API-REST	10
4.2	Frontend.....	11
4.2.1	Motor de desarrollo.....	11
4.3	Herramientas de diseño.....	12
4.4	Herramientas adicionales	13
5	Backend-Explicación	14
5.1	Modelos	14
5.2	DTOs	14
5.2.1	Usuarios	14
5.2.2	Puntuaciones	15
5.3	Mapeadores	15
5.4	Repositorios	16
5.5	Validadores	16
5.6	Configuración	17
5.6.1	APIConfig	17
5.6.2	LoadSchema.....	17
5.6.3	Archivo de propiedades.....	17
5.7	Seguridad	17
5.7.1	Password.....	17
5.7.2	JWT	18
5.7.3	SecurityConfig.....	18
5.7.4	SSL.....	18
5.8	Servicios	18
5.8.1	UserService	18
5.9	Controladores.....	19
5.9.1	UserController	19
5.10	Programación orientada a ferrocarril y Manejo de Excepciones/Errores	20
5.11	Documentación	20
5.11.1	KDoc y Dokka	20
5.11.2	Swagger	21

5.12	Información Adicional	21
5.12.1	DB	21
5.13	Tests	21
5.13.1	Postman (E2E)	21
5.13.2	Junit 5 + MockK	22
5.14	Despliegue	22
6	Game Desing Document	23
6.1	Diferencias en un desarrollo software de un videojuego respecto al de una aplicación comercial al uso	23
6.2	Información General	23
6.2.1	Título	23
6.2.2	Resumen Del Juego	23
6.2.3	Objetivos que alcanzar	24
6.2.4	Justificación del juego	24
6.3	Core Gameplay	24
6.4	Características del juego	25
6.4.1	Género	25
6.4.2	Número de jugadores	25
6.4.3	Plataformas de destino	25
6.4.4	Descripción de estética	25
6.4.5	Resumen de la historia	25
6.4.6	Tutorial	25
6.5	Interfaz del Juego	26
6.5.1	Iteración de nivel 0	26
6.5.2	Iteración de nivel 1	26
6.5.3	Comparación de iteraciones	27
6.6	Estética y Arte	34
6.6.1	Jugador	34
6.6.2	Enemigos	34
6.6.3	Mundo	35
6.7	Elementos sonoros	35
6.8	PEGI	35
7	Bibliografía	36
7.1	Back End	36
7.2	Front End / Godot	36
7.3	Assets	36

1 INTRODUCCIÓN

1.1 DESCRIPCIÓN

Se trata de un juego del género **Bullet Hell**, es decir, *esquivar balas y sobrevivir el máximo tiempo posible* para así conseguir el *máximo de puntuación*; además de lograr generar una sensación de juego competitivo sin ser estrictamente un *multijugador*, usando una tabla de puntuaciones.



"Ikaruga" (2001) // Género: Bullet Hell

El usuario podrá jugar una partida de modo casual, aguantando el máximo posible para así obtener una puntuación más alta que la del resto de jugadores.

Se dispondrá de una base de datos donde se almacenarán tanto a los usuarios como sus puntuaciones más altas.

Igualmente, no todo el mundo posee conexión a internet las 24 horas del día, por lo que pienso que un modo **offline/sin iniciar sesión** es muy necesario, aunque se pierda la posibilidad de ver esa tabla de puntuaciones.

Con este proyecto, busco ampliar mis conocimientos sobre el mundo del desarrollo de software, en uno de los campos que más me llaman la atención junto al de las *Inteligencias Artificiales*, los **videojuegos**.

Personalmente también me lo planteo como un reto el hecho de lograr conectar tecnologías conocidas con desconocidas. También busco mejorar mis dotes de diseñador de interfaces, muy necesario.

2 REQUISITOS

2.1 REQUISITOS FUNCIONALES

El *usuario* podrá realizar las siguientes acciones:

- Jugar
- Iniciar Sesión
- Cerrar la Sesión
- Crear una Cuenta
- Ver una tabla de puntuaciones
- Ver su propio perfil
- Jugar de modo **offline** o **sin iniciar sesión**, en caso de no disponer de conexión a internet.
- Elegir entre tres dificultades.
- Cambiar la contraseña.
- Borrar la Cuenta

2.2 REQUISITOS NO FUNCIONALES

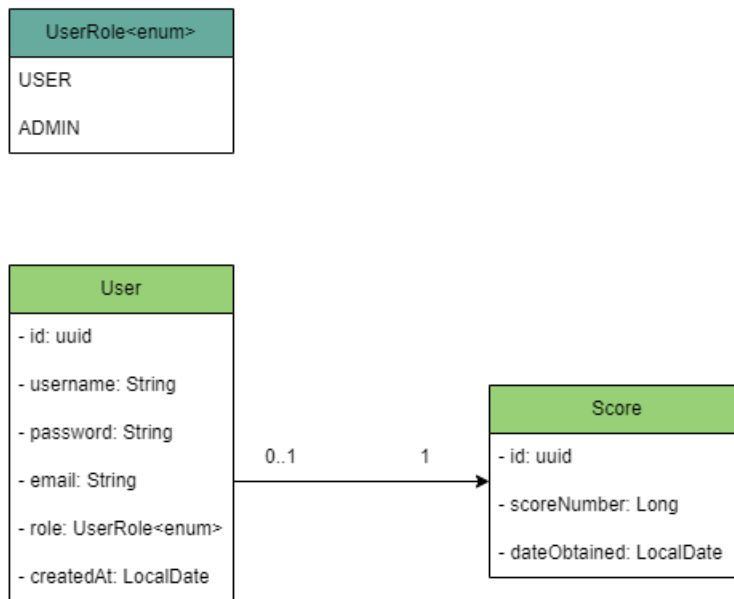
- La base de datos será SQL.
- Se dispondrá de un servicio API-REST que comunique la base de datos con la aplicación.
- El usuario iniciara sesión con su nombre de usuario y su contraseña, luego la aplicación almacenara el token correspondiente y este será utilizado, mientras no caduque, para realizar la comunicación con el servidor.
- La puntuación se subirá de manera automática si la puntuación es superior a la obtenida anteriormente.
- Para poder actualizar la contraseña, se deberá disponer de la contraseña actual, como medida de seguridad.

2.3 REQUISITOS DE INFORMACIÓN

Se dispondrán de **dos** entidades:

- Usuarios
- Puntuaciones

La relación entre ellas será simple, un usuario podrá tener desde 0..1 puntuación y la puntuación solo podrá pertenecer a un único usuario.



“Diagrama de clases”



“Diagrama entidad-relación”

3 ANÁLISIS DE MERCADO

La idea del proyecto surgió principalmente de **tres** videojuegos:

- **Akane** (2018)



- **Nier: Automata** (2017)

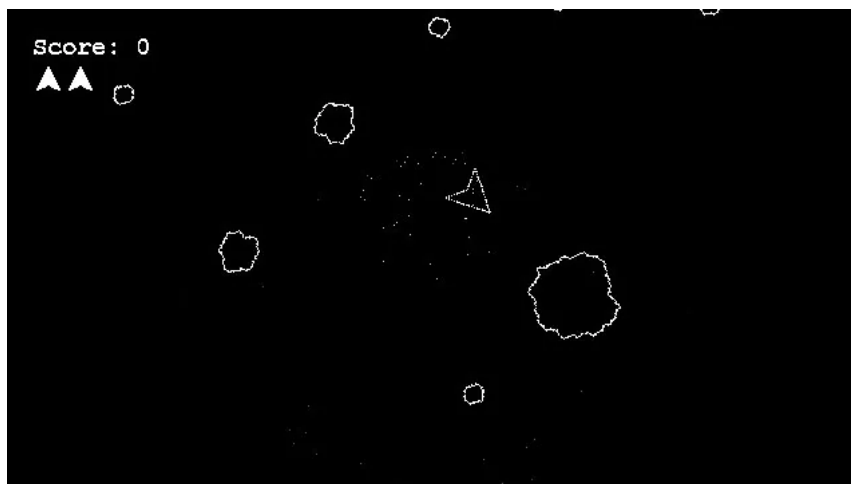


En el **primer caso**, es un juego basado en sobrevivir el mayor tiempo posible sobre una **arena de combate**, sobre la cual no dejan de aparecer enemigos. Desgraciadamente este juego no cuenta con tabla de puntuaciones, por lo que su aspecto competitivo se reduce únicamente a hablar con amigos o subir la puntuación máxima conseguida a algún foro por internet. Igualmente me resulta interesante el reto que ofrece, porque siempre perderás con tan solo un toque de algún enemigo.

En el **segundo caso**, me inspire en los pequeños fragmentos que, a su vez, se inspiraban en el propio género **bullet hell**. El resto del juego no tiene nada que ver, pero las opciones jugables, y el permitir destruir esas propias balas de distintas formas me resulto interesante.

Finalmente, mi idea **final** de diseño gráfico y en parte jugable proviene de un videojuego bastante más antiguo:

- **Asteroids** (1979)



Para el enfoque arcade este último caso es perfecto, y cuenta con las mismas razones que por ejemplo *Nier*, el poder destruir elementos, y el reto ofrecido por *Akane*, traducido a los distintos niveles de dificultad que serán ofrecidos.

En la perspectiva más estricta, ninguno de esos tres juegos son un *bullet hell* en su totalidad, pero sí que encuentro elementos individuales que podrían hacer uno muy competitivo y divertido.

4 ANÁLISIS TECNOLÓGICO

4.1 BACKEND

4.1.1 Gestión de la información

La gestión de la información será gestionada usando una **base de datos relacional** o **SQL**, debido a que, para este problema en específico, pienso que es la mejor solución.

- Una de las mayores ventajas ofrecidas por una base de datos **NoSQL** es el poder congelar la información, algo muy útil para mantener un historial de pedidos. Pero en este caso, al usuario solo le interesa mantener una única puntuación, la más alta obtenida, y el cambio de contraseña no afecta en nada a ese almacenamiento, por lo tanto, para poder mantener esa relación de forma adecuada, la mejor solución para este problema es aplicar una base de datos **SQL**.
- Debido a eso, quedó descartado el uso de **Firestore**.



- Para poder acceder a los datos de forma correcta, he decidido usar un servicio API-REST, ya que me permite acceder a los datos de la propia base de datos, y aplicar una serie de capas de seguridad para así mantener integra la información; desde aquí podría gestionar las distintas operaciones, como el registro de nuevos usuarios, los inicios de sesión más la generación de tokens...



- Para lanzar el servidor de la base de datos, se lanzará en un **contenedor o Docker**, con su respectivo **Docker-Compose**.



4.1.2 Servicio API-REST

- La **API-Rest** se realizaría con **Java o Kotlin**; ambas ofrecen un apoyo parecido, si bien Kotlin facilita la implementación reactiva gracias a las corrutinas y flujos, frente a Java, y el uso de flux y mono.
 - Estos lenguajes han sido seleccionados por un motivo de diseño del proyecto que se mencionó en la introducción, mi idea es juntar tecnología conocida con desconocida; de ese modo, el *backend* será realizado con tecnología que he usado a lo largo del curso.
 - Este servicio será reactivo, principalmente por la gestión de puntuaciones, que al ser accedida un X número de veces de forma continua, y esta a su vez necesitar información del usuario asociado, mejorará la velocidad del servicio.
 - Este servicio, además de contar con un lenguaje de los ya mencionados, se aplicaría un **framework**, y la decisión sería: **Ktor o Spring Boot**.
 - **Ktor**:
 - **Puntos positivos:** Es ligero, y ofrece mayor control sobre sus distintas configuraciones.
 - **Puntos negativos:** Es menos maduro que la contraparte, ofrece menos integraciones, por lo que se gana en control se pierde en eficacia y optimización de código. Solo es posible con Kotlin.
 - **Spring Boot**:
 - **Puntos positivos:** Tanto Java como Kotlin son soportados, gracias a su historial ofrece una seguridad en la implementación difícil de alcanzar.
 - **Puntos negativos:** Si bien su implementación es relativamente más rápida, la configuración es más estricta que en Ktor, aunque subsanado con la cantidad ingente de información que hay por Internet, es un factor importante para tener en cuenta. Es bastante más pesado, y tarda más en ejecutarse por primera vez.

- Personalmente, prefiero implantar **Spring Boot**, debido a la robustez que ofrece gracias a su madurez e historial.



- En la elección del lenguaje, usare **Kotlin**, debido a la facilidad de implementar la **reactividad** que busco.



- Además, para probar los diferentes *end points*, se hará uso de una plataforma diseñada para API llamada **Postman**.



4.2 FRONTEND

4.2.1 Motor de desarrollo

- **Unreal Engine** queda **descartado** desde un comienzo por su alta dificultad de aprendizaje tanto del programa como del lenguaje usado principalmente, **C++**, si bien ofrece una serie de ventajas que los demás motores no son capaces de cubrir, como su gran capacidad para gráficos 3D o entornos virtuales; son **ventajas** que para este proyecto **no** son **necesarias**. Además, es conocido por no manejar de forma correcta entornos 2D.
- La decisión se decide principalmente entre dos motores *alternativos*: **Unity** o **Godot**.
 - **Unity:**
 - **Puntos positivos:** Cuenta con una comunidad muy activa, el aprendizaje es de dificultad media y cuenta con una gran tienda de *assets* **oficial**. Gran número de plataformas a las que exportar el resultado final.
 - **Puntos negativos:** Solo permite un lenguaje de desarrollo, **C#**, es **software** **privativo**. Es un motor **pesado** para ejecutar en un ordenador **modesto**.

- **Godot:**
 - **Puntos positivos:** En cuanto a curva de aprendizaje, es el más sencillo para comenzar, gracias al diseño por nodos. Al ser un **motor** de **código libre**, entre otras ventajas, es sencillo encontrar gran parte de ayuda por Internet. Permite el uso tanto de **C#**, de **C++** y finalmente, de su propio lenguaje, **GDScript**, para desarrollar el juego. Es el motor perfecto para el desarrollo en 2D. Es un motor **ligero** de ejecutar.
 - **Puntos negativos:** No dispone de una tienda oficial de *assets*, por lo que el resultado final se debe de hacer a mano o buscar alternativas. El número de plataformas de exportación es más limitado. Los ambientes 3D no son su especialidad. Su motor de físicas es algo inferior al ofrecido en Unity. Es el menos conocido laboralmente.
- **Personalmente**, me encanta la programación funcional de **Python**, y lo más parecido a eso en este mundillo es *GDScript*; el que sea software de código libre me llama la atención. Finalmente, pienso que al ser una experiencia nueva y ofrecer la mejor curva de aprendizaje, me decantaría por **Godot**.
- La **elección del lenguaje** en ese sentido es clara, usaría **GDScript**, ya que es un lenguaje desarrollado única y exclusivamente para este motor, y es el lenguaje que cuenta con la mejor implementación. Además, al ser tan parecido a *Python*, lo veo una buena forma de cimentar conocimientos parecidos para el mismo.



4.3 HERRAMIENTAS DE DISEÑO

- **Draw.io:** Editor de diagramas.



- **Paint:** Editor de imágenes.



4.4 HERRAMIENTAS ADICIONALES

- **GitHub:** Plataforma para alojar código utilizando el sistema de control de versiones **Git**.



"Git"

- **GitKraken:** Cliente Git de escritorio multiplataforma.



- **Itch.io:** Plataforma de distribución digital. Haciendo uso *específico* de la tienda de **assets**.



5 BACKEND-EXPLICACIÓN

5.1 MODELOS

Siguiendo el diagrama de clases, contamos con dos elementos:

- **Usuario / User**
 - **ID:** Id que identifica al elemento en la base de datos. (**Primary Key**) (**UUID**)
 - **Username:** Nombre de usuario (**String**)
 - **Password:** Contraseña del usuario, almacenada cifrada con **Bcrypt** por seguridad. (**String**)
 - **Email:** Correo electrónico del usuario. (**String**)
 - **Role:** Rol del usuario; existen dos tipos:
 - **USER**
 - **ADMIN**
 - **CreatedAt:** Fecha donde se creó al usuario. (**LocalDate**)
- **Puntuación / Score**
 - **ID:** Id que identifica al elemento en la base de datos. (**Primary Key**) (**UUID**)
 - **UserId:** Id que relaciona al usuario con la puntuación. (**Foreing Key**) (**UUID**)
 - **ScoreNumber:** El número obtenido de la puntuación conseguida. (**Long**)
 - **DateObtained:** Fecha donde se obtuvo la puntuación. (**LocalDate**)

5.2 DTOs

Según las distintas necesidades del programa se han desarrollado distintos DTO para poder facilitar el trabajo con los datos.

5.2.1 Usuarios

- **UserDTOLogin:** Usado para iniciar sesión.
 - **Username** (String)
 - **Password** (String)
- **UserDTORegister:** Usado para el registro de nuevos usuarios
 - **Username** (String)
 - **Password** (String)
 - **RepeatPassword** (String)
 - **Email** (String)
- **UserDTOCreate:** Usado para la creación de un nuevo usuario por parte de un administrador.
 - **Username** (String)
 - **Password** (String)
 - **Email** (String)
 - **Role** (String)
- **GenericResponse:** Usado para dar una respuesta genérica con un único valor.
 - **Value** (String)

- **UserDTOProfile:** Usado para mandar la información que un usuario puede ver sobre él mismo.
 - o **Username** (String)
 - o **Email** (String)
 - o **CreatedAt** (String)
 - o **Score** (ScoreDTOResponse?)
 - Si el usuario es nuevo y no ha subido ninguna puntuación es posible que no tenga ninguna.
- **UserDTOLeaderBoard:** Usado para facilitar el acceso según la puntuación obtenida y su posición.
 - o **Position** (String)
 - o **Username** (String)
 - o **Score** (ScoreDTOResponse)
- **UserDTOPasswordUpdate:** Usado para la actualización de la contraseña de un usuario.
 - o **ActualPassword** (String)
 - o **NewPassword** (String)
 - o **RepeatNewPassword** (String)

5.2.2 Puntuaciones

- **ScoreDTOCreate:** Usado para la creación de una nueva puntuación
 - o **UserId** (String)
 - o **ScoreNumber** (String)
- **ScoreDTOResponse:** Usado cuando se solicite una puntuación.
 - o **ScoreNumber** (String)
 - o **DateObtained** (String)

5.3 MAPEADORES

Hacemos uso de distintos *mappers* como funciones de *extensión* para el paso de Modelo a DTO.

En estas funciones es donde, generalmente, indicamos la fecha de creación tanto de usuarios como de las puntuaciones. También, y para facilitar el paso de la información, es donde se transformarán ciertos datos de un dato complejo a String (es decir, para facilitar el uso de **JSON** como formato para el intercambio de datos).

Menciono el traspaso de un String a un rol real en el mapper de **UsuarioDTOCreate**.



5.4 REPOSITORIOS

La estructura de los repositorios es la siguiente:

- Contamos con dos interfaces por modelo:
 - o **XRepository**: En esta se implementa el repositorio necesitado de los ofrecidos por **Spring**, en mi caso, **CoroutineCrudRepository**; además de indicarle, si es el caso, algún método extra que sea necesario escrito de una forma adecuada para que sea reconocido.
 - o **IXRepository**: En esta interfaz, se escriben los métodos que se usaran realmente, apoyándose en la interfaz anterior.
- Luego de esto, contamos con una clase por modelo donde se pondrán en uso ambas interfaces descritas:
 - o **XRepositoryCached**: En esta clase es donde se escribirá la lógica, inyectando por constructor los repositorios e implementando el **IRepository** que se necesite.
 - Además, y como el nombre de la clase indica, es donde se pondrá en uso las anotaciones propias de Spring para implementar una posible **cache**, siempre teniendo en cuenta que puede no ser necesario.

5.5 VALIDADORES

Para no hacer un código demasiado denso, se han diseñado una serie de **funciones de extensión** encargadas de verificar los distintos DTO que se reciban. Se ha implantado el uso de **Result** para ello.

La mayoría comprueban, por ejemplo, si el nombre del usuario en su registro está en blanco, si el correo electrónico es correcto.

Lo más importante a destacar es:

- La **contraseña** deberá de tener como mínimo: **5 caracteres** para considerarse correcta.
- El **rol** deberá de ser o bien **User** (por defecto en el **registro**) o **Admin** (en la opción de **creación**).

Si en cualquier caso no se cumple uno de los requisitos, se hará uso del resultado **“incorrecto”** de la clase **Result**, que resultará en un mensaje de aviso al usuario.

Se han aplicado validadores en los siguientes DTO:

- **UserDTORegister**
- **UserDTOCreate**
- **UserDTOLogin**
- **UserDTOPasswordUpdate**

5.6 CONFIGURACIÓN

5.6.1 APIConfig

Clase encargada de disponer de información general que será usada por las distintas clases del proyecto.

- Aquí contaremos con la **ruta** base de la API: “/sp4ceSurvival”, a modo de constante.

5.6.2 LoadSchema

Clase genérica encargada de cargar el archivo **schema.sql** (localizado en **Resources**) cuando se ejecute la aplicación.

Existen dos archivos, uno orientado al desarrollo y otro orientado en la producción, este se **define** en el **archivo de propiedades**.

5.6.3 Archivo de propiedades

- Se define el puerto que escuchara la aplicación: **6969**
- Junto al Main, se define que los repositorios de **R2DBC** estén activados.
- Se define la cadena de conexión con la base de datos. (Esta variable será definida con **POSTGRES_CONNECTION** para luego ser modificada cuando se lance en un Docker junto a una base de datos sin los puertos expuestos)
- Entre otras cosas que ayudan al desarrollo del proyecto, se encuentra el **secreto** usado para los tokens.
- La configuración completa de la seguridad **SSL** aplicada.
- El **Schema.sql** a cargar si es el *Docker-compose* de **desarrollo** o de **producción**.

5.7 SEGURIDAD

La implementación de la seguridad propia de Spring es compleja, pero siempre suele ser la misma implementación, y aunque siga perteneciendo al apartado de **configuración**, prefiero explicarlo en un nuevo apartado.

5.7.1 Password

Dentro de este paquete, se encuentra la clase **EncoderConfig**, que será la encargada de cifrar la contraseña usando **Bcrypt** haciendo uso de una función.



5.7.2 JWT



"JSON Web Token"

Dentro de este paquete, se encuentra toda la lógica relacionada con los tokens de identificación.

- **JwtTokensUtils:** Dentro de esta clase, se encuentran los métodos que generan y decodifican los tokens usando el mismo secreto y algoritmo.
 - o Lo más destacable es la fecha de caducidad del token: **2 días**
- **JwtAuthenticationFilter:** Esta clase se encarga de autenticar el token que se reciba en las posibles acciones del usuario.
- **JwtAuthorizationFilter:** Esta clase se encarga de autorizar el token, es decir, se asegura de que el token no haya caducado y de verificar el usuario asociado, así como sus roles. Si en la función de autenticación se devuelve un **nulo**, *Spring* lanza un código de error 403 (**Forbidden**).

5.7.3 SecurityConfig

Esta clase será la encargada de implementar las distintas medidas de seguridad en el proyecto, así como, según el **endpoint**, permitir el paso según el rol, por ejemplo.

5.7.4 SSL

La conexión respecto la API y el cliente se encuentra cifrada con SSL para así evitar un ataque muy conocido llamado **Ataque de intermediario** (o **Man in the Middle Attack**).

Los **certificados**, realizados usando la herramienta propia de Java, **Keytool**, se encuentran en formato **PKCS12** para una mayor compatibilidad frente a **JKS** y tienen una duración de **5 años** (o **1825 días**) a partir de la siguiente fecha: **16/04/2023**.

5.8 SERVICIOS

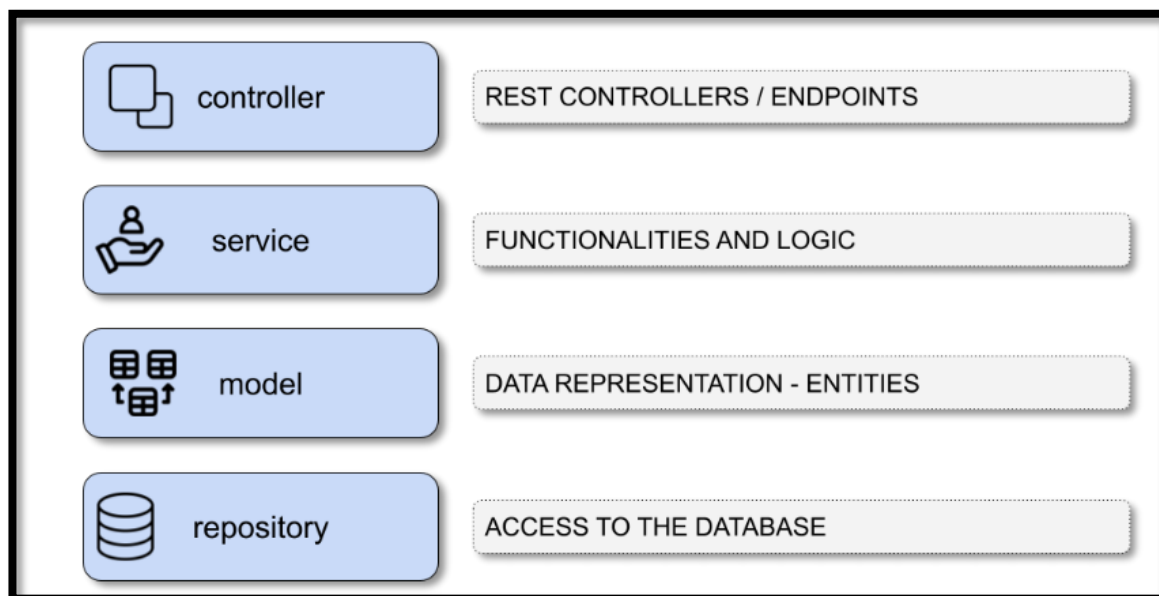
5.8.1 UserService

En este proyecto solo se dispondrá de un servicio, además de ser necesario para la configuración de la seguridad de Spring, me permite juntar ambos repositorios cacheados y poder trabajar con los métodos que luego, realmente serán los implementados en el controlador.

En la mayoría de las funciones trabajan con la clase **Result** si la respuesta por parte del repositorio ha sido de tipo **null**.

Aquí es donde se **cifrará** la contraseña de los usuarios usando la clase descrita anteriormente, **EncoderConfig**.

5.9 CONTROLADORES



5.9.1 UserController

Aquí **verificaremos** los distintos DTO que el usuario pueda enviar haciendo uso de las **validaciones** explicadas anteriormente, y según el resultado obtenido, se realizará la operación deseada o se enviará un aviso.

Esta clase será la encargada de recibir y enviar información con el cliente, llamando a los distintos métodos del servicio, que como sucedía con las validaciones, trabajará según el resultado obtenido.

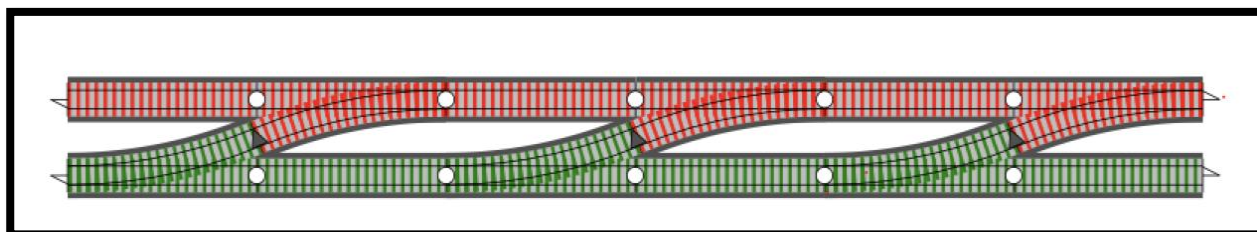
Existen un par de funciones que sirven para iniciar una serie de datos la primera vez que se ejecute la aplicación, cuentan con el sufijo **Initializer** y se encuentran debajo del resto de funciones.

5.10 PROGRAMACIÓN ORIENTADA A FERROCARRIL Y MANEJO DE EXCEPCIONES/ERRORES

En un comienzo se desarrollaron una serie de clases de excepciones personalizadas, usando herramientas propias de **Spring**:

- **UserException**: Implementa **RuntimeException**; se encarga de recoger la excepción, y podemos escribir un mensaje personalizado.
 - **UserExceptionBadRequest**: Implementa **UserException**.
 - **UserExceptionNotFound**: Implementa **UserException**.
 - **UserDataBaseConflict**: Implementa **UserException**.

Pero, gracias al uso de la clase **Result** en los *validadores*, *servicio* y el manejo de esta en el *controlador*, se ha logrado implementar una **Programación Orientada a Ferrocarril** (o **Railway Oriented Programming**), dando la posibilidad de mejorar la legibilidad del código y el entendimiento frente a los errores.



A modo de *resumen*, podría decirse que en el código de las *validaciones* existen dos lados de la *Fuerza*, el **Lado Luminoso**, donde la validación sobre el dato pasa sin **ningún problema**, y el **Lado Oscuro**, donde si salta alguno de los filtros dados, devuelve el aviso que hayamos indicado.

Esto significa que se ha reducido enormemente el uso de herramientas como el **Try...Catch** o **Throws**, que, si bien aún pueden servir en algún caso extremo, es importante tener en cuenta que las *excepciones* son, **excepciones**, y no deberían formar parte de la lógica de un programa de forma convencional.

Por lo tanto, las **excepciones** actualmente actúan como un **complemento** y para mejorar la legibilidad del código y gracias a sus descriptivos nombres, se puede saber qué tipo de mensaje será enviado en el controlador a modo de respuesta.

5.11 DOCUMENTACIÓN

Además de contar con el propio archivo PDF, se han implementado varios tipos de documentación en el propio código.

5.11.1 KDoc y Dokka

En todas las clases del proyecto se ha documentado usando **KDoc**, y gracias a una herramienta adicional, **Dokka**, esta puede verse a través de un archivo **HTML**. Esta documentación se encuentra en **inglés**.

5.11.2 Swagger

Además, se ha aplicado la clásica herramienta para documentar APIs, **Swagger**.

La ruta base de Swagger será la siguiente, siempre y cuando la aplicación se encuentre en ejecución:

<https://localhost:6969/swagger-ui/index.html#/>



5.12 INFORMACIÓN ADICIONAL

5.12.1 DB

- **Data**: Clase que cuenta con datos de prueba, cargados en el Main la primera vez que se ejecute la aplicación con las funciones **Initializer**.

5.13 TESTS

5.13.1 Postman (E2E)

Se ha realizado una prueba exhaustiva de todos los endpoints disponibles, tanto de forma correcta, de forma incorrecta y no autorizados.

Se ha exportado la colección a la carpeta **postman** en la raíz del proyecto. Los tokens han sido almacenados como variables de entorno, y podemos observar tres tipos:

- **token_admin**: Token de un administrador.
- **token_user**: Token de un usuario base.
- **token_noScore**: Token de un usuario recién creado.
- **token_expired**: Token caducado.

Los primeros **2** tokens necesitarán ser actualizados a un nuevo valor cada 2 días (usando los propios métodos de **inicio de sesión**)

El **tercer** token deberá de ser actualizado cada vez que se ejecute el backend, puesto que es un usuario totalmente nuevo sin una puntuación asociada (este token se obtiene usando o bien un **registro** o una **creación** por parte de un *Administrador*).

El **cuarto** token no necesitará ninguna actualización.

5.13.2 Junit 5 + MockK



Se han realizado pruebas de los **tres** elementos principales de la aplicación:

- Repositorios.
- Servicios.
- Controladores.

A continuación, se muestra un ejemplo de una prueba, el **registro** en el servicio, aplicando MockK y trabajando con la clase Result. Con una comparación directa a su respectiva función del servicio.

```

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun register() = runTest { this: TestScope
    coEvery { repository.findByUsername(any()) } returns null
    coEvery { repository.findByEmail(any()) } returns null
    coEvery { passwordEncoder.encode(any()) } returns userDtoRegister.password
    coEvery { repository.save(any()) } returns user

    val result = service.register(userDtoRegister)

    assertAll(
        { assertNotNull(result) },
        { assertEquals(userDtoRegister.username, result.component1().username) }
    )

    coVerify { repository.findByUsername(any()) }
    coVerify { repository.findByEmail(any()) }
    coVerify { passwordEncoder.encode(any()) }
    coVerify { repository.save(any()) }
}

```

```

/**
 * Register Function for new Users
 */
suspend fun register(userDtoRegister: UserDtoRegister): Result<User, UserException> {
    log.info { "Registering User with username: ${userDtoRegister.username}" }

    val usernameUsed = userRepositoryCached.findByUsername(userDtoRegister.username)

    return if (usernameUsed == null) {
        val emailUsed = userRepositoryCached.findByEmail(userDtoRegister.email)

        if (emailUsed == null) {
            val user = userDtoRegister.toUser()

            val userNew = user.copy(
                password = passwordEncoder.encode(user.password)
            )

            Ok(userRepositoryCached.save(userNew))
        } else Err(UserDataBaseConflict("Email already used"))
    } else Err(UserDataBaseConflict("Username already used"))
}

```

Finalmente, se adjunta un ejemplo de prueba con un resultado negativo posible:

```

@OptIn(ExperimentalCoroutinesApi::class)
@Test
fun registerFailed() = runTest { this: TestScope
    coEvery { repository.findByUsername(any()) } returns user

    val result = service.register(userDtoRegister)

    assertAll(
        { assertNotNull(result) },
        { assertEquals(expected: "Username already used", result.component2().message) }
    )

    coVerify { repository.findByUsername(any()) }
}

```

5.14 DESPLIEGUE

Actualmente la aplicación tiene dos tipos de *despliegue* a través de dos archivos **Docker-Compose**

- *Docker-compose de desarrollo (dev)*: Se expone la base de datos y se ejecuta la aplicación desde el IDE. Los datos **serán reiniciados** cada vez que se levante la base de datos.
- *Docker-compose de producción (production)*: Se lanza el servicio (haciendo uso de un **dockerfile**) con los puertos expuestos, mientras que la base de datos se encuentra protegida, y los datos serán conservados.

6 GAME DESING DOCUMENT

6.1 DIFERENCIAS EN UN DESARROLLO SOFTWARE DE UN VIDEOJUEGO RESPECTO AL DE UNA APLICACIÓN COMERCIAL AL USO

Con una aplicación de estilo ERP ('Enterprise Resource Planning' o 'Planificación de Recursos Empresariales') se busca facilitar la vida de las personas de una empresa, respecto al negocio que estos posean. Es decir, se busca una aplicación **útil y para uso profesional o personal**, como podría ser una aplicación de **entrenamiento personal físico**, por citar un ejemplo.

Si bien un videojuego no es que sea precisamente inútil, y existen casos aplicados al *uso académico*, su objetivo suele ser uno muy distinto, y es el **divertir al jugador**.

Es decir, el objetivo a conseguir se desvía de lo habitual, si bien en mi opinión se logran una serie de objetivos que una aplicación al uso no podría conseguir, de ahí mi interés respecto al mismo.

6.2 INFORMACIÓN GENERAL

6.2.1 Título

El juego se titula: "Sp4ce Survival". Se deberá sobrevivir el mayor tiempo en un escenario espacial, el 4 agrega una capa de profundidad al nombre, eso, o parece la cuarta entrega de una saga.

6.2.2 Resumen Del Juego

El jugador deberá aguantar lo máximo posible en la arena de combate, aguantando hordas de enemigos mientras estos disparan balas *siguiendo patrones clásicos de un **bullet hell***.



"Enter the Gungeon (2016)"

El juego dispondrá de una cuenta atrás que limitará la duración de estas.

Según el nivel de dificultad y el desempeño del jugador, si está jugando en **línea**, podrá subir su puntuación de forma automática, siempre que sea superior a la ya publicada.

6.2.3 Objetivos que alcanzar

El objetivo del jugador será lograr la máxima puntuación posible.

6.2.4 Justificación del juego

Basándonos en el análisis del mercado inicialmente explicado en el documento, opino que se puede aportar ciertos matices de aquellas experiencias en una sola y que esta sea divertida.

El usuario dispondrá de varias herramientas para lograr cumplir el objetivo principal:

- Niveles de dificultad que, según la selección, multiplicarán la puntuación final obtenida por un número X.
- La posibilidad de aumentar el temporizador, y, por lo tanto, aguantar más tiempo.
- La posibilidad de destruir ciertas balas.
- La posibilidad de destruir enemigos.
- El tiempo resistido.

Con el listado descrito, el jugador podrá ir sumando puntos hasta obtener la puntuación final, esta será visible en una **tabla de puntuaciones in-game**.

6.3 CORE GAMEPLAY

La actividad principal del juego será aguantar el mayor tiempo posible evitando cualquier contacto con los enemigos.

El jugador podrá disparar proyectiles que, podrán eliminar un tipo de balas enemigas y a todos los enemigos en sí mismos. Otro tipo de proyectiles solo podrán ser esquivados.

El jugador dispondrá de balas infinitas, pero un cargador limitado, que obligará a tener cierta cautela a la hora de abrir fuego.

El **tiempo inicial** proporcionado será de **90 segundos**.

Existen *dos formas* de augmentar el tiempo proporcionado:

1. Obtener una mejora de campo que, de forma aleatoria, saldrá en algún punto de la arena, y el jugador tendrá que acercarse para obtenerlo.
2. Eliminar 7 enemigos.

Los niveles de dificultad afectan de forma grave al aguante del jugador, pero también a la puntuación final obtenida:

- **Fácil:** 3 Golpes – x2 a la puntuación.
- **Medio:** 2 Golpes – x4 a la puntuación.
- **Difícil:** 1 Golpe – x7 a la puntuación.

6.4 CARACTERÍSTICAS DEL JUEGO

6.4.1 Género

- **Arcade:** Debido a su enfoque casual, jugabilidad sencilla y de acción rápida.
- **Matamarcianos (o Shoot 'em up):** Se maneja a un personaje que dispara contra hordas de enemigos.
 - o Dentro de este género y debido a los patrones del enemigo, se consideraría un **Bullet Hell**.

6.4.2 Número de jugadores

Solo un jugador, pero cuenta con capacidad competitiva online.

6.4.3 Plataformas de destino

Actualmente, se tiene en mente un lanzamiento único en ordenadores, pero no se descarta el lanzamiento en otras plataformas en un futuro, por lo que se tendrá en cuenta durante el desarrollo.

6.4.4 Descripción de estética.

La estética será espacial de ciencia ficción.

6.4.5 Resumen de la historia

El jugador trata de escapar de un planeta luego de obtener cierta información confidencial de la mayor organización criminal de la galaxia Omega, pero es perseguido por los agentes de seguridad, que tratarán de detener su escape.

Visto que no puede salir con vida, decide enviar los datos que pueda mediante conexión estelar a su propia organización, durante el tiempo que resista.

6.4.6 Tutorial

No existirá un tutorial real, lo más cercano será el modo Fácil, pues el juego seguirá siendo exactamente el mismo a excepción del número de vidas.

6.5 INTERFAZ DEL JUEGO

Se han realizado diferentes iteraciones respecto a la interfaz del usuario, las cuales han seguido una serie de reglas:

1. Medianamente **atractivo**.
2. Elementos **perfectamente** visibles y de **fácil** acceso.
3. Interfaz fácilmente integrable en otras plataformas, como podría ser una **consola** o un **móvil**.

Ambas **iteraciones** se han desarrollado usando la herramienta **Paint**.

Nota: En ambas iteraciones se optó por un fondo de color **negro** y se usaron colores variados y vistosos para diferenciar elementos, no dejan de ser bocetos.

6.5.1 Iteración de nivel 0

En esta iteración, desarrollada **en conjunto** con el **backend**, buscaba mostrar cuantos menús serían necesarios y que elementos podrían estar en el mismo. Es una versión abstracta y algunos de sus elementos podrían ser alterados en cualquier momento, pero ofrecen una visión general.

6.5.2 Iteración de nivel 1

En esta iteración, desarrollada una vez el **backend** estaba **cimentado**, se buscaba perfilear y detallar algún elemento de los menús, así como replantear la organización de estos, si bien en iteración del nivel anterior algunos elementos estaban algo descolocados.

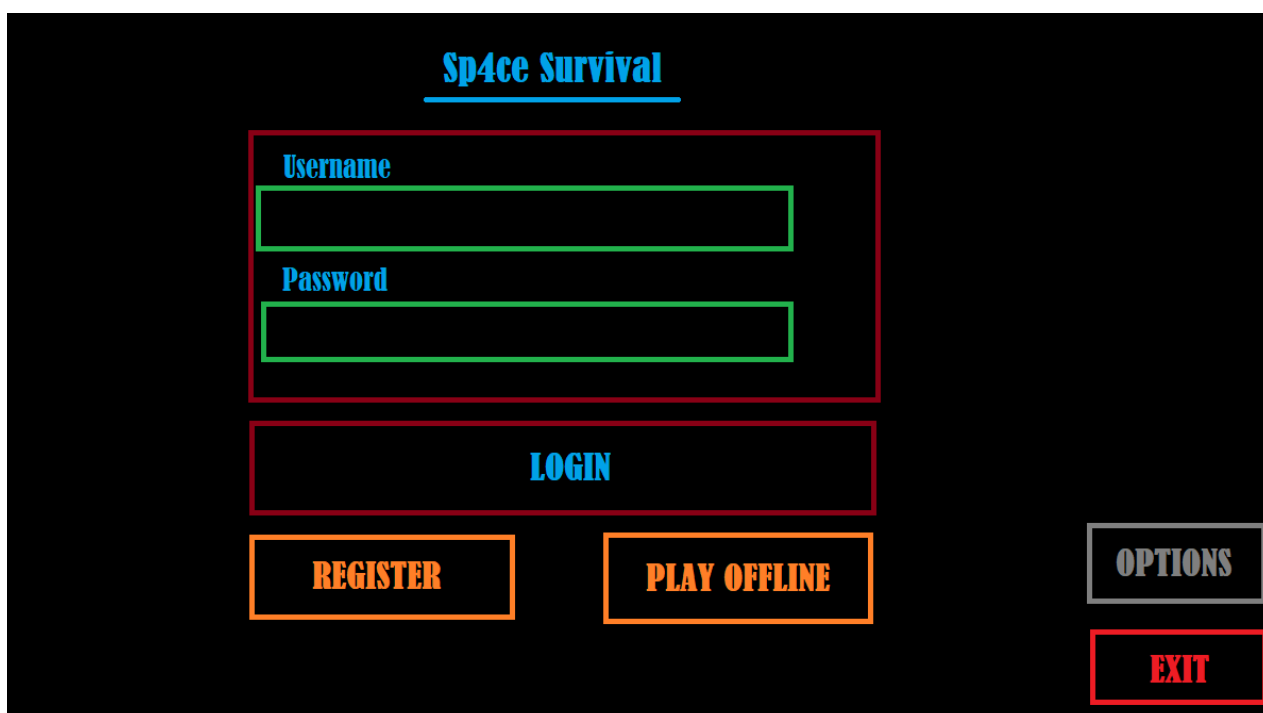
6.5.3 Comparación de iteraciones

A continuación, se mostrará una comparación directa de algunos de los mismos menús o elementos de ambas iteraciones, con el objetivo de mostrar las diferencias, mostrando primero la iteración de nivel 0 y luego su contraparte de nivel 1.

- Menú de Bienvenida



The screenshot shows a black background with several elements highlighted by colored boxes. At the top, a blue box contains the text **Game_Name**. Below it, the text **Username** is followed by a green-outlined input field. Then, the text **Password** is followed by another green-outlined input field. Below the password field, there are four buttons: **LOGIN** (red box), **PLAY OFFLINE** (yellow box), **CREATE ACCOUNT** (blue box), and **OPTIONS** (grey box). In the bottom right corner, there is a red box containing the text **EXIT**.



The screenshot shows a black background with several elements highlighted by colored boxes. At the top, the text **Sp4ce Survival** is underlined with a blue line. Below it, the text **Username** is followed by a green-outlined input field. Then, the text **Password** is followed by another green-outlined input field. Below the password field, there is a large red box containing the text **LOGIN**. At the bottom, there are three buttons: **REGISTER** (orange box), **PLAY OFFLINE** (orange box), and **OPTIONS** (grey box). In the bottom right corner, there is a red box containing the text **EXIT**.

- Menú de Registro

New Account

Username

Password

Repeat_Password

Email

CREATE ACCOUNT

RETURN

Username

Email

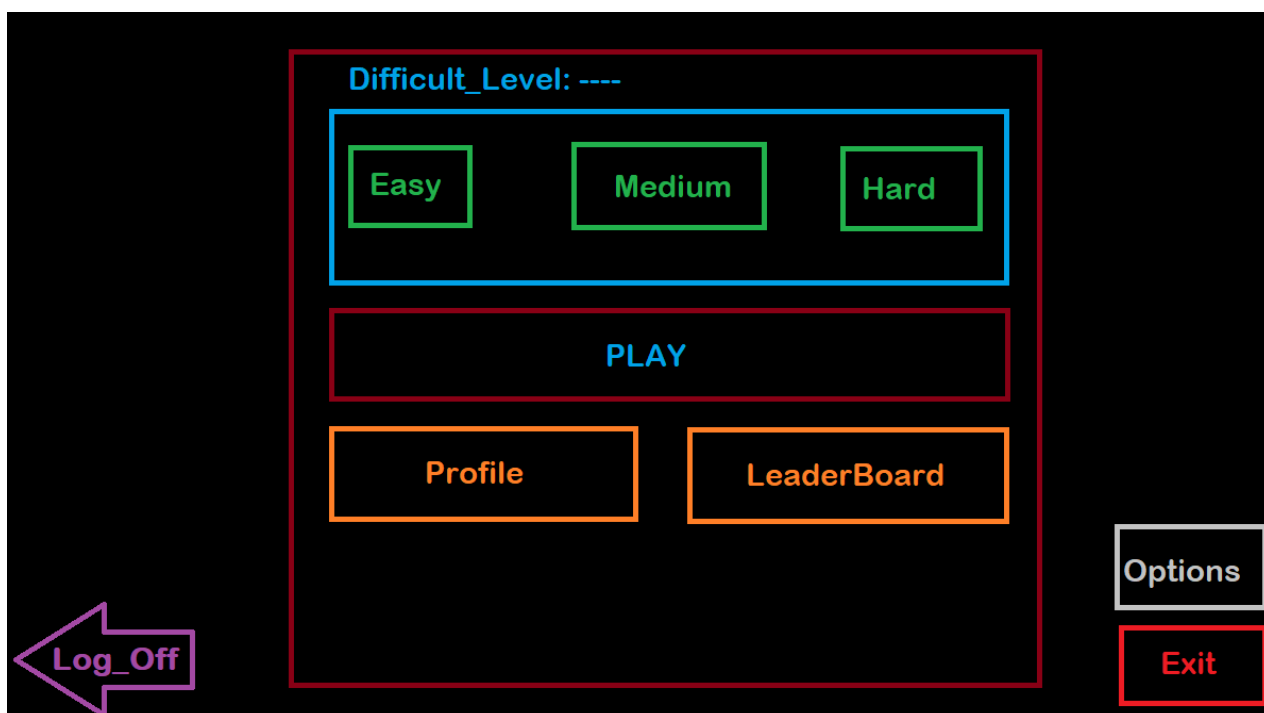
Password

Repeat_Password

REGISTER

RETURN

- Menú Principal



- Menú de Perfil de Usuario

USER PROFILE

Username: xxxx

Email: xxxx@xxxx

Created_at: xxxx

Max_Score: xxxx

Date_Score: xxxx

**CHANGE
PASSWORD**

**DELETE
ACCOUNT**

RETURN

User_Profile

Username: ----

Email: ----

Creation_Date: ----

Max_Score: ----

Date_Score_Obtained: ----

Change_Password

Delete_Account

Return

- Menú para cambiar la contraseña

CHANGE_PASSWORD

Actual Password:

New_Password:

Repeat_New_Password:

**UPDATE
PASSWORD**

RETURN

Actual_Password

New_Password

Repeat_New_Password

CHANGE_PASSWORD

Return

[illegible]

Return

←

→

Position	Username	Score	Date

Page: ----

- Menú Game Over.



6.6 ESTÉTICA Y ARTE

Aviso: *Todos* los assets han sido sacados de la plataforma online **Itch.io**; todos ellos han sido gratuitos, y han sido referenciados en la **bibliografía** del documento con:

- Nombre del Asset/Pack.
 - Nombre del Autor.
 - Enlace a la página de la tienda.

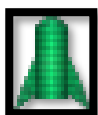
Se han tenido en cuenta las distintas formas de los elementos para facilitar la visión del jugador.

6.6.1 Jugador

- Nave:



- Proyectil:



6.6.2 Enemigos

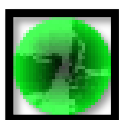
- Nave:



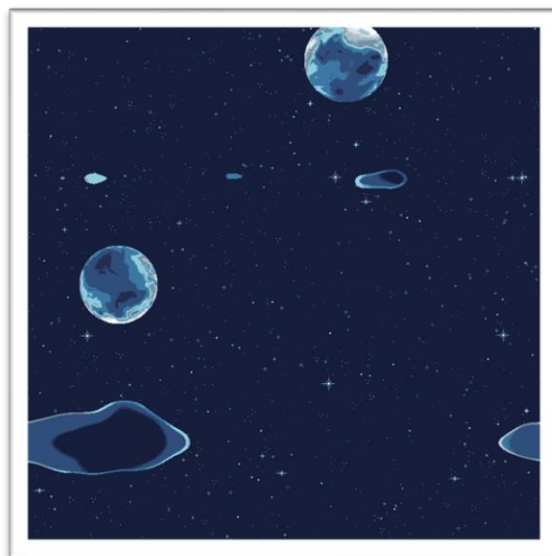
- Proyectiles:
 - Destruible por el jugador o Tipo 1



- No destruible por el jugador o Tipo 2



6.6.3 Mundo



- Ampliación de tiempo:



Se ha elegido una gama de colores que no choque ni moleste al jugador para así diferenciar de forma evidente que elementos son jugables e interactivos.

6.7 ELEMENTOS SONOROS

Se han elegido música y efectos sonoros futuristas, acompañando la ambientación descrita.

6.8 PEGI

El juego será PEGI 7.

7 BIBLIOGRAFÍA

7.1 BACK END

Documentación Spring:

- <https://docs.spring.io/spring-data/r2dbc/docs/current/reference/html/>
- <https://spring.io/projects/spring-boot>

7.2 FRONT END / GODOT

Se han realizado multitud de ejemplos desde varias fuentes; todos ellos han sido documentados en un *repositorio personal* en **Github**, visibles desde el **Readme**.

Estos ejemplos han sido la base del videojuego.

https://github.com/Mario999X/TrasteandoConGDScript_Godot

7.3 ASSETS

1. **Space Ultimate Megapack**
 - GameSupplyGuy
 - <https://gamesupply.itch.io/ultimate-space-game-mega-asset-package>
2. **Pixel Space Background Generator**
 - Deep-Fold
 - <https://deep-fold.itch.io/space-background-generator>