UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

ibermedia

PROYECTO FINAL DE CARRERA:

DESARROLLO DE UNA PLATAFORMA SOFTWARE PARA EL DESPLIEGUE DE JUEGOS MULTIJUGADOR

Proyecto presentado por **Francesc García Sáez** para la titulación de Ingeniero de Telecomunicación en la Universidad Politécnica de Valencia

---

Director:
José Enrique López Patiño
Departamento de Comunicaciones, ETSIT-UPV

Tutor:
José Ricardo López Simarro

Al meu pare,
i a tota la meva familia.

Resumen

El presente documento constituye la memoria del proyecto de final de carrera titulado ***Desarrollo de una plataforma software para el despliegue de juegos multijugador*** que ha sido realizado por Francesc García Sáez bajo la tutela del profesor del Departamento de Comunicaciones de la ETSIT, José Enrique López Patiño, y supervisado por José Ricardo López Simarro. El proyecto se ha desarrollado en su totalidad en Ibermedia durante nueve meses, en el marco del convenio de colaboración que dicha empresa mantiene con la Universidad Politécnica de Valencia.

Este proyecto se enmarca dentro de un proyecto de investigación de Ibermedia, financiado por el Centro para el Desarrollo Tecnológico Industrial del Ministerio de Economía (CDTI), orientado a desarrollar herramientas que faciliten el proceso de desarrollo y publicación de videojuegos sociales multijugador. El objetivo del proyecto consiste en el desarrollo de una plataforma basada en SmartFoxServer para desplegar un servidor de videojuegos capaz de albergar simultáneamente la lógica de varios juegos. Dicha plataforma debe contar, por un lado, con una serie de servicios básicos comunes a todos los juegos, y, por otro, con un conjunto de clases que permitan añadir los servicios particulares que necesitará cada uno de ellos.

El documento se estructura en cinco capítulos, el primero de los cuales, a modo de introducción, detalla la motivación del proyecto. Se explican las peculiaridades de los juegos sociales y se expone brevemente la evolución y la situación actual del sector. de los videojuegos sociales para justificar el desarrollo de una herramienta de estas características.

El segundo capítulo expone los principales objetivos del proyecto, así como los objetivos secundarios que han permitido alcanzarlos.

En el tercer capítulo se detalla el proceso de realización del proyecto. Se profundizará en la metodología empleada y se explicarán y justificarán las diversas tecnologías que lo han hecho posible. También incluirá un apartado donde expondremos el modelo de datos de la plataforma.

El cuarto capítulo corresponde a la exposición de los resultados. En él se describe el producto obtenido, la manera en que se han combinado las tecnologías comentadas para crear una unidad funcional y la forma en la que pueden utilizarse sus capacidades.

El quinto y último capítulo incluye la discusión de los resultados obtenidos, las conclusiones alcanzadas y las lineas futuras de trabajo a seguir sobre la plataforma.

Índice general

Resumen 4

1. Introducción **10**

1.1. Motivación 10

1.1.1. Juegos sociales 11

1.2. Antecedentes 17

2. Objetivos **18**

2.1. Objetivos principales 18

2.2. Objetivos secundarios 18

3. Metodología y materiales **20**

3.1. Metodología 20

3.1.1. Escenarios 20

3.1.2. Necesidad 22

3.1.3. Casos de uso 23

3.1.4. Requisitos 25

3.1.5. Método de evaluación 27

3.2. Materiales 28

3.2.1. Tecnologías 28

3.2.2. Entorno de programación y diseño 42
3.3. Modelo de datos 43
3.3.1. Tablas de traducciones 62
3.3.2. Tablas de registro 62

4. Resultados 66

4.1. La Extensión de Lobby 68
4.2. Extensiones de juego 72
4.2.1. Ejemplo de juego: El conecta cuatro 73
4.3. Cómo desplegar un juego 73

5. Discusión, conclusiones y líneas futuras 80

5.1. Discusión de los resultados 80
5.2. Conclusiones 81
5.2.1. Revisión de objetivos 81
5.3. Líneas futuras 82

A. Glosario de términos 86

B. Diagrama del modelo de datos 88

C. Descripción de los casos de uso 92

Capítulo 1

Introducción

1.1. Motivación

A principios del año 2012, Ibermedia desarrolla las bases de un proyecto para la creación de una herramienta capaz de facilitar la creación de videojuegos sociales a los desarrolladores. El proyecto es presentado al Centro para el Desarrollo Tecnológico Industrial (CDTI) del Ministerio de Economía del Gobierno de España, que decide avalarlo y financiar su desarrollo durante dos años.

Los juegos multijugador en línea requieren de un servidor que de soporte a los múltiples clientes que se encuentran jugando en un momento dado para autenticar usuarios, crear y buscar partidas o sincronizar el estado de las mismas. La herramienta que proyecta crear Ibermedia debe contar, por lo tanto, con un servidor de juegos flexible, capaz de dar soporte a cualquier tipo de juego social.

Si bien es cierto que cada juego dispondrá de una lógica diferente a la de todos los demás y requerirá de unos servicios específicos, las peculiaridades de los juegos sociales multijugador nos permiten distinguir una serie de funciones comunes que posibilitan esta tarea. Además, como el mercado en este sector evoluciona especialmente rápido y los costes de producción permiten a los desarrolladores publicar varios juegos, un servidor funcional debe ser capaz de albergar varios juegos funcionando de manera simultánea.

En definitiva, las necesidades particulares del proyecto de Ibermedia requieren la creación de un prototipo de servidor que permita a cualquier desarrollador desplegar sus juegos de manera simultánea, proporcionando los servicios básicos que todo juego multijugador necesita y facilitando en la medida posible el agregado de los servicios y funciones inherentes a la mecánica particular propia de cada juego.

1.1.1. Juegos sociales

Para entender qué es un juego social, es necesario definir previamente el concepto de juego ocasional[1]. Gracias a la proliferación de dispositivos móviles tipo smartphone o tablet y a la aparición de consolas como la Wii de Ninetendo, los juegos ocasionales ha evolucionado rápidamente, pasando de ser un pequeño nicho de mercado a una poderosa tendencia que está cambiando no sólo el mercado de los videojuegos sino la industria del entretenimiento en general.

Anteriormente conocidos como "games for the rest of us", los juegos ocasionales no están dirigidos a los videojugadores habituales, sino al público general. Todo el mundo, desde niños a mayores, independientemente de su profesión o género, es un jugador ocasional en potencia. La *International Games Developers Association* (IGDA) define una serie de características[8]:

- **Mecánica de juego y controles extremadamente sencillos e intuitivos.** Un nuevo jugador emplea poco tiempo para decidir si continuará jugando, por lo que debe ser capaz de familiarizarse casi instantáneamente con el funcionamiento del juego.
- **Orientados a partidas generalmente breves.** El tiempo que un jugador ocasional dedica al juego se distribuye en ráfagas cortas, por lo que las partidas o niveles deben poder completarse en poco tiempo o continuarse a posteriori sin necesidad de guardar el progreso.
- **Adictivo y rejugable.** Dado que no están pensados para un juego continuado, es importante que la mecánica del juego sea adictiva y que el jugador siga teniendo una experiencia de juego agradable en las sucesivas partidas.
- **Comprensivos con el jugador.** Aunque el nivel de desafío que pueda proponer varía siginificativamente con el tipo de juego, por lo general suelen ser más permisivos con los errores que los videojuegos habituales y tienden a guiar al jugador a través del juego antes que a penalizarlo.
- **Rampa de complejidad cuidadosamente elaborada.** La dificultad del juego debe aumentar progresivamente a un ritmo que no resulte tedioso pero que tampoco desanime al jugador.
- **Tienden a ser de temática inclusiva.** Suelen disponer de una temática que pueda identificar al máximo número de usuarios para evitar, dentro de lo posible, limitar su audiencia potencial. Raramente recurren a la violencia o sexualidad explícitas.
- **Orientados al usuario que busca diversión y relajación.**

[1]Por motivos de corrección lingüística, utilizaremos el término *juego ocasional* en lugar del más popular *juego casual*, dado que este último es una traducción imprecisa del término inglés *casual game*.

La integración de los juegos ocasionales con las redes sociales ha dado lugar a la aparición de los llamados juegos sociales, que añaden además la posibilidad de compartir la experiencia de juego con los amigos ya sea para comunicarse, competir o intercambiar objetos virtuales. Estos juegos por lo general son accesibles desde las propias redes sociales, por lo que estas se convierten a su vez en excelentes canales de distribución. Existen además muchos ejemplos de juegos para smartphones y tablets que, aunque no puedan considerarse juegos 100 % sociales, también incluyen algunas de las capacidades que estas ofrecen, como por ejemplo la autenticación de usuarios mediante las credenciales del usuario en la red social y la publicación de logros y puntuaciones obtenidas en el muro o el timeline del usuario.

Los jugadores ocasionales

El fenómeno de los smartphones y las redes sociales está facilitando cada vez más el acceso a este tipo de juegos, por lo que el volumen y el perfil del público objetivo de los mismos también está evolucionando. Como ya hemos comentado anteriormente, los juegos sociales están dirigidos al consumidor masivo, al público general. Aunque un estudio de mercado de la Casual Games Asociation (CGA)[9] afirma que el perfil del jugador social medio responde al de una mujer de 40 años, lo cierto es que este perfil varía significativamente dependiendo del tipo de juego: desde niños y niñas jugando a juegos infantiles a hombres de edad más avanzada jugando al chinchón. Incluso los videojugadores tradicionales juegan de vez en cuando a juegos ocasionales buscando un poco de diversión rápida para desconectar.

Las características generales del jugador ocasional son las que siguen:

- Prefieren los juegos con controles sencillos.
- Prefieren juegos que sean fácilmente accesibles sin necesidad de configuración previa.
- Prefieren juegos que permitan jugar en sesiones cortas.
- Juegan para relajarse, matar el tiempo, socializarse o conseguir logros.
- No se perciben a sí mismos como videojugadores.
- Por lo general no juegan a juegos violentos.
- Por lo general no invierten dinero en periféricos o hardware específico para videojuegos.

A pesar de la creencia generalizada es que los jugadores ocasionales no juegan habitualmente o lo hace en sesiones cortas, existe un gran grupo de jugadores que no cumplen este estereotipo. Según un estudio de Lightspeed Research de 2010[11], el 54 % de los jugadores sociales de Facebook juega una o más veces cada día a sus juegos sociales. La mayoría de ellos, el

65 %, dedica 30 minutos o menos a cada sesión, lo que nos deja un 35 % que juega en sesiones de más de media hora. Incluso a pesar de que una partida del juego en cuestión puede completarse en pocos minutos, existe un 8 % de jugadores que dedican más de una hora a sus sesiones de juego. La tendencia por edades es que los jugadores más jóvenes dedican menos tiempor por sesión a sus juegos sociales que los jugadores de más edad.

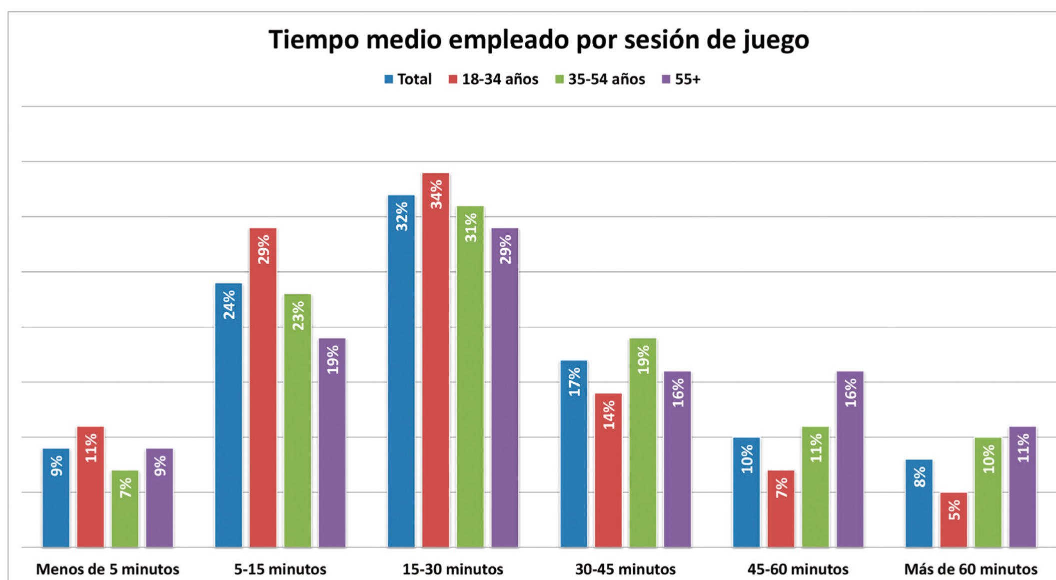


Figura 1.1: Promedio de minutos empleados por sesión de juego según rango de edad

Otra de las características del jugador ocasional es la competitividad, pero entendida de una manera diferente al de los juegos habituales. A los jugadores habituales les gusta competir cara a cara con sus amigos ya sea online o en el salón de casa, mientras que el juador ocasional desarrolla su competitividad a partir de ránkings de puntuación o en torneos, pero también compite contra sí mismo para obtener logros que le permiten desbloquear ciertos objetos o niveles especiales.

Monetización

Los juegos sociales siguen por lo general el modelo *freemium* o *free-to-play*, es decir, el usuario puede jugar de forma gratuita y, si así lo desea, paga por actualizaciones, objetos virtuales, ayudas consumibles, monedas del juego y otros tipos de contenido extra. Siguiendo este modelo de negocio, los ingresos llegan por tres vías: según datos de la GDA, el 60 % viene de los clientes, mientras que el 40 % restante proviene a partes iguales de la publicidad tradicional y de las ofertas[2].

[2]Algunas empresas ofrecen a los jugadores la posibilidad de completar encuestas o inscribirse en listas de publicidad a cambio de monedas virtuales que se pueden usar después para comprar objetos del juego.

El porcentaje de usuarios de un juego social que pagan por estos contenidos oscilaba entre el 1 % y el 5 % el año pasado, pero Facebook acaba de anunciar[13] que, entre marzo de 2012 y el mismo mes de 2013, el número de jugadores de pago en su plataforma ha aumentado un 24 %. Al tratarse de micropagos esporádicos, el perfil de gasto del jugador de pago es bastante heterogéneo. Así, nos encontramos que un pequeño grupo, no superior al 15 % de la base de usuarios de pago de un juego determinado, soporta más del 50 % de los ingresos de este tipo. Estos usuarios, que gastan cantidades superiores a los $25 al mes, son conocidos en la industria del juego ocasional como whales(ballenas) debido a que tienen un peso significativo en los beneficios. Con un peso de alrededor del 25 % de los ingresos, destinando entre $5 y $10 mensuales a la compra de contenido adicional, encontramos a los usuarios de pago moderados, que representan el 25-40 % del total de usuarios de pago. La mayoría de jugadores premium, en cambio, gasta menos de $5, y genera menos del 15 % de los ingresos.

La cantidad media que un usuario gastará en un juego social también varía dependiendo del género del juego: mientras los juegos de casino y RPGs generan de 5 a 10 céntimos de dolar al mes por usuario, los jugadores de juegos de arcade, de simulación o los populares juegos de caretaking, similares a FarmVille, generan entre 1 y 5 céntimos mensuales. El término medio lo encontramos en los juegos del tipo aventura gráfica o torneos, que ingresan entre 3 y 7 céntimos por usuario y mes.

Otro modelo de negocio, más común en el sector de juegos para smartphones y tablets, sobretodo sobre la plataforma iOS de Apple, consiste en el abono de una cantidad moderada (entre $3 y $10) por descargar y jugar al juego. En este caso es habitual que exista una versión de demostración gratuita, que permite jugar un número limitado de partidas o niveles, para permitir al jugador que pruebe el producto antes de comprarlo. Además, no es extraño que, al igual que en los juegos freemium, exista también contenido extra que requiera de un pago adicional.

Mercado

Un informe de IBISWorld[12] sitúa el sector de los juegos sociales entre los 8 mercados que ofrecen mayores oportunidades para nuevas empresas en 2013. La baja barrera de inversión necesaria, la expansión de las redes sociales y un internet cada vez más accesible han permitido que durante los últimos 5 años el mercado estadounidense en este sector crezca una media del 184.3 % interanual. En términos globales, se espera que durante este año los ingresos de la industria asciendan a 7.49 miles de millones de dólares y que el año que viene se disparen hasta los 8.64 miles de millones, pudiendo superar por primera vez al mercado de vidojuegos de consola tradicionales si las ventas en este sector continúan descendiendo.

Aunque Facebook es la mayor red social a nivel mundial, hay que tener en cuenta que existen muchas otras redes sociales de ámbito más localizado que en sus países, en muchos casos, tienen incluso más éxito que Facebook. En España, por ejemplo, tenemos el caso de

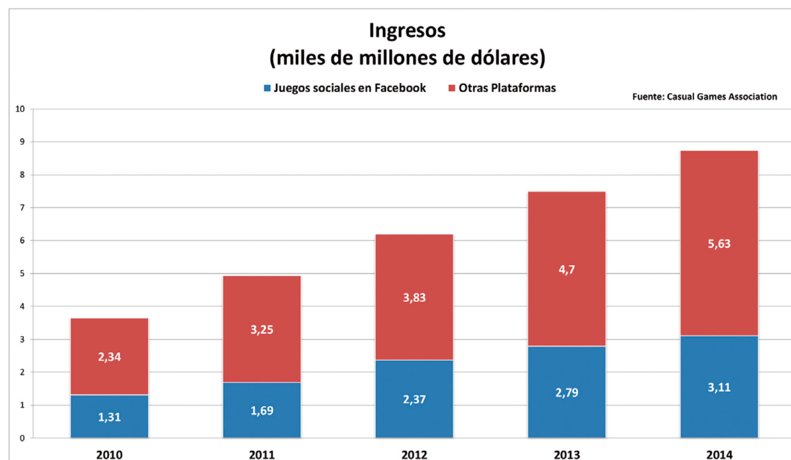


Figura 1.2: Facturación total de la industria de juegos sociales

Tuenti, pero existen otras más potentes a nivel de base de usuarios, como la China QZone y la rusa VKontakte. El efecto de esto es que la mayoría de los beneficios que obtiene la industria proceden de plataformas ajenas a Facebook[10]. Esta tendencia se espera que se siga acentuando a medida que los desarrolladores vayan moviendo sus juegos bien a sus propias plataformas o bien expandiéndose a otras redes sociales locales o menos masificadas como Google+.

Geográficamente hablando, el atractivo de una región viene determinado por tres factores: una fuerte presencia de smartphones y acceso a internet de alta velocidad, alta penetración de redes sociales que cuenten con una audiencia atractiva y buena disponibilidad de sistemas que faciliten las compras online. Quizá por ello no resulte una gran sorpresa que el líder mundial sea el mercado Asiático, responsable del 40 % de los ingresos mundiales durante 2011. Le siguen en importancia Norteamérica (28 %) y Europa(23 %).

En la actualidad, Brasil y, sobretodo, Rusia destacan como mercados emergentes. En ambos casos se trata de países geográficamente muy extensos con amplias áreas rurales a las que no siempre es sencillo llevar el acceso a internet. De 12.3 millones de jugadores sociales que había en el país carioca en 2009 se estima que en 2015 se lleguen a los 58.7 millones y, en el caso de Rusia,se espera pasar de 12.9 a 69.5 millones de usuarios. En cuanto a ingresos se esperan crecimientos de 51 a 257 millones de dólares y de 31 a 215 millones respectivamente. España resulta también un caso interesante: a pesar de ser un país relativamente pequeño, presenta unos porcentajes de crecimiento similares a los de Brasil y Rusia, y una media de ingresos por jugador cercana a la de Alemania, que es líder indiscutible en este aspecto fuera de los mercados asiático y norteamericano.

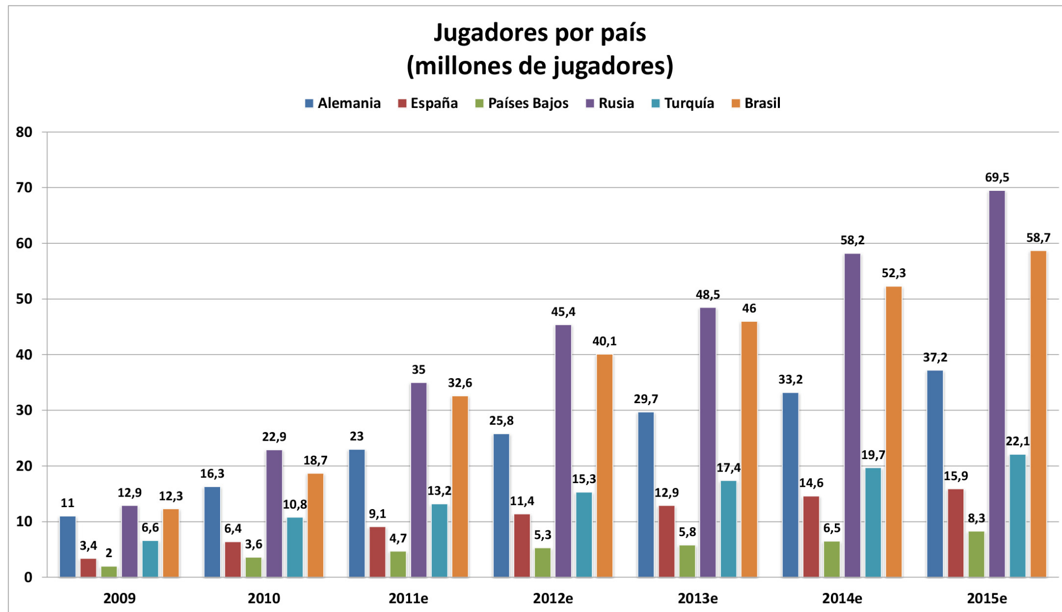


Figura 1.3: Cantidad de jugadores por país (Excluyendo EE.UU. y Asia)

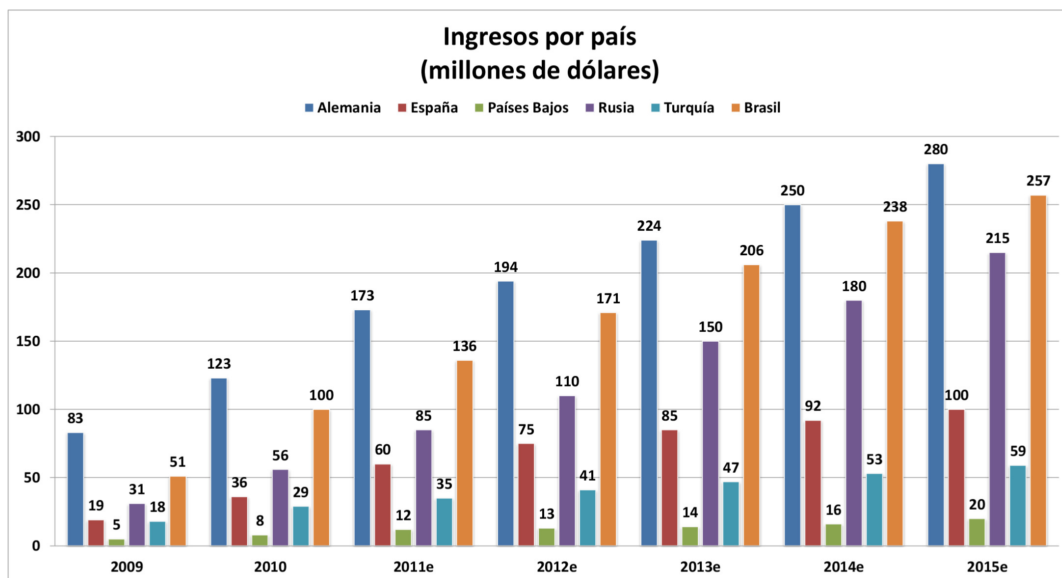


Figura 1.4: Ingresos por país (Excluyendo EE.UU. y Asia)

1.2. Antecedentes

La cantidad de empresas que se dedican en la actualidad al desarrollo de juegos sociales es innumerable. Desde startups independientes que han conseguido medrar con diferente grado de éxito y apoyo por parte de terceras compañías hasta grandes empresas del videojuego tradicional que se han introducido en el sector social ya sea por medios propios o adquiriendo alguna de las startups anteriormente mencionadas. Muchas de ellas suelen ofertar más de un juego, en varias redes sociales y con versiones para smartphone y/o tablet.

En esta tesitura, una herramienta que facilite el trabajo de los desarrolladores puede ser de gran ayuda para aquellos que están estudiando introducirse en la industria de los juegos sociales y no acaban de decidirse por no disponer de los medios técnicos o económicos para hacerlo de la manera tradicional. SmartFoxServer 2X es una tecnología desarrollada en este sentido, que permite organizar a un grupo de usuarios en diversas salas (partidas) dentro de un servidor y que en la actualidad está siendo utilizada, tanto por desarrolladores independientes como por gigantes de la industria como Electronic Arts o Sony Online, para crear plataformas para dar servicio online a sus juegos.

El proyecto de Ibermedia pretende ir un paso más allá y construir una plataforma abierta para terceros desarrolladores, cuyo embrión será el prototipo de servidor que se describirá a lo largo de este proyecto.

Capítulo 2

Objetivos

2.1. Objetivos principales

La finalidad del proyecto es, como ya se ha dicho, la creación de un servidor de videojuegos flexible y capaz de ejecutar varias aplicaciones simultáneamente. La plataforma que se obtenga debe estar orientada no sólo a ser un servidor de juegos funcional, sino a ser también una herramienta que permita desplegar nuevos juegos de manera rápida y eficaz. Teniendo en cuenta esto, podemos expresar dos objetivos principales:

- **Desarrollar un servidor para dar soporte online a juegos sociales multijugador.** El servidor debe proporcionar todas las funciones que un juego multijugador necesita, incluyendo persistencia, y ser compatible con cualquier plataforma social o móvil.
- **Desarrollar un conjunto de clases que, a modo de API, faciliten la creación de extensiones para la plataforma.** De esta manera añadiremos a las funciones básicas de las que dispone el servidor, una serie de funciones propias de la lógica de cada juego que se quiera desplegar sobre la plataforma.

2.2. Objetivos secundarios

Para poder alcanzar los objetivos principales, se han marcado una serie de objetivos secundarios:

- **Estudiar y familiarizarse con la tecnología de SmartFoxServer2x (o SFS2X) y sus librerías para Java.**

- **Aprender a utilizar el Framework de Spring para el desarrollo de proyectos en Java Enterprise Edition.**
- **Aprender a utilizar una herramienta ORM como Hibernate.**
- **Familiarizarse con las herramientas que ofrece Maven para facilitar la gestión y compilación de proyectos en Java.**

Capítulo 3

Metodología y materiales

En este capítulo detallaremos el procedimiento seguido para la realización del proyecto. Empezaremos analizando un par de escenarios de aplicación, a partir del cual valoraremos cuáles pueden ser los requisitos que deberá reunir la plataforma objeto de este proyecto. Con estos requisitos como base, se podrá construir el modelo de datos a partir del cuál se desarrollará toda la plataforma y que se describirá con detalle al final del capítulo.

A continuación se describirán las tecnologías empleadas en el proceso de desarrollo del proyecto, tanto en la fase de programación del servidor como en el ejemplo del juego de billar, justificando su elección frente a otras soluciones. También se detallará en el siguiente apartado los equipos y entornos de programación que se han utilizado.

3.1. Metodología

3.1.1. Escenarios

Escenario 1: El jugador social

Alicia, una usuaria de Facebook, lleva cuatro horas seguidas estudiando y necesita despejarse un rato. Le encanta jugar al billar, pero no le apetece desplazarse a su local favorito a jugar, así que decide buscar un juego de billar virtual en su red social. Al entrar en el juego por primera vez le aparecerá la habitual ventana con las condiciones de privacidad requeridas por el juego, que aceptará sin siquiera leer a pesar de que su hermano Berto, ingeniero de Telecomunicaciones, siempre le advierte sobre la importancia de leer y comprender bien todos los avisos de privacidad en la red antes de aceptarlos. Tras elegir un nombre de usuario y sin requerir ninguna configuración adicional, Alicia recibe una cantidad de monedas virtuales y entra

en el Lobby del juego, la pantalla inicial donde todos los jugadores conectados pueden chatear, crear nuevas partidas y torneos o unirse a las ya existentes. Como tiene prisa por jugar, elige la opción de buscar partida rápida y el servidor le emparejará inmediatamente con un usuario de nivel similar al suyo. La mecánica del juego es muy sencilla y aprende a jugar en apenas un par de jugadas. Tras tres partidas más decide dejarlo por hoy y volver al estudio para un último repaso.

Al día siguiente se levanta temprano por los nervios. Tras asearse y desayunar, decide matar el tiempo hasta que sea hora de coger el autobús hasta la universidad jugando un par de partidas más al billar. Esta vez entra al juego directamente, pues su nombre de usuario, al igual que las estadísticas y monedas virtuales que recibió ayer, se han almacenado en un perfil que el servidor del juego ha creado para ella en su base de datos. Tras recibir las monedas virtuales de regalo diarias y ganar un par de partidas, se da cuenta que tiene dinero suficiente para comprar algún objeto en la tienda, así que decide abrir la pestaña correspondiente y compra un estampado su personaje de dibujos animados preferido para el tapete de su mesa de billar virtual. Como ya se va acercando la hora, apaga el ordenador y se dirige a hacer su examen.

El examen le va bien y, como era el último de la carrera, ella y sus amigas deciden marcharse de excursión el fin de semana para celebrarlo. En alguna de las conversaciones sale el tema del juego y su amiga Cristina, que es muy competitiva, toma buena nota. Alicia regresa cansada de la excursión y decide darse una ducha e ir a dormir. Al entrar a Facebook a la mañana siguiente encuentra en su muro un mensaje del juego de billar avisándole de que su amiga Cristina le ha enviado un regalo virtual. Entra al juego y descubre que su amiga le ha enviado un objeto consumible que le permite disponer de una guía de ayuda para apuntar válido para sólo una partida y decide enviarle su propio regalo en respuesta. Además, descubre con horror que Cristina no sólo ha entrado al juego sino que ha superado su puntuación. Gracias a la barra de amigos del juego, descubre que Cristina se encuentra conectada en este momento y decide enviarle un reto para recuperar su posición en la cabeza del ránking de amigos, que esta acepta y ambas empiezan a jugar el duelo definitivo sobre la nueva mesa de Alicia con su flamante estampado.

Unas semanas después, Berto va a visitar a su hermana Alicia y la encuentra jugando a su nueva afición. En este tiempo, Alicia ha conseguido comprar también un juego de bolas personalizadas con los colores de su equipo de baloncesto favorito. A Berto le pica el gusanillo y empieza a jugar también. Impaciente por tener una mesa como la de su hermana, decide comprar un pack de monedas virtuales del juego con su tarjeta de crédito y adquiere tres estampados diferentes para su mesa, de entre los que podrá elegir uno en todo momento, y un taco con la forma de una espada láser.

Con el tiempo, más amigos del entorno de Alicia van entrando al juego. Unos jugarán más habitualmente que otros, algunos comprarán contenido extra y otros no, pero todos podrán recurrir al billar cuando busquen un momento de ocio y competir con sus amigos y otros usuarios de Facebook.

Escenario 2: El desarrollador

Berto, un ingeniero en telecomunicaciones aficionado a los videojuegos, siempre ha tenido la inquietud, compartida con su grupo de amigos, de crear algún día un juego que cumpliese todas sus espectativas. Precisamente, en una de sus conversaciones cotidianas, uno de sus mejores amigos, Daniel, le comenta que está trabajando en el desarrollo de un juego sencillo de ajedrez para dispositivos Android y surge la idea de crear un juego social en el que los jugadores pueden retar a otros usuarios, ascender en la clasificación de mejores jugadores y adquirir diferentes diseños para el tablero y las piezas.

Para llevar a cabo esta tarea necesitarían disponer de un servidor que contase con una base de datos donde guardar toda la información relativa a los usuarios, las relaciones de amistad entre los mismos, su estadística de victorias o los conjuntos de fichas que puede utilizar, así como de un motor de sockets que gestione las conexiones provinientes de los clientes. Además necesitarán una serie de funciones para mantener la sesión de los usuarios conectados y permitir que estos creen partidas nuevas o se unan a las de otros jugadores.

Investigando por internet, Berto descubre la tecnología de SmartFoxServer, que puede servirle de gran ayuda en muchas de estas tareas, pero aún necesitará diseñar el modelo de datos y toda la arquitectura de la capa de persistencia. Una vez terminada la capa de datos, será necesario combinar las funciones de estas con las de SmartFoxServer para crear una aplicación de servidor que funcione. Tras varios meses de desarrollo, al fin, Berto y Daniel consiguen publicar su primer juego.

Unos meses después, Daniel llega con otra de sus magníficas ideas: un juego de hundir la flota. Al principio Berto se muestra un poco reticente, pues no le parece que el éxito moderado del juego de ajedrez compense el esfuerzo que realizaron para publicarlo, pero tras pensarlo unos segundos se da cuenta de que puede aprovechar gran parte del trabajo realizado para su primer juego: la autenticación de los usuarios, la creación y búsqueda de partidas o la compra de, en este caso, diferentes diseños para los barcos de la flota requiere de un conjunto de procesos y tablas de base de datos idénticos o, cuanto menos, muy similares a los empleados en el juego de ajedrez.

Tras ponerse manos a la obra descubren que, efectivamente, el desarrollo del servidor para el hundir la flota resulta mucho menos tedioso, pues pueden aprovechar todas las clases programadas para el juego anterior, modificande sólamente las que se encargan de gestionar la propia mecánica del juego.

3.1.2. Necesidad

La mayoría de desarrolladores de videojuegos sociales disponen de un amplio catálogo de juegos con diferente grado de éxito. Esto es así porque diversificando sus productos aumenta

el espectro del público objetivo y, con ello, las probabilidades de éxito de la empresa, pero también es posible gracias a que el coste de desarrollo se reduce tras la publicación del primer juego.

Esta reducción de costes se debe a que las plataformas de juego multijugador, por lo general, funcionan de la misma manera para casi todos los juegos y, una vez desarrollada la plataforma para el primer juego, es posible reutilizar sus funciones para todos los demás. Todos los desarrolladores disponen de sus propias plataformmas que dan soporte a todos sus juegos, pero en todos los casos se trata de plataformas privadas, por lo que toda empresa que pretenda adentrarse en el mundo de los juegos sociales debe desarrollar su propia solución, a pesar de que no será, en esencia, muy diferente de las del resto.

Si pretendemos crear una serie de herramientas para democratizar aún más el desarrollo de juegos sociales, sería bueno comenzar por desarrollar una plataforma abierta que permita desplegar todo tipo de juegos multijugador, ahorrando a los desarrolladores la tarea de programar su propia plataforma y centrarse directamente en la lógica de servidor de sus juegos.

3.1.3. Casos de uso

A continuación, describiremos los servicios básicos de los que dispondrá nuestra plataforma para cada uno de los actores implicados

Jugador/Aplicación Cliente

Cada uno de los juegos dispondrá de una aplicación cliente a través de la cual los jugadores accederán a los servicios del servidor. Estos casos de uso corresponden a las funciones básicas de la plataforma multijugador que hemos considerado comunes a todos los juegos. El desarrollador que pretenda crear una aplicación para nuestra plataforma podrá, por tanto, olvidarse del desarrollo de estas funciones y centrarse en implementar los servicios particulares que necesite para desarrollar la mecánica de su juego.

- Crear un usuario nuevo.
- Obtener clasificación de los jugadores.
- Obtener clasificación semanal de los jugadores.
- Lanzar una acción.
- Obtener packs de monedas disponibles.
- Obtener el tiempo de servidor.

- Responder a ping.
- Listar amigos.
- Listar logros completados.
- Crear una partida.
- Unirse a una partida.
- Unirse a una partida rápida.
- Jugar la revancha a una partida.
- Devolver jugador al lobby.
- Invitar.
- Respuesta a invitación.
- Canjear un token o código de invitación.
- Listar acciones disparadas.
- Listar acciones que restan por disparar.
- Listar los artículos de la tienda.
- Listar inventario de jugador.
- Comprar un artículo.
- Equipar un artículo.
- Desequipar un artículo.
- Listar objetos equipados.
- Loguear usuario.
- Inscribir un usuario en una zona (juego).
- Desloguear jugador.
- Desconectar jugador.

Cabe notar que cada juego que se conecte a la plataforma añadirá nuevos casos de uso que dependerán en cada caso de la mecánica del mismo y por ese motivo no los nombraremos en este punto.

Desarrollador/Administrador

El desarrollador o administrador dispone de una herramienta de administración, accesible a través de un explorador web, propia de SmartFoxServer, AdminTool, que le permite gestionar las zonas, salas y usuarios en caliente, así como configurar diferentes ajustes del servidor. Esta herramienta dispone de todas las funciones que necesitará el desarrollador para desplegar su juego sobre la plataforma.

3.1.4. Requisitos

En este apartado se repasan los requisitos críticos para el buen funcionamiento de la plataforma.

Requisitos funcionales de la plataforma

- Debe estar siempre online.
- Debe tener soporte multilingüe.
- Debe permitir dos o más juegos corriendo a la vez.
- Los usuarios de un juego no deben ser capaces de ver e interactuar con los usuarios de otro juego.
- Un usuario que entra al Lobby siempre debe encontrar algún otro usuario y al menos una partida abierta.

Requisitos funcionales de la aplicación cliente

- Debe desarrollarse en uno de los lenguajes para los que existe el API cliente de SmartFoxServer 2X (Flash Payer o AIR, Unity3D, iOS, HTML5, Java2 SE/Android, .NET/Mono, C++ o C#).
- Debe poder ser capaz de establecer conexiones TCP/UDP con el puerto 9933 o túneles HTTP al servidor.
- Debe enviar todas sus peticiones a través del API de SmartFoxServer.

3.1.4.1. Requisitos no funcionales

Son los requisitos que no resultan críticos para el funcionamiento básico de la plataforma, pero sí contribuyen a que su uso sea más amigable. Se pueden distinguir varios subgrupos de requisitos no funcionales.

Look & Feel

Determinan cómo debe ser la aplicación en cuanto a interfaz y diseño. Al ser nuestra plataforma una aplicación de servidor, no dispondrá de interfaz para los usuarios, dependiendo esta de la implementación de cada cliente. El desarrollo de la aplicación cliente escapa al propósito de este proyecto, pero los requisitos Look&Feel que esta debería reunir son los siguientes.

- Los controles de la aplicación cliente han de ser intuitivos.
- La comunicación cliente-servidor debe realizarse de manera completamente transparente para el usuario.
- El estado de una partida debería poder recuperarse desde el servidor si se produjese algún error de sincronización entre los jugadores fruto de un mal funcionamiento puntual de la aplicación cliente.

Usabilidad

Se refieren a las características que debe cumplir la plataforma para que sus servicios puedan ser accedidos sin problema.

- Los nombres de las peticiones de servicio deben ser sencillos y autoexplicativos.
- Los datos adjuntos han de ir encapsulados en estructuras de datos conocidas por el cliente.

Prestaciones

Son los requisitos que se han de cumplir en cuanto a estabilidad y capacidad de respuesta o de almacenamiento.

- El tiempo de respuesta a peticiones ha de ser mínimo.
- El tiempo de acceso a datos ha de ser lo más pequeño posible.

- Un error interno de servidor debe afectar lo mínimo posible a la experiencia de usuario del jugador que ejecuta el cliente.
- El servidor debe ser capaz de recuperarse rápidamente de una caída repentina.

Mantenimiento

Se refieren a los rasgos de la aplicación que permiten realizar cambios en el sistema, instalar actualizaciones y desplegar nuevas versiones.

- El despliegue de una nueva versión de la extensión de servidor debe poder hacerse en caliente sin que el usuario llegue a percibirlo.
- En caso de actualizaciones críticas, el tiempo de inactividad del servidor debe ser mínimo.
- El despliegue de una nueva versión de la extensión de servidor debe poder hacerse en caliente o, en caso de actualizaciones complejas, con un tiempo de caída mínimo.

Seguridad

La capa de red de SmartFox ya implementa muchos mecanismos de seguridad, pero la implementación de la plataforma debería incluir algún aspecto adicional:

- El servidor debe aceptar, como máximo, la conexión de un usuario por IP y juego.

3.1.5. Método de evaluación

Como los juegos sociales son aplicaciones que se ejecutan en entornos un tanto difíciles de predecir, hemos decidido que la mejor manera de testear la plataforma y los juegos que para ella se realicen será sometiéndolos a un primer periodo de prueba alpha, realizado por miembros del grupo de desarrollo, y posteriormente a un periodo de prueba beta (abierto o cerrado según el caso) en el que participarán jugadores ajenos al equipo, que serán capaces de generar un patrón de tráfico y conductas mucho más reales y reveladores a efectos de detección y corrección de errores.

3.2. Materiales

3.2.1. Tecnologías

SmartFox Server

SmartFoxServer es una tecnología cliente/servidor multiplataforma diseñada para facilitar el desarrollo de aplicaciones multiusuario, desarrollada por la empresa italiana gotoAndPlay(). Aunque se puede usar para crear aplicaciones de todo tipo, el componente de servidor está firmemente orientado y optimizado para el desarrollo de juegos multijugador. En este proyecto utilizaremos la segunda versión de la plataforma, SmartFoxServer 2X (conocido también por sus siglas, SFS2X).

La filosofía de SmartFoxServer se desarrolla en torno a tres conceptos clave: simplicidad de uso, versatilidad y rendimiento. En este sentido, dispone de APIs intuitivos, ágiles y ricos tanto para cliente como para servidor y de una arquitectura en capas que permite a los desarrolladores una mayor flexibilidad a la hora de construir sus aplicaciones. Su instalador unificado permite instalar el software en plataformas Windows (32/64 bits) y Linux (32/64), ya sea en un servidor convencional o en la nube como plataforma Software como Servicio (SaaS).

A nivel de red, SmartFoxServer 2X integra BitSwarm 3.0, un potentísimo motor de sockets TCP/UDP optimizado para juegos multijugador. BitSwarm 3.0 ha superado en todas las pruebas a otras soluciones de ámbito más general, como JBoss Netty o Apache Mina, tanto en estabilidad como en rendimiento, manteniendo varios miles de sockets con un consumo mínimo de memoria y prácticamente sin añadir latencia. Gracias a su protocolo binario, SmartFox es capaz de aprovechar mejor los recursos del servidor y la red, reduciendo el tamaño hasta 6 veces y llegando a dividir por 5 los tiempos de codificación.

BitSwarm también proporciona de una serie de mecanismos para situaciones especiales, como la tecnología BlueBox, que permite que usuarios con los puertos bloqueados por un firewall puedan conectarse al servidor mediante túneles HTTP o mecanismos para recuperar conexiones caídas que funcionan de manera totalmente transparente al desarrollador, y de seguridad, como filtros anti-flooding para evitar ataques DoS, filtros de IPs y codificación de la información transmitida a nivel de socket.

SmartFox es capaz de soportar varios juegos simultáneamente sobre el mismo servidor, aislados unos de otros, gracias a su estructura organizada en zonas y salas. Cada uno de los juegos

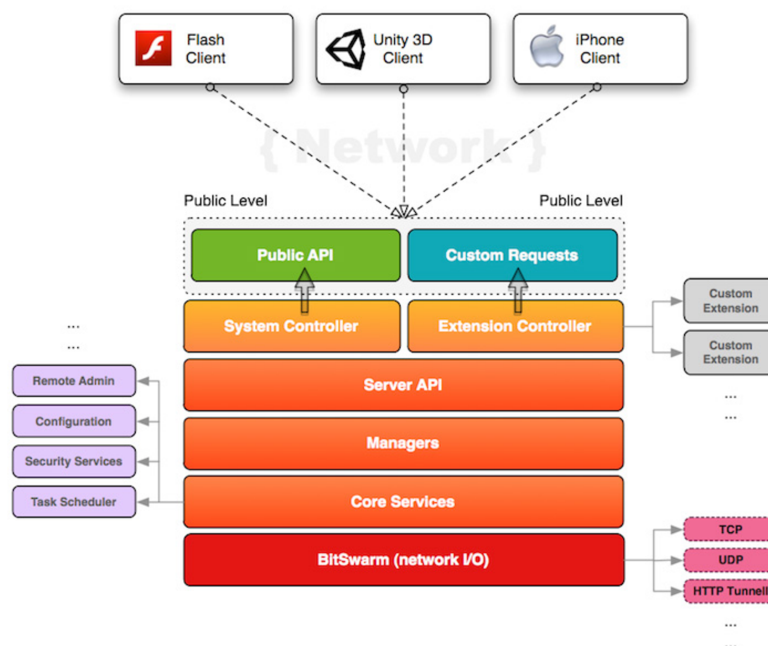


Figura 3.1: Pila de SmartFoxServer 2X

o aplicaciones de Smartfox funciona en una de estas zonas, y cada zona, a su vez, consta de una o varias salas. En la mayoría de juegos existe una slaa por defecto llamada Lobby, que es el lugar en el que todos los usuarios se encuentran cuando han entrado al juego pero no han empezado ninguna partida. En el Lobby pueden chatear con otros usuarios, configurar las opciones del juego, acceder a la tienda, crear partidas, retar a otros usuarios o unirse a partidas abiertas. Las salas, por su parte, son el lugar en el que se desarrolla el juego propiamente dicho, aunque pueden destinarse también a otros fines, como por ejemplo salas de chat privadas. Algunos juegos dispondrán además de una sala de juego permanente, mientras que otros requerirán que las salas sean creadas dinámicamente a medida que los jugadores crean y terminan partidas.

Los componentes de cualquier aplicación que corra sobre SFS2X debe heredar de los componentes básicos que ofrece la plataforma, de manera que estos se comporten tal y como el servidor espera que lo hagan. El componente principal de toda aplicación de SmartFox es la extensión. Todas las extensiones heredarán de la clase SFSExtension y se encargarán de implementar la lógica de la aplicación: podrán reaccionar ante los eventos que se produzcan en el servidor mediante handlers de eventos, que heredarán de BaseServerEventHandler, y capturar peticiones de los clientes mediante clases que extiendan BaseClientEventHandler. Todos los componentes así construidos podrán interactuar con las capas inferiores del servidor de SmartFox de manera natural y manejar las estructuras de datos y mensajes propios de la plataforma de manera transparente al desarrollador.

Para conectar una aplicación a SmartFox sólo debemos asociar su extensión a una

zona o sala desde la herramienta de administración del servidor. Cada zona o sala tendrá asociado como mucho una extensión y, cuando sean iniciadas por el servidor, arrancarán también automáticamente su extensión ejecutando el método init() de la clase que la implementa. Para facilitar la localización de esta clase, se recomienda que toda clase que implemente una extensión incluya el sufijo *-Extension* en su nombre.

Desde la herramienta del administración del servidor podemos asociar a la extensión un fichero de configuración (por defecto *config.properties*) y/o un filtro de palabras, así como ajustar diversos parámetros de las zonas y salas. Estos ficheros deben encontrarse en el mismo directorio que el fichero .jar con las clases de la aplicación. SmartFox organiza todos los ficheros relativos a extensiones en una carpeta llamada Extensions, en la que deberemos crear un directorio (por ejemplo *Extensions/miApp*) para colocar allí todos los ficheros de nuestra aplicación. Los ficheros .jar con las librerías de terceros que utilice cualquiera de nuestros componentes pueden colocarse también en este directorio, pero de esta manera sólo serían accesibles desde nuestra aplicación. Si queremos que una librería pueda ser utilizada por aplicaciones diferentes situadas en diferentes directorios debemos guardarla en la carpeta *Extensions/lib*.

Por su parte, el lado de cliente también cuenta con su propia API, que se comunicará con la parte pública del servidor. El cliente puede solicitar a través de ella una lista de usuarios conectados, las salas que se encuentran abiertas en cada momento o una serie de datos que se encuentren almacenados en la base de datos. Los mensajes entre servidor y cliente se transmiten a través de túneles TCP o UDP y se sirven de objetos SFSObject y SFSArray (estructuras de datos propias de SFS2X que soportan hasta 20 tipos diferentes de daros) para transportar datos de un lado a otro de la aplicación. Las peticiones y respuestas que se intercambiarán entre el cliente y las extensiones personalizadas que puedan correr sobre el servidor deben definirse claramente al desarrollar la aplicación. La API de cliente dispone de librerías para varias plataformas:

- Flash platform (Flash/Flex/AIR)
- Unity3D (standalone, web-player, iOS, Android)
- iOS 4-5-6
- HTML5
- Java2 SE and Android
- .Net/Mono
- C++
- C#

Java EE

Java es un lenguaje de programación orientado a objetos propiedad de Oracle Corporation, diseñado para desarrollar programas que corran sobre una máquina virtual Java sin importar la arquitectura subyacente. Esto permite a los desarrolladores generar programas que pueden ser ejecutados en cualquier plataforma sin necesidad de volver a ser compilados, lo que ha permitido que se convierta en uno de los lenguajes más populares en la actualidad.

Java Enterprise Edition o Java EE es la versión de Java para empresa, orientada a construir aplicaciones escalables de servidor distribuidas en varias capas. Incluye una serie de beneficios, especificaciones para API y componentes que facilitan la tarea a los desarrolladores, permitiéndoles centrarse en la lógica de negocio sin tener que preocuparse de las funciones de bajo nivel. En este proyecto usaremos la última versión del SDK de Java EE, que es Java EE 6.

Una de las cosas más interesantes de Java EE es la gran variedad de alternativas a las tecnologías que emplea de serie desarrolladas por su extensa comunidad de desarrolladores. Muchas de estas soluciones son de código abierto y surgen para reducir la complejidad de muchos de los componentes de Java EE, pero también para proporcionar enfoques alternativos a determinados problemas. En el presente proyecto usaremos algunas de estas alternativas para tareas tales como la inyección de dependencias o la persistencia de datos.

ORM / Hibernate

A la hora de utilizar un lenguaje orientado objetos como Java con un sistema de base de datos relacional surgen una serie de problemas debidos a una serie de incompatibilidades derivadas de las características inherentes a la forma de organizar e interpretar la información en ambos entornos. Para dotar a nuestra aplicación Java de soporte para bases de datos relacionales disponemos del API de JDBC, que permite establecer una conexión con la base de datos, ejecutar sentencias SQL contra ella y recuperar y persistir datos desde y hacia la misma. Anque se trata de una muy buena solución para transmitir información entre Java y la base de datos (o RDBMS[1]), la representación de la misma en el mundo orientado a objetos y el mundo relacional presenta una serie de diferencias:

- **Conceptos de identidad e igualdad:** En Java existen dos definiones para la identidad y la igualdad de objetos: dos objetos se consideran idénticos si apuntan a la misma referencia de memoria (a==b), e iguales si contienen los mismos datos sin importar a la posición de memoria que referencien (a.equals(b)). En un RDBMS, los datos se representan en filas, y cada fila se identifica en base a su contenido: dos objetos Java que no son idénticos porque apuntan a registros de memoria diferentes pueden ser considerados idénticos en la base de

[1] Relational DataBase Management System.

datos porque contienen los mismos datos. Este problema puede solucionarse incluyendo una propiedad extra en la clase del objeto y una columna extra en la tabla correspondiente con un número de id único para identificar a cada objeto.

- **Mapeo de la herencia:** En todo lenguaje orientado objetos existe el concepto de herencia, que es completamente desconocido para el RDBMS. Es necesario definir una estrategia clara para traducir la herencia entre dos clases al esquema de la base de datos. Una estrategia válida puede ser usar una tabla para la superclase y otra para la sublcase con los campos que se han agregado, relacionadas con el id del objeto. En este caso, los objetos de la superclase se almacenarían sólo en la primera tabla, mientras que los de la subclase se almacenarían en ambas.

- **Mapeo de asociaciones:** En una aplicación Java los objetos están asociados unos con otros. Estos objetos pueden ser entidades (objetos con su propia identidad de persistencia) o valores (no disponen de identidad de persistencia). Un objeto valor asociado con un objeto entidad se almacenará en la misma tabla que este como otro dato cualquiera, por lo que esta situación no presentará problema alguno. En cambio, cuando la asociación se da entre dos objetos entidad la solución no es tan trivial. La persistencia de un grupo (o grafo) de objetos entidad debe hacerse de manera que pueda recuperarse posteriormente. Una estrategia común consiste en persistir cada entidad en su propia tabla y establecer una relación entre ambas tablas mediante una columna con el valor del campo clave de la otra tabla. Cuando la relación entre ambas entidades es de muchos a muchos, esta solución se complica aún más, siendo más sencillo cometer algún error.

El mapeo objeto-realacional, más conocido como ORM, es una técnica de programación que viene a solucionar este problema. Por lo general, las soluciones ORM funcionan sobre JDBC y actúan como mediador entre el entorno orientado a objetos y el entorno relacional de forma totalmente transparente al desarrollador, mediante el mapeado de las clases persistentes en un RDBMS. La forma en que las propiedades de cada clase, la herencia entre las mismas y sus relaciones son mapeadas en tablas y columnas de la base de datos se puede definir en un fichero XML, metadatos o anotaciones en el propio código de Java.

Desde el punto de vista de la aplicación, ORM permite tratar la base de datos como si fuese un almacén de objetos persistentes: convierte los objetos de entidades a parámetros de una sentencia JDBC cuando son persistidos, y transforma de vuelta los resultados a su representación de objetos cuando se realiza una consulta. Esto permite abstraer por completo la lógica de negocio de cualquier cuestión relativa a la capa de persistencia, ahorrando al programador hasta un 95 % del código orientado a estas tareas.

Java EE dispone por defecto del estándar Enterprise Java Beans (EJB) para acometer las tareas de persistencia, pero se trata de una solución compleja y poco transparente y, además, precisa de un contenedor para los beans de las entidades, por lo que los desarrolladores prefieren, por lo general, usar otros ORMs. En el extremo opuesto encontramos Hibernate, una solución

ORM de código abierto fácil de aprender, comprensiva, que no requiere de servidor, bien documentada y con muchos recursos disponibles, lo que le convierte en uno de los frameworks favoritos de los desarrolladores y en el que usaremos en este proyecto.

La configuración de Hibernate se lleva a cabo mediante un fichero XML ubicado en la raíz del proyecto, que contendrá todos los datos relevantes de la base de datos: su nombre, la contraseña, la URL donde se encuentra, el driver que Java debe utilizar, el dialecto de SQL empleado en la base de datos, etc. Aunque también se puede configurar mediante programación, esta solución es la más común en proyectos reales dada su mayor flexibilidad.

Otro tipo de fichero importante en Hibernate son los que contienen las definiciones del mapeado: en nuestro proyecto deberíamos tener uno de estos ficheros por cada entidad persistente, y se recomienda que estén situados en la misma carpeta que la clase de la entidad a la que pertenecen. Sin embargo, en este caso, prescindiremos de estos ficheros y definiremos el mapeo de nuestras clases persistentes mediante anotaciones en Java. Este método se introdujo pro primera vez en el API de persistencia de Java (JPA) de la versión de EJB que acompañaba a Java EE 5, y fue rápidamente adoptada por Hibernate. Las anotaciones proporcionan una manera potente, sencilla y clara de mapear las entidades de un proyecto, a la vez que permiten al desarrollador librarse de la tarea tediosa y propensa a errores de escribir y mantener un fichero XML por cada una de las clases persistentes. Para que Hibernate interprete correctamente estas anotaciones es necesario configurarlo para trabajar con JPA.

Los objetos persistentes que manejará Hibernate, por su parte, no precisan de una configuración especial, salvo por el hecho de que tienen que cumplir una serie de normas POJO[2]. Los POJO son clases de Java sencillas, una versión simplificada de las JavaBeans empleadas en EJB que, a diferencia de estas, pueden ser utilizadas en cualquier capa de la arquitectura de la aplicación. Se definen como meras estructuras de datos, sin más componentes que una serie de atributos accesibles mediante métodos *setter* y *getter*. Para ser poder ser utilizados en Hibernate, como ya se ha dicho, estos POJO han de seguir una serie de reglas, algunas obligatoiras y otras opcionales:

- **Implementar un constructor sin argumentos (obligatoria).** Permite que Hibernate genere proxies en tiempos de ejecución para poder realizar algunas de sus funciones, como por ejemplo el *lazy-loading*.
- **Proporcionar métodos para acceder a todas las propiedades de la clase (opcional).** Las propiedades de los POJO se definen normalmente como variables privadas, y

[2]Plain Old Java Objects

para poder ser accedidas desde otras partes del código deben disponer de métodos *getter* y *setter*. Estos métodos deben tener el mismo nombre que la propiedad a la que acceden, seguida del prefijo *get* o *set* según sirvan para recuperarla o modificarla (por ejemplo, la propiedad *xxx* dispondrá de sus métodos *getXxx* y *setXxx*). La única excepción a esta regla son las propiedades booleanas, que sustituirán el método *get* por un método *is* (*isXxx*).

- **Definir clases *nonfinal* (opcional).** Para que Hibernate pueda generar proxies y extender las calses en tiempo de ejecución, los POJO deben estar definidos como nonfinal o implementar una interfaz de Hibernate, práctica que no se recomienda, pues al hacerlo se crearía una dependencia no deseada entre la clase POJO e Hibernate.
- **Implementar métodos *equals()* y *hashCode()* (opcional).** Estos métodos son los utilizados por HIbernate para determinar la igualdad o identidad de dos objetos. No es necesario implementarlos, salvo si la aplicación usa objetos desacoplados (objetos que ya están persistidos y se están modificando en tiempo de ejecución), pero si se implementa uno debe implementarse también el otro.

En tiempo de ejecución, los objetos relevantes para realizar las tareas de persistencia con Hibernate son Configuration (que permite acceder a la configuración y generar objetos SessionFactory que cumplan con la misma), SessionFactory (que permite crear objetos Session), Session y Transaction. Estos objetos permiten abrir conexiones con la base de datos (sesiones) y realizar bloques de operaciones que deben ejecutarse de manera atómica o deshacerse (transacciones). En nuestro caso, como ya hemos comentado, usaremos Hibernate en conjunción con JPA, por lo que en lugar de un objeto Configuration tendremos un objeto EntityManager. El EntityManager se configura a través de un fichero llamado *persistence.xml* donde se declaran todas las clases a persistir y permite crear objetos SessionFactory que funcionan de la misma manera que los que se obtienen a través de un objeto Configuration.

Hibernate dispone, además, de una memoria caché de dos niveles. La caché de primer nivel está embebida dentro de la arquitectura de Hibernate y se conoce como *transactional cache*. Su función es la de mantener los objetos persistentes de una sesión para minimizar la cantidad de accesos a base de datos. La caché de segundo nivel es opcional y permite utilizar cualquier tecnología de caché desarrollada por terceros. La función de esta caché es la de mantener los objetos persistentes utilizados más repetitivamente en cada sesión.

Spring Framework

Spring es un framework compuesto por una gran variedad de componentes que ofrecen soluciones a diferentes cuestiones y conforman una herramienta versatil y muy completa para construir aplicaciones funcionales en Java EE. Spring ayuda a obtener un código más limpio, proporcionando separación de conceptos, manteniendo los componentes de la aplicación desaco-

plados y minimazando las complejidades que pueden surgir al desarrollar aplicaciones Java EE complejas.

La amplia gama de soluciones de código abirto disponibles para la mayoría de problemas que podemos encontrar a la hora de desarrollar en Java EE requiere una manera de poder combinarlas de forma sencilla para construir una aplicación. En este sentido, Spring permite utilizar diferentes herramientas en un segundo plano, con muy pocas líneas de código y manteniendo la lógica de la aplicación completamente agnóstica a las mismas.

Los componentes de Spring se organizan en módulos, lo que nos permite incluir en nuestra aplicación sólo los que realmente necesitamos, ignorando al resto. A continuación describimos los más populares.

- **Contenedor de Dependencias/Inversión de Control.** La Inyección de Dependencias (DI), también llamada en ocasiones, aunque de manera inexacta, Inversión de Control (IoC), es la tecnología que comúnmente más se asocia con Spring. Consiste en dejar que Spring gestione las dependencias de los objetos que conforman la aplicación: en lugar de ser el objeto el que crea sus dependencias es el contenedor de Spring el que inyecta estas dependencias en los objetos en tiempo de ejecución.
- **Framework de programación orientada a aspectos (AOP).** Se trata de un complemento a la programación orientada a objetos, introduciendo el concepto de aspecto para modelar objetos reales. Los aspectos son procesos que no están directamente relacionados con la jerarquía de objetos, sino más bien con una función concreta que debe ser realizada en diferentes lugares. La programación orientada a aspectos permite dotar a las clases de una aplicación de diversas funcionalidades como mantenimiento de logs, seguridad o gestión de transacciones sin necesidad de duplicar el código y permitiendo habilitar y deshabilitar estas funciones cuando se desee.
- **Abstracción de Acceso a Datos.** Permite implementar el acceso a datos de manera consistente, abstrayendo esta capa del resto de la aplicación mediante una serie de clases auxiliares y una nutrida jerarquía de excepciones. Entre las tecnologías de acceso a datos compatibles con este módulo destacaremos, por lo que nos atañe, Hibernate.
- **Abstracción de Transacciones.** Spring también dispone de una capa de abstracción de

Listing 3.1: Clase BookLister.

```
public class BookLister{

        private BookFinderFromTextFile finder=
                new BookFinderFromTextFile("libros1.txt");
        ...
        public Book[] booksByAuthor(String author){
                List all=finder.findAll();
                for(Iterator it=all.iterator();it.hasNext();){
                        Book b=(Book)it.next();
                        if(!b.getWriter().equals(writer))
                                it.remove;
                }
                return (Book[]) all.toArray(new Book[all.size()]);
        }
        ...
}
```

transacciones que soporta tanto transacciones globales sobre JTA (Java Transaction API) como transacciones locales sobre JDBC o, en nuestro caso, Hibernate.

Inyección de Dependencias

Toda aplicación Java consiste en una serie de objetos que colaboran para realizar una tarea, y, por tanto, estos objetos dependerán unos de otros en mayor o menor medida, creando lo que se conoce como dependencias. El código del listado 3.1. muestra un caso de esta situación: una clase BookLister que depende de un objeto finder para obtener un listado de libros filtrado por autor en su método booksByAuthor. En este caso, el objeto finder será una dependencia de los objetos de la clase BookLister, que la propia clase BookLister se encarga de obtener creando una instancia de BookFinderFromTextFile.

Aunque la instanciación directa es un método perfectamente válido de obtener una dependencia, pero presenta el inconveniente de que si en algún momento deseamos cambiar la clase del objeto finder para obtener los datos, por ejemplo, de una base de datos tenemos que reescribir toda la aplicación. Para evitar este problema podemos reformular la solución, añadiendo una interfaz IBookFinder que definirá el método findAll como método común a todas las clases que implementen el objeto finder para BookLister.

Ahora es posible cambiar la clase que implementará el finder sin tocar todo el código de la clase lister, pero seguiría siendo necesario volver a compilar cada vez que lo cambiemos, puesto que la instancia de la dependencia se sigue creando en el constructor de la clase.

Listing 3.2: Interfaz para BookFinder.

```
public interface IBookFinder{
        List findAll();
}
```

Listing 3.3: Inserción de una implementeación concreta de IBookFinder.

```
public class BookLister{
        ...
        private IBookFinder finder;
        public BookLister(){
                this.finder=new BookFinderFromDB("libros.db");
        }
        ...
}
```

El problema adquiere especial relevancia en el mundo real, cuando el proyecto que se está desarrollando es grande y hay varios equipos diferentes trabajando en bloques de código diferentes con escaso conocimiento de las labores que están realizando los demás. En este caso, es deseable que las clases sean reutilizables y versátiles y puedan ser testeadas de manera independiente. Para ello es necesario que las clases estén desacopladas de sus dependencias.

La utilización de interfaces nos permite abstraer el funcionamiento de una clase de la implementación concreta de sus dependencias, pero seguimos necesitando una instancia de esa dependencia y, si queremos mantener las clases desacopladas, es necesario proporcionarla desde fuera de la clase. En estos casos es donde entra en juego la inyección de dependencias (DI).

La inyección de dependencias es una técnica que permite al entorno de la aplicación inyectar las dependencias de un objeto cuando este se crea en tiempo de ejecución. La inyección puede realizarse por constructor o mediante un método setter en la clase dependiente. Todos los objetos que han de ser gestionados deben registrarse en el contenedor de dependencias, en nuestro caso mediante el fichero de configuración de Spring (en XML), donde también se definirán los objetos (dependencias) que les serán inyectados a través de su constructor o setter. Los objetos registrados en el contenedor de dependencias se conocen como beans, y la tarea de definir cuáles son las beans que se inyectarán en otras recibe el nombre de wiring (cableado).

Desde la versión 2.0 de Spring es posible definir las beans mediante anotaciones en el código. Además, también es posible configurar Spring para que busque automáticamente las dependencias a inyectar. Cuando empleamos esta técnica, llamada autowiring, Spring escanea las clases de la aplicación en busca de aquellas que implementan las interfaces que hay definidas en la clase de la bean. Para ayudar a Spring en esta tarea podemos definir en qué ruta se han de buscar las dependencias y, en caso de tener más de una clase para cada interfaz, cuál de ellas

Listing 3.4: Inyección de dependencias basada en constructor

```
1  public class BookLister{
2          ...
3          private IBookFinder finder;
4          public BookLister(IBookFinder finder){
5                  this.finder=finder;
6          }
7          ...
8  }
```

Listing 3.5: Definición de beans en el fichero de configuración para el caso del listado 3.4

```
1
2  <bean id="lister" class="com.miprograma.BookLister">
3          <constructor-arg>
4                  <ref local="finder">
5          </constructor-arg>
6  </bean>
7
8  <bean id="finder" class="com.miprograma.BookFinderFromDB">
9          ....
10 </bean>
```

ha de ser inyectada. En este proyecto utilizaremos una combinación de estas dos técnicas.

El patrón Data Access Object (DAO)

Aunque es posible crear un objeto Session para conectar con la base de datos a través de Hibernate en cualquier lugar de la aplicación, se recomienda que todas las operaciones de persistencia sean realizadas sólo a través de determinadas clases. Siguiendo esta recomendación, en este proyecto seguiremos el patrón de diseño DAO. Este patrón recibe su nombre de las clases que lo implementan: los objetos de acceso a datos o DAOs. Como su propio nombre indica, estos objetos tienen la misión de comunicarse con la capa de datos y mantener la lógica de negocio de la aplicación separada de los asuntos de persistencia, lo que nos permitirá elaborar un código más manejable y fácil de mantener, pudiendo modificar la estrategia de persistencia de manera flexible si tener que modificar las clases de capas superiores.

El patrón define que por cada clase DAO tenemos que definir su interfaz correspondiente, encargada de definir la estructura de la clase, así como todas las operaciones de persistencia que ofrecerán. Esto permitirá, además, mantener desacopladas las clases DAO de la capa de

negocio gracias la inyección de dependencias. Típicamente tendremos una clase DAO por cada entidad persistente, y todas ellas incluirán la anotación *@Repository* para indicar al contenedor de dependencias de Spring que se trata de almacenes de datos.

Acompañando al patrón DAO solemos encontrar el patrón Service Facade. Este consiste en usar un objeto intermedio adicional, llamado objeto de servicio, entre los objetos de la capa de negocio y los DAO. Los objetos de servicio combinan los métodos de varios objetos DAO para conseguir que estos funcionen como una unidad de trabajo, realizando operaciones de persistencia más complejas. Cada clase de servicio instanciará uno o, típicamente, más DAOs a través de sus interfaces. También se recomienda que cada clase de servicio, o manager como las llamaremos en nuestra aplicación, disponga de su propia interfaz. Estas clases se registrarán en el contenedor de dependencias mediante la anotación *@Service* y, en nuestra aplicación, serán ofrecidas a las capas superiores a través de un objeto tipo Factory llamado ManagerDelegate. Todos los managers se instanciarán en el Manager Delegate a través de sus interfaces.

Facebook Graph API

Todas las redes sociales disponen de servicios web para los desarrolladores que quieran construir una aplicación con algún tipo de integración con su plataforma. Gracias a ellos, un desarrollador podrá autenticar al jugador con las credenciales de la red social, dirigir hacia la misma peticiones de datos o publicar en el muro. El interés de Ibermedia es publicar un juego embebido en una red social a corto plazo, y en este aspecto Facebook ofrece el marco más atractivo tanto por número de usuarios como por las numerosas herramientas que ofrece a los desarrolladores. En cambio, los desarrolladores que deseen publicar sus aplicaciones en otras redes sociales, como por ejemplo Google+ o Tuenti, requieren de aprobación previa por parte de los responsables de la misma antes de tener acceso a la plataforma.

Facebook ofrece estos servicios web a través de su Graph API. Se trata de un API de bajo nivel, sencilla de utilizar, basada en peticiones HTTP que puede usarse para solicitar datos del usuario, publicar mensajes en el muro y ejecutar toda una serie de funciones sociales. El sitio web de Facebook Developers ofrece varias librerías basadas en este API, pero por desgracia ninguna se ajusta a nuestras necesidades, por lo que tendremos que desarrollar una nosotros mismos.

OAuth 2.0

El protocolo OAuth 2.0 es un sistema de autorización estándar recogido en la RFC 6749 del IETF. Tradicionalmente, si una aplicación necesitaba acceder a un recurso protegido, el propietario del recurso (usuario) debía compartir sus credenciales de acceso a dicho recurso con la aplicación, lo que ocasiona una serie de inconvenientes.

Listing 3.6: Porción de código de la clase GameManager.

```
@Service
public class GameManager implements IGameManager {
	...
	@Autowired
	private IGameDAO gameDao;
	@Autowired
	private IVideoGameDAO vgDao;
	...
	@Autowired
	private IUserManager userMan;
	...
	@Override
		@Transactional(propagation=Propagation.REQUIRED)
		public Game create(...) throws GloriaException{
			Game game;
			if(idUser!=null){
				UserVideoGameProfile profile=userMan.findVideoGameProfile(...);
				VideoGame vg=vgDao.find(...);
				...
				userMan.updateProfile(...);
				game=gameDao.save(...));
			}else{
				game=gameDao.save(new Game(...));
			}
			...
			return game;
		}
}
```

- La aplicación debería almacenar las credenciales del usuario para uso futuro, por lo general una contraseña sin cifrar.
- Los servidores necesitan implementar autenticación mediante password, a pesar de las vulnerabilidades.
- La aplicación obtiene un acceso total a todos los recursos protegidos por esas credenciales, sin límite de tiempo ni permisos.
- Si el propietario del recurso necesita revocar el acceso de una aplicación determinada sólo puedo hacerlo cambiando su contraseña, lo que revocaría el acceso también de las otras aplicaciones que puedan estar accediendo al mismo recurso.

- Si la seguridad de cualquiera de las aplicaciones con permisos de acceso al recurso se viera comprometida, la contraseña del usuario y sus recursos dejarían de estar protegidos.

Usando OAuth 2.0, un usuario puede permitir a una aplicación acceder durante un periodo de tiempo limitado a uno o varios de los recursos que tiene almacenados en un servidor, manteniendo el resto protegidos. Para explicar el proceso, es necesario introducir los cuatro actores que participarán en el diálogo:

- **El propietario del recurso:** Entidad que otorga permiso para acceder a un recurso protegido. Puede ser una persona (usuario final) o una máquina.
- **El servidor de recursos:** Servidor que almacena los recursos protegidos. Acepta y responde peticiones de recursos utilizando tokens de acceso.
- **La aplicación cliente:** Aplicación que realizará peticiones de recursos en nombre del propietario del recurso y con su autorización.
- **El servidor de autorización:** Servidor que emite tokens de acceso válidos a las aplicaciones cliente una vez ha podido autenticar con éxito al propietario del recurso y ha recibido su permiso para compartir el recurso. Puede ser el mismo servidor que el servidor de recursos.

El proceso de autorización se realiza en seis fases:

1. El cliente solicita autorización para acceder a un recurso al propietario del mismo. Esta solicitud puede hacerse directamente o a través del servidor de autorización.
2. Una vez aceptada la solicitud, el cliente recibe un código de permiso por parte del propietario del recurso.
3. El cliente solicita al servidor de autorización que le canjee el código de permiso por un token de acceso.

4. El servidor de autorización autentica al cliente, valida el código de permiso y devuelve al cliente un token de acceso.
5. El cliente solicita el recurso protegido al servidor de recursos, presentando el token de acceso.
6. El servidor de recursos valida el token y, si es válido, devuelve el recurso solicitado.

Facebook, Tuenti y la inmensa mayoría de redes sociales utilizan OAuth para proteger los recursos de sus usuarios. En el caso de Facebook, para obtener acceso a los datos de un usuario es necesario adjuntar un token de acceso válido a las peticiones realizadas mediante la Graph API.

3.2.2. Entorno de programación y diseño

Equipo de desarrollo

Todo el proceso de diseño, investigación y desarrollo ha sido realizado en un equipo con las siguientes características básicas:

- Intel Core2 CPU 6300 a 1.86GHz.
- 4 GB RAM.
- Windows 7 Professional SP1 64 bits.

En este equipo se han instalado una serie de herramientas software para llevar a cabo estas tareas:

- **Eclipse IDE.**
  Eclipse es el IDE más popular a día de hoy para desarrollar proyectos en JavaSE y JavaEE. Está desarrollado por IBM, y al ser un proyecto de código abierto, permite la creación de contenido adicional por parte de terceros.
- **Maven.**
  Maven es una herramienta de software para la gestión y construcción de proyectos Java, similar en funcionalidad a Apache Ant. La configuración del proyecto se especifica en un fichero de configuración llamado *pom.xml*, o simplemente POM, donde se indica el orden en que deben ser compilados los distintos módulos y las dependencias entre ellos y con componentes externos. Además, Maven cuenta con un repositorio de librerías en

línea, desde donde podemos obtener automáticamente las que necesitemos incluir en cada módulo del proyecto sin necesidad de tener que buscarlas y descargarlas nosotros mismos. Maven dispone de plugins que permiten su utilización de manera integrada en los IDEs más populares, incluido Eclipse.

Equipo servidor

El servidor utilizado para desplegar la plataforma es una máquina virtual corriendo dentro de un servidor Fujitsu E3-1220 con Windows Server 2008. La máquina virtual cuenta con 8GB de RAM y funciona con un sistema operativo Ubuntu 12.04 LTS de 64 bits.

Este equipo tiene instaladas las siguientes herramientas:

- **MySQL.**
  Para dar soporte de datos a la aplicación se ha optado por una solución MySQL. Para poder utilizar transacciones de tipo ACID[3] emplearemos InnoDB en lugar del motor de almacenamiento por defecto de MySQL, MyISAM. La elección de esta solución viene impuesta por tratarse de una tecnología fiable y consistente, además de ser sobradamente conocida.

- **Subversion.**
  Subversion, también conocido como SVN, es un sistema de control de versiones libre que viene a sustituir al popular CVS. Permite crear un repositorio donde podemos mantener diferentes versiones del proyecto sobre el que estamos trabajando, de manera que si se comete algún error grave en algún punto del desarrollo podamos recuperar una versión anterior y retomar el trabajo desde ese punto. Al poder ser accedido a través de la red, SVN permite que varios desarrolladores trabajen sobre los mismos recursos desde sus respectivas ubicaciones, sin temor a que la calidad de los mismos se vea comprometida gracias al control de versiones.

3.3. Modelo de datos

En este apartado detallaremos el modelo de datos que se ha utilizado para la plataforma. Agruparemos por separado las tablas que se emplean para el funcionamiento básico de la plataforma de las que se utilizan para guardar un registro (log) de las actividades relevantes.

En los anexos podrá encontrarse un desplegable con el diagrama del modelo de datos.

[3] Atomicity, Consistency, Isolation y Durability, o, en castellano, Atomicidad, Consistencia, Aislamiento y Durabilidad

Achievement

En esta tabla se encuentran los logros o achievements que los usuarios de un juego pueden desbloquear alcanzando ciertos objetivos. Un logro puede ser, por ejemplo, ganar 10 partidas seguidas, conseguir 10 amigos en el juego o ganar a un jugador de un nivel superior. Cuando un jugador consigue uno de estos logros recibe una medalla en forma de icono que puede mostrar en su sala de trofeos virtual o compartir en su muro de la red social para que lo vean sus amigos.

Un logro será desbloqueado siepre a raíz de que una acción de las contenidas tabla Action sea activada.

- **idAchievement *(String)*:** Identificador único del logro.
- **idVideoGame *(Short)*:** Identificador del videojuego al que pertenece.
- **ord *(int)*:** Ordinal del logro.
- **points *(int)*:** Cantidad de puntos que se conseguirán al alcanzar el logro.
- **idGraphicResources *(Long)*:** Identificador del icono de la medalla asociada.
- **idSwfResources *(Long)*:** Animación asociada.

Action

Un jugador puede realizar dos tipos de acciones: acciones simples o sociales, relacionadas con la actividad social del jugador, y acciones de juego, relacionadas con las azañas del jugador dentro del propio juego. Ejemplos de acciones simples pueden ser darle al botón "me gusta.[en] la página el juego o invitar a un amigo. Ejemplos de acciones de juego son conseguir cierta puntuación o vencer a un jugador de nivel superior.

Algunas de estas acciones requieren ser realizadas un número de veces determinado para ser activada y poder recibir los puntos o monedas asociadas o desbloquear el logro que tienen asociado. Cuando una acción alcanza el número de usos o realizaciones requerido diremos que ha sido "disparada".

En ocasiones, algunas acciones pueden estar asociadas a conseguir un conjunto de otras acciones. Las acciones que se componen de varias acciones corresponden a un tercer tipo de acción llamadas "misiones"(M) y las relaciones de estas con las acciones asociadas se establecen en la tabla Missions.

- **idAction *(Long)*:** Identificador de la acción.

- **idVideoGames** ***(Short)*****:** Identificador del videojuego al que está asociada la acción..
- **type** ***(Char)*****:** Tipo de acción. Puede ser simple (S), de juego (G) o misión (M).
- **coins** ***(int)*****:** Cantidad de monedas que se consiguen al realizar la acción.
- **description** ***(String)*****:** Descripción de la acción.
- **level** ***(int)*****:** Nivel requerido para realizar la acción.
- **points** ***(int)*****:** Puntos que se reciben al realizar la acción..
- **usesAmount** ***(int)*****:** Cantidad de veces que es encesario realizar la acción para activarla o dispararla.
- **idGraphicResources** ***(Long)*****:** Icono asociado a la acción.
- **idAchievement** ***(String)*****:** Identificador del logro al que se asocia la acción.
- **idVGAchievement** ***(Short)*****:** Identificador del logro al que se asocia la acción.

Affiliate

En esta tabla se guardan los grupos de afiliados. Se trata de conjuntos de usuarios que se pueden crear y a los que se puede asignar ciertas ventajas, como descuentos o monedas de regalo extra cada día.

- **idAffiliate** ***(Short)*****:** Identificador del grupo de afiliados.
- **description** ***(String)*****:** Descripción del grupo de afiliados.

AffiliateBenefits

Beneficios asignados a los diferentes grupos de afiliados.

- **idAffiliate** ***(Short)*****:** Identificador único del logro.
- **idBenefit** ***(int)*****:** Identificador del videojuego al que pertenece.

Benefit

Tabla con los beneficios que un jugador puede obtener al equiparse con un objeto, alcanzar cierta cantidad de puntos o pertenecer a un determinado grupo de afiliados. Un beneficio puede ser de varios tipos: de acción, gráfico, entero, string o item.

- **idBenefit *(int)*:** Identificador del beneficio.
- **type *(int)*:** Tipo de beneficio. Se definen como 'acción' (0), 'gráfico' (1), 'entero' (2), 'string' (3) o 'item' (4).
- **description *(String)*:** Descripción del beneficio.
- **showOnClient *(Boolean)*:** Indica si el beneficio ha de ser visible por el cliente.
- **intVal *(int)*:** Valor del beneficio de tipo entero.
- **strVal *(String)*:** Valor del beneficio de tipo string.
- **idAction *(Long)*:** Acción asociada al beneficio de tipo acción. Primer campo del identificador.
- **idVGAction *(Short)*:** Acción asociada al beneficio de tipo acción. Segundo campo del identificador.
- **idGraphicResources *(Long)*:** Recurso gráfico asociado al beneficio de tipo gráfico.
- **idItem *(int)*:** Objeto asociado al recurso de tipo item.

CompletedAchievement

Esta tabla incluye los logros que ha completado cada usuario. El campo de identificador del videojuego se utiliza a la vez para identificar unívocamente tanto el logro como el perfil del jugador.

- **idAchievement *(String)*:** Identificador del logro.
- **idVideoGame *(Short)*:** Identificador del videojuego.
- **idUser *(Long)*:** Identificador del usuario.
- **completed *(Date)*:** Fecha en la que el jugador desbloqueó el logro.
- **displayed *(Boolean)*:** Indica si el logro ha sido mostrado.

ComposedItem

En algunos juegos existen objetos que se componen a su vez de otros objetos. Un objeto compuesto tendrá en esta tabla una entrada por cada uno de los objetos que lo componen.

- **idCompound *(int)*:** Identificador del objeto compuesto.
- **idComponent *(int)*:** Identificador del componente.
- **ord *(int)*:** Ordinal del componente.

CurrencyPack

En esta tabla definiremos los diferentes packs de monedas o créditos que los jugadores podrán comprar con su tarjeta de crédito u otros medios de pago en línea como PayPal. Algunos pueden ser ofertas con descuentos o algún objeto de regalo.

- **idCurrencyPack *(Short)*:** Identificador del pack.
- **idVideoGame *(Short)*:** Identificador del videojuego al que pertenece.
- **amount *(int)*:** Cantidad de monedas o créditos.
- **currencyType *(int)*:** Indica si el pack contiene monedas o créditos.
- **discount *(Short)*:** Descuento aplicable a la compra.
- **dollars *(Float)*:** Precio en dólares.
- **euros *(Float)*:** Precio en Euros.
- **fbCredits *(Float)*:** Precio en Facebook Credits.
- **onlyFreeUser *(Boolean)*:** Indica si la oferta está sólo disponible para los jugadores gratuitos, es decir, los que realizan una compra por primera vez.
- **testAB *(Boolean)*:** Sirve para comprobar si el cobro ha sido aprobado por la plataforma de pago utilizada.
- **idItem *(int)*:** Objeto que se regala al comprar el pack.
- **idGraphicResources *(Long)*:** Icono del pack de monedas.

DisputesChargebacksFB

Cuando un usuario reclama la devolución de una compra realizada con Facebook Credits, su solicitud y el estado de la misma se almacena en esta tabla.

- **idDispute *(Long)*:** Identificador de la solicitud.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.
- **credits *(int)*:** Cantidad de Facebook Credits reclamada.
- **finalState *(String)*:** Estado final de la transacción. Puede ser SETTLED, DISPUTED o REFUNDED.
- **initState *(String)*:** Estado inicial de la transacción.
- **timeStamp *(Date)*:** Fecha de la reclamación.

EquipmentArea

En esta tabla se definen las áreas donde se pueden los equipar objetos. Ejemplos de áreas que podemos definir son çabeza.º ”pies”.

- **idEquipmentArea *(String)*:** Identificador del área equipable.

EquipmentType

Al igual que las áreas, también es conveniente definir una serie de tipos de objeto. Estos tipos pueden ser, por ejemplo, ”sombrero.º çalzado”.

- **idEquipmentType *(String)*:** Identificador del tipo de objeto.

EquipmentAreaType

Es necesario definir una nueva tabla para relacionar las dos anteriores entre sí. De esta manera establecemos que el objeto ”sombrero.esté relacionado con el área çabeza”. Un mismo tipo de objeto puede asociarse con dos o más áreas diferentes y viceversa.

- **idArea *(String)*:** Identificador del área equipable.
- **idType *(String)*:** Identificador del tipo de objeto.

FacebookUserAuth

Establece una relación entre el id del jugador dentro de la plataforma de Facebook y su id dentro del servidor de juegos. Habrá una tabla análoga para cada red social que se añada en el futuro.

- **idPlatform *(Long)*:** Identificador del usuario de Facebook.
- **idUser *(Long)*:** Identificador del usuario en nuestra plataforma.

FiredAction

Las acciones de la tabla Actions que un usuario ha realizado en alguna ocasión se almacenan en esta tabla, junto al número de veces que la acción ha sido realizada. Cuando el número de usos de la acción alcance el umbral, la acción será disparada.

- **idAction *(Long)*:** Identificador de la acción.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.
- **uses *(Long)*:** Número de veces que se ha realizado la acción.

Game

Cada vez que se crea una partida nueva, se crea una entrada en esta tabla. Se incluyen todos los datos referidos a la misma: estado de la partida, precio de entrada, nombre y contraseña, etcétera. De esta manera podemos recuperar partidas que un jugador haya dejado a medias. Cuando la partida se termine, la entrada correspondiente será eliminada.

- **idGame *(Long)*:** Identificador de la partida.
- **coins *(int)*:** Cantidad de monedas que cuesta entrar en la partida.

- **coinsAllocated** ***(int)*:** Cantidad de monedas acumuladas que aportaron los jugadores al principio de la partida.
- **comesFrom** ***(Long)*:** Partida de la que deriva la partida actual, si se trata de una revancha o de una ronda de un torneo.
- **gameStatus** ***(int)*:** Estado actual de la partida.
- **maxUsers** ***(int)*:** Número máximo de jugadores.
- **name** ***(String)*:** Nombre de la partida.
- **password** ***(String)*:** Contraseña de la partida.
- **server** ***(String)*:** Servidor en el que se encuentra la partida.
- **idUser** ***(Long)*:** Identificador del jugador que ha creado la partida.
- **idVideoGame** ***(Short)*:** Identificador del videojuego.

GamePlayers

Esta tabla contiene información sobre los jugadores que se encuentran jugando o esperando a que empiece una partida.

- **idGame** ***(Long)*:** Identificador de la partida.
- **coins** ***(int)*:** Cantidad de monedas aportadas por el usuario.
- **idUser** ***(Long)*:** Identificador del usuario.
- **forcedExit** ***(Boolean)*:** Indica si el jugador ha sido forzado a abandonar la partida.
- **playing** ***(Boolean)*:** Indica si el jugador se encuentra jugando en estos momentos.

GraphicResources

Todos los recursos gráficos y/o sonoros, ya sean animaciones, dibujos, banners, recursos Flash o sintonías, se definen aquí.

- **idGraphicResource** ***(Long)*:** Identificador único del logro.
- **animation** ***(Boolean)*:** Indica si se trata de una animación.

- **duration *(Long)*:** Duración de la animación.
- **name *(String)*:** Nombre del recurso.
- **path *(String)*:** Ruta del fichero correspondiente al recurso.
- **triggerTime *(Long)*:** Tiempo de espera antes de lanzar el recurso.

InventoryItem

En esta tabla almacenaremos todos los objetos que un usuario posee en su inventario, ya sea por haberlo ganado jugando o porque lo ha comprado con dinero real o virtual. También se indica en esta tabla si el objeto se encuntra actualmente equipado o no.

- **idInventoryItem *(Long)*:** Identificador del objeto de inventario.
- **buyDate *(Date)*:** Fecha de compra o consecución del objeto.
- **durationDate *(Date)*:** Duración del objeto.
- **expirationDate *(Date)*:** Fecha de caducidad del objeto. Superada esta fecha, no se podrá volver a utilizar.
- **ord *(int)*:** Ordinal del objeto dentro del inventario.
- **remainingGames *(int)*:** Partidas que le quedan al objeto.
- **remainingTime *(Long)*:** Tiempo restante.
- **usedDate *(Date)*:** Fecha en la que se usó por última vez el objeto.
- **idEquipmentArea *(String)*:** Área donde está equipado el objeto.
- **idItem *(int)*:** Identificador de objeto.
- **idUser *(Long)*:** Identificador de usuario del propietario.
- **idVideoGame *(Short)*:** Identificador del videojuego.

Item

En esta tabla se definen todos los objetos que se pueden adquirir en la tienda del juego.

- **idItem *(int)*:** Identificador del objeto.
- **active *(Boolean)*:** Indica si el objeto se encuentra actualmente activo.
- **autoequip *(Boolean)*:** Indica si el objeto se equipará automáticamente al adquirirlo.
- **costCoins *(int)*:** Coste del objeto en monedas.
- **costCredits *(int)*:** Coste del objeto en créditos.
- **discountCoins *(int)*:** Descuento aplicable al precio de adquisición en monedas.
- **discountCredits *(int)*:** Descuento aplicable al precio de adquisición en créditos.
- **duration *(Long)*:** Indica la cantidad de tiempo que un objeto puede utilizarse antes de que caduque.
- **equipable *(Boolean)*:** Indica si el objeto es equipable.
- **expiration *(Long)*:** Indica el tiempo a partir del cual el objeto caducará irrevocablemente, haya sido utilizado o no.
- **finishDate *(Date)*:** Indica la fecha en la que el objeto caducará si sigue en uso.
- **initDate *(Date)*:** Indica la fecha en la que se empezó a usar el objeto por última vez.
- **maxGames *(int)*:** Máximo de partidas en las que puede utilizarse el objeto.
- **requiredFriends *(int)*:** Cantidad de amigos requerida para desbloquear la compra del objeto.
- **requiredLevel *(int)*:** Nivel requerido para desbloquear la compra del objeto.
- **requirementsOnCredits *(Boolean)*:** Indica si las restricciones en el uso de un objeto son aplicables cuando se ha comprado con créditos en lugar de monedas.
- **scope *(int)*:** Permite acotar las condiciones en las que se puede equipar el objeto.
- **stock *(int)*:** Se puede definir un stock limitado para algunos objetos.
- **idEquipmentGraphic *(Long)*:** Recurso gráfico del objeto equipado.
- **idFiredAction *(Long)*:** Acción que dispara el objeto.

- **idVGFiredAction** ***(Short)*****:** Segunda parte del identificador de la acción asociada.
- **idRequiredAction** ***(Long)*****:** Acción requerida para obtener el objeto.
- **idShareGraphic** ***(Long)*****:** Recurso gráfico que aparecerá cuando se comparta el presente objeto.
- **idSpecialGraphic** ***(Long)*****:** Recurso gráfico especial del objeto.
- **idStoreGraphic** ***(Long)*****:** Recurso gráfico del objeto dentro de la tienda.
- **idVideoGame** ***(Short)*****:** Identificador del videojuego.

ItemBenefits

Los objetos pueden aportar al jugador una serie de beneficios, que se detallarán en esta tabla. Un mismo objeto puede tener múltiples beneficios, al igual que el mismo beneficio puede ser compartido por numerosos objetos.

- **idItem** ***(int)*****:** Identificador del objeto.
- **idBenefit** ***(int)*****:** Identificador del correspondiente benefit.

ItemCollectionOrder

En esta tabla se establece la pertenencia de un objeto a una colección determinada. De esta manera posibilitamos que un mismo objeto pueda pertenecer a más de una colección.

- **idItem** ***(int)*****:** Identificador del objeto.
- **idCollection** ***(int)*****:** Identificador de la colección.
- **ord** ***(int)*****:** Ordinal del objeto dentro de la colección.

ItemEquipmentType

Esta tabla establece una relación varios-a-varios entre la tabla Item y la tabla EquipmentType.

- **idItem** ***(int)*****:** Identificador del objeto.
- **idEquipmentType** ***(String)*****:** Identificador del tipo de equipo.

MissionTasks

Algunas acciones, llamadas misiones, se comoponen a su vez de otras acciones, llamadas tareas. Para que una acción de tipo "misión"sea activada necesita que se activen antes todas las acciones de tipo "tarea"que tiene asociadas.

- **idMission *(int)*:** Identificador de la Acción de tipo misión.
- **idVGMission *(Short)*:** Identificador del juego de la Acción de tipo misión.
- **idTask *(int)*:** Identificador de la Acción asociada.
- **idVGTask *(Short)*:** Identificador del juego de la Acción asociada.

OAuthToken

La autenticación con los servidores de Facebook, Google, Twitter y la mayoría de redes sociales se realiza empleando OAuth 2.0. En este estándar, el usuario es autenticado mediante un token que la red social en cuestión proporciona y que tiene un periodo de validez. En esta tabla se guardan los datos del token que un jugador tiene activo en cada momento, sobreescribiendo los datos del que caduca.

- **idToken *(Long)*:** Identificador del token.
- **expires *(Long)*:** Tiempo de validez del token.
- **oAuthToken *(String)*:** Token de autorización.
- **type *(int)*:** Tipo de token.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.

Platform

En esta tabla se almacenan las credenciales de la aplicación para cada red social. Facebook, por ejemplo, asigna a cada aplicación un identificador y una contraseña que deberemos incluir cada vez que realicemos una petición a sus servidores, y el resto de plataformas tendrán un mecanismo similar.

- **idPlatform *(Short*:** Identificador de la plataforma de juego.
- **type *(int)*:** Tipo de plataforma social. Aquí codificaremos la red social a la que pertenece esta instacnia del juego mediante un entero. Por el momento, el único valor válido es el que corresponde a Facebook (0).
- **coinsRate *(float)*:** Identificador único del logro.
- **creditsRate *(float)*:** Factor de conversión aplicable a los créditos de la plataforma social.
- **idFacebookApp *(Long)*:** Identificador de la palicación en Facebook.
- **secretFacebookApp *(String)*:** Contraseña de Facebook para la aplicación.

PointsLevel

Define la cantidad de puntos necesarios para acceder a cada nivel en cada juego.

- **idPointsLevel *(Short*:** Identificador del nivel.
- **level *(Short)*:** Nivel.
- **maxPoints *(int)*:** Cota máxima de puntos.
- **minPoints *(int)*:** Cota mínima de puntos para acceder al nivel.
- **idVideoGame *(Short)*:** Identificador del videojuego.

PointsLevelBenefits

Alcanzar un nuevo nivel supondrá en ocasiones obtener algún tipo de beneficio. Dichos beneficios se definen en esta tabla.

- **idPointsLevel *(Short)*:** Identificador del nivel.
- **idBenefit *(String)*:** Identificador del beneficio.

Properties

Esta tabla permite definir una serie de propiedades generales para la plataforma.

- **name *(String)*:** Nombre de la propiedad.
- **section *(String)*:** Sección.
- **type *(String)*:** Tipo.
- **value *(String)*:** Valor.

Round

Esta tabla nos permite definir rondas para torneos.

- **idRound *(Long)*:** Identificador de ronda.
- **date *(Date)*:** Fecha del torneo.
- **number *(int)*:** Número de ronda.
- **idTournament *(Long)*:** Identificador del torneo al que pertenece.

RoundContenders

Los participantes de una ronda se incluyen en la siguiente tabla.

- **idRound *(Long)*:** Identificador de la ronda.
- **idUser *(Long)*:** Identificador del jugador.
- **idVideoGame *(Short)*:** Identificador del videojuego.

Statistics

Para la construcción de ránkings de jugadores es necesario disponer de una tabla donde almacenar diferentes estadísticas de juego.

- **week *(int)*:** Semana del año codificada como un entero de 4 cifras siguiendo el formato WWYY, siendo WW el número de semana de 1 a 52 y YY las últimas dos cifras del año actual.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.
- **games *(int)*:** Partidas jugadas por el usuario.
- **wonCoins *(int)*:** Monedas ganadas en la semana.
- **wonGames *(int)*:** Partidas ganadas.
- **wonPoints *(int)*:** Puntos ganados.

Store

Cada juego puede tener varias tiendas, y una tienda puede pertenecer a varios juegos diferentes (tiendas de complementos para avatares, por ejemplo).

- **idStore *(int)*:** Identificador de la tienda.
- **idVideoGame *(Short)*:** Identificador del videojuego.

StoreCollection

Las colecciones son conjuntos de objetos que comparten una temática especial. Es común que en los juegos sociales aparezcan diferentes colecciones de objetos con motivos adecuados a la temporada del año y a todo tipo de festividades.

- **idStoreCollection *(int)*:** Identificador de la colección.
- **idVideoGame *(Short)*:** Identificador del videojuego.
- **idGraphicResource *(Long)*:** Recursos gráficos de la tienda.

StoreCollectionOrder

Relaciona una colección con una tienda.

- **idCollection *(int)*:** Identificador de la colección.
- **idStore *(int)*:** Identificador de la tienda.
- **ord *(int)*:** Ordinal de la colección dentro de la tienda.

Tournament

En esta tabla se pueden definir torneos de jugadores.

- **idTournament *(Long)*:** Identificador del torneo.
- **cost *(int)*:** Coste de entrada.
- **jackpot *(int)*:** Bote acumulado.
- **prize *(int)*:** Premio para el ganador.
- **startDate *(Date)*:** Fecha de inicio.

TournamentContenders

Los jugadores del torneo se almacenan en esta tabla.

- **idTournament *(Long)*:** Identificador del torneo.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.

User

Tabla para almacenar los usuarios. Cada usuario que se conecte a la plataforma dispondrá de una entrada en esta tabla. Si un mismo usuario accede a dos juegos diferentes dentro de la plataforma, seguirá contando como un solo usuario.

- **idUser** ***(Long)*****:** Identificador del usuario.
- **birthday** ***(Date)*****:** Fecha de nacimiento del usuario. Puede usarse para aplicar filtros de edad a determinados contenidos o para regalar algún objeto a un usuario que cumple años.
- **email** ***(String)*****:** Correo del usuario.
- **gender** ***(int)*****:** Género del usuario.
- **locale** ***(String)*****:** Identificador de zona del usuario. Se usa para asignarle un idioma u otro.
- **password** ***(String)*****:** Contraseña de acceso. No será necesaria en los juegos que se accedan desde el entorno de una red social o utilizando sus credenciales.
- **realName** ***(String)*****:** Nombre real del jugador.
- **rol** ***(String)*****:** Nivel de permisos.

UserFriend

Cada entrada de la tabla relaciona a dos jugadores como amigos.

- **idUser** ***(Long)*****:** Identificador de usuario.
- **idFriend** ***(Long)*****:** Identificador del amigo.

UserVideoGameProfile

Cada usuario que juegue a un videojuego dispondrá de un perfil de jugador dentro del mismo. En este se almacenarán todos los datos del jugador relacionados con dicho juego. Un usuario que juegue a dos juegos diferentes tendrá dos perfiles, uno para cada juego.

- **idUser** ***(Long)*****:** Identificador del usuario.
- **idVideoGame** ***(Short)*****:** Identificador del videojuego.
- **avatar** ***(String)*****:** Ruta a la foto de avatar del jugador.
- **coins** ***(int)*****:** Cantidad de monedas.
- **concurrentGames** ***(int)*****:** Partidas que el jugador tiene abiertas actualmente.

- **credits *(int)*:** Cantidad de créditos de que el jugador dispone.
- **destroyCoins *(int)*:** Indica la cantidad de monedas que se le restarán al jugador la próxima vez que se loguee con motivo de algún tipo de penalización.
- **extraCoins *(int)*:** Indica un extra de monedas de regalo que el jugador recibirá en su próximo logueo.
- **extraCredits *(int)*:** Indica un extra de créditos de regalo que el jugador recibirá en su próximo logueo.
- **games *(int)*:** Partidas jugadas.
- **karma *(int)*:** Puntos de karma. Suben o bajan dependiendo del comportamiento deportivo del jugador. Por ejemplo, salir de una partida sin terminar penalizará, mientras que enviar regalos a amigos periódicamente lo aumentará.
- **lastLogin *(Date)*:** Fecha del último login del usuario.
- **nickname *(String)*:** Áleas del usuario dentro del juego.
- **online *(Boolean)*:** Indica si el usuario se encuentra conectado.
- **payments *(int)*:** Compras realizadas por el jugador.
- **points *(int)*:** Puntos conseguidos.
- **registrationDate *(Date)*:** Fecha en la que el usuario se registró por primera vez.
- **registrationIP *(String)*:** IP desde la que el usuario se registró por primera vez.
- **wonCoins *(int)*:** Monedas conseguidas.
- **wonGames *(int)*:** Partidas ganadas.
- **idAffiliate *(Short)*:** Identificador del grupo de afiliados del jugador.

VideoGame

Tabla con los videojuegos que están corriendo sobre el servidor.

- **idVideoGame *(Short)*:** Identificador único del videojuego.
- **cdnURL *(String)*:** URL de la página de la red social que albergará el juego.

- **createRooms *(Boolean)*:** Indica si se permite a los usuarios crear nuevas salas para las partidas (true) o si las salas estarán ya creadas.
- **dailyCoins *(int)*:** Cantidad de monedas que se regalan a los jugadores diariamente por entrar al juego.
- **initialCoins *(int)*:** Monedas que recibe un jugador al registrarse.
- **initialCredits *(int)*:** Créditos que recibe un jugador al registrarse.
- **lang *(String)*:** Idioma original del juego.
- **maxConcurrentGames *(int)*:** Máximo número de partidas simultáneas permitidas por jugador. Un valor de 0 indica que no hay límite.
- **maxGameUsers *(int)*:** Máximo número de jugadores por partida.
- **minGameCoins *(int)*:** Mínimo de monedas para crear o entrar a una partida.
- **minGameUsers *(int)*:** Mínimo número de jugadores para empezar una partida.
- **name *(String)*:** Nombre del videojuego.
- **pwdServer *(String)*:** Contraseña del servidor, si fuera necesaria.
- **serverURL *(String)*:** URL del servidor que alberga el videojuego.
- **idGraphicResources *(Long)*:** Recurso gráfico con el logo del juego.
- **idPlatform *(Short)*:** Identificador de plataforma.

VideoGameEquipmentArea

Relaciona las áreas equipables con los videojuegos. Habrá juegos en los que no se podrá equipar objetos en una determinada área.

- **idVideoGame *(Short)*:** Identificador del videojuego.
- **idArea *(String)*:** Identificador del área.

3.3.1. Tablas de traducciones

XXXX_translation

Son las tablas que almacenan las traducciones de las tablas que lo necesitan. Las tablas que disponen de traducción son Achievement, CurrencyPack, Item, Store y StoreCollection y los campos son los mismos en todos los casos

- **idXXXX *(Variable)*:** Identificador de la entrada de la tabla original. Cabe tener en cuenta que cuando se trate de traducciones de tablas con doble campo clave (Achievement_tanslation, por ejemplo), tendremos dos campos de id.
- **description *(String)*:** Texto de la traducción de la descripción del item.
- **shortDescription *(String)*:** Texto de la traducción del título o la descripción corta del item.
- **lang *(String)*:** Código internacional del idioma.

3.3.2. Tablas de registro

Popularmente conocidas como 'Logs', registran y almacenan toda la actividad de los usuarios que se produce en la plataforma.

ActionLog

Registro de las acciones de los usuarios.

- **idLog *(Long)*:** Identificador de la entrada de log.
- **date *(Date)*:** Fecha y hora en la que se produjo el evento.
- **message *(String)*:** Mensaje para loguear, que corresponderá a la descripción de la acción registrada.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.

ChatLog

Registro de los mensajes de chat.

- **idLog *(Long)*:** Identificador de la entrada de log.
- **date *(Date)*:** Fecha y hora en la que se produjo el evento.
- **message *(String)*:** Mensaje para loguear.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.

ConnectionLog

Registro de las conexiones.

- **idLog *(Long)*:** Identificador de la entrada de log.
- **date *(Date)*:** Fecha y hora en la que se produjo el evento.
- **message *(String)*:** Mensaje para loguear.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.
- **ip *(String)*:** Dirección IP desde la que se produjo el registro.
- **concurrent *(Int)*:** Número de conexiones consecutivas realizadas.

CreditLog

Registro de la compra de monedas que se efectúen.

- **idLog *(Long)*:** Identificador de la entrada de log.
- **date *(Date)*:** Fecha y hora en la que se produjo el evento.
- **message *(String)*:** Mensaje para loguear.

- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.
- **coins *(Int)*:** Cantidad de monedas.
- **type *(Int)*:** Tipo de moneda.
- **FBCredits *(Int)*:** Cantidad de Facebook Credits.
- **euros *(Float)*:** Cantidad en euros.
- **dolars *(Float)*:** Cantidad en dólares.
- **idOrder *(Int)*:** Número de conexiones consecutivas realizadas.

DailyLog

Registro de los usuarios y pagos realizados durante un día.

- **idVideoGame *(Short)*:** Identificador del videojuego.
- **day *(Int)*:** Identificador del día.
- **dau *(Int)*:** Número de usuarios activos durante el día.
- **payerUsers *(Int)*:** Número de usuarios activos que han realizado alguna compra.
- **credits *(Int)*:** Créditos recaudados durante el día.

EconomicLog

Registro de momentos en los que un jugador recibe monedas virtuales.

- **idLog *(Long)*:** Identificador de la entrada de log.
- **date *(Date)*:** Fecha y hora en la que se produjo el evento.
- **message *(String)*:** Mensaje para loguear.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.
- **coins *(Int)*:** Cantidad de monedas.
- **currencyType *(Int)*:** Tipo de moneda.

GameLog

Registro de eventos de juego.

- **idLog *(Long)*:** Identificador de la entrada de log.
- **date *(Date)*:** Fecha y hora en la que se produjo el evento.
- **message *(String)*:** Mensaje para loguear.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.

ItemLog

Registro de compras de objetos en la tienda.

- **idLog *(Long)*:** Identificador de la entrada de log.
- **date *(Date)*:** Fecha y hora en la que se produjo el evento.
- **message *(String)*:** Mensaje para loguear.
- **idUser *(Long)*:** Identificador del usuario.
- **idVideoGame *(Short)*:** Identificador del videojuego.
- **idItem *(Int)*:** Identificador del objeto.
- **coins *(Int)*:** Precio en monedas.
- **credits *(Int)*:** Precio en créditos.

Capítulo 4

Resultados

El producto que se ha obtenido al finalizar este proyecto es una plataforma servidor que permite alojar y ejecutar simultáneamente la lógica de varios videojuegos y guardar sus estados en una base de datos común. La plataforma ejecutará una o más zonas sobre la instancia de SmartFox, una por cada juego, y cada una de estas zonas contará con una o más salas, que pueden ser de dos tipos diferentes:

- **Lobby:** es la sala principal del juego, y cada zona contará como máximo con un Lobby. El Lobby (vestíbulo en inglés) es la única sala que estará activa en todo momento: arranca cuando arranca la zona a la que pertenece y se destruye cuando la zona es destruida. Todos los jugadores que no se encuentren en ninguna otra sala, bien porque acaban de loguearse o porque han terminado una partida, deben estar registrados en el Lobby. Desde esta sala es posible chatear con los otros usuarios, crear partidas, consultar las clasificacones, comprar objetos de la tienda, gestionar el inventario del jugador y, en general, todas las funciones externas al juego que haya disponibles o vayamos añadiendo en el futuro.
- **Salas de juego (Game Rooms):** son las salas donde se encontrarán registrados los jugadores que estén jugando una partida y, en ocasiones, los usuarios que se encuentren observándola. Las salas de juego se crean y destruyen de forma dinámica desde la lógica de la zona: el servidor creará una sala por cada nueva partida que se cree, y la destruirá cuando esta termine. Las salas de este tipo tendrán asociada una extensión de sala encargada de ejecutar la lógica del juego. Esta extensión se encargará de ejecutar funciones típicas del juego, como por ejemplo realizar una tirada de dados o calcular el resultado de una jugada determinada, mantener el estado de la partida e informar a los jugadores cuando este cambie o se produzca algún otro evento como que un jugador haya ganado la partida o se haya desconectado.

La lógica de nuestra aplicación se conecta al servidor a través de una extensión asociada a la zona. Siguiendo la filosofía de SmartFox, todas las extensiones que corren sobre la plataforma

heredan de una clase de extensión básica llamada GloriaExtension. Esta clase, a su vez, hereda de la extensión básica de SFS2X, SFSExtension, sobreescribiendo algunos de sus métodos y añadiendo otros nuevos. Algunos de los objetos que GloriaExtension añade a SFSExtension son una instancia de ManagerDelegate, a través de la cual efectuará todas las operaciones de persistencia en la capa de datos, y un BotManager, que se encargará de poblar el servidor de bots[1] cuando el número de usuarios humanos esté por debajo de un umbral. De la misma manera, el resto de componentes de la aplicación (EventHandlers, RequestHandlers, excepciones,...) también deberán heredar de las clases básicas de nuestra plataforma, que, a su vez, heredarán de la clase correspondiente de SmartFox.

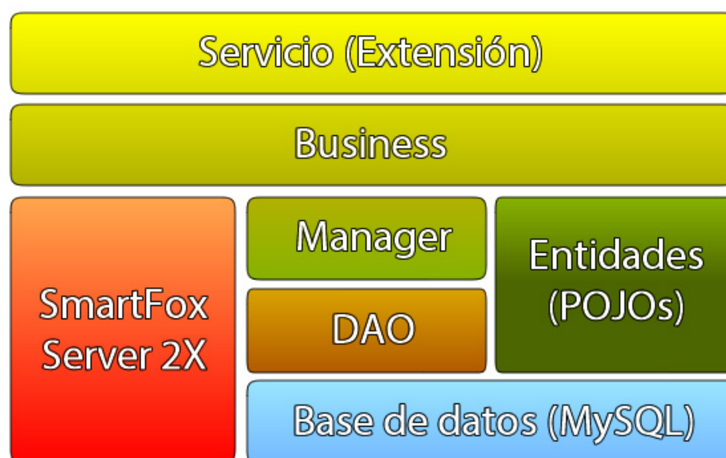


Figura 4.1: Pila de la plataforma de videojuegos.

La extensión de zona es la encargada de registrar y loguear a los usuarios que se conectan a la aplicación, enviarlos al Lobby y atender todas las peticiones que le llegan desde allí, por lo que habitualmente nos referiremos a ella como extensión de Lobby[2]. Siempre a instancias de las acciones y peticiones de los usuarios, la extensión de Lobby creará y destruirá partidas, tramitará compras de objetos, recuperará ránkings y gestionará los logros completados. Al tratarse en todos los casos de funciones de ámbito general que no dependen de la mecánica de ningún juego en concreto, podremos usar una misma extensión de Lobby para todos los juegos realizando sólo algunos pequeños ajustes en su fichero de configuración, sin necesidad de tocar el código ni volver a compilar.

Cuando la extensión de Lobby cree una partida, a nivel de servidor creará una nueva sala y le asociará la extensión de juego correspondiente, que definiremos, por lo general, en

[1]Usuarios ficticios, controlados por el ordenador, que contarán con una inteligencia artificial básica que les permite crear y jugar partidas.

[2]El Lobby, como tal, no dispondrá de una extensión de sala propia asociada, sino que más bien actúa como un contenedor de usuarios para que la extensión de zona pueda gestionar sus peticiones.

el fichero de configuración de la zona. Las extensiones de juego dispondrán de su propia clase básica, GameExtension, que heredará a su vez de la genérica, GloriaExtension, y son de caracter más específico que la extensión de Lobby. Al ser la responsable de la lógica de las partidas, una extensión de juego será única y diferente para cada videojuego que se despliegue en la plataforma, por lo que una nueva extensión tendrá que desarrollarse desde cero para cada caso.

4.1. La Extensión de Lobby

Como ya hemos comentado, la extensión de Lobby es una aplicación genérica, encargada de funciones comunes a cualquier tipo de juego. La primera responsabilidad de esta extensión es la de registrar y loguear nuevos usuarios. Cuando un usuario arranca la aplicación cliente, por ejemplo, desde Facebook, el servidor recibe un mensaje con su nombre de usuario, una contraseña[3] y una serie de metadatos entre los que se encontrará o bien el token OAuth o bien un código que podremos canjear por un token OAuth mediante el Graph API más adelante. El servidor comprobará que las credenciales son correctas, completará la conexión del usuario al servidor y lo enviará directo al Lobby.

En el caso del logueo desde Facebook, el registro de un nuevo usuario se realiza de manera completamente al jugador. Como la autenticación del usuario ya la habrá realizado Facebook previamente, pedimos sus datos a la Graph API y creamos un nuevo usuario en la plataforma. Si el usuario se intentara loguear, por ejemplo, desde una aplicación de Android y desease utilizar credenciales propias en lugar de las de Facebook, debería completar un formulario de registro.

Una vez en el lobby, un usuario puede empezar a realizar acciones que desembocarán en el envío de una serie de peticiones desde la aplicación cliente hasta el servidor. Las peticiones se envían mediante una conexión TCP y se componen de un nombre, que identifica el servicio solicitado, y unos datos adjuntos. Los nombres de las peticiones constan de dos campos separados mediante un punto, el primero identificando el tipo o familia de servicio y el segundo el servicio concreto que se solicita. En nuestro caso hemos clasificado los servicios en cuatro familias, y hemos asignado un nombre a la petición asociada a cada uno de ellos:

- Peticiones relacionadas con el lobby.
  - lobby.currencypacks
  - lobby.savenewuser
  - lobby.rankingusers

[3] Cuando el login se realiza desde Facebook, el usuario ya viene autenticado por la red social. En este caso, la contraseña es un código que se genera nuevamente con cada petición y básicamente sirve como firma digital para detectar posibles ataques man-in-the-middle.

  - lobby.rankingusersbyweek
  - lobby.timeserver
  - lobby.action
  - lobby.ping
  - lobby.listfriends
  - lobby.completedAchievements

- Peticiones relacionadas con las salas.
  - room.new
  - room.join
  - room.lobby
  - room.repeat
  - room.play
  - room.guest
  - room.invite
  - room.inviteResponse
  - room.inviteToken

- Peticiones relacionadas con la tienda.
  - store.findStoreItems
  - store.inventory
  - store.buy
  - store.use
  - store.unUse
  - store.equipped

- Peticiones relacionadas con acciones sociales.
  - social.fired
  - social.unFired

La gestión de las peticiones se realiza mediante unas clases especiales llamadas Request Handlers. Para utilizar estas funciones es encesario registrar un listener en la clase de la extensión que asocie un Handler con una familia de servicios determinada. Cuando reciba una petición, la extensión discriminará, a partir del identificador de familia de servicios, cuál es el handler

Listing 4.1: Registrando listeners para peticiones y eventos.

```
public class LobbyExtension extends GloriaExtension implements ILobbyExtension {
        ...
        @Override
        public void initExtension() {

        ...
                //LISTENERS
                //---Requests
                addRequestHandler("lobby",LobbyRequestHandler.class);
                addRequestHandler("room",RoomRequestHandler.class);
                addRequestHandler("social",SocialRequestHandler.class);
                addRequestHandler("store",StoreRequestHandler.class);

                //---Events
                addEventHandler(SFSEventType.USER_LOGIN, LoginEventHandler.class);
                addEventHandler(SFSEventType.USER_JOIN_ZONE, JoinZoneEventHandler.class);
                addEventHandler(SFSEventType.USER_DISCONNECT, LogoutEventHandler.class);
                addEventHandler(SFSEventType.USER_LOGOUT, LogoutEventHandler.class);
                addEventHandler(SFSEventType.SERVER_READY,ServerReadyEventHandler.class);
        ...
        }
        ...
}
```

asociado que ha de gestionar el mensaje. De la misma manera, en una extensión se pueden registrar también Event Handlers, que funcionarán de un modo similar pero estarán asociados a eventos en lugar de peticiones.

Cuando un handler es invocado por la extensión recibe como parámetros el usuario que ha enviado la petición y un objeto SFSObject que incluirá información asociada a la petición o el evento que gestionan. Cuando un Request Handler se asocia a una familia que incluye más de un servicio, ha de implementarse como un *MultiHandler*. Los *MultiHandler* reciben como parámetro, dentro de los datos asociados a la petición, un campo MULTIHANDLER_REQUEST_ID con el segundo miembro del nombre de la petición (identificador de servicio), lo que nos permite determinar cuál será el método encargado de proporcionar el servicio. Se recomienda que las clases que implementen los Handlers de las extensiones de nuestra plataforma hereden de las clases GloriaRequestHandler y GloriaEventHandler, según corresponda.

El acceso a la base de datos desde la aplicación de servidor se realiza exclusivamente a través de la instancia de ManagerDelegate de la extensión, accediendo a los métodos de los managers que esta contiene. En ningún caso debería instanciarse directamente un objeto

Listing 4.2: Discriminación de peticiones en una clase MultiHandler.

```
1  @MultiHandler
2  public class LobbyRequestHandler extends GloriaRequestHandler{
3      ...
4      @Override
5      public void handleRequest(User user, ISFSObject params) {
6          ...
7          String requestId=
8              params.getUtfString(SFSExtension.MULTIHANDLER_REQUEST_ID);
9
10         if(requestId.equals("currencypacks")){
11             sendCurrencyPacks(user, params);
12         }else if(requestId.equals("coinspacks")){
13             sendCoinsPacks(user, params);
14         }else if(requestId.equals("savenewuser")){
15             saveNewUser(user, params);
16         }else if(requestId.equals("rankingusers")){
17             sendRankingUsers(user, params);
18         }else if ...
19     }
20     ...
21 }
```

manager, y mucho menos acceder a la capa de objetos DAO. Sólo accediendo la capa de persistencia a través del ManagerDelegate nos aseguraremos de que las operaciones de persistencia se realicen correctamente, respetando las transacciones, además de mantener las clases de nuestra aplicación desacopladas de las capas subyacentes.

Otro componente importante de la extensión de Lobby es el fichero de configuración. Por defecto, este fichero se llamará config.parameters y estará situado en la raíz de la carpeta donde se encuentren las clases de la aplicación. A través del fichero de configuración podremos ajustar una serie de valores que afectarán al funcionamiento de la extensión sin necesidad de volver a compilarla. La función más importante de este fichero es especificar los datos que la extensión de Lobby necesitará para acceder a la información del juego en la base de datos y asociar la clase de extensión adecuada a las salas de juego cuando cree nuevas partidas.

La extensión de Lobby es lo suficientemente genérica como para poder ser utilizada por todos los juegos que se desplieguen sobre el servidor, modificando tan solo el fichero de configuración. Aun así, no podemos decir que se trate de un producto cerrado, ya que a medida que se vayan desplegando nuevos juegos es posible que nos encontremos con la necesidad de añadir nuevas funciones para completar su funcionamiento.

Listing 4.3: Ejemplo de fichero de configuración.

```
1
2 lobby.game.id = 1
3 lobby.game.name = chess
4 lobby.game.class = com.ibermedia.games.chess.ChessExtension
5
6 lobby.store = true
7 lobby.gloria.friendsAllowed = true
8 lobby.autopublish = false
9
10 config.facebook.post.initial.name= Ajedrez
```

4.2. Extensiones de juego

La lógica de un juego que se vaya a desplegar en la plataforma no reside en la extensión de juego, sino que residirá en una clase a la que llamaremos Game. Esta clase se encargará de realizar las jugadas y decidir cuándo terminará la partida y quien será el ganador. Los jugadores, por su parte, estarán representados dentro de la clase Game mediante instancias de una segunda clase a la que llamaremos Player. Cada partida que se ejecute en el servidor será una instancia de esta clase Game, y cada uno de los usuarios controlará una de las instancias de Player. La misión principal de la extensión de juego en este asunto consistirá en actuar como plataforma entre los usuarios de SmartFox y la clase Game.

Al igual que la extensión de Lobby, la extensión de juego tendrá registrados una serie de Handlers para responder a eventos y peticiones. Cuando la partida esté en marcha, los jugadores enviarán sus jugadas a la extensión a través de una serie de peticiones y será la extensión la que se encargue de realizar la jugada correspondiente sobre el objeto Game y responder con el resultado, que puede ser, por ejemplo, una determinada puntuación o un cambio de turno. Una vez terminada la partida, la extensión informará a todos los usuarios en la sala, repartirá el premio entre los ganadores, asignará puntos de experiencia a cada jugador y persistirá los cambios en la base de datos.

Para construir una extensión de juego tendremos que heredar de la clase GameExtension, que añade a GloriaExtension todos los elementos necesarios para mantener un objeto Game y asociar a los jugadores con sus respectivas instancias de Player. Las clases Game y Player no disponen de clases básicas de las que heredar, pero sí de sendas interfaces: IGame (o ITurnGame, para juegos por turnos) e IPlayer. Al igual que ocurría con la extensión de Lobby, las extensiones de juego disponen de un fichero config.properties que podremos utilizar según nos convenga.

4.2.1. Ejemplo de juego: El conecta cuatro

El conecta cuatro es uno de los juegos más conocidos y sencillos del mundo. La versión clásica consiste en una rejilla con 6 filas y 7 columnas en la que los dos jugadores, por turnos, van dejando caer fichas de dos colores diferentes. El primer jugador que consiga alinear cuatro o más fichas de su color será declarado vencedor.

Antes de empezar la partida, la extensión asignará un color de ficha a cada jugador y decidirá cuál de los dos empezará a jugar. A partir de este momento el servidor necesitará recibir en cada turno el número de columna donde el jugador correspondiente quiere dejar caer su ficha y calcular si con la jugada recibida se ha cumplido la condición de victoria. Si así fuera, informará del ganador mediante un mensaje a todos los usuarios en la sala y repartrá puntos y monedas entre los jugadores de acuerdo al resultado. En caso contrario, se limitará a informar a cuál de los jugadores corresponde el siguiente turno.

Por tanto, implementar este juego sólo supondría incluir un caso de uso más: el de realizar una jugada. Implementaremos este caso de uso mediante una servicio asociado a la llamada *game.drop*, que sólo deberá llevar adjunto el número de columna seleccionada. En la extensión de juego, asociaremos esta petición a un Handler que llamaremos, por ejemplo, GameRequestHandler que será el encargado de darle servicio. Cuando el servidor reciba esta petición, el Handler comprobará que el jugador al que corresponde el turno es el mismo que realizó la petición y accederá a la clase Game para situar la ficha en la fila correspondiente de la columna indicada. Luego realizará la comprobación de la condición de victoria e informará a los jugadores del resultado.

Con el fin de mantener sincronizados los displays en los clientes de cada uno de los usuarios presentes en la sala, el servidor reenviará el mensaje game.drop recibido a todos los usuarios de la sala excepto al que realizó la jugada.

4.3. Cómo desplegar un juego

Una vez programadas las extensiones, desplegar el nuevo juego en el servidor es un proceso muy sencillo. En primer lugar es recomendable introducir los datos del juego en la base de datos: cada juego ha de tener una nueva entrada en la tabla VideoGame para asignarle un número de identificador y en la tabla Platform para introducir la información relativa a la aplicación cliente o plataforma desde la que se accede. El resto de información del juego, la relativa a los ítems de la tienda, logros, misiones, etcétera puede introducirse más adelante. A continuación es necesario crear sendas carpetas en el directorio extensions del servidor de SmartFox para albergar los ficheros con las extensiones de Lobby y del juego, así como sus respectivos ficheros de configuración. El fichero de configuración de la extensión de juego es opcional, pero el de la extensión de Lobby es obligatorio porque debe incluir el identificador del

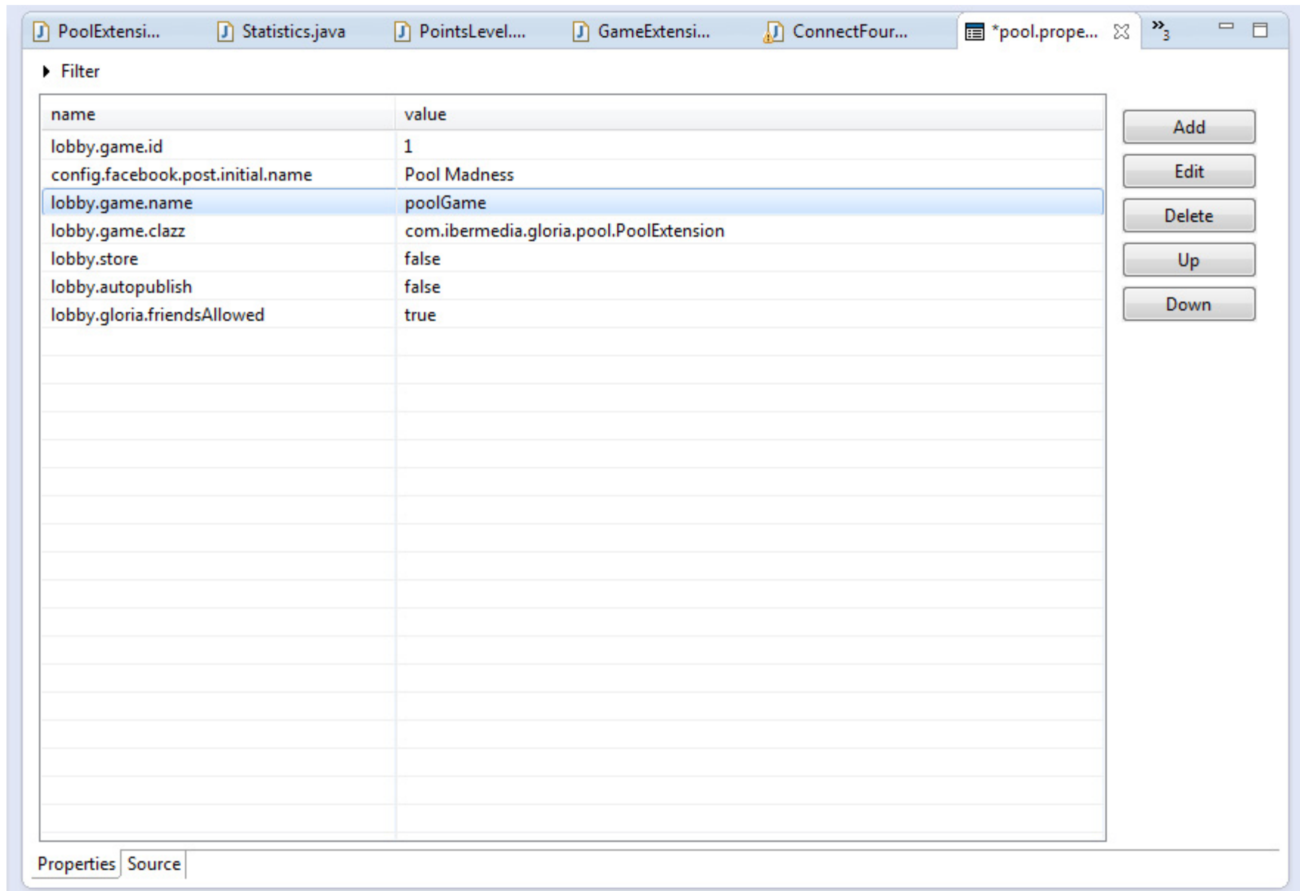


Figura 4.2: Contenido del fichero de configuración de la extensión de Lobby

juego en la base de datos, así como el nombre de la carpeta que alberga la extensión de juego y el nombre completo de la clase del mismo.

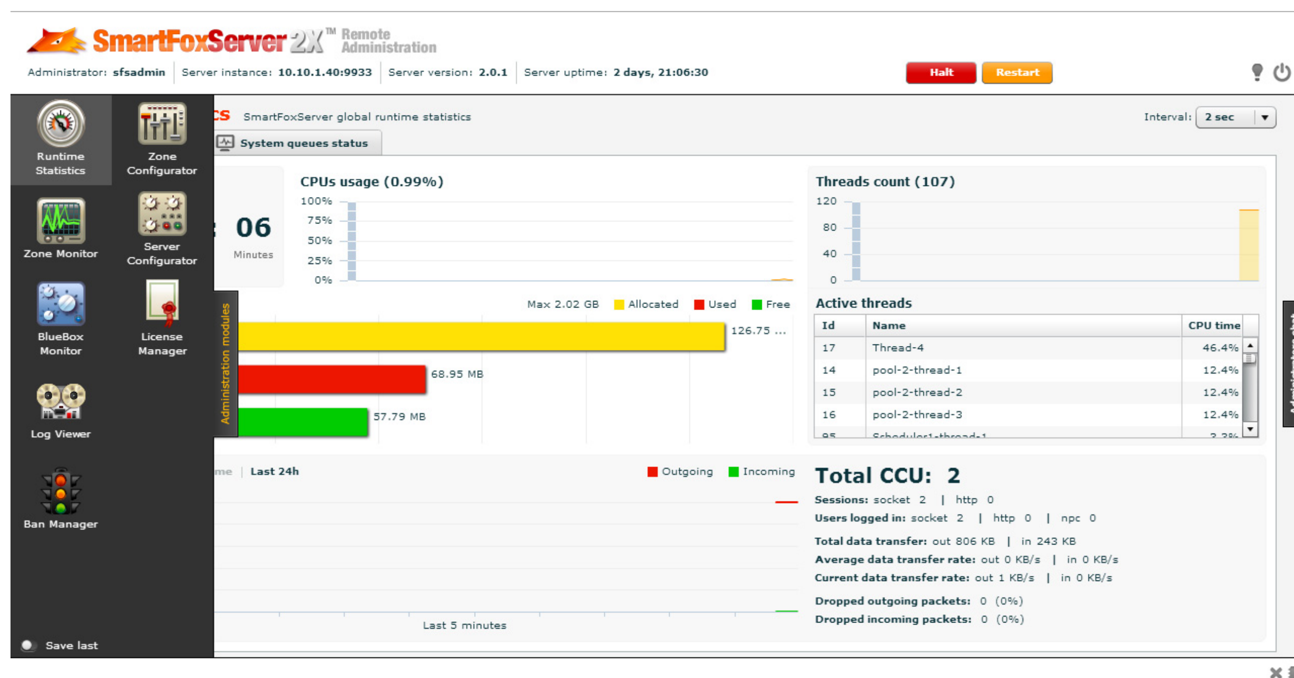


Figura 4.3: Menú de la herramienta de administración de SmartFox

Una vez hechos los preparativos, pasaremos a configurar la zona del servidor de SmartFox. En primer lugar, entraremos a la herramienta de administración a través de nuestro explorador de internet y, tras introducir nuestras credenciales de administrador, iremos al apartado de gestión de zonas (Zone Configurator).

La herramienta Zone Configurator dispone de un panel, situado a la izquierda, compuesto por dos columnas: la primera es un listado de las zonas disponibles, y en la segunda, una vez seleccionada una zona, aparecerán las salas permanentes de las que la zona dispone. Bajo las dos columnas encontramos un grupo de botones que nos permiten crear nuevas zonas o salas y eliminar o cambiar la configuración de la zona o sala que se encuentre seleccionada.

Al hacer click en el botón de crear zona, se abrirá inmediatamente el formulario con todas las opciones de configuración de la nueva zona, organizadas en varias pestañas. La herramienta permite realizar una configuración bastante detallada de la zona, permitiendo limitar el número de usuarios, el tiempo de inactividad permitido y otras muchas opciones. Todos estos parámetros dependerán del juego y la forma en la que queramos que los usuarios puedan conectarse al mismo, pero para cumplir el propósito de esta sección nos ceñiremos sólo a los ajustes más necesarios: asignar un nombre a la zona, para que los clientes puedan conectarse a la misma, y conectar la extensión de Lobby a la misma.

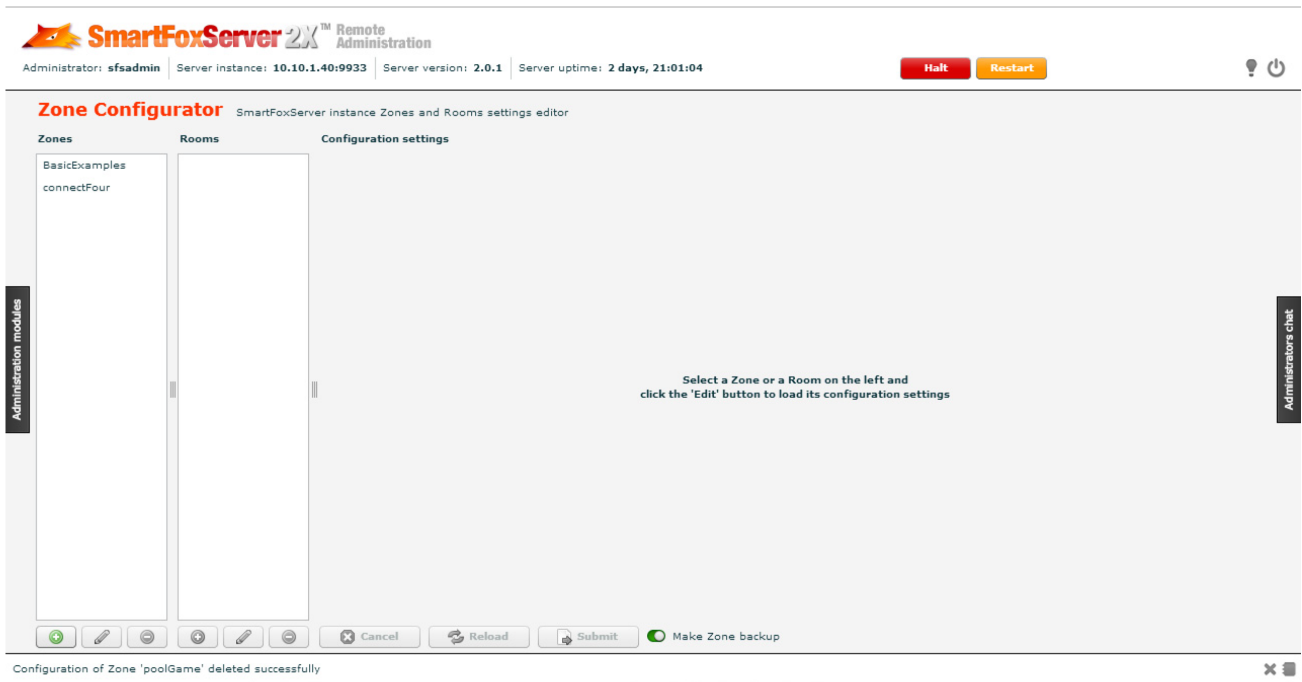


Figura 4.4: Panel de la herramienta Zone Configurator

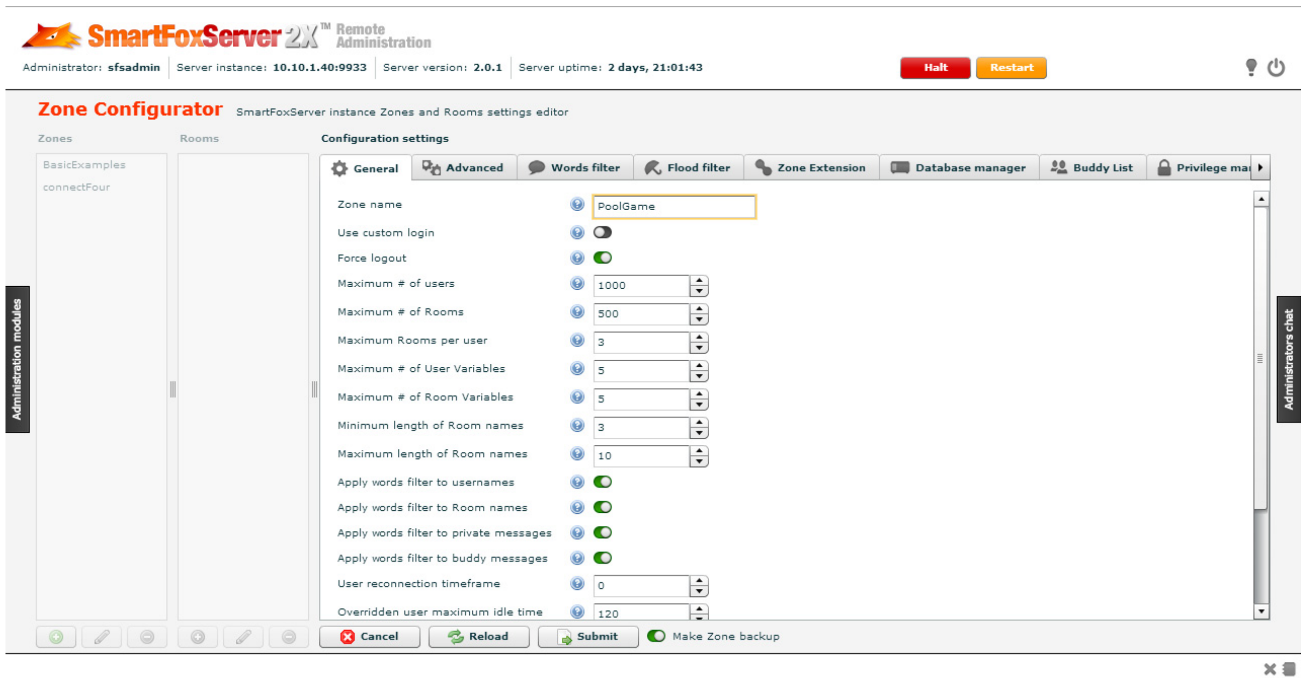


Figura 4.5: Formulario para los ajustes generales de la zona.

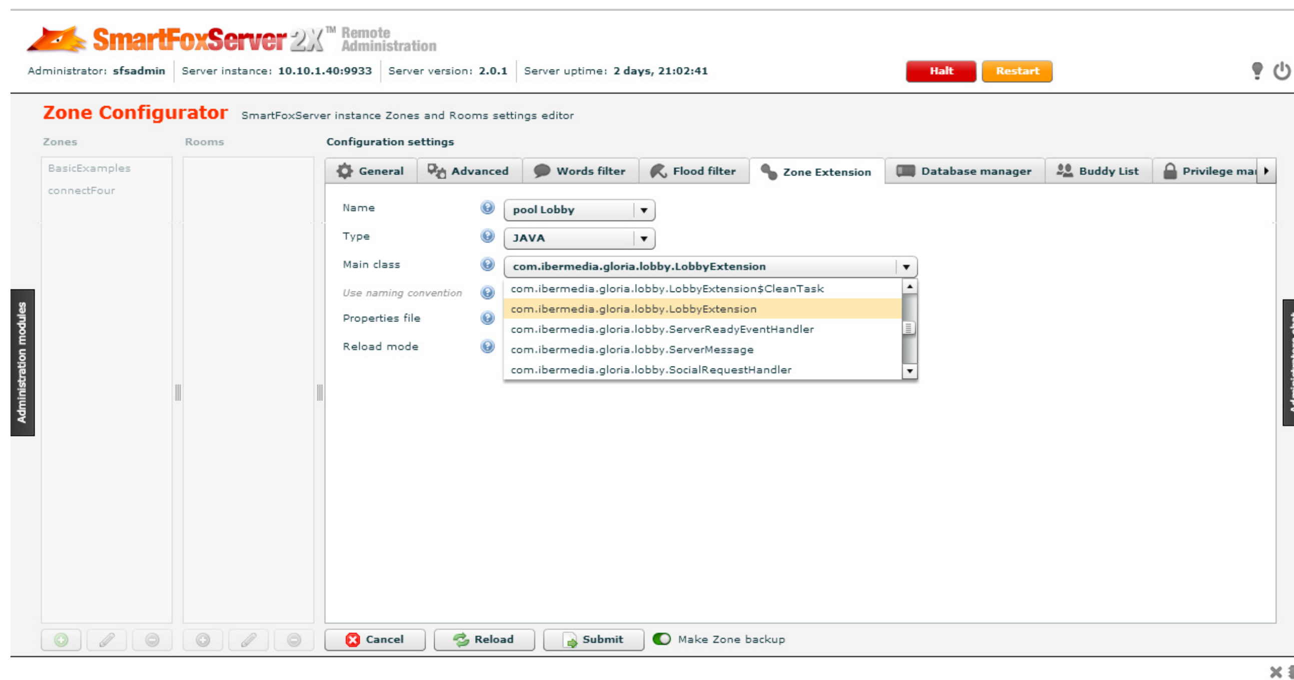


Figura 4.6: Asiganción de la extensión de Lobby a la zona.

La herramienta de configuración es capaz de detectar por sí misma cuál de las clases corresponde a la extensión, siempre y cuando hayamos seguid la recomendación de añadir el sufijo -Extension al nombre de la clase. La detección automática se activa con la opción *Follow Naming Conventions*, y en caso de disponer de más de una sala que cumpla dicha recomendación podremos seleccionar la correcta del menú desplegable. En la misma página en la que añadimos la extensió a la sala también podemos especificar el nombre de nuestro fichero de configuración. Si no se especifica ninguno, tomará el nombre por defecto, *config.properties*.

Una vez configurada la zona, es necesario crear una sala permanente llamada *"The Lobby"*. El procedimiento para crear y configurar la sala es igual que el que hemos seguido para la zona: basta con seleccionar la zona en la que queremos crear la nueva sala y pulsar el botón de añadir nueva sala. Nuevamente se abrirá el formulario para ajustar los parámetros de la sala recién creada, pero esta vez no asignaremos ninguna extensión ya que, como hemos comentado, el Lobby estará gobernado por la extensión de zona.

Cuando ya tengamos la zona configurada con su correspondiente sala *The Lobby*, podremos arrancar la sala pulsando el botón de *Play* que aparece junto al nombre de la nueva zona en la primera de las columnas. Al arrancar la zona, arrancará la extensión asociada a la misma y ya podrá recibir peticiones desde los clientes, dar de alta a los nuevos usuarios y gestionar todos los aspectos del juego de los que hemos hablado a lo largo de este capítulo.

Figura 4.7: Configuración de la sala *The Lobby*

Capítulo 5

Discusión, conclusiones y líneas futuras

El capítulo final de esta memoria se divide en tres partes. En la primera de ellas se realiza un análisis crítico de los resultados del producto final a partir del proceso de testeo y de la experiencia a la hora de desarrollar nuevas extensiones de juego. Seguidamente, en la sección Conclusiones, se repasarán los objetivos que se marcaron inicialmente para el desarrollo de la plataforma, analizando el grado de cumplimiento alcanzado en cada un a de ellas. El último apartado de este capítulo está dedicado a las líneas de trabajo a seguir en el futuro inmediato.

5.1. Discusión de los resultados

La plataforma ha sido testeada con dos zonas funcionando simultáneamente, montando cada una una extensión de juego diferente. Todos los casos de uso de la extensión de Lobby, así como las de las extensiones de juego, han sido probados con éxito realizando llamadas desde la aplicación cliente. El servidor es capaz de gestionar a la vez las llamadas dirigidas a cada una de las zonas y persistir correctamente todos los cambios en la base de datos a través de la clase ManagerDelegate.

El proceso de desarrollo y despliegue de nuevas extensiones de juego ha resultado ser ágil y sencillo. Aunque la complejidad de las aplicaciones desarrolladas dependerá siempre de lo intrincada que pueda llegar a ser la mecánica del juego, las clases básicas para extensiones y handlers y las interfaces IGame e IPlayer proporcionan un marco de desarrollo muy similar al que encontraríamos al desarrollar una aplicación corriente para SmartFoxServer, con la ventaja de disponer de una capa de persistencia y de todos los servicios que la plataforma multijugador ofrece. Estas herramientas de desarrollo cuentan con una detallada documentación para ayudar a los futuros desarrolladores que las utilicen en la comprensión de su funcionamiento.

5.2. Conclusiones

5.2.1. Revisión de objetivos

Como ya hemos comentado en el apartado anterior, los dos objetivos principales se han cumplido satisfactoriamente.

- **Desarrollar un servidor para dar soporte online a juegos sociales multijugador.**
- **Desarrollar un conjunto de clases que, a modo de API, faciliten la creación de extensiones para la plataforma.**

Los dos primeros juegos para la plataforma ya están en camino, estando pendiente de finalizar el desarrollo de las aplicaciones cliente. Ambas extensiones han sido desarrolladas utilizando las clases proporcionadas a este efecto y ya se encuentran funcionando sobre la plataforma.

Cumplir estos objetivos no hubiese sido posible sin haber completado antes una serie de objetivos secundarios:

- **Estudiar y familiarizarse con la tecnología de SmartFoxServer2x (o SFS2X) y sus librerías para Java.**
- **Aprender a utilizar el Framework de Spring para el desarrollo de proyectos en Java Enterprise Edition.**
- **Aprender a utilizar una herramienta ORM como Hibernate.**
- **Familiarizarse con las herramientas que ofrece Maven para facilitar la gestión y compilación de proyectos en Java.**

Aprender todas estas tecnologías ha resultado de gran ayuda, no sólo para la realización del presente proyecto sino también para cualquier proyecto futuro, ya que se trata de tecnologías muy extendidas en el mundo del desarrollo en Java. Los conocimientos adquiridos durante la carrera en asignaturas como Programación Avanzada, Sociedad de la Información o Comunicaciones de Empresa, entre muchas otras, correspondientes sobretodo a la rama de Telemática, han facilitado enormemente la comprensión de estas herramientas y el diseño de la base de datos y la arquitectura de la plataforma.

5.3. Líneas futuras

Actualmente, ibermedia está trabajando en los clientes para dos juegos que harán uso de esta plataforma: KoKo Resort, un juego arcade basado en el conecta cuatro, y Pool Madness, un juego de billar en 3D. Se espera que la versión beta de KoKo Resort sea publicada en Facebook en las próximas semanas, mientras que el desarrollo de Pool Madness está pendiente de solucionar una serie de problemas derivados del estado, aún muy incipiente, de las tecnologías disponibles para desarrollar juegos en 3D sobre Flash.

Aparte del desarrollo de nuevos juegos, el futuro inmediato de la plataforma pasa por el desarrollo de una aplicación SaaS en la nube que controle la creación, escalado y destrucción de instancias del servidor de juegos que se ha desarrollado en este proyecto. Esta aplicación sería utilizada por terceros desarrolladores para desplegar sus juegos de manera sencilla, centrándose sólo en la mecánica del juego y sin preocuparse de los detalles de funcionamiento del Lobby. De manera complementaria, esta aplicación deberá contar con un front-end atractivo y completo con todas las herramientas que el desarrollador necesite para crear nuevas zonas, subir sus extensiones de juego al servidor y modificar el contenido de la base de datos, entre otras funciones.

APÉNDICES

Apéndice A

Glosario de términos

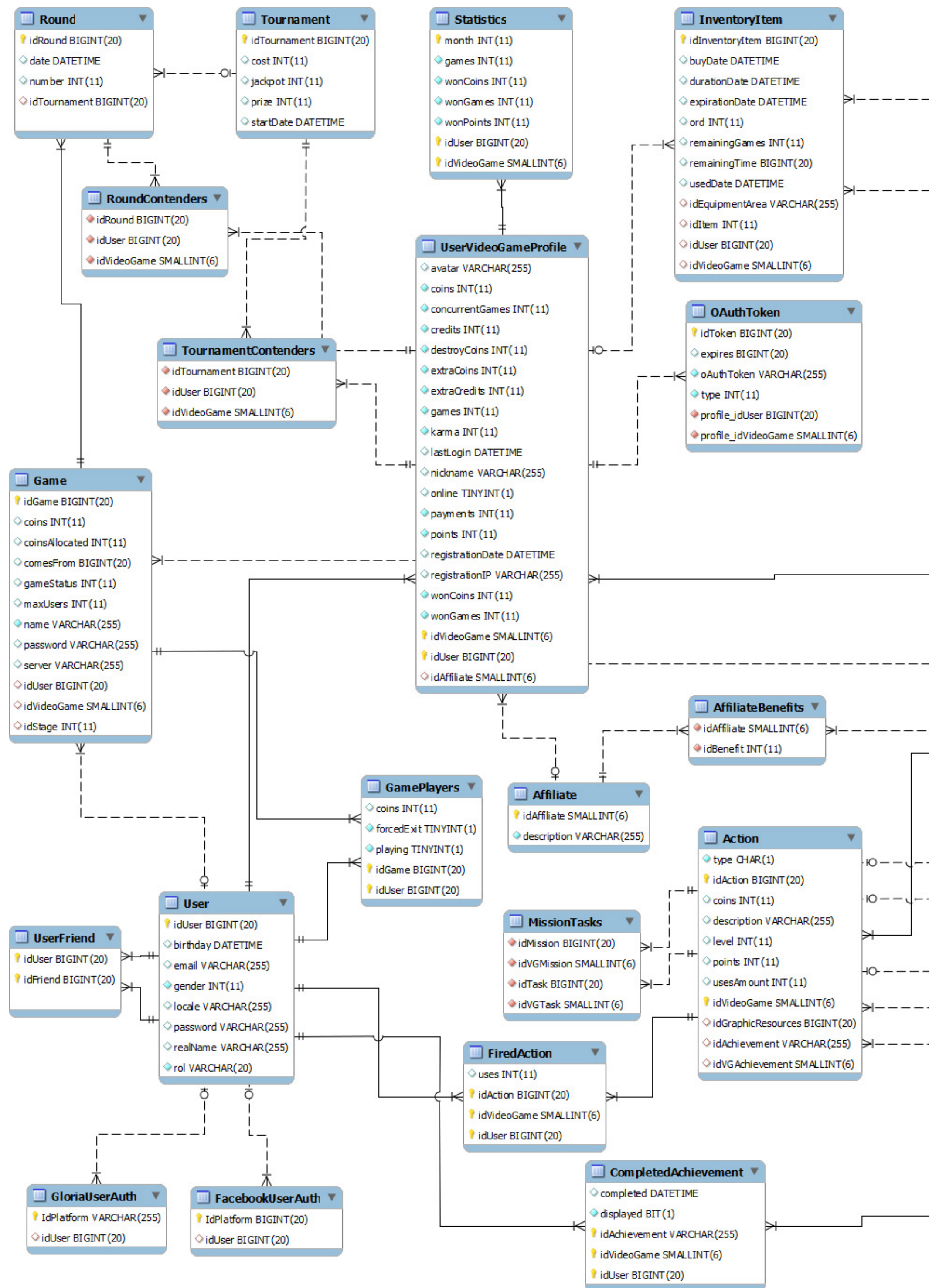
- **Bot:** Usuario no humano que implementa inteligencia artificial para simular el comportamiento de un jugador corriente de la plataforma.
- **BotManager:** Módulo que activa y desactiva a los bots.
- **Cliente:** Aplicación, o en este caso videojuego, que accede a los servicios ofrecidos por la plataforma a través de peticiones enviadas a instancias del usuario o jugador.
- **DAO:** Clase encargada de realizar operaciones de persistencia sencillas relacionadas con una entidad concreta.
- **Entidad:** Clase u objeto persistente gestionado por el ORM, cuyas propiedades se almacenarán o recuperarán desde la base de datos.
- **Extensión:** Clase ejecutable de las aplicaciones que corren sobre la plataforma, ya sea en una zona o en una sala.
- **Game (clase):** Clase ejecutada desde la extensión de juego que contiene toda la lógica del servidor del mismo.
- **Handler:** Componente que utiliza una extensión para gestionar las peticiones de los clientes o los eventos producidos en el servidor.
- **Jugador:** Usuario que participa activamente en una partida dentro del servidor. En una sala de juego pueden existir usuarios que no sean jugadores y se limiten sólo a observar el desarrollo de la misma.
- **Lobby:** Sala especial que no ejecuta ninguna aplicación por si misma y que contiene a todos los usuarios que no están jugando ninguna partida.

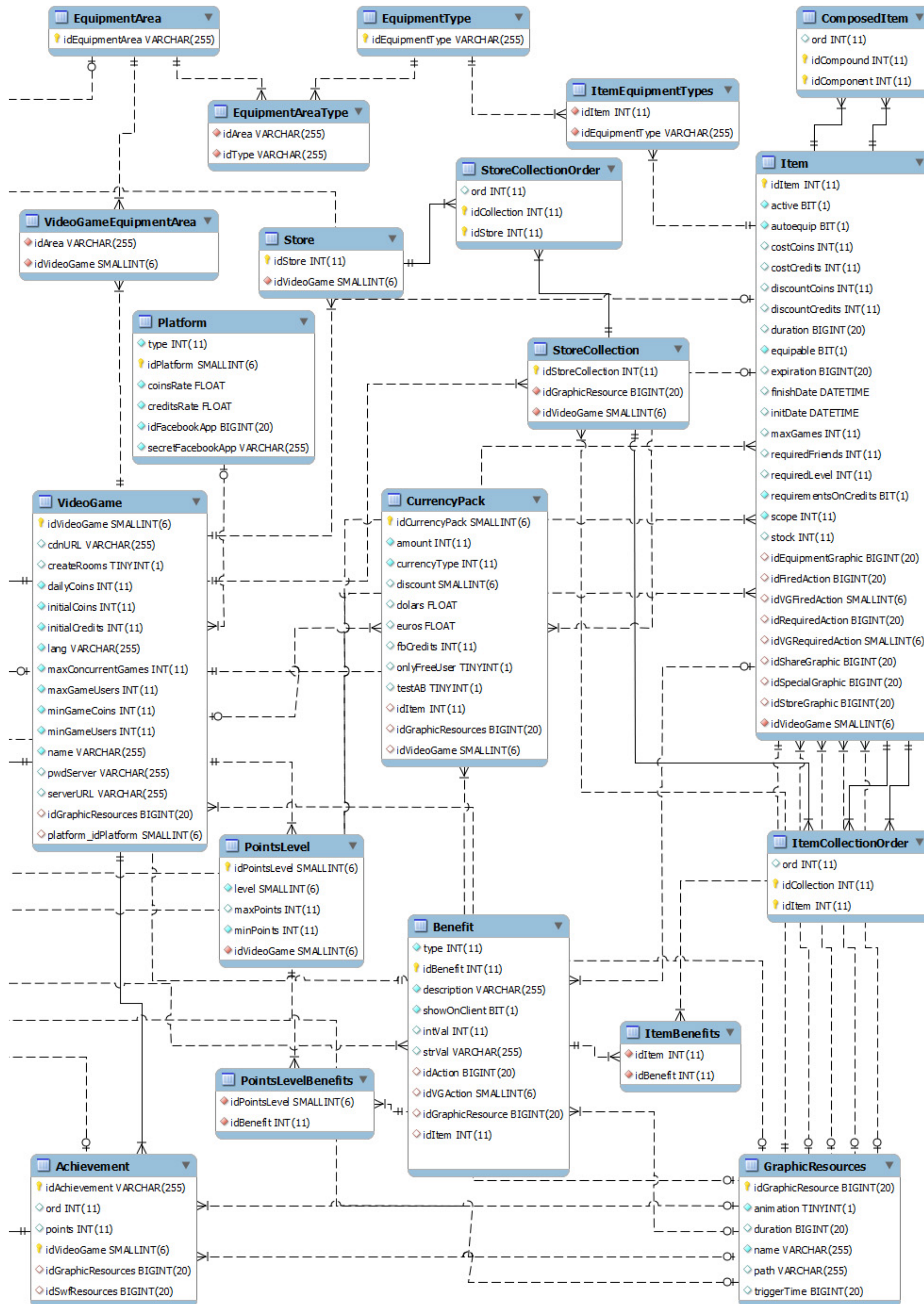
- **Manager:** Clase encargada de compaginar las funciones de varios DAOs y otros managers para realizar operaciones de persistencia complejas que involucran varias acciones sencillas. Por lo general, estas operaciones se realizan dentro de una transacción.
- **ManagerDelegate:** Clase de tipo desde la que se puede invocar cualquiera de los managers a través de un método get.
- **ORM:** Object-Relational Mapping. Es una técnica que permite mapear las entidades de una aplicación Java sobre las tablas de una base de datos relacional. El término ORM también puede usarse para referirnos al framework que implementa esta técnica (en este caso, Hibernate).
- **Player (clase):** Clase que sintetiza a los jugadores del juego dentro de la clase Game. La extensión asociará un player a cada uno de los usuarios en la sala de juego.
- **POJO:** Clase sencilla que consiste en una serie de propiedades accesibles mediante métodos get y set.
- **Sala:** Entes lógicos en los que se encuentra dividida una zona. Las salas contienen a los usuarios logueados en la plataforma y pueden ejecutar una segunda aplicación en paralelo a la que ejecuta la zona.
- **SFS2X:** Siglas de SmartFoxServer 2X.
- **Usuario:** Cada uno de los agentes que están logueados en la plataforma, ya sea un ser humano humana o un bot. No todos los usuarios son jugadores.
- **Zona:** Entes lógicos en los que se encuentra dividida una instancia del servidor de SmartFox. Cada una de las zonas corresponde con una aplicación.

Apéndice B

Diagrama del modelo de datos

A continuación se ofrece un diagrama simplificado del modelo de datos. Debido a la cantidad de tablas y las complejas relaciones entre las mismas, hemos optado por eliminar las menos importantes para el funcionamiento básico de la plataforma, como las referentes a los logs y las traducciones, de manera que el grafo resulte un poco más comprensible.


Round
idRound BIGINT(20)
date DATETIME
number INT(11)
idTournament BIGINT(20)
Tournament
idTournament BIGINT(20)
cost INT(11)
jackpot INT(11)
prize INT(11)
startDate DATETIME
Statistics
month INT(11)
games INT(11)
wonCoins INT(11)
wonGames INT(11)
wonPoints INT(11)
idUser BIGINT(20)
idVideoGame SMALLINT(6)
InventoryItem
idInventoryItem BIGINT(20)
buyDate DATETIME
durationDate DATETIME
expirationDate DATETIME
ord INT(11)
remainingGames INT(11)
remainingTime BIGINT(20)
usedDate DATETIME
idEquipmentArea VARCHAR(255)
idItem INT(11)
idUser BIGINT(20)
idVideoGame SMALLINT(6)
RoundContenders
idRound BIGINT(20)
idUser BIGINT(20)
idVideoGame SMALLINT(6)
UserVideoGameProfile
avatar VARCHAR(255)
coins INT(11)
concurrentGames INT(11)
credits INT(11)
destroyCoins INT(11)
extraCoins INT(11)
extraCredits INT(11)
games INT(11)
karma INT(11)
lastLogin DATETIME
nickname VARCHAR(255)
online TINYINT(1)
payments INT(11)
points INT(11)
registrationDate DATETIME
registrationIP VARCHAR(255)
wonCoins INT(11)
wonGames INT(11)
idVideoGame SMALLINT(6)
idUser BIGINT(20)
idAffiliate SMALLINT(6)
OAuthToken
idToken BIGINT(20)
expires BIGINT(20)
oAuthToken VARCHAR(255)
type INT(11)
profile_idUser BIGINT(20)
profile_idVideoGame SMALLINT(6)
TournamentContenders
idTournament BIGINT(20)
idUser BIGINT(20)
idVideoGame SMALLINT(6)
Game
idGame BIGINT(20)
coins INT(11)
coinsAllocated INT(11)
comesFrom BIGINT(20)
gameStatus INT(11)
maxUsers INT(11)
name VARCHAR(255)
password VARCHAR(255)
server VARCHAR(255)
idUser BIGINT(20)
idVideoGame SMALLINT(6)
idStage INT(11)
AffiliateBenefits
idAffiliate SMALLINT(6)
idBenefit INT(11)
GamePlayers
coins INT(11)
forcedExit TINYINT(1)
playing TINYINT(1)
idGame BIGINT(20)
idUser BIGINT(20)
Affiliate
idAffiliate SMALLINT(6)
description VARCHAR(255)
Action
type CHAR(1)
idAction BIGINT(20)
coins INT(11)
description VARCHAR(255)
level INT(11)
points INT(11)
usesAmount INT(11)
idVideoGame SMALLINT(6)
idGraphicResources BIGINT(20)
idAchievement VARCHAR(255)
idVGAchievement SMALLINT(6)
User
idUser BIGINT(20)
birthday DATETIME
email VARCHAR(255)
gender INT(11)
locale VARCHAR(255)
password VARCHAR(255)
realName VARCHAR(255)
rol VARCHAR(20)
UserFriend
idUser BIGINT(20)
idFriend BIGINT(20)
MissionTasks
idMission BIGINT(20)
idVGMission SMALLINT(6)
idTask BIGINT(20)
idVGTask SMALLINT(6)
FiredAction
uses INT(11)
idAction BIGINT(20)
idVideoGame SMALLINT(6)
idUser BIGINT(20)
CompletedAchievement
completed DATETIME
displayed BIT(1)
idAchievement VARCHAR(255)
idVideoGame SMALLINT(6)
idUser BIGINT(20)
GloriaUserAuth
IdPlatform VARCHAR(255)
idUser BIGINT(20)
FacebookUserAuth
IdPlatform BIGINT(20)
idUser BIGINT(20)


EquipmentArea
idEquipmentArea VARCHAR(255)
EquipmentType
idEquipmentType VARCHAR(255)
ComposedItem
ord INT(11)
idCompound INT(11)
idComponent INT(11)
ItemEquipmentTypes
idItem INT(11)
idEquipmentType VARCHAR(255)
EquipmentAreaType
idArea VARCHAR(255)
idType VARCHAR(255)
StoreCollectionOrder
ord INT(11)
idCollection INT(11)
idStore INT(11)
Item
idItem INT(11)
active BIT(1)
autoequip BIT(1)
costCoins INT(11)
costCredits INT(11)
discountCoins INT(11)
discountCredits INT(11)
duration BIGINT(20)
equipable BIT(1)
expiration BIGINT(20)
finishDate DATETIME
initDate DATETIME
maxGames INT(11)
requiredFriends INT(11)
requiredLevel INT(11)
requirementsOnCredits BIT(1)
scope INT(11)
stock INT(11)
idEquipmentGraphic BIGINT(20)
idFiredAction BIGINT(20)
idVGFiredAction SMALLINT(6)
idRequiredAction BIGINT(20)
idVGRequiredAction SMALLINT(6)
idShareGraphic BIGINT(20)
idSpecialGraphic BIGINT(20)
idStoreGraphic BIGINT(20)
idVideoGame SMALLINT(6)
VideoGameEquipmentArea
idArea VARCHAR(255)
idVideoGame SMALLINT(6)
Store
idStore INT(11)
idVideoGame SMALLINT(6)
Platform
type INT(11)
idPlatform SMALLINT(6)
coinsRate FLOAT
creditsRate FLOAT
idFacebookApp BIGINT(20)
secretFacebookApp VARCHAR(255)
StoreCollection
idStoreCollection INT(11)
idGraphicResource BIGINT(20)
idVideoGame SMALLINT(6)
VideoGame
idVideoGame SMALLINT(6)
cdnURL VARCHAR(255)
createRooms TINYINT(1)
dailyCoins INT(11)
initialCoins INT(11)
initialCredits INT(11)
lang VARCHAR(255)
maxConcurrentGames INT(11)
maxGameUsers INT(11)
minGameCoins INT(11)
minGameUsers INT(11)
name VARCHAR(255)
pwdServer VARCHAR(255)
serverURL VARCHAR(255)
idGraphicResources BIGINT(20)
platform_idPlatform SMALLINT(6)
CurrencyPack
idCurrencyPack SMALLINT(6)
amount INT(11)
currencyType INT(11)
discount SMALLINT(6)
dolars FLOAT
euros FLOAT
fbCredits INT(11)
onlyFreeUser TINYINT(1)
testAB TINYINT(1)
idItem INT(11)
idGraphicResources BIGINT(20)
idVideoGame SMALLINT(6)
PointsLevel
idPointsLevel SMALLINT(6)
level SMALLINT(6)
maxPoints INT(11)
minPoints INT(11)
idVideoGame SMALLINT(6)
ItemCollectionOrder
ord INT(11)
idCollection INT(11)
idItem INT(11)
Benefit
type INT(11)
idBenefit INT(11)
description VARCHAR(255)
showOnClient BIT(1)
intVal INT(11)
strVal VARCHAR(255)
idAction BIGINT(20)
idVGAction SMALLINT(6)
idGraphicResource BIGINT(20)
idItem INT(11)
ItemBenefits
idItem INT(11)
idBenefit INT(11)
PointsLevelBenefits
idPointsLevel SMALLINT(6)
idBenefit INT(11)
Achievement
idAchievement VARCHAR(255)
ord INT(11)
points INT(11)
idVideoGame SMALLINT(6)
idGraphicResources BIGINT(20)
idSwfResources BIGINT(20)
GraphicResources
idGraphicResource BIGINT(20)
animation TINYINT(1)
duration BIGINT(20)
name VARCHAR(255)
path VARCHAR(255)
triggerTime BIGINT(20)

Apéndice C

Descripción de los casos de uso

En las siguientes páginas se incluirá la descripción de los diferentes casos de uso de la plataforma. Al tratarse de servicios remotos, todos los casos de uso requeriran de un intercambio de mensajes entre el cliente y el servidor. Esta comunicación se realiza a través de la conexión TCP que se realiza entre ambos extremos de la API de SmartFox, que se encargará de las operaciones de empaquetado/desempaquetado, cifrado y compresión en caso de ser necesario.

Todos los mennsajes que se transmiten entre un cliente y un servidor de SmartFox tienen un tipo, definido por una cadena de caracteres de dos partes separada por un punto ("lobby.newuser", por ejemplo), donde la primera identifica uno de los grupos de servicio y el segundo a un servicio concreto dentro de el mismo tipo. Los datos adjuntos que transporta el mensaje viajan empaquetados en una estructura SFSObject o SFSArray.

Caso de uso:	Loguear usuario
Actores: Jugador/Cliente	
Datos de Entrada: Nombre de usuario, contraseña, nombre de Zona(Juego). Token OAuth si es necesario.	
Precondiciones: 1. Usuario está ejecutando la aplicación cliente. 2. La aplicación cliente tiene una conexión TCP activa con el servidor.	
Secuencia:	
1. El cliente envía una petición de login a la zona correspondiente al juego.	2. El servidor comprueba que el usuario no se encuentra ya logueado. 3. El servidor comprueba el tipo de autenticación que usa la petición y ejecuta el método correspondiente. 4. El servidor comprueba la veracidad de las credenciales que acompañan a la petición, recupera los datos del usuario y su perfil de la base de datos y los actualiza si fuera necesario. Si no existen el usuario o el perfil de juego, el servidor los creará en este punto. 5. El servidor configura la sesión del usuario, le inscribe en la Zona del juego y confirma al cliente de que ha completado el login con éxito. 6. El servidor comprueba el *nick name* del perfil del jugador. 7. El servidor comprueba si el jugador ha recibido la recompensa diaria por entrar al juego. 8. El servidor inscribe al jugador en la sala principal, "The Lobby".
Secuencias Alternativas: 1. En 3, el tipo de autenticación indicado es desconocido. Se interrumpe el proceso y se informa al cliente con el código de error BAD_REQUEST. 2. En 4, las credenciales son erróneas. Se interrumpe el proceso y se informa al cliente con el código de error BAD_REQUEST 3. En 6, el *nick* está en blanco. Se envía al cliente un mensaje con una sugerencia de *nick*. 4. En 7, el jugador no ha recibido las recompensas. Se le da al jugador la recompensa que le corresponde, se informa al cliente y se sigue con el proceso.	
Postcondiciones:	

Caso de uso:	Desloguear/Desconectar usuario
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El jugador está logueado.	
Secuencia:	
1. El cliente envía una petición de logout o directamente interrumpe su conexión.	2. El servidor cambia el estado del jugador a fuera de línea.
	3. El servidor busca las partidas abiertas que tenía el jugador. 4. El servidor saca al jugador de todas las partidas encontradas. 5. El servidor actualiza las variables de sesión del jugador. 6. El servidor cierra la sesión del jugador y cierra su conexión si aún persistía.
Secuencias Alternativas: 1. En 4, la partida queda vacía o con un solo jugador, se cierra nombrando ganador al jugador que quedó. 2. En 4, quedan jugadores suficientes para seguir la partida, pero el jugador que se desconecta era el propietario de la partida. Se nombra un nuevo propietario de entre los jugadores restantes. y la partida continúa	
Postcondiciones: 1. Se notifica a los amigos del jugador que se encuentrén online de la desconexión.	

<table>
<tr><td>Caso de uso:</td><td>Guardar nick de nuevo usuario</td></tr>
<tr><td colspan="2">Actores:
Jugador/Cliente</td></tr>
<tr><td colspan="2">Datos de Entrada:
Nick name elegido.</td></tr>
<tr><td colspan="2">Precondiciones:
1. El usuario está ejecutando la aplicación cliente.
2. El usuario está dado de alta en la plataforma y dispone ya de un perfil de juego.
3. La aplicación cliente tiene una conexión TCP activa con el servidor.</td></tr>
<tr><td colspan="2">Secuencia:</td></tr>
<tr><td>1. El cliente envía una petición con el nick name seleccionado.</td><td>2. El servidor recupera el perfil de juego del usuario.
3. El servidor comprueba que el nick elegido se ajusta a las restricciones de tamaño máximo y mínimo y que no contiene palabras prohibidas o malsonantes.
4. El servidor asocia el nuevo nick al perfil del usuario en la base de datos.
5. El servidor informa al cliente de que la operación se ha completado con éxito.</td></tr>
<tr><td colspan="2">Secuencias Alternativas:
1. Estando en 3, el nick no cumple las normas, se envía un mensaje de error con el código BAD_WORD al cliente.
2. Estando en 4, el nick ya se encuentra en uso por otro jugador, se envía al cliente un mensaje de error EXISTING_NICKNAME con una sugerencia de nick name.
3. Estando en 4, se produce un error de persistencia al intentar guardar datos en la base, se envía al cliente un mensaje de error PERSISTENCE_ERROR.</td></tr>
<tr><td colspan="2">Postcondiciones:</td></tr>
</table>

Caso de uso:	Solicitar packs de monedas
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El usuario está logueado. 2. El usuario se encuentra en el Lobby.	
Secuencia:	
1. El cliente envía la petición.	2. El servidor recupera toda la información sobre packs de monedas y créditos del juego. 3. El servidor empaqueta la información recuperada y la envía al cliente.
Secuencias Alternativas:	
Postcondiciones:	

Caso de uso:	Ping
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El usuario está logueado.	
Secuencia:	
1. El cliente envía el ping request.	2. El servidor contesta al ping, incluyendo el número de jugadores conectados y el número de partidas en juego en el momento de recibir la petición.
Secuencias Alternativas:	
Postcondiciones:	

Caso de uso:	Solicitar tiempo de servidor.
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El usuario está logueado.	
Secuencia: 1. El cliente envía la petición.	2. El servidor envía un mensaje al cliente con el tiempo del servidor en el momento de recibir la petición.
Secuencias Alternativas:	
Postcondiciones:	

Caso de uso:	Solicitar logros completados.
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El usuario está logueado.	
Secuencia: 1. El cliente envía la petición.	2. El servidor empaqueta los logros completados y los envía al cliente.
Secuencias Alternativas:	
Postcondiciones:	

Caso de uso:	Acción realizada.
Actores: Jugador/Cliente	
Datos de Entrada: Descripción de la acción, número de veces realizada.	
Precondiciones: 1. El usuario está logueado.	
Secuencia: 1. El cliente envía la petición.	2. El servidor guarda o incrementa en la base de datos el número de veces que se ha realizado la acción.
Secuencias Alternativas:	
Postcondiciones:	

<table>
<tr><td>Caso de uso:</td><td>Mostrar clasificación.</td></tr>
<tr><td colspan="2">Actores:
Jugador/Cliente</td></tr>
<tr><td colspan="2">Datos de Entrada:
Tipo de clasificación (puede referirse a la clasificación absoluta, la de la semana actual o de la semana anterior.).</td></tr>
<tr><td colspan="2">Precondiciones:
1. El usuario está logueado.</td></tr>
<tr><td colspan="2">Secuencia:</td></tr>
<tr><td>1. El cliente envía la petición.</td><td>2. El servidor comprueba el tipo de clasificación que se solicita.
3. El servidor recupera la clasificación de la base de datos, la empaqueta y la envía al cliente.</td></tr>
<tr><td colspan="2">Secuencias Alternativas:</td></tr>
<tr><td colspan="2">Postcondiciones:</td></tr>
</table>

<table>
<tr><td>Caso de uso:</td><td>Mostrar clasificación de una semana determinada.</td></tr>
<tr><td colspan="2">Actores:
Jugador/Cliente</td></tr>
<tr><td colspan="2">Datos de Entrada:
Semana(número del 1 al 52) y año.</td></tr>
<tr><td colspan="2">Precondiciones:
1. El usuario está logueado.</td></tr>
<tr><td colspan="2">Secuencia:</td></tr>
<tr><td>1. El cliente envía la petición.</td><td>2. El servidor recupera la clasificación correspondiente a la semana indicada.
3. El servidor empaqueta la clasificación y la envía al cliente.</td></tr>
<tr><td colspan="2">Secuencias Alternativas:</td></tr>
<tr><td colspan="2">Postcondiciones:</td></tr>
</table>

Caso de uso:	Actualizar y consultar lista de amigos.
Actores: Jugador/Cliente	
Datos de Entrada: Lista con los id de los amigos del usuario en la red social.	
Precondiciones: 1. El usuario está logueado.	
Secuencia:	
1. El cliente envía la petición.	2. El servidor comprueba cuáles de los id de la lista están registrados como jugadores en la plataforma. 3. El servidor añade los usuarios encontrados a la lista de amigos del jugador. 4. El servidor añade al jugador encontrados a la lista de amigos de los usuarios encontrados. 5. El servidor actualiza las variables de sesión del jugador. 6. El servidor empaqueta la lista de amigos del jugador y la envía al cliente.
Secuencias Alternativas:	
Postcondiciones: 1. El servidor actualiza las variables de sesión de los usuarios que se encuentren conectados en ese momento. 2. El servidor informa a los usuarios encontrados que se encuentren conectados en ese momento de que tienen un nuevo amigo.	

<table>
<tr><td>Caso de uso:</td><td>Crear nueva sala de juego.</td></tr>
<tr><td colspan="2">Actores:
Jugador/Cliente</td></tr>
<tr><td colspan="2">Datos de Entrada:
Nombre de la sala, contraseña (si procede), cuota de entrada en monedas, número de jugadores, lista de nick names de los jugadores invitados (si procede). Booleanos: partida privada, permitir observadores.</td></tr>
<tr><td colspan="2">Precondiciones:
1. El jugador está logueado.
2. El jugador se encuentra en el Lobby.</td></tr>
<tr><td colspan="2">Secuencia:</td></tr>
<tr><td>1. El cliente envía la petición.</td><td>2. El servidor comprueba el nombre de la sala: tamaños máximo y mínimo, palabras malsonantes, nombre en uso.
3. El servidor comprueba que el juego admite partidas con el número de jugadores indicado.
4. El servidor comprueba que el jugador no se encuentra ya en otra sala de juego.
5. Se crea la partida en la base de datos y la sala en la Zona del juego.
6. Se inscribe al jugador en la partida (reservando las monedas) y en la sala (como propietario de la misma).</td></tr>
<tr><td colspan="2">Secuencias Alternativas:
1. En 2, si el nombre incumple alguna de las normas se envía la correspondiente notificación al cliente, incluyendo una sugerencia de nombre válido.
2. En 3, si el número de jugadores excede el máximo o no llega al mínimo permitido, se notifica al cliente.
3. En 4, si el usuario ya se encuentra en una sala de juego se detiene el proceso y se notifica al cliente.</td></tr>
<tr><td colspan="2">Postcondiciones:
1. Se elimina al jugador de los grupos de multicast de mensajes ajenos a la sala de juego.
2. Se envía invitaciones a los usuarios indicados en la lista de invitados.</td></tr>
</table>

Caso de uso:	Unirse a sala de juego.
Actores: Jugador/Cliente	
Datos de Entrada: Nombre de la sala, contraseña (si procede).	
Precondiciones: 1. El jugador está logueado. 2. El jugador se encuentra en el Lobby.	
Secuencia:	
1. El cliente envía la petición.	2. El servidor busca la sala con el nombre indicado. 3. El servidor comprueba si la partida es privada y en caso positivo comprueba que el jugador cumple los requisitos para unirse (si está en la lista de invitados o si la contraseña es correcta). 4. El servidor comprueba que la partida no ha empezado ni está ya finalizada. 5. El servidor comprueba que la partida no esté llena. 6. El servidor comprueba que el jugador no se encuentra ya en otra sala de juego. 7. Se inscribe al jugador en la partida (reservando las monedas) y en la sala de juego.
Secuencias Alternativas: 1. En 2, si la sala no existe, se notifica al cliente. 2. En 3, si no está en la lista de invitados o la contraseña es incorrecta, se notifica al cliente. 3. En 4, si la partida está en marcha o finalizada, se notifica al cliente. 4. En 5, si la sala está llena, se notifica al cliente. 3. En 6, si el jugador está en otra sala, se notifica al cliente. 3. En 7, si el jugador no tiene monedas suficientes para entrar a la partida, se notifica al cliente.	
Postcondiciones: 1. Se elimina al jugador de los grupos de multicast de mensajes ajenos a la sala de juego.	

<table>
<tr><td>Caso de uso:</td><td>Responder a invitación.</td></tr>
<tr><td colspan="2">Actores:
Jugador/Cliente</td></tr>
<tr><td colspan="2">Datos de Entrada:
Nombre del propietario la sala, respuesta (true/false) en booleano.</td></tr>
<tr><td colspan="2">Precondiciones:
1. El jugador está logueado.
2. El jugador se encuentra en el Lobby.
3. El jugador ha recibido una invitación para unirse a una partida.</td></tr>
<tr><td colspan="2">Secuencia:</td></tr>
<tr><td>1. El cliente envía la petición.</td><td>2. El servidor busca la sala propiedad del jugador indicado.
3. El servidor comprueba si el jugador está en la lista de invitados.
4. Si el jugador está invitado, se le inscribe en la partida (reservando las monedas) y en la sala de juego.</td></tr>
<tr><td colspan="2">Secuencias Alternativas:
1. En 2, si la sala no existe, se notifica al cliente.
2. En 3, si no está en la lista de invitados o la contraseña es incorrecta, se notifica al cliente.
3. En 4, si la partida está en marcha o finalizada o el jugador no dispone de monedas suficientes, se notifica al cliente.</td></tr>
<tr><td colspan="2">Postcondiciones:
1. Se elimina al jugador de los grupos de multicast de mensajes ajenos a la sala de juego.
2. Se notifica al creador de la partida si la invitación fue aceptada o rechazada por parte del jugador.</td></tr>
</table>

Caso de uso:	Canjear token de partida.
Actores: Jugador/Cliente	
Datos de Entrada: Token de partida (cadena de caracteres).	
Precondiciones: 1. El jugador está logueado. 2. El jugador se encuentra en el Lobby. 3. El jugador ha recibido un token de invitación a una partida.	
Secuencia:	
1. El cliente envía la petición.	2. El servidor busca la partida correspondiente al token. 3. El servidor inscribe al jugador en la partida (reservando las monedas) y en la sala de juego.
Secuencias Alternativas: 1. En 2, si la partida no existe, se crea y se sigue en 3. 2. En 3, si el jugador no dispone de monedas suficientes, se notifica al cliente.	
Postcondiciones: 1. Se elimina al jugador de los grupos de multicast de mensajes ajenos a la sala de juego.	

Caso de uso:	Volver al lobby.
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El jugador está logueado.	
Secuencia:	
1. El cliente envía la petición.	2. El servidor busca partidas donde se encuentre el jugador. 3. El servidor saca al jugador de las partidas encontradas. 4. El servidor manda al jugador al lobby.
Secuencias Alternativas:	
Postcondiciones:	

Caso de uso:	Partida rápida.
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El jugador está logueado. 2. El jugador se encuentra en el lobby.	
Secuencia: 1. El cliente envía la petición.	2. El servidor busca partidas abiertas. 3. El servidor inscribe al jugador en una de las partidas encontradas.
Secuencias Alternativas: 1. En 2, si no se encuentra ninguna partida abierta, se crea una nueva.	
Postcondiciones:	

Caso de uso:	Partida de revancha.
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El jugador está logueado. 2. El jugador acaba de terminar una partida.	
Secuencia: 1. El cliente envía la petición.	2. El servidor busca la sala de la partida actual. 3. El servidor busca si ya se ha creado una sala para jugar la partida de revancha. 4. El servidor saca al jugador de la partida terminada y lo mete en la partida de revancha.
Secuencias Alternativas: 1. En 3, no se encuentra una partida de revancha, se crea una nueva. 2. En 4, si el jugador no tiene monedas suficientes, se notifica al cliente.	
Postcondiciones:	

Caso de uso:	Consultar acción realizada.
Actores: Jugador/Cliente	
Datos de Entrada: Id de la acción, número de veces realizada.	
Precondiciones: 1. El jugador está logueado.	
Secuencia: 1. El cliente envía la petición.	2. El servidor recupera la acción del registro de acciones del jugador en la base de datos. 3. El servidor incrementa el número de veces que se ha realizado la acción con el número indicado en la petición. 4. El servidor informa al cliente del número de veces que es necesario volver a realizar la acción para llegar al umbral que la dispara.
Secuencias Alternativas:	
Postcondiciones:	

Caso de uso:	Consultar acciones no realizadas.
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El jugador está logueado.	
Secuencia: 1. El cliente envía la petición.	2. El servidor recupera la lista de acciones que le faltan por realizar al jugador para completar las misiones sociales. 3. El servidor empaqueta las acciones recuperadas junto con las veces que es necesario realizar cada una y las envía al cliente.
Secuencias Alternativas:	
Postcondiciones:	

Caso de uso:	Consultar objetos de la tienda.
Actores: Jugador/Cliente	
Datos de Entrada: Id de la tienda.	
Precondiciones: 1. El jugador está logueado.	
Secuencia: 1. El cliente envía la petición.	2. El servidor recupera todas las colecciones de objetos de la tienda. 3. El servidor empaqueta los objetos de cada colección en un SFSArray. 4. El servidor empaqueta los arrays de objetos correspondientes a cada colección en un SFSArray y lo envía todo al cliente.
Secuencias Alternativas:	
Postcondiciones:	

Caso de uso:	Recuperar inventario del jugador.
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El jugador está logueado.	
Secuencia: 1. El cliente envía la petición.	2. El servidor busca todos los objetos que el jugador tiene en su inventario. 3. El servidor empaqueta la lista de objetos recuperados y la envía al cliente.
Secuencias Alternativas:	
Postcondiciones:	

Caso de uso:	Comprar objeto.
Actores: Jugador/Cliente	
Datos de Entrada: Id del objeto, tipo de moneda empleada (coins o credits).	
Precondiciones: 1. El jugador está logueado.	
Secuencia:	
1. El cliente envía la petición.	2. El servidor busca el objeto que corresponde al id. 3. El servidor resta al jugador al cantidad de monedas o créditos correspondiente. 4. El servidor añade el objeto al inventario del jugador. 5. El servidor empaqueta el objeto y lo envía al cliente.
Secuencias Alternativas: 1. En 2, el stock del objeto está agotado o el usuario no cumple los requisitos (nivel, número de amigos, etc.) para obtenerlo, se notifica al cliente. 2. En 3, el jugador no dispone de monedas suficientes, se notifica al cliente. 3. En 4, el objeto es autoequipable, se equipa el objeto al jugador.	
Postcondiciones:	

Caso de uso:	Equipar objeto.
Actores: Jugador/Cliente	
Datos de Entrada: Id del objeto, id del área de equipo.	
Precondiciones: 1. El jugador está logueado.	
Secuencia:	
1. El cliente envía la petición.	2. El servidor busca el objeto en el inventario del jugador. 3. El servidor equipa el objeto. 4. El servidor envía el inventario del jugador al cliente con los cambios realizados.
Secuencias Alternativas: 1. En 2, el jugador no tiene el objeto en su inventario, se notifica al cliente. 2. En 3, el objeto ya está equipado, se notifica al cliente. 3. En 3, el objeto no puede equiparse en el área indicada, se notifica al cliente.	
Postcondiciones:	

Caso de uso:	Desequipar.
Actores: Jugador/Cliente	
Datos de Entrada: Id del objeto.	
Precondiciones: 1. El jugador está logueado.	
Secuencia:	
1. El cliente envía la petición.	2. El servidor desequipa el objeto al jugador. 3. El servidor envía el inventario del jugador al cliente con los cambios realizados.
Secuencias Alternativas: 1. En 2, el jugador no tiene el objeto equipado, se notifica al cliente.	
Postcondiciones:	

Caso de uso:	Recuperar objetos equipados del jugador.
Actores: Jugador/Cliente	
Datos de Entrada:	
Precondiciones: 1. El jugador está logueado.	
Secuencia:	
1. El cliente envía la petición.	2. El servidor busca todos los objetos que el jugador tiene en su inventario. 3. El servidor filtra la lista de objetos recuperados para quedarse sólo con los equipados. 3. El servidor empaqueta la lista de objetos equipados y la envía al cliente.
Secuencias Alternativas:	
Postcondiciones:	

Bibliografía

[1] A. R. Seddighi, *Spring Persistence with Hibernate*, Packt Publishing (2009).

[2] VV.AA., *Spring 3.1 Reference Documentation*, (2012).

[3] VV.AA., *Hibernate 4.0 Reference Documentation*, (2012).

[4] *Apache Maven Tutorial*, TutorialsPoint.com (2012).

[5] F. J. Moldes Teo, *Java 7*, Anaya Multimedia (2011).

[6] *SmartFoxServer 2X Documentation Central*, gotoAndPlay() (2009). http://docs2x.smartfoxserver.com

[7] D. Hardt, *The OAuth 2.0 Authorization Framework*, RFC:6749 (Octubre 2012).

Informes

[8] International Game Developers Association, *2008-2009 Casual Games White Paper*, (2008).

[9] Casual Games Association, *Casual Games Sector Report 2012: Social Network games*, (2012).

[10] SuperData Research, *Beyond Facebook: A Look At Local Social Networks in Europe and Latin America*, (Octubre 2011).

Artículos

[11] Lightspeed Research, *It's Game On for Social Networkers*, (2010). http://www.lightspeedresearch.com/press-releases/it %E2 %80 %99s-game-on-for-social-networkers

[12] Kevin Boyland, *8 Hot Industries for Tsart-Ups*, Ibis World (Febrero 2013). http://www.ibisworld.com/media/2013/02/06/8-hot-industries-for-start-ups/

[13] Sean Ryan, *Game Developers Conference SF: Video Recap*, Facebook Developers (Abril 2013).
http://developers.facebook.com/blog/post/2013/04/29/game-developers-conference-sf–video-recap/